The LDAP guys.

The Lightning Memory-Mapped Database (LMDB)

Howard Chu
CTO, Symas Corp. hyc@symas.com
Chief Architect, OpenLDAP hyc@openldap.org

OpenLDAP Project

- Open source code project
- Founded 1998
- Three core team members
- A dozen or so contributors
- Feature releases every 18-24 months
- Maintenance releases as needed

A Word About Symas

- Founded 1999
- Founders from Enterprise Software world
  - PLATINUM technology (Locus Computing)
  - IBM
- Howard joined OpenLDAP in 1999
  - One of the Core Team members
  - Appointed Chief Architect January 2007

The LDAP guys.

Topics

- Overview
- Background
- Features
- Internals
- Special Features
- Results

Overview

- OpenLDAP has been delivering reliable, high performance for many years
- The performance comes at the cost of fairly complex tuning requirements
- The implementation is not as clean as it could be; it is not what was originally intended
- Cleaning it up requires not just a new server backend, but also a new low-level database
- The new approach has a huge payoff

Background

- OpenLDAP provides a number of reliable, high performance transactional backends
  - Based on Oracle BerkeleyDB (BDB)
  - back-bdb released with OpenLDAP 2.1 in 2002
  - back-hdb released with OpenLDAP 2.2 in 2003
  - Intensively analyzed for performance
  - World's fastest since 2005
  - Many heavy users with zero downtime

Background

- These backends have always required careful, complex tuning
  - Data comes through three separate layers of caches
  - Each cache layer has different size and speed characteristics
  - Balancing the three layers against each other can be a difficult juggling act
  - Performance without the backend caches is unacceptably slow - over an order of magnitude...

Background

- The backend caching significantly increased the overall complexity of the backend code
  - Two levels of locking required, since the BDB database locks are too slow
  - Deadlocks occurring routinely in normal operation, requiring additional backoff/retry logic

The LDAP guys.

Background

- The caches were not always beneficial, and were sometimes detrimental
  - data could exist in 3 places at once - filesystem, database, and backend cache - thus wasting memory
  - searches with result sets that exceeded the configured cache size would reduce the cache effectiveness to zero
  - malloc/free churn from adding and removing entries in the cache could trigger pathological heap behavior in libc malloc

SYMAS The LDAP guys.™

Background

- Overall the backends required too much attention
  - Too much developer time spent finding workarounds for inefficiencies
  - Too much administrator time spent tweaking configurations and cleaning up database transaction logs

Obvious Solutions

- Cache management is a hassle, so don't do any caching
  - The filesystem already caches data, there's no reason to duplicate the effort
- Lock management is a hassle, so don't do any locking
  - Use Multi-Version Concurrency Control (MVCC)
  - MVCC makes it possible to perform reads with no locking

Obvious Solutions

- BDB supports MVCC, but it still requires complex caching and locking
- To get the desired results, we need to abandon BDB
- Surveying the landscape revealed no other database libraries with the desired characteristics
- Time to write our own...

OpenLDAP LMDB

- Features At A Glance
  - Key/Value store using B+trees
  - Fully transactional, ACID compliant
  - MVCC, readers never block
  - Uses memory-mapped files, needs no tuning
  - Crash-proof, no recovery needed after restart
  - Highly optimized, extremely compact
    - under 40KB object code, fits in CPU L1 Icache
  - Runs on most modern OSs
    - Linux, Android, *BSD, MacOSX, Solaris, Windows, etc...

Features

- Concurrency Support
  - Both multi-process and multi-thread
  - Single Writer + N Readers
    - Writers don't block readers
    - Readers don't block writers
    - Reads scale perfectly linearly with available CPUs
    - No deadlocks
  - Full isolation with MVCC
  - Nested transactions
  - Batched writes

Features

- Uses Copy-on-Write
  - Live data is never overwritten
  - Database structure cannot be corrupted by incomplete operations (system crashes)
  - No write-ahead logs needed
  - No transaction log cleanup/maintenance
  - No recovery needed after crashes

Features

- Uses Single-Level-Store
  - Reads are satisfied directly from the memory map
    - no malloc or memcpy overhead
  - Writes can be performed directly to the memory map
    - no write buffers, no buffer tuning
  - Relies on the OS/filesystem cache
    - no wasted memory in app-level caching
  - Can store live pointer-based objects directly
    - using a fixed address map
    - minimal marshalling, no unmarshalling required

Single-Level Store

- The approach is only viable if process address spaces are larger than the expected data volumes
  - For 32 bit processors, the practical limit on data size is under 2GB
  - For common 64 bit processors which only implement 48 bit address spaces, the limit is 47 bits or 128 terabytes
  - The upper bound at 63 bits is 8 exabytes

Implementation Highlights

- Resulting library was under 32KB of object code
  - Compared to the original btree.c at 39KB
  - Compared to BDB at 1.5MB
- API is loosely modeled after the BDB API to ease migration of back-bdb code to use LMDB
- Everything is much simpler than BDB

Config Comparison

- LMDB config is simple, e.g. slapd

```
database mdb
directory /var/lib/ldap/data/mdb
maxsize 4294967296
```

- BDB config is complex

```
database hdb
directory /var/lib/ldap/data/hdb
cachesize 50000
idlcachesize 50000
dbconfig set_cachesize 4 0 1
dbconfig set_lg_regionmax 262144
dbconfig set_lg_bsize 2097152
dbconfig set_lg_dir /mnt/logs/hdb
dbconfig set_lk_max_locks 3000
dbconfig set_lk_max_objects 1500
dbconfig set_lk_max_lockers 1500
```

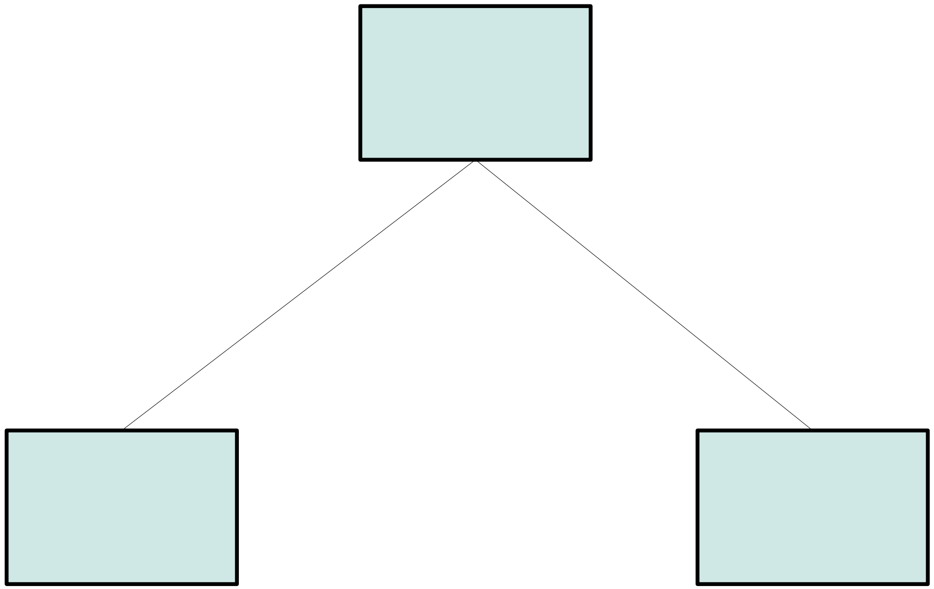
Internals

- B+tree Operation
  - Append-only, Copy-on-Write
  - Corruption-Proof
- Free Space Management
  - Avoiding Compaction/Garbage Collection
- Transaction Handling
  - Avoiding Locking

B+tree Operation

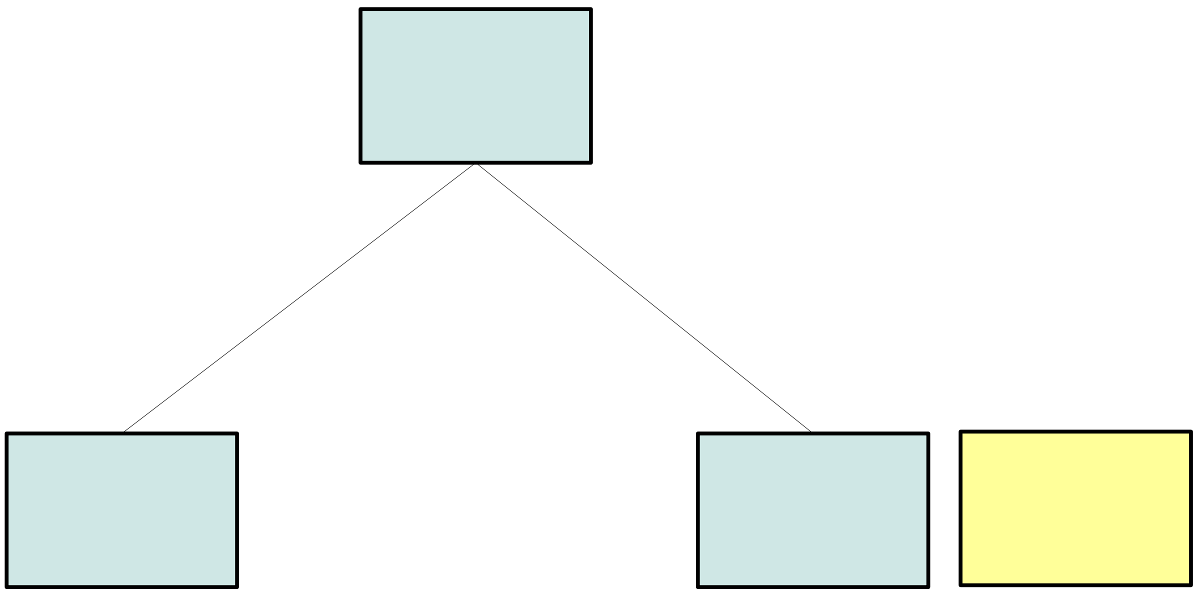
- How Append-Only/Copy-On-Write Works
  - In a pure append-only approach, no data is ever overwritten
  - Pages that are meant to be modified are copied
  - The modification is made on the copy
  - The copy is written to a new disk page
  - The structure is inherently multi-version; you can find any previous version of the database by starting at the root node corresponding to that version

B+tree Operation



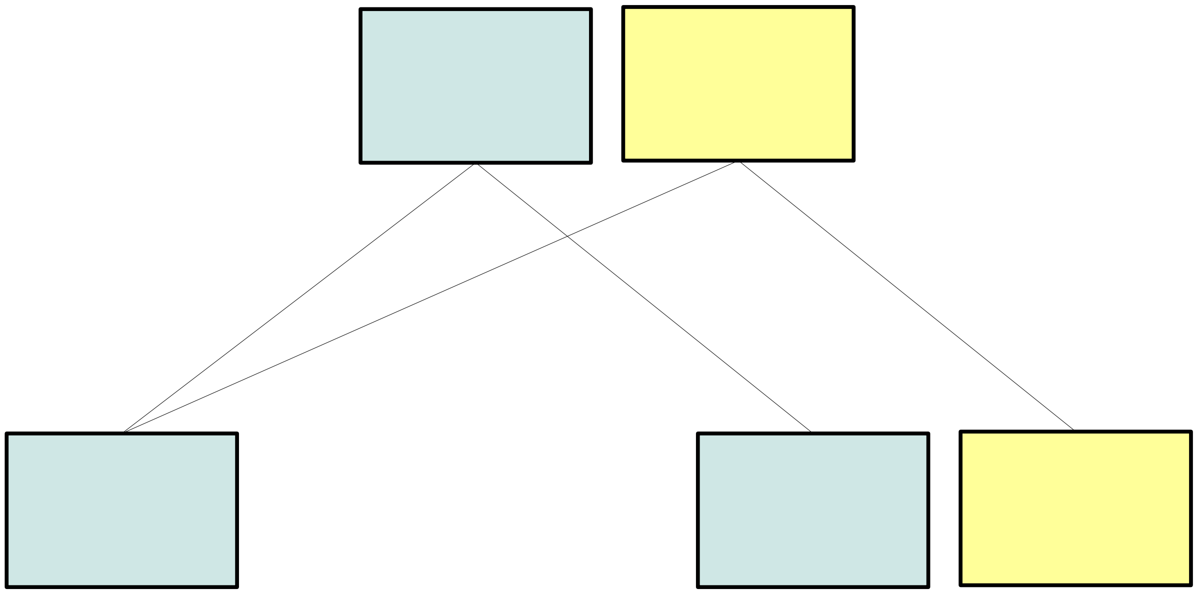
Start with a simple tree

B+tree Operation



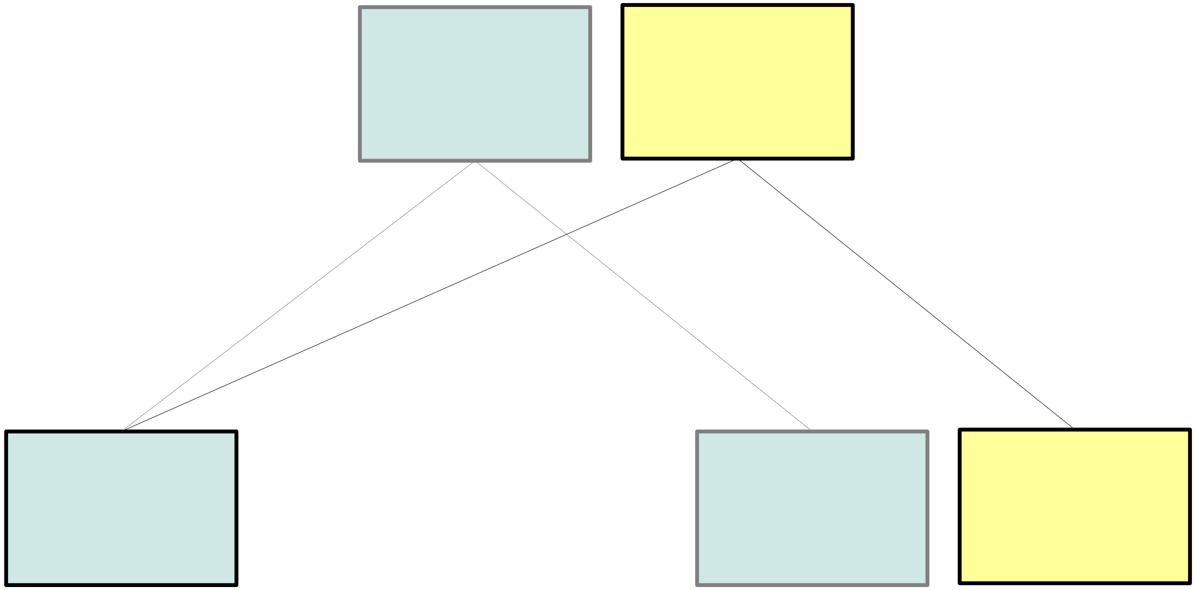
Update a leaf node by copying it and
updating the copy

B+tree Operation



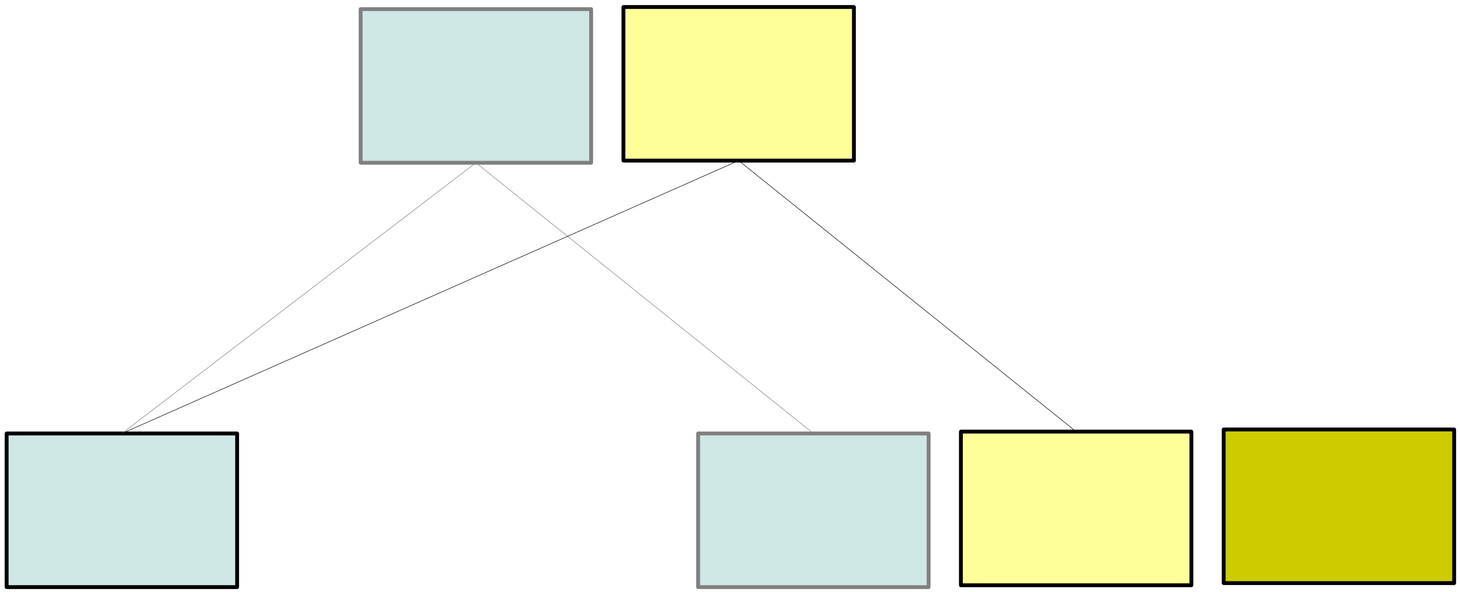
Copy the root node, and point it at the new leaf

B+tree Operation



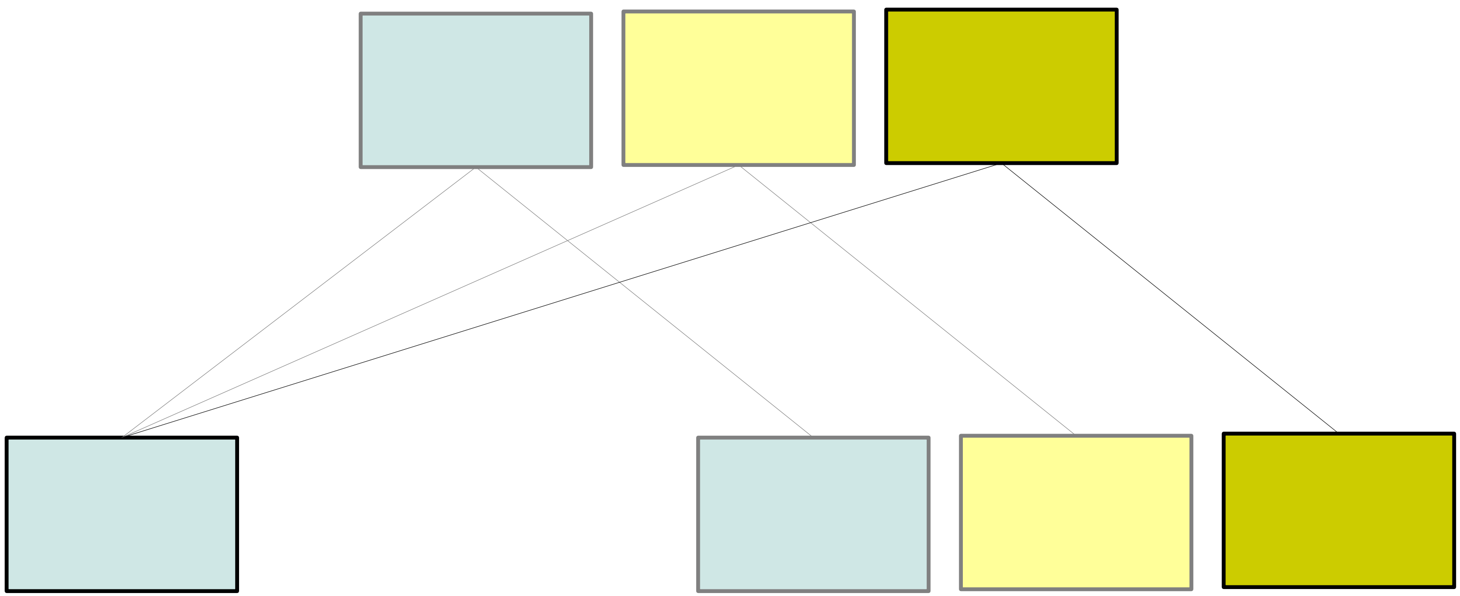
The old root and old leaf remain as a
previous version of the tree

B+tree Operation

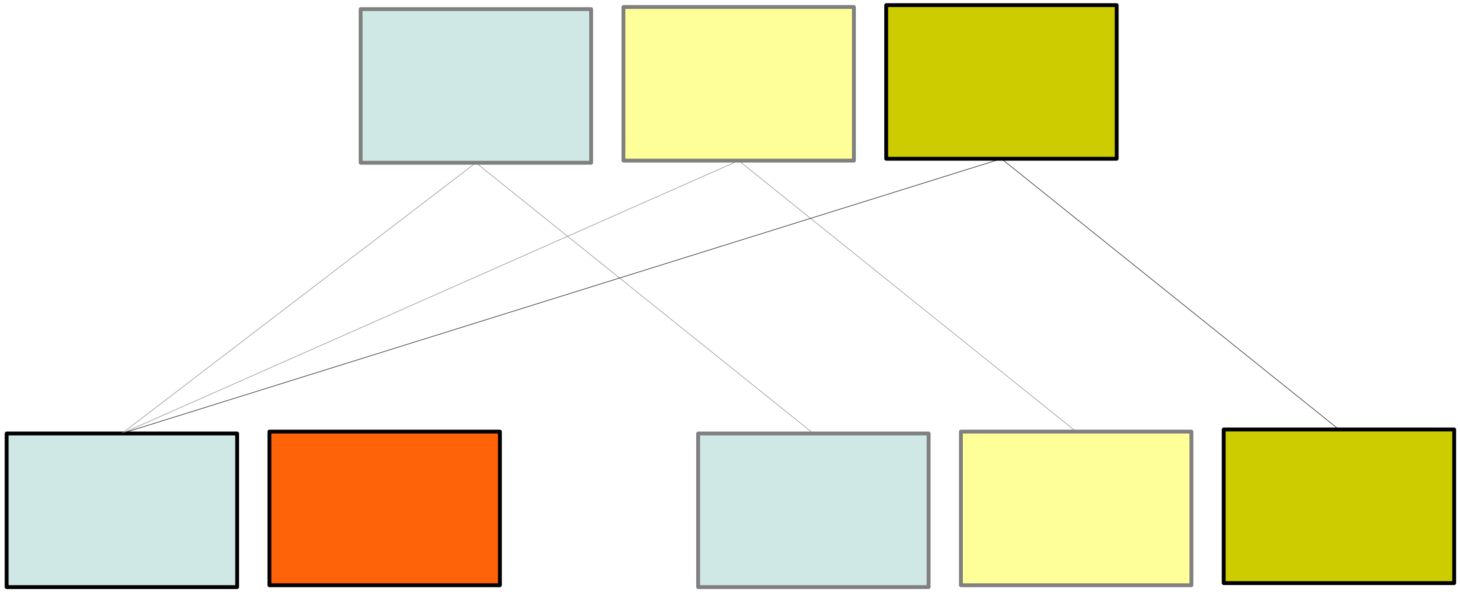


Further updates create additional versions

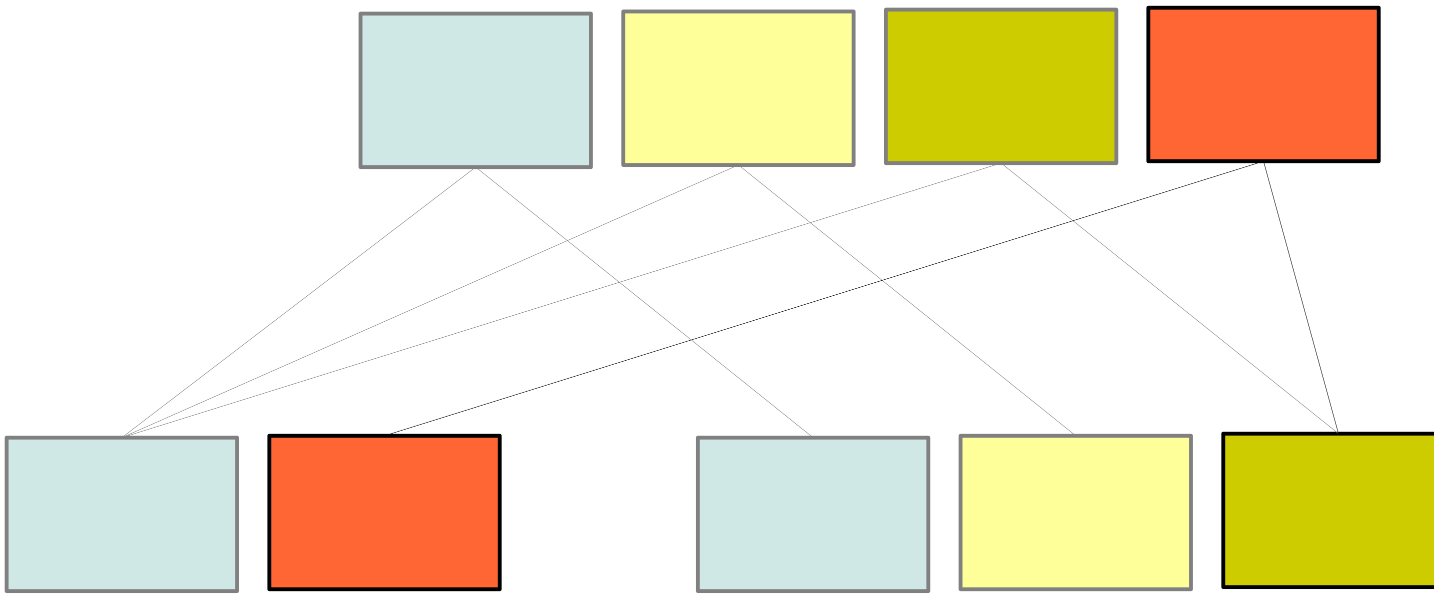
B+tree Operation



B+tree Operation



B+tree Operation



B+tree Operation

- How Append-Only/Copy-On-Write Works
  - Updates are always performed bottom up
  - Every branch node from the leaf to the root must be copied/modified for any leaf update
  - Any node not on the path from the leaf to the root is left unaltered
  - The root node is always written last

B+tree Operation

- In the Append-Only tree, new pages are always appended sequentially to the database file
  - While there's significant overhead for making complete copies of modified pages, the actual I/O is linear and relatively fast
  - The root node is always the last page of the file, unless there was a system crash
  - Any root node can be found by searching backward from the end of the file, and checking the page's header
  - Recovery from a system crash is relatively easy
    - Everything from the last valid root to the beginning of the file is always pristine
    - Anything between the end of the file and the last valid root is discarded

B+tree Operation

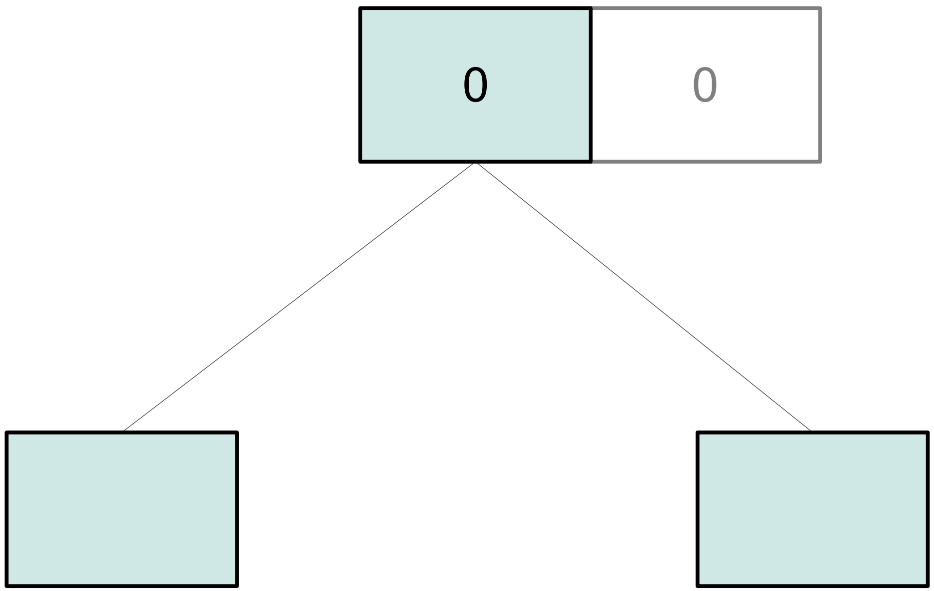
- Append-Only disk usage is very inefficient
  - Disk space usage grows without bound
  - 99+% of the space will be occupied by old versions of the data
  - The old versions are usually not interesting
  - Reclaiming the old space requires a very expensive compaction phase
  - New updates must be throttled until compaction completes

The LDAP guys.

B+tree Operation

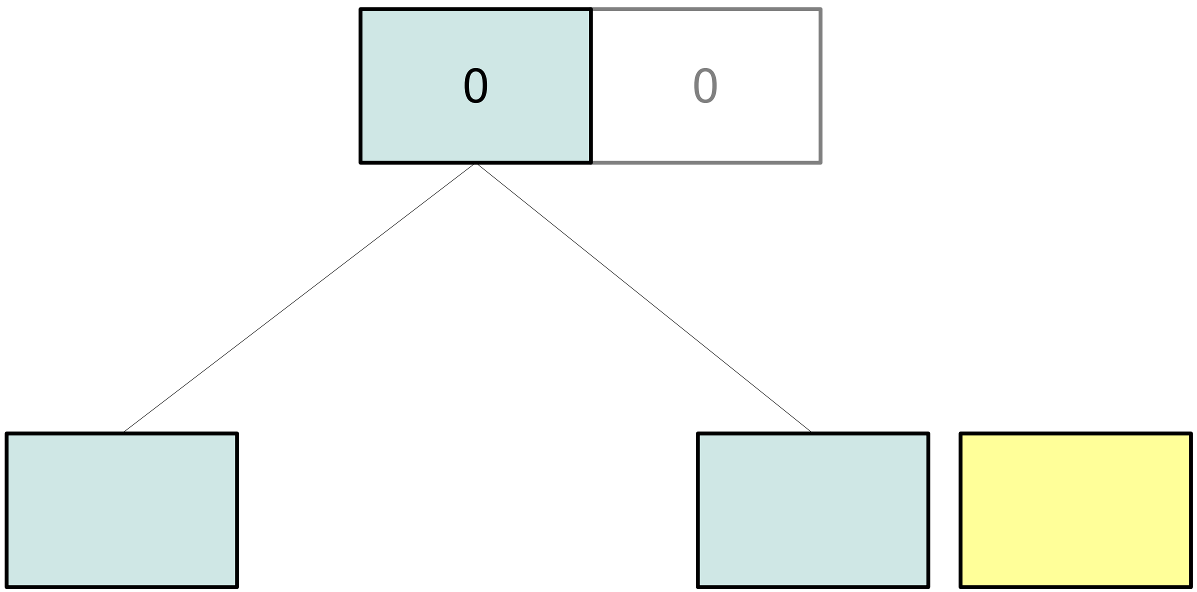
- The LMDB Approach
  - Still Copy-on-Write, but using two fixed root nodes
    - Page 0 and Page 1 of the file, used in double-buffer fashion
    - Even faster cold-start than Append-Only, no searching needed to find the last valid root node
    - Any app always reads both pages and uses the one with the greater Transaction ID stamp in its header
    - Consequently, only 2 outstanding versions of the DB exist, not fully "multi-version"

B+tree Operation

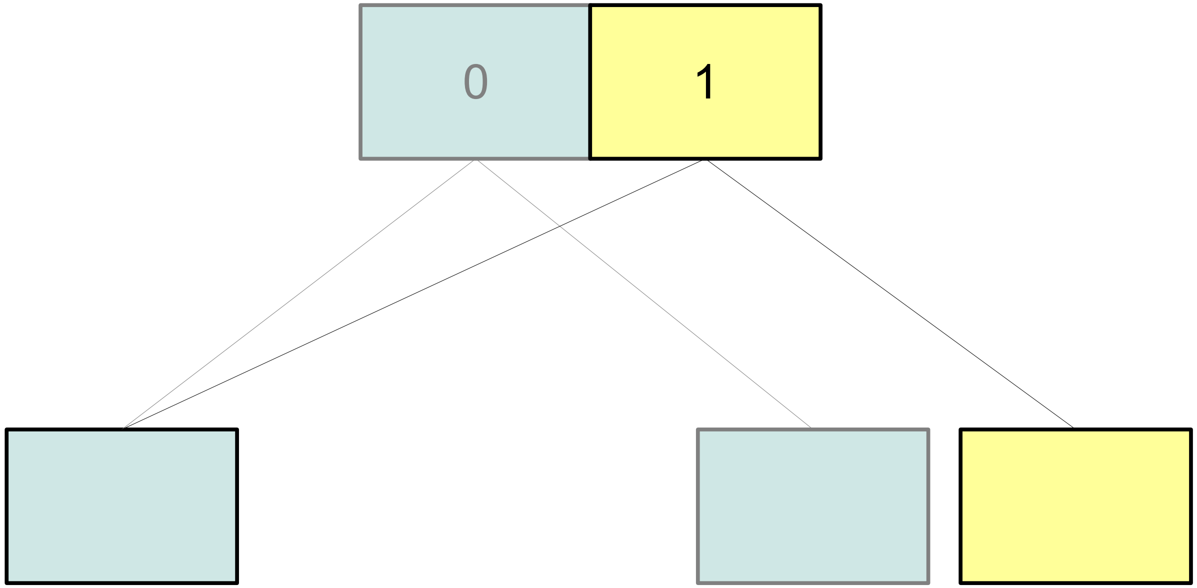


The root nodes have a transaction ID stamp

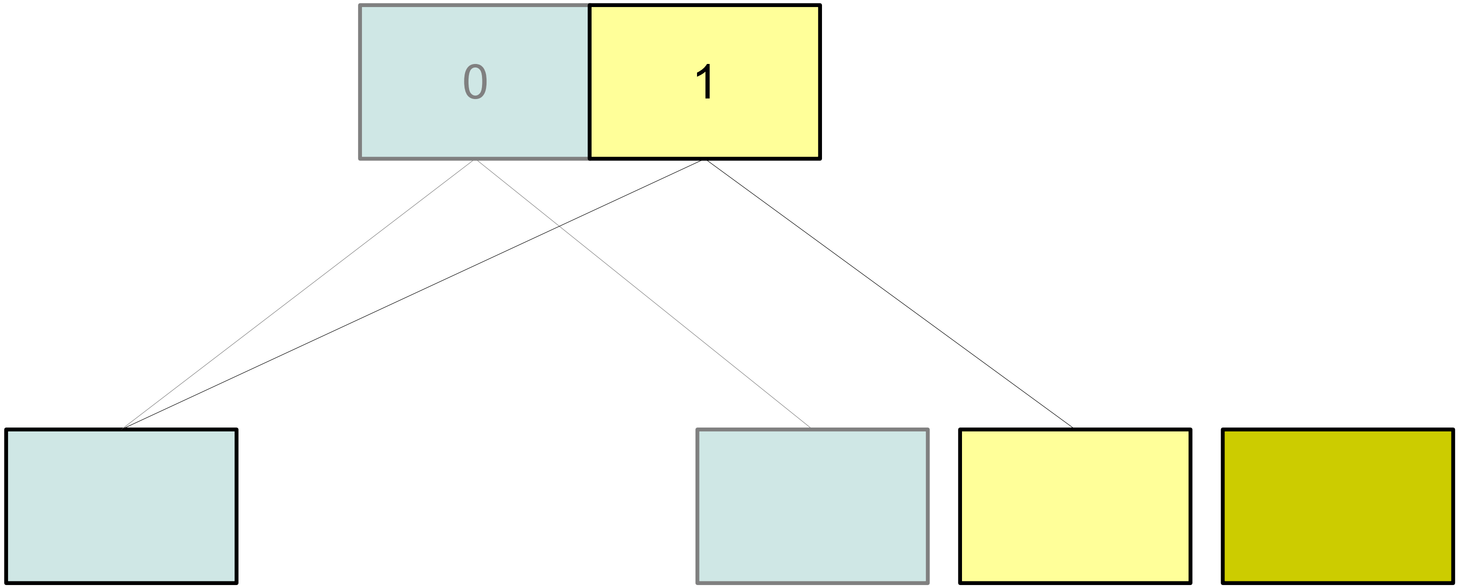
B+tree Operation



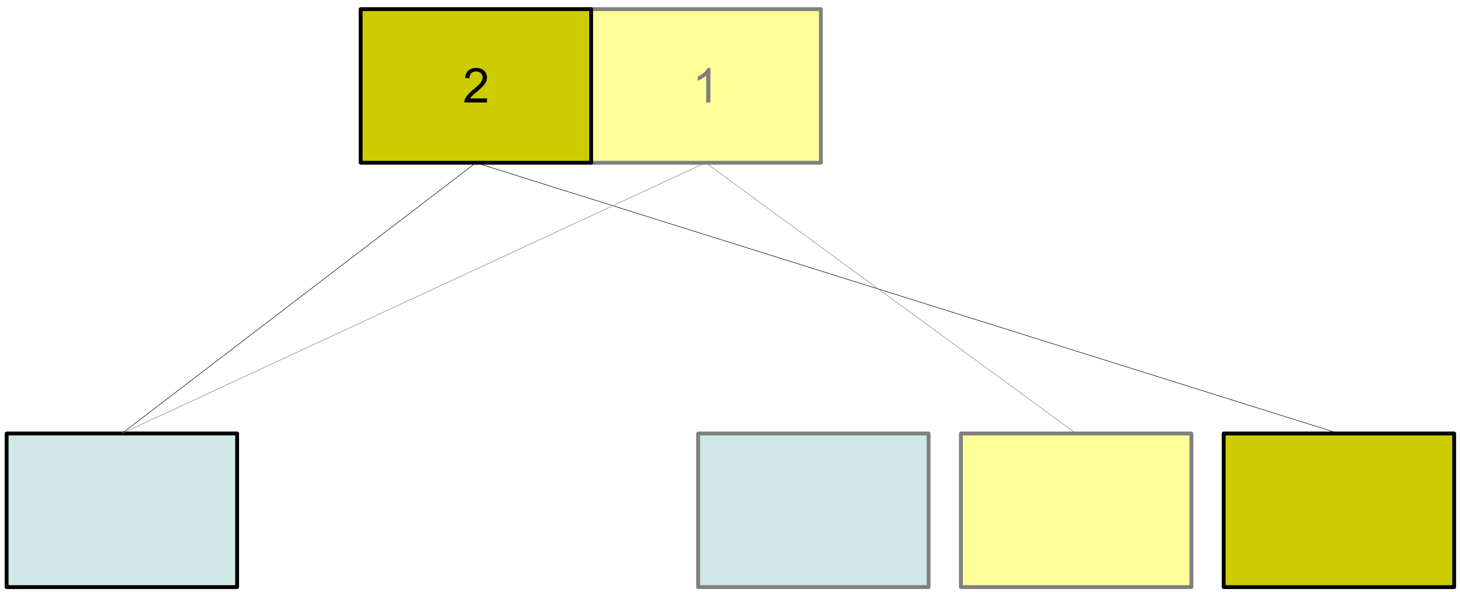
B+tree Operation



B+tree Operation

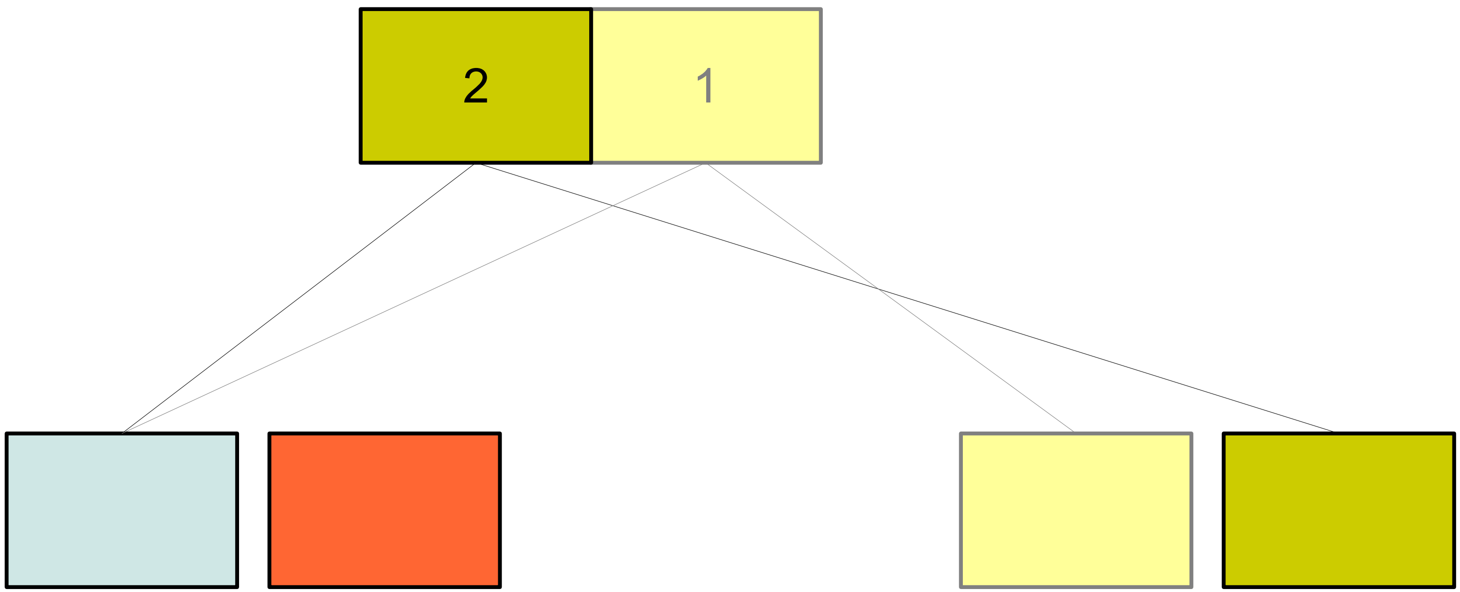


B+tree Operation

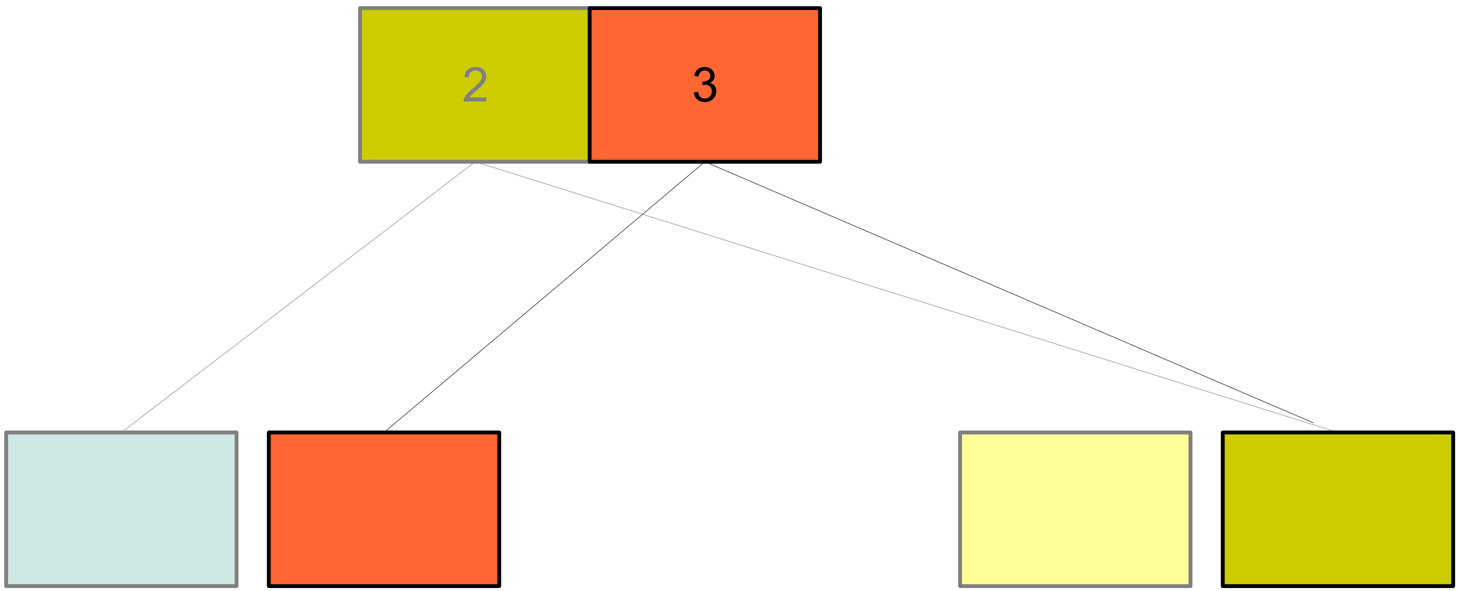


After this step the old blue page is no longer referenced by anything else in the database, so it can be reclaimed

B+tree Operation



B+tree Operation



After this step the old yellow page is no longer referenced by anything else in the database, so it can also be reclaimed

Free Space Management

- LMDB maintains two B+trees per root node
  - One storing the user data, as illustrated above
  - One storing lists of IDs of pages that have been freed in a given transaction
  - Old, freed pages are used in preference to new pages, so the DB file size remains relatively static over time
  - No compaction or garbage collection phase is ever needed

SYMAS™
The LDAP guys.

Free Space Management

Meta Page	Meta Page
Pgno: 0 Misc... TXN: 0 FRoot: EMPTY DRoot: EMPTY	Pgno: 1 Misc... TXN: 0 FRoot: EMPTY DRoot: EMPTY

Free Space Management

Meta Page	Meta Page	Data Page
Pgno: 0 Misc... TXN: 0 FRoot: EMPTY DRoot: EMPTY	Pgno: 1 Misc... TXN: 0 FRoot: EMPTY DRoot: EMPTY	Pgno: 2 Misc... offset: 4000 1,foo

Free Space Management

Meta Page	Meta Page	Data Page
Pgno: 0 Misc... TXN: 0 FRoot: EMPTY DRoot: EMPTY	Pgno: 1 Misc... TXN: 1 FRoot: EMPTY DRoot: 2	Pgno: 2 Misc... offset: 4000 1,foo

Free Space Management

Meta Page	Meta Page	Data Page	Data Page
Pgno: 0 Misc... TXN: 0 FRoot: EMPTY DRoot: EMPTY	Pgno: 1 Misc... TXN: 1 FRoot: EMPTY DRoot: 2	Pgno: 2 Misc... offset: 4000 1,foo	Pgno: 3 Misc... offset: 4000 offset: 3000 2,bar 1,foo

Free Space Management

Meta Page	Meta Page	Data Page	Data Page	Data Page
Pgno: 0 Misc... TXN: 0 FRoot: EMPTY DRoot: EMPTY	Pgno: 1 Misc... TXN: 1 FRoot: EMPTY DRoot: 2	Pgno: 2 Misc... offset: 4000 1,foo	Pgno: 3 Misc... offset: 4000 offset: 3000 2,bar 1,foo	Pgno: 4 Misc... offset: 4000 txn 2,page 2

Free Space Management

Meta Page	Meta Page	Data Page	Data Page	Data Page
Pgno: 0 Misc... TXN: 2 FRoot: 4 DRoot: 3	Pgno: 1 Misc... TXN: 1 FRoot: EMPTY DRoot: 2	Pgno: 2 Misc... offset: 4000 1,foo	Pgno: 3 Misc... offset: 4000 offset: 3000 2,bar 1,foo	Pgno: 4 Misc... offset: 4000 txn 2,page 2

The LDAP guys.

Free Space Management

Meta Page	Meta Page	Data Page	Data Page	Data Page
Pgno: 0 Misc... TXN: 2 FRoot: 4 DRoot: 3	Pgno: 1 Misc... TXN: 1 FRoot: EMPTY DRoot: 2	Pgno: 2 Misc... offset: 4000 1,foo	Pgno: 3 Misc... offset: 4000 offset: 3000 2,bar 1,foo	Pgno: 4 Misc... offset: 4000 txn 2,page 2

Data Page
Pgno: 5 Misc... offset: 4000 offset: 3000 2,bar 1,blah

Free Space Management

Meta Page	Meta Page	Data Page	Data Page	Data Page
Pgno: 0 Misc... TXN: 2 FRoot: 4 DRoot: 3	Pgno: 1 Misc... TXN: 1 FRoot: EMPTY DRoot: 2	Pgno: 2 Misc... offset: 4000 1,foo	Pgno: 3 Misc... offset: 4000 offset: 3000 2,bar 1,foo	Pgno: 4 Misc... offset: 4000 txn 2,page 2

Data Page	Data Page
Pgno: 5 Misc... offset: 4000 offset: 3000 2,bar 1,blah	Pgno: 6 Misc... offset: 4000 offset: 3000 txn 3,page 3,4 txn 2,page 2

The LDAP guys.

Free Space Management

Meta Page	Meta Page	Data Page	Data Page	Data Page
Pgno: 0 Misc... TXN: 2 FRoot: 4 DRoot: 3	Pgno: 1 Misc... TXN: 3 FRoot: 6 DRoot: 5	Pgno: 2 Misc... offset: 4000 1,foo	Pgno: 3 Misc... offset: 4000 offset: 3000 2,bar 1,foo	Pgno: 4 Misc... offset: 4000 txn 2,page 2

Data Page	Data Page
Pgno: 5 Misc... offset: 4000 offset: 3000 2,bar 1,blah	Pgno: 6 Misc... offset: 4000 offset: 3000 txn 3,page 3,4 txn 2,page 2

Free Space Management

Meta Page	Meta Page	Data Page	Data Page	Data Page
Pgno: 0 Misc... TXN: 2 FRoot: 4 DRoot: 3	Pgno: 1 Misc... TXN: 3 FRoot: 6 DRoot: 5	Pgno: 2 Misc... offset: 4000 offset: 3000 2,xyz 1,blah	Pgno: 3 Misc... offset: 4000 offset: 3000 2,bar 1,foo	Pgno: 4 Misc... offset: 4000 txn 2,page 2

Data Page	Data Page
Pgno: 5 Misc... offset: 4000 offset: 3000 2,bar 1,blah	Pgno: 6 Misc... offset: 4000 offset: 3000 txn 3,page 3,4 txn 2,page 2

Free Space Management

Meta Page	Meta Page	Data Page	Data Page	Data Page
Pgno: 0	Pgno: 1	Pgno: 2	Pgno: 3	Pgno: 4
Misc...	Misc...	Misc...	Misc...	Misc...
TXN: 2	TXN: 3	offset: 4000	offset: 4000	offset: 4000
FRoot: 4	FRoot: 6	offset: 3000	offset: 3000	
DRoot: 3	DRoot: 5	2,xyz	2,bar	
		1,blah	1,foo	txn 2,page 2

Data Page	Data Page	Data Page
Pgno: 5	Pgno: 6	Pgno: 7
Misc...	Misc...	Misc...
offset: 4000	offset: 4000	offset: 4000
offset: 3000	offset: 3000	offset: 3000
2,bar	txn 3,page 3,4	txn 4,page 5,6
1,blah	txn 2,page 2	txn 3,page 3,4

The LDAP guys.

Free Space Management

Meta Page	Meta Page	Data Page	Data Page	Data Page
Pgno: 0 Misc... TXN: 4 FRoot: 7 DRoot: 2	Pgno: 1 Misc... TXN: 3 FRoot: 6 DRoot: 5	Pgno: 2 Misc... offset: 4000 offset: 3000 2,xyz 1,blah	Pgno: 3 Misc... offset: 4000 offset: 3000 2,bar 1,foo	Pgno: 4 Misc... offset: 4000 txn 2,page 2

Data Page	Data Page	Data Page
Pgno: 5 Misc... offset: 4000 offset: 3000 2,bar 1,blah	Pgno: 6 Misc... offset: 4000 offset: 3000 txn 3,page 3,4 txn 2,page 2	Pgno: 7 Misc... offset: 4000 offset: 3000 txn 4,page 5,6 txn 3,page 3,4

Free Space Management

- Caveat: If a read transaction is open on a particular version of the DB, that version and every version after it are excluded from page reclaiming
- Thus, long-lived read transactions should be avoided, otherwise the DB file size may grow rapidly, devolving into the Append-Only behavior until the transactions are closed

Transaction Handling

- LMDB supports a single writer concurrent with many readers
  - A single mutex serializes all write transactions
  - The mutex is shared/multiprocess
- Readers run lockless and never block
  - But for page reclamation purposes, readers are tracked
- Transactions are stamped with an ID which is a monotonically increasing integer
  - The ID is only incremented for Write transactions that actually modify data
  - If a Write transaction is aborted, or committed with no changes, the same ID will be reused for the next Write transaction

Transaction Handling

- Transactions take a snapshot of the currently valid meta page at the beginning of the transaction
- No matter what write transactions follow, a read transaction's snapshot will always point to a valid version of the DB
- The snapshot is totally isolated from subsequent writes
- This provides the Consistency and Isolation in ACID semantics

The LDAP guys.

Transaction Handling

- The currently valid meta page is chosen based on the greatest transaction ID in each meta page
  - The meta pages are page and CPU cache aligned
  - The transaction ID is a single machine word
  - The update of the transaction ID is atomic
  - Thus, the Atomicity semantics of transactions are guaranteed

Transaction Handling

- During Commit, the data pages are written and then synchronously flushed before the meta page is updated
  - Then the meta page is written synchronously
  - Thus, when a commit returns "success", it is guaranteed that the transaction has been written intact
  - This provides the Durability semantics
  - If the system crashes before the meta page is updated, then the data updates are irrelevant

The LDAP guys.

Transaction Handling

- For tracking purposes, Readers must acquire a slot in the readers table
  - The readers table is also in a shared memory map, but separate from the main data map
  - This is a simple array recording the Process ID, Thread ID, and Transaction ID of the reader
  - The first time a thread opens a read transaction, it must acquire a mutex to reserve a slot in the table
  - The slot ID is stored in Thread Local Storage; subsequent read transactions performed by the thread need no further locks

Transaction Handling

- Write transactions use pages from the free list before allocating new disk pages
  - Pages in the free list are used in order, oldest transaction first
  - The readers table must be scanned to see if any reader is referencing an old transaction
  - The writer doesn't need to lock the reader table when performing this scan - readers never block writers
    - The only consequence of scanning with no locks is that the writer may see stale data
    - This is irrelevant, newer readers are of no concern; only the oldest readers matter

Special Features

- Explicit Key Types
  - Support for reverse byte order comparisons, as well as native binary integer comparisons
  - Minimizes the need for custom key comparison functions, allows DBs to be used safely by applications without special knowledge
    - Reduces the danger of corruption that Berkeley databases were vulnerable to, when custom key comparators were used

Special Features

- Append Mode
  - Ultra-fast writes when keys are added in sequential order
  - Bypasses standard page-split algorithm when pages are filled, avoids unnecessary memcpy's
  - Allows databases to be bulk loaded at the full sequential write speed of the underlying storage system

Special Features

- Reserve Mode
  - Allocates space in write buffer for data of user-specified size, returns address
  - Useful for data that is generated dynamically instead of statically copied
  - Allows generated data to be written directly to DB output buffer, avoiding unnecessary memcpy

Special Features

- Fixed Mapping
  - Uses a fixed address for the memory map
  - Allows complex pointer-based data structures to be stored directly with minimal serialization
  - Objects using persistent addresses can thus be read back with no deserialization
  - Useful for object-oriented databases, among other purposes

Special Features

- Sub-databases
  - Store multiple independent named B+trees in a single LMDB environment
  - A SubDB is simply a key/data pair in the main DB, where the data item is the root node of another tree
  - Allows many related databases to be managed easily
    - Used in back-mdb for the main data and all of the associated indices
    - Used in SQLightning for multiple tables and indices

The LDAP guys.

Special Features

- Sorted Duplicates
  - Allows multiple data values for a single key
  - Values are stored in sorted order, with customizable comparison functions
  - When the data values are all of a fixed size, the values are stored contiguously, with no extra headers
    - maximizes storage efficiency and performance
  - Implemented by the same code as SubDB support
    - maximum coding efficiency

Special Features

- Atomic Hot Backup
  - The entire database can be backed up live
  - No need to stop updates while backups run
  - The backup runs at the maximum speed of the target storage medium
  - Essentially: write(outfd, map, mapsize);
    - no memcpy's in or out of user space
    - pure DMA from the database to the backup

Results

- Support for LMDB is already available for many open source projects:
  - OpenLDAP slapd - back-mdb backend
  - Cyrus SASL - sasldb plugin
  - Heimdal Kerberos - hdb plugin
  - OpenDKIM - main data store
  - SQLite3 - replacing the original Btree code
  - MemcacheDB - replacing BerkeleyDB
  - Postfix - replacing BerkeleyDB
  - CfEngine - replacing Tokyo Cabinet/QDBM

Results

- Wrappers for many other languages besides C are available:
  - C++
  - Erlang
  - Lua
  - Python
  - Ruby
  - Java wrapper being developed

The LDAP guys.

Results

- Coming Soon
  - Riak - Erlang LMDB wrapper already available
  - SQLite4 - in progress
  - MariaDB - in progress
  - HyperDex - in progress
  - XDAndroid - port of Android using SQLite3 based on LMDB
  - Mozilla/Firefox - using SQLite3 based on LMDB

Results

- In OpenLDAP slapd
  - LMDB reads are 5-20x faster than BerkeleyDB
  - Writes are 2-5x faster than BerkeleyDB
  - Consumes 1/4 as much RAM as BerkeleyDB
- In SQLite3
  - Writes are 10-25x faster than stock SQLite3
  - Reads .. performance is overshadowed by SQL inefficiency

Results

- In MemcacheDB
  - LMDB reads are 2-200x faster than BerkeleyDB
  - Writes are 5-900x faster than BerkeleyDB
  - Multi-thread reads are 2-8x faster than pure-memory Memcached
    - Single-thread reads are about the same
    - Writes are about 20% slower

Results

- Full benchmark reports are available on the LMDB page
  - http://www.symas.com/mdb/
- Supported builds of LMDB-based packages available from Symas
  - http://www.symas.com/
  - OpenLDAP, Cyrus-SASL, Heimdal Kerberos
  - MemcacheDB coming soon

Microbenchmark Results

- Comparisons based on Google's LevelDB
- Also tested against Kyoto Cabinet's TreeDB, SQLite3, and BerkeleyDB
- Tested using RAM filesystem (tmpfs), reiserfs on SSD, and multiple filesystems on HDD
  - btrfs, ext2, ext3, ext4, jfs, ntfs, reiserfs, xfs, zfs
  - ext3, ext4, jfs, reiserfs, xfs also tested with external journals

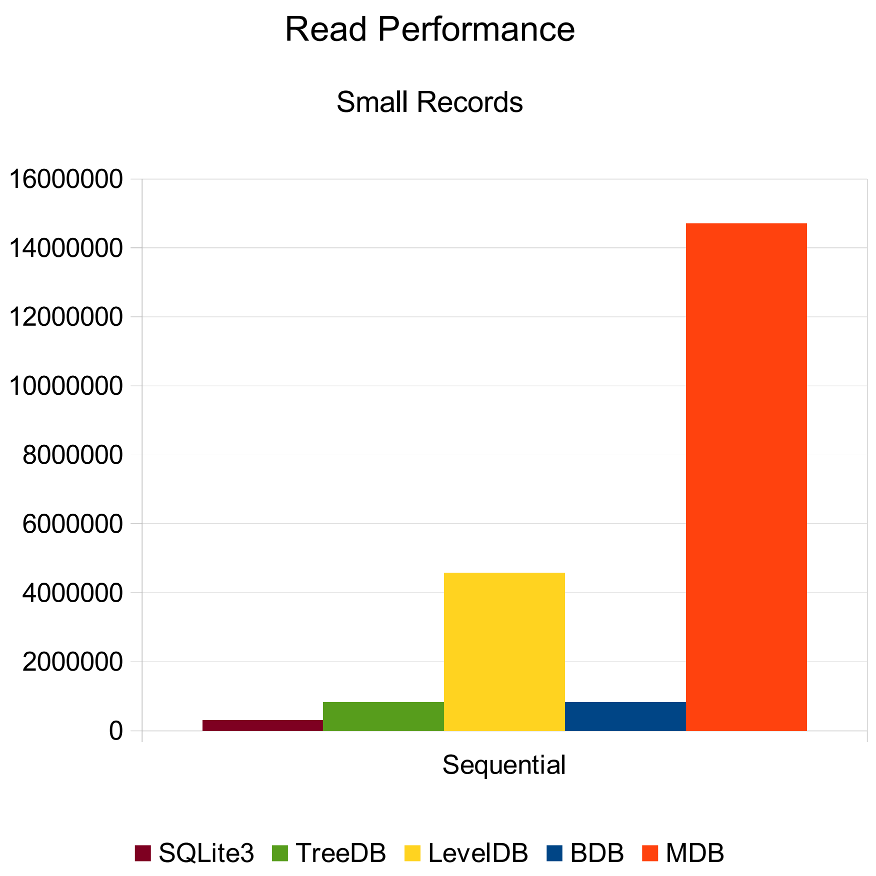
Microbenchmark Results

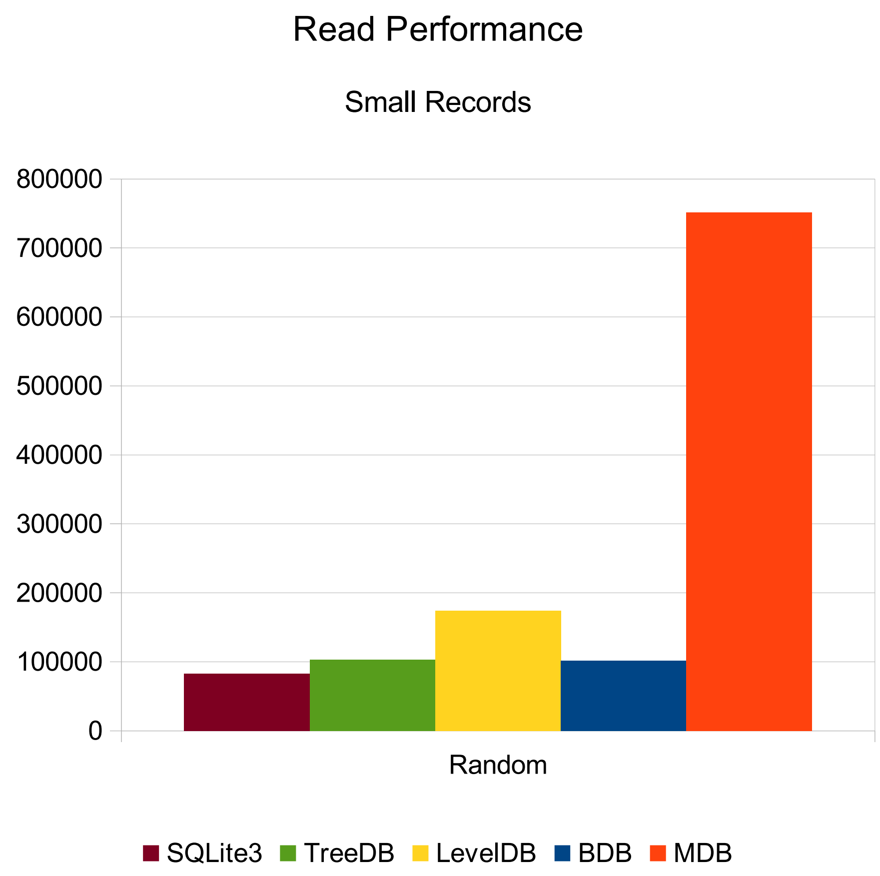
- Relative Footprint

text	data	bss	dec	hex	filename
272247	1456	328	274031	42e6f	db_bench
1675911	2288	304	1678503	199ca7	db_bench_bdb
90423	1508	304	92235	1684b	db_bench_mdb
653480	7768	1688	662936	a2764	db_bench_sqlite3
296572	4808	1096	302476	49d8c	db_bench_tree_db

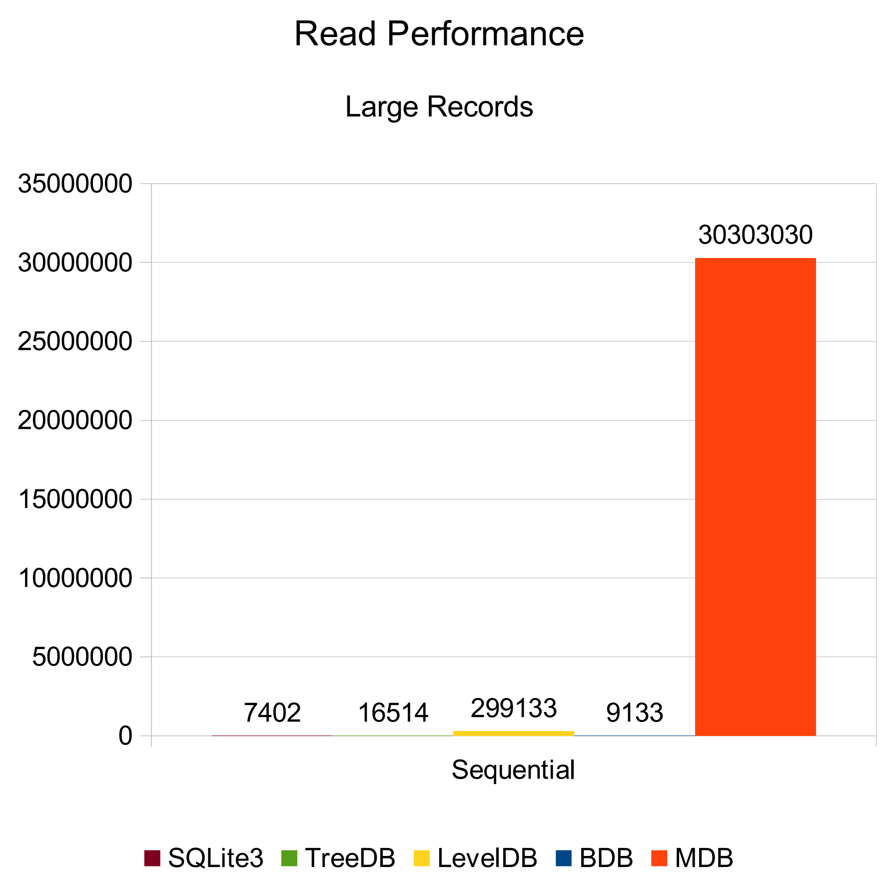
- Clearly LMDB has the smallest footprint
  - Carefully written C code beats C++ every time

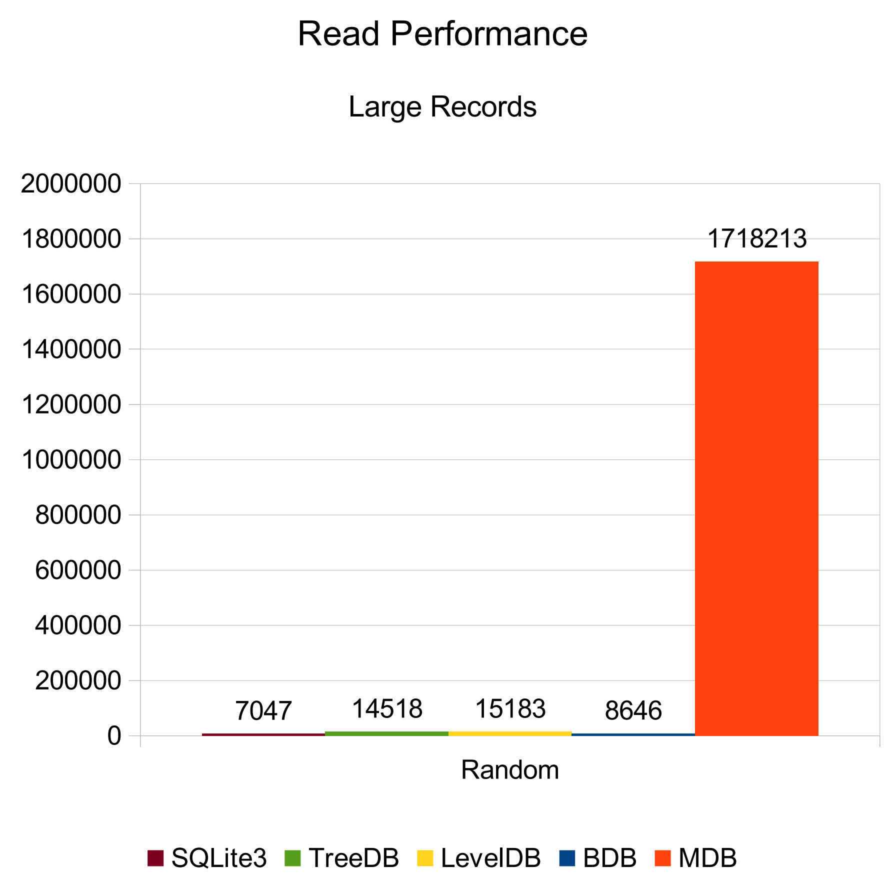
Microbenchmark Results





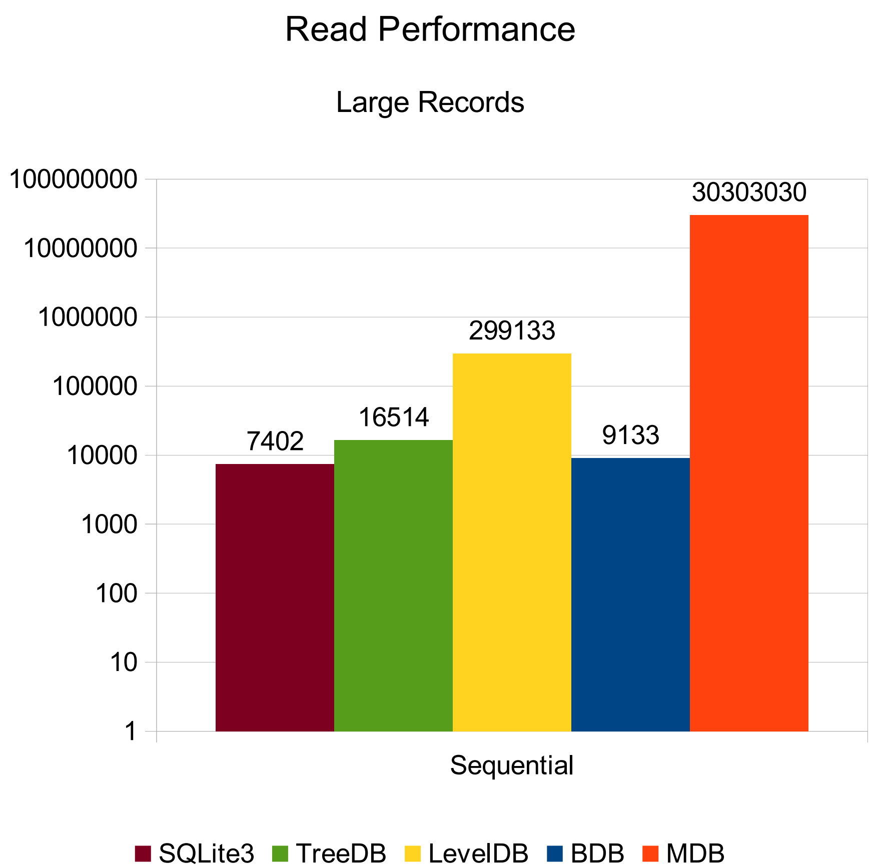
Microbenchmark Results

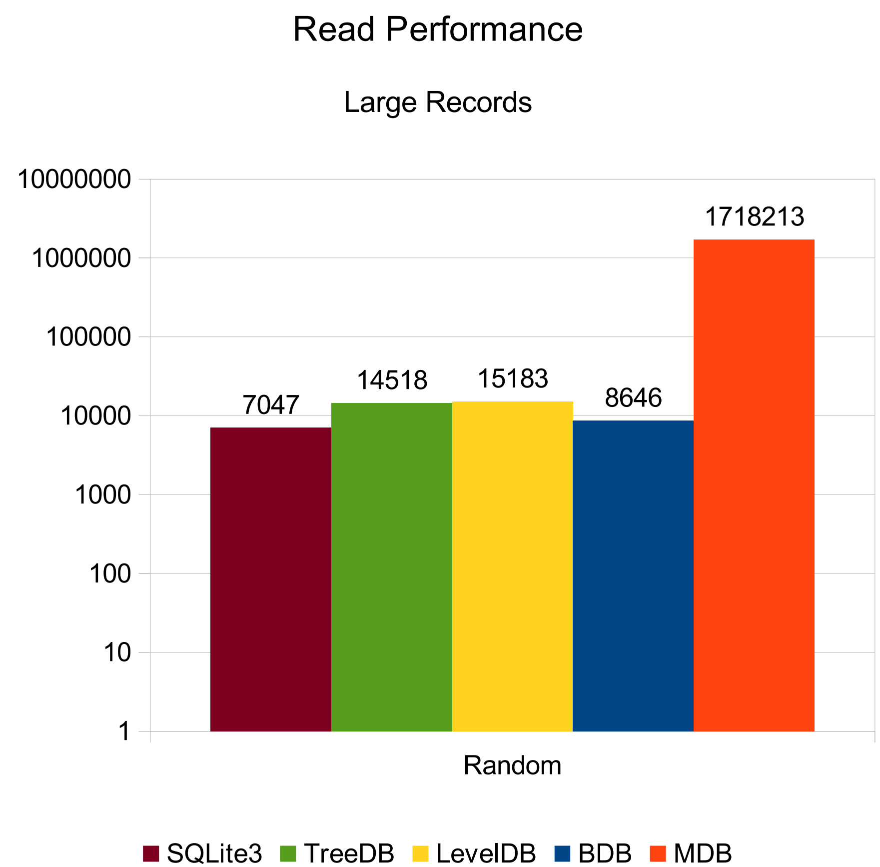




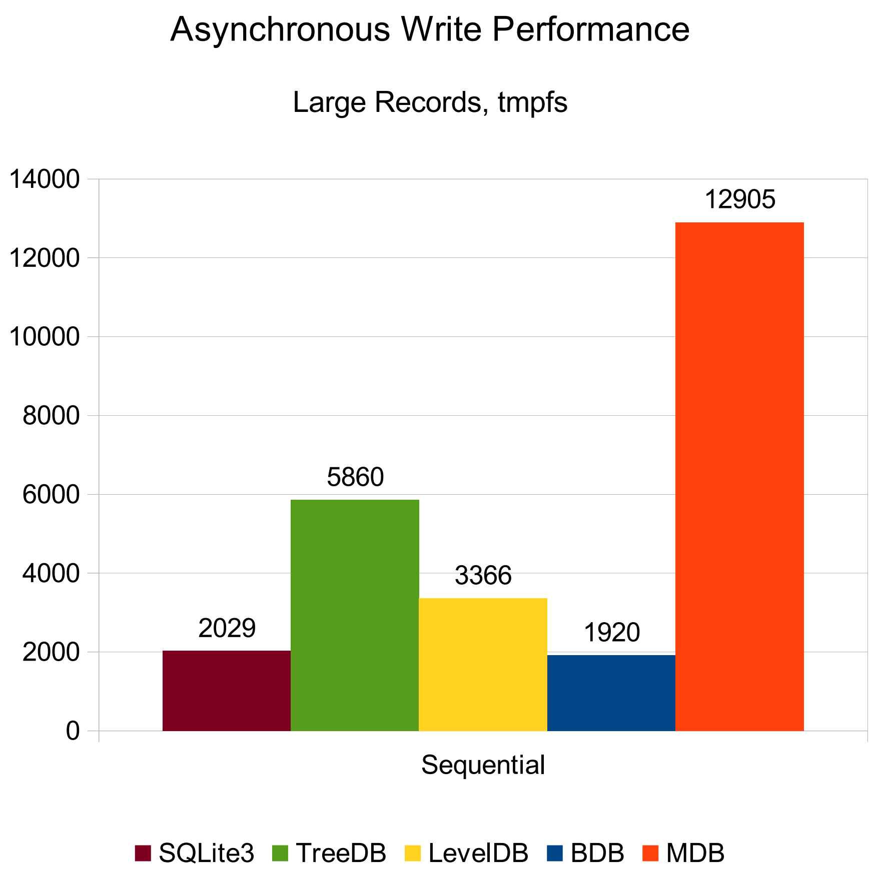
Microbenchmark Results

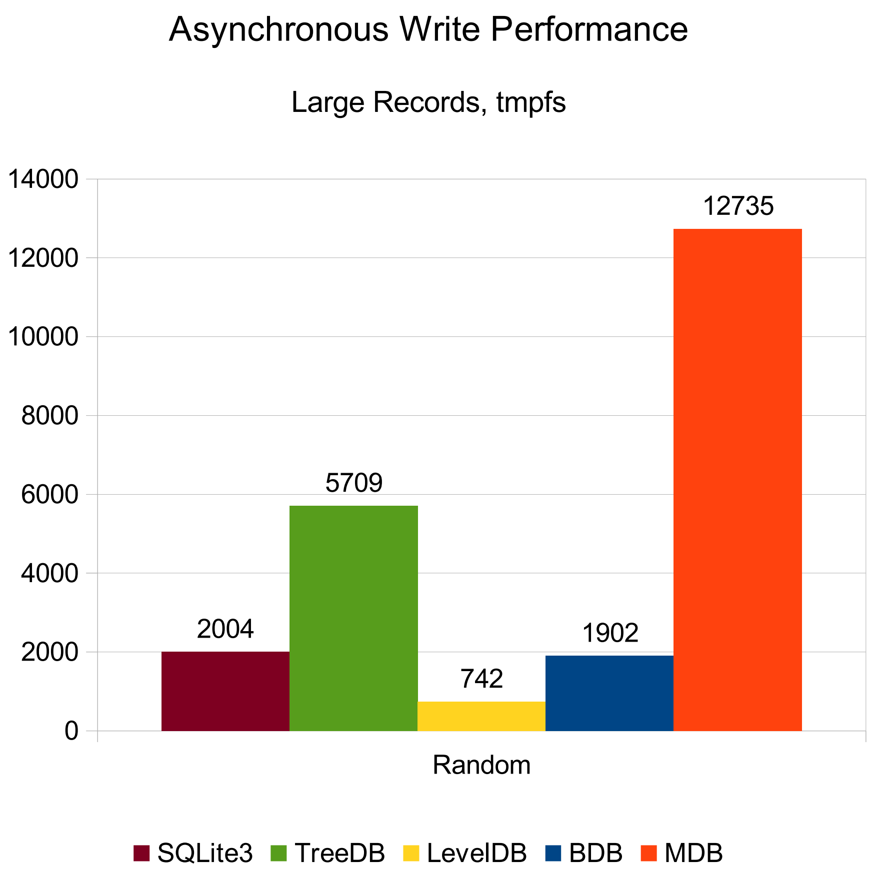
Log Scale



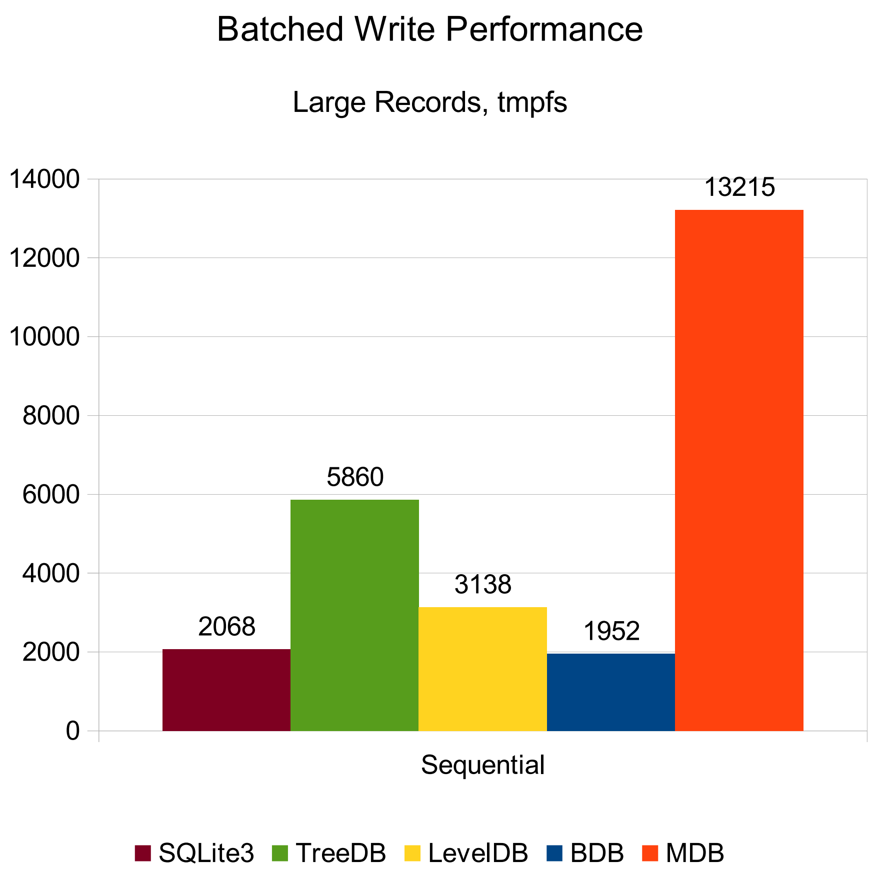


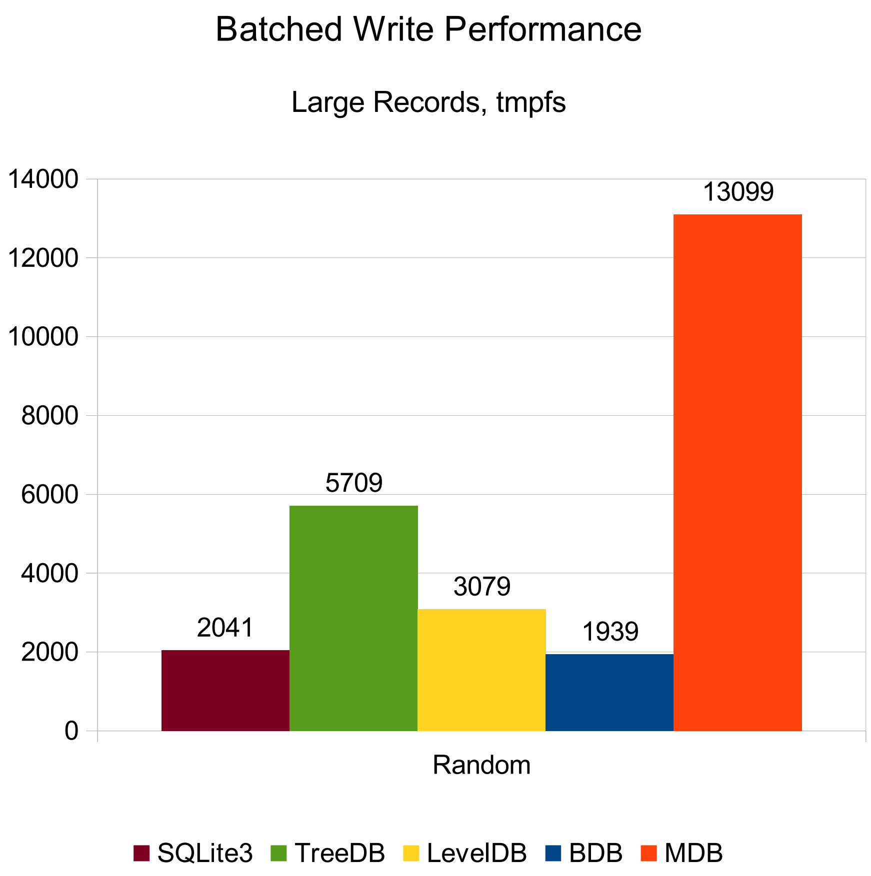
Microbenchmark Results



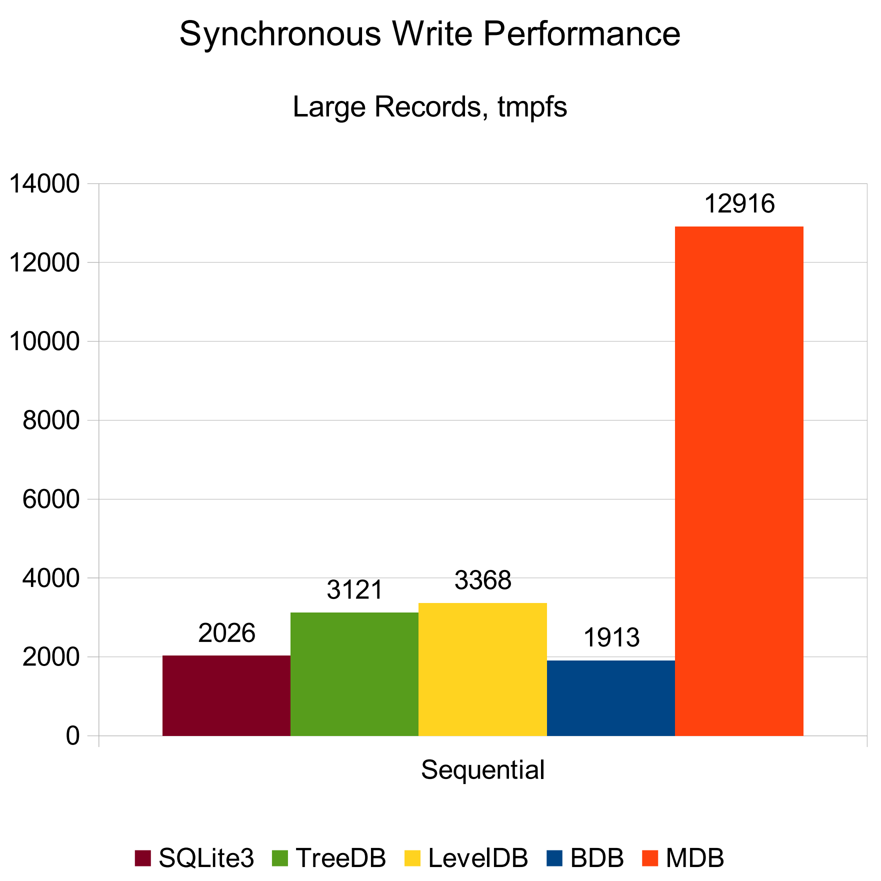


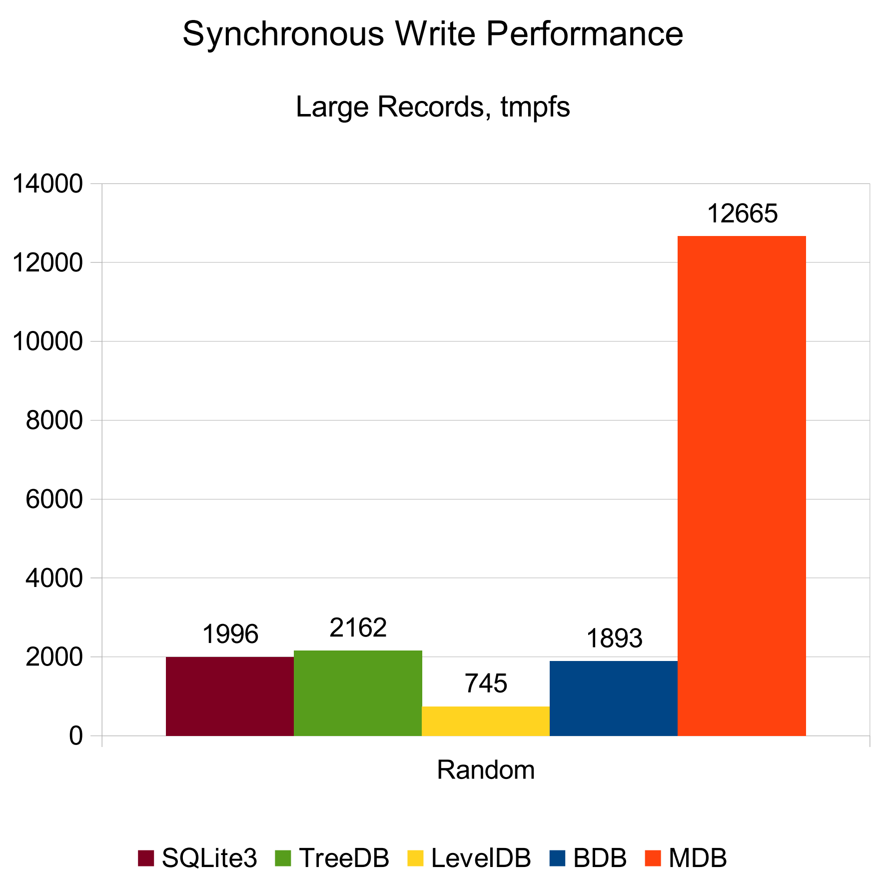
Microbenchmark Results



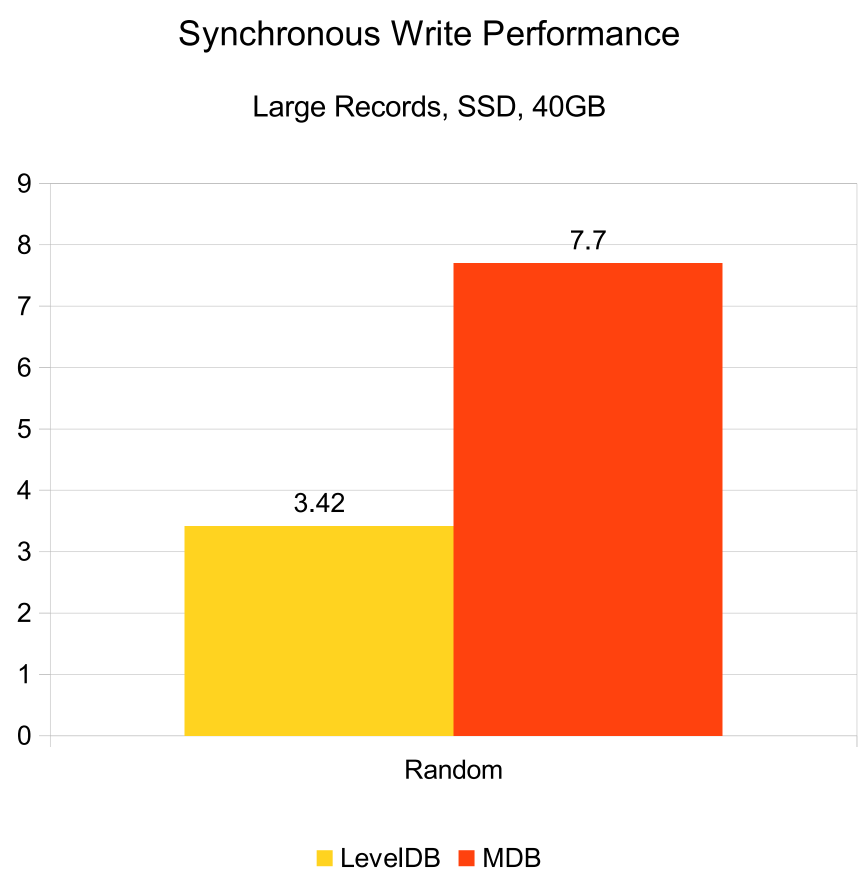


Microbenchmark Results





Microbenchmark Results

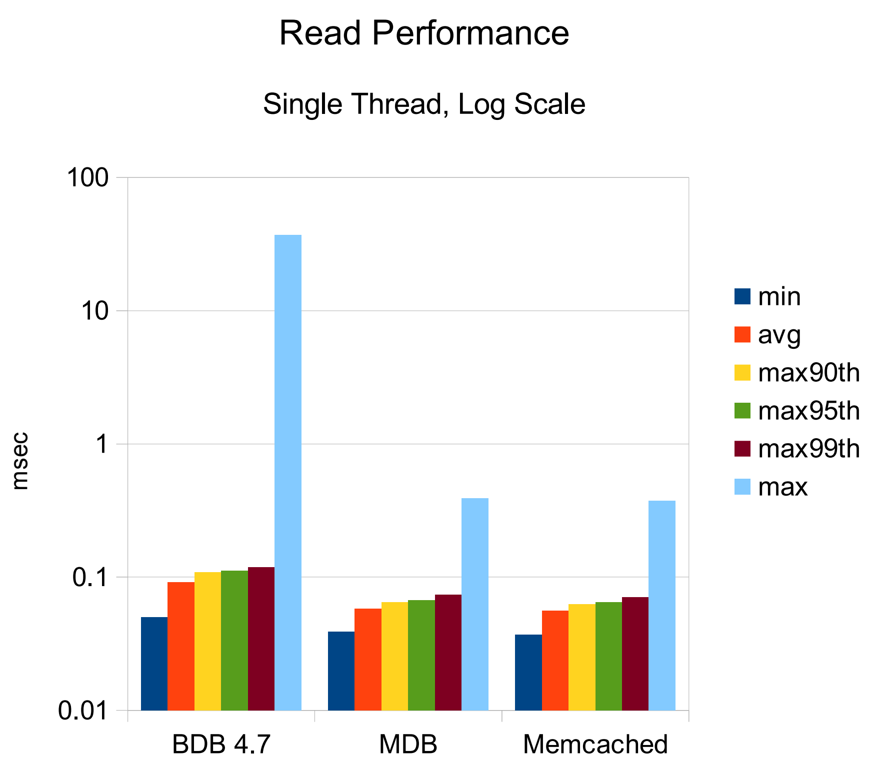


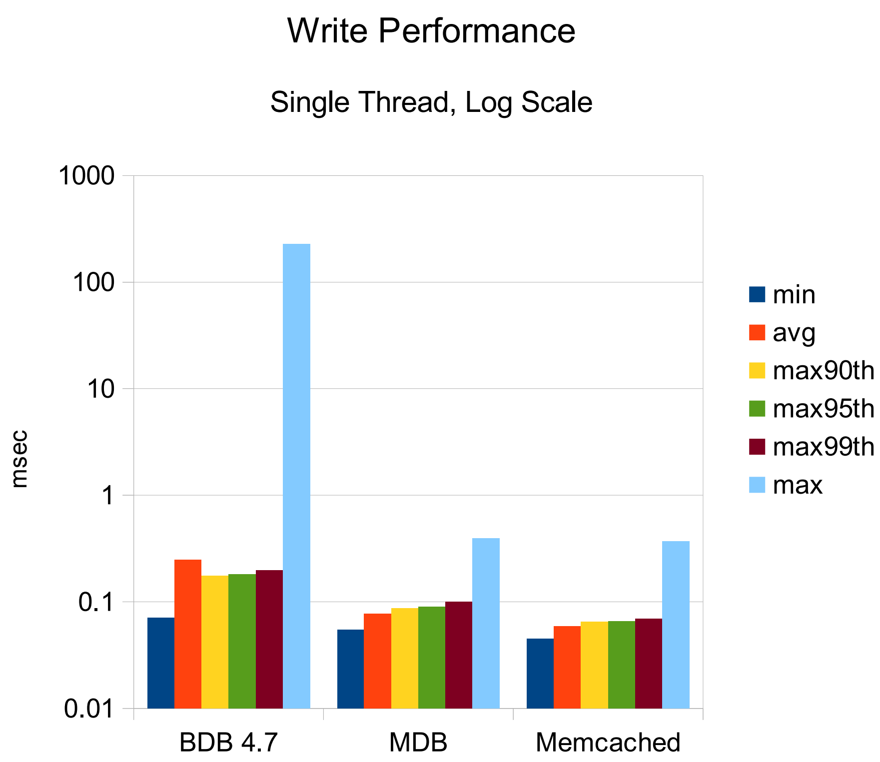
- Test random write performance when DB is 5x larger than RAM
- Supposedly a best case for LevelDB and worst case for B-trees
- Result in MB/sec, higher is better

Benchmarking...

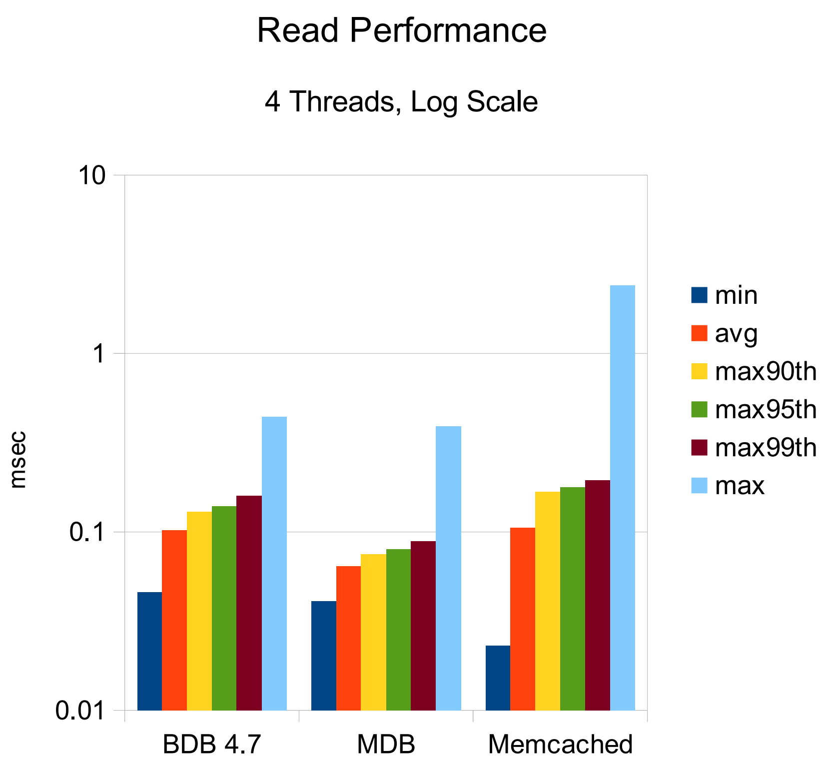
- LMDB in real applications
  - MemcacheDB, tested with memcachetest
  - The OpenLDAP slapd server, using the back-mdb slapd backend

MemcacheDB

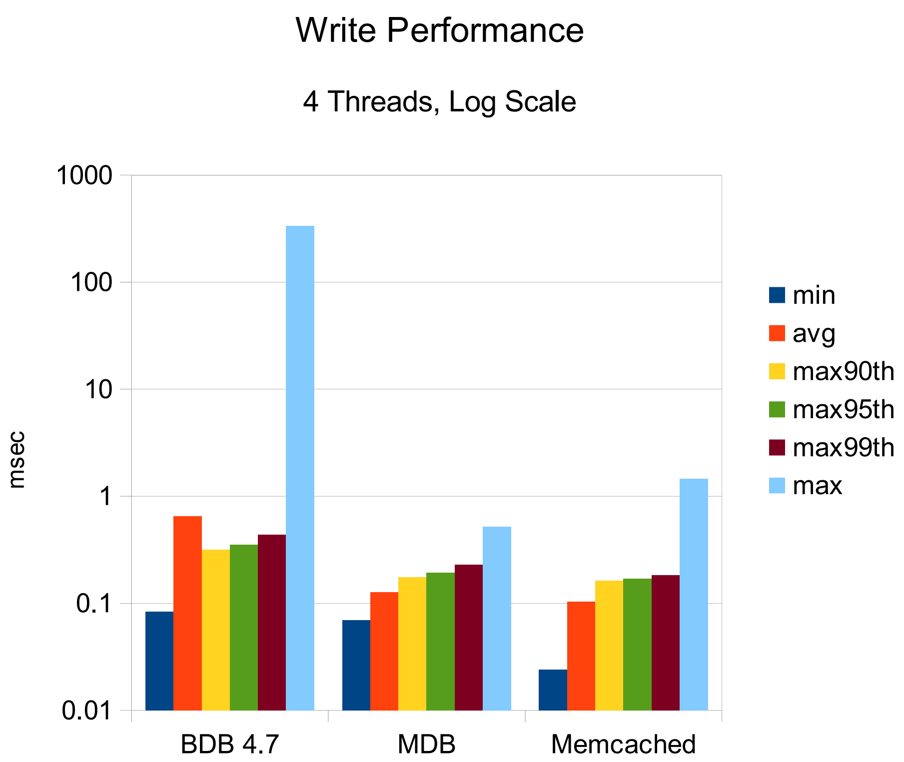

Read Performance
Single Thread, Log Scale
100
10
1
0.1
0.01
msec
BDB 4.7
MDB
Memcached
min
avg
max90th
max95th
max99th
max


Write Performance
Single Thread, Log Scale
1000
100
10
1
0.1
0.01
msec
BDB 4.7
MDB
Memcached
min
avg
max90th
max95th
max99th
max

MemcacheDB

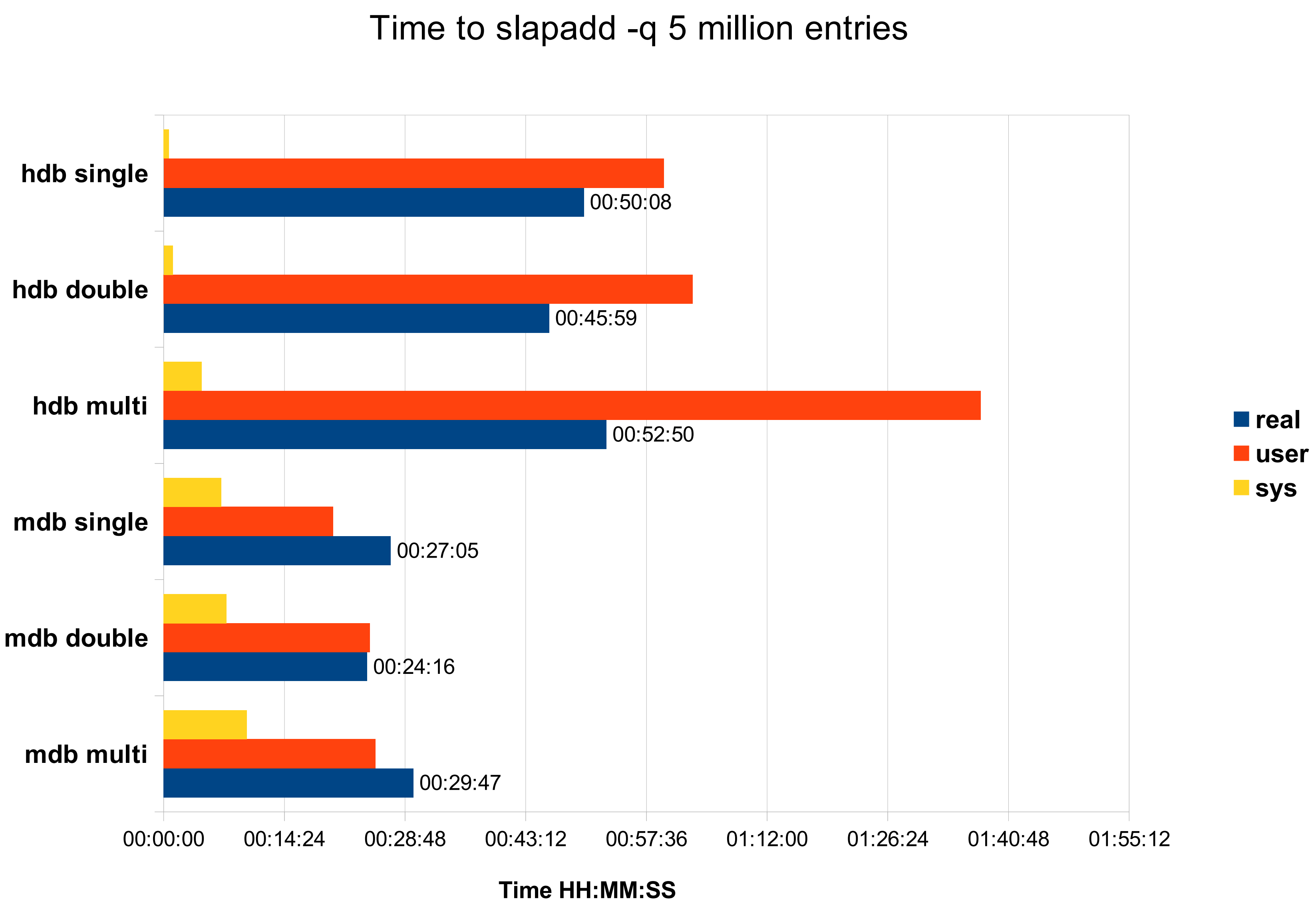


Read Performance
4 Threads, Log Scale
msec
10
1
0.1
0.01
BDB 4.7
MDB
Memcached
min
avg
max90th
max95th
max99th
max

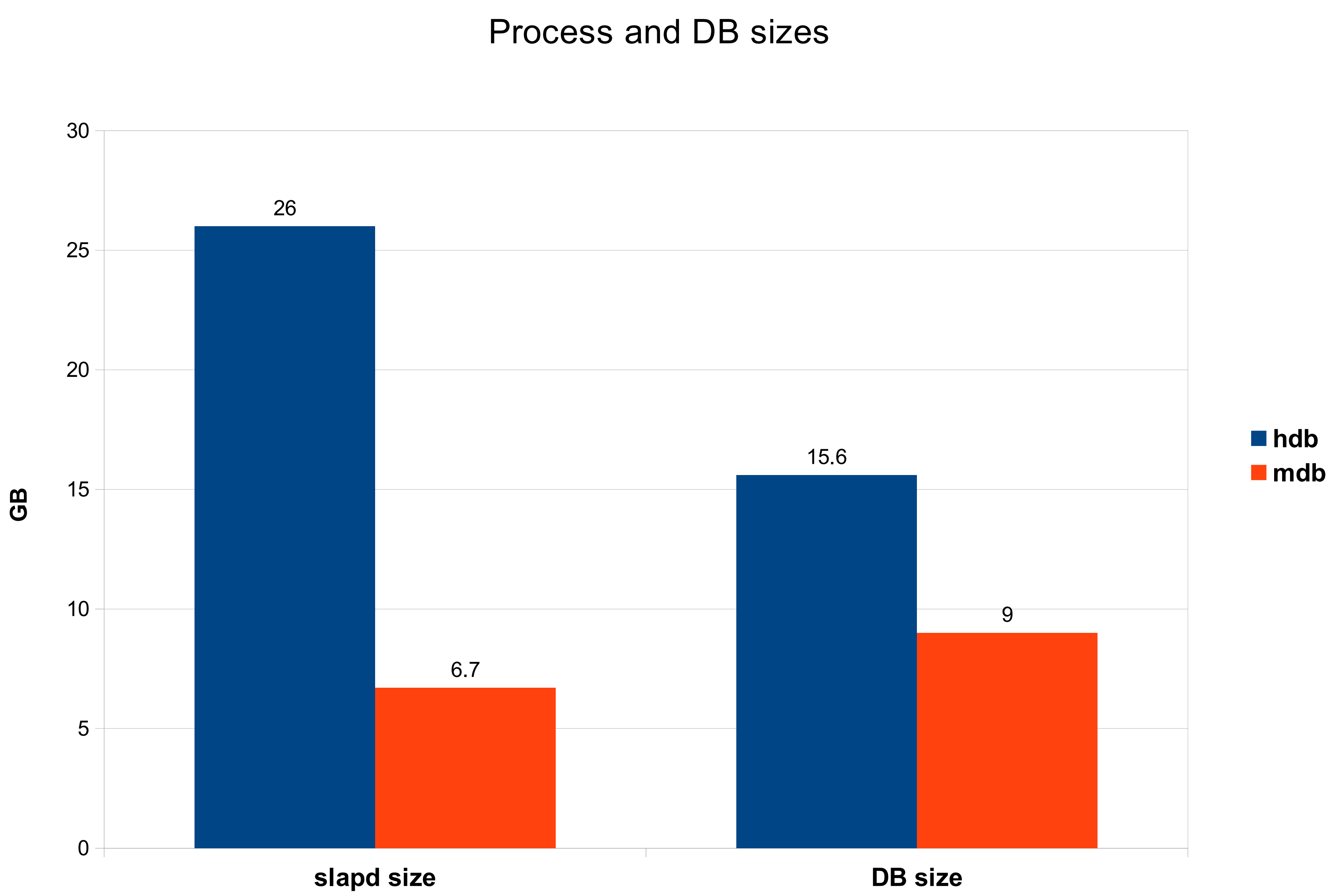


Write Performance
4 Threads, Log Scale
msec
1000
100
10
1
0.1
0.01
BDB 4.7
MDB
Memcached
min
avg
max90th
max95th
max99th
max

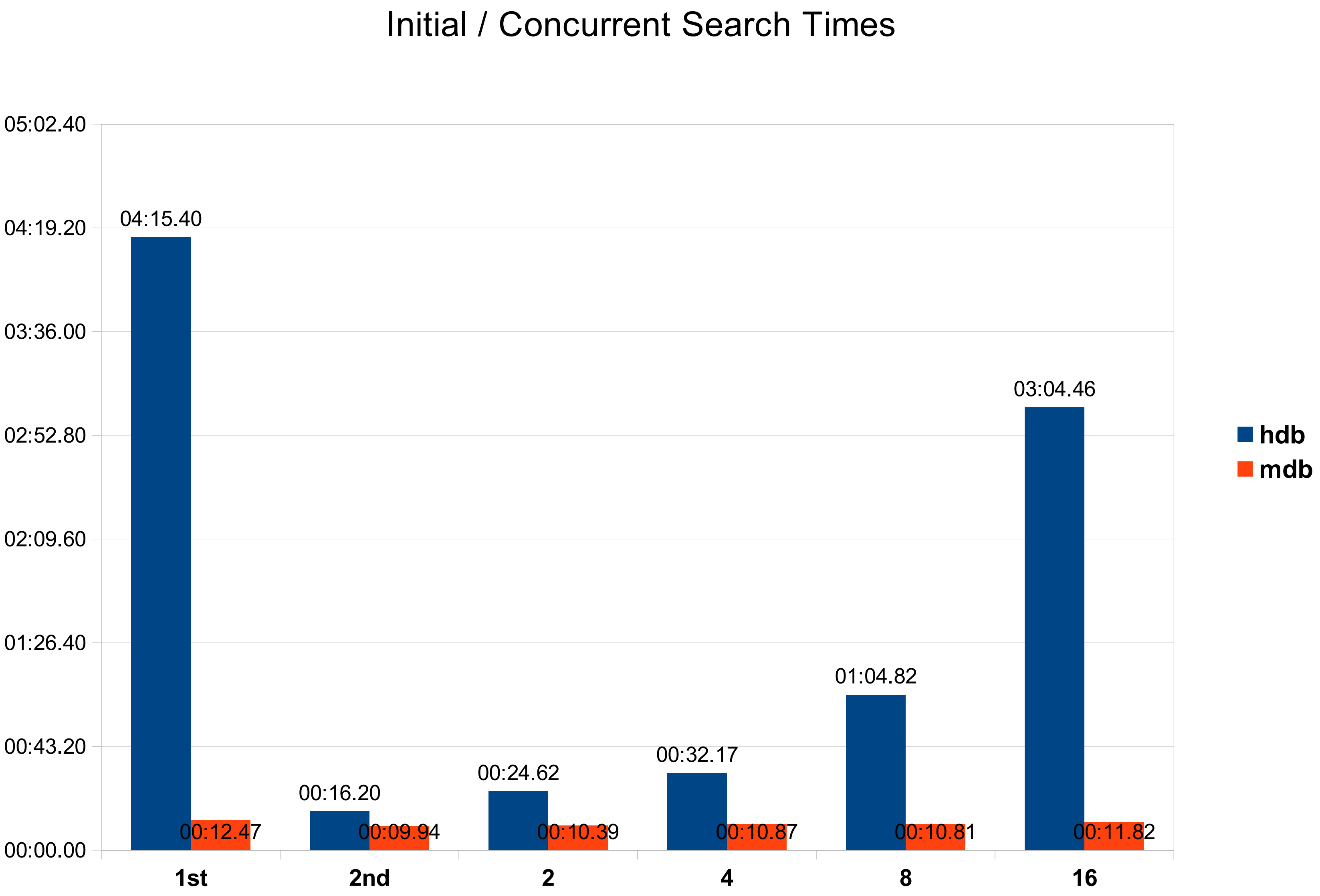
Slapd Results



Slapd Results

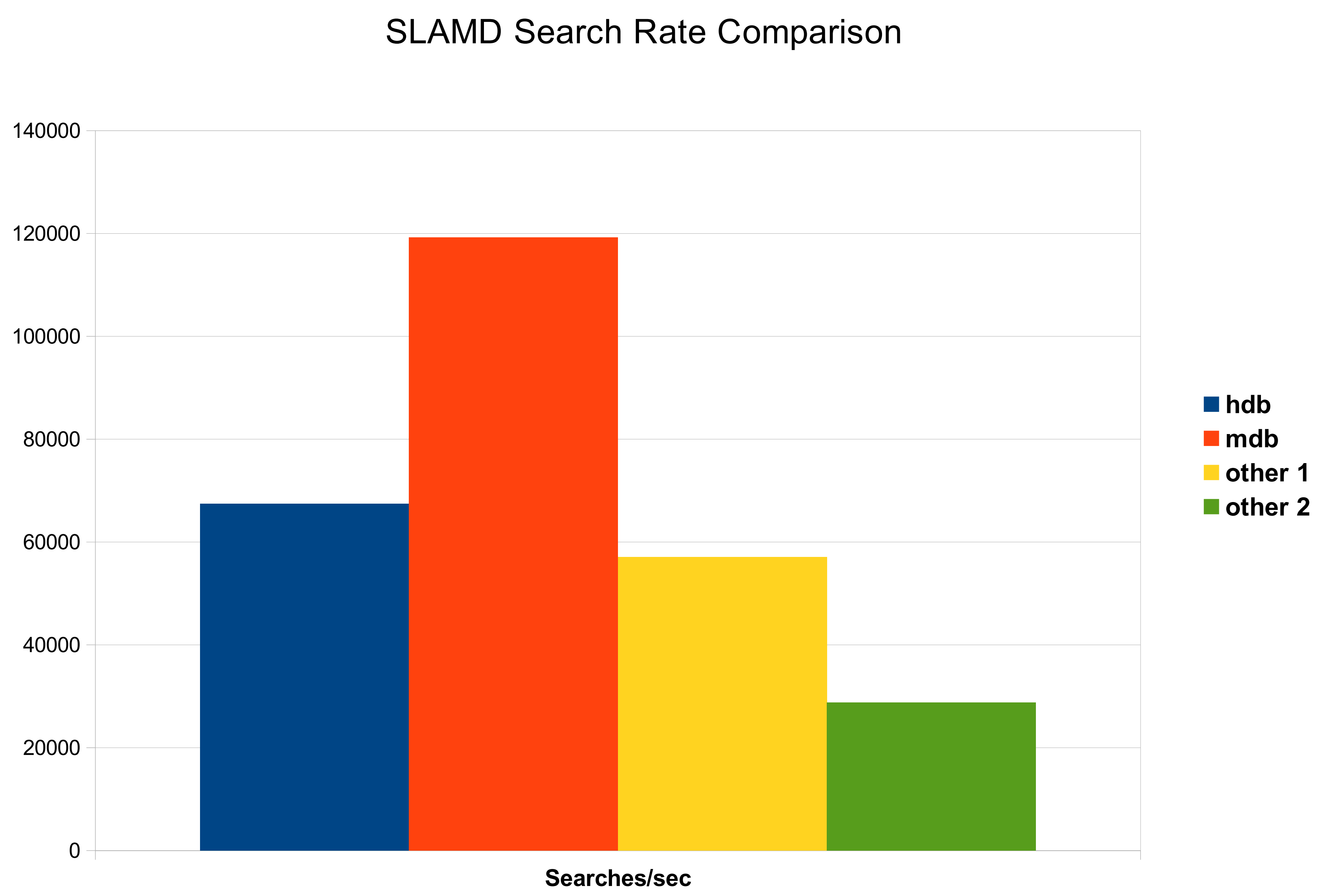


Slapd Results

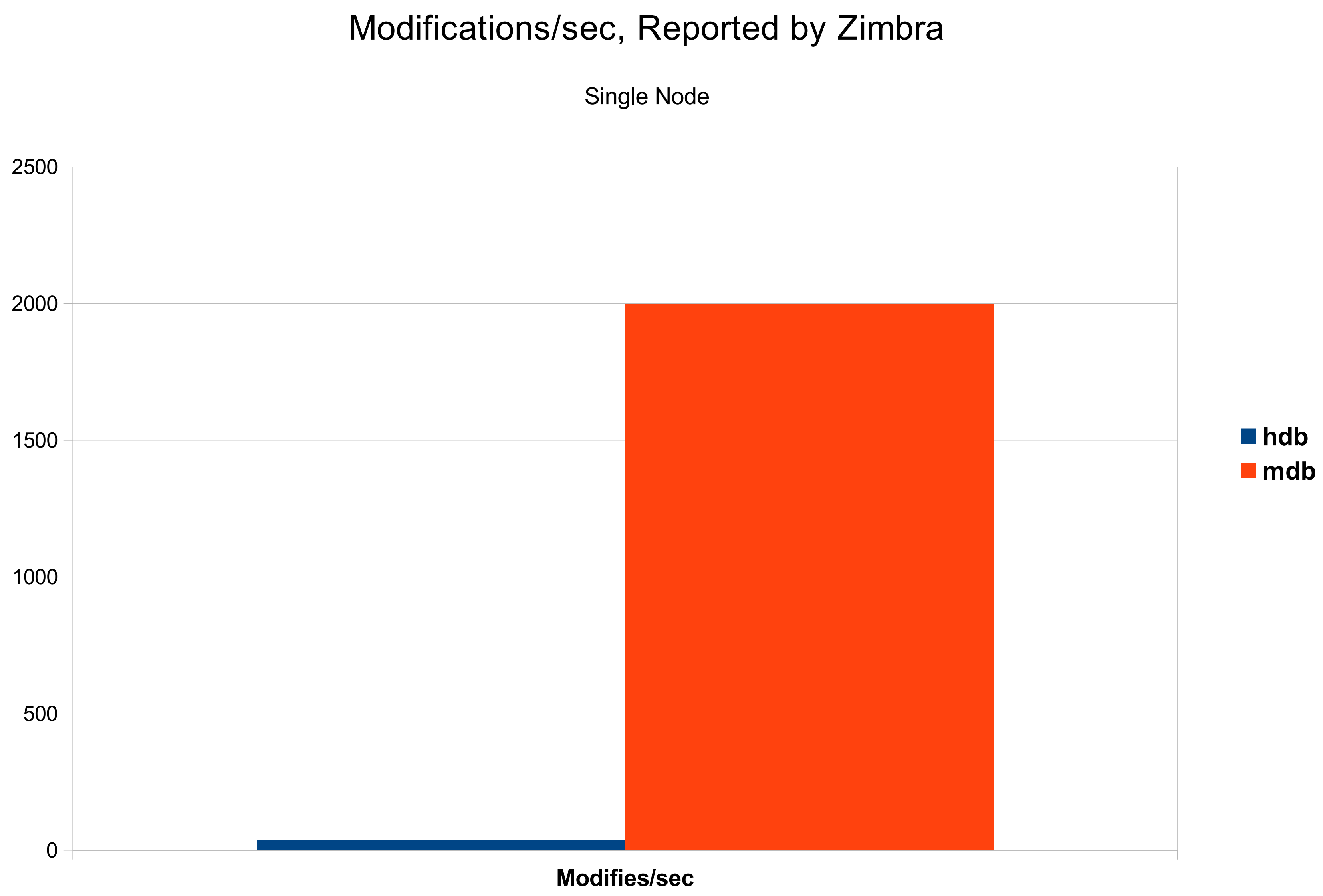


The LDAP guys.

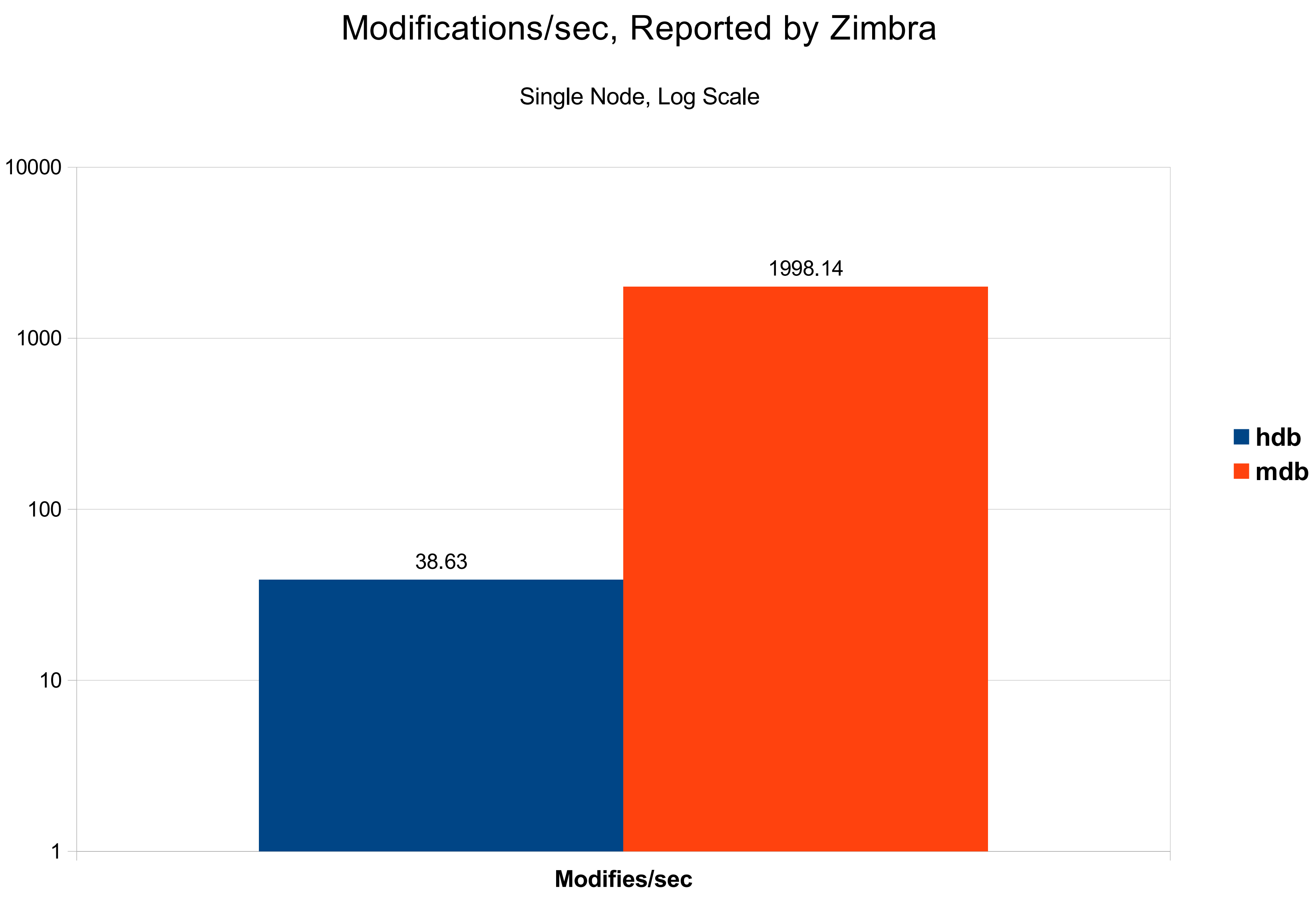
Slapd Results



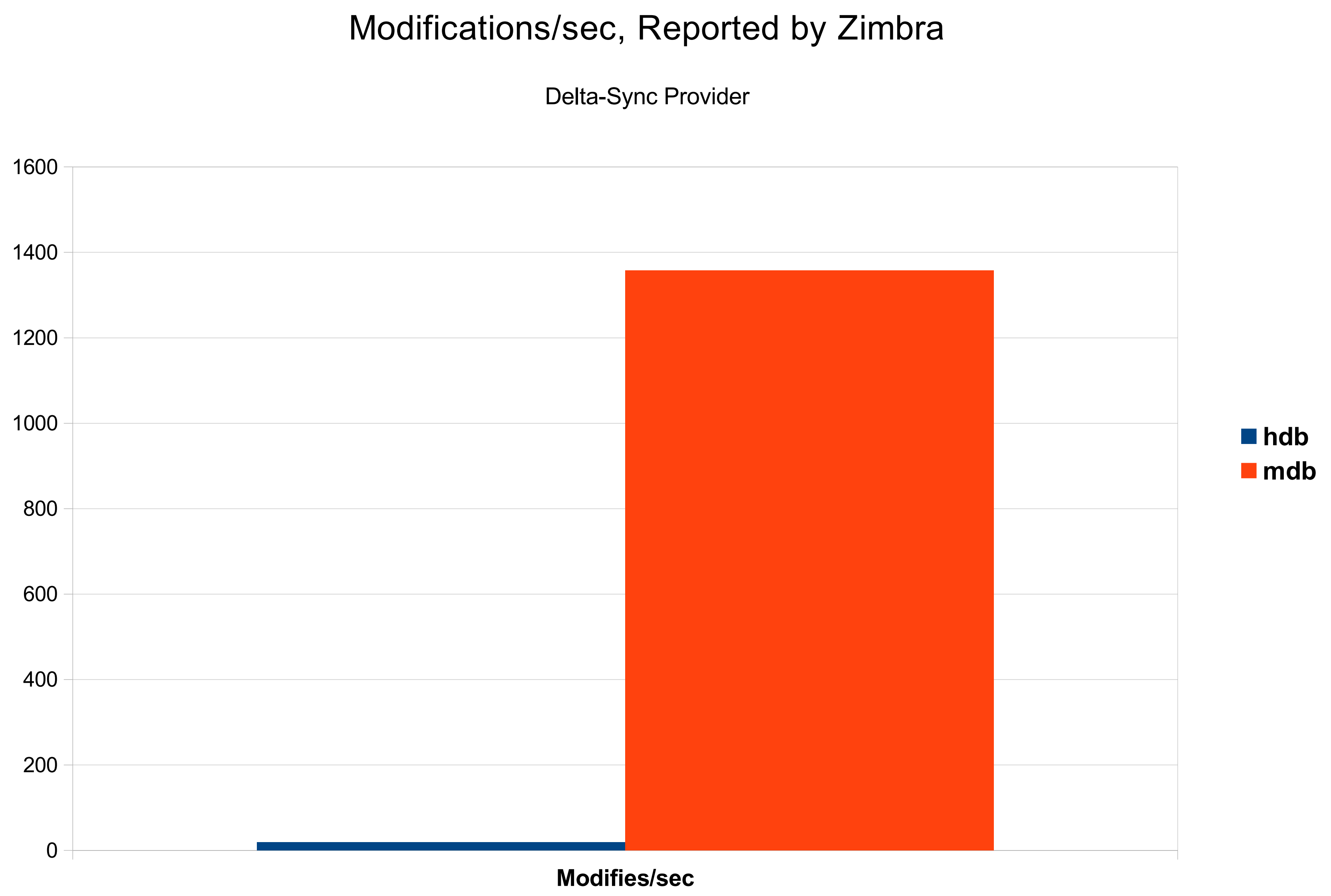
Slapd Results



Slapd Results



Slapd Results



Slapd Results