TDBM: A DBM Library With Atomic Transactions

Barry Brachman, Gerald Neufeld – Dept. of Computer Science, University of British Columbia

ABSTRACT

The dbm database library [1] introduced disk-based extensible hashing to UNIX. The library consists of functions to use a simple database consisting of key/value pairs. A number of work-alikes have been developed, offering additional features [5] and free source code [14,25]. Recently, a new package was developed that also offers improved performance [19]. None of these implementations, however, provide fault-tolerant behaviour.

In many applications, a single high-level operation may cause many database items to be updated, created, or deleted. If the application crashes while processing the operation, the database could be left in an inconsistent state. Current versions of dbm do not handle this problem. Existing dbm implementations do not support concurrent access, even though the use of lightweight processes in a UNIX environment is growing. To address these deficiencies, tdbm was developed. Tdbm is a transaction processing database with a dbm-like interface. It provides nested atomic transactions, volatile and persistent databases, and support for very large objects and distributed operation.

This paper describes the design and implementation of tdbm and examines its performance.

1. Introduction

In the UNIX environment, the dbm database library[1] [1] has become widely used to provide disk-based extensible hashing for a variety of applications. The library consists of functions to use a simple database consisting of *items* (key/value pairs). A number of work-alikes have been developed, offering additional features [5] and free source code [14,25]. Recently, a new package was developed that also offers improved performance [19] and there are plans to add a transaction mechanism to this package [20].

As an integral part of our distributed system research, an efficient and reliable database was required. In these and many other applications, a single high-level operation may result in several objects being updated, created, or deleted. If the application or host system crashes while processing the operation, the database must not be left in an inconsistent state.

Many distributed applications have a server component that can handle many client requests simultaneously. For example, in the case of the X.500 Directory Service [4], a server called the Directory System Agent is most naturally implemented as a multi-threaded application, with one or more threads servicing each client request. To maximize the level of concurrency, the database should permit simultaneous read-only and update operations while guarding the database against inconsistencies.

Current versions of dbm, however, do not meet the requirements of these types of applications. Most importantly, they do not guarantee consistency in the face of crashes. Existing dbm implementations cannot be used in a multi-threaded application, even though the use of lightweight processes in a UNIX environment is growing. Also, no assistance for implementing distributed and replicated databases is given.

To meet these requirements, tdbm (dbm with transactions) was developed. Tdbm provides nested atomic transactions [13], volatile and persistent databases, support for very large data, stores the database within a single UNIX file, and provides assistance for managing distributed databases. Tdbm can be configured to operate either as a conventional UNIX library or as part of a multi-threaded application. The EAN object store [17], used by the EAN X.500 directory service [16], is based on tdbm.

In the next section, the major design decisions associated with tdbm are examined. In Section 3, we look at the implementation of tdbm and in Section 4 an evaluation of the performance of tdbm is given. Finally, the paper concludes with some thoughts about our experiences with tdbm and possible extensions and improvements. The manual page for the library appears in the appendix.

[1]This work was made possible by a grant from OSIware, Inc.

2. Design

In this section, a summary of important issues and requirements concerning transaction systems are presented, including a discussion of how `tdbm` addresses them. Although we discuss transactions in the general context of a multi-threaded application, transactions can be employed to advantage in a single-threaded environment. A discussion of important aspects of the environment in which `tdbm` was to be used follows. An overview of recovery techniques, nested transactions, and design considerations of the external hashing component are then given. The section continues with a description of volatile and persistent databases, the Threads lightweight process kernel, and support for distributed operations.

Why atomic transactions?

An atomic transaction is a sequence of operations that are performed as a unit. The collection of operations within the scope of the transaction is executed indivisibly and in isolation from any concurrent actions outside of the transaction. The concept of indivisibility is illustrated in Figure 1. If a process executing `Transaction1` explicitly aborts the transaction (e.g., because `BalanceA` is found to be less than $10) or the process crashes before the end of `Transaction1` is reached, then neither balance should be changed. Furthermore, if two or more processes execute `Transaction1` concurrently, the results should be the same as some sequence of non-concurrent executions of the transaction. This characteristic is called serializability.

```
Begin Transaction1
    Read     BalanceA
    Subtract $10 from BalanceA
    Write    BalanceA
    Read     BalanceB
    Add      $10 to BalanceB
    Write    BalanceB
End Transaction1
```

Figure 1: A Simple Transaction

Transactions provide implicit concurrency control, freeing the programmer from the need to explicitly manage locks on objects. Lock management typically involves operations such as creating a lock, obtaining a read or write lock for an object, upgrading a read lock to a write lock, releasing a lock, and detecting and resolving deadlock.

Because transactions provide atomicity, they also simplify exception handling for the programmer since an explicit abort ‘‘undoes’’ a partially completed request that may involve many objects and a considerable amount of intermediate state. Transactions provide a simple and easy to use facility to create fault-tolerant applications.

The Transaction Paradigm

To achieve indivisibility, a transaction must have four properties which together form the ‘‘ACID principle’’ [7]: All-or-none atomic behaviour, Consistency, Isolation, and Durability. These properties are defined as follows:

- Atomicity. If a transaction successfully commits, all actions within the transaction are reflected in the database, otherwise the transaction does not modify the database at all as far as the application is concerned.
- Consistency. The consistency of the database is preserved whether the transaction commits or aborts. A database is consistent if and only if it represents the results of successful transactions.
- Isolation. The intermediate states of data manipulated by a transaction are not visible outside of the transaction. This prevents other processes from reading and acting on these intermediate results.
- Durability. Once a transaction commits, its results will survive any subsequent failures.

As part of the mechanism to attain these characteristics, the transaction system must have a recovery component that is executed when the database is opened. Recovery involves ‘‘undoing’’ any intermediate results applied to the database by an incomplete transaction and reexecuting any completed transactions whose results may not be fully reflected in the database.

Environmental Issues

The primary motivation behind the development of `tdbm` was to replace our EAN X.500 directory service’s original `dbm` work-alike database with a more powerful and efficient one. The nature of database operations for the X.500 directory service (and directory services in general) is that there are relatively few updates compared to lookups; some of the design decisions were made in light of this observation.

An examination of the original database showed that keys were relatively short and that there was a mixture of small objects and larger objects. Table 1 shows some statistics gathered from an X.500 database consisting of about 5,900 entries and 36,249 items. Because of the way key structures were flattened into buffers, they all ended up being 20 bytes in length. These figures helped to guide the design of `tdbm`.

Type	Min	Max	Mean	Size
Keys	20	20	20.0	724,980
Values	38	23,122	181.4	6,577,713

Table 1: X.500 Database Item Sizes (bytes)

During the performance evaluation of the `dbm` work-alike database it was found that when large page sizes were used, there could be many items in a page (often several hundred) and a significant amount of time was spent searching sequentially within a page for a particular key. This information suggested a different page format.

The fact that the database is used by the object store within the directory service also influenced the design. The object store uses encoding functions to flatten and compress a data structure into a contiguous buffer before calling `tdbm` to store it. When fetching an item from the database, the inverse operation is performed by the object store to restore the original data structure. Because encoding and decoding always result in a new copy, `tdbm` gives the caller a pointer to the item. A side effect of this, however, is that the item may not be properly aligned with respect to the requirements of the processor architecture because the item's location within the buffer has been shifted during space compaction[2]. Likewise, a user storing a binary integer could encounter this problem. To solve this alignment problem, the caller can specify alignment requirements for the value so that it can be maintained within `tdbm`'s buffer.

The directory service required only two database files and there was no need for transactions across the two databases. This was taken advantage of to simplify the implementation of recovery

Recovery Techniques

In the interest of brevity, only the major approaches to recovery will be outlined. The reader interested in more detail is referred to [7].

A recovery technique must write log information to persistent storage (e.g., disk) either so that is later possible to remove the results of incomplete transactions applied to the database (UNDO) or to apply the results of complete transactions that are not reflected in the database when it restarts after a crash (REDO). If the log contains the physical representation of objects, it is referred to as *physical logging* ; if higher-level objects are recorded then it is called *logical logging.*

If physical logging is done at the page (disk block) level, whenever any part of a physical page is modified the entire page must be written to the log. If recovery is based on the UNDO operation, the old page must be saved; if the REDO operation is used, instead of modifying the old page, the new page image is logged. It is possible to reduce the size of the log by only recording the differences between the initial and final page images. At the logical level, for example, it is possible to record the operations and their parameters in the log so that a REDO of the user's request can be executed later, if necessary.

When UNDO is used, log information must be written before the database is modified (called *write-ahead logging*), while when REDO is used, log information must be written before the end of transaction is acknowledged.

With each of these approaches there is a tradeoff between normal processing overhead, recovery processing cost, and implementation complexity. The degree of assistance provided by the file system is also an important factor.

Nested Transactions

If a transaction is permitted to have one or more subtransactions associated with it, a hierarchical grouping structure called nested transactions [13] result. When a child transaction commits, its state is passed up to its parent; only when a top-level transaction commits can the changes be made durable.

Nested transactions fail independently of each other; subtransactions may abort without causing other subtransactions or the entire transaction to abort. They also form a convenient unit for parallelism, with each child transaction mapping on to a thread of control. Nested transactions provide synchronization among the subtransactions, making it easy to compose new transactions from existing transactions without introducing data inconsistency arising from concurrency.

Extensible Hashing

A considerable amount of work has been done on extensible hashing schemes [18,22]. In these algorithms, a database or hash file is composed of some number of (usually) equal-length pages, with each page holding zero or more items. Most of these schemes aim to retrieve the page holding the item of interest in one or two disk operations as the hash file grows and shrinks. The goal is to maintain this performance without having to do a costly rehashing of all of the items as the size of the hash file changes. While the various approaches vary in their complexity, space overhead, and ease of implementation, they all tend to depend on a secondary data structure, such as a directory or index, to assist in locating the page containing the item. When the occupancy or load factor of a page falls outside its allowed range, a reorganization takes place; e.g., an overfilled page will be split into two partially filled pages and a record of the page split is made in the directory structure. Variations on this theme include Extendible Hashing [6,12], Dynamic Hashing [8], Linear (Virtual) Hashing [9,10,11,19], and Thompson's `dbm` method [24].

In some algorithms, relatively large directory structures may result. Schemes that require special page overflow handling (e.g., via bucket chaining)

[2]The starting address for dynamically allocated memory is typically chosen so that any data type stored there will be properly aligned. Therefore, making a copy solves the alignment problem.

have additional complexity for these special cases. In the context of transactions, schemes that access many pages have the unfortunate consequence that these pages will need to be read or write locked. Apart from the extra I/O overhead of reading and writing these pages, there might be substantial concurrency control overhead.

In any of these schemes, some additional mechanism is required to handle keys or values that are too large to fit in a page. One way to handle the problem is to put a pointer to the item in the location where the item should go. The pointer could simply be the name of a separate file containing the item, as was done in our previous `dbm` work-alike. While this is relatively easy to implement, it suffers comparatively high overhead in creating, opening, closing, and removing the separate file. This can be particularly inefficient if the large item is frequently updated. Another approach is to effectively implement a simple file system so that pages can be dynamically allocated and released within the hash file. The page number generated by the extensible hashing scheme is then treated as a logical page number which is mapped to a physical page number within the hash file. Physical pages need not all be the same size. The Berkeley Hash Package (`bdbm`) [19] uses an algorithm called buddy-in-waiting to support multiple physical pages per logical page.

After surveying the literature and experimenting with several hashing schemes, it was decided to reimplement Thompson's method because its performance was known to be good, we have had considerable experience with it, and it also appears to behave reasonably well with respect to its page access patterns. We elected to implement a simple bitmap-based dynamic page allocation mechanism to manage operations on contiguous page ranges to handle the problem of indirect items. It was also decided to maintain the database within a single UNIX file to keep the design cleaner.

Volatile and Persistent Databases

In many applications, such as the directory service, it is necessary to maintain a database of persistent objects; i.e., objects written to non-volatile storage so that they can exist independently of the process that created them and survive a system crash. On the other hand, there are applications that want these objects to always be in memory, perhaps for performance reasons, or because it is unnecessary or impractical to save the objects outside of a process' volatile memory. Because of this, transactions on volatile databases should not require any disk accesses. Some applications might like the choice between volatile and persistent databases to be made at runtime; having to use a separate package such as `hsearch` [2] for volatile databases is not a viable option.

The desire to avoid a separate set of functions for each of the two database storage modes called for the notions of volatile and persistent databases to be integrated. As a consequence, the most straightforward design seemed to have all operations occur in memory up to commit time, when secondary storage became involved in the case of persistent databases. Designs involving writing intermediate states to disk were ruled out. Also, rather than implementing complicated secondary storage management functionality as for differential files, it was decided to let UNIX's virtual memory system deal with memory management.

Note that because transactions on volatile databases are not durable, no recovery component is needed for them outside of handling aborts.

Threads

Threads [15] is a lightweight process kernel that resides within a single host operating system user process. It provides fast thread creation, a shared address space for all threads, non-preemptive scheduling, and efficient IPC (using blocking Send/Receive/Reply) and context switching between threads. Of considerable significance to `tdbm`, the implementation of locks, semaphores, and shared memory does not require any intervention by the UNIX kernel since there is a single address space for all threads.

Portability is facilitated through the sub-kernel technique because instead of making system calls to the host kernel directly, applications must call the sub-kernel's versions. As a result, there is little application code that directly relies on the operating system. Threads has been ported to several different flavours of UNIX as well as several different machine architectures.

Support for Distributed Databases

One of the original design goals called for databases capable of being used with an object store that supports distributed operations and replication. The distributed object store is responsible for interprocess communication and execution of an atomic commit protocol (such as the two phase commit protocol [3]), but the underlying databases must provide some assistance.

Consider the case shown in Figure 2 where a distributed object store updates items in two or more databases within an object store transaction, each database running in a different server process, possibly on different hosts. One of the cooperating object store servers is distinguished as the transaction coordinator. After the last update in the transaction has been executed, the coordinator wants the transaction to be committed at all databases or at none. In the first commit phase, the coordinator requests all databases to ''precommit'' (prepare to commit) their part of the transaction and report the outcome. After a successful precommit, each database guarantees that

a subsequent commit (the only operation allowed on the transaction beside abort) will succeed. If one or more databases fail to precommit, the transaction is aborted. If the coordinator proceeds to the second phase, all databases will be asked to commit their transaction.

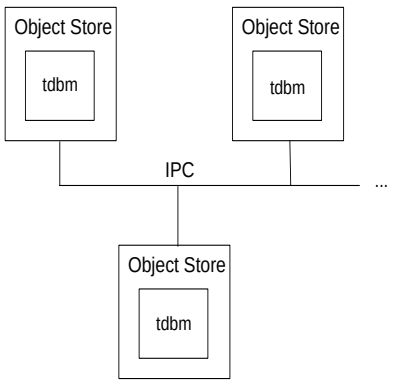

Figure 2: A Distributed Object Store

When the object store restarts, it needs a way of determining whether there was a distributed transaction in progress, and if so, which databases were involved and which phase the transaction was in. If a database crashes after its transaction is precommitted but before the second phase completes, the object store must determine whether to commit or abort the transaction when it restarts. This requires that the object store keep some state information about a transaction until it knows that all databases have committed or aborted. Also, as part of recovery, a database may need to contact the coordinator to determine the outcome of the transaction.

3. Implementation

The `tdbm` library is implemented as three independent layers: the item layer, page layer, and transaction layer. It would be relatively straightforward to replace a layer or have multiple versions of a layer. The `tdbm` library consists of approximately 6,500 lines of C source code, including header files and comments.

Although for our purposes it was not necessary to maintain compatibility with `ndbm`, the interfaces are quite similar. The major differences are that most `tdbm` functions require database and transaction identifying parameters and most functions return a result code.

The `tdbm` library can currently be compiled to run in ''non-concurrent'' mode (i.e., without any locking) and in multi-threaded mode under the Threads lightweight process kernel. In either case, `tdbm` runs as a single user-level UNIX process. When configured for concurrency, `tdbm` uses Threads' semaphores and lock manager, so no extra UNIX system calls are required.

The Item Layer

The item layer deals with the layout of key/value pairs in a page. There are two kinds of pages. The first kind is similar to that used by `dbm` and contains a directory for the items stored in the page and zero or more items. All of these pages are the same length. The second kind, indirect pages, are a variable number of physical pages long and simply contain data values that are too large to fit in any normal page.

In addition to its simplicity, the original `dbm` page format has the advantage that the contents of a page are packed so that there is no fragmentation between items. Reducing the number of pages helps to keep the database small and makes iterating through all items in the database more efficient. In the context of transactions, this is important because it reduces main memory requirements since many pages might be held in memory during the life of a transaction.

To help lower the overhead of searching for a key within a page, the `tdbm` page format was designed so that a binary search could be used to locate an item. This data structure permitted the same efficient item packing within a page at the expense of maintaining additional directory information. Keeping the directory ordered slows adding and deleting entries somewhat, but significantly reduces search times when there are many items per page.

Each normal page consists of a variable length directory and some number of items (Figure 3). The directory is a vector of unsigned short integers, beginning with a count of the number of entries in the page and `indexlast`, the index for the directory entry that points to the innermost item (i.e., the one bordering on the free space). Each directory entry consists of a tuple: `(keyoffset, keylength, dataoffset, indexnext)`. These directory entries are kept sorted by their key, in ascending `keylength` order and lexicographically for equal keylengths. The first directory entry is for the ''smallest'' key and the last is for the ''largest''. The `indexnext` fields form a circular list and keep track of the relative positions of the items; that is, `indexlast`'s `indexnext` is the index of the directory entry for the item closest to the end of the page. The `indexnext` of that directory entry is for the item second from the end of the page, and so on. If the value is stored indirectly, the value field in the current page indicates the physical page number where the real content is and the real size of the content.

Each entry has a `flag` byte that indicates the alignment requirement for the value and whether the value is stored in the current page or indirectly. The flag byte consists of four fields. The alignment constraint is specified by the user at the time the value

is stored: no constraint, even address, and addresses divisible by 4 or 8. For example, a character string probably would not require special alignment but an integer might require an address divisible by 4 so that after fetching the value it can be accessed directly from the page buffer without copying. Alignment requirements for the key, as specified by the user, are currently stored but not enforced by the system. The system sets the `Indirect` bit if the value is too big to fit on a page (the key remains on the page).

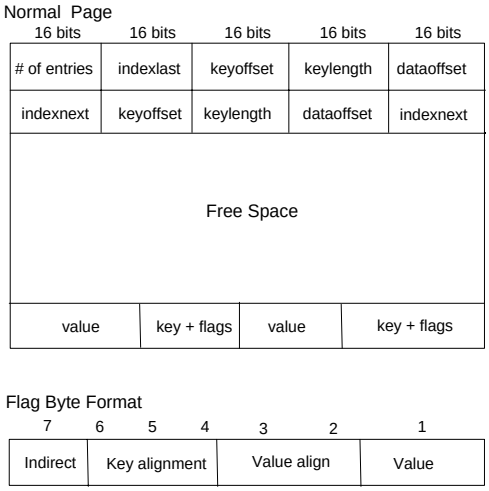

Figure 3: Page Layout

This organization provides a good tradeoff between lookup speed and space utilization, although if there are a great many very small items the directory space required per item could be unacceptable. Both the original `dbm` organization and the new one can spend a considerable amount of time compacting or coalescing page contents after a deletion to eliminate fragmentation. This is particularly evident in page splitting, since on average one half of the entries are deleted and moved to a new page. While this is acceptable when lookups are more frequent than updates, it may not be in the reverse situation. One solution is to allow alternative page directory formats, either per database or per page. The user could select a data structure to optimize for space or time or the system could automatically convert formats based on the space utilization.

The Page Layer

The page layer deals with management of logical pages, allocation of physical pages, page caching, and the mappings from keys to logical pages and logical to physical pages. It is not concerned with the page contents.

At the time a database is created, the user can specify the physical page size and the allocation unit size. The page size must be a power of 2 and should take into account the best I/O block sizes for the filesystem and internal fragmentation of indirect items. An entry that won't fit into a page is stored outside the normal page space and suffers an extra disk read and, likely, internal fragmentation. This internal fragmentation is ameliorated somewhat by having an allocation unit size in addition to the page size. The allocation unit size is the size of each page in the file as far as page allocation is concerned.

In addition to configuration constants like the pagesize, the header of the database contains three tables: the splitmap, physmap, and freemap. The splitmap is equivalent to the `.dir` file of a `dbm` -style database. It keeps track of how many times each page has split. The physmap maps a logical page number to a starting physical page number. The maximum size of these tables, in pages, is determined at compile-time. The maximum number of logical pages is (`NPHYSMAP_PAGES` * *pagesize*) / *sizeof(u_int)*, where `NPHYSMAP_PAGES` is a compile-time constant.

The freemap is a bitmap representation of physical page allocation. Each bit in the freemap represents *(pagesize / allocunits)* bytes. For example, if *pagesize* is 8192 and *allocunits* is 8, then disk space allocation is in blocks of 1024 bytes and 8 contiguous blocks are needed to allocate one page. As *allocunits* gets larger, external fragmentation tends to grow but internal fragmentation in indirectly stored data tends to decrease. With a *pagesize* of 8192 bytes, one freemap page contains (8192 bytes/page * 8 bits/byte) == 65536 bits/freemap page, which can map (65536 pages * pagesize bytes/page) == 536Mb. Physical page 0 is the file header and pages `INITIAL_FREEMAP_PAGE` through (`INITIAL_FREEMAP_PAGE` + `NFREEMAP_PAGES` - 1) are preallocated for the freemap. Pages after this last page are dynamically allocated and page `FIRST_AVAIL_PAGE` will be the first physical page represented in the freemap. While this approach leaves several holes at the beginning of the file (making it look much bigger than it really is), it simplifies the problem of having the freemap allocate new space for itself.

Here are the values currently being used, requiring a minimum page size of 2048 bytes to accomodate the header:

```
#define NSPLITMAP_PAGES         100
#define NPHYSMAP_PAGES          400
#define INITIAL_FREEMAP_PAGE    1
#define NFREEMAP_PAGES          10
#define FIRST_AVAIL_PAGE        \
    (INITIAL_FREEMAP_PAGE + NFREEMAP_PAGES)
```

The hash function that maps the user's key string to an integer is part of the page layer. A good hashing function is critical to the performance of any extensible hashing package. Nine functions were evaluated by having each hash a list of 84,165 strings made up of English words and symbol table entries and counting the number of collisions. One

of the best, the hash function from `sdbm` [25] (also used in `bdbm`), was chosen for `tdbm` because it did not generate a single collision.

The Transaction Layer

The transaction layer provides nested transactions over logical pages. The transaction layer is responsible for locating the appropriate version of a page for a transaction, concurrency control, and commit and recovery processing.

Every page that is read from the database is cached as a top-level or base copy [3]. All transactions that read the page share this base copy. If a subtransaction updates or creates a page, it retains a private copy of the page that may be later accessed by it and its subtransactions. The correct instance of a page for a particular subtransaction is quickly located by associating a simple hash table with each transaction identifier. The search process involves examining the transaction's hash table, proceeding up the hierarchy through its superiors' hash tables, and finally reading the database, if necessary, until the page is found. As transactions commit, their state is merged with that of their parent; when a top-level transaction commits, the new pages are propagated back to the hash file.

Apart from reading base pages from the database during transactions, writing pages to the transaction file while preparing to commit, and copying pages from the transaction file to the database during the commit, no other file I/O is performed. When dynamically allocated pages are no longer required, they are freed for general use by the application. Keeping all accessed pages in virtual memory eliminates any I/O from a temporary file and makes the integration of persistent and volatile databases seamless. It was felt that performance would be better if a temporary file could be avoided and that, in our environment, long-running transactions are unlikely. A shortcoming of this approach is that the number of pages a transaction can access is limited by the constraints of UNIX's virtual memory subsystem; a transaction will fail if it requires more memory than is available. `Bdbm`, on the other hand, may use a temporary file in conjunction with a volatile database if its cache size is exceeded. For best performance, the `bdbm` user has to specify a sufficiently large cache size at the time the database is opened. This memory is reserved until the database is closed.

Currently, the programmer is responsible for synchronizing threads executing subtransactions with that of the parent transaction; a parent transaction cannot commit or abort until all of its subtransactions have terminated. Automatic blocking in these cases is a possible extension to the package.

[3] Currently, this copy is kept in memory only as long as some transaction needs it.

Concurrency Control

Concurrency control is only performed when `tdbm` runs under Threads; it is simply not configured into the system otherwise. When concurrency control is available, a particular database can be opened multiple times by different threads and there can be many concurrent `tdbm` transactions on the database.

Threads provides a lock manager that allocates, obtains, and releases a lock on behalf of a client thread. The `tdbm` library uses Threads semaphores to protect critical sections. As a tradeoff between overhead and the level of concurrency, concurrency control is performed at the page level rather than at the hash file level (as our work-alike did) or the item level. Before a page can be read, `tdbm` obtains a read lock for the page. `Tdbm` must obtain a write lock for a page, perhaps by upgrading a read lock that the transaction already holds, before a page can be written. In keeping with strict two phase locking [13], locks are not released until the top-level transaction commits or an abort occurs. When a subtransaction commits, the parent transaction inherits any locks.

A good deal of the complexity of the transaction layer is due to the lock management protocol required by nested transactions and sharing unmodified pages. Only a limited degree of deadlock detection is currently implemented.

Commit Processing and Recovery

Commit processing of a top-level `tdbm` transaction is done by creating a transaction file (also called an intention list [23]) that represents the actions that must be executed to update the database. This approach is known as *after-image physical logging* [7]. The transaction file is stored in the same directory as the database file.

The transaction file contains some header information followed by a variable number of fixed-length shadow pages that represent the new contents of physical pages in the database. When the transaction file has been written and secured to disk using the `fsync()` system call, a `chmod()` is done to it to atomically indicate that the transaction has been committed. At this point, the new pages in the transaction file can overwrite the old pages in the database file. Upon successful completion of top-level commit processing, the results of the transaction have been applied to the database and the transaction file can be removed. This technique is similar to the idea of differential files [21], although `tdbm` operations never access the transaction file.

Recovery is automatically initiated when `tdbm` is started so that incomplete transaction files (those without a file mode indicating they've been committed) can be removed and the contents of completed transaction files can be applied (or reapplied) to the database. Note that this recovery procedure is

idempotent; if the system crashes during the overwriting process, recovery can be retried until successful.

This approach to recovery was taken primarily because of its simplicity. For example, it is not necessary to make a copy of the old data. Also, it was used by the package `tdbm` replaced and known to work well in practice. Physical logging was chosen because it was straightforward to make commit processing atomic and idempotent; it was felt that it would be more difficult to implement and debug logical logging of operations. Alternate approaches require maintaining log information, perhaps in terms of `tdbm` commands rather than disk pages (logical logging), so that changes can be undone or reapplied. A disadvantage to this approach is that a number of small changes to many pages may result in a large transaction file, but this can be mitigated by careful choice of page size and was not deemed to be as significant as benefits arising from the scheme's simplicity.

Since modified pages are not directly written to the database file, no log information needs to be maintained. On the other hand, modified pages must be held in memory until commit time, effectively caching all accessed pages. For transactions that do not involve a huge amount of data, this is not a significant penalty in terms of memory requirements and should normally result in improved performance. Also, this greatly simplified implementation of volatile (non-persistent) databases since virtually all of the same code can be shared with that necessary for persistent databases. The same storage and retrieval algorithms are used whether the database is volatile or persistent.

Several extensions to this basic scheme were necessary to support two phase commit. A precommit operation was added for top-level transactions. It is called to precommit any local updates within the transaction and to track the state of a transaction that has successfully completed the first phase. In either case, the caller can associate arbitrary data with the precommit. If, at the time the database is opened, transactions are found that terminated after phase one but before phase two completed, the application receives an appropriate return code. It may then invoke the recovery operation to obtain the data associated with the transaction and subsequently complete the transaction. In this way, the application can update the transaction's global state. The database is unavailable for normal operation while recovery is going on. The transaction can be committed (and the transaction file removed) when all databases have agreed on the outcome.

4. Evaluation

In this section, the performance of `tdbm` is compared to that of the most widely-used extensible hashing library under UNIX, `ndbm`, and the package expected to be its replacement, `bdbm`. All experiments were performed on a Sun Sparcstation 1 running SunOS 4.1.1 with 24Mb of memory and a CDC Wren IV disk drive.

In the first experiment, the performance of the three libraries was measured for creating and retrieving data from persistent databases, varying page sizes and the amount of data being stored. The second experiment involved repeating the first series for `tdbm` and `bdbm` using volatile databases. In all cases, consecutive integers (as ASCII strings) were used as keys. Three different sets of data values were used; they are characterized in Table 2. The first is the list of words in `/usr/dict/words`, the second is the word list referred to in Section 3, and the third is a set of 200 RFC documents. All times reported are the sum of the times spent in user mode and system mode. The error in these measurements is approximately ±5%. `Tdbm` was tested outside of threads, so there is no concurrency control cost associated with its measurements. Each `tdbm` test run involves a single transaction.

To reduce the number of experimental factors being varied, the `bdbm` fill factor parameter[4] was set at 128 for all `bdbm` runs since that value tended to result in the best performance for all page sizes [19]. Also, `bdbm` was run with both the default cache size of 65,536 bytes and a size of 4Mb. The latter configuration was the smallest size that virtually eliminated `bdbm`'s need to access a temporary file. Since `ndbm`'s page size can't be determined at run time, several versions were compiled.

[4]The fill factor indicates a desired density within the hash table and approximates the number of keys allowed to accumulate in any one bucket.

Name	Number of Values	Value Sizes (bytes) Minimum	Maximum	Mean	Total
`/usr/dict/words`	25,144	1	22	7.2	206,672
`wordlist`	84,165	1	40	8.9	836,663
`rfc`	200	143	799,768	78,990.6	15,877,102

Table 2: Data Value Sets

Persistent Databases

The first experiment examined creating and reading persistent databases. Figure 4 shows the time to create a persistent database using the contents of `/usr/dict/words` as data values. The results of fetching all data items by using the data keys sequentially are shown in Figure 5 and the results of having the databases iterate through each key are shown in Figure 6.

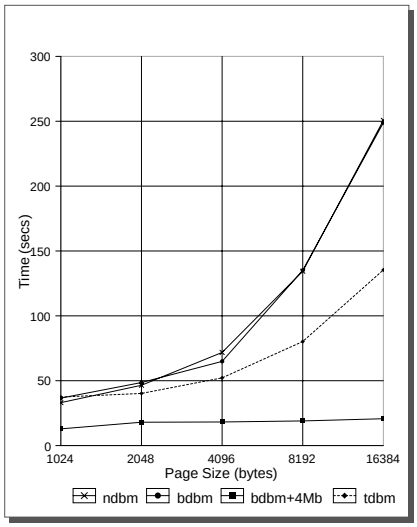

Figure 4: Creating Persistent Databases (`/usr/dict/words`)

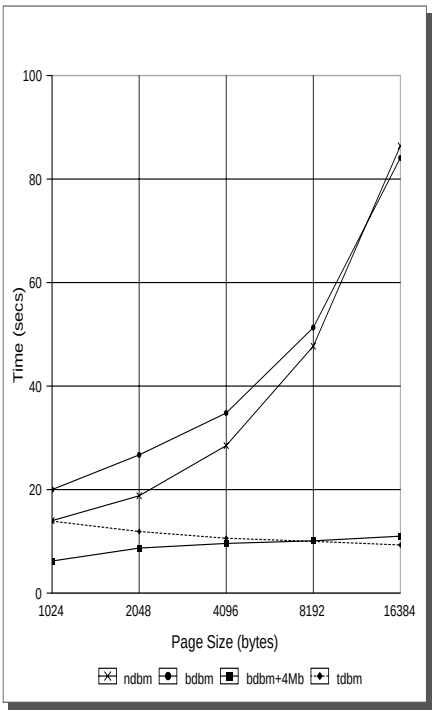

Figure 5: Reading Persistent Databases (`/usr/dict/words`)

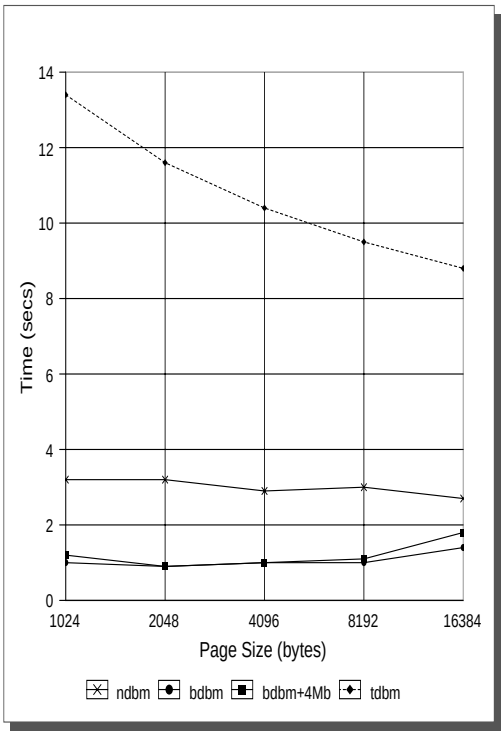

Figure 6: Iterating Through Persistent Databases (`/usr/dict/words`)

`Tdbm` performs well compared to both `ndbm` and `bdbm` in the creation and reading tests. A trial where the data keys were shuffled for the reading test did not yield significantly different results from their sequential use. Because `tdbm` keeps all pages accessed during a transaction in memory, it performs relatively poorly on the iterative key retrieval test. In fact, this operation is almost functionally identical to that used by the sequential reading test so this test was not repeated in the other configurations.

The next set of runs (Figures 7 and 8) repeats the previous set with the larger set of data values in `wordlist`. Here, `tdbm` performs substantially better than both `ndbm` and `bdbm` with the small cache. The last data point in Figure 8 for `bdbm` with 4Mb cache probably is indicative of the performance hit taken when `bdbm` switches to its temporary file after its cache fills.

The last run in the series (Figure 9) shows that `tdbm` performs well in conjunction with large data values. Since `ndbm` cannot load data values larger than its page size, it was excluded from the `rfc` data value runs.

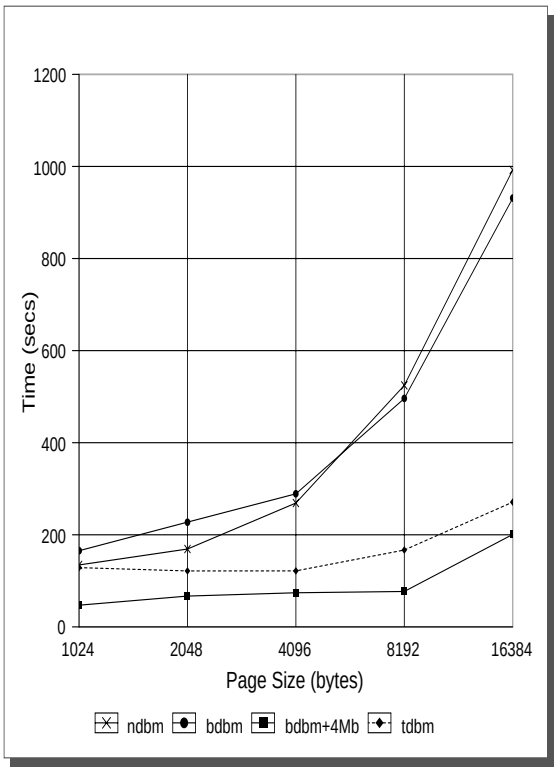

Figure 7: Creating Persistent Databases (`wordlist`)

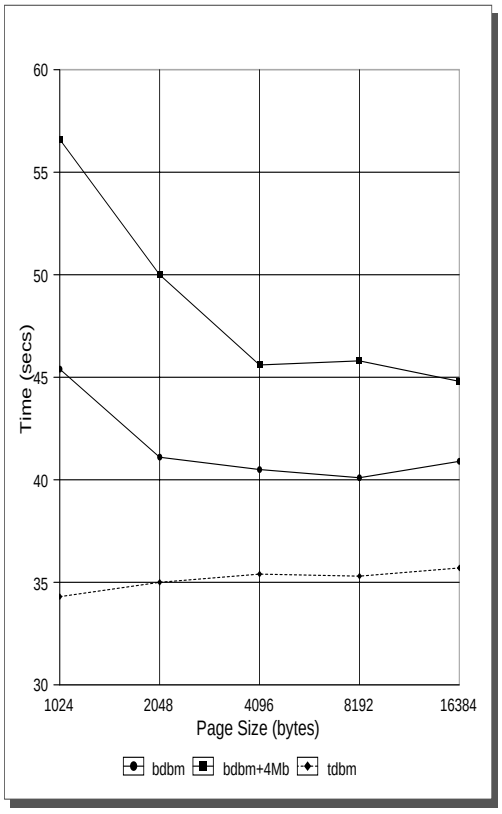

Figure 9: Creating Persistent Databases (`rfc`)

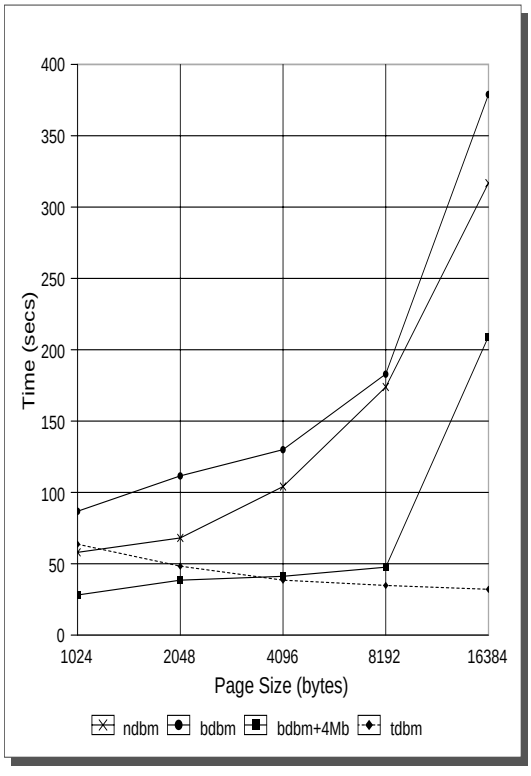

Figure 8: Reading Persistent Databases (`wordlist`)

When creating a new database, all libraries tend to prefer a small page size. The exception is the `rfc` test where `bdbm` does better with a larger page size. While `ndbm` and `bdbm` also prefer a small page size for reading, `tdbm` does better as the page size increases. These observations are also true for volatile databases, discussed in the next section.

Volatile Databases

The second experiment examined the performance of `bdbm` and `tdbm` with volatile databases (`ndbm` doesn't support volatile databases). For both the `/usr/dict/words` and `wordlist` data value sets, `tdbm` performs about the same as for a persistent database. This indicates that for a modestly-sized update, `tdbm`'s commit processing is not expensive. Note that with the small cache, `bdbm`'s performance is slower than for a persistent database. Figure 14 shows that for very large updates, `tdbm` outperforms `bdbm`. `Tdbm`'s commit cost for very large updates can be seen by comparing its performance in Figures 9 and 14.

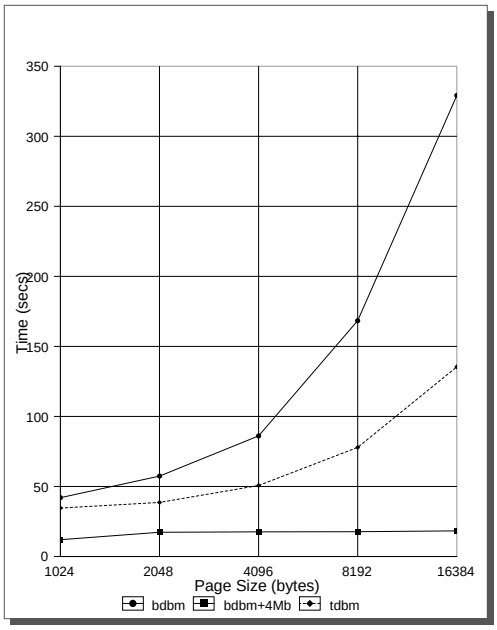

Figure 10: Creating Volatile Databases (`/usr/dict/words`)

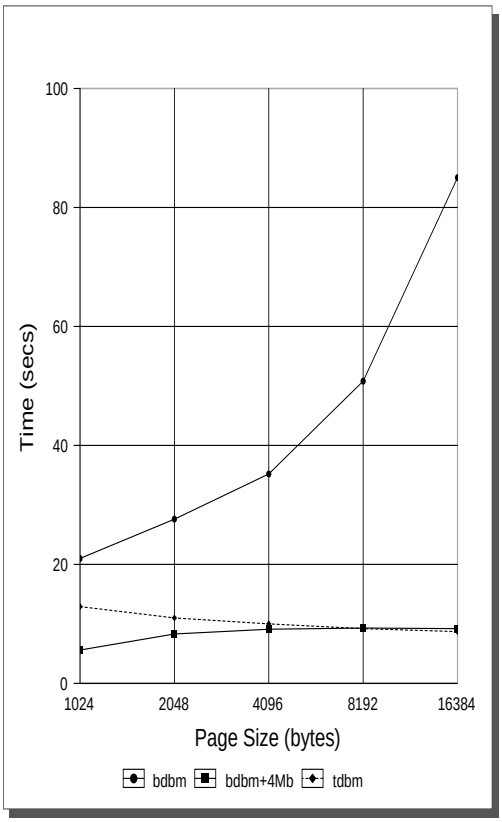

Figure 11: Reading Volatile Databases (`/usr/dict/words`)

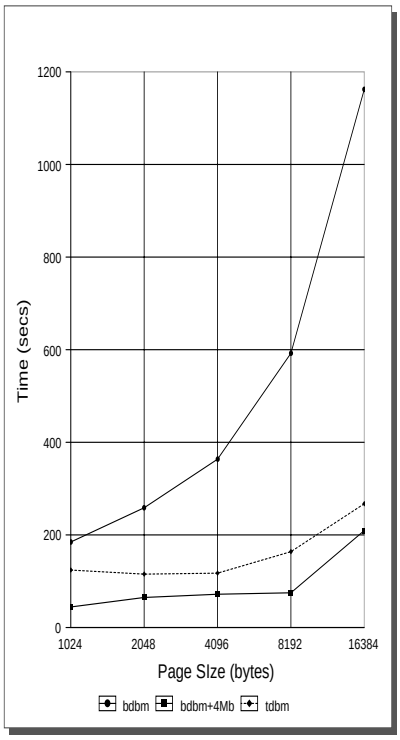

Figure 12: Creating Volatile Databases (`wordlist`)

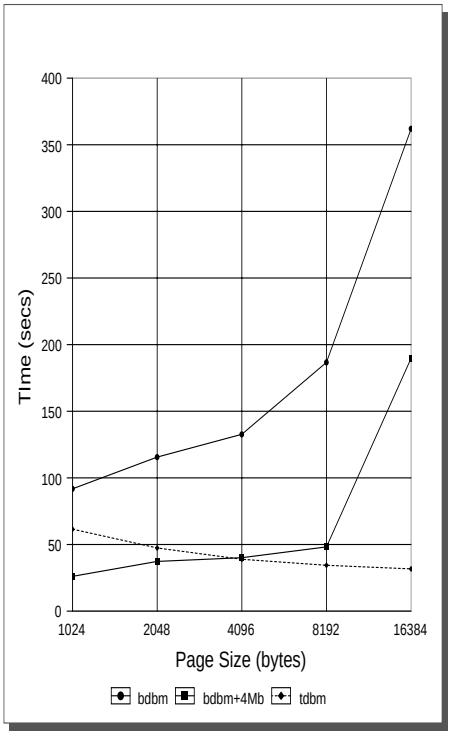

Figure 13: Reading Volatile Databases (`wordlist`)

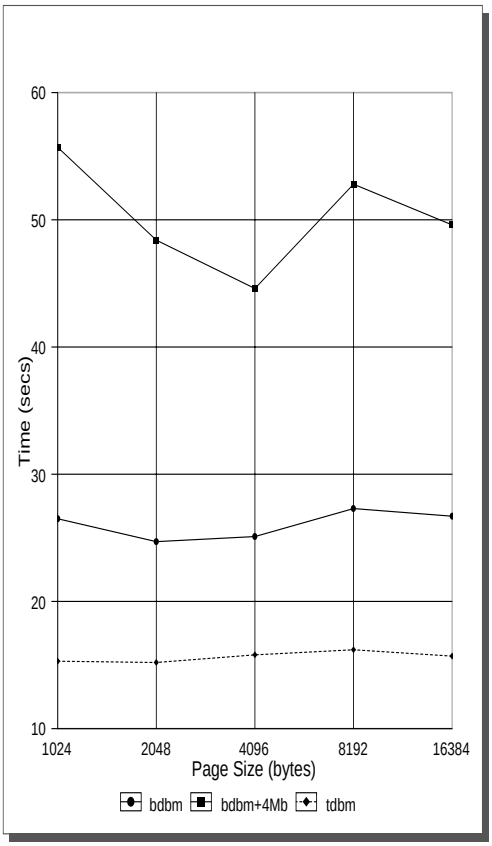

Figure 14: Creating Volatile Databases (`rfc`)

5. Conclusions

The `tdbm` library has been incorporated into the experimental version of the object store used by the EAN X.500 directory service. It has successfully met its design goals as a component of the directory service and, although there is room for improvement, we are satisfied with it. It performs well in comparison with `ndbm` and `bdbm` while providing important new features, such as nested atomic transactions, fault tolerance, and multi-threaded operation.

There is certainly much more that could be done to improve `tdbm`. This might include features such as base page caching (pages that are no longer referenced are currently flushed at top-level commit time), multiple database transactions, multi-volume databases (one database spread over multiple file systems), user selectable page formats and user maintained page formats, and statistics gathering. There is currently no page unsplitting, which would make iterating through the database more efficient after many deletions, but which probably wouldn't reduce the size of the database file under UNIX. Some simpler features may eventually be added, such as arbitrary size keys, user-specifiable hash functions, and deadlock detection, however we currently have no need for them.

Performance evaluation of `tdbm` under Threads, using both real and apparent concurrency, is planned. It might also be interesting to configure `tdbm` to run under SunOS's or Mach's threads. Separating the atomic transaction mechanism into a separate UNIX process could also be examined.

6. Availability

The library will be made available for anonymous ftp from `cs.ubc.ca` (137.82.8.5) as `pub/local/src/tdbm.tar.Z`.

7. Bibliography

[1] AT&T. **dbm(3X)**, *UNIX Programmer's Manual*, Seventh Edition, Volume 1, Bell Laboratories, Jan. 1979.

[2] AT&T. **hsearch(BA_LIB)**, *UNIX System User's Manual*, System V.3, 1985, pp. 506-508.

[3] P. Bernstein, V. Hadzilacos, and N. Goodman. ''Concurrency Control and Recovery in Database Systems'', Addison-Wesley, 1987.

[4] CCITT. ''Recommendation X.500: The Directory – Overview of Concepts, Models and Services'', Dec. 1988.

[5] Computer Systems Research Group, Computer Science Division, EECS. **ndbm(3)**, *4.3BSD UNIX Programmer's Reference Manual (PRM)*, University of California, Berkeley, Apr. 1986.

[6] R. Fagin, J. Nievergelt, N. Pippenger, and H. Strong. ''Extendible Hashing – A Fast Access Method for Dynamic Files'', *ACM Trans. on Database Systems*, Vol. 4, No. 3, (Sept. 1979), pp. 315-344.

[7] T. Haerder and A. Reuter. ''Principles of Transaction-Oriented Database Recovery'', *Computing Surveys*, Vol. 15, No. 4, (Dec. 1983), pp. 287-317.

[8] P. Larson. ''Dynamic Hashing'', *BIT*, Vol. 18, 1978, pp. 184-201.

[9] P. Larson. ''Linear Hashing with Partial Expansions'', *Proc. 6th Int. Conf. on Very Large Data Bases*, 1980, pp. 224-232.

[10] P. Larson. ''Linear Hashing with Separators – A Dynamic Hashing Scheme Achieving One-Access Retrieval'', *ACM Trans. on Database Systems*, Vol. 13, No. 3, (Sept. 1988), pp. 366-388.

[11] W. Litwin. ''Linear Hashing: A New Tool for File and Table Addressing'', *Proc. 6th Int. Conf. on Very Large Data Bases*, 1980, pp. 212-223.

[12] D. Lomet. ''Bounded Index Exponential Hashing'', *ACM Trans. on Database Systems*, Vol. 8, No. 1, (Mar. 1983), pp. 136-165.

[13] J. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*, MIT Press, 1985.

[14] P. Nelson. ''`gdbm` source distribution and README file'', Version 1.5, Feb. 1991.

[15] G. Neufeld, M. Goldberg, and B. Brachman. ''The UBC OSI Distributed Application Programming Environment – User Manual'', Technical Report 90-37, Department of Computer Science, University of British Columbia, Jan. 1991.

[16] G. Neufeld, B. Brachman, M. Goldberg, and D. Stickings. ''The EAN X.500 Directory Service'', Technical Report 91-29, Dept. of Computer Science, University of British Columbia, Nov. 1991.

[17] G. Neufeld and B. Brachman. ''The EAN Object Store'', in preparation, 1992.

[18] B. Salzberg. *File Structures: An Analytic Approach*, Prentice-Hall, 1988.

[19] M. Seltzer and O. Yigit. ''A New Hashing Package for UNIX'', *Proc. of the Winter Usenix Conf.*, Usenix Association, Jan. 1991, pp. 173-184.

[20] M. Seltzer and M. Olson. ''LIBTP: Portable, Modular Transactions for UNIX'', *Proc. of the Winter Usenix Conf.*, Usenix Association, Jan. 1992, pp. 5-25.

[21] D. Severance and G. Lohman. ''Differential Files: Their Application to the Maintenance of Large Databases'', *ACM Trans. on Database Systems*, Vol. 1, No. 3, (Sept. 1976), pp. 256-267.

[22] P. Smith and G. Barnes. *Files & Databases: An Introduction*, Addison-Wesley, 1987.

[23] H. Sturgis, J. Mitchell, and J. Israel. ''Issues in the Design and Use of a Distributed File System'', *Operating Systems Review*, Vol. 14, No. 3, (July 1980), pp. 55-69.

[24] C. Torek. ''Re: dbm.a and ndbm.a archives'', *USENET newsgroup comp.unix*, Apr. 17, 1989. This article discusses the structure of `dbm` databases and the algorithms used by the library.

[25] O. Yigit. ''`sdbm` – Substitute DBM or Berkeley `ndbm` for Every UN*X Made Simple'', `sdbm` source distribution, Dec. 1990.

Author Information

Gerald W. Neufeld is an assistant professor in the Department of Computer Science at the University of British Columbia, Vancouver, B.C. V6T 1Z2. Neufeld is the director of the Open Distributed Systems Group. His interests include computer communications, distributed applications, and distributed operating systems. He is currently working on the Raven project; an object-oriented distributed system. Neufeld received a Ph.D. in computer science from the University of Waterloo in 1987. He received a B.Sc. (Honours) and M.Sc. from the University of Manitoba. Reach him electronically at neufeld@cs.ubc.ca.

Barry Brachman is a research associate and lecturer in the Department of Computer Science at the University of British Columbia. His research interests include distributed systems, computer communications, and operating systems. He has recently been involved in the design and implementation of the EAN X.500 Directory Service and distributed databases and object stores. Dr. Brachman received the B.Sc. Honours degree from the University of Regina in 1981, and the M.Sc. and Ph.D. degrees from the University of British Columbia in 1983 and 1989, respectively, all in Computer Science. Contact him via e-mail at brachman@cs.ubc.ca.