Berkeley DB

Margo Seltzer and Keith Bostic

Conway's Law states that a design reflects the structure of the organization that produced it. Stretching that a bit, we might anticipate that a software artifact designed and initially produced by two people might somehow reflect, not merely the structure of the organization, but the internal biases and philosophies each brings to the table. One of us (Seltzer) has spent her career between the worlds of filesystems and database management systems. If questioned, she'll argue the two are fundamentally the same thing, and furthermore, operating systems and database management systems are essentially both resource managers and providers of convenient abstractions. The differences are "merely" implementation details. The other (Bostic) believes in the tool-based approach to software engineering and in the construction of components based on simpler building blocks, because such systems are invariably superior to monolithic architectures in the important "-bilities": understandability, extensibility, maintainability, testability, and flexibility.

When you combine those two perspectives, it's not surprising to learn that together we spent much of the last two decades working on Berkeley DB—a software library that provides fast, flexible, reliable and scalable data management. Berkeley DB provides much of the same functionality that people expect from more conventional systems, such as relational databases, but packages it differently. For example, Berkeley DB provides fast data access, both keyed and sequential, as well as transaction support and recovery from failure. However, it provides those features in a library that links directly with the application that needs those services, rather than being made available by a standalone server application.

In this chapter, we'll take a deeper look at Berkeley DB and see that it is composed of a collection of modules, each of which embodies the Unix "do one thing well" philosophy. Applications that embed Berkeley DB can use those components directly or they can simply use them implicitly via the more familiar operations to get, put, and delete data items. We'll focus on architecture—how we got started, what we were designing, and where we've ended up and why. Designs can (and certainly will!) be forced to adapt and change—what's vital is maintaining principles and a consistent vision over time. We will also briefly consider the code evolution of long-term software projects. Berkeley DB has over two decades of on-going development, and that inevitably takes its toll on good design.

4.1. In the Beginning

Berkeley DB dates back to an era when the Unix operating system was proprietary to AT&T and there were hundreds of utilities and libraries whose lineage had strict licensing constraints. Margo Seltzer was a graduate student at the University of California, Berkeley, and Keith Bostic was a member of Berkeley's Computer Systems Research Group. At the time, Keith was working on removing AT&T's proprietary software from the Berkeley Software Distribution.

The Berkeley DB project began with the modest goal of replacing the in-memory `hsearch` hash package and the on-disk `dbm/ndbm` hash packages with a new and improved hash implementation able to operate both in-memory and on disk, as well as be freely redistributed without a proprietary license. The `hash` library that Margo Seltzer wrote [SY91] was based on Litwin's Extensible Linear Hashing research. It boasted a clever scheme allowing a constant time mapping between hash values and page addresses, as well as the ability to handle large data—items larger than the underlying hash bucket or filesystem page size, typically four to eight kilobytes.

If hash tables were good, then Btrees and hash tables would be better. Mike Olson, also a graduate student at the University of California, Berkeley, had written a number of Btree implementations, and agreed to write one more. The three of us transformed Margo's hash software and Mike's Btree software into an access-method-agnostic API, where applications reference hash tables or Btrees via database handles that had handle methods to read and modify data.

Building on these two access methods, Mike Olson and Margo Seltzer wrote a research paper ([SO92]) describing LIBTP, a programmatic transactional library that ran in an application's address space.

The hash and Btree libraries were incorporated into the final 4BSD releases, under the name Berkeley DB 1.85. Technically, the Btree access method implements a B+link tree, however, we will use the term Btree for the rest of this chapter, as that is what the access method is called. Berkeley DB 1.85's structure and APIs will likely be familiar to anyone who has used any Linux or BSD-based system.

The Berkeley DB 1.85 library was quiescent for a few years, until 1996 when Netscape contracted with Margo Seltzer and Keith Bostic to build out the full transactional design described in the LIBTP paper and create a production-quality version of the software. This effort produced the first transactional version of Berkeley DB, version 2.0.

The subsequent history of Berkeley DB is a simpler and more traditional timeline: Berkeley DB 2.0 (1997) introduced transactions to Berkeley DB; Berkeley DB 3.0 (1999) was a re-designed version, adding further levels of abstraction and indirection to accommodate growing functionality. Berkeley DB 4.0 (2001) introduced replication and high availability, and Oracle Berkeley DB 5.0 (2010) added SQL support.

At the time of writing, Berkeley DB is the most widely used database toolkit in the world, with hundreds of millions of deployed copies running in everything from routers and browsers to mailers and operating systems. Although more than twenty years old, the Berkeley DB tool-based and object-oriented approach has allowed it to incrementally improve and re-invent itself to match the requirements of the software using it.

Design Lesson 1

It is vital for any complex software package's testing and maintenance that the software be designed and built as a cooperating set of modules with well-defined API boundaries. The boundaries can (and should!) shift as needs dictate, but they always need to be there. The existence of those boundaries prevents the software from becoming an unmaintainable pile of spaghetti. Butler Lampson once said that all

problems in computer science can be solved by another level of indirection. More to the point, when asked what it meant for something to be object-oriented, Lampson said it meant being able to have multiple implementations behind an API. The Berkeley DB design and implementation embody this approach of permitting multiple implementations behind a common interface, providing an object-oriented look and feel, even though the library is written in C.

4.2. Architectural Overview

In this section, we'll review the Berkeley DB library's architecture, beginning with LIBTP, and highlight key aspects of its evolution.

Figure 4.1, which is taken from Seltzer and Olson's original paper, illustrates the original LIBTP architecture, while Figure 4.2 presents the Berkeley DB 2.0 designed architecture.

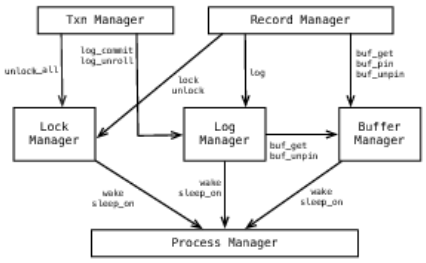

Figure 4.1: Architecture of the LIBTP Prototype System

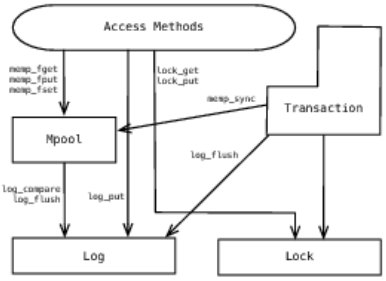

Figure 4.2: Intended Architecture for Berkeley DB-2.0.

The only significant difference between the LIBTP implementation and the Berkeley DB 2.0 design was the removal of the process manager. LIBTP required that each thread of control register itself with the library and then synchronized the individual threads/processes rather than providing subsystem level synchronization. As is discussed in Section 4.4, that original design might have served us better.

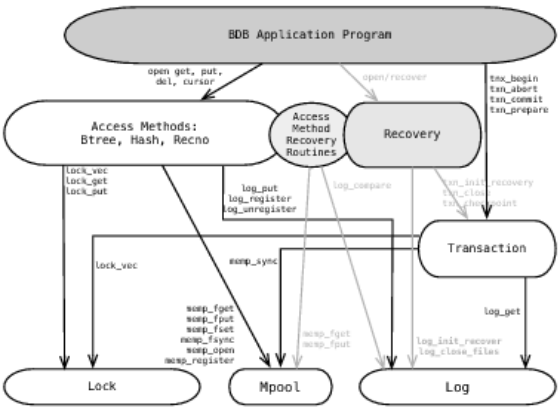

Figure 4.3: Actual Berkeley DB 2.0.6 Architecture.

The difference between the design and the actual released db-2.0.6 architecture, shown in Figure 4.3, illustrates the reality of implementing a robust recovery manager. The recovery subsystem is shown in gray. Recovery includes both the driver infrastructure, depicted in the recovery box, as well as a set of recovery redo and undo routines that recover the operations performed by the access methods. These are represented by the circle labelled "access method recovery routines." There is a consistent design to how recovery is handled in Berkeley DB 2.0 as opposed to hand-coded logging and recovery routines in LIBTP particular to specific access methods. This general purpose design also produces a much richer interface between the various modules.

Figure 4.4 illustrates the Berkeley DB-5.0.21 architecture. The numbers in the diagram reference the APIs listed in the table in Table 4.1. Although the original architecture is still visible, the current architecture shows its age with the addition of new modules, the decomposition of old modules (e.g., `log` has become `log` and `dbreg`), and a significant increase in the number of intermodule APIs).

Over a decade of evolution, dozens of commercial releases, and hundreds of new features later, we see that the architecture is significantly more complex than its ancestors. The key things to note are: First, replication adds an entirely new layer to the system, but it does so cleanly, interacting with the rest of the system via the same APIs as does the historical code. Second, the `log` module is split into `log` and `dbreg` (database registration). This is discussed in more detail in Section 4.8. Third, we have placed all inter-module calls into a namespace identified with leading underscores, so that applications won't collide with our function names. We discuss this further in Design Lesson 6.

Fourth, the logging subsystem's API is now cursor based (there is no `log_get` API; it is replaced by the `log_cursor` API). Historically, Berkeley DB never had more than one thread of control reading or writing the log at any instant in time, so the library had a single notion of the current seek pointer in the log. This was never a good abstraction, but with replication it became unworkable. Just as the application API

supports iteration using cursors, the log now supports iteration using cursors. Fifth, the `fileop` module inside of the access methods provides support for transactionally protected database create, delete, and rename operations. It took us multiple attempts to make the implementation palatable (it is still not as clean as we would like), and after reworking it numerous time, we pulled it out into its own module.

Design Lesson 2

A software design is simply one of several ways to force yourself to think through the entire problem before attempting to solve it. Skilled programmers use different techniques to this end: some write a first version and throw it away, some write extensive manual pages or design documents, others fill out a code template where every requirement is identified and assigned to a specific function or comment. For example, in Berkeley DB, we created a complete set of Unix-style manual pages for the access methods and underlying components before writing any code. Regardless of the technique used, it's difficult to think clearly about program architecture after code debugging begins, not to mention that large architectural changes often waste previous debugging effort. Software architecture requires a different mind set from debugging code, and the architecture you have when you begin debugging is usually the architecture you'll deliver in that release.

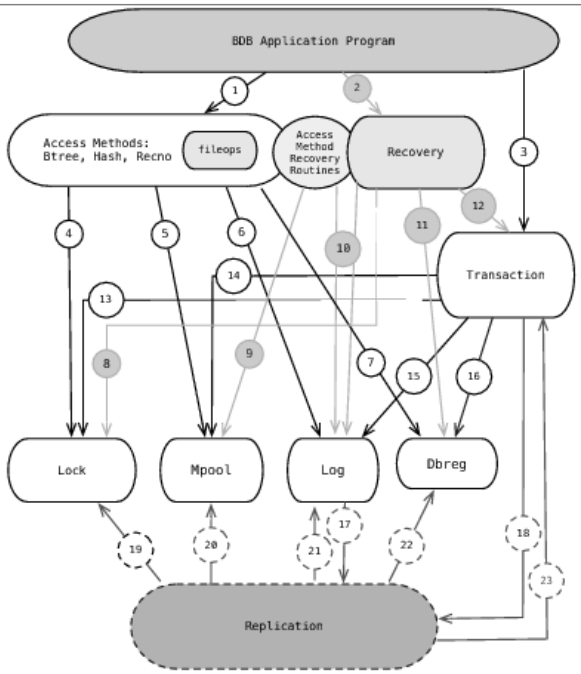

Figure 4.4: Berkeley DB-5.0.21 Architecture

Application APIs		
1. DBP handle operations	**2. DB_ENV Recovery**	**3. Transaction APIs**
open	open(… DB_RECOVER …)	DB_ENV->txn_begin
get		DB_TXN->abort
put		DB_TXN->commit
del		DB_TXN->prepare
cursor		
APIs Used by the Access Methods		
4. Into Lock	**5. Into Mpool**	**6. Into Log**	**7. Into Dbreg**
__lock_downgrade	__memp_nameop	__log_print_record	__dbreg_setup
__lock_vec	__memp_fget		__dbreg_net_id
__lock_get	__memp_fput		__dbreg_revoke

__lock_put	__memp_fset		__dbreg_teardown	
	__memp_fsync		__dbreg_close_id	
	__memp_fopen		__dbreg_log_id	
	__memp_fclose			
	__memp_ftruncate			
	__memp_extend_freelist			
Recovery APIs				
8. Into Lock	**9. Into Mpool**	**10. Into Log**	**11. Into Dbreg**	**12. Into Txn**
__lock_getlocker	__memp_fget	__log_compare	__dbreg_close_files	__txn_getckp
__lock_get_list	__memp_fput	__log_open	__dbreg_mark_restored	__txn_checkpoint
	__memp_fset	__log_earliest	__dbreg_init_recover	__txn_reset
	__memp_nameop	__log_backup		__txn_recycle_id
		__log_cursor		__txn_findlastckp
		__log_vtruncate		__txn_ckp_read
APIs Used by the Transaction Module				
13. Into Lock	**14. Into Mpool**	**15. Into Log**	**16. Into Dbreg**	
__lock_vec	__memp_sync	__log_cursor	__dbreg_invalidate_files	
__lock_downgrade	__memp_nameop	__log_current_lsn	__dbreg_close_files	
			__dbreg_log_files	
API Into the Replication System				
		17. From Log		**18. From Txn**
		__rep_send_message		__rep_lease_check
		__rep_bulk_message		__rep_txn_applied
				__rep_send_message
API From the Replication System				

19. Into Lock	20. Into Mpool	21. Into Log	22. Into Dbreg	23. Into Txn
__lock_vec	__memp_fclose	__log_get_stable_lsn	__dbreg_mark_restored	__txn_recycle_id
__lock_get	__memp_fget	__log_cursor	__dbreg_invalidate_files	__txn_begin
__lock_id	__memp_fput	__log_newfile	__dbreg_close_files	__txn_recover
	__memp_fsync	__log_flush		__txn_getckp
		__log_rep_put		__txn_updateckp
		__log_zero		
		__log_vtruncate		

Table 4.1: Berkeley DB 5.0.21 APIs

Why architect the transactional library out of components rather than tune it to a single anticipated use? There are three answers to this question. First, it forces a more disciplined design. Second, without strong boundaries in the code, complex software packages inevitably degenerate into unmaintainable piles of glop. Third, you can never anticipate all the ways customers will use your software; if you empower users by giving them access to software components, they will use them in ways you never considered.

In subsequent sections we'll consider each component of Berkeley DB, understand what it does and how it fits into the larger picture.

4.3. The Access Methods: Btree, Hash, Recno, Queue

The Berkeley DB access methods provide both keyed lookup of, and iteration over, variable and fixed-length byte strings. Btree and Hash support variable-length key/value pairs. Recno and Queue support record-number/value pairs (where Recno supports variable-length values and Queue supports only fixed-length values).

The main difference between Btree and Hash access methods is that Btree offers locality of reference for keys, while Hash does not. This implies that Btree is the right access method for almost all data sets; however, the Hash access method is appropriate for data sets so large that not even the Btree indexing structures fit into memory. At that point, it's better to use the memory for data than for indexing structures. This trade-off made a lot more sense in 1990 when main memory was typically much smaller than today.

The difference between Recno and Queue is that Queue supports record-level locking, at the cost of requiring fixed-length values. Recno supports variable-length objects, but like Btree and Hash, supports only page-level locking.

We originally designed Berkeley DB such that the CRUD functionality (create, read, update and delete) was key-based and the primary interface for applications. We subsequently added cursors to support iteration. That ordering led to the confusing and wasteful case of largely duplicated code paths inside the library. Over time, this became unmaintainable and we converted all keyed operations to cursor operations (keyed operations now allocate a cached cursor, perform the operation, and return the cursor to the cursor pool). This is an application of one of the endlessly-repeated rules of software development: don't optimize a code path in any way that detracts from clarity and simplicity until you know that it's necessary to do so.

Design Lesson 3

Software architecture does not age gracefully. Software architecture degrades in direct proportion to the number of changes made to the software: bug fixes corrode the layering and new features stress design. Deciding when the software architecture has degraded sufficiently that you should re-design or re-write a module is a hard decision. On one hand, as the architecture degrades, maintenance and development become more difficult and at the end of that path is a legacy piece of software maintainable only by having an army of brute-force testers for every release, because nobody understands how the software works inside. On the other hand, users will bitterly complain over the instability and incompatibilities that result from fundamental changes. As a software architect, your only guarantee is that someone will be angry with you no matter which path you choose.

We omit detailed discussions of the Berkeley DB access method internals; they implement fairly well-known Btree and hashing algorithms (Recno is a layer on top of the Btree code, and Queue is a file block lookup function, albeit complicated by the addition of record-level locking).

4.4. The Library Interface Layer

Over time, as we added additional functionality, we discovered that both applications and internal code needed the same top-level functionality (for example, a table join operation uses multiple cursors to iterate over the rows, just as an application might use a cursor to iterate over those same rows).

Design Lesson 4

It doesn't matter how you name your variables, methods, functions, or what comments or code style you use; that is, there are a large number of formats and styles that are "good enough." What does matter, and matters very much, is that naming and style be consistent. Skilled programmers derive a tremendous amount of information from code format and object naming. You should view naming and style inconsistencies as some programmers investing time and effort to lie to the other programmers, and vice versa. Failing to follow house coding conventions is a firing offense.

For this reason, we decomposed the access method APIs into precisely defined layers. These layers of interface routines perform all of the necessary generic error checking, function-specific error checking, interface tracking, and other tasks such as automatic transaction management. When applications call into Berkeley DB, they call the first level of interface routines based on methods in the object handles. (For example, `__dbc_put_pp`, is the interface call for the Berkeley DB cursor "put" method, to update a data item. The "_pp" is the suffix we use to identify all functions that an application can call.)

One of the Berkeley DB tasks performed in the interface layer is tracking what threads are running inside the Berkeley DB library. This is necessary because some internal Berkeley DB operations may be performed only when no threads are running inside the library. Berkeley DB tracks threads in the library by marking that a thread is executing inside the library at the beginning of every library API and clearing that flag when the API call returns. This entry/exit checking is always performed in the interface layer, as is a similar check to determine if the call is being performed in a replicated environment.

The obvious question is "why not pass a thread identifier into the library, wouldn't that be easier?" The answer is yes, it would be a great deal easier, and we surely wish we'd done just that. But, that change would have modified every single Berkeley DB application, most of every application's calls into Berkeley DB, and in many cases would have required application re-structuring.

Design Lesson 5

Software architects must choose their upgrade battles carefully: users will accept minor changes to upgrade to new releases (if you guarantee compile-time errors, that is, obvious failures until the upgrade is complete; upgrade changes should never fail in subtle ways). But to make truly fundamental changes, you must admit it's a new code base and requires a port of your user base. Obviously, new code bases and application ports are not cheap in time or resources, but neither is angering your user base by telling them a huge overhaul is really a minor upgrade.

Another task performed in the interface layer is transaction generation. The Berkeley DB library supports a mode where every operation takes place in an automatically generated transaction (this saves the application having to create and commit its own explicit transactions). Supporting this mode requires that every time an application calls through the API without specifying its own transaction, a transaction is automatically created.

Finally, all Berkeley DB APIs require argument checking. In Berkeley DB there are two flavors of error checking—generic checks to determine if our database has been corrupted during a previous operation or if we are in the midst of a replication state change (for example, changing which replica allows writes). There are also checks specific to an API: correct flag usage, correct parameter usage, correct option combinations, and any other type of error we can check before actually performing the requested operation.

This API-specific checking is all encapsulated in functions suffixed with `_arg`. Thus, the error checking specific to the cursor `put` method is located in the function `__dbc_put_arg`, which is called by the `__dbc_put_pp` function.

Finally, when all the argument verification and transaction generation is complete, we call the worker method that actually performs the operation (in our example, it would be `__dbc_put`), which is the same function we use when calling the cursor put functionality internally.

This decomposition evolved during a period of intense activity, when we were determining precisely what actions we needed to take when working in replicated environments. After iterating over the code base some non-trivial number of times, we pulled apart all this preamble checking to make it easier to change the next time we identified a problem with it.

4.5. The Underlying Components

There are four components underlying the access methods: a buffer manager, a lock manager, a log manager and a transaction manager. We'll discuss each of them separately, but they all have some common architectural features.

First, all of the subsystems have their own APIs, and initially each subsystem had its own object handle with all methods for that subsystem based on the handle. For example, you could use Berkeley DB's lock manager to handle your own locks or to write your own remote lock manager, or you could use Berkeley DB's buffer manager to handle your own file pages in shared memory. Over time, the subsystem-specific handles were removed from the API in order to simplify Berkeley DB applications. Although the subsystems are still individual components that can be used independently of the other subsystems, they now share a common object handle, the `DB_ENV` "environment" handle. This architectural feature enforces layering and generalization. Even though the layer moves from time-to-time, and there are still a few places where one subsystem reaches across into another subsystem, it is good discipline for programmers to think about the parts of the system as separate software products in their own right.

Second, all of the subsystems (in fact, all Berkeley DB functions) return error codes up the call stack. As a library, Berkeley DB cannot step on the application's name space by declaring global variables, not to mention that forcing errors to return in a single path through the call stack enforces good programmer discipline.

Design Lesson 6

In library design, respect for the namespace is vital. Programmers who use your library should not need to memorize dozens of reserved names for functions, constants, structures, and global variables to avoid naming collisions between an application and the library.

Finally, all of the subsystems support shared memory. Because Berkeley DB supports sharing databases between multiple running processes, all interesting data structures have to live in shared memory. The most significant implication of this choice is that in-memory data structures must use base address and offset pairs instead of pointers in order for pointer-based data structures to work in the context of multiple processes. In other words, instead of indirecting through a pointer, the Berkeley DB library must create a pointer from a base address (the address at which the shared memory segment is mapped into memory) plus an offset (the offset of a particular data structure in that mapped-in segment). To support this feature, we wrote a version of the Berkeley Software Distribution `queue` package that implemented a wide variety of linked lists.

Design Lesson 7

Before we wrote a shared-memory linked-list package, Berkeley DB engineers hand-coded a variety of different data structures in shared memory, and these implementations were fragile and difficult to debug. The shared-memory list package, modeled after the BSD list package (`queue.h`), replaced all of those efforts. Once it was debugged, we never had to debug another shared memory linked-list problem. This illustrates three important design principles: First, if you have functionality that appears more than once, write the shared functions and use them, because the mere existence of two copies of any specific functionality in your code guarantees that one of them is incorrectly implemented. Second, when you develop a set of general purpose routines, write a test suite for the set of routines, so you can debug them in isolation. Third, the harder code is to write, the more important for it to be separately written and maintained; it's almost impossible to keep surrounding code from infecting and corroding a piece of code.

4.6. The Buffer Manager: Mpool

The Berkeley DB Mpool subsystem is an in-memory buffer pool of file pages, which hides the fact that main memory is a limited resource, requiring the library to move database pages to and from disk when handling databases larger than memory. Caching database pages in memory was what enabled the original hash library to significantly out-perform the historic `hsearch` and `ndbm` implementations.

Although the Berkeley DB Btree access method is a fairly traditional B+tree implementation, pointers between tree nodes are represented as page numbers, not actual in-memory pointers, because the library's implementation uses the on-disk format as its in-memory format as well. The advantage of this representation is that a page can be flushed from the cache without format conversion; the disadvantage is that traversing an index structures requires (costlier) repeated buffer pool lookups rather than (cheaper) memory indirections.

There are other performance implications that result from the underlying assumption that the in-memory representation of Berkeley DB indices is really a cache for on-disk persistent data. For example, whenever Berkeley DB accesses a cached page, it first pins the page in memory. This pin prevents any other threads or processes from evicting it from the buffer pool. Even if an index structure fits entirely in the cache and need never be flushed to disk, Berkeley DB still acquires and releases these pins on every access, because the underlying model provided by Mpool is that of a cache, not persistent storage.

4.6.1. The Mpool File Abstraction

Mpool assumes it sits atop a filesystem, exporting the file abstraction through the API. For example, `DB_MPOOLFILE` handles represent an on-disk file, providing methods to get/put pages to/from the file. While Berkeley DB supports temporary and purely in-memory databases, these too are referenced by `DB_MPOOLFILE` handles because of the underlying Mpool abstractions. The `get` and `put` methods are the primary Mpool APIs: `get` ensures a page is present in the cache, acquires a pin on the page and returns a pointer to the page. When the library is done with the page, the `put` call unpins the page, releasing it for eviction. Early versions of Berkeley DB did not differentiate between pinning a page for read access versus pinning a page for write access. However, in order to increase concurrency, we extended the Mpool API to allow callers to indicate their intention to update a page. This ability to distinguish read access from write access was essential to implement multi-version concurrency control. A page pinned for reading that happens to be dirty can be written to disk, while a page pinned for writing cannot, since it may be in an inconsistent state at any instant.

4.6.2. Write-ahead Logging

Berkeley DB uses write-ahead-logging (WAL) as its transaction mechanism to make recovery after failure possible. The term write-ahead-logging defines a policy requiring log records describing any change be propagated to disk *before* the actual data updates they describe. Berkeley DB's use of WAL as its transaction mechanism has important implications for Mpool, and Mpool must balance its design point as a generic caching mechanism with its need to support the WAL protocol.

Berkeley DB writes log sequence numbers (LSNs) on all data pages to document the log record corresponding to the most recent update to a particular page. Enforcing WAL requires that before Mpool writes any page to disk, it must verify that the log record corresponding to the LSN on the page is safely on disk. The design challenge is how to provide this functionality without requiring that all clients of Mpool use a page format identical to that used by Berkeley DB. Mpool addresses this challenge by providing a collection of `set` (and `get`) methods to direct its behavior. The `DB_MPOOLFILE` method `set_lsn_offset` provides a byte offset into a page, indicating where Mpool should look for an LSN to enforce WAL. If the method is never called, Mpool does not enforce the WAL protocol. Similarly, the `set_clearlen` method tells Mpool how many bytes of a page represent metadata that should be explicitly cleared when a page is created in the cache. These APIs allow Mpool to provide the functionality necessary to support Berkeley DB's transactional requirements, without forcing all users of Mpool to do so.

Design Lesson 8

Write-ahead logging is another example of providing encapsulation and layering, even when the functionality is never going to be useful to another piece of software: after all, how many programs care about LSNs in the cache? Regardless, the discipline is useful and makes the software easier to maintain, test, debug and extend.

4.7. The Lock Manager: Lock

Like Mpool, the lock manager was designed as a general-purpose component: a hierarchical lock manager (see [GLPT76]), designed to support a hierarchy of objects that can be locked (such as individual data items), the page on which a data item lives, the file in which a data item lives, or even a collection of files. As we describe the features of the lock manager, we'll also explain how Berkeley DB uses them. However, as with Mpool, it's important to remember that other applications can use the lock manager in completely different ways, and that's OK—it was designed to be flexible and support many different uses.

The lock manager has three key abstractions: a "locker" that identifies on whose behalf a lock is being acquired, a "lock_object" that identifies the item being locked, and a "conflict matrix".

Lockers are 32-bit unsigned integers. Berkeley DB divides this 32-bit name space into transactional and non-transactional lockers (although that distinction is transparent to the lock manager). When Berkeley DB uses the lock manager, it assigns locker IDs in the range 0 to 0x7fffffff to non-transactional lockers and the range 0x80000000 to 0xffffffff to transactions. For example, when an application opens a database, Berkeley DB acquires a long-term read lock on that database to ensure no other thread of control removes or renames it while it is in-use. As this is a long-term lock, it does not belong to any transaction and the locker holding this lock is non-transactional.

Any application using the lock manager needs to assign locker ids, so the lock manager API provides both `DB_ENV->lock_id` and `DB_ENV->lock_id_free` calls to allocate and deallocate lockers. So applications need not implement their own locker ID allocator, although they certainly can.

4.7.1. Lock Objects

Lock objects are arbitrarily long opaque byte-strings that represent the objects being locked. When two different lockers want to lock a particular object, they use the same opaque byte string to reference that object. That is, it is the application's responsibility to agree on conventions for describing objects in terms of opaque byte strings.

For example, Berkeley DB uses a DB_LOCK_ILOCK structure to describe its database locks. This structure contains three fields: a file identifier, a page number, and a type.

In almost all cases, Berkeley DB needs to describe only the particular file and page it wants to lock. Berkeley DB assigns a unique 32-bit number to each database at create time, writes it into the database's metadata page, and then uses it as the database's unique identifier in the Mpool, locking, and logging subsystems. This is the `fileid` to which we refer in the DB_LOCK_ILOCK structure. Not surprisingly, the page number indicates which page of the particular database we wish to lock. When we reference page locks, we set the type field of the structure to DB_PAGE_LOCK. However, we can also lock other types of objects as necessary. As mentioned earlier, we sometimes lock a database handle, which requires a DB_HANDLE_LOCK type. The DB_RECORD_LOCK type lets us perform record level locking in the queue access method, and the DB_DATABASE_LOCK type lets us lock an entire database.

Design Lesson 9

Berkeley DB's choice to use page-level locking was made for good reasons, but we've found that choice to be problematic at times. Page-level locking limits the concurrency of the application as one thread of control modifying a record on a database page will prevent other threads of control from modifying other records on the same page, while record-level locks permit such concurrency as long as the two threads of control are not modifying the same record. Page-level locking enhances stability as it limits the number of recovery paths that are possible (a page is always in one of a couple of states during recovery, as opposed to the infinite number of possible states a page might be in if multiple records are being added and deleted to a page). As Berkeley DB was intended for use as an embedded system where no database administrator would be available to fix things should there be corruption, we chose stability over increased concurrency.

4.7.2. The Conflict Matrix

The last abstraction of the locking subsystem we'll discuss is the conflict matrix. A conflict matrix defines the different types of locks present in the system and how they interact. Let's call the entity holding a lock, the holder and the entity requesting a lock the requester, and let's also assume that the holder and requester have different locker ids. The conflict matrix is an array indexed by `[requester][holder]`, where each entry contains a zero if there is no conflict, indicating that the requested lock can be granted, and a one if there is a conflict, indicating that the request cannot be granted.

The lock manager contains a default conflict matrix, which happens to be exactly what Berkeley DB needs, however, an application is free to design its own lock modes and conflict matrix to suit its own purposes. The only requirement on the conflict matrix is that it is square (it has the same number of rows and columns) and that the application use 0-based sequential integers to describe its lock modes (e.g., read, write, etc.). Table 4.2 shows the Berkeley DB conflict matrix.

	Holder								
Requester	**No-Lock**	**Read**	**Write**	**Wait**	**iWrite**	**iRead**	**iRW**	**uRead**	**wasWrite**
No-Lock									
Read			✓		✓		✓		✓
Write		✓	✓	✓	✓	✓	✓	✓	✓
Wait									
iWrite		✓	✓					✓	✓
iRead			✓						✓
iRW		✓	✓					✓	✓
uRead			✓		✓		✓		
iwasWrite		✓	✓		✓	✓	✓		✓

Table 4.2: Read-Writer Conflict Matrix.

4.7.3. Supporting Hierarchical Locking

Before explaining the different lock modes in the Berkeley DB conflict matrix, let's talk about how the locking subsystem supports hierarchical locking. Hierarchical locking is the ability to lock different items within a containment hierarchy. For example, files contain pages, while pages contain individual elements. When modifying a single page element in a hierarchical locking system, we want to lock just that element; if we were modifying every element on the page, it would be more efficient to simply lock the page, and if we were modifying every page in a file, it would be best to lock the entire file. Additionally, hierarchical locking must understand the hierarchy of the containers because locking a page also says something about locking the file: you cannot modify the file that contains a page at the same time that pages in the file are being modified.

The question then is how to allow different lockers to lock at different hierarchical levels without chaos resulting. The answer lies in a construct called an intention lock. A locker acquires an intention lock on a container to indicate the intention to lock things within that container. So, obtaining a read-lock on a page implies obtaining an intention-to-read lock on the file. Similarly, to write a single page element, you must acquire an intention-to-write lock on both the page and the file. In the conflict matrix above, the `iRead`, `iWrite`, and `iWR` locks are all intention locks that indicate an intention to read, write or do both, respectively.

Therefore, when performing hierarchical locking, rather than requesting a single lock on something, it is necessary to request potentially many locks: the lock on the actual entity as well as intention locks on any containing entities. This need leads to the Berkeley DB `DB_ENV->lock_vec` interface, which takes an array of lock requests and grants them (or rejects them), atomically.

Although Berkeley DB doesn't use hierarchical locking internally, it takes advantage of the ability to specify different conflict matrices, and the ability to specify multiple lock requests at once. We use the default conflict matrix when providing transactional support, but a different conflict matrix to provide simple concurrent access without transaction and recovery support. We use `DB_ENV->lock_vec` to perform lock coupling, a

technique that enhances the concurrency of Btree traversals [Com79]. In lock coupling, you hold one lock only long enough to acquire the next lock. That is, you lock an internal Btree page only long enough to read the information that allows you to select and lock a page at the next level.

Design Lesson 10

Berkeley DB's general-purpose design was well rewarded when we added concurrent data store functionality. Initially Berkeley DB provided only two modes of operation: either you ran without any write concurrency or with full transaction support. Transaction support carries a certain degree of complexity for the developer and we found some applications wanted improved concurrency without the overhead of full transactional support. To provide this feature, we added support for API-level locking that allows concurrency, while guaranteeing no deadlocks. This required a new and different lock mode to work in the presence of cursors. Rather than adding special purpose code to the lock manager, we were able to create an alternate lock matrix that supported only the lock modes necessary for the API-level locking. Thus, simply by configuring the lock manager differently, we were able provide the locking support we needed. (Sadly, it was not as easy to change the access methods; there are still significant parts of the access method code to handle this special mode of concurrent access.)

4.8. The Log Manager: Log

The log manager provides the abstraction of a structured, append-only file. As with the other modules, we intended to design a general-purpose logging facility, however the logging subsystem is probably the module where we were least successful.

Design Lesson 11

When you find an architectural problem you don't want to fix "right now" and that you're inclined to just let go, remember that being nibbled to death by ducks will kill you just as surely as being trampled by elephants. Don't be too hesitant to change entire frameworks to improve software structure, and when you make the changes, don't make a partial change with the idea that you'll clean up later—do it all and then move forward. As has been often repeated, "If you don't have the time to do it right now, you won't find the time to do it later." And while you're changing the framework, write the test structure as well.

A log is conceptually quite simple: it takes opaque byte strings and writes them sequentially to a file, assigning each a unique identifier, called a log sequence number (LSN). Additionally, the log must provide efficient forward and backward traversal and retrieval by LSN. There are two tricky parts: first, the log must guarantee it is in a consistent state after any possible failure (where consistent means it contains a contiguous sequence of uncorrupted log records); second, because log records must be written to stable storage for transactions to commit, the performance of the log is usually what bounds the performance of any transactional application.

As the log is an append-only data structure, it can grow without bound. We implement the log as a collection of sequentially numbered files, so log space may be reclaimed by simply removing old log files. Given the multi-file architecture of the log, we form LSNs as pairs specifying a file number and offset within the file. Thus, given an LSN, it is trivial for the log manager to locate the record: it seeks to the given offset of the given log file and returns the record written at that location. But how does the log manager know how many bytes to return from that location?

4.8.1. Log Record Formatting

The log must persist per-record metadata so that, given an LSN, the log manager can determine the size of the record to return. At a minimum, it needs to know the length of the record. We prepend every log record with a log record header containing the record's length, the offset of the previous record (to facilitate backward traversal), and a checksum for the log record (to identify log corruption and the end of the log file). This metadata is sufficient for the log manager to maintain the sequence of log records, but it is not sufficient to actually implement recovery; that functionality is encoded in the contents of log records and in how Berkeley DB uses those log records.

Berkeley DB uses the log manager to write before- and after-images of data before updating items in the database [HR83]. These log records contain enough information to either redo or undo operations on the database. Berkeley DB then uses the log both for transaction abort (that is, undoing any effects of a transaction when the transaction is discarded) and recovery after application or system failure.

In addition to APIs to read and write log records, the log manager provides an API to force log records to disk (`DB_ENV->log_flush`). This allows Berkeley DB to implement write-ahead logging—before evicting a page from Mpool, Berkeley DB examines the LSN on the page and asks the log manager to guarantee that the specified LSN is on stable storage. Only then does Mpool write the page to disk.

Design Lesson 12

Mpool and Log use internal handle methods to facilitate write-ahead logging, and in some cases, the method declaration is longer than the code it runs, since the code is often comparing two integral values and nothing more. Why bother with such insignificant methods, just to maintain consistent layering? Because if your code is not so object-oriented as to make your teeth hurt, it is not object-oriented enough. Every piece of code should do a small number of things and there should be a high-level design encouraging programmers to build functionality out of smaller chunks of functionality, and so on. If there's anything we have learned about software development in the past few decades, it is that our ability to build and maintain significant pieces of software is fragile. Building and maintaining significant pieces of software is difficult and error-prone, and as the software architect, you must do everything that you can, as early as you can, as often as you can, to maximize the information conveyed in the structure of your software.

Berkeley DB imposes structure on the log records to facilitate recovery. Most Berkeley DB log records describe transactional updates. Thus, most log records correspond to page modifications to a database, performed on behalf of a transaction. This description provides the basis for identifying what metadata Berkeley DB must attach to each log record: a database, a transaction, and a record type. The transaction identifier and record type fields are present in every record at the same location. This allows the recovery system to extract a record type and dispatch the record to an appropriate handler that can interpret the record and perform appropriate actions. The transaction identifier lets the recovery process identify the transaction to which a log record belongs, so that during the various stages of recovery, it knows whether the record can be ignored or must be processed.

4.8.2. Breaking the Abstraction

There are also a few "special" log records. Checkpoint records are, perhaps, the most familiar of those special records. Checkpointing is the process of making the on-disk state of the database consistent as of some point in time. In other words, Berkeley DB aggressively caches database pages in Mpool for performance. However, those pages must eventually get written to disk and the sooner we do so, the more quickly we will be able to recover in the case of application or system failure. This implies a trade-off between the frequency of checkpointing and the length of recovery: the more frequently a system takes checkpoints, the more quickly it will be able to recover. Checkpointing is a transaction

function, so we'll describe the details of checkpointing in the next section. For the purposes of this section, we'll talk about checkpoint records and how the log manager struggles between being a stand-alone module and a special-purpose Berkeley DB component.

In general, the log manager, itself, has no notion of record types, so in theory, it should not distinguish between checkpoint records and other records—they are simply opaque byte strings that the log manager writes to disk. In practice, the log maintains metadata revealing that it does understand the contents of some records. For example, during log startup, the log manager examines all the log files it can find to identify the most recently written log file. It assumes that all log files prior to that one are complete and intact, and then sets out to examine the most recent log file and determine how much of it contains valid log records. It reads from the beginning of a log file, stopping if/when it encounters a log record header that does not checksum properly, which indicates either the end of the log or the beginning of log file corruption. In either case, it determines the logical end of log.

During this process of reading the log to find the current end, the log manager extracts the Berkeley DB record type, looking for checkpoint records. It retains the position of the last checkpoint record it finds in log manager metadata as a "favor" to the transaction system. That is, the transaction system needs to find the last checkpoint, but rather than having both the log manager and transaction manager read the entire log file to do so, the transaction manager delegates that task to the log manager. This is a classic example of violating abstraction boundaries in exchange for performance.

What are the implications of this tradeoff? Imagine that a system other than Berkeley DB is using the log manager. If it happens to write the value corresponding to the checkpoint record type in the same position that Berkeley DB places its record type, then the log manager will identify that record as a checkpoint record. However, unless the application asks the log manager for that information (by directly accessing `cached_ckp_lsn` field in the log metadata), this information never affects anything. In short, this is either a harmful layering violation or a savvy performance optimization.

File management is another place where the separation between the log manager and Berkeley DB is fuzzy. As mentioned earlier, most Berkeley DB log records have to identify a database. Each log record could contain the full filename of the database, but that would be expensive in terms of log space, and clumsy, because recovery would have to map that name to some sort of handle it could use to access the database (either a file descriptor or a database handle). Instead, Berkeley DB identifies databases in the log by an integer identifier, called a log file id, and implements a set of functions, called `dbreg` (for "database registration"), to maintain mappings between filenames and log file ids. The persistent version of this mapping (with the record type `DBREG_REGISTER`) is written to log records when the database is opened. However, we also need in-memory representations of this mapping to facilitate transaction abort and recovery. What subsystem should be responsible for maintaining this mapping?

In theory, the file to log-file-id mapping is a high-level Berkeley DB function; it does not belong to any of the subsystems, which were intended to be ignorant of the larger picture. In the original design, this information was left in the logging subsystems data structures because the logging system seemed like the best choice. However, after repeatedly finding and fixing bugs in the implementation, the mapping support was pulled out of the logging subsystem code and into its own small subsystem with its own object-oriented interfaces and private data structures. (In retrospect, this information should logically have been placed with the Berkeley DB environment information itself, outside of any subsystem.)

Design Lesson 13

There is rarely such thing as an unimportant bug. Sure, there's a typo now and then, but usually a bug implies somebody didn't fully understand what they were doing and implemented the wrong thing. When you fix a bug, don't look for the symptom: look for the underlying cause, the misunderstanding, if you will, because that leads to a better understanding of the program's architecture as well as revealing fundamental underlying flaws in the design itself.

4.9. The Transaction Manager: Txn

Our last module is the transaction manager, which ties together the individual components to provide the transactional ACID properties of atomicity, consistency, isolation, and durability. The transaction manager is responsible for beginning and completing (either committing or aborting) transactions, coordinating the log and buffer managers to take transaction checkpoints, and orchestrating recovery. We'll visit each of these areas in order.

Jim Gray invented the ACID acronym to describe the key properties that transactions provide [Gra81]. Atomicity means that all the operations performed within a transaction appear in the database in a single unit—they either are all present in the database or all absent. Consistency means that a transaction moves the database from one logically consistent state to another. For example, if the application specifies that all employees must be assigned to a department that is described in the database, then the consistency property enforces that (with properly written transactions). Isolation means that from the perspective of a transaction, it appears that the transaction is running sequentially without any concurrent transactions running. Finally, durability means that once a transaction is committed, it stays committed—no failure can cause a committed transaction to disappear.

The transaction subsystem enforces the ACID properties, with the assistance of the other subsystems. It uses traditional transaction begin, commit, and abort operations to delimit the beginning and ending points of a transaction. It also provides a prepare call, which facilitates two phase commit, a technique for providing transactional properties across distributed transactions, which are not discussed in this chapter. Transaction begin allocates a new transaction identifier and returns a transaction handle, `DB_TXN`, to the application. Transaction commit writes a commit log record and then forces the log to disk (unless the application indicates that it is willing to forego durability in exchange for faster commit processing), ensuring that even in the presence of failure, the transaction will be committed. Transaction abort reads backwards through the log records belonging to the designated transaction, undoing each operation that the transaction had done, returning the database to its pre-transaction state.

4.9.1. Checkpoint Processing

The transaction manager is also responsible for taking checkpoints. There are a number of different techniques in the literature for taking checkpoints [HR83]. Berkeley DB uses a variant of fuzzy checkpointing. Fundamentally, checkpointing involves writing buffers from Mpool to disk. This is a potentially expensive operation, and it's important that the system continues to process new transactions while doing so, to avoid long service disruptions. At the beginning of a checkpoint, Berkeley DB examines the set of currently active transactions to find the lowest LSN written by any of them. This LSN becomes the checkpoint LSN. The transaction manager then asks Mpool to flush its dirty buffers to disk; writing those buffers might trigger log flush operations. After all the buffers are safely on disk, the transaction manager then writes a checkpoint record containing the checkpoint LSN. This record states that all the operations described by log records before the checkpoint LSN are now safely on disk. Therefore, log records prior to the checkpoint LSN are no longer necessary for recovery. This has two implications: First, the system can

reclaim any log files prior to the checkpoint LSN. Second, recovery need only process records after the checkpoint LSN, because the updates described by records prior to the checkpoint LSN are reflected in the on-disk state.

Note that there may be many log records between the checkpoint LSN and the actual checkpoint record. That's fine, since those records describe operations that logically happened after the checkpoint and that may need to be recovered if the system fails.

4.9.2. Recovery

The last piece of the transactional puzzle is recovery. The goal of recovery is to move the on-disk database from a potentially inconsistent state to a consistent state. Berkeley DB uses a fairly conventional two-pass scheme that corresponds loosely to "relative to the last checkpoint LSN, undo any transactions that never committed and redo any transactions that did commit." The details are a bit more involved.

Berkeley DB needs to reconstruct its mapping between log file ids and actual databases so that it can redo and undo operations on the databases. The log contains a full history of `DBREG_REGISTER` log records, but since databases stay open for a long time and we do not want to require that log files persist for the entire duration a database is open, we'd like a more efficient way to access this mapping. Prior to writing a checkpoint record, the transaction manager writes a collection of `DBREG_REGISTER` records describing the current mapping from log file ids to databases. During recovery, Berkeley DB uses these log records to reconstruct the file mapping.

When recovery begins, the transaction manager probes the log manager's `cached_ckp_lsn` value to determine the location of the last checkpoint record in the log. This record contains the checkpoint LSN. Berkeley DB needs to recover from that checkpoint LSN, but in order to do so, it needs to reconstruct the log file id mapping that existed at the checkpoint LSN; this information appears in the checkpoint *prior* to the checkpoint LSN. Therefore, Berkeley DB must look for the last checkpoint record that occurs before the checkpoint LSN. Checkpoint records contain, not only the checkpoint LSN, but the LSN of the previous checkpoint to facilitate this process. Recovery begins at the most recent checkpoint and using the `prev_lsn` field in each checkpoint record, traverses checkpoint records backwards through the log until it finds a checkpoint record appearing before the checkpoint LSN. Algorithmically:

```
ckp_record = read (cached_ckp_lsn)
ckp_lsn = ckp_record.checkpoint_lsn
cur_lsn = ckp_record.my_lsn
while (cur_lsn > ckp_lsn) {
    ckp_record = read (ckp_record.prev_ckp)
    cur_lsn = ckp_record.my_lsn
}
```

Starting with the checkpoint selected by the previous algorithm, recovery reads sequentially until the end of the log to reconstruct the log file id mappings. When it reaches the end of the log, its mappings should correspond exactly to the mappings that existed when the system stopped. Also during this pass, recovery keeps track of any transaction commit records encountered, recording their transaction identifiers. Any transaction for which log records appear, but whose transaction identifier does not appear in a transaction commit record, was either aborted or never completed and should be treated as aborted. When recovery reaches the end of the log, it reverses direction and begins reading backwards through the log. For each transactional log record encountered, it extracts the transaction identifier and consults the list of transactions that have committed, to determine if this record should be undone. If it finds that the transaction identifier does not belong to a committed transaction, it extracts the record type and calls a recovery routine for that log record, directing it to undo the operation described. If the record belongs to a committed transaction, recovery ignores it on the backwards pass. This backward pass continues all the way back to the checkpoint LSN[1]. Finally, recovery reads the log one last time in the forward direction, this time redoing any log records belonging to committed transactions. When this final pass completes, recovery takes a checkpoint. At this point, the database is fully consistent and ready to begin running the application.

Thus, recovery can be summarized as:

1. Find the checkpoint prior to the checkpoint LSN in the most recent checkpoint
2. Read forward to restore log file id mappings and construct a list of committed transactions
3. Read backward to the checkpoint LSN, undoing all operations for uncommitted transactions
4. Read forward, redoing all operations for committed transactions
5. Checkpoint

In theory, the final checkpoint is unnecessary. In practice, it bounds the time for future recoveries and leaves the database in a consistent state.

Design Lesson 14

Database recovery is a complex topic, difficult to write and harder to debug because recovery simply shouldn't happen all that often. In his Turing Award Lecture, Edsger Dijkstra argued that programming was inherently difficult and the beginning of wisdom is to admit we are unequal to the task. Our goal as architects and programmers is to use the tools at our disposal: design, problem decomposition, review, testing, naming and style conventions, and other good habits, to constrain programming problems to problems we *can* solve.

4.10. Wrapping Up

Berkeley DB is now over twenty years old. It was arguably the first general-purpose transactional key/value store and is the grandfather of the NoSQL movement. Berkeley DB continues as the underlying storage system for hundreds of commercial products and thousands of Open Source applications (including SQL, XML and NoSQL engines) and has millions of deployments across the globe. The lessons we've learned over the course of its development and maintenance are encapsulated in the code and summarized in the design tips outlined above. We offer them in the hope that other software designers and architects will find them useful.

Footnotes

1. Note that we only need to go backwards to the checkpoint LSN, not the checkpoint record preceding it.