Copyright © 1982 American Telephone and Telegraph Company
THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 61, No. 9, November 1982
Printed in U.S.A.

Database Systems:

Structure of a Database File System for the *UNIX* Operating System

By M. J. ROCHKIND

(Manuscript received October 20, 1982)

A database file system is the low-level part of a database management system that handles data storage, indexing, concurrency control, transaction control, and recovery. This article describes a structure for a database file system that encapsulates important design decisions into separate modules. This permits a choice of implementations for each module. A database file system can be generated for a particular application by choosing the appropriate implementation for each module from a catalog (one from column A, one from column B, etc.). This approach can be both more flexible and more efficient than the usual approach of building a monolithic database file system to satisfy everyone. The challenge is to partition the database file system functions into modules that hide enough information to allow flexibility, but not so much as to be inefficient. The structure described here, which is biased towards information hiding, is an initial attempt at a suitable partitioning. An experimental system, XFS, *was built to test this structure.*

I. INTRODUCTION

This article describes the structure of a database file system for the *UNIX** operating system. A database file system, as the term is used here, handles these database functions:

- Ensuring that transactions are *atomic*: that they are either done in their entirety or not at all.

* Trademark of Bell Laboratories.

- Ensuring that transactions are *permanent*: that their effects are not lost, even if the hardware fails.
- Ensuring that transactions are *serializable*: that concurrently run transactions have the same effect as some sequence of the same transactions run one at a time.
- *Storing*, *updating*, and *deleting* arbitrary data records.
- *Indexing* data records (there may be more than one key per record, and more than one record per key).

A database file system does *not* perform these database functions:

- Enforcing a user-defined data model, specified by a schema.
- Providing a high-level query language.
- Automatically generating keys and updating the appropriate indexes as records are stored, updated, and deleted.
- Automatically maintaining relationships between records.

To summarize the above lists: a database file system provides record storage, access methods, recovery, and concurrency control. It is a low-level basis onto which a database management system can be built. It can support any of the traditional data models: hierarchical, network or relational.

My reasons for focusing on the low-level end of the database problem, rather than the (probably more popular) high-level end, are these:

(*i*) The functions (user interface) are not controversial. While people might disagree on whether a function should be called "store," "insert," or "install," the passion with which they argue is mild compared to the viciousness with which the hierarchial, network, and relational advocates argue.

(*ii*) We know how to build a database file system. The challenges are engineering ones (how to package it, how to tune it, etc.). We don't know how to build a "complete" database system without producing a monster or a toy.[1] Furthermore, even if we did know how to build one, the market would be small (because most users wouldn't like our model) and it would take too long to build.

(*iii*) Many applications only need the database file system. Schemas, query languages and automatic indexing would be overkill.

(*iv*) We have to start somewhere—why not at the bottom so we get something useful fairly early?

II. ALL THINGS TO ALL PEOPLE?

We can't build a database file system that satisfies every need, or even most needs. But we can design a structure for a database file system that satisfies most needs. This situation is analogous to that with compilers: there are many compilers, even many for the same language. But nearly all of them have the same structure: lexical analyzer, parser, code generator, optimizer, etc. In fact, the progress in compiler writing over the last 15 years is in large part due to the establishment of a standard structure for compilers.

It would be premature to propose a standard structure for database file systems, but I am proposing a structure that can handle a reasonably wide range of applications, from tiny, personal databases (e.g., a mailing list) to large, serious, operation-support systems, such as those developed by Bell Laboratories for automating telephone company operations. This structure divides the problem into modules, each of which hides design decisions.[2] For example, the recovery module hides the decision to use a redo log or not. Since this decision is hidden, the module may be implemented in two ways: with a redo log and without. Those users who need a redo log and are willing to pay for it can have one, and those users who don't need it don't pay for it.

So the goal of my design is to provide a framework onto which multiply-implemented modules may be hung. To configure a specific database file system, the user picks one from column A, one from column B, etc. If the structure is right, evolution will eventually result in a wide spectrum of high-quality implementations for each module.

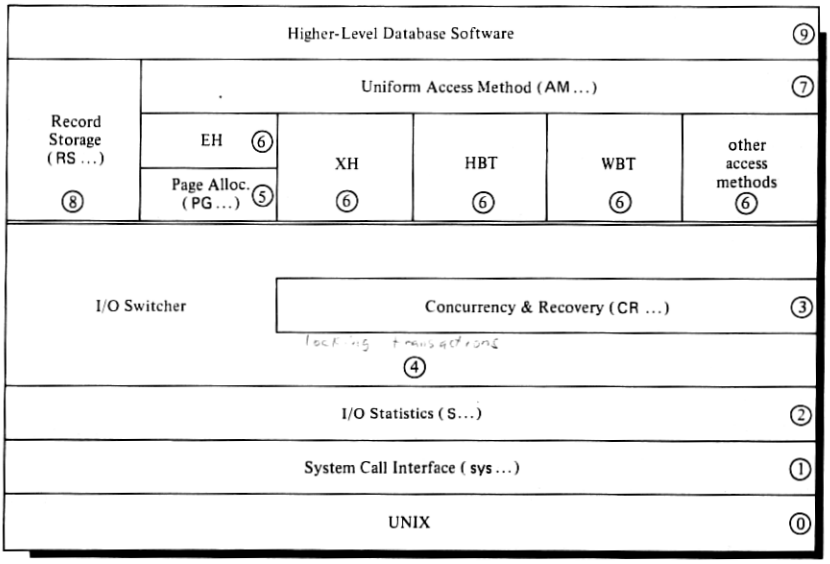
To get the structure right, a series of experimental systems should be built. I have already built the first system, consisting of a single implementation for most modules. I will describe here the structure of this system, called XFS. I encourage the reader to focus on the structure of XFS rather than its implementation, which is admittedly inadequate for most purposes.

In fact, I invite the reader to try to imagine multiple implementations of each module. If these multiple implementations cover the range of users' needs, and if the result is esthetically pleasing and efficient, then the structure is a success, and additional "real" implementations should be developed. If the structure needs modification, then it can be modified easily (assuming we know what we want!), since the investment in implementation is minimal.

Figure 1 illustrates the structure of XFS. The interface to level 4, as marked by the double line, is the same as the standard interface to the *UNIX* file system (open, read, write, etc.). Software above the double line may be written without knowing whether it will run directly on the *UNIX* operating system or on top of software that handles concurrency and recovery. This is a key feature of this structure.

The following sections describe each level of abstraction, from lowest to highest. The encircled numbers in Fig. 1 are the levels. The reader may want to skim the entire article before reading thoroughly, since when reading about the lower levels it may be useful to know where we are headed.

This structure is by no means perfected. The text mentions some unresolved problems that I knew about when I designed this structure. I urge the reader to try to find the ones I didn't know about.



Higher-Level Database Software 9
Uniform Access Method (AM ...) 7
Record Storage (RS ...) 8
EH 6
Page Alloc. (PG ...) 5
XH 6
HBT 6
WBT 6
other access methods 6
I/O Switcher
Concurrency & Recovery (CR ...) 3
locking transactions
4
I/O Statistics (S...) 2
System Call Interface (sys ...) 1
UNIX 0

Fig. 1—Structure of experimental system.

III. LEVEL 0: *UNIX* FILE SYSTEM

This level is the file system, as described in Section II of the *UNIX System User's Manual*.[3] The operating system is entered by executing a sys instruction, which can be done only in assembly language.

IV. LEVEL 1: C SYSTEM CALL INTERFACES

4.1 Level 1a: Standard C Interfaces

For each system call there is a small C interface written in assembler. We are interested in these interfaces:

```
int open(path, oflag)
char *path;
int oflag;

int creat(path, mode)
char *path;
int mode;

int close(fildes)
int fildes;

int read(fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;
```

```
int write(fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;

long lseek(fildes, offset, whence)
int fildes;
long offset;
int whence;
```

4.2 Level 1b: C Interfaces, Alternate Names

This module is the same as 1a, except the names of the entry points are prefixed with sys: sysopen, syscreat, etc. This allows the unprefixed names to be used in higher levels (2a and 4). To produce the code for this module, I simply edited the assembly language functions of module 1a to change the entry point names.

In XFS, 1b is used instead of 1a.

V. LEVEL 2:I/O STATISTICS

5.1 Level 2a: I/O Statistics, Standard Names

This module consists of small C functions that have the same names as the functions in level 1a: open, creat, etc. Each function gathers statistics about activity on each file and then calls the corresponding sys... function. An additional function prints the statistics and zeros the counters:

```
void iostat( )
```

Statistics can be gathered on any existing *UNIX* system program by inserting a call to iostat and reloading it with the functions from modes 1b and 2a.

Figure 2 shows some sample output from iostat. This I/O statistics facility is useful in its own right, and is available as a package separate from XFS. I have used it extensively to evaluate the access methods described in Section IX.

5.2 Level 2b: I/O Statistics, Alternate Names

This is the same as 2a, but with an S prefixed to each function name (except for iostat). This allows the standard names (open, creat, etc.) to be used at level 4. XFS uses module 2b.

VI. LEVEL 3: CONCURRENCY CONTROL AND RECOVERY

This module (CR) provides the notion of transactions, and contains algorithms that ensure the properties of atomicity, permanence, and

```
FILE             OPENS   READS   nonbuf   WRITES   nonbuf   SEEKS
stdin              1       9       9        0        0        0
stdout             1       0       0      1409     1409       0
stderr             1       0       0        0        0        0
w.F                2       0       0        0        0        0
w.T                2       1       0        1        0        2
TMPbaa014870       1       0       0        1        1        0
TMPcaa014870       1       6       6        3        3       11
```

Fig. 2—Sample **iostat** output.

serializability. I/O is in units of pages, which are referenced by page numbers.

6.1 Specification

These are the interfaces to module CR:

```
#include <stdio.h>
#include "cram.h"

void CRopen(dbname)
char *dbname;

FNUM CRfopen(fname)
char *fname;

XNUM CRbegin ( )

void CRcommit(x)
XNUM x;

void CRabort(x)
XNUM x;

void CRwrite(x, f, p, ptr)
XNUM x;
FNUM f;
PNUM p;
PAGE *ptr;

void CRread(x, f, p, ptr)
XNUM x;
FNUM f;
PNUM p;
PAGE *ptr;
```

The function CRopen must be the first function called. It opens the database named by its argument (only one database may be open at a time).

CRbegin starts a transaction and returns a transaction number (XNUM) which must be used in subsequent calls on behalf of this transaction. A single process may have several simultaneous transactions (with different XNUMs, of course). If CRcommit is called, then when it returns the transaction is guaranteed to be permanent. If CRabort is called, then all traces of the transaction are erased as though it had never started in the first place. After a call to CRcommit or CRabort, the XNUM is no longer valid.

If transaction number 0 is used instead of an XNUM obtained via CRbegin, the properties of atomicity, permanence and serializability will not be ensured for transaction 0 or for any other concurrently running transaction. In other words, there is no logging or locking for transaction 0. This feature is used by level 5 (page allocation).

Files are opened by calling CRfopen. The FNUM that is returned is used in I/O operations on that file. (It is analogous to a *UNIX* system file descriptor). CRfopen automatically creates files that are nonexistent. Each file consists of an implementation-defined number of pages of sizeof(PAGE) bytes. Any page may be read or written; reading a page that has never been written returns a page of all zeros. In XFS, each file has 2048 512-byte pages.

Pages are numbered consecutively starting with 0. The typedef PNUM is used for page numbers, but PNUMs behave like integers and may be operated on accordingly.

CRwrite writes the data pointed to by ptr to the page p in file f on behalf of transaction x. Similarly, CRread reads a page.

I am not sure how much information about the implementation of serializability should be part of the externals of module CR. On the one hand, I want to hide as much as I can to allow alternative implementations, but on the other hand, these functions can't be used efficiently unless the user knows what various combinations of calls will do to concurrency. So the following is an initial attempt.

The granularity of locking can be as small as a page and as large as the entire database. There are two kinds of locks: share and exclusive. A granule can be share locked if no other transaction has it exclusively locked. A granule can be exclusively locked if no other transaction has it locked (share or exclusive). All locks are held until the transaction terminates (via CRcommit or CRabort).

Calling CRread puts a share lock on the granule that contains the page being read. Calling CRwrite puts an exclusive lock on the granule that contains the page being written. For example, if the granule is a page, then CRread share locks a page and CRwrite exclusively locks a page (this is in fact what XFS does).

Since share and exclusive locking is automatic with reads and writes, transactions are guaranteed to be consistent in the sense of Ref. 4.

That is:

- They are serializable.
- Each transaction sees a consistent state of the database.
- Undoing a transaction loses no updates of completed transactions.
- Undoing a transaction produces a consistent state.

This is the strongest degree of consistency, and I do not think that lowering the degree of consistency (e.g., not holding share locks to transaction termination) to gain concurrency is a good idea. My main reason is that lower degrees of consistency can cause unreproducible errors, making debugging almost impossible. On the other hand, there are some applications where read-only, statistical reports must be run concurrently with other more critical transactions. Here it makes sense to let the read-only transaction run at a lower degree of consistency. My conservative design would be too inefficient for these applications (but the design is not hard to change).

CRread and CRwrite wait until the appropriate lock is available. Deadlock can occur, so an implementation of module CR must be able to detect and resolve deadlock. Deadlock is resolved by choosing a transaction as the victim and murdering it with CRabort. The trick is picking the best victim.

6.2 XFS Implementation

Module CR is implemented in XFS as follows: The standard *UNIX* operating system call names are used for I/O (open, creat, etc.). These system calls could be supplied by level 1a, but, as the next section explains, they are actually supplied by level 4.

In XFS the property of permanence is not ensured; that is, if the files containing permanent data are corrupted, then the entire database has to be restored from the last backup tape maintained by the system administrator. All transactions committed after the backup would be lost. Because losing transactions is *not* one of the implementation options for module CR, XFS must be viewed as unfaithful to the specifications with regard to this property.

A real implementation of module CR could use a "redo" log to ensure permanence.[2] With this approach all writes are written to the database and to the log (on tape or disk). If permanent data are lost, the affected files are restored from the last backup and the log is used to bring them up to date. (This step can take hours, but it is rare). If the log is lost, a new backup is made immediately. The redo-log technique is used by many commercial database management systems (e.g., IBM's IMS).

In XFS serializability is ensured by share and exclusive locks at page granularity, as described above. The lock manager is functionally correct but inefficiently implemented. Deadlock detection and resolution is particularly naive: after trying three times to obtain a lock, sleeping for a few seconds between tries, the transaction requesting the lock is murdered.

In XFS atomicity is ensured by an "undo" log. Before any page is written, the old data is logged. If the transaction aborts, the undo log is used to restore the database by rewriting the old pages. This must also be done after a crash, since that also causes transactions to be aborted. In fact, whenever CRopen is called, all active transactions are aborted before the process invoking CRopen is allowed to proceed (these transactions must have resulted from a previous life that ended in a crash).

The XFS implementation of the undo log is inefficient in time and space. Logged pages are appended to a file, and the space occupied by unneeded pages is not reclaimed. After a while the log file gets too large. This turns out to be inconvenient even in an experimental system, because some of my experiments used too much "tape." A real system could solve this problem in any number of ways, so I am not too worried about this problem with XFS.

The property of atomicity can also be achieved by differential files,[5] shadow pages,[6] and other techniques. However, some of these techniques are not easily adapted to a situation where multiple transactions are active simultaneously, since there must be a way to undo one transaction without disturbing the other transactions.

VII. LEVEL 4: I/O SWITCHER

The purpose of this module (the "switcher") is to hide the difference between ordinary system files and files controlled by module CR (level 3). The interface to the switcher is the same as the standard *UNIX* system calls (open, creat, etc.). A program written to use these system calls does not know whether it is using the facilities of levels 2 and 3 or running directly on level 1. This has three advantages:

(*i*) Programmers do not have to learn another file system interface.

(*ii*) Many programs written for the *UNIX* operating system, using the system calls directly or using the standard I/O library (fopen, getc, etc.), can benefit from concurrency control and recovery simply by running them on top of levels 2 and 3. Hence, many existing programs can be used in database environments without change.

(*iii*) Higher-level database software (access methods, retrieval algorithms, etc.) can be debugged and tested without the overhead of concurrency control and recovery. This cuts down on loading time, program size, files, file descriptors, etc.

The switcher works as follows: by default, operations (open, read, write, etc.) are routed directly to level 2b, so that the effect of any system call is exactly the same as if the switcher were not involved. Alternatively, a file may be registered with the switcher by calling freg:

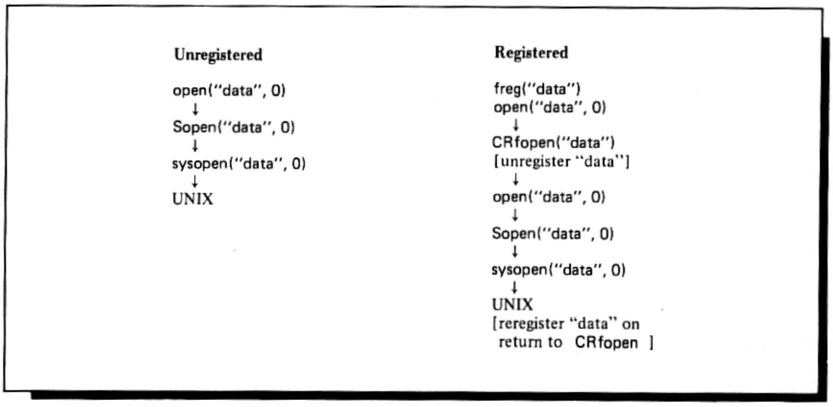
```
void freg(path)
char *path;
```

Once a file is registered, operations on it are routed to module CR instead of 2b. Ultimately, module CR will also need to perform I/O operations (e.g., a call to CRread may cause a call to read). Since there is only one entry point for each system call, module CR will make a recursive call on the switcher. To break the recursion, the file is temporarily unregistered, so that the operation can be switched directly to module 2b. Figure 3 illustrates the sequence of calls.

For registered files, the behavior of I/O routed through module CR is not exactly like that of unregistered files, because module CR operates in units of complete pages. In fact, on registered files, calling read or write with a byte count other than 512 causes a fatal error. If the standard I/O library is used with buffering, most of the I/O is in units of pages, but the last block of the file can be short. Another related problem is that module CR does not keep track of the file size (all files have a fixed number of pages), so there is no notion of an end-of-file.

I know of no clean solution to these problems. Therefore, the transparency provided by the switcher is only usable with "database" software, which usually deals entirely with complete pages. There is still enough software in this category to make the switcher worthwhile.

Since the switcher hides the information as to whether module CR is engaged, programmers working at higher levels of abstraction (level 5 and above) may ignore concurrency and recovery issues. They may



```
Unregistered                         Registered

open("data", 0)                      freg("data")
   ↓                                 open("data", 0)
Sopen("data", 0)                        ↓
   ↓                                 CRfopen("data")
sysopen("data", 0)                   [unregister "data"]
   ↓                                    ↓
UNIX                                 open("data", 0)
                                        ↓
                                     Sopen("data", 0)
                                        ↓
                                     sysopen("data", 0)
                                        ↓
                                     UNIX
                                     [reregister "data" on
                                      return to CRfopen ]
```

Fig. 3—Call sequence.

read and write arbitrary pages of arbitrary files willy-nilly, and still be assured that their transactions will be atomic, permanent, and serializable. This is a nice state of affairs, because higher-level database software is complicated enough without complicating it further with concurrency and recovery considerations. In addition, the programs implementing the concurrency and recovery algorithms are made easier to write, debug, and test because little functionality is supported—just the ability to read and write pages. Since these programs have so little functionality, they will require less modification than the rest of the system as maintenance, enhancement, tuning, and adaptation take place over time. Once debugged, the concurrency and recovery software is therefore likely to stay that way.

VIII. LEVEL 5: PAGE ALLOCATION

8.1 Specification

This module provides a mechanism to keep track of allocated and free pages. Its use is entirely optional—some algorithms need such a facility and some don't. The notions of "allocated" and "free" are imaginary anyway, since every file has a fixed number of pages at all times. This is the interface:

```
PNUM PGalloc(x, f)
XNUM x;
FNUM f;

void PGfree(x, f, p)
XNUM x;
FNUM f;
PNUM p;
```

PGalloc allocates a page on behalf of transaction x in the file, indicated by f. PGfree frees a page previously allocated. A freed page can be reallocated.

8.2 XFS Implementation

In XFS, the first page of a file on which PGalloc and PGfree are to be used is a bit map that indicates whether each page is allocated or free. Since writing a page automatically sets an exclusive lock, and since exclusive locks are held until a transaction terminates, the use of PGalloc would ordinarily increase the locking granularity from a page to a file. To avoid this, PGalloc does not make use of its XNUM argument, but instead reads and writes the bit map as transaction 0. Writing a single page is inherently atomic, so a partially written bit map cannot occur. The only disadvantage of suppressing recovery is that if a transaction aborts, its allocated pages will not be freed. An offline utility (not implemented) could restore these lost pages to the free list.

The same game cannot be played with PGfree, since if a transaction aborts after freeing a page, the bit map must be restored to show the page as allocated. So PGfree does run on behalf of the transaction that calls it. From the first PGfree until the transaction terminates no other transaction can free a page, so PGfree calls should be issued close to transaction termination for maximum concurrency.

Although PGalloc can run as transaction 0 and still be atomic and serializable, it needs the services of the recovery module for permanence (i.e., writes to the bit map must be on the redo log). Otherwise, when a file is restored after disk damage, the bit map (not being on the redo log) won't be restored. The problem is avoided in XFS, because there is no guarantee of permanence anyhow. But, in a real system, writes to the bit map would have to be logged. That in turn means that an exclusive lock would have to be held until transaction termination, with a corresponding bad effect on concurrency. I don't have a clean solution to this problem right now.

The reader may wonder why, when a file is restored from the redo log, the bit map couldn't be constructed at that time. Unfortunately, this isn't possible with the structure described here, since recovery is done at level 3 and the bit map (where one is used) is at level 5. In a real system one doesn't have to be as strict about the levels as I choose to be, but I'd like to keep things separate until I am convinced that there is no alternative.

IX. LEVEL 6: ACCESS METHODS

This level provides mechanisms to store and retrieve information via a unique key. The only information stored is a disk address (typedef DADDR) which can be used to access a data record using the facilities of level 8. There are four different access methods, two using B-trees[7] and two using extendible hashing:[8]

EH	(Extendible Hashing: Rochkind) This access method is based on the algorithms given in Ref. 8. This module is the only one in XFS that uses the page allocation functions (level 5).
XH	(Extendible Hashing: Thompson) This access method, although similar to the method in Ref. 8, was independently invented by Ken Thompson. Minor modifications were made by Peter Weinberger and Matthew Hecht.
HBT	(B-tree: Hecht) This access method is described in Ref. 9.
WBT	(B-tree: Weinberger) This crash-resistant implementation of B-trees is described in Refs. 10 and 11. This access method is at a disadvantage in XFS, since its crash recovery mech-

anism is redundant, and the resulting overhead makes it much less efficient than HBT. Also, instead of storing the data directly in the leaf, a disk address is stored there and the data is stored in another file. In XFS, the data in the other file is also a disk address (DADDR), which could just as easily have been stored in the leaf, eliminating one level of indirection. However, I did not want to change the WBT functions, since their inclusion in XFS is only temporary anyhow. (I ignored the extra I/O when comparing WBT to the other methods).

Note that three out of the four access methods were not written specifically for XFS, but were integrated into XFS nonetheless—without change. This demonstrates the viability of hiding module CR (level 3) behind an interface that looks like the standard *UNIX* system calls (level 4).

A real file system probably needs only one extendible hashing access method and one B-tree access method. (Ordinary hashing and a user-defined method are other possibilities). I have included two of each type so that I could run some experiments comparing them. (These experiments are not reported on here.)

The interfaces to the four access methods are all different, and each offers different features. I won't cover the specifics here, since in XFS these modules are used only through a Uniform Access Method Interface (level 7). Suffice it to say that each module is capable of implementing the features of level 7.

X. LEVEL 7: UNIFORM ACCESS METHOD INTERFACE

This module hides the differences among the four access methods of level 6 behind a uniform interface. The idea is that one chooses the access method when a file is created, but after that the operations are identical for each interface. This has obvious advantages:

(*i*) There is only one interface to learn.

(*ii*) The decision as to which access method to use can be delayed. The software above level 7 can be developed using one method, and later another method can be substituted if performance experiments dictate.

(*iii*) A uniform interface makes it easier to test and compare the access methods, since the same program can be used on every method.

A disadvantage of the uniform interface is that it is a least common denominator—several features offered by only one method had to be omitted. However, a user who must have an omitted feature is free to use the access method directly at level 6 (giving up the advantages of the uniform interface).

One feature that is not uniformly provided, but that is not omitted either, is the ability of B-trees to efficiently scan the keys in alphabetical order. Omitting this property would be foolish, since it is the primary reason for using B-trees (instead of hashing) in the first place! So the externals of the uniform interface specifically state that sequential scanning is ordered if a B-tree method is used. If the higher-level algorithm needs this property, then one will not be able to substitute a non-B-tree method transparently.

The following paragraphs describe the uniform interface in detail.

```
#include <stdio.h>
#include "cram.h"

AMD *AMcreate(file, method, sort)
char *file, *method, *sort;

AMD *AMopen(file)
char *file;

void AMclose(a)
AMD *a;

void AMinsert(a, key, info)
AMD *a;
char *key;
DADDR info;

void AMdelete(a, key)
AMD *a;
char *key;

STATUS AMfetch(a, key, infop)
AMD *a;
char *key;
DADDR *infop;

void AMtake(a, keyp, infop)
AMD *a;
char **keyp;
DADDR *infop;

AMscan(a)
AMD *a;

AMnext(a, keyp, infop)
AMD *a;
char **keyp;
DADDR *infop;

void AMstat(a)
AMD *a;
```

The state of a file is kept in an access method descriptor (AMD). AMcreate and AMopen return pointers to AMDs, and the other functions take an AMD pointer as their first argument. (An AMD pointer is analogous to a FILE pointer as used with the standard I/O library).

AMcreate creates a new file and opens it. Method may be EH, XH, HBT or WBT. Sort, meaningful only for the B-tree methods (HBT and WBT), may be A for alphabetic or N for numeric. In alphabetic sorting, the keys are sorted in lexical order ("10" comes before "2").

In numeric sorting, the keys are sorted as numbers (2 comes before 10). If sort is NULL, A is assumed. For EH and XH sort must be NULL.

AMopen opens an existing file. The method is deduced from the file itself.

If the services of level 3 are desired (concurrency control and recovery), then a file should be registered with freg (see Section VII), before AMcreate or AMopen is called. Furthermore, the external XNUM Xnum (defined in cram.h) must be set to the transaction number, like this:

```
    Xnum = CRbegin( );
```

This bit of awkwardness is forced by the fact that the AM functions (level 7) do not know anything about concurrency and recovery (level 3), so they can't help make transaction control convenient. However, things don't stop with level 7—there are many levels of database software before we reach user code, and one of these levels could arrange for transaction control.

AMclose closes a file. Files must be closed explicitly—a "close" is not automatic on process termination.

AMinsert inserts a DADDR (disk address), given by info, associated with a null-terminated ASCII string, given by key. There may be only one key per DADDR, and one DADDR per key. (Of course, a DADDR may be used to indirectly reference a list of DADDRs, but this is outside of the AM module). Before AMinsert is called, the data record has presumably been stored somewhere, and the DADDR says where. The RS functions of level 8 may be used to store records.

It may be useful at this point to recap the steps needed to store a record ("Colorado") associated with a key (CO). (RSinstall is described in Section XI.)

```
    #include <stdio.h>
    #include "cram.h"

    AMD *a;
    DADDR d;
    int fd;
    char *rcd = "Colorado";
```

```
char *key = "CO";
:
freg("dfile");                                          /*level 4 */
freg("kfile");                                          /* 4 */
Xnum = CRbegin( );                                      /* 3 */
fd = c-eat("dfile", 0666);                              /* 4 */
a = AMcxreate("kfile", "XH", NULL);                     /* 7 */
d = RSinstall(fd, strlen(rcd) + 1, rcd);                /* 8 */
AMinsert(a, key, d);                                    /* 7 */
CRcommit(Xnum);                                         /* 3 */
```

Note that atomicity, permanence, and serializability are guaranteed for this transaction, even though functions have been used (creat, AMinsert, etc.) that know nothing about concurrency or recovery.

AMdelete deletes a key and its associated information.

AMfetch retrieves the information associated with a key. The argument infop must point to a DADDR where the information will be stored. A STATUS is returned; it is defined in cram.h like this:

```
typedef enum {AMFOUND, AMMISSED, AMFAILED} STATUS;
```

If the key is found, AMFOUND is returned. For the hashing methods (EH and XH), AMFAILED is returned if the key isn't found. For the B-tree methods (HBT and WBT), if the key isn't matched exactly, AMMISSED is returned and the next key greater than the desired one may be obtained via AMtake. If there is no next key, AMFAILED is returned.

AMtake, which may be called only after AMfetch returns AMMISSED, sets the character pointer pointed to by keyp and the DADDR pointed to by infop to the key and associated information actually found. This function makes available one of the useful features of B-trees: the ability to reference a key by looking up a prefix and taking the next key in sequence.

AMscan and AMnext are used to scan the entire file. The scan is in sort order (alphabetic or numeric) for B-trees and in random order for hashing. AMscan starts (or restarts) the scan. AMnext retrieves a key and associated DADDR (in a manner like that of AMtake) each time it is called. It returns 1 if a key and DADDR were returned, and 0 on EOF. The functions are designed to be used in a for statement:

```
AMD *a;
char *key;
DADDR info;
:
for (AMscan(a); AMnext(a, &key, &info); ) {
    [do something with key and info]
}
```

Finally, AMstat prints statistics on the standard output. The statistics vary with each method and are not described here. Figure 4 shows a (baffling) example for an EH file.

XI. LEVEL 8: RECORD STORAGE

This module is capable of storing and retrieving records containing arbitrary data. This is the interface:

```
DADDR RSinstall(fd, nbyte, rcd)
int fd, nbyte;
char *rcd;

char *RSaccess(fd, d, nbytep)
int fd;
DADDR d;
int *nbytep;
```

RSinstall stores nbyte bytes of data pointed to by rcd in the file corresponding to file descriptor fd. The disk address of the stored record is returned. Fd is a normal *UNIX* file descriptor, obtained via creat or open.

RSaccess retrieves the record at disk address d. It returns a pointer to the record, and sets the integer to which nbytep points to the record's length. Normally, after a record is stored, its DADDR will be inserted into one or more indexes using the uniform access method interface (level 7). This approach is atypical; usually, a database file system has a primary key and secondary keys, and the record is stored along with the primary key. Here, however, there is no primary key. (Or, if you prefer, the DADDR is the primary key.)

Note that levels 7 and 8 are totally independent. Higher-level programs are responsible for ensuring that new records are indexed in all the ways they should be, that indexes are appropriately updated for changed records, and that appropriate deletions are made for

```
441 accesses; 441 installs; 0 deletes
9 splits; 3 doublings; 507 reinstalls
16 dir ptrs; 10 leaves; 441 keys; 1392 findleafs
3932 bytes of data; 77% leaf fill
1392 rdleafs; 957 wrtleafs; 948 avoided moves
281 lost reads; 0 premature writes
1211 512-byte reads; 949 512-byte writes
```

Fig. 4—Example of AMstat data for EH file.

deleted records. These tasks cannot be done at this level, since the data is arbitrary and there is no way to determine what the keys are. Higher levels will presumably impose a fielded record format on the data record.

There is no RSdelete function. If one is needed, it could, of course be added. Alternatively, garbage collection could be done by an offline utility. There is also no way to overwrite an existing record.

The RS functions can be used with or without concurrency control or recovery (i.e., freg called or not). In most of my experimental use of XFS, files used with the RS functions were not registered, since RSinstall always seeks to the end of the file before writing. If the transaction aborts, the written part of the file will have been wasted, but no logical inconsistency will result. I have a feeling that this property should not be made part of the specification, however, because it unnecessarily restricts implementation options.

XII. HIGHER LEVELS

My work on this file system structure stopped with level 8. I consider this to be the boundary between a database file system and the higher-level parts of the database. Above level 8 different designs tend to have a drastic effect on efficiency and user convenience, and much work remains to be done before a satisfactory higher-level structure can be designed.

For many purposes, one could program a simple level 9 module that would store fielded data records and automatically maintain one or more indexes. It would be driven by a table specifying the fields, how the keys are to be formed, what access method is to be used for each index, and so on.

XIII. ACKNOWLEDGMENTS

My design benefited greatly from discussions with Terry Bergeron, Mike Bianchi, Don Carter, Dan Fishman, Mathew Hecht, Reuven Kaswin, John Linderman, Bill Roome, and Glenn Rose.

REFERENCES

1. M. Stonebraker, "Retrospective on a Database System," ACM TODS, *5,* No. 2 (June 1980), pp. 225–240.
2. D. L. Parnas, "A Technique for Software Module Specification With Examples," CACM, *15,* No. 5 (May 1972), pp. 330–336.
3. UNIX User's Manual, ed. T. A. Dolotta, S. B. Olsson, and A. G. Petrucelli, New York: Holt, Rinehart and Winston, 1982.
4. J. N. Gray, "Notes on Data Base Operating Systems," in *Advanced Course in Operating Systems,* Technical Univ. Munich, New York: Elsevier North-Holland, 1977.
5. D. G. Severence and G. M. Lohman, "Differential Files and Their Application to the Maintenance of Large Databases," ACM TODS, *1,* No. 3 (September 1976),

pp. 256–267.
6. R. A. Lorie, "Physical integrity in a large segmented database," ACM TODS, *2*, No. 1 (March 1977), pp. 91–104.
7. D. Comer, "The Ubiquitous B-Tree," Computing Surveys, *11*, No. 2 (June 1979), pp. 121–137.
8. R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, "Extendible Hashing—A Fast Access Method for Dynamic Files," ACM TODS, *4*, No. 3 (September 1979), pp. 315–344.
9. M. S. Hecht, unpublished work.
10. P. J. Weinberger, unpublished work.
11. P. J. Weinberger, unpublished work.