A Library for Managing Persistent Storage

Dennis Heimbigner

CU-CS-351-86 March 1988

Revisions: April 3, 1987
February 5, 1988
March 9, 1988

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309-0430

The authors gratefully acknowledge the support of the National Science Foundation grant #DCR-8745444 in cooperation with the Defense Advanced Research Project Agency and the IBM Corporation.

A Library for Managing Persistent Storage

Dennis Heimbigner

CU-CS-351-86 March 1988

Revisions:
April 3, 1987
February 5, 1988
March 9, 1988

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

The authors gratefully acknowledge the support of the National Science Foundation grant #DCR-8745444 in cooperatio;n with the Defense Advanced Research Project Agency and the IBM Corporation.

1. Introduction and Philosophy

The storage package provides a library of routines by which a program can manage variable length records or fixed length records in a persistent heap (using Unix files), and index them using either btrees or extendible hashing.

2. Heap Interface

Figure 1 defines the interface to the heap portion of the library. This heap library can be used for either fixed length or variable length records. It uses a record size as its principal parameter. Records of any length can be allocated, although they will be embedded in an integral number of records.

```
typedef struct {
        int recordsize;
        int descr;
        filename_p name;
        int cachequota;
} heapparams_t, *heapparams_p;

#define MINBYTESPERRECORD 1

typedef int heap_p;

char *heap_errmsg;

extern int heapcreateparam(heap_p *hpp; heapparams_p htp);
extern int heapcreate(heap_p *hpp; char *name; int recordsize);
extern int heapopenparam(heap_p *hpp; heapparams_p htp);
extern int heapopen(heap_p *hpp; char *name);
extern int  heapclose(heap_p hp; int deleteit);
extern void heapset(heap_p hp; int value);
extern int heapget(heap_p hp);
extern void heapquery(heap_p hp; heapparam_p htp);
extern int heapdescriptor(heap_p hp);
extern char *heapname(heap_p hp);
extern int heaprecordsize(heap_p hp);
extern void heapprint(heap_p hp);
extern heap_p heapgen(heap_p hp);
extern int heaprecgen(heap_p hp; int prevaddr);
extern int heapalloc(heap_p hp; int nbytes);
extern int heaprealloc(heap_p hp; int address,oldnbytes,newbytes);
extern int heapfree(heap_p hp; int addr,nbytes);
extern int heapread(heap_p hp; int addr; char *buf; int nbytes);
extern int heapwrite(heap_p hp; int addr; char *buf; int nbytes);
```

Figure 1. Heap Interface.

One can create a new heap (using *heapcreate()*) by specifying its name and the recordsize (see below), or one can open an existing heap (using *heapopen()*), by specifying its name. In either case, the caller gets back a handle of type *heap_p*. When finished, a programmer calls *heapclose()* to close a heap, optionally deleting it if the *deleteit* parameter has the value TRUE (i.e. !0).

Given a handle to an open heap, one can allocate and free space in it much as one would using *malloc()* and *free()*. Space is allocated using *heapalloc()*, which is passed the heap handle and the number of bytes to allocate. It returns a handle to the allocated space. This record handle is actually a record offset into the heap file. The space allocated may be larger than the requested space because the request size is rounded up to the next multiple of the recordsize as specified in the *heapcreate* call used to create the specified heap. The minimum legal record size is one byte. Specifying larger sizes wastes space in allocation, but saves space in the bitmap used to record allocated space. Generally, larger record sizes are preferable.

Heapfree() is used to free allocated heap space. It takes a heap handle, an offset (the value returned by a call to *heapalloc()*), and a length specifying the length of space that is to be freed. The length should be the same as was specified in the corresponding call to *heapalloc()*.

The *heaprealloc()* routine is used much like *realloc()*, namely to enlarge or shrink a previously allocated space. This may result in moving the space to a different offset in the heap. That offset (changed or not) is returned as the value of this routine. *Heaprealloc()* copies the contents of the old space into the contents of the new space.

Access to the contents of an allocated space is made using *heapread()* and *heapwrite()*. These operate like the low-level Unix *read()* and *write()* system calls, except that the file descriptor is replaced with the heap handle.

It is often convenient to store a piece of information with the heap, such as the offset of a header record. To this end, two routines are provided: *heapset()* and *heapget()*. These allow a program to store and later retrieve a single arbitrary integer with the heap. This value is persistent, so closing a heap will not cause it to vanish.

When errors occur in the heap library, the routines normally return the value -1. At the same time, a pointer to a more informative error message is left in the global variable *heap_errmsg*. *Heapprint()* can be called to get a terse debugging printout of the state of the heap.

The heap also provides for more detailed management of the heap. An instance of the structure type *heapparams_t* can be filled in and passed to the routine *heapcreateparam* as an alternative way to create a heap and get a heap handle returned. The heap parameter type allows the specification not only of the file name and recordsize, but also which file descriptor to use. Presumably this descriptor is already open. Similarly, an existing heap can be opened using *heapopenparam*.

Given a heap handle, the record size, descriptor number, and name can be obtained using the routines *heaprecordsize()*, *heapdescriptor()*, and *heapname()* respectively.

It is possible to scan a list of all open heap handles using *heapgen()*. If *heapgen* is called with a zero argument, then it returns the first heap handle. Subsequent calls to *heapgen* should pass the previously obtained handle. When there are no more handles, *heapgen* will return a zero.

It is possible to scan the complete sequence of allocated records using *heaprecgen*. It is called with the heap handle and, initially, a -1 record offset. It returns a record offset of the first allocated record in the heap. Subsequent calls should pass in the last record offset obtained and will get back the next record offset. When no more record offsets can be found, a -1 is returned. This routine is not very useful if the heap has

been used to allocate variable length records because these records will span the underlying record structure of the heap.

3. Type Management

The storage manager provides indexing methods as its principal feature. Any indexing method represents a map from one type of value (a *domain* value) to another type of value (a *range* value). For example, a btree might map strings representing the values of some attribute into integers representing locations in a heap. The index methods described subsequently all assume that their users will specify sufficient information about the types used in the index. To this end, a specific interface has been established to allow users to specify necessary type descriptions.

Figure 2 defines the interface to the type manager portion of the storage library. A type is represented by an instance of the structure type *ixtype_t,* which is a collection of pointers to the functions which define the semantics of the type. This set of operations is somewhat ad-hoc since it was designed to meet the needs of the indexing systems provided in this library. The last element in that structure is a single word of user defined data. Users may extend this structure to include more space for data or operations as long as the first part is compatible with the existing structure.

This package supports three predefined types and is fairly easy to extend to include others. The predefined types are integers, fixed length strings, and strings of arbitrary length. The integer type assumes that it is dealing with 32 bit signed integers and does not use the data field of the *ixtype_t* structure. For fixed length strings, any instance of the *ixtype_t* structure is expected to put the length of the strings into the data field. Thus different copies of this type structure can be used to manage strings of different length (but fixed within an instance). Instantiations of arbitrary length strings are identical in properties. Since they store the strings on disk, they differ (potentially) in the file into which they store the strings. The data field for variable strings is expected to be a heap handle to a heap in which to store the strings.

The global variable table *ixtypetab* is an array of pointers to function tables (i.e. instances of *ixtype_t*). The indices for each predefined type are specified by the constants *IXT_INT, IXT_STR,* and *IXT_VARSTR.*

The semantics of the various operations is as follows:

create
When a type is instantiated by making a copy of the appropriate *ixtype_t* structure, it is created with whatever arguments are needed to create a completed type. Thus for fixed strings, the actual size of the strings is needed to make a concrete string type.

destroy
When an instance of *ixtype_t* is no longer needed, it should be destroyed to reclaim resources. At the moment, none of the types use this entry.

name
Return a string that is a printable name for this type. This is used mostly for debugging.

size When a type value is allocated into the index, it takes up some space. It is assumed that all instances of some type are the same size. This function returns that size.

tomemory
Since indices will be stored on disk as well as partly in memory, it must make provisions for moving type values to and from the disk. Many types have different

```
typedef struct {
        int (*create)(ixtype_p dp;...);
        int (*destroy)(ixtype_p dp;);
        char *(*name)(ixtype_p dp;);
        int (*size)(ixtype_p dp;);
        int (*tomemory)(ixtype_p dp; byte_p disk,mem;);
        int (*todisk)(ixtype_p dp; byte_p mem,disk;);
        int (*print)(ixtype_p dp; byte_p instance; FILE * f;);
        int (*compare)(ixtype_p dp; byte_p instance1, *instance2;);
        int (*hash)(ixtype_p dp; byte_p instance;);
        int (*insert)(ixtype_p dp; byte_p newinstance,space;);
        int (*delete)(ixtype_p dp; byte_p instance;);
        int (*update)(ixtype_p dp; byte_p newinstance,oldinstance;);
        byte_p data;            /* type specific data */
} ixtype_t, *ixtype_p;

extern ixtype_p ixtypetab[];

#define IXT_INT             0 /* Integers */
#define IXT_STR             1 /* Fixed size strings */
#define IXT_VARSTR          2 /* Varying size string */

#define ixtcreate(dp, arg)    ((dp)->create(dp, arg))
#define ixtdestroy(dp)    ((dp)->destroy(dp))
#define ixtname(dp)    ((dp)->name(dp))
#define ixtsize(dp)    ((dp)->size(dp))
#define ixttomemory(dp, disk,mem)    ((dp)->tomemory(dp, disk,mem))
#define ixttodisk(dp, mem,disk)    ((dp)->todisk(dp, mem,disk))
#define ixtprint(dp, instance, f)    ((dp)->print(dp, instance, f))
#define ixtcompare(dp, instance1, instance2)    ((dp)->compare(dp, instance1, instance2))
#define ixtinsert(dp, newinstance,space)    ((dp)->insert(dp, newinstance,space))
#define ixtupdate(dp, newinstance,oldinstance)    ((dp)->update(dp, newinstance,oldinstance))
#define ixtdelete(dp, instance)    ((dp)->delete(dp, instance))
#define ixthash(dp, instance)    ((dp)->hash(dp, instance))
#define ixtcopy(dpsize, srcixt, dstixt)    ((void)bcopy(srcixt,dstixt,dpsize))

typedef struct {
        int length; /* including the terminating null, if any */
        union { /* discriminated by length*/
          diskaddr ptr; /* length > 0 */
          char *mem; /* length < 0 */
        }u;
        char prefix[32]; /* must be multiple of ALIGNMENT */
} varstr_t,*varstr_p;

extern int true_fcn(),false_fcn(),simple_copy(ixtype_p; byte_p; byte_p;);
```

Figure 2. Type Management Interface.

representations on disk as opposed to memory. This entry takes a pointer to a disk representation converts it to a memory representation, and places it at the location specified by the second pointer. It should be designed so that the two pointers can point to the same place. Note that for the size function above, the size refers to the disk representation, not the memory representation.

todisk
Take a memory representation and convert it to a disk representation. It should be designed so that the two argument pointers can point to the same place in memory.

print
Print a string representation of a value onto a standard IO channel (i.e., type FILE *).

compare
Compare two instances and return a value (<0, ==0, >0) as the first instance relates to the second instance. This is intended to act like *strcmp* but for arbitrary types.

hashGiven an instance of a type, produce a 32 bit integer representing a hash of that instance.

insert
When an instance is first inserted into an index system, this routine is called. It is passed a pointer to the value to be inserted and a pointer to the space into the index into which it will be copied. A typical use of this is to convert the instance into a disk representation and copying that representation into the index space. This function is made call compatible with the **todisk** function so that the same function can be used for both.

delete
At some point, an instance is no longer needed in an index. This routine is called to reclaim any resources used by the instance.

update
When an instance is to be inserted into an index and a previous instance already exists in the index, this routine is called with both the instance to be inserted and the existing instance.

The **todisk**, **insert**, **update**, and **delete** entries are expected to return the value TRUE or FALSE to indicate if they succeeded, but currently, no use is made of that return value.

All of these operators are called with the a pointer to the index type (i.e., the instance of *ixtype_t)* as the first argument. The remaining arguments, if any, depend on the operator. In the case of create, they also depend on the type being created.

4. Pair Management

It is assumed by the indexing methods that there is a "pair" structure (actually a triple) that combines together some index specific data called overhead, a domain value, and a range value, in that order. This structure is termed a "pair". Figure 3 gives some macros that can be used to get pointers to the various parts of the pair. In these macros, *pairsize* is the total size (in bytes) of a pair, *oversize* is the size (in bytes) of the index specific data, and domsize is the size (in bytes) of a domain value. The *ithpair()* macro can be used to obtain a pointer to the i'th pair in a vector of pairs. Similarly, *ithover(), ithdom(), and ithrange()* can be used to get pointers to the overhead, domain, or range, respectively, of the i'th pair.

```
#define ithpair(pairsize,listptr,index)
        ((byte_p)((listptr)+(pairsize)*(index)))

#define ithover(pairsize,listptr,index)
        ((byte_p)((listptr)+(pairsize)*(index)))

#define ithdom(pairsize,oversize,listptr,index)
        ((byte_p)(ithpair(pairsize, listptr, index)+(oversize)))

#define ithrange(pairsize, oversize, domsize, listptr, index)
        ((byte_p)(ithpair(pairsize, listptr, index)+(oversize)+(domsize)))

#define paircopy(pairsize, pair1, pair2)
        (void)bcopy(pair1,pair2,pairsize)

#define calc_packing(overhead, pairsize, spacesize)
        (((spacesize)-(overhead)) / pairsize)
```

Figure 3. Pair Interface.

Paircopy() can be used to copy a pair from one location to another. Finally, *calc_packing()* will calculate the number of complete pairs that can be packed into a given amount of space.

5. Generic Index Interface

Most of this library is dedicated to indexing packages. Specifically, btrees and extendible hashing indexing are provided in the package. These methods have a great deal of common or generic structure, which is described in this section. Figure 4 defines the basic data structures used by an index. Figure 5 defines the basic functions provided by an index. The words "btree" or "hash" may be substituted in these figures for the word "INDEX" to get the name for an actual function in a specific index method.

An instance of an index can be created using *INDEXcreate()*. When creating an index, the caller must specify a place to return the handle to the index, a name for the index, and the sizes and types of its domain and range. An existing index can be opened using *INDEXopen()* by specifying a place to store the handle, a name, and, again, the domain and range types. The sizes of the domain and range will be returned in the last two parameters if they are non-nil. The specified domain and range should be compatible with those specified at the time the index was created. In this case, compatible only means that they have the same sizes as the types used when the index was created. In either case, the caller gets back a handle of type *INDEX_p*. When finished, a programmer calls *INDEXclose()* to close a INDEX, optionally deleting it.

Given a handle to an open index, a new (domain,range) pair can be inserted using *INDEXinsert()*. This takes an index handle, a pointer to a a domain value and a pointer to a range value. The pair is inserted into the INDEX. The *insert* or *update* functions for the type may be called during the insert. If they fail, then the insert is not

```
typedef struct {
   /* first five entries are essential */
   filename_p name;
   int domainsize;
   int rangesize;
   ixtype_p domain;   /* the domain type of this INDEX_ */
   ixtype_p range;   /* the range type of this INDEX_ */
   /* Following should be needed/set only rarely */
   int heapgrain;  /* for heap open */
   byte_p  heap;   /* heap for the index */
   byte_p pool;   /* pool for the index */
   int poolquota;  /* amount of space for pool cache */
} INDEXparams_t, *INDEXparams_p;

typedef struct {
        /* This is almost all index specific */
} INDEXstat_t,*INDEXstat_p;

typedef int INDEX__p;

typedef int INDEX_cursor_p;


/*
Copyright (C) 1988 Dennis Heimbigner
*/
```

Figure 4. Generic Index Types.

```
extern int INDEXcreateparam(INDEX_p *IXp; INDEXparams_p ipp);

extern int INDEXcreate(INDEX_p *IXp; filename_p ipname; int domainsize,rangesize;
                       ixtype_p domain,range);

extern int INDEXopenparam(INDEX_p *IXp; INDEXparams_p ipp);

extern int INDEXopen(INDEX_p *IXp; filename_p ipname; int *domainsize,*rangesize;
                       ixtype_p domain,range);

extern int INDEXclose(INDEX_p ip; bool_t deleteit);

extern void INDEXquery(INDEX_p IX; INDEXparams_p ipp);

extern int INDEXfind(INDEX_p IX; byte_p domainval,rangeval; INDEXcursor_p IXc);

extern INDEXcursor_p INDEXsearch_start(INDEX_p IX);

extern void INDEXsearch_stop(INDEX_p IX; INDEXcursor_p IXc);

extern int INDEXsearch_next(INDEX_p IX; INDEXcursor_p IXc, byte_p domainval,rangeval);

extern int INDEXdomain(INDEX_p IX; INDEXcursor_p IXc, byte_p domainval);

extern int INDEXrange(INDEX_p IX; INDEXcursor_p IXc, byte_p rangeval);

extern int INDEXinsert(INDEX_p IX; byte_p domainval,rangeval);

extern int INDEXdelete(INDEX_p IX; byte_p domainval);

extern INDEX_p INDEXgen(INDEX_p prev);

extern void INDEXset(INDEX_p IX; int userval);

extern int INDEXget(INDEX_p IX);

extern int INDEXheap(INDEX_p IX);

extern void INDEXgrain(INDEX_p IX; int *domsize,*rngsize);

extern char *INDEX_errmsg;

extern void INDEXprint(INDEX_p IX);

extern void INDEXstats(INDEX_p IX, INDEXstat_p IXs, bool_t detailed);
```

Figure 5. Generic Index Functions.

performed.

Given a handle to an open index, a (domain,range) pair can be deleted by specifying the domain value to *INDEXdelete()*. If no matching entry is found, then the routine will return FALSE (i.e. 0) otherwise TRUE (i.e. !0).

It is possible to interrogate an index to see if a particular domain value occurs in the index using the function *INDEXfind()*. It is called with an index handle and a domain value. If no matching entry is found, then the routine will return FALSE otherwise TRUE. If a matching domain value is found, the corresponding range value may be returned if the *rangeval* parameter is a non-nil pointer. Additionally, a cursor pointer will be returned in the *ixc* argument if it is non-nil (see below).

Providing range scanning is an important feature for indices. This means locating some domain value in the index and then scanning higher values from the index in domain order. In this package, a scan is established by calling *INDEXsearch_start()* or calling *INDEXfind()* with a non-nil *ixc* parameter. In the latter case, the scan starts with the first domain less or equal to the requested domain. In the former call, the scan starts with the lowest domain in the index. Since multiple scans may be occurring simultaneously on the index, a handle to a structure called a *cursor* is returned to the user to track a particular scan. A scan is done by repeatedly calling *INDEXsearch_next()* with the index handle and the cursor handle as inputs. Each time it is called, *INDEXsearch_next()* returns a (domain,range) pair of values if space is provided. If no more pairs exist, the function returns the value FALSE else it returns TRUE. When a scan is finished, the routine *INDEXsearch_stop()* should be called to clean up the cursor and reclaim resources.

At any point in the scan, the current domain can be obtained by calling *INDEXdomain()*. If no search is established, or there is no legitimate current domain, *INDEXdomain()* will return FALSE else it will return TRUE. Analogous to *INDEXdomain()*, there is a function called INDEXrange() to return the current range value.

It is possible to perform inserts and deletes while simultaneously scanning an index. The effect depends on the position of the cursor with respect to the domain value being inserted or deleted. If the value inserted or deleted is before the cursor position, then it will not be seen during the scan. If it is beyond the current cursor position, then its effect will be seen when the cursor reaches that point. It the pair right at the current cursor position is deleted, which means that it has already been scanned, then it will not be seen, but attempts to call *INDEXdomain()* or *INDEXrange()* will fail until the cursor is advanced. Inserting a value at the current cursor location will be seen during the scan.

When an error occurs in an index function, it normally return the value -1. A more detailed error message will be left in the external variable *INDEX_errmsg*. The routine *INDEXprint()* can be called to get a terse debugging printout of the state of the index.

A number of functions are similar to some provided for the heap described above. All of the open index handles can be generated using *INDEXgen()*. It operates much like *heapgen()*. It is also possible to store a word of data with an index using *INDEXset()* and *INDEXget()*.

The index also provides for more detailed management. An instance of the structure type *INDEXparams_t* (see figure 4) can be filled in and passed to the routine *INDEXcreateparam()* as an alternative way to create an index and get an index handle returned. The index parameter type allows one to specify a whole raft of parameters about the index to be created. Generally, any of the entries can be filled with a zero and a default value will then be used during creation. Similarly, one can open an existing index using *INDEXopenparam()*.

Given an index handle, one can obtain its current set of parameters using *INDEXquery()*. this can be done for any open index. As a special case, an *INDEXgrain()* function is provided to obtain the domain and range sizes associated with an index handle.

It is expected that every index package will record statistics about its performance on a per-index basis. The set of meaningful statistics is entirely index specific. At any time, the current set of statistics may obtained by calling the function *INDEXstats()*, which takes an index handle, a pointer to an instance of the structure *INDEXstat_t*, and a boolean to indicate the level of detail of the statistics.

6. Btree Indexing

One of the instances of the generic indexing is a btree package. Btrees provide reasonably fast access, plus a scan of the tree can be done in domain order. The types *btreeparams_t* and *btreestat_t* are specific to the btree and are shown in figure 6.

7. Extendible Hash Indexing

One of the instances of the generic indexing is an extendible hashing package. As a rule, this is faster than a btree, but unlike a btree, it is impossible to do true range scanning. The scan order is entirely determined by the hash function. Thus, the only use would be to scan all instances in the table in some arbitrary order. The types *hashparams_t* and *hashstat_t* are specific to the hash and are shown in figure 7.

8. Pool Subsystem

The pool module is a major subsystem used by both the btree and the the extendible hashing packages. However, users may want to use it as a frontend to the heap package to obtain its caching facilities.

The pool package serves two purposes:

(1) It ensures that all references to a leaf or node point to one copy in memory,

(2) It caches a limited number of nodes and/or leaves in memory to improve performance.

The pool operates as an interface between the index package and the underlying heap that is used to store the data for the index. It consists of a hash table to map heap addresses to buffers within the pool. It also features an Lru chain of released buffers to serve as the cache.

Figure 8 shows the interface provided by the pool package. A pool is created using *poolcreate()* by specifying a heap handle, a size for the hash table, and a cache limit (i.e., quota). The quota refers to the number of buffers to cache and not to the total space. A pool may be destroyed using *pooldestroy()*.

The principal operations on the pool provide for the allocation of space in the heap and reading and writing from the heap. However, since these activities are cached, The data may not be immediately written to heap. The *poolalloc()* function operates more or less like *heapalloc*, except that it takes a pool handle rather than a heap handle. Note that the heap offset is returned in an argument (addr) rather than as the value of the function. The value returned by the function is a pointer to a memory buffer which will be written (eventually) to the specified offset in the heap. There is also a *poolrealloc()* function with the obvious semantics.

```
typedef struct {
   /* first five entries are essential */
   filename_p name;
   int domainsize;
   int rangesize;
   ixtype_p domain;   /* the domain type of this btree */
   ixtype_p range;    /* the range type of this btree */
   /* Following should be needed/set only rarely */
   int heapgrain;  /* for heap open */
   byte_p  heap;    /* heap for the tree */
   byte_p pool;    /* pool for the tree */
   /* actual poolquota is this poolquota-1, so 0 still means default*/
        /* => poolquota of 1 in this structure is really 0 internally*/
   int poolquota;
   /* Index specific data */
   int nodesize; /* physical size in bytes. */
   int leafsize; /* physical size in bytes */
   int pairspernode;
   int pairsperleaf;
} btreeparams_t, *btreeparams_p;

typedef struct {
        int levels;             /* number of layers in tree */
        int nodeno;             /* # nodes in tree */
        int leafno;             /* # leaves in tree */
        int pairspernode;
        int pairsperleaf;
        int *leafstats;         /* for each leaf, # of entrys */
        /* following should match poollib.h */
        struct {
                int quota;      /* max # of buffers in lru chain */
                int probes;     /* # of probes into table */
                int hit; /* # of times buffer is found in table */
                int lruhit;     /* # of times buffer is found and is on
                                        lru chain*/
                int read;       /* # of buffers read from disk */
                int write;      /* # of buffers written to disk */
        } poolstats;
} btreestat_t,*btreestat_p;

#ifndef BTREEKERNEL
typedef int btree_p;

typedef int btreecursor_p;
#endif
```

Figure 6. Btree specific types.

```
typedef struct {
   /* first five entries are essential */
   filename_p name;
   int domainsize;
   int rangesize;
   ixtype_p domain;   /* the domain type of this tablee */
   ixtype_p range;   /* the range type of this tablee */
   /* Following should be needed/set only rarely */
   int heapgrain;  /* for heap open */
   byte_p  heap;   /* heap for the tree */
   byte_p pool;   /* pool for the tree */
   /* actual poolquota is this poolquota-1, so 0 still means default*/
        /* => poolquota of 1 in this structure is really 0 internally*/
   int poolquota;  /* for pool create */
   /* Index specific data */
   int leafsize; /* physical size in bytes */
   int pairsperleaf;
   int max_depth;
   float pad_factor;
} hashparams_t, *hashparams_p;

typedef struct {
        int tablesize;          /* current # slots in hash table */
        int leafno;             /* # leaves in table */
        int pairsperleaf;
        struct leafstat {
                int full;/* # full slots in leaf */
                int deleted;    /* # deleted  slots in leaf */
        } *leafstats;           /* length = leafno */
        /* following should match poollib.h */
        struct {
                int quota;      /* max # of buffers in lru chain */
                int probes;     /* # of probes into table */
                int hit; /* # of times buffer is found in table */
                int lruhit;     /* # of times buffer is found and is on
                                        lru chain*/
                int read;       /* # of buffers read from disk */
                int write;      /* # of buffers written to disk */
        } poolstats;
} hashstat_t,*hashstat_p;

#ifndef HASHKERNEL
typedef int hash_p;

typedef int hashcursor_p;
#endif
```

Figure 7. Extendible Hash specific types.

```
typedef struct {
        int quota;      /* max # of buffers in lru chain */
        int probes;     /* # of probes into table */
        int hit; /* # of times buffer is found in table */
        int lruhit;     /* # of times buffer is found and is on lru chain*/
        int read;       /* # of buffers read from disk */
        int write;      /* # of buffers written to disk */
} poolstat_t,*poolstat_p;

#define NILPOOL (pool_p)0

typedef int pool_p;

extern int poolcreate(pool_p *pp; heap_p hp; int tablesize,quota);
extern void pooldestroy(pool_p cc);
extern char *poolalloc(pool_p pp; int length,*addr);
extern void poolfree(pool_p pp; int addr,length);
extern char *poolread(pool_p pp; int length,addr);
extern void poolwrite(pool_p pp; char *data);
extern void poolsync(pool_p pp);
extern void poolrelease(pool_p pp; char *data);
extern int pooladdr(pool_p pp; char *data);
extern void poolbumpref(pool_p pp; char *data);
extern char *poolrealloc(pool_p pp; char *data; int newlength,*addr);
extern poolprint(pool_p pp);
extern void poolstats(pool_p pp; poolstat_p ps);
extern void poolbufquery(pool_p pp; char *buf; int *addr,*length,*bufid);
```

Figure 8. Poolib.h.

One may read a specific chunk of the heap by using *poolread()* and specifying a length and a heap offset. Again, the result is a memory pointer to a buffer containing that data. If the requested data is already in memory (as determined by the heap offset), then the pre-existing buffer will be returned. The *poolwrite()* function indicates that a buffer has been modified in memory and should be written back to the heap. The write will be delayed until buffer is to be destroyed or re-used. *Poolsync()* may be used at any point to force all dirty buffers to be written back to the heap.

It is important to realize that the a reference count is associated with the pool buffers. Every *poolalloc()* and *poolread()* increments the count. As long as the count is greater than zero, the buffer will never be re-used. *Poolrelease()* tells the pool to decrement the reference count. As side effect, the buffer is placed on the cache list, which in turn may force a buffer off the cache to keep within quota.

There are a couple of miscellaneous functions for manipulating the pool or buffers. It is possible to get the heap address associated with a buffer using *pooladdr(),* and it is possible to increment the referenc count for a specific buffer using *poolbumpref().*

Finally, it is possible to get a debugging printout of the state of the pool using *poolprint(),* and statistics may be obtained using *poolstats().* The statistics returned can be seen by examining the type *poolstat_t.*

9. Using the Library

The storage system consists of the heap, the btree, the hash table, the pre-defined types, and miscellaneous support packages, all combined into a single library. The user of this library will need the following three files:

storagelib.a

This is the library file for the storage system. It contains the code for btree, extended hash, heap, and pre-defined types. If you intend to use the library, then you need to make sure that this library is loaded with your code. For example:

```
cc yourprog.c storagelib.a
```

storagelib.h

The definitions required to use the storage library are kept in this file. You should **#include** it in appropriate places in your code.

llib-lstore.ln

This is a lint library for the storage system. You may use use it in your lint commands to keep lint quiet and to check conformity between your code and the storage library.

10. Implementation Notes

10.1. Conditional Compilation Flags

This system is parameterized in very few ways. But there are two conditional compilation flags whose meaning you should understand.

NOALIGN

Some machines (e.g., IBM RT, Sun4) require integers pointers to point to 4 byte boundaries. Other machines (e.g., Vax, Sun3) do not have such a requirement. if the NOALIGN flag is defined, then the index methods will do a certain amount of space compaction in the index. There is a price to be paid for turning this flag on, however, because some other algorithms must switch from 4 bytes at a time copy to 1 byte at a time, so the system will slow down noticeably. In general, since the space saved is minimal and the performance degradation is substantial, this flag should be left undefined.

11. Setup

Once you have unpacked the files into their appropriate subdirectories, you should take an inventory by the following command:

```
make Inventory
```

the only differences should be the names of files that you specifically placed in the directory.

Next, you need to modify the Makefiles to match your environment. In particular for each of the following conditions, you should execute the corresponding command:

(1) If your C compiler does not support the -M flag for generating Makefile dependencies, execute:

```
make NoM
```

(2) If your machine is not a Sun, then if probably does not support the Sun debug version of malloc(), so you should execute:

```
make NoMalloc
```

After this, you need to build some header files and set up the dependency lists in the Makefiles. To do this, execute the command

make setup

At this point, you can construct the library itself by executing:

make all

This will construct storagelib.a (the library) and llib-lstore.ln (a lint library).

12. Testing the Library

The subdirectory named *tests* contains a large number of programs to test the operation of the components of the storage library. If you can get these tests to execute correctly on your machine, there is a good chance that the library will work for your applications. If you execute the command

make all

then the test will be compiled, executed, and compared with a reference set of results. The results are placed in a file *tests,* the reference set is in *tests.ref* and the comparison (using *diff*) is in testdiffs.

If you see differences, then that indicates that there is a problem and you will have to investigate the cause of the difference. Unfortunately, you are on your own at this point.

This package has been successfully tested under the following machines: Sun 3, Sun 4, IBM RT, Pyramid, Vax 780, Mips, and Encore.

13. Performance

The subdirectory *perfs* contains two programs, bperf0 and eperf0, to measure some aspects of the performance of the btree and extendible hash packages respectively. Each operates by creating an index and then inserting some number of entries into the index. Each takes a number of optional command line arguments:

-c <count>
: Specifies the number of items to insert into the index

-d
: If specified, then more detailed statistics will be provided as output.

-k
: If specified, then the index will not be deleted at the end of the program's execution.

-l <leafsize>
: Specifies the size (in bytes) of leaves in the index.

-n <nodesize>
: Specifies the size (in bytes) of nodes in the index. Applies only to btrees.

-o
: If specified then the entries to be inserted will be consecutive integers starting at 0.

-q <quota>
: Specifies the number of buffers to cache in the pool subsystem. Since the clock time is totally driven by the number of disk accesses, increasing this can have a dramatic effect on execution time.

-s <seed>
: If "-c" is not specified, then the the entries to be inserted are random 32 bit integers. The seed for the random number generator is specified by this parameter.

The default settings are as follows:

```
bperf0 -c 10000 -l 4096 -n 4096 -q 6 -s 1
eperf0 -c 10000 -l 4096  -q 14 -s 1
```

The quota settings were chosen so that both programs would have approximately 50% hit ratios in the cache and would do the approximately the same number of read and write systems calls.

Appendix A: The Common Definitions Header

The file common.h contains a set of miscellaneous definitions that are used in the library. This file is included as part of storagelib.h and is the most likely to cause conflicts with your applications code.

```
#include <sys/param.h>
/* define a bunch of library stuff so that lint wont complain */
extern char *getwd();
extern long atol();
extern char *malloc();
extern char *memalign();
extern char *realloc();
extern char *strcpy();
extern char *strcat();
extern char *strncpy();
extern long lseek();
/*VARARGS*/
extern int fprintf();
/*VARARGS*/
extern char *sprintf();

#include <errno.h>
extern int errno;

/* Define some useful types */
#ifndef TRUE
typedef int  boolean;
typedef int bool_t;
#define true 1
#define false 0
#define TRUE 1
#define FALSE 0
#endif
```

Figure A1 (a). Common.h (Part 1).

```
/* allow us to be more specific about pointers */
typedef char byte, *byte_p;

#define nil 0

typedef char name_t[MAXPATHLEN],*name_p;
typedef char filename_t[MAXPATHLEN],*filename_p;

typedef int diskaddr; /* must potentially allow negatives */
#define NOADDRESS -1      /* Assumed to be impossible disk address */

/* defined some sizes so the macros can work */
#define BITSPERBYTE       NBBY
#define BYTESPERINT       NBPW
#define BITSPERINT(BITSPERBYTE*BYTESPERINT)
#define MAXINT            ((int)0x7fffffff)

/* misc macros */
#define min(x,y) ((x)<=(y)?(x):(y))
#define max(x,y) ((x)>=(y)?(x):(y))
#define multiple(x,y)  (y*(1+(x-1)/y)) /* find next multiple of y larger than x */


/* use an alternative malloc that checks nil result */
extern char *xmalloc();
extern char *xrealloc();
extern char *xmemalign();

/* Define the Alignment requirement for this Machine */
#ifdef NOALIGN
#define ALIGNMENT 1
#else
#define ALIGNMENT sizeof(int)
#endif
```

Figure A1 (b). Common.h (Part 2).

Appendix B: Sample Performance Measurements

Figures B1 and B2 show some sample performance measurements. from various machines. This table should be read as anecdotal and not as a comprehensive set of tests. In the table, *bperf0* and *eperf0* were each run twice.

Index Parameters				
Index Method	Iterations	Node size (bytes)	Leaf size (bytes)	Cache quota
Btree	10000	4096	4096	6
Hash	10000	-	4096	14

Index Computed Parameters				
Index Method	Node Fanout	Leaf fanout	Cache hit rate (%)	#reads/#writes
Btree	512	511	54	4603
Hash	-	337	53	4751

	Times (msec/insert)					
Machine	Btree			Hash		
	Clock	CPU	System	Clock	CPU	System
Sun 3/75 (client)	5.7	1.5	3.6	9.0	0.9	3.1
Sun 3/260 (server)	12.3	1.0	2.6	6.8	0.6	2.3
IBM RT	10.2	1.2	2.5	15.4	0.6	2.1
Pyramid 90x	13.4	3.2	9.5	9.6	1.4	7.5
Vax 780	10.4	4.2	8.7	11.7	2.1	7.1
Encore (8 Processor)	21.4	5.1	21.1	18.2	2.3	17.9
Mips	4.6	0.4	1.4	5.3	0.2	1.6

Figure B1. Test 1 Measurement Samples

Index Parameters				
Index Method	Iterations	Node size (bytes)	Leaf size (bytes)	Cache quota
Btree	10000	4096	4096	0
Hash	10000	-	4096	0

Index Computed Parameters				
Index Method	Node Fanout	Leaf fanout	Cache hit rate (%)	#reads/#writes
Btree	512	511	0	10032
Hash	-	337	0	10163

	Times (msec/insert)					
Machine	Btree			Hash		
	Clock	CPU	System	Clock	CPU	System
Sun 3/75 (client)	10.3	1.8	6.1	12.0	1.2	5.9
Sun 3/260 (server)	12.2	1.2	4.1	11.5	0.8	4.2
IBM RT	19.2	1.2	4.3	25.8	0.8	3.9
Pyramid 90x	25.8	3.7	17.2	19.9	1.8	15.2
Vax 780	20.0	4.7	14.4	20.3	2.6	12.9
Encore (8 Processor)	40.0	5.8	39.7	38.2	3.0	37.5
Mips	8.4	0.5	2.6	9.9	0.3	2.6

Figure B2. Test 2 Measurement Samples