Life After BerkeleyDB: OpenLDAP's Memory-Mapped Database

Howard Chu

Symas Corp., OpenLDAP Project

hyc@symas.com, hyc@openldap.org

http://www.symas.com, http://www.openldap.org

Abstract

OpenLDAP's new MDB library is a highly optimized B+tree implementation that is orders of magnitude faster and more efficient than everything else in the software world. Reads scale perfectly linearly across arbitrarily many CPUs with no bottlenecks, and data is returned with zero memcpy's. Writes are on average twenty times faster than commonly available databases such as SQLite. The entire library compiles down to only 32K of object code, allowing it to execute completely inside a typical CPU's L1 cache. Backends for OpenLDAP slapd, Cyrus SASL, Heimdal Kerberos, SQLite 3, and OpenDKIM have already been written, with other projects in progress.

1. Introduction

The MDB library was written specifically for OpenLDAP as a result of over a decade of experience working with BerkeleyDB (Oracle BerkeleyDB "BDB"[1]). BDB's performance characteristics were too slow in its native form, and the caching we introduced on top of it made configuration quite complex. Also, while a lot of interesting new research has been published in the database field, none of these ideas were filtering into the BDB code base. It was apparent that BDB was holding us back and a simpler solution was needed.

The MDB library is fully transactional and implements B+ trees[2] with Multi-Version Concurrency Control[3]. The entire database is mapped into virtual memory and all data fetches are performed via direct access to the mapped memory instead of through intermediate buffers and copies.

2. Background

Before describing the improvements offered by the MDB design, an overview of the existing BDB-based backends (back-bdb and back-hdb) will be presented.

LDAP and BDB have a long history together; Netscape commissioned the 2.0 release of BDB specifically for use in their LDAP server[4]. The OpenLDAP Project's first release using the BDB-specific APIs was OpenLDAP 2.1 in June 2002. Since BDB maintains its own internal cache, it was hoped that the back-bdb backend could be deployed without any backend-level caching, but early benchmark results showed that retrieving entries directly from the database on every query was too slow. Despite radical improvements in entry fetch and decoding speed[5], the decision was made to introduce an entry cache for the backend, and the cache management problems grew from there.

These problems include:

- Multiple caches that each need to be carefully configured. On top of the BDB cache, there are caches for entries, DNs, and attribute indexing in the backend. All of these waste memory since the same data may be present in three places - the filesystem cache, the BDB cache, and the backend caches. Configuration is a tedious job because each cache layer has different size and speed characteristics and it is difficult to strike a balance that is optimal for all use cases.
- Caches with very complex lock dependencies. For speed, most of the backend caches are protected by simple mutexes. However, when interacting with the BDB API, these mutexes must be dropped and exchanged for (much slower) database locks. Otherwise deadlocks which could not be detected by BDB's deadlock detector may occur. Deadlocks occur very frequently in routine operation of the backend.
- Caches with pathological behavior if they were smaller than the whole database. When the cache size was small enough that a significant number of queries were not being satisfied from the cache, extreme heap fragmentation was observed[6], as the cache freed existing entries to make room for new entries. The fragmentation would cause the size of the slapd process to rapidly grow, defeating the purpose of setting a small cache size. The problem was worst with the memory allocator in GNU libc[7], and could be mitigated by using alternatives such as Hoard[8] or Google tcmalloc[9], but additional changes were made in slapd to reduce the number of calls to malloc() and free() to delay the onset of this fragmentation issue[10].
- Caches with very low effectiveness. When multiple queries arrive whose result sets are larger than the entry cache, the cache effectiveness drops to zero because entries are constantly being freed before they ever get any chance of being re-used[11]. A great deal of effort was expended exploring more advanced cache replacement algorithms to combat this problem[12][13].

From the advent of the back-bdb backend until the present time, the majority of development and debugging effort in these backends has all been devoted to backend cache management. The present state of affairs is difficult to configure, difficult to optimize, and extremely labor intensive to maintain.

Another issue relates to administrative overhead in general. For example, BDB uses write-ahead logs for its transaction support. These logs are written before database updates are performed, so that in case an update is interrupted or aborted, sufficient information is present to undo the updates and return the database to the state it was in before the update began. The log files grow continuously as updates are made to a database, and can only be removed after an expensive checkpoint operation is performed. Later versions of BDB added an auto-remove option to delete old log files automatically, but if the system crashed while this option was in use, generally the database could not be recovered successfully because the necessary logs had been deleted.

3. Solutions

The problems with back-bdb and back-hdb can be summed up in two main areas: cache management, and lock management. The approach to a solution with back-mdb is simple - do no caching, and do no locking. The other issues of administrative overhead are handled as side-effects of the main solutions.

3.1 Eliminating Caching

One fundamental concept behind the MDB approach is known as "Single-Level Store"[14]. The basic idea is to treat all of computer memory as a single address space. Pages of storage may reside in primary storage (RAM) or in secondary storage (disk) but the actual location is unimportant to the application. If a referenced page is currently in primary storage the application can use it immediately, if not a page fault occurs and the operating system brings the page into primary storage. The concept was introduced in 1964 in the Multics[15] operating system but was generally abandoned by the early 1990s as data volumes surpassed the capacity of 32 bit address spaces. (We last knew of it in the Apollo DOMAIN[16] operating system, though many other Multics-influenced designs carried it on.) With the ubiquity of 64 bit processors today this concept can again be put to good use. (Given a virtual address space limit of 63 bits that puts the upper bound of database size at 8 exabytes. Commonly available processors today only implement 48 bit address spaces, limiting us to 47 bits or 128 terabytes.)

Another operating system requirement for this approach to be viable is a Unified Buffer Cache. While most POSIX-based operating systems have supported an mmap() system call for many years, their initial implementations kept memory managed by the VM subsystem separate from memory managed by the filesystem cache. This was not only wasteful (again, keeping data cached in two places at once) but also led to coherency problems - data modified through a memory map was not visible using filesystem read() calls, or data modified through a filesystem write() was not visible in the memory map. Most modern operating systems now have filesystem and VM paging unified, so this should not be a concern in most deployments[17][18][19].

The MDB library is designed to access an entire database thru a single read-only memory map. Keeping the mapping read-only prevents stray writes from buggy application code from corrupting the database. Updates are performed using regular write() calls. (Updating through the map would be difficult anyway since files cannot be grown through map references; only updates to existing pages could be done through the map. For simplicity all updates are done using write() and it doesn't matter whether the update grows the file or not.) This update approach requires that the filesystem and VM views are kept coherent, thus the requirement that the OS uses a Unified Buffer Cache.

The memory-mapped approach makes full use of the operating system's filesystem cache, and eliminates any database-level caching. Likewise the back-mdb backend performs no caching of its own; it uses information from the database directly. Using the memory-mapped

data thus eliminates two levels of caching relative to back-hdb, as well as eliminating redundant memcpy() operations between those caches. It also eliminates all cache tuning/configuration issues, thus easing deployment for administrators.

Of course, by eliminating caching, one would expect to incur a significant performance hit. It should be much faster to dump out the contents of a cached, fully decoded entry in response to a search request, than to read the entry in from disk and decode it on every request. Early results with back-mdb showed this to be true, but further optimization in back-mdb has mostly eliminated this performance hit.

3.2 Eliminating Locking

The other fundamental concept behind MDB is the use of Multi-Version Concurrency Control (MVCC). The basic idea is that updates of data never overwrite existing data; instead the updates write to new pages and thus create a new version of the database. Readers only ever see the snapshot of the database as it existed when a read transaction began, so they are fully isolated from writers. Because of this isolation read accesses require no locks, they always have a self-consistent view of the database.

BDB has supported MVCC since version 4.5.20, but because of the caching layer in back-bdb/hdb there was no benefit to using it. The only way to get any gain from using MVCC was to also eliminate the backend caching layer, and without the caching layer back-bdb/hdb's performance would be too slow because data lookups in BDB were still too slow.

A major downside of MVCC-based systems is that since they always write new data to new disk pages, the database files tend to grow without bound. They need periodic compaction or garbage collection in order to keep their disk usage constrained, and the required frequency of such compaction efforts is very high on databases with high update rates. Additionally, systems based on garbage collection generally require twice as much disk space as the actual data occupies. Also, in order to sustain a write rate of N operations/second, the I/O system must actually support >>2N operations/second, since the compaction task needs to run faster than the normal write task in order to catch up and actually complete its job, and the volume of data already written always exceeds the volume being written. If this over-provisioning of I/O resources cannot be guaranteed, then the typical solution to this problem is to deny updates while compaction is being performed.

Causing a service outage for writes while garbage collection is performed is unacceptable, so MDB uses a different approach. Within a given MDB database environment, MDB maintains two B+tree structures - one containing application data, and another one containing a free list with the IDs of pages that are no longer in use. Tracking the in-use status is typically done with reference counters and other such mechanisms that require locking. Obviously the use of locking would defeat the purpose of using MVCC in the first place, so a lockless solution was designed instead. With this solution, pages that are no longer in use by any active snapshot of the database are re-used by updaters, so the database size remains relatively static. This is a key advantage of MDB over other well-known MVCC databases such as CouchDB[20].

4. Implementation Highlights

The MDB library API was loosely modeled after the BDB API, to ease migration of BDB-based code. The first cut of the back-mdb code was simply copied from the back-bdb source tree, and then all references to the caching layers were deleted. After a few minor API differences were accounted for, the backend was fully operational (though still in need of optimization). As of today back-mdb comprises 340KB of source code, compared to 476KB for back-bdb/hdb, so back-mdb is approximately 30% smaller. Development was rapid; the MDB library code was feature-complete August 31, 2011 at which point coding of back-mdb began. Back-mdb was feature-complete 5 days later. This demonstrates the ease of porting from the BDB API.

The MDB code itself started from Martin Hedenfalk's append-only Btree code in the OpenBSD ldapd source repository[21]. The first cut of the MDB code was simply copied from the ldapd source, and then all of the Btree page cache manager was deleted and replaced with mmap accesses. The original Btree source yielded an object file of 39KB; the MDB version was 32KB. Initial testing with the append-only code proved that approach to be completely impractical. With a small test database and only a few hundred add/delete operations, the DB occupied 1027 pages but only 10 pages actually contained current data; over 99% of the space was wasted.

Along with the mmap management and page reclamation, many other significant changes were made to arrive at the current MDB library, mostly to add features from BDB that back-mdb would need. As of today the MDB library still comprises only 32KB of object code. (Comparing source code is not very informative since the MDB source code has been heavily expanded with Doxygen comments. The initial version of mdb.c was 59KB as opposed to btree.c at 76KB but with full documentation embedded mdb.c is now 162KB. Also for comparison, BDB is now over 1.5MB of object code.)

4.1 MDB Change Summary

The append-only Btree code used a meta page at the end of the database file to point at the current root node of the Btree. New pages were always written out sequentially at the end of the file, followed by a new meta page upon transaction commit. Any application opening the database needed to search backward from the end of the file to find the most recent meta page, to get a current snapshot of the database. (Further explanation of append-only operation is available at Martin's web site[22].)

In MDB there are two meta pages occupying page 0 and page 1 of the file. They are used alternately by transactions. Each meta page points to the root node of two Btrees - one for the free list and one for the application data. New data first re-uses any available pages from the free list, then writes sequentially at the end of the file if no free pages are available. Then the older meta page is written on transaction commit. This is nothing more than standard double-buffering - any application opening the database uses the newer meta page, while a committer overwrites the older one. No locks are needed to protect readers from writers; readers are guaranteed to always see a valid root node.

The original code only supported a single Btree in a given database file. For MDB we wanted to support multiple trees in a single database file. The back-mdb indexing code uses individual databases for each attribute index, and it would be a non-starter to require a sysadmin to configure multiple mmap regions for a single back-mdb instance. Additionally, the indexing code uses BDB's sorted duplicate feature, which allows multiple data items with the same key to be stored in a Btree, and this feature needed to be added to MDB as well. These features were both added using a subdatabase mechanism, which allows a data item in a Btree to be treated as the root node of another Btree.

4.2 Locking

For simplicity the MDB library allows only one writer at a time. Creating a write transaction acquires a lock on a writer mutex; the mutex normally resides in a shared memory region so that it can be shared between multiple processes. This shared memory is separate from the region occupied by the main database. The lock region also contains a table with one slot for every active reader in the database. The slots record the reader's process and thread ID, as well as the ID of the transaction snapshot the reader is using. (The process and thread ID are recorded to allow detection of stale entries in the table, e.g. threads that exited without releasing their reader slot.) The table is constructed in processor cache-aligned memory such that False Sharing[23] of cache lines is avoided. (On Windows a named mutex is used instead. Likewise, on platforms like MacOSX and FreeBSD that don't support POSIX process-shared mutexes, a named semaphore is used.)

Readers acquire a slot the first time a thread opens a read transaction. Acquiring an empty slot in the table requires locking a mutex on the table. The slot address is saved in thread-local storage and re-used the next time the thread opens a read transaction, so the thread never needs to touch the table mutex ever again. The reader stores its transaction ID in the slot at the start of the read transaction and zeroes the ID in the slot at the end of the transaction. In normal operation, there is nothing that can block the operation of readers.

The reader table is used when a writer wants to allocate a page, and knows that the free list is not empty. Writes are performed using copy-on-write semantics; whenever a page is to be written, a copy is made and the copy is modified instead of the original. Once copied, the original page's ID is added to an in-memory free list. When a transaction is committed, the in-memory free list is saved as a single record in the free list DB along with the ID of the transaction for this commit. When a writer wants to pull a page from the free list DB, it compares the transaction ID of the oldest record in the free list DB with the transaction IDs of all of the active readers. If the record in the free list DB is older than all of the readers, then all of the pages in that record may be safely re-used because nothing else in the DB points to them any more.

The writer's scan of the reader table also requires no locks, so readers cannot block writers. The only consequence of a reader holding onto an old snapshot for a long time is that page reclaiming cannot be done; the writer will simply use newly allocated pages in the meantime.

Configuration for an application using MDB is extremely simple - there are no cache configuration settings. The library requires only a pathname for storing the database files, and a maximum allowed size for the database. The configuration settings only affect the capacity of the database, not its performance; there is nothing to tune.

4.3 Notable Special Features

A number of special options are provided in the MDB API that are worth mentioning:

- Explicit Key Types: while BDB supported custom sort functions for database keys, it was inconvenient to use them because their code was private to a given application and not available to the generic BDB management tools. Manipulating a database without using the correct collation would result in a corrupted database. With MDB, flags can be set to explicitly denote keys that are sorted in reverse byte order, as well as keys that are in native binary integer format. These cover the most common cases that required custom sorting in our use of BDB, and the flags are stored in the database header so that any MDB application will sort correctly without any special effort.
- Append Mode: BDB provided a special bulk-load API aimed at improving the speed of database bulk loads, but it was extremely awkward to format the input data for this API. In MDB simply setting the MDB_APPEND flag on a put operation triggers MDB's bulk-load support. When this flag is set, data items are added sequentially to the database, allowing writes to proceed at the full sequential write speed of the underlying storage system.
- Reserve Mode: Typically when storing records in the database, data is copied from a user-supplied buffer into an internal write buffer, before being written to the underlying storage. With the MDB_RESERVE flag, the database reserves a space of the desired size in its write buffer and returns the address to the caller, instead of copying the user's data. This is useful when the output data is being generated on the fly, as opposed to being a simple copy of a static data item. In back-mdb this is used when serializing a slapd Entry structure for storage, and allows these large objects to be stored without an extra memcpy step.
- Fixed Mapping: An option is available to always map the database at a fixed address. This feature allows complex pointer-based data structures to be stored directly in the database with minimal serialization, and to be read from the database with no deserialization. This feature could be used for an object-oriented database, among other purposes.

5. Results

Profiling was done using multiple tools, including FunctionCheck[24], valgrind callgrind[25], and oprofile[26], to aid in optimization of MDB. Oprofile has the least runtime overhead and provides the best view of multi-threaded behavior, but since it is based on random samples it tends to miss some data of interest. FunctionCheck is slower, at four times slower than normal, but since it uses instrumented code it always provides a complete profile of overall function run times. callgrind is slowest, at thirty times slower than normal, and only provides

relevant data for single-threaded operation, but since it does instruction-level profiling it gives the most detailed view of program behavior. Since program behavior can vary wildly between single-threaded and multi-processor operation, it was important to gather performance data from a number of different perspectives.

5.1 Microbenchmark Results

The benchmarking code developed for Google's LevelDB[27] was adapted for use with MDB and BDB. LevelDB caught our attention because it specifically claims to be fast and lightweight. We repeated their tests against LevelDB as well as SQLite 3 and Kyoto Cabinet's TreeDB, in addition to testing MDB and BDB.

One needs to be particularly careful when microbenchmarking, since the results in isolation tend to change drastically when the code moves to real world deployments. Testing under just a single enviroment also may give a skewed image of the results one may expect in more general use. As such, all of the test scenarios that Google outlined were repeated under multiple conditions:

1. Using a tmpfs (memory-based) filesystem, to show the raw efficiency of each database implementation's algorithms, independent of any I/O overhead.
2. Using an SSD, to show performance when some I/O overhead is present, but without seek latency.
3. Using a standard HDD, to show worst-case performance.

The reiserfs filesystem was used for both (2) and (3), but it became apparent that this was not fully representative of the range of deployments. As such, all of the tests were repeated again using the HDD across a wide range of filesystems, including btrfs, ext2, ext3, ext4, jfs, ntfs, reiserfs, xfs, and zfs. In addition, the journaling filesystems that support using an external journal were retested with their journal stored on a tmpfs file. Testing in this configuration shows how much overhead the filesystem's journaling mechanism imposes, and how much performance is lost by using the default internal journal configuration.

These tests were performed using Linux kernel version 3.2.0-26 as shipped in Ubuntu 12.04, which was the latest released for Ubuntu at the time of writing. Newer kernel versions are rumored to provide improved performance in ext4, but it was not available for testing. The test machine is a Dell Precision M4400 laptop with a quad-core Intel(R) Core(TM)2 Extreme CPU Q9300 running at 2.53GHz, with 6144KB of total L3 cache and 8GB of DDR2 RAM at 800MHz. Tests were all run in single-user mode to prevent variations due to other system activity. CPU performance scaling was disabled (scaling_governor = performance) to ensure a consistent CPU clock speed for all tests. The numbers reported below are the median of three measurements. The resulting databases are completely deleted between each of the three measurements.

The SSD used for testing is a relatively old model, Samsung PM800 Series 256GB with original firmware, version VBM15D1Q. The drive has been in use for several years and was

not reformatted before running these tests.

The HDD used for testing is a Western Digital WD20EARX 2TB SATA drive. This is a nominally 5400rpm drive, optimized for low power consumption as opposed to high performance. It was newly purchased for the tests, and each filesystem was created fresh for each test run.

How Lightweight is "Lightweight"?

Table 5.1 shows the output of the "size" command run across the test programs, each linked with the static archive of their respective database libraries. This shows the minimum size of an application using each library, minus any other runtime libraries they depend on. The *db_bench* app uses LevelDB.

text	data	bss	dec	hex	filename
271991	1456	320	273767	42d67	db_bench
1682579	2288	296	1685163	19b6ab	db_bench_bdb
96879	1500	296	98675	18173	db_bench_mdb
655988	7768	1688	665444	a2764	db_bench_sqlite3
296244	4808	1080	302132	49c34	db_bench_tree_db

Table 5.1: Relative Footprint

MDB is written in plain C; aside from the C library its only other dependency is the system's threading library. BDB and SQLite3 are also plain C, but they include a lot of features that are extraneous to plain key/value access. LevelDB and Kyoto Cabinet are written in C++. Clearly MDB has the smallest footprint. The core of the library fits entirely within a CPU's Level 1 cache. Read requests can be serviced with essentially zero instruction fetch overhead.

Read Performance

MDB has no cache of its own, but the other databases all use various forms of caching. Table 5.2 shows the results for read operations where all data resides in the respective database caches. In this test the keys are fixed at 16 bytes in length, the data items are 100 bytes, and there are 1 million records in the database.

	Sequential ops/sec	Random ops/sec
MDB	14,492,754	768,640
BerkeleyDB	879,507	173,641
LevelDB	4,504,505	187,196
Kyoto TreeDB	1,282,051	218,675
SQLite3	339,328	101,276

Table 5.2: Fully Cached Read Performance, Small Records

Read performance is even more dramatic for larger records, demonstrated here using a database with 100,000 byte keys, and 1,000 records. As Table 5.3 shows, MDB's read

performance is orders of magnitude faster than everything else.

	Sequential ops/sec	Random ops/sec
MDB	33,333,333	2,012,072
BerkeleyDB	9,174	9,347
LevelDB	194,628	17,115
Kyoto TreeDB	18,536	17,207
SQLite3	7,476	7,690

Table 5.3: Read Performance, Large Records

With MDB's zero-memcpy reads, read performance is determined solely by the number of keys in the database, not the size of the records being returned.

Write Performance

The write tests across multiple filesystems generated too much data to simply summarize in a few tables, so several charts are provided instead. With a few exceptions, Google's LevelDB handily outperforms all of the other databases. Illustration 5.1 shows the performance for asynchronous sequential writes across all of the tested filesystems. Again, this is using 16 byte keys, 100 byte data values, and 1 million records.

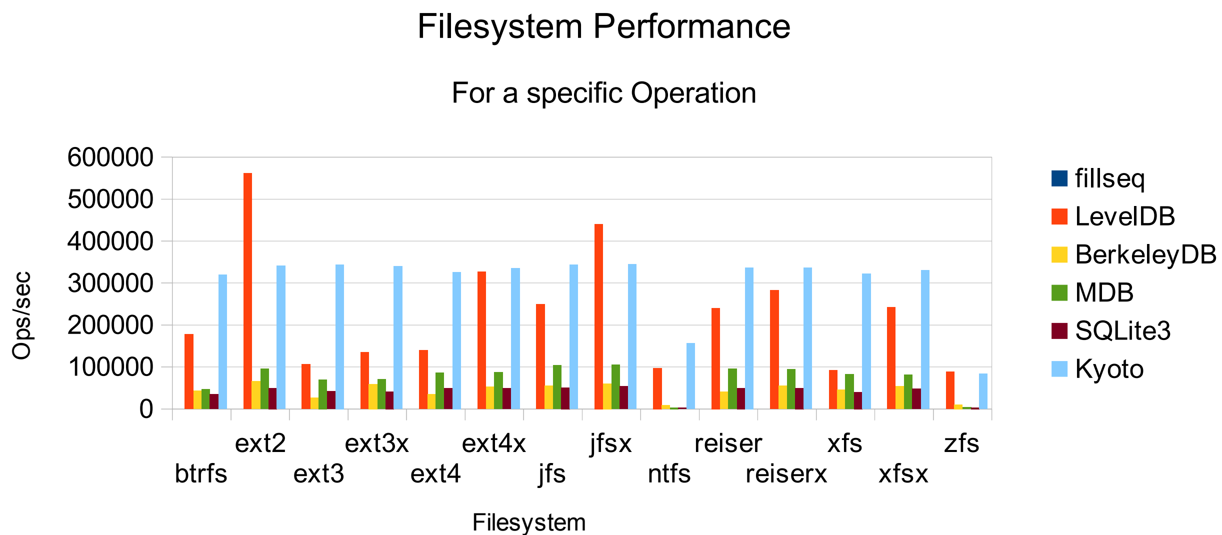

Illustration 5.1: Asynchronous Sequential Writes

(Note that both ntfs and zfs are FUSE-based, so their performance is inherently crippled compared to the other filesystems.)

Illustration 5.2 shows the performance for asynchronous random writes across all of the tested filesystems.

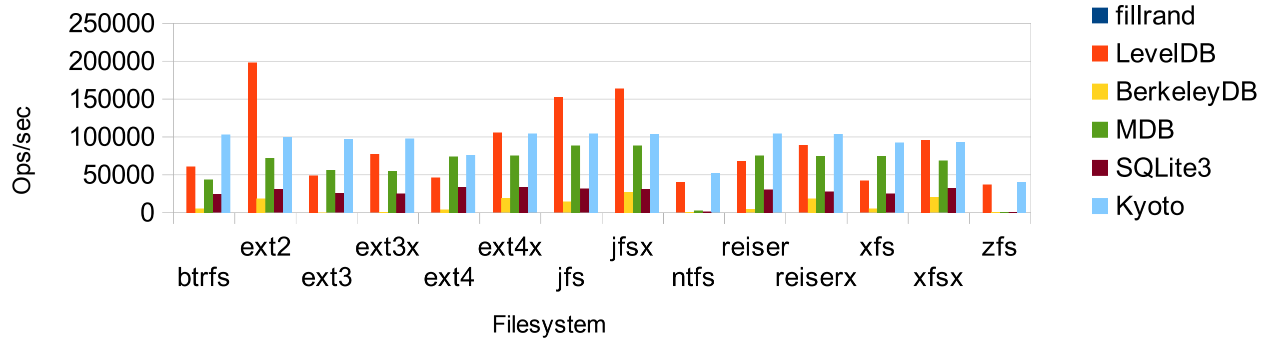

Illustration 5.2: Asynchronous Random Writes

From these results we can see that even though the asynchronous writes should be fully cached in memory, their performance is still heavily dependent on the underlying filesystem type. Also, despite its age, ext2 is still the fastest for many of these workloads.

All of these databases except Kyoto TreeDB also support the batching of multiple operations into a single transaction. Generally this allows greater throughput than just performing one operation per transaction. The following two charts show the performance for asynchronous writes with batches of 1000 operations per transaction. (For Kyoto TreeDB its single-operation speed is shown again, just for reference.)

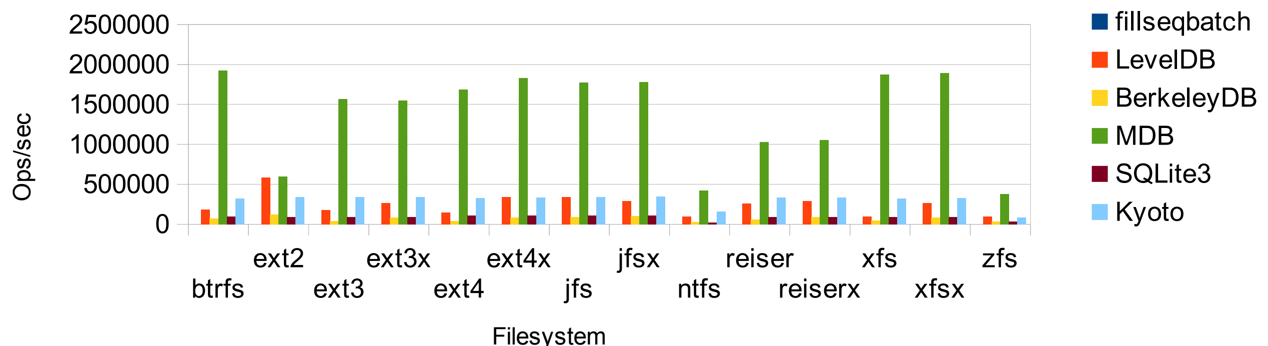

Illustration 5.3: Batched Asynchronous Sequential Writes

In this mode MDB's bulk loading optimizations take effect, allowing it to store millions of records per second. Illustration 5.3 clearly demonstrates how effective the MDB_APPEND option is. It also shows that ext2 is ill-suited to large write operations.

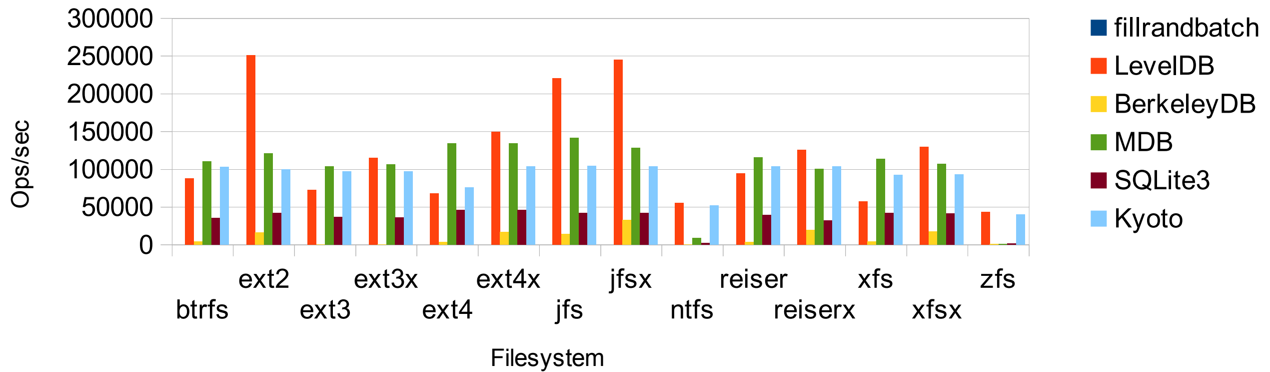

Illustration 5.4: Batched Asynchronous Random Writes

In Illustration 5.4 we see a fairly surprising result - MDB writes are fastest of all the DBs, on btrfs, ext3, ext4, reiserfs, and xfs. Given that MDB's focus is read performance and not write performance, this result is quite unexpected.

But the results for synchronous write operations are even more surprising, as shown in the next two charts.

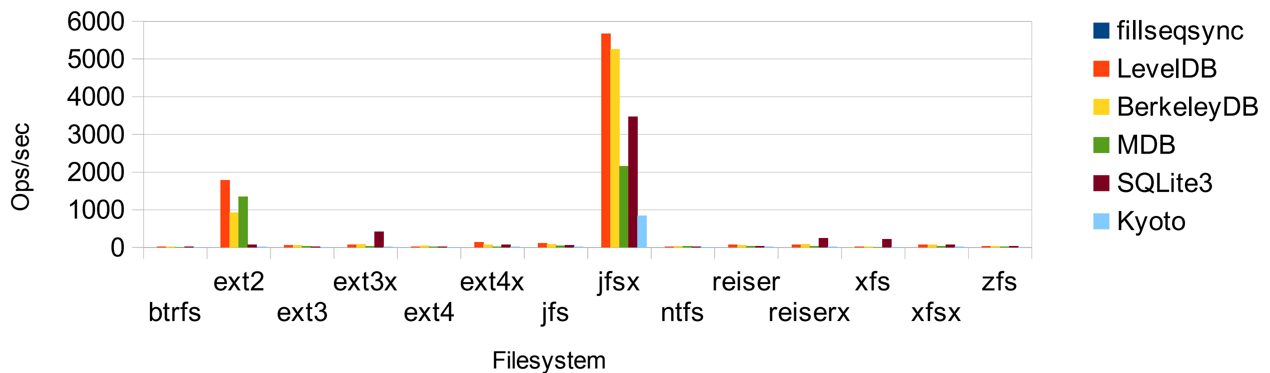

Illustration 5.5: Synchronous Sequential Writes

For most of the filesystems, the synchronous write rates are on the order of tens of operations per second. But for jfs using an external journal, the write rate jumps to several thousand per second. Clearly, if your application requires fully synchronous writes and as high performance as possible, you need to use jfs with a journal stored on a separate device from the main filesystem. This is true regardless of whether you're performing sequential writes, as in

Illustration 5.5 or random writes as in Illustration 5.6.

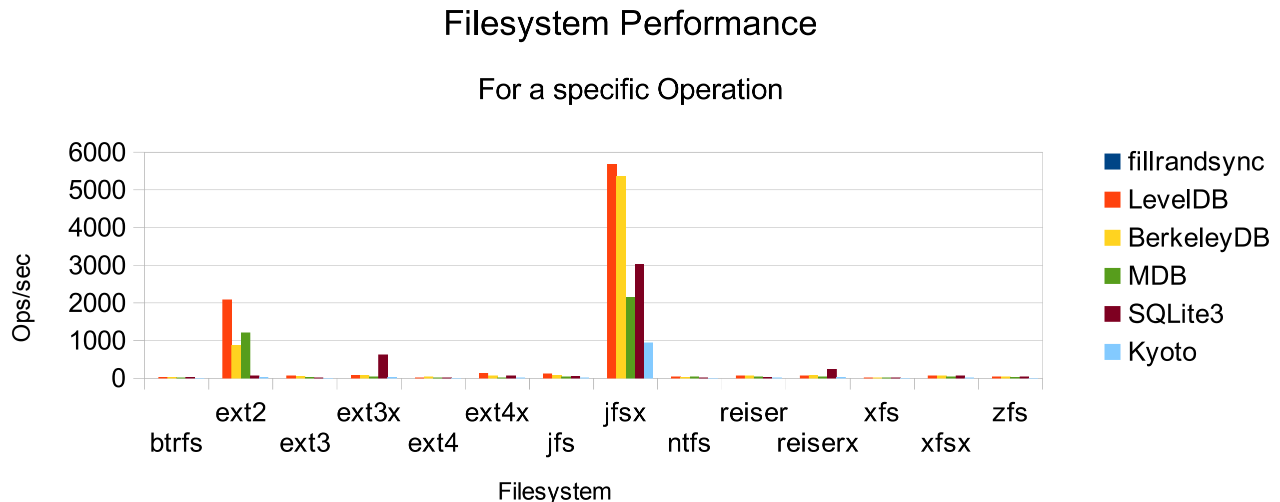

Illustration 5.6: Synchronous Random Writes

5.2 OpenLDAP slapd Results

The microbenchmark results provided a lot of useful data, but we also need to see how MDB performs in a real application. The following results summarize performance with MDB in slapd's back-mdb backend, compared to BDB in the back-hdb backend. These tests were run on an HP DL585G with four AMD quad-core Opteron 8354 processors and 128GB of RAM.

Table 5.4 compares basic performance of back-mdb vs back-hdb for initially loading a test database using slapadd in "quick" mode.

	real	user	sys
back-hdb	66m09.831s	115m52.374s	5m15.860s
back-mdb	29m33.212s	22m21.264s	7m11.851s

Table 5.4: Time to slapadd -q 5 million entries

back-hdb has a much higher user time than real time because it was using multi-threaded indexing. At present back-mdb doesn't support multi-threaded indexing. back-hdb was using BDB 4.7.25 in these tests, but results with BDB 5.2.28 were essentially the same.

With the databases loaded, the next test was to start up slapd and time how long it took to scan the entire database with a single ldapsearch. Also the slapd process sizes were compared, relative to their DB sizes on disk. These results are summarized in Table 5.5.

	first	second	slapd size	DB size
back-hdb	4m15.395s	0m16.204s	26GB	15.6GB
back-mdb	0m12.47s	0m9.94s	6.7GB	9GB

Table 5.5: ldapsearch comparison

back-hdb is configured with an entry cache size of 5 million entries, so all of the database is fully cached after the first ldapsearch is run. Also note that the DB files are entirely resident in the filesystem cache since slapadd had just completed before. Also the BDB cache was configured at 32GB so the entire database is resident there too; no disk I/O occurs during these tests. This table shows the overhead of retrieving data from the BDB cache and decoding it into the back-hdb entry cache. But even with that overhead eliminated in the second run, back-mdb is still faster. For back-mdb the extra time required in the first run reflects the time needed for the OS to map the database pages into the slapd process' address space. The slapd process size for mdb is smaller than the DB size for a couple of reasons: first, the DB contains attribute indices, and this search doesn't reference any indices, so those pages are not mapped into the process. second, the DB contains a number of free pages that were left over from the last slapadd transaction.

Before development began it was estimated that the MDB approach would use 1/3 to 1/2 as much RAM as the equivalent back-hdb database; in fact back-mdb was using only 25% as much RAM as back-hdb on our fully cached test database.

Next, a basic concurrency test was performed by running the same ldapsearch operation 2, 4, 8, and 16 times concurrently and measuring the time to obtain the results. The averages of the result times are shown in Table 5.6.

	2	4	8	16
back-hdb, debian	0m23.147s	0m30.384s	1m25.665s	17m15.114s
back-hdb	0m24.617s	0m32.171s	1m04.817s	3m04.464s
back-mdb	0m10.39s	0m10.87s	0m10.81s	0m11.82s

Table 5.6: Concurrent Search Times

The first time this test was run with back-hdb yielded some extraordinarily poor results. Later testing revealed that this test was accidentally run using the stock build of BDB 4.7 provided by Debian, instead of the self-compiled build we usually use in our testing. The principle difference is that we always build BDB with the configure option --with-mutex=POSIX/pthread, whereas by default BDB uses a hybrid of spinlocks and pthread mutexes. The spinlocks are fairly efficient within a single CPU socket, but they scale extremely poorly as the number of processors increases. back-mdb's scaling is essentially flat across arbitrary numbers of processors since it has no locking to slow it down. The performance degrades slightly at the 16 search case because at that point all of the processors on our test machine are busy so the clients and slapd are competing with other system processes for CPU time. As another point of reference, the time required to copy the MDB database to /dev/null using 'dd' was

10.20 seconds. Even with all of the decoding and filtering that slapd needed to do, scanning the entire DB was only 6% slower than a raw copy operation.

The previous tests show worst-case performance for search operations. For more real-world results, we move on to using SLAMD[28]. (SLAMD has known performance issues, but we've gotten used to them, and staying with the same tool lets us compare with historical results from our previous work as well.) Table 5.7 summarizes the results for back-hdb vs back-mdb with randomly generated queries across the 5 million entry database.

	Searches/sec	Duration, msec
back-hdb	67456.11	1.89
back-mdb	119255.42	0.63

Table 5.7: SLAMD Search Rate Results

The back-hdb result is actually extremely good - it's about 15% faster than the second fastest directory software we've tested previously on this machine (OpenDS 2.3). But they're all utterly outclassed by back-mdb. If you look at the actual stats in Illustration 5.7 you'll see that the performance was still increasing as the process' page map was filling in.

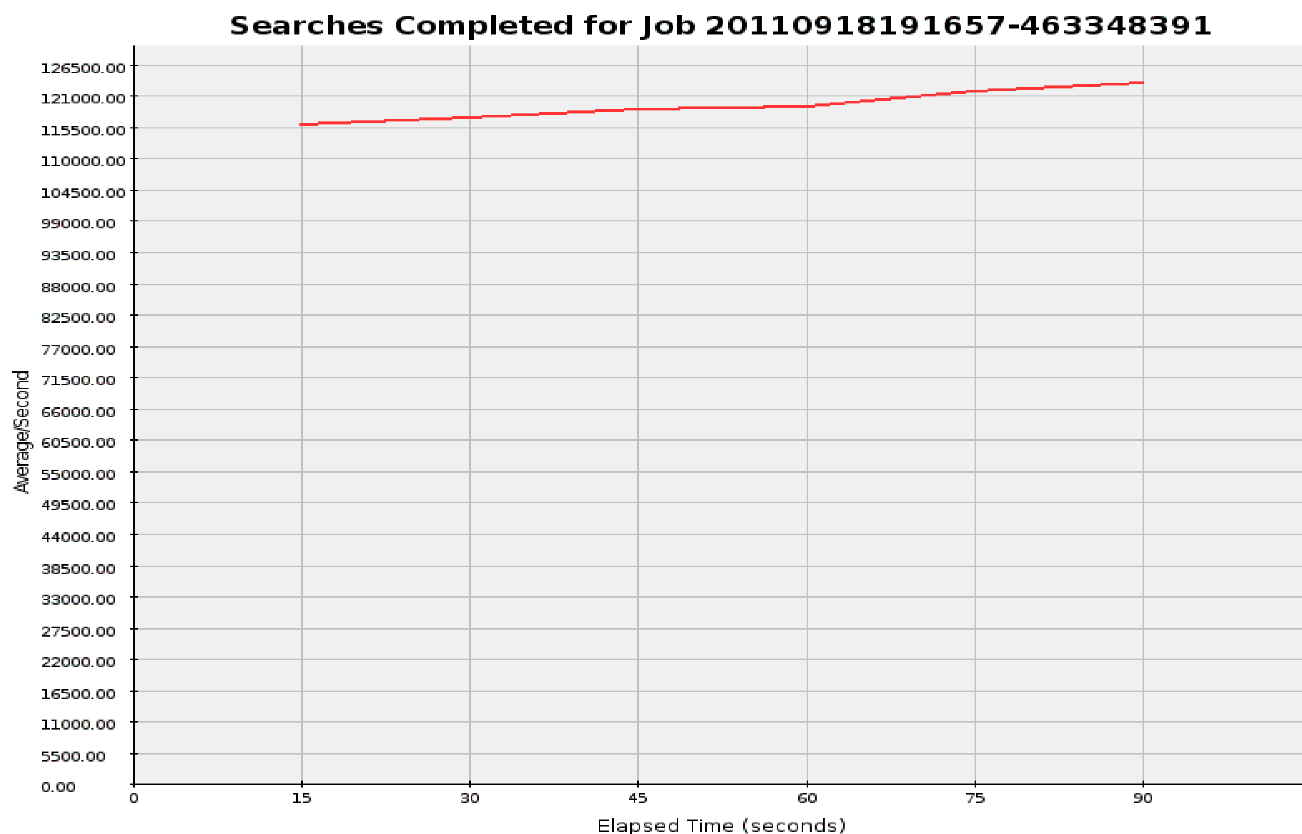

Illustration 5.7: back-mdb Search Rate results

After seeing these results we considered renaming MDB as "LightningDB" - its read performance is blindingly fast and totally unparalleled.

Another interesting observation can be made from Table 5.7 - back-mdb's search duration is

one third of back-hdb's, but the overall search rate is only twice as fast. This indicates that some other bottleneck is present in the system, possibly in the SLAMD load generator, or possibly in the slapd frontend. This will be the subject of a future investigation, as it shows that there is the potential for a further 50% increase in slapd's search rate using back-mdb.

For write speeds, back-mdb is significantly slower than back-hdb. Table 5.8 shows the throughput in a pure Modify test, modifying a single attribute in random entries across the 5 million entry database.

	Modifies/sec	**Duration, msec**
back-hdb	20440.83	1.56
back-mdb	6131.77	1.29

Table 5.8: SLAMD Modify Rate Results

Note that back-mdb actually completes modifies quickly, but because MDB enforces single-writer behavior, it does not accept as many writes per second. Our final comparison in Table 5.9 shows a Modify Rate job running concurrently with a Search Rate job.

	Searches/sec	**Search msec**	**Modifies/sec**	**Modify msec**
back-hdb	40629.49	1.47	12321.36	1.62
back-mdb	85918.92	1.77	2844.95	2.80

Table 5.9: SLAMD Combined Search and Modify Rate

Most of the effort has been focused on read performance so far; future work may be able to boost MDB's write performance but it is not perceived as a critical problem for now.

6. Conclusions

The combination of memory-mapped operation with Multi-Version Concurrency Control proves to be extremely potent for LDAP directories. The administrative overhead is minimal since MDB databases require no periodic cleanup or garbage collection, and no particular tuning is needed. The copy-on-write architecture means the database can never be corrupted by system crashes, so the database is always immediately usable on system boot - there are no lengthy recovery or rebuild procedures. Code size and complexity have been drastically reduced, while read performance has been significantly raised. Write performance has been traded for read performance, but this is acceptable and can be addressed in more depth in the future.

6.1 Portability

While initial development was done on Linux, MDB and back-mdb have been ported to MacOSX and Windows. No special problems are anticipated in porting to other platforms.

6.2 Other Directions

A port of SQLite to use MDB has also been done. The MDB library needed to be extended to support nested transactions, but otherwise needed very little changes. Basic functionality is working already, and the code can be obtained at http://gitorious.org/mdb/sqlightning. There are probably many other applications for a small-footprint database library with relatively low write rates and near-zero read overhead. Some candidates are perl and python DB wrappers, as well as a version of MemcacheDB[29] using MDB instead of BDB. A port of XDAndroid[30] using the MDB port of SQLite is also in progress.

6.3 Future Work

A number of items remain on our ToDo list.

- Write optimization has not yet been investigated.
- It would be nice to allow the database map size and other configuration settings to be grown dynamically instead of statically configured.
- Functions to facilitate incremental and/or full backups would be nice to have.
- A back-mdb that stores entries in the DB in their in-memory format, thus requiring no decoding at all, is still being considered.

None of these items are seen as critical show-stoppers. MDB and back-mdb already meet all the goals set for them and fulfill all of the functions required of an OpenLDAP backend, while setting a new standard for database efficiency, scalability, and performance.

References

1: Oracle, BerkeleyDB, 2011, http://www.oracle.com/technetwork/database/berkeleydb/overview/index.html

2: Wikipedia, B+trees, , http://en.wikipedia.org/wiki/B+_tree

3: Wikipedia, MVCC, , http://en.wikipedia.org/wiki/Multiversion_concurrency_control

4: Margo Seltzer, Keith Bostic, Berkeley DB, The Architecture of Open Source Applications, 2011, http://www.aosabook.org/en/bdb.html

5: Jong-Hyuk Choi and Howard Chu, Dissection of Search Latency, 2001, http://www.openldap.org/lists/openldap-devel/200111/msg00042.html

6: Howard Chu, Better malloc strategies?, 2006, http://www.openldap.org/lists/openldap-devel/200607/msg00005.html

7: Howard Chu, Malloc Benchmarking, 2006, http://highlandsun.com/hyc/malloc/

8: Emery Berger, The Hoard Memory Allocator, 2006-2010, http://www.hoard.org

9: Sanjay Ghemawat, Paul Menage, TCMalloc: Thread-Caching Malloc, 2005, http://goog-perftools.sourceforge.net/doc/tcmalloc.html

10: Howard Chu, Minimize malloc fragmentation, 2006, http://www.openldap.org/lists/openldap-devel/200608/msg00033.html

11: Wikipedia, Page replacement algorithms, , http://en.wikipedia.org/wiki/Page_replacement_algorithm#Least_recently_used

12: Howard Chu, CLOCK-Pro cache replacement code, 2007, http://www.openldap.org/lists/openldap-bugs/200701/msg00039.html

13: Howard Chu, Cache-thrashing protection, 2007, http://www.openldap.org/lists/openldap-commit/200711/msg00068.html

14: Wikipedia, Single-Level Store, , http://en.wikipedia.org/wiki/Single-level_store

15: Multicians, Multics General Information and FAQ, , http://www.multicians.org/general.html

16: Apollo Computer Inc., Domain/OS Design Principles, 1989, http://bitsavers.org/pdf/apollo/014962-A00_Domain_OS_Design_Principles_Jan89.pdf

17: R.A. Gingell, J.P. Moran, and W.A. Shannon, Virtual Memory Architecture in SunOS, USENIX Summer Conference, 1987, http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.132.8931

18: Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, John S. Quarterman, Memory Management, The Design and Implementation of the 4.4BSD Operating System, 1996, http://www.freebsd.org/doc/en/books/design-44bsd/overview-memory-management.html

19: Linus Torvalds, Status of the buffer cache in 2.3.7+, 1999, http://lkml.indiana.edu/hypermail/linux/kernel/9906.3/0670.html

20: Apache Software Foundation, Apache CouchDB: Technical Overview, 2008-2011, http://couchdb.apache.org/docs/overview.html

21: Martin Hedenfalk, OpenBSD ldapd source repository, 2010-2011, http://www.openbsd.org/cgi-bin/cvsweb/src/usr.sbin/ldapd/

22: Martin Hedenfalk, How the Append-Only Btree Works, 2011, http://www.bzero.se/ldapd/btree.html

23: Suntorn Sae-eung, Analysis of False Cache Line Sharing Effects on Multicore CPUs, 2010, http://scholarworks.sjsu.edu/etd_projects/2

24: Howard Chu, FunctionCheck, 2005, http://highlandsun.com/hyc/#fncchk

25: Valgrind Developers, Callgrind: a call-graph generating cache and branch prediction profiler, 2011, http://valgrind.org/docs/manual/cl-manual.html

26: , OProfile - A System Profiler for Linux, 2011, http://oprofile.sourceforge.net/news/

27: Google, A fast and lightweight key/value database library by Google, 2011, https://code.google.com/p/leveldb/

28: UnboundID Corp., SLAMD Distributed Load Generation Engine, 2010, http://www.slamd.com

29: , MemcacheDB: A distributed key-value storage system designed for persistent., 2009, http://memcachedb.org/

30: , The XDAndroid Project, 2011, http://xdandroid.com/wiki/Main_Page