1984

Relational-like file structure

Robert A. Fabbio

ROCHESTER INSTITUTE OF TECHNOLOGY
SCHOOL OF COMPUTER SCIENCE ANO TECHNOLOGY

Relational-like File Structure

by

Robert A. Fabbio
August 21, 1984

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

Approved by:

Professor Jeffery Lasky
Chairman

Professor Warren Carithers

Professor Andrew Kitchen

rised sample statement for granting or denying permission to produce an RIT thesis.

tle of Thesis _____

_____

_____ hereby (grant, ny) permission to the Wallace Memorial Library, of R.I.T., to produce my thesis in whole or in part. Any reproduction will ; be for commercial use or profit.

Or

ROBERT FABBIO prefer to be ntacted each time a request for reproduction is made. I can be ached at the following address. 15 MARY ANNE DR FRANKLIN, MA 02038

:e 4-22-85

1.5 TABLE OF CONTENTS

ABSTRACT

The currently operational relational-like structure, as well as a primitive database management system is described. The proposed file structure integrates a B-tree variant, inverted files, and other structures to provide the underlying facility. The database management system supports multi-user, multi-database retrieval through relational views of both data and documents, as well as the interface to non-procedural languages.

Emphasis regarding design decisions and tradeoffs were related to: 1) the Unix* operating system; 2) the access methods supported; 3) future development, such as document processing (information storage and retrieval), concurrency control and recovery.

KEY WORDS AND PHRASES

relational, file structure, database, integrating, B-tree, inverted files, non-procedural, UNIX, concurrency, information storage and retrieval, documents

ACM COMPUTING REVIEWS

H.2.2, H.3.2, H.3.3

*UNIX - A registered trademark of AT&T Bell Laboratory.

I. INTRODUCTION AND BACKGROUND

The purpose of this thesis as stated in the proposal (Appendix VI.1) was to investigate an alternate approach (different from INGRES, MISTRESS, YARD) to implementing a relational database file structure under UNIX.

The design of database internals and its file structure is dependent upon many factors. The three packages mentioned above all utilize radically different approaches. YARD, a relational database, from Computer Consoles, Inc., does not support indexed retrieval and thus is almost useless in most commercial situations. A YARD relation consists of a single ASCII file containing a very rudimentary relation definition, followed by the data for each tuple. Each field (column) in a tuple is distinguished by a special delimiter. MISTRESS from Rhodnius supports both keyed and sequential retrieval. The underlying index structure used is a static B-tree with pointers at the lowest level to a datafile. This implementation is fairly common and straight forward. The B-tree structure utilized by MISTRESS operates with fixed length pages and keys, which implies that variable length strings are not available. Though the B-tree structure used is very powerful, it does not take advantage of many of the features provided by other B-tree derivatives such as Prefix or B* trees. INGRES from Relational Technology, Inc., is one of the most sophisticated data base management systems running under UNIX. INGRES supports many index structures which range from hashing to static B-trees as well as sequential processing. It is the decision of the user at query time to define the type of index structure desired for each relation. Though this is a very dynamic feature, the user suffers in response time when building new index structures at query time. This approach seems to assume that no one index structure can accurately define a user's needs all the time.

Clearly, there are many approaches to implementing a system that supports relational tables. The approach taken here is much different than the above-mentioned systems. Unlike INGRES, a single index structure made up of a combination of files will be used to model the user's world.

Unlike YARD, keyed retrieval on any arbitrary data item will be supported, as well as sequential processing. Though INGRES and MISTRESS utilize B-trees as a form of indices, the proposed B-tree structure here provides more flexibility and versatility. Lastly, different from all three systems, the designed file structure will integrate the management of data and text, through the use of an inverted file.

II. FUNCTIONAL SPECIFICATION

Within this section of the thesis it is the intent to describe in some detail the functionality of the system and set forth the overall requirements. It is not meant, at this stage, to supply more detail than is necessary in describing each component, but rather to list in point form, the requirements.

II.1 Utilities

Dataload -

1) provide a mechanism to parse and interpret the user defined relation. (schema).

2) method to read and lightly audit the input data.

3) produce from the user-defined schema and the input data, a properly formatted data file.

Keyword Extraction -

1) use as input, the properly created data file and automatically extract keys (or index documents).

2) remove all the duplicate keys extracted from the data.

3) sort the keys in ascending order in preparation to create the index file.

IDXLOAD -

1) supply a method whereby a sorted list of keys is used as input and creates an inverted file.

2) the inverted file should contain a single entry for each unique key, along with a list of pointers back to the data records that had that key.

DIRLOAD -

1) given a properly created inverted file, read it as input and generate an index structure.

2) as decided, the index structure is a variant of a B-tree which handles variable length keys and levels.

II.2 DBMS Primitive Functions

FUNCTION: (struct user *) D_LOGON

DESCRIPTION:

Establish user-dbms logon communications. Builds the user node.

TO CALL:

```
= D_LOGON(uid,tty):

char *uid; /* input - user id */
char *tty; /* input - tty id */
struct user *node;  /* output - prt to user node */
```

RETURNS:

pointer to user node

FUNCTION: (struct usrhskp *) D_OPEN

DESCRIPTION:

Opens database files and creates user links to corresponding dbd areas. Creates dbd templates.

TO CALL:

```
= D_OPEN(dbname,pw,mode,logon)

char *dbname;                  /* input - database name */
char *pw;                      /* input - database password */
char mode;                     /* input - open mode ('r','w','e'*/
struct user *logon;            /* input - ptr to user node */
struct usrhskp *dbdtab;        /* input - pointer to dbd table */
```

RETURNS:

pointer to array element for that particular dbd

error: return NULL

DESIGN NOTES:

build the dbd table in sorted field name order

FUNCTION: D_CLOSE

DESCRIPTION:

Subtracts 1 from the number of users for a given database. If number of users becomes equal to zero 1.) deallocate the dbd template 2.) null out ptr. to template 3.) close corresponding files. NULL out user ptr to dbd table.

TO CALL:

```
= D_CLOSE(logon,dbdtab);

struct user *logon;          /* input - ptr to user node */
struct usrhskp *dbdtab;      /* input - pointer to dbd table */
```

RETURNS:

return (-1) error, (0) successful

FUNCTION: D_LOGOFF

DESCRIPTION:

Take pointer for user node and perform closes to any databases that may still open (D_CLOSE can be called). Then deallocate the user node.

TO CALL:

```
= D_LOGOFF(logon);

struct user *logon;          /* input - ptr to user node */
```

RETURNS:

return (-1) error, (0) successful

FUNCTION: D_GETKEY

DESCRIPTION:

Given logon information and database information, pass back (return) a datafile offset where the data resides.

Assumes a call to D_FNDKEY has been performed.

TO CALL:

```
= D_GETKEY(logon,dbdtab,daoffst)

struct user *logon;                    /* input - ptr to user
                                          node */
struct usrhskp *dbdtab;                /* input - ptr to
                                          database */
long *daoffst;                         /* output - datafile
                                          offset */
```

RETURNS:

error: daoffst with a value of -1

FUNCTION: D_FLDLOC

DESCRIPTION:

Given the database pointer search for given field (will call (D_SYMLOC).

TO CALL:

```
= D_FLDLOC (logon,dbdtab,flename,flddef,fldbuf);

struct user * logon;                   /* input - the database
                                          */
struct usrhskp *dbdtab;                /* input - the fieldname
                                          */
char *fldname;                         /* output - where field
                                          is in dbd */
struct dbdbd **flddef;                 /* output - ptr to
                                          contents of field */
char **fldbuf;
```

RETURNS:

```
flddef - pointer to where field info starts in dbd
       - if no field by that name -- return NULL for flddef
fldbuf - NULL if no data read in yet!
```

FUNCTION: D_SYMLOC

DESCRIPTION:

This function is used by D_FLDLOC to search the dbd for a given field name.

TO CALL:

```
= D_SYMLOC (dbdtab,fldname)

struct usrhskp *dbdtab;        /* input - ptr to database */
char *fldname;                 /* input - fieldname */
```

RETURNS:

return (-1) error, (0) successful

FUNCTION: D_FNDKEY

DESCRIPTION:

This function given a key value, a search operator (root,key, nxtkey), database name and logon. Search through directory for key and obtain index file offset. Read (if key exists) index file information into memory, storing pointer. Does not read any data from data file.

TO CALL:

```
= D_FNDKEY(logon,dbdtab,keyval,operator,keycnt);

struct user * logon;           /* input - user logon ptr */
struct usrhskp *dbdtab;        /* input - keyword */
char *keyval;                  /* input - search operator */
short operator;                /* input - database pointer */
long *keycnt;                  /* output - keycount for that key
                               */
```

DEFINES:

```
#define D_KEY 1
#define D_ROOT 2
#define D_NXTKEY 3
```

RETURNS:

return (-1) error, (0) successful

FUNCTION: D_GETDAT

DESCRIPTION:

This function reads the datafile and sets up buffer for information to fit into and then links the array of pointers to each field value.

TO CALL:

```
= D_GETDAT(logon,dbdtab,daoffset);

struct user * logon;          /* input - user logon ptr */
struct usrhskp *dbdtab;       /* input - database pointer */
long daoffset;                /* input - datafile offset to start
                              */
```

RETURNS:

return (-1) error, (0) successful

FUNCTION: D_GETSEQ

DESCRIPTION:

This function reads the datafile and sets up buffer for information to fit into a buffer and then links the array of pointers to each field value. This function performs a sequential read.
(****when pos = 0 (start at top of file, if pos> 0 start reading at that record).

TO CALL:

```
= D_GETSEQ(logon,dbdtab,pos);

struct user * logon;          /* input - user logon ptr */
struct usrhskp *dbdtab;       /* input - database pointer */
long pos;                     /* input - start position of search
                              */
```

RETURNS:

return (-1) error, (0) successful

FUNCTION: D_SETDAT

DESCRIPTION:

This function reads the datafile at the lowest level given a file pointer and offset.

TO CALL:

```
= D_SETDAT(daoffset,fp)

long daoffset                          /* input - offset into
                                       datafile */
file *fp                               /* input - file pointer
                                       */
```

RETURNS:

return (-1) error, (0) successful

FUNCTION: D_LISTFLDS

DESCRIPTION:

Returns an integer representing the number of fields in a relation and the field name concat. together as strings.

TO CALL:

```
= D_LISTFLDS (logon,dbdtab,numflds,fields;

struct user * logon;                   /* INPUT */
struct usrhskp *dbdtab;                /* INPUT */
int *numflds;                           /* OUTPUT */
char **fields;                         /* OUTPUT - */
```

RETURNS:

Error (-1), Successful (0)

FUNCTION: D_MAKKEY

DESCRIPTION:

This function creates the key made up of both the dbd field number and value in ASCII format, given the field name and key value.

TO CALL:

```
= D_MAKKEY(logon,dbdtab,fldname,keyval,madekey);

struct user * logon;          /* input - ptr to user node */
struct usrhskp *dbdtab;       /* input - ptr to db opened */
char *fldname;                /* input - database fieldname */
char *keyval;                 /* input - desired key value */
char *madekey;                /* output - key generated */
```

RETURNS:

Error (-1), Successful (0)

FUNCTION: D_KEYCNT

DESCRIPTION:

This routine searches for a specific key, if found, returns the index file offset and a keycount. If not found, a negative keycount is returned.
NO INDEX BUFFERS ARE SET UP!

TO CALL:

```
= D_KEYCNT(logon,dbdtab,keyval,operator,keycount,indxoff);

struct user * logon;          /* input */
struct usrhskp *dbdtab;       /* input */
char *keyval;                 /*
short operator;
long *keycount;               /* output */
long indxoff;                 /* output index file offset */
```

RETURNS:

0 (success), -1 (failure)

FUNCTION: D_INDXGET

DESCRIPTION:

Given the index offset of the inverted file, set up the index buffer.

TO CALL:

```
= D_INDXGET(logon,dbdtab,key,indxoff);

struct user * logon;            /* input */
struct usrhskp *dbdtab;         /* input */
char *key;                      /* input */
long indxoff;                   /* input */
```

RETURNS:

0 (success), -1 (failure)

III. DESIGN REQUIREMENTS

In this section an overview of the DBMS design and memory management is outlined, along with the final file design.

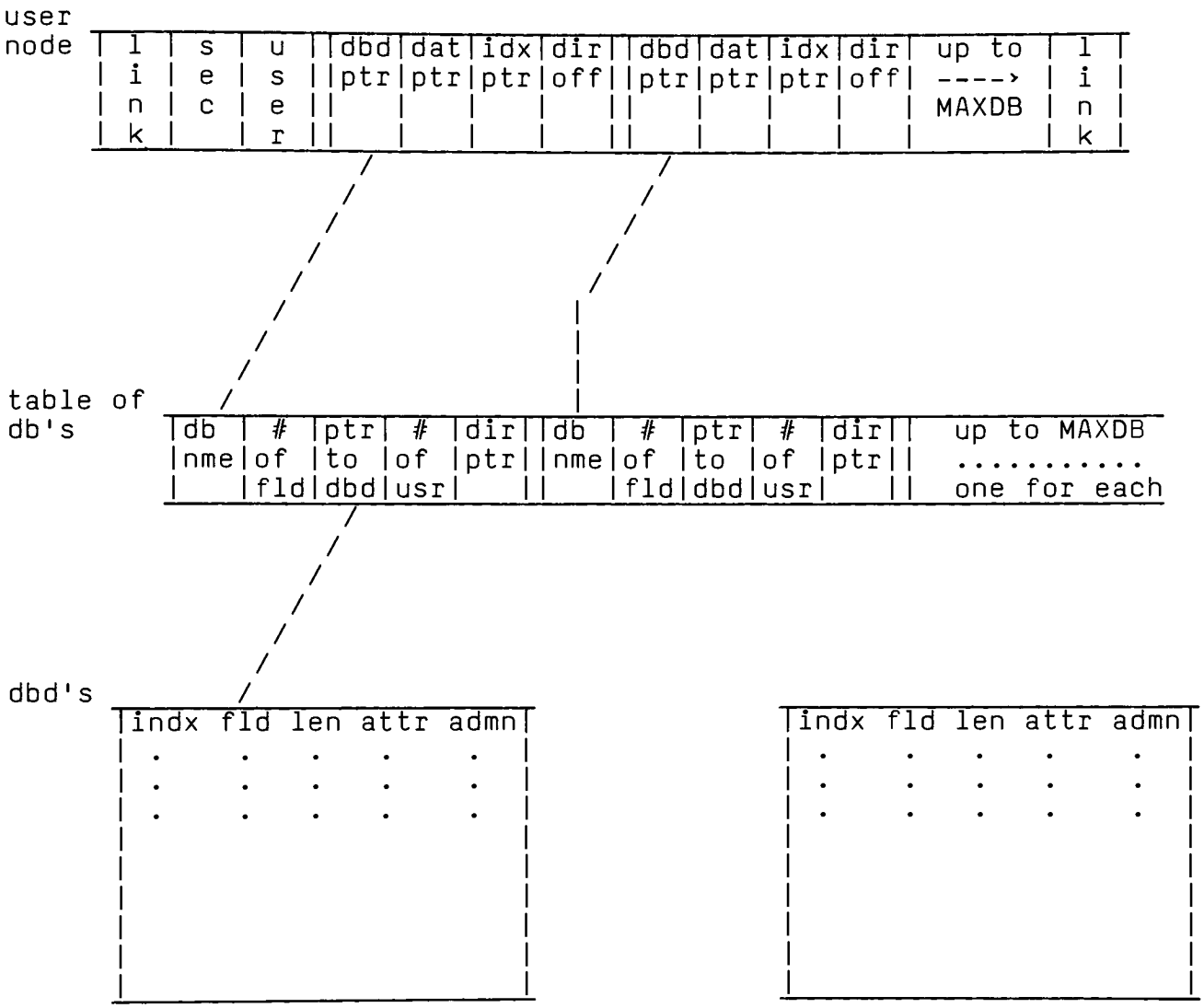
III.1 DBMS Internal Declarations

NEED:

```
# define MAXDB 10

struct user                                  /* alloc dynamically for
                                             each */
        {                                    /* user as a linked
                                             list      */
        struct user *blink;                  /* backward ptr */
        char security[6];
        char usrid[8];
        char tty[8];
        struct usrhskp *maxdb;               /* chg raf 84.3.3 */
        struct user *flink;                  /* forward link */
        };

struct usrhskp                               /* per user per db - keep
                                             buffer */
        {                                    /* pointers
        struct dbopn *dbdptr;                /* ptr to dbd table */
        char **datbuf;                       /* ptr to data buffer */
        struct bucket *inxbuf;               /* ptr to index buffer */
        long diroffst;                       /* directory offset */
        short *curlen;                       /* ptr in index buffer */
        };

struct dbopn                                 /* opened database info
                                             and ptr */
        {                                    /* to its dbd
        char dbname[8];
        FILE *ptri;                          /* index file pointer */
        FILE *ptrd;                          /* data file pointer */
        FILE ptrr;                           /* directory file pointer
                                             */
        int numflds;                         /* no. of fields */
        int nusers;                          /* no. of users using
                                             database */
        char *dircroot;                      /* prt to directory root
        struct dbd *dbptr;
        };

struct dbopn tabptr[MAXDB];                  /* allow max of 10
                                             databases opened */
```

```
struct dbdbd                                    /* holds the database
                                                definition */
        {
        short fldindx;                          /* index back into datbuf
                                                */
        short len;
        char attrib;                            /* field attributes */
        char admin;
        };

/*-----------------------------------*/
/*  need to define additional ptrs  */
/*-----------------------------------*/

struct user *uhead,utail;          /* keep track of user nodes */
struct user *lastuid;              /* ptr to last user id */
```

III.2 DBMS Internal Memory Layout

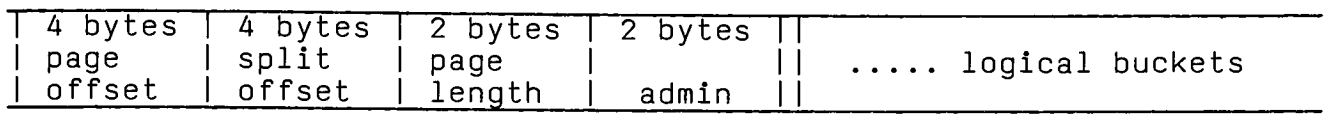


***the above diagram is a general overview of the internal mechanism.

III.3 DBMS File Structure

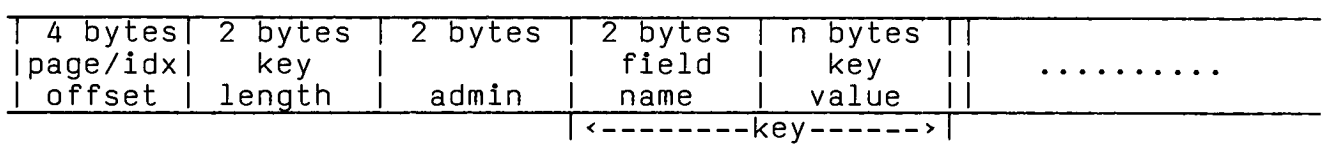
DIRECTORY FILE

Internal page layout:



12 bytes overhead / page

Internal logical bucket:



8 bytes overhead / bucket

DIRECTORY FILE DESCRIPTION

The directory is the highest level of access to any specific database. This file is comprised of a hierarchy of 1024 byte pages which eventually contain the location of specific keyed fields in the index file.

Each 1024 byte page contains twelve bytes of overhead which define the specific page. This page overhead is defined as follows:

page offset (4 bytes) - this is the location of this page in the directory file. This is used to traverse through the hierarchy of pages.

split offset (4 bytes) - unused.

page length (2 bytes) - this is the number of bytes that are presently used in this specific 1024 byte page.

administration (2 bytes) - this defines the tree level.

Following this twelve bytes of overhead on each page are what are referred to as directory buckets. Each bucket contains the value of a specific keyed field in the datafile. Each bucket has eight bytes of overhead defined as follows:

page/index offset (4 bytes) - (refer to administration of page overhead)
if the present page is not the most dense level of the directory tree, this value is the location of the beginning of next lower level page in this directory file; this value is used to traverse through the levels of the directory to eventually point into the index file.
if the present page is the most dense level of the directory file, this value is the offset into the index file of the specific keyed field index bucket.

key length (2 bytes) - this is the number of characters of this keyed field value, this is always greater than zero.

DIRECTORY FILE DESCRIPTION

administration (2 bytes)	-	the first bit of the first byte is the delete bit, if this bit is a one, the key value has been deleted from the datafile; all other bits of these two bytes are reserved for future development.
database field name (2 bytes)	-	this value is the number of the field definition as defined in the beginning of the data file (e.g., if this value is five, the field name of the fifth field definition of the datafile is the name for this field).
keyed field value (n bytes)	-	this is the actual value of this keyed field, it is at least one byte in length and defined by key length.

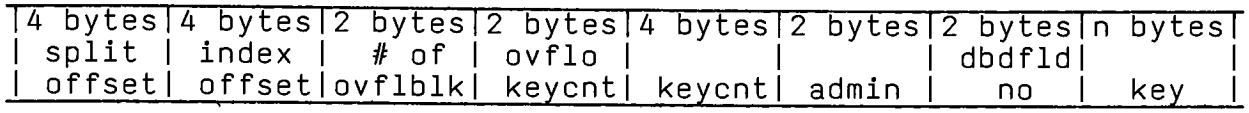
This concludes the definition of the composition of a directory file page. The following is a description of how a collection of these pages are used in a hierarchical structure to define the actual database directory.

At the lowest level of the directory file is the value for each keyed field in the database. The page/index offsets of the directory buckets point directly into the index file. These key values are sorted alphabetically from lowest to highest and by field number starting with the lowest. The next highest level contains only as many directory buckets (and pages to contain them) as there are pages in the lowest level. Each page/index offset of each bucket at this higher level is the location of the beginning of a page of the lower level. The corresponding key values of these higher level buckets are the highest key values of the lower level pages each bucket points to. The pages on any level do not necessarily have to be full. This leveling of pages halts when the highest level contains only one page.

DBMS File Structure (cont'd)

INDEX FILE

Internal header layout:



20 bytes overhead / header

Internal index file logical record:

8 bytes / logical record

INDEX FILE DESCRIPTION

The index file contains information which is used to find the database file records which contain a given key value. For each unique key, there will be any entry in the index file. If a key appears many times in the datafile file, the locations in the database (which are the "database file offsets") of each occurrence will appear in the index file as a list of "index buckets". Specifically, if "name = 'mark'" appeared in 12 database file records, then the index file would have one entry for "name = 'mark'" and this entry would look like this:

[overhead information] [key value] [index buckets]

"overhead information" will be described below.

"key value" will contain the key value 'mark' terminated by a backslash zero. The key value is of course variable length.

"index buckets" will contain a list of the 12 database file offsets where 'mark' appears in the database file. Each database file offset in the list of 12 will also have 2 bytes of "sent" information - this is used to indicate such things like whether or not this key or record has been deleted, etc. In addition, a 2 byte field for word has been added for text.

Suppose that a 13th database file record with "name = 'mark'" were to be added to the file. In our example, there would be no room in the list of 12 index buckets to put the added database file offset. We will, therefore, add a new "entry" for this key to the bottom of the index file. This entry will be called a "continuation" entry. This new entry will look much the same as the first entry above, except some of the overhead information will be different to tell us that it is indeed a continuation entry. Also, because the [key value] appears in the original index file entry, the key value will not be written into the continuation entry. This continuation entry will be linked to the original incex file entry using a parameter from the "overhead information".

Now for the overhead. The index overhead for a key value contains:

"split" -- the offset at which the first/next continuation entry is located. Split will be zero if ther are no continuation entries (or if we are already in a continuation entry, then split will be 0 at the last continuation entry).

"idxoffst" -- the location in the index file at which this index entry is located ("where we are now").

"nopblks" -- the total number of continuation entries associated with this key value. Nopblks will be zero if there are no continuation entries. Nopblks will never be higher than "OVERBLOC", which is defined in /rellib/idxovh.h.

"nookeys" -- the total number of filled index buckets for this key value that appear AFTER this index entry (i.e., in later continuation blocks).

"keycnt" -- the number of filled index buckets in THIS index entry...does not include the buckets that may appear in later continuation entries.

"admin" -- the total number of buckets (filled + deleted) that appear in THIS index entry (again -- does not include the buckets that appear in later continuation entries). For the first entry for any given key, "admin" will equal the number of occurrences for the key that was found when the database was loaded. For continuation entries, "admin" will equal "CONTINUE" which is defined in idxovh.h.

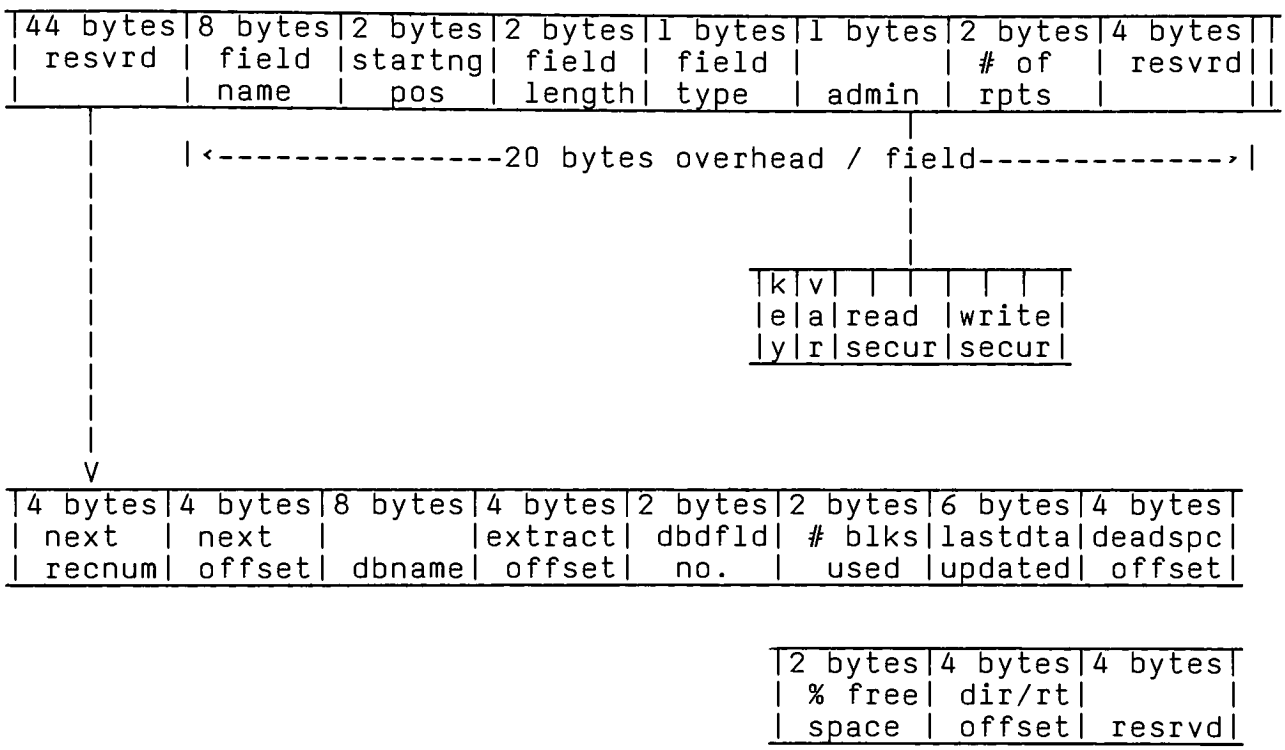
NOTES:

The total number of occurrences of a key can be found by adding "nookeys" and "keycnt" for this key's first index entry.

DBMS File Structure (cont'd)

DATA FILE

Internal dbd layout:

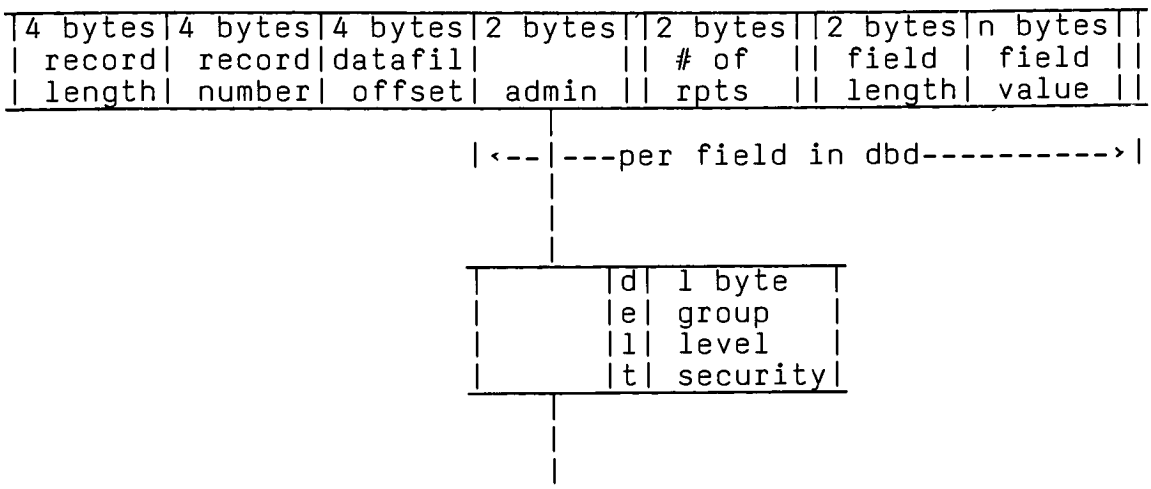


44 bytes overhead / 1024 dbd block

DBMS File Structure (cont'd)

DATA FILE

Internal raw data record:



14 bytes overhead / datafile record

DATA FILE DESCRIPTION

The datafile (name.dat) contains the database definition at the top of the file followed by the actual data. The database definition, referred to as the dbd layout is merely a definition of how the data is structured in the lower part of the datafile.

This definition is broken into blocks which are 1024 bytes in length. The first forty-four bytes of each of these blocks is reserved for the overall definition of the datafile. This forty-four byte overhead is valid in the first 1024 byte block only. In any successive blocks, this reserved area can be ignored.

The forty-four byte database definition contains the following data in the defined order:

next record number (4 bytes)	-	this is the value (sequential number) which is assigned to the next data record which is added to the specific datafile.
next record offset (4 bytes)	-	this is the number of the byte starting from the beginning of the datafile (starts with byte 0) where the next data record is to be added. (Used to quickly position at the end of the data records.)
database name (8 bytes)	-	this is the ASCII name of the file in which the data is stored, excluding the extension (.dat).
extract offset (4 bytes) (zero at first)	-	this is the byte at which the next invocation of extract will start; if data records have been added after extract has been performed, this value will not be equal to the number of blocks used times 1024 bytes per block.
database field number (2 bytes)	-	this defines the number of fields contained in each record of the specific database; this can never change without creating a new database.

number of blocks (2 bytes) - this is the number of 1024 byte blocks contained at the beginning of the datafile before the actual data values.

last date updated (6 bytes) - this is the last date that either a load or update was performed on this datafile; the format for the date is MMDDYY.

The remaining fourteen bytes are reserved for future design.

Following the forty-four reserved bytes are blocks of twenty bytes, each block being a definition for a different field. For example, if each record contained 10 fields, there would be 10 field block definitions or 200 bytes following the forty-four reserved bytes. Each of the 1024 byte blocks can contain at most forty-nine field definition blocks. If there are more than forty-nine fields per record, a second (or third, etc.) 1024 byte block will be present to hold those definitions.

Each of the twenty-byte field definitions contain the following data in the defined order:

field name (8 bytes) - this is the ASCII field name; if the name is less than eight characters, the field is padded with blanks.

starting position (2 bytes) - this value is zero.

field length (2 bytes) - this is the maximum number of bytes of the specific field (limited to 32k bytes) for a variable length field; for non-variable length fields, this is the field length.

field type (1 byte) - - this defines the type of the field; valid types are c(C) for character, i(I) for integer, f(F) for float (double percision), t(T) for text.

administrative (1byte) - a reserved byte.

number of repeats (2 bytes) - this contains the maximum number of repeated fields, a zero if the field is not repeated.

The remaining four bytes are reserved.

Following the 1024 byte blocks is the start of the field values called the internal raw data records. Each data record has fourteen bytes of overhead defined as follows:

record length (4 bytes) - this is the length (in bytes) of the whole record, including the fourteen bytes of overhead.

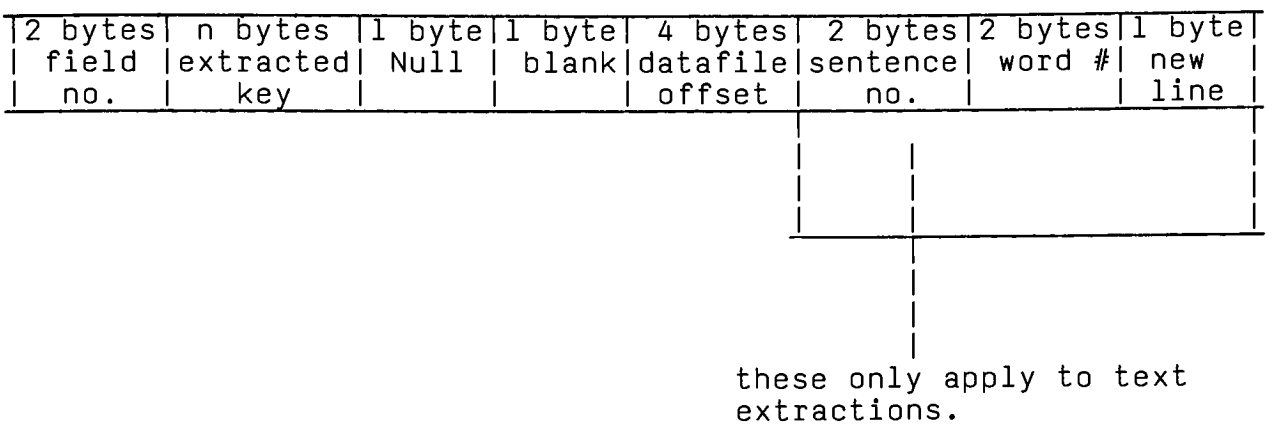
DATA FILE DESCRIPTION

record number (4 bytes)	-	this is the number of the specific record in the data-file, all records are sequentially numbered starting with one.
datafile offset (4 bytes)	-	this is the position of the first byte of the specific record in the data file starting with byte 0; this includes the 1024 byte block(s).
administrative (2 bytes)	-	this was never used.

After the fourteen bytes of record overhead are the actual definitions of the field values. Each field value is defined as follows:

number of repeats (2 bytes)	-	this would contain the number of repeats, but never implemented.
field length (2 bytes)	-	this is the length of the field in bytes; if this value is zero, the field does not exist. Certain fields have fixed lengths; for example, i is 4 bytes, f is 8 bytes.
field value (field length bytes)	-	this is the value of the field as described in the 1024 byte block; if the length is zero, this value does not exist.

EXTRACT FILE



IV VERIFICATION AND VALIDATION

IV.1 Test Plan

- the driving force behind the test plan is two fold:

1) to thoroughly test and validate all the utility functions and properly build the necessary files for different relations.

2) to test the internal database management routines and all the low level access functions.

IV.2 Test Procedure

- In order to meet the above requirements, an extensive and nearly exhaustive number of schemas were defined, along with input data for each relation. Each of the test cases listed on the following page were correctly loaded and found to be usable. In order to verify that the files were correct, the second requirement of the test plan had to be met. A forms-based test program was developed to demonstrate the usage of the database management entry point routines, as well as the capability of navigating correctly through the files.

The test program, using "curses", provided a mechanism to open multiple relations, as well as navigate through the relation, with via a keyed search or a sequential search.

IV.3 Test Results

- The test results speak for themselves when you use the tools availale.

For all the different schemas, the relations loaded correctly. Upon successful completion of each load, one could navigate through the relation using the test program which utilizes the dbms primitives.

V CONCLUSION

V.1 Consideration Involved When Designing the File Structure

I assume that the reader is familiar with B-trees [3,17,22,32,33] and with derivatives of B-trees called B*-trees [3,4] and prefix B-trees [4], as well as inverted files [6,16,17,25,28] and their variants. The database file structure implemented within the scope of this thesis reflects many considerations. The structure had to support general purpose UNIX applications, relational views, dynamic growth, flexibility, ease of use, and most importantly, the management of both data and text.

Many papers have been written which describe versatile file structures such as AVL-trees [17], B-trees, B$^+$-trees [9], B*-trees, dense trees [12], VSAM [20], ISAM [19], inverted files and others, but few, if any, address the same issues that I have tried to explore. Each of the above provide specific functionality as a general purpose index structure, but none serve as a stand-alone entity providing a database structure with flexibility and generalization together with the integrated solution of managing both data and documents.

Due to the above requirements, it was apparent that a single file structure would not be sufficient and that an index structure, as mentioned above, was necessary but married with other enhanced file structures.

As my research commenced with respect to the exact index structure, I had to weed out those which were not appropriate, as well as those which were too complex or required too much time for the scope of this thesis. I selected to implement, as the primary index structure, a variation of an m-way tree, prefix B-tree and B*-tree which guarantees height balancing with variable length keys and separators. The implemented structure, though not a B-tree in the theoretical sense, provides many advantages and performance efficiencies inherent to B-trees, while meeting the specific needs of the database structure. The index structure generated stores pointers to the records in the index file, together with the keys identifying them only as the leaves, and all other non-leaf branch nodes consists of key and offset information for navigating through the tree. This characteristic most resembles a B*-tree where the actual addressing information is carried, in entirety, at the leaf level. To maximize the number of separators per node and facilitate the management of textual information, each node or page consists of variable length separators and keys, while providing dense usage of each node down to the leaf level. At the leaf level, a vast amount of information is kept to further navigate through the other associated files. The underlying difference with the dense leaf level of this B-tree structure is that it holds pointers to a record within an inverted file and not the actual data record(s) themselves.

The extra level of indirection to the inverted file allows me to take advantage of such a structure, particularly when dealing with textual and boolean operations. The inverted file provides a means to perform merge processing on its lists, in order to satisfy a search criteria without actually searching the data. This file, though providing many strengths, is not necessary just to store lists of reference to non-unique keys. An equally important reason for it in this design is to provide the capability of proximity or content searching of textual information. Because the inverted files serve two purposes, it has been augmented to store both word and sentence numbers from the textual information it was extracted from. These two fields are not used when dealing with non-textual information and provides a uniform structure independent of the data types origin. This was one of the major changes to the file structure in order to handle textual information. In many of the commercial-based information storage and retrieval systems, this technology is also utilized. Though not ever completely implemented, modifications to the inverted file were added to manage both textual information as well as data. Ultimately, the primary function of the inverted file is to point to the specific data records that contain keys imbedded within the data.

When choosing a database file structure scheme, there are advantages as well as disadvantages. To outline some of the tradeoffs would be a very effective approach to evaluate its strengths and weaknesses.

In order to discuss the disadvantages or potential shortcomings of the design, one must look at the most publicized deficiency of the UNIX operating system, its filesystem. The filesystem has strongly been criticized for its multiple levels of indirection when storing large files. Though it is not clear what all the implications of such a structure are, it certainly has to be a concern to anyone designing a database file structure. With the current three file structure implementation per relation, I have concerns that as these files grow, the levels of indirection between both the UNIX filesystem and the database file structure may severely degrade performance. A second issue that comes to mind is the limitation on the number of files that can be opened by any one process simultaneously. Again, with the need for three files to be opened per relation when keyed searching is desired, such a limitation could pose problems when multiple relations are opened. Certainly, there are other shortcomings of the current design, such as how to effectively implement maintenance, concurrency control, etc., which were consciously omitted due to the scope of the thesis.

Conclusion (Cont'd)

There are a number of advantages to the implementation set forth. In deciding to handle both text and data, it quickly became clear that a dense index scheme should be used over a sparse one. With a dense index, such as an inverted file, all records are keyed or have an entry in the index structures. What this provides, over a sparse index, is the ability for boolean or conjunction searches to be decided upon at the index level, as opposed to at the actual data pages. This implies that most searches should require less I/O operations in order to determine whether a particular search criteria can be met. A second, very important advantage of this package, is the current implementation of the B-tree structure. By compressing the key in the structure and handling variable length keys and separators, the B-tree quickly becomes very applicable for many applications.

As one can surmise, there are many considerations when designing a database file structure. It was the goal, as previously stated, to provide an underlying structure which supported, through relational views, both data and text.

V.2 Implementation Evolution

The evolution of this package was much like that of many other systems. A crude functional specification was formulated for the file structure, the utilities, and the DBMS primitives. Once that was completed, a more detailed design specification was outlined for certain portions of the system.

The order in which development proceeded was very critical. Each step was a milestone in determining its success or failure. I set out to develop the necessary utilities to produce the component files as I envisioned them. The tasks were to first take input, of some fashion, and generate a datafile. Second, to produce the inverted file and then lastly, the B-tree structure. When this had been accomplished successfully, the real test was to develop the database primitives to navigate through the files efficiently and accurately. Though there were three major functional components of the thesis -- file structure, utilities, and DBMS primitives, they were tightly coupled.

As the utilities began to take shape, many issues arose which caused altercations to the file structure. It was the intent, through the evolution of both the file structure and the utilities, that the DBMS primitives would remain isolated and unaffected.

The following pages describe some of the implementation considerations and/or road blocks encountered while developing the system.

Implementation Issues and Road Blocks

When designing and implementing a large software system, there are always some problems and oversights. With the changes in today's hardware technology and scope of this thesis, many of the concerns or what seem to be "off beat" design issues that INGRES [28], System R [1] and others made, are not issues from my perspective.

In as much as the design and implementation of this project was successful, a few issues arose which are worth discussing. The implementation issues or roadblocks can be divided into three categories: 1) UNIX considerations, 2) file structure considerations, 3) general "C" programming language considerations.

Unix Considerations

There are many peculiarities with every operating system and its supporting utilities. One of the design concerns was to develop this package to be as upward compatible as possible. This meant being aware of blocking sizes and variables which are used by the operating system, but could change in new releases. The intent was to use standard conventions of buffering, etc., so that if block sizes, etc. change, which they did from BSD4.1 to BSD4.2, the source code would run with little change in the most efficient manner.

A second issue which was encountered was related to an idiosyncrasy in the UNIX sort. As might be anticipated with such a system, a great deal of sorting needed to be performed to build the indices. Unfortunately, I chose to implement all the file structures in binary and not ASCII format in order to compact space within each file. What this meant, was that the sort did not work properly. To facilitate the proper execution of the sort, those pieces or fields of a given record that needed to be sorted, had to be converted to ASCII strings terminated by a '\n'. Upon successful completion of the sort, a conversion back to binary was performed.

File Structure Considerations

Defining a file structure requires an assessment of the classic space vs. time tradeoff. The file structure had to correctly represent the relation defined by the user, as well as provide a flexible way to maintain it. Typically, all the data placed on the disk was represented through "C" language structures. I decided to maintain the relation definition at the top of the data file, so that as a relation was opened, I quickly gained all the information for that relation.

Programming Language Considerations

The last concerns are those surrounding the "C" programming language and its restrictions. It has been the intent to keep the code as portable as possible. One of the ways to provide maximum portability was not to use any system-dependent, low-level routines, to cast everywhere possible, use all the facilities available in the STDIO.H library, and avoid the use of integer as a declaration type.

Most of the code was written using the above rules, so that the style and methods are consistent. Two problems arose part way through the development, when using the sizeof function and FOPEN/FCLOSE routines. Due to unexpected results, and alignment problems, the sizeof operator did not always return the value expected. In certain situations, this caused severe problems and so I later defined macros which represented the expected size of certain structures. The second problem arose when using the FOPEN/FCLOSE routines in that they seemingly wiped out files when trying to perform backward seeks on the files and rewriting a portion of the file. This problem arose on a single utility and so the less portable OPEN/CLOSE routines were used instead.

V.3 Concurrency Control and Locking

Concurrency Control is one of the most complex and prominent issues in database technology. This is the ability to allow multiple users to share the same data for retrieval and maintenance. Certainly, many systems provide this facility through both hardware and software. It is the purpose of this portion of the thesis to discuss the general issues of concerrency control for B-trees and other files.

The logical place for concurrency control decisions to be made is within the access method. Although the current implementation of this thesis project is a tightly coupled access method and database manager, a separation of the two could be made. With such a layered approach, the access method would be independent from the database manager and could, in effect, perform many operations without affecting the other layers. At a subordinate layer, the lock manager could be implemented, suggesting that the access method is a layer above the locking scheme. It would be the responsibility of the access method, on an I/O operation, to perform calls to the lock manager; who would, in turn, maintain the lock table and perform any necessary checks. The lock table, must naturally be a common or shared file which is used by any number of processes for any number of disk files or disk pages.

Although it is outside the scope of this research to discuss in-depth the internal workings of the lock manager, its concurrency granularity, or the actual set-up of the lock table, it is still useful to discuss some general approaches to lock management. Though the definition of concurrency control is much the same for most file structures, the B-tree structure poses some challenging problems unique to itself. For this reason, the major thrust of this discussion will be geared toward B-trees.

There has been a great deal of research in the area of B-trees and its variants, along with how to provide suitable concurrency control for such structures [3,22,23]. One of the factors that comes into play when designing an access method, is the design of the concurrency control mechanism. The two entities, the index structure and access method, are typically coupled together by a lock manager, which in the UNIX environment, is implemented through pseudo I/O device drivers, shared memory or IPC.

Regardless of the implementation, certain locking decisions have to be made as to the approach the lock manager will take when ensuring concurrency control on any file, and more specifically, the B-tree. There are various techniques used to tackle the difficulties which arise because of concurrency in balanced tree structures, but basically they all fit into the two categories discussed below.

It should be noted that all these solution make sure of the following observation. For any updater operating in a sequential environment, there exists a node which is the root of a subtree beyond which all changes in the data and structure due to the update cannot propagate. This is referred to as the safe node. The deepest safe node for an updater is typically referred to by the smallest subtree affected by the updater, which is the deepest in the access path.

Definition: A node in a B-tree is insertion-safe (or i-safe) if it is unsaturated, and it is deletion-safe (or d-safe) if it is not minimal. It is assumed that the reader is familiar with the requirements of a B-tree.

As previously mentioned, there are two basic categories of solutions to concurrency control of B-trees. the two categories can be summarized as follows:

> Type 1: The scope for an updater remains invariant during restructuring, which implies that an updater must lock its scope so that no other updaters can be in it.
>
> Type 2: The scope for an updater may change during restructuring. Only the nodes affected by an updater at each restructuring step are locked and made off-limits to other updaters.

Certainly, some semantic or heuristic approach has to be decided upon, based on the operational requirements of a specific system. Both approaches have tradeoffs, strengths and weaknesses and so when designing the access method and the lock manager, the environment and requirements must be closely evaluated.

A simple solution to the problem of concurrent access would be to strictly serialize all updaters, by requiring each updater to gain exclusive control of the tree, e.g. by placing an exclusive lock on the whole tree - before it begins accessing it, thus preventing all other updaters and readers from altering or reading the index while the update takes place. Readers, on the other hand, could access the structure concurrently with other readers. Clearly, this approach could only be used if there is very low activity against the tree.

In order to provide a higher level of concurrency control, some form of placing locks on nodes or pages must be used. In general, all of the contemporary approaches utilize the compatibility and convertibility graphs, known as CCG, based on the different kinds of locks (read locks, write locks, exclusive locks, read-write locks, etc.) to be used and when.

The important issue for this discussion is the functionality of each layer. If one is to design an access method which deals with homogeneous or heterogeneous file types, it should be the access method which requests the locks of the specific nodes or pages. It is the sole function of the lock manager to service the lock requests from the access method. Based on the algorithm used by the access method and the interaction with the lock manager, the proper concurrency control can be ensured.

It should also be understood that a layered approach to locking carries certain strengths and weaknesses. Certainly, its strength is the independence of each layer and the ability of the lock manager to easily interface to many different access methods. But with any layered architecture, you almost always pay the price of less efficiency due to interlayer communications.

An attempt has been made to describe concurrency control, in a general sense, through a layered approach. The intent was not to reiterate the dozen or so different algorithms that can be used by the access method, in order to determine whether a B-tree node is safe or not, but to outline the functionality of the file structure, the access method and the lock manager. I hope it has become clear that through a layered approach, the access methods responsibility is solely to retrieve the proper information. In an attempt to do so, it must utilize certain concurrency algorithms and communicate directly with the lock manager for the request of specific locks. At a layer below and independent from the access method, the lock manager simply honors valid lock requests and properly maintains them. With such an approach, it is apparant that any system that uses either heterogeneous or homogeneous files, can be used in a multi-user environment for both retrieval and maintenance.

V.4 Integration of the File Access Method Within the Kernel

The beauty of the UNIX operating system is its flexibility and portability. Certainly, if one were to modify the operating system in order to integrate an access method into it, there is a great probability that the operating system would not remain as portable and "pure vanilla". In certain environments this may not be an issue, but at this point, portability is what is truly driving the success of UNIX.

If one can overlook the issue of modification of a standard operating system, there may be many advantages to an integrated file access method, and even possibly a database management system.

The first positive point that comes to mind, is that with such an integration, locking can be performed at a lower level, than with the typical layered approach of a FAM or DBMS. The UNIX operating system provides, at this point, minimal locking and concurrency control, far inferior to the requirements desired by a FAM or DBMS. If by modifications to the operating system, a more sophisticated locking/concurrency control scheme could be integrated, certainly this would be a more efficient and powerful facility. Many studies have been done which show that DBMS applications show very little sequentiality in the way they process information. What this implies is that many forms of Input/Output buffering performed within the FAM/DBMS is not beneficial and may even be detrimental. In certain environments, what may often occur is double page faulting, whereby the FAM/DBMS determines that the record buffer it has is not sufficient to perform a given search and performs some psuedo-page fault. As a by product of this activity, the operating system also performs a page fault, and the result is very inefficient. This kind of problem could be avoided if the FAM/DBMS was an integrated component of the operating system. Lastly, without hard-set proof, I feel that such an integration could be nothing more than a more efficient way to implement an access method because a layer has been removed and all other inherent communication between the operating system and FAM/DBMS goes away.

Though my research was not involved at the operating system level, such an idea sets off some stimulating thoughts. This integration would provide a number of terrific advantages, but does not necessarily satisfy the generalized requirements that I set out to accomplish.

V.5 Conclusion

When dealing with the many complex issues of database technology, and more specifically the relational model, tradeoffs are made. For every idea, there is a counter idea, and so I believe that this thesis, was in fact, a successful exercise, allowing me to explore many avenues. Certainly, this idea or project could be enhanced in the areas of the file structure, maintenance, dynamic creation and reorganization, multi-user optimization, garbage collection and concurrency control. It was not an exercise to provide a commercial based product, but yet to justify some of the ideas and thoughts I had in such an approach.

In looking back at this approach and some of the research that addresses a combination of models, a relational-network model [24] is very intriguing. A future extention of this approach could be to provide or resolve some of the typical concerns with relational systems. Such enhancements may be repeated fields, fast-track join indices or expansion of the data model to capture more semantic content of the world.

An intriguing alternative (as opposed to extention) to the design of a relational database file structure which handles both data and text could be to implement each relation with two files instead of three. Assuming such an approach is not too complex, one could combine the B-tree and the inverted file, so that the dense level of the tree points to positions elsewhere within the same file. These positions would appear as the inverted file does now but would be in the same file with the B-tree. This would reduce the number of files opened per relation, minimize extra housekeeping, and somewhat diminish the level of indirection currently required.

Database technology is an integral part of today's modern data processing world, but more importantly, it is a technology which continues to address and conquer some of the most complex issues in the industry. I found this thesis project to be one of the most mentally stimulating exercises I have ever participated in. It has allowed me to explore many of the complex issues of the technology and gain immense exposure into areas which I would have never been exposed to before.

VI. APPENDICES

VI.1 PROPOSAL

Rochester Institute of Technology
School of Computer Science and Technology

A Relational-like Database File Structure

by

Robert Fabbio
3/7/84

DEPARTMENT APPROVALS:

approved by: Jeffrey Jaslow, Chairperson 5-10-84

The following signatures represent an approval of this thesis proposal by the above committee and agree to grant permission to register for the thesis course.

Table of Contents

TABLE OF CONTENTS (continued)

INTRODUCTION AND BACKGROUND

Problem Statement

The purpose of this thesis is to investigate an alternate approach (different from INGRES, MISTRESS, YARD) to internal database file structure under UNIX, ultimately developing a structure which is more efficient and flexible for relational tables. In addition to the file structure, a number of low-level I/O routines used to access the data will be provided.

Previous Work

There has been a great deal of published literature on the subject of file structures, access methods, database design, and other related topics. Refer to the Bibliographic section for more detail on the literature relavent to the work reported here.

THEORETICAL AND CONCEPTUAL DEVELOPMENT

The following section briefly describes the internal file structure design.

The major thrust of this work is to develop, through research and good design, a powerful and flexible general purpose relational database file structure which can operate under the UNIX operating system. This task is non-trivial for two major reasons. First, the UNIX operating system does not support a random retrieval access method which is essential for this application. In the absence of such an access method, all processing against a relation would be sequential causing performance to be unacceptable in most situations.

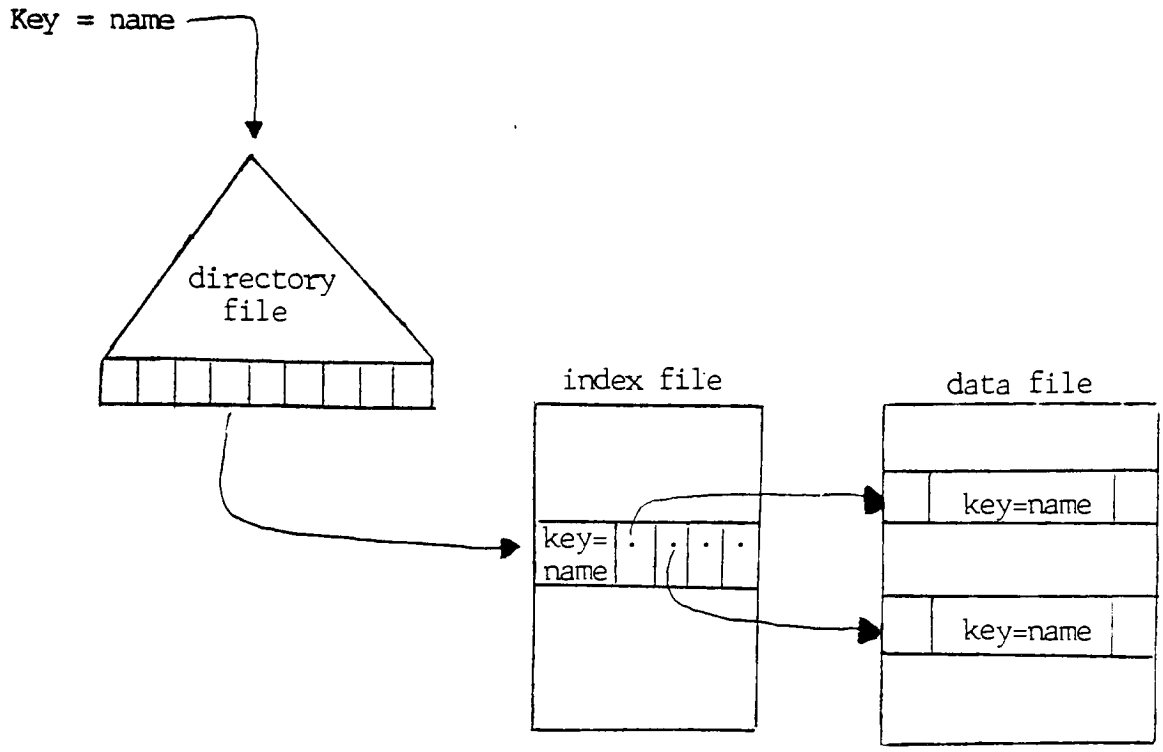
Second, is the complexity of designing and implementing all the necessary additional files needed to support keyed retrieval on any data item, while continuing to portray, from a logical view, a simple table.

The approach taken here will allow a user to search his/her data from a logical view and let the underlying physical file structure perform the searching tasks. Due to the nature of relational databases, all the logical relationships must be formulated "on the fly" and, therefore, are not represented within the internal structure. The physical file structure incorporated for logical database consists of three files. The three component files provide a B-tree structure containing a single entry for each unique key.

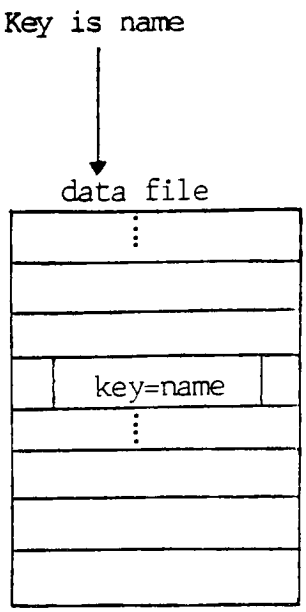
At the most dense level there are pointers to an inverted file which contains an entry for every data file record with that particular key (one- to-one correspondence between the inverted file and the lowest level of the B-tree). With the lists of offsets that the inverted file provide into the data file, we can quickly find all the records with a given key. Together with key retrieval, there is also the capability to sequentially retrieve records by navigating through the data file.

Much of the research reported here discusses the algorithms needed to design and implement both the inverted file and the B-tree structure.

Key Searching



Sequential Searching



THESIS ENHANCEMENTS

When designing a database system, both retrieval and updating must be considered in order to formulate a thorough design. The updating aspects of a database system are usually not trivial, but almost guaranteed to be needed in one form or another.

Although the scope of this thesis does not include the implementation of any maintenance functions, it is a topic which can be addressed. For every database system, there are a number of ways to provide this facility, depending upon the environment, the file structure involved and the maintenance requirements of the applications. These are some of the major issues which will dictate the maintenance algorithms, concurrency control, locking approaches and other related topics.

To address how maintenance could be implemented with the proposed file structure, we must examine each individual component and assess the appropriate tradeoffs given specific operating environments. In light of the following discussions, it should be kept in mind that the design presented here may not have all the necessary functionality; however, each component could be augmented to provide such a facility.

B-TREE

The B-tree structure, which serves as the directory into the inverted file, can be maintained in different ways. Again, this is very dependent upon the operating environment. If the operating environment is a single user application, the tree structure may be dynamically maintained with each request. Although one may encounter some additional overhead or slower response time during maintenance requests, search times should remain reasonably constant and acceptable as a result of a continually balanced tree. In this scenario, the system does not incorporate concurrency control, locking or any other such mechanisms, greatly reducing the complexity of the task.

If the operating environment were a shared multi-user situation, the magnitude of the maintenance tasks grows immensely. One needs to worry about concurrency control of the critical resources, namely the databases, as well as how to lock out the resources when needed. A later discussion will follow that will briefly describe a method to provide suitable concurrency control and locking.

There is no straight-forward approach when attempting to maintain a B-tree in a shared environment. Depending on the number of users and the types of maintenance transactions or requests, a dynamic B-tree implementation may not be advantageous. This is so because as the tree is altered at its lowest level, these changes can potentially ripple up toward the root. This effect can be less significant if the tree structure was created with some additional freespace. In the absence of concurrency control, the state of the tree could be catastrophically altered. The problem arises when multiple users attempt to alter the tree and inevitably some of these modifications ripple up toward the root. Without control of these activities, the tree could be changed simultaneously by multiple users leaving it in an unpredictable state.

The other underlying concern with B-tree maintenance is those issues concerning deadlock or contention. If concurrency control is provided and assuming it need not be exclusive control, as changes arise and are rippled upward, the number of pages (nodes) at the upper levels decrease as the requests for the resources increases, thus causing a severe contention problem. This typically arises in situations where each request locks out a vertically large portion of the tree. Certainly one could alleviate such problems by allowing only one user at a time to update the tree with a simple locking algorithm, but this could result in a severe response time problem. In this situation, it may be more advantageous to maintain the tree with a deferred key process, where the tree does not, in fact, dynamically change, but, instead, changes are flagged. For example, a delete operation may not actually remove the key, but instead flag it as deleted. This solution, although not dynamic, certainly provides a less volatile situation with increased response time and minimal concurrency control. Of course, there are tradeoffs since the tree may

quickly become out of balance, depending on the maintenance activities and one could notice a severe degradation in search times. With this approach, it would be a standard procedure to have to run periodic reorganization utilities.

With the design of the B-tree structure supplied in this thesis, some augmentation would have to be made based on the type of maintenance required. In either case, at the page level of the tree, additional information would be required. In the static approach, information indicating deleted keys and continuation page offset for key additions would be required. In the dynamic approach, information indicating the amount of free space remaining in each page, as well as split offsets are minimally needed. Though the information necessary for either approach is not much different, the symantic definition of how maintenance will be accomplished is much different.

INVERTED FILE

The second component of this database structure is an inverted file. By virtue of its sequential nature, it is always a difficult process to efficiently alter this file dynamically. As with all of the components, freespace could be added to this file at create time so that as insertions or additions are made, they can be done more efficiently. The real problem occurs when the variable list of addresses into the datafile needs to be expanded. There are, as with most situations, many approaches to this problem.

One such approach is to simply copy the entire existing variable list of pointers to a new position in the file and then augment the variable length chain. This could be extremely costly and inefficient. Also, it would require a pointer change in the B-tree for that key. If such a method were used, it would not support sorted keys, unless an overly complex internal chaining scheme was established because of the internal shuffling of the inverted file.

A second approach would be to provide the facility to cause redirection of the initial variable length pointer list to additional areas on the disk to support the added records. This may or may not be a viable approach depending upon the size of the files, the environment, and whether the increased redirection will, in fact, cause a dramatic response time problem due to the additional I/O requests.

A third approach is to implement the inverted file as a multi-list structure which tends to solve a great deal of the maintenance issues of the variable lists, but makes the internal structure much more complex.

If the application environment is single user, the updating issues of the inverted file deal only with the topics described above; concurrency control issues are not present.

In a multi-user environment, the complexity grows; not to the extent of that of the B-tree, but more than with the single user situation. With the inverted file, once a user has locked a page which holds a key, no other user will be able to modify it. In this scenario, one does not need to worry about the rippling of change through the entire file; the tradeoff is a severe degradation in response time when locking out an entire page in order to update a single key.

To implement the second approach of this file structure would require some additional overhead. The file structure as it exists would need an area for indicating the offset for the redirection of new offsets, as well as, information about the overflow pages themselves. Such information may be the number of overflow blocks and the number of overflow keys added.

DATAFILE

The last file in this design is the datafile which actually contains the data. Ultimately, this is the file which needs to be protected from any damage. The maintenance request discussed will pertain to this file, but it should be realized that any maintenance to this file, automatically causes changes to the other two. As previously mentioned, freespace can be added to each record at create time which would make the following maintenance activities more efficient.

If a request for a record is to be deleted, the operation can simply be to flag the record as deleted. If a request for a record is to be added, the procedure could be, in the simplest sense, an addition to the bottom of the file. The request for a change is the most complex of the three activities. If the changes to the original record require that the record grow larger than its previous space requirements, it must be moved and could be treated as a delete and then an addition. If the space consumption does not change or decreases in size, the record can be put back where it was without any apparent changes to the datafile.

As with the other files, if one is running in a single user environment, the mechanics remain exactly as described above, without any other complications. Once again, the real complexity arises in the multi-user environment. The complexity here is much different, since this file does not cascade dynamically throughout with a maintenance request. The real problems arise during the locking operation and concurrency control.

In order to implement maintenance with the current design of the data file, very little, if any, would have to change. Depending upon the environment, however, a very sophisticated locking scheme may be needed.

CONCURRENCY CONTROL AND LOCKING

The real issues surrounding multi-user activity in a database system is ensuring integrity of the data. This typically is a very difficult issue because its implementation is based on a great many factors. Some of those governing factors include: the operating environment, support the operating system provides, and the underlying file structure. In the UNIX environment, little support is supplied from the operating system, so a typical approach is to write a lock manager. This facility, which usually is implemented in memory, keeps track of all the locks requested for specific pages on the disk. The three basic modes of locking are read, write, and read with the "intent" to write. One of the common ways to implement the lock manager is to model it as a pseudo-I/O device. Thus, the I/O driver serves as the interface between the lock manager and the file access method.

SUMMARY

It should be realized that much of the design and implementation issues are very dependent upon the desired capacity of the system and the intended operating environment. In order to implement maintenance in either operating environment, an integrated combination of approaches for each of the files must be applied, in order to preserve the integrity of the database.

The purposed three file approach provides advantages as well as disadvantages. It is the intent of this thesis to explore these maintenance issues and to incorporate findings in the conclusion of this thesis.

FUNCTIONAL SPECIFICATIONS

Functions Performed

The system will allow a user to define a relational database in a "user- friendly" fashion. From a correct definition of the database, a set of utilities will be provided to create and to load the database with user supplied input data. Upon successful creation of the database, the user may, through simple primitive functions, navigate through the database performing keyed and sequential searches.

Limitations and Restrictions

The system will not support any form of maintenance, either through batch utilities or via a language. These functions are considered to be beyond the scope of this thesis.

User Inputs

The design of any system should take into account the work required by the "end-user" to effectively use the system. This was the intent when designing this system and should be recognized when reviewing the small number of files required and their easy-to-use format.

Database Definitions

The following describes the very simple format in which the user defines a database. It should be noted that the entire format is designed to be free-form.

Conventions

fieldname	-	up to 8 characters starting with a letter (required)
field length	-	up to 32K (where it makes sense) (required)
field type	-	c/C (char) default
	-	i/I (int)
	-	f/F (float)
field key	-	k/K
field variable	-	v/V

Definitions

char	-	implies character data or strings (terminated by ' 0') strings may be fixed or variable length
int	-	implies 4 byte integer
float	-	implies 8 byte double
key	-	specify whether a given field can be searched via keyed retrieval or not (applicable for only CHAR fields)
variable	-	allows you to define a type CHAR to be variable length where the user supplies the maximum length and load calc. actual length

DBD Examples

Contents of an ascii file named parts.dbd created using the VI editor. (suggested database name is parts)

```
partno          C    5    K
partname        C    25    K    V
quantity        I
price           f
```

another example

```
ssno            C    9
name            C    25
age             C    3
address         C    40    V
employer        100
```

The second of the two required input files is the users data file, which has a naming convention of "db".inp. This file is an ASCII file which has the following format.

Conventions

```
columns 1 -----> 8 field name
column  9
columns 10 ----> field value length
```

Special Field Names

----> at least one input record, indicates the end of the data for a record - must be in column 1.

Example

```
1---5----10---15---20.....

partno      vt100
partname    terminal
quantity    1
#
partno      3278
partname    terminal
#
```

User Outputs

There are really four files that are generated during a database create. They are the three component files and a log file, reporting the status of the load.

The three component files have the following naming conventions when created:

"db".dat	-	for the data file
"db".idx	-	for the index file
"db".dir	-	for the directory file

The last output file is the log file whose naming convention is "db".log. This file contains any reported problems, statistics or general comments.

System Files

Due to the nature of this project, it is expected that there will be a number of files accessed and created.

The two user files accessed are the data base definition ("db".dbd), which is an ASCII file (which can be created using the editor), and an input data file ("db".inp) of ASCII data which can be loaded into the database.

The final output created for the user is the three component files for each database in binary format. It is important that the files remain unaltered by the user, because their internal format is critical for successful usage. These three files comprise a single database and are used to resolve all queries.

The next four pages provided give an internal view of the three component files layout.

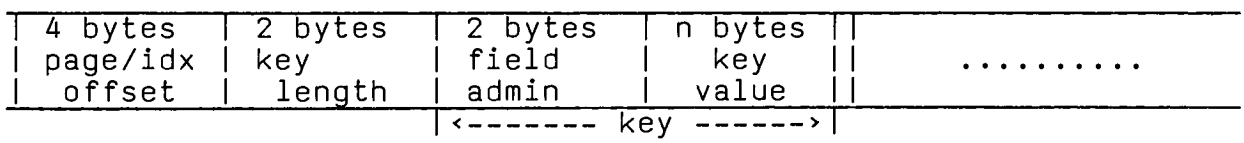
INTERNAL FILE LAYOUT

DIRECTORY FILE

Internal page layout:

6 bytes overhead / page

Internal logical bucket:

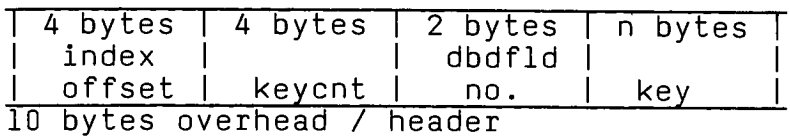


6 bytes overhead / bucket

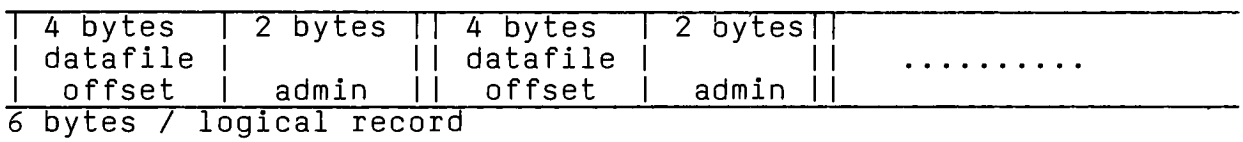
INTERNAL FILE LAYOUT

INDEX FILE

Internal header layout:



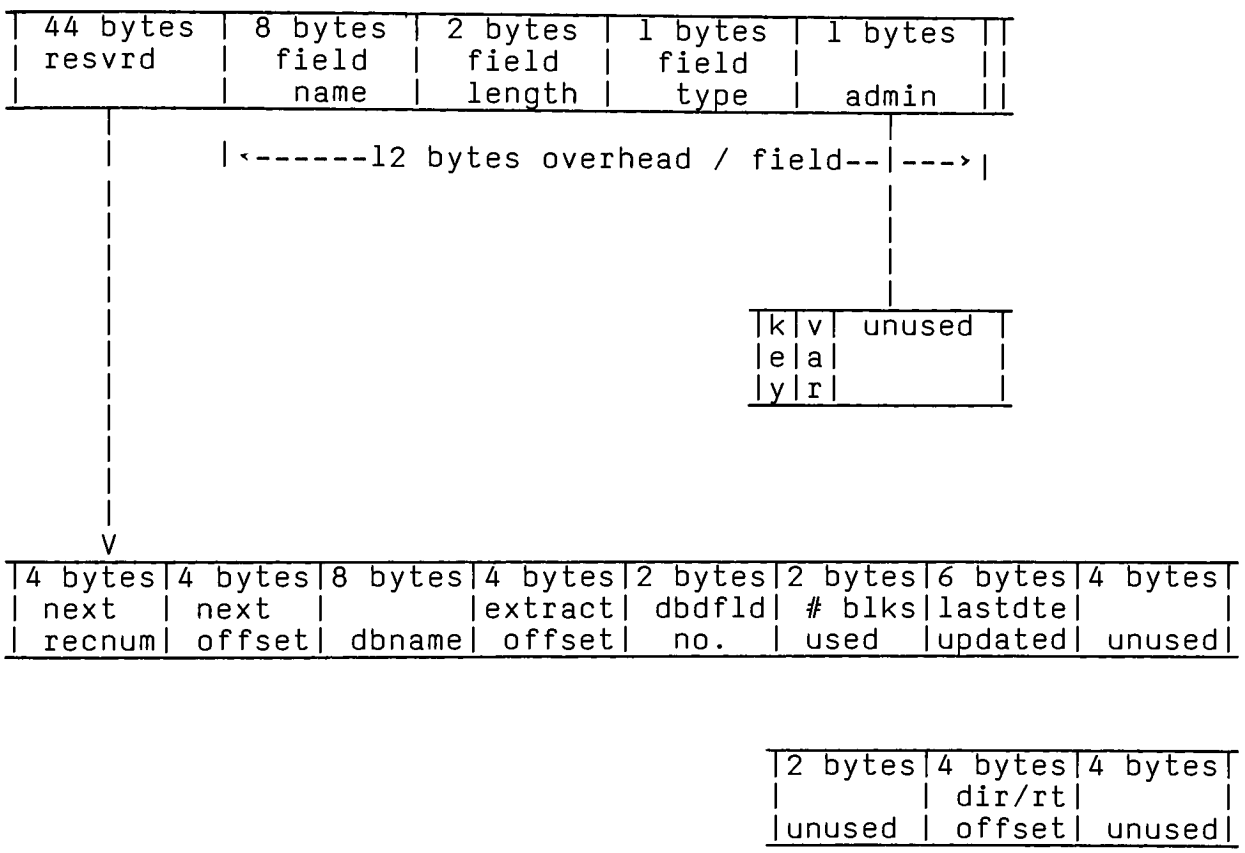
Internal index file logical record:



INTERNAL FILE LAYOUT

DATA FILE

Internal dbd layout:

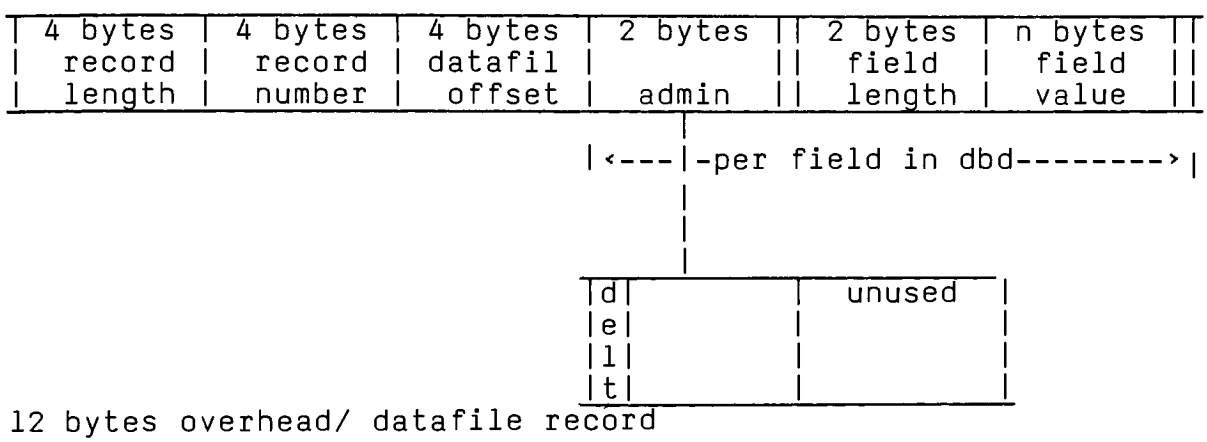


44 bytes overhead / 1024 dbd block
12 bytes unused

INTERNAL FILE LAYOUT

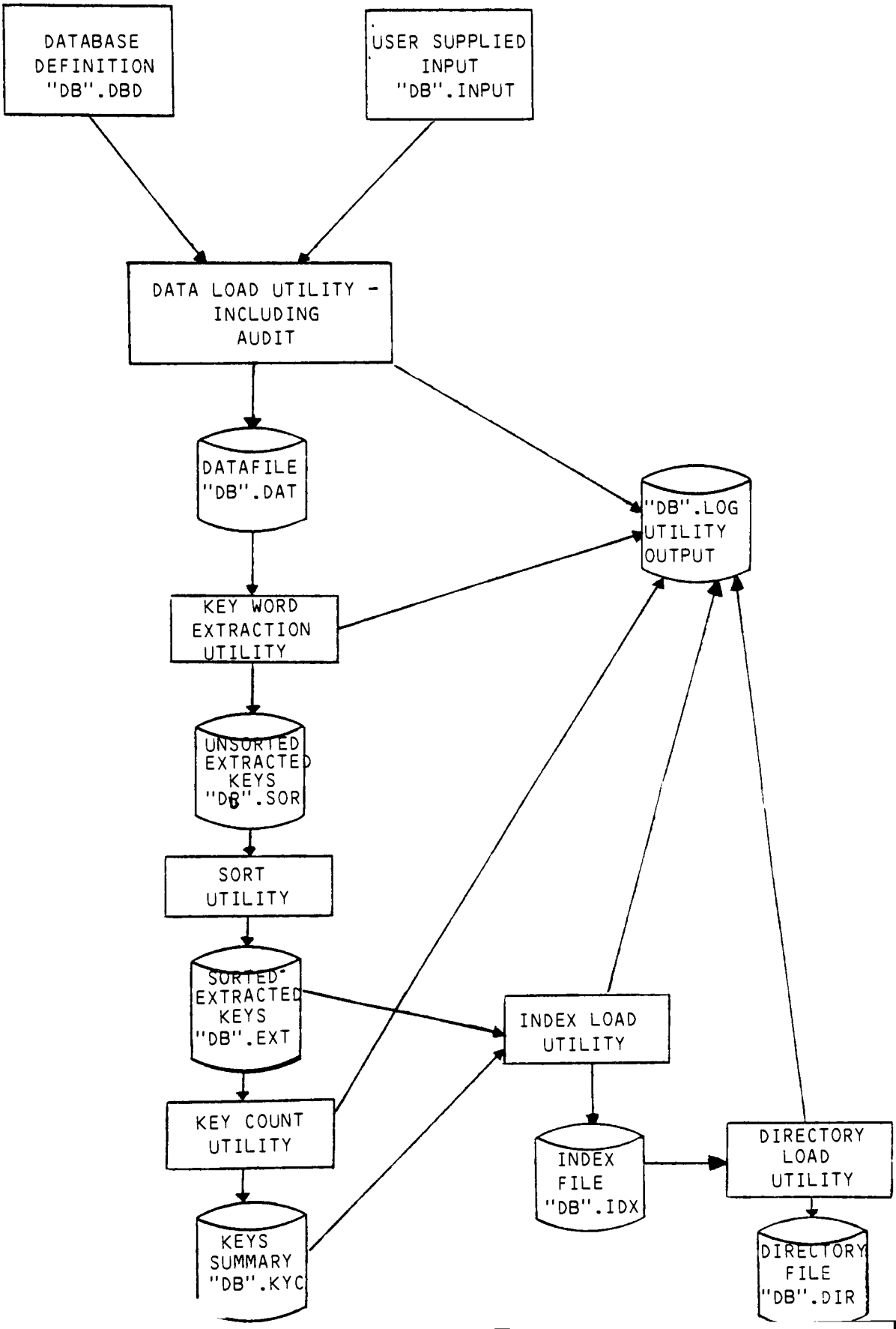
DATA FILE

Internal raw data record:



12 bytes overhead/ datafile record

SYSTEM ORGANIZATIONAL CHART

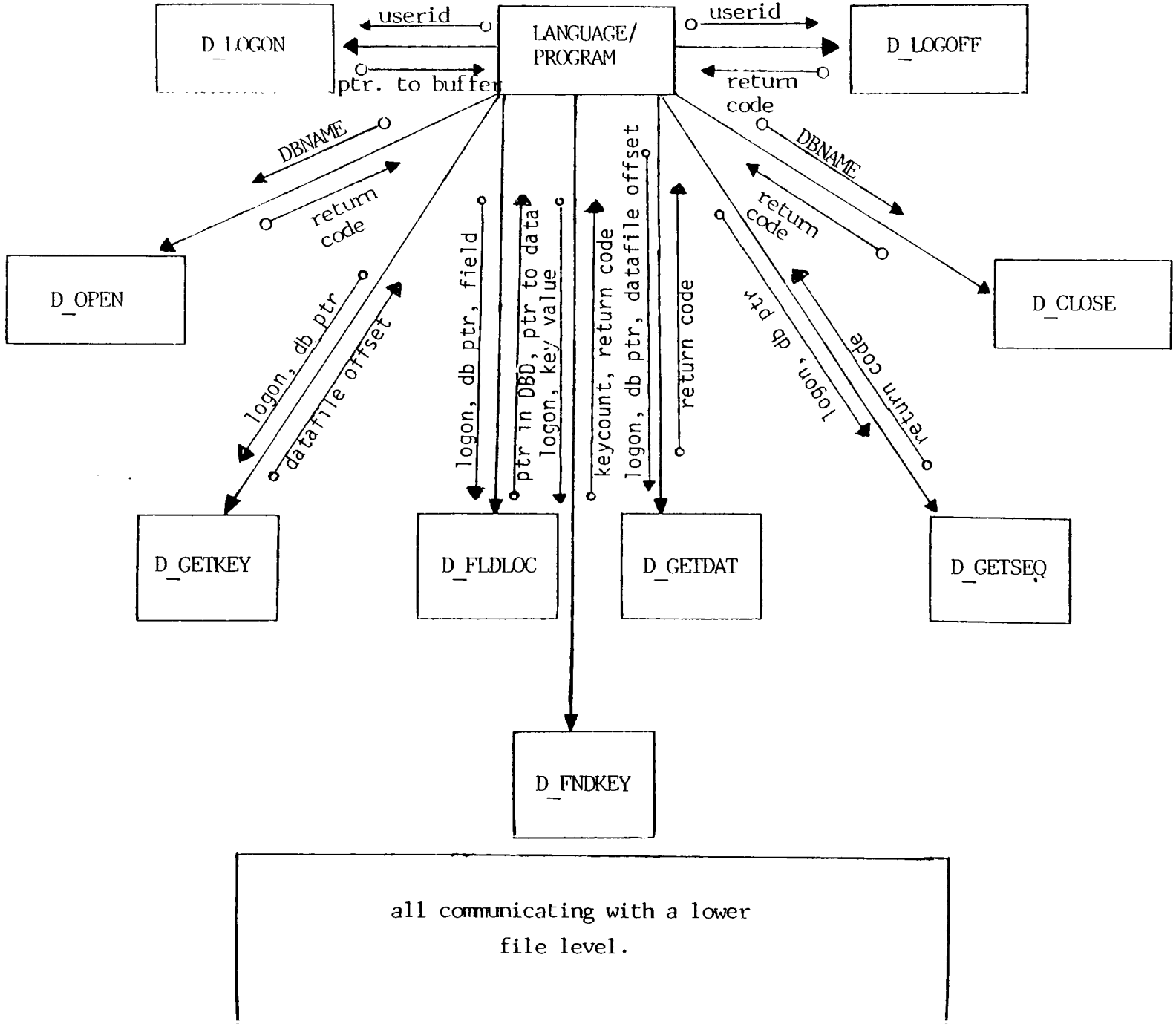


System Primitives and Data Flow Chart

The following lists the dbms primitives supplied, along with a brief description of their functionality.

Primitives Supplied

D_LOGON	-	establish a user-dbms communication
D_LOGOFF	-	remove communication and perform necessary house-keeping
D_OPEN	-	will open a given database and perform necessary internal allocations
D_CLOSE	-	will close a given database
D_GETKEY	-	given the database, get datafile offset from index file
D_FLDLOC	-	given the database, search that a particular field exists
D_FNDKEY	-	given a database and key value, return a keycount for that key - navigate through directory file
D_GETDAT	-	given the database, retrieve the actual datafile record
D_GETSEQ	-	given a database, perform a sequential search on database


D_LOGON
userid
ptr. to buffer
LANGUAGE/
PROGRAM
userid
return
code
D_LOGOFF
DBNAME
return
code
D_OPEN
logon, db ptr
datafile offset
logon, db ptr, field
ptr in DBD, ptr to data
logon, key value
keycount, return code
logon, db ptr, datafile offset
return code
DBNAME
return
code
logon, db ptr
return code
D_CLOSE
D_GETKEY
D_FLDLOC
D_GETDAT
D_GETSEQ
D_FNDKEY
all communicating with a lower
file level.

PROJECT DESCRIPTION

Implementation Plan

- Deliverable Items:

  1. data file load utility
  2. keyword extract utility
  3. index file load utility
  4. directory file load
  5. any necessary low-level file system calls to allow one to navigate through database
  6. simple database manager calls

- Milestone Schedule and Identification

  1. implement the data file load and ensure that all its internal format is correct
  2. develop the capability to extract keyed field values from the data file
  3. with the extracted keys and relative offsets from the data file generate the index file
  4. ensure the internal format of the index file is correct
  5. ensure that the two component files (index and data) can be properly navigated
  6. build from the index file a directory file
  7. write the I/O routines to navigate through all these files for both dequential retrieval as well as keyed
  8. write simple primitive database manager functions

- Milestone Schedule

  1. Most of the tasks identified must be developed in a sequential order.

  2. The anticipated and desired completion time of all the software development is in May of 1984, barring any major unforeseen problems.

Thesis Content

The following major headings describe the topics to be discussed in each chapter of the thesis.

1. Considerations involved when designing the file structure.

   - This will describe the reason for choosing the proposed file structure, along with its tradeoffs under the UNIX Operating System.

2. Implementation evolution

   - A discussion pertaining to the evolution of the implementation, describing any changes or roadblocks.

3. Locking and related issues surrounding the file access method.

   - A detailed discussion of how locking and concurrency control can be implemented under the UNIX Operating System.

4. The integration of the file access method within the Kernel.

   - My views of how the file access method could be integrated into the Operating Systems Kernel, along with its tradeoffs.

5. Conclusion

   - Tie up all the loose ends and discuss the overall outcome of the thesis exercise.

Implementation Tools

I decided to write the system in the programming language "C" for a number of reasons. The major reason being the power, versatility and speed that the language naturally provides when running under Unix.

The only additional tools that will be used and needed are the standard Unix debuggers (SDB, DBX) and other Unix utilities.

QUALIFICATIONS

Personal Background

I have worked in or around the database industry for approximately six years with four different employers. With each employer, the type of work was much different and very rewarding from a learning point of view.

Employment Experience

Eastman Kodak Company	--	group leader of on-line information system group. Primarily designed and implemented application software around a relational-like dbms called INQUIRE by Infodata. (3.5 years)
Infodata Systems, Inc.	--	was the northeast regional technical database consultant. Worked in many different environments, in may different capacities providing technical database assistance, advice, and support. (1.5 years)
Computer Consoles, Inc.	--	was in charge of an area which designs and implements internal dbms software for a "network" database package marketed by Computer Consoles, Inc.
Applix, Inc.	--	currently developing a proprietary relational database component for the Alis™ product.

QUALIFICATIONS (continued)

COURSES TAKEN

My educational background is as follows:

1. A.A.S. - Chemistry (MVCC) (G.P.A. 3.00)
2. B.A. - Chemistry/Computer Science (SUNY at POTSDAM)
3. M.S. - Computer Science (currently completing at RIT) (G.P.A. 3.5)

The following are the courses which I have taken as both an undergraduate and graduate organized by subject matter.

Undergraduate	Graduate
1. Computer Science I (BASIC)	Computer Architecture
2. Fortran	Foundations of Computing Theory
3. Intro. to Problem Solving (pl/c)	Software Architecture
4. APL	Programming Language Theory
5. COBOL	Computer Graphics I
6. Assembler (IBM)	Oatabase Implementation
7. Discrete and Data Structures (PL/1)	Database Concepts
	Online Information Systems Design
8. Programming Structures (PL/1)	Online Information Stor. & Retr.
9. Data Communications	Microprocessors and Minicomputers
10. Symbolic Logic	Operating Systems
11. Along with 43 hours of Chemistry, 1 year each of Calculus, Physics, Biology	

BIBLIOGRAPHY

1. Babad, J., "A Record and File Partioning Model", University of Chicago, Communication of the ACM, January 1977, Volume 20, No. 1.

2. Chen, Peter P., "Design and Performance Tools for Data Base Systems", Sloan School of Management, Massachusetts Institute of Technology, 1977

3. Cardenas, A., "Analysis and Performance of Inverted Data Base Structures", Communications of the ACM, Volume 18, No. 4, May 1975.

4. Codd, E. F., "A Relational Model of Data for Large Shared Data Banks", Communications of the ACM, Volume 13, No. 6, June 1970.

5. Date, C. J., An Introduction to Database Systems, Hayden Book Company, Inc., 1974.

6. Hawthorne, P., Stonebraker, M., "The Use of Technological Advances to Enhance Data Management System Performance", College of Engineering, University of California, Berkeley, January 15, 1979.

7. Hutt, A.T.F., A Relational Database Management System, John Wiley and Sons, 1979.

8. McKusick, M., Joy, J., Lefflert, S., and Fabry, R., "A Fast File System for UNIX", University of California, Berkeley, 1983.

9. Reilly, Paul, "Tailoring UNIX for the Supermicro Business Environment (UNIX DBMS)", Mini-Micro Systems, November 1983.

10. Siler, K., "A Stochastic Evaluation Model for Database Drganizations in Data Retrieval Systems", University of California, Los Angeles, 1976.

11. Ullman, J., Principles of Data Base Systems, Computer Science Press, 1980.

GRADING CRITERIA

The following percentages applied to the major categories, represent my input as how I feel the grading criteria should be applied.

Theoretical Investigation: 20%

System Design: 25%

Correctness of Implementation: 15%

Demonstratable Quality: 5%

Operating Efficiency: 25%

Goals Met Within Deliverable Time-Frame: 10%

VI.2 Bibliography

1. Astrahan, M., Blasgen, M., Chamberlin, D., Eswaran, K., Gray, J., Griffiths, P., King, W., Lorie, R., McJones, P., Mehl, J., Putzolu, G., Traiger, I., Wade, W., Watson, V., "System F: Relational Approach to Database Management", Transactions on Data Base Systems, Volume 1, No. 6, June 1976.

2. Babad, J., "A Record and File Partioning Model", University of Chicago, Communication of the ACM, January 1977, Volume 20, No. 1.

3. Bayer, R., and Schkolnick, M., "Concurrency of Operations on B-Trees", Acta Informatica 9, 1977.

4. Bayer, R. and Unterauer, K., "Prefix B-Trees", ACM Transactions on Database Systems, Volume 2, No. 1, March 1977.

5. Blair, D., "The Data Document Distinction in Formation Retrieval", Communications of the ACM, Volume 27, No. 4, April 1984.

6. Cardenas, A., "Analysis and Performance of Inverted Data Base Structures", Communications of the ACM, Volume 18, No. 4, May 1975.

7. Carlson, E., Sorenson, P., "Data Base Techniques and Models of the Office Environment", Data Base, Volume 15, No. 2, June 1970.

8. Codd, E. F., "A Relational Model of Data for Large Shared Data Banks", Communications of the ACM, Volume 13, No. 6, June 1970.

9. Comer, D., "The Ubiquitous B-Tree", ACM Computing Surveys, Volume 11, 1979.

10. Connet, G., "Unstructured Data Bases or Very Efficient Text Searching", ACM, August, 1983.

11. Croft, B., "A File Organization for Cluster-Based Retrieval", Unpublished Ph.D. dissertation, University of Cambridge.

12. Culik, O. K., Th. and Wood, D., "Dense Multiway Trees", ACM Transactions on Database Systems, Volume 6, No. 3, September 1981.

13. Date, C. J., An Introduction to Database Systems, Hayden Book Company, Inc., 1974.

Bibliography (Cont'd)

14. Eastman, C., "A Tree Algorithm for Nearest Neighbor Searching in Document Retrieval Systems", University of North Carolina at Chapel Hill, Unpublished Ph.D. disseration, 1977.

15. Eastman, C.M., "Current Practice in the Evaluation of Multi-key Search Algorithms", Southern Methodist University, ACM SIGIR, Volume 17, No. 4, 1983.

16. Hill, E., "Analysis of An Inverted Data Base Structure", Laboratory of Applied Studies.

17. Horowitz, E., and Sahni, S., Fundamentals of Data Structures, Chapter 10, Computer Science Press, Inc., 1977.

18. Hutt, A.T.F., A Relational Database Management System, John Wiley and Sons, 1979.

19. IBM Corp., "OS ISAM Logic", GY28-6618, White Plains, N.Y., 1966.

20. IBM Corp., "OS/VS Virtual Storage Access Method Logic", Armonk, N.Y., SY26-3841.

21. King, R., Korth, H., and Willner, B., "Design of a Document Filing and Retrieval System", Data Base, Volume 15, No. 2, 1984.

22. Kwong, Y.S., and Wood, D., "A New Method for Concurrency in B-Trees", IEEE Transactions on Software Engineering, Volume SE-8, No. 3, May 1982.

23. Lamport, L., "Concurrent Reading and Writing", Communications of the ACM, November 1877, Volume 20, No. 11.

24. Larson, J., "Bridging the Gap Between Network and Relational Database Management Systems", Computer, Volume 16, No. 9, September 1983.

25. Larson, P., "A Method For Speeding Up Text Retrieval", Data Base, Volume 15, No. 2, 1984.

26. McKusick, M., Joy, J., Lefflert, S., and Fabry, R., "A Fast File System for UNIX", University of California, Berkeley, 1983.

27. Salton, F., Fox, E., and Wu, H., "Extended Boolean Information Retrieval", Communications of the ACM, December 1983, Volume 26, No. 12.

28. Siler, K., "A Stochastic Evaluation Model for Database Organizations in Data Retrieval Systems", University of California, Los Angeles, 1976.

Bibliography (Cont'd)

29. Stonebraker, M., "A Distributed Data Base Version of Ingres", Berkeley Workshop on Distributed Data Management and Computer Networks, 1977.

30. Stonebraker, M., Stettner, H., Lynn, N., Kalash, J., and Guttman, S., "Document Processing in a Relational Database System", ACM Transactions on Office Information Systems, Volume 1, No. 2, April 1983.

31. Stonebraker, M., Wond, E., Kreps, P., and Held, G., "The Design and Implementation of INGRES", ACM Transactions on Database Systems, Volume 1, No. 3, September 1976.

32. Writh, N., Algorithms and Data Structures, 1976.

33. Ullman, J., Principles of Data Base Systems, Computer Science Press, 1980.

Program Listings

DATALOD

This utility provides a mechanism by which the user can load/update the datafile component of a given database. The utility using run-time parameters to augment the performance and ease of this process.

PARAMETERS (order not important)

db= (database name)

mode= (load)

REQUIRED FILES:

INPUT

"db".dbd -- this file is nothing more than an ASCII file which correctly describes the database definition (DBD. The utility audits the dbd for validity. (See documentation on format.)

OUTPUT

"db".dat -- the specified datafile is created.

LOAD MODE:

This mode implies the user does not have a data file component that already exists. If a data file does not exist, the utility will fail. It creates and builds a new data file component for the selected database.

DATA BASE DEFINITION

The following describes the way in which a user may define a database using their standard editor. It was meant to make this task very easy to use, so nearly the entire format is free-form. The two required pieces of information on each line in the file is a field name and a length. (The default type is character.)

CONVENTIONS

fieldname - up to 8 characters starting with an alphabet (required)

field length - up to 32k (where it makes sense) (required)

field type - c/c (char)
i/I (int)
f/F (float)
s/S (short)
t/T (text) (not implemented)

field key - k/K

field variable - v/V

field repeated - r/R value (maximum no. of repeats)

DEFINITIONS

CHAR -- implies character data or strings (terminating by '§0') - strings may be fixed or variable length.

INT -- implies 4 byte integer

FLOAT -- implies 8 byte double integer

SHORT -- implies 2 byte integer

REPEATED -- allows multiple occurrences of a given field applicable for all of the above types -- must specify maximum number of repeats, not implemented.

VARIABLE -- allows you to define a type CHAR to be variable length where the user supplies the MAXIMUM length it may be and the DATALOD utility actually calculates its variable length.

KEY -- specify whether a given field can be searched via keyed retrieval or not (applicable for type CHAR fields only). Implies index sequential retrieval with that field as opposed to sequential.

TEXT -- this specifies that there is type CHAR which is to be used for text processing (proximity searching). This implies that the entire database is setup with extra overhead in the index file to allow proximity searching.
(** important ** this field cannot be keyed (k), overlay (o), int (i), short (s), float (f)), not implemented.

DBD EXAMPLES

contents of an ASCII file named parts.dbd created using the VI editor. (suggests database name is parts)

```
partno c 5 k
partname c 25 k v
quantity i
supplier c 60 v
price f
```

another example

```
ssno c 9
name c 25
lastname 10 v
first name 10 k
address c 25 v
employer 100
```

DATALOD INPUT FILE FORMAT

The following describes the very simple format in which the input file must be in to be usable by the utility ("db".inp).

CONVENTIONS

```
columns 1 --> 8   field name
column  9         blank
columns 10 -->    field value length
```

SPECIAL FIELD NAMES

+ ---> following a previous valid field name indicates repeated information - must be in column 1

---> following at least one input record, indicates the end of the data for that given record -- must be in column 1

orientation with a file by columns

```
1---5----10---15---20 .......

partno 100
partname terminal
supplier DEC
+        QUME
#
supplier IBM
partno   3278
partname terminal
price    2000.0
#
```

The order of the input within a given record is unimportant and not all fields are required.

IDXLDAD

This utility provides a mechanism for building or updating the index file in a batch mode. The utility uses run-time parameters to augment the performance and ease of processing.

PARAMETERS (order unimportant)

db= (database name)

mode= (load)

REQUIRED FILES

INPUT

"db".dat -- this is the previously loaded data file which is used for key extraction

OUTPUT

"db".idx -- the associated index file is created

"db".ext -- temporary intermediate file which holds all keys extracted

"db".kyc -- temporary intermediate file which holds values of number of unique keys extracted

LOAD MODE:

This mode implies that the user does not have an index file component that already exists. If an index file does exist, the utility will fail. It creates and builds a new file component for the selected database.

DIRLOAD

This utility provides the mechanism for building a tree structure, where the densest levels contain one entry for each unique key. The levels above it contain the high keys from each page to allow rapid searching down to the lowest level.

PARAMETERS (order unimportant)

db= (database name)

mode= (load)

REQUIRED FILES

INPUT

"db".idx -- this is the previously loaded data file which contains all offsets into datafile

OUTPUT

"db".dir -- the directory file which is a tree structure used to rapidly locate a given key

LOAD MODE:

This mode implies that the user does not already have a directory file component that exists. If directory file does exist, the utility will fail.