The Lightning Memory-Mapped Database

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

2015-11-18

HTTP://SYMAS.COM

OpenLDAP Project

- Open source code project
- Founded 1998
- Four core team members
- A dozen or so contributors
- Feature releases every 12-18 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
- No debt, no VC investments: self-funded

Intro

- Howard Chu
  - Founder and CTO Symas Corp.
  - Developing Free/Open Source software since 1980s
    - GNU compiler toolchain, e.g. "gmake -j", etc.
    - Many other projects...
  - Worked for NASA/JPL, wrote software for Space Shuttle, etc.

Topics

(1) Background

(2) Features

(3) Design Approach

(4) Internals

    (1) Dark Underside

(5) Special Features

(6) Results

LMDB

HTTP://SYMAS.COM

(1) Background

- API inspired by Berkeley DB (BDB)
  - OpenLDAP has used BDB extensively since 1999
  - Deep experience with pros and cons of BDB design and implementation
  - Omits BDB features that were found to be of no benefit
    - e.g. extensible hashing
  - Avoids BDB characteristics that were problematic
    - e.g. cache tuning, complex locking, transaction logs, recovery

(2) Features

LMDB 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 I$
- Runs on most modern OSs
  - Linux, Android, *BSD, MacOSX, iOS, 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 - Serializable
  - Nested transactions
  - Batched writes

Features

- Uses Copy-on-Write
  - Live data is never overwritten
  - DB 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

Features

- LMDB config is simple, e.g. slapd

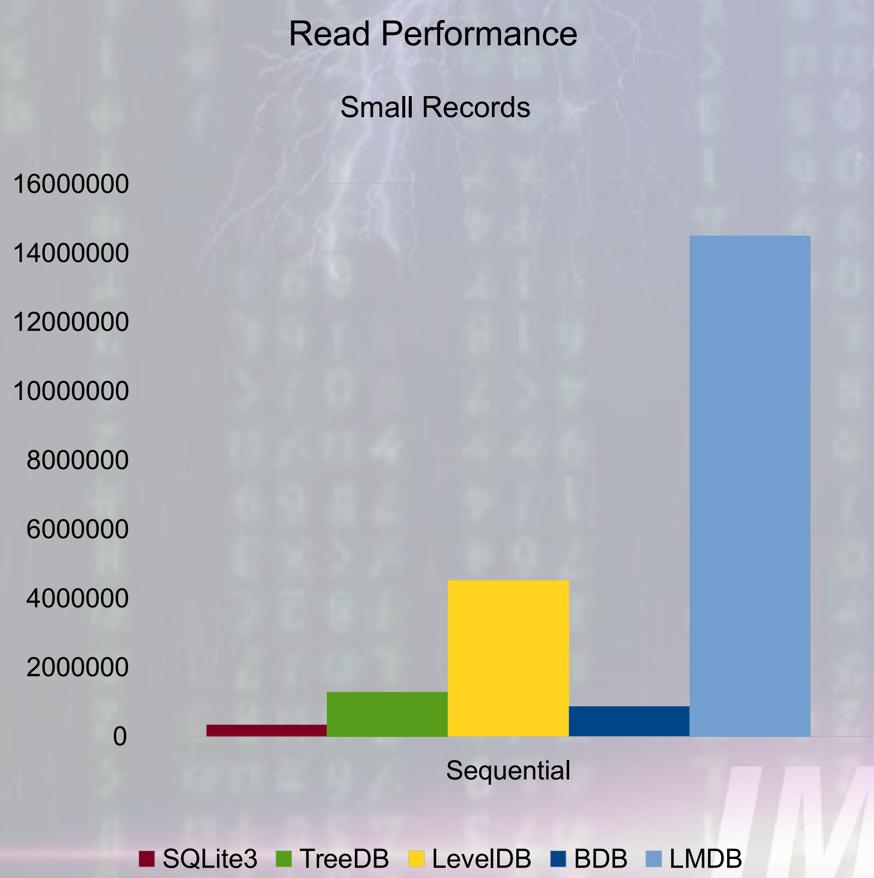
```
database mdb
directory /var/lib/ldap/data/mdb
maxsize 4294967296
```

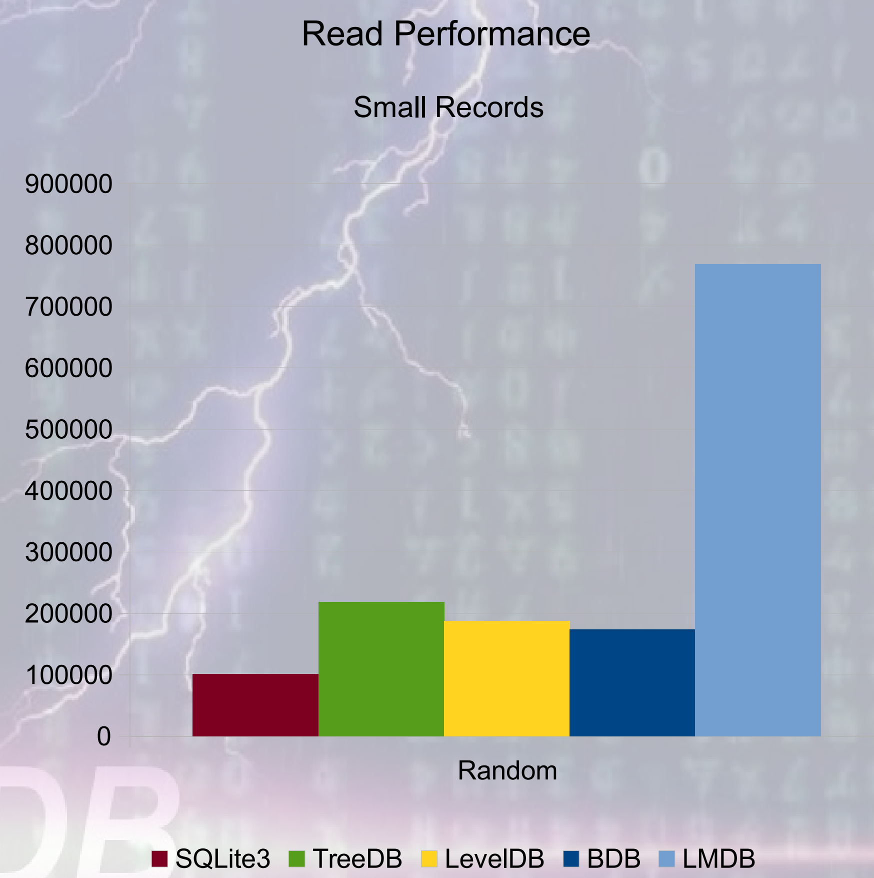
- BDB config is complex

```
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
```

Features

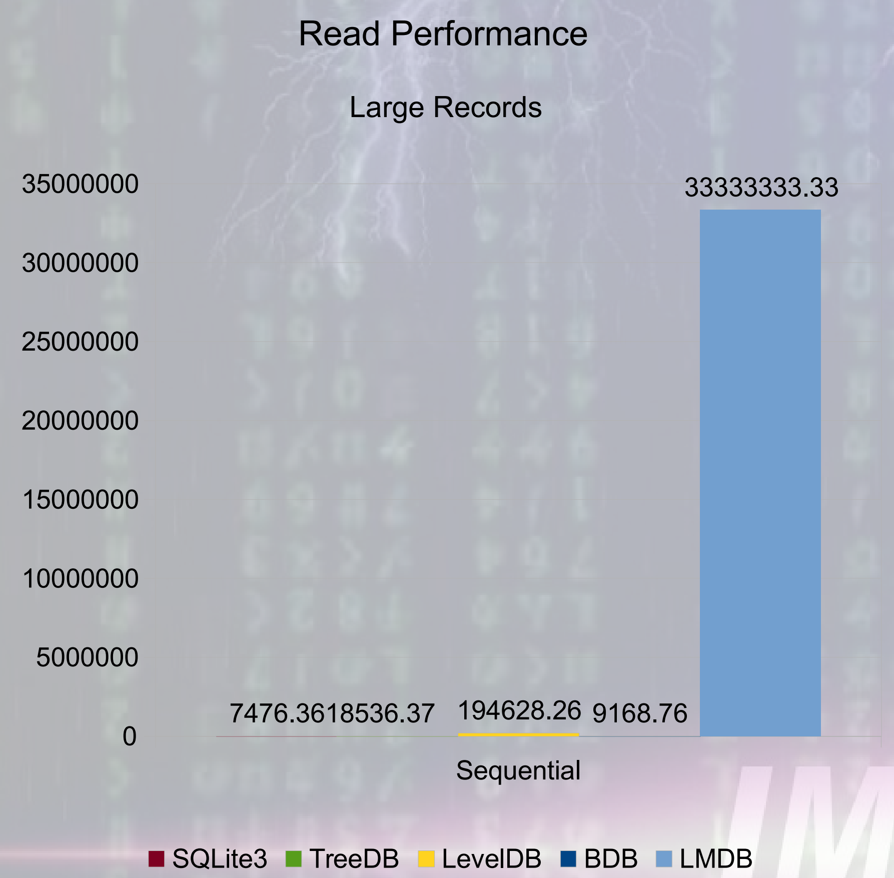
- World's fastest for reads - nothing else comes close

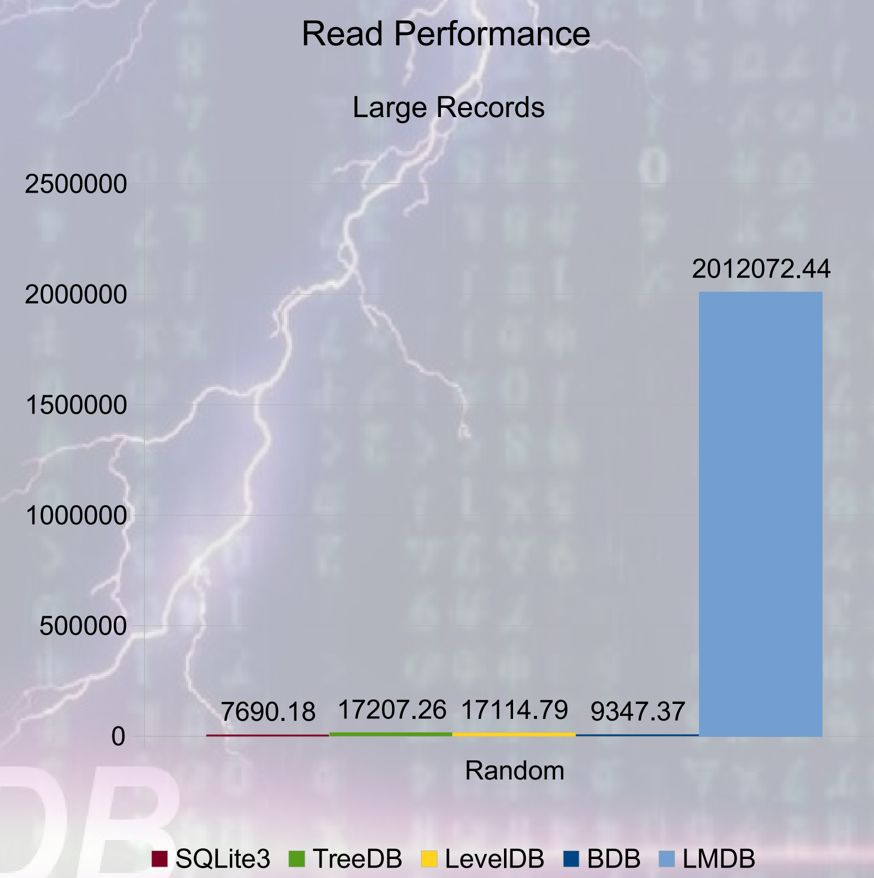




Features

- World's fastest for reads - nothing else comes close



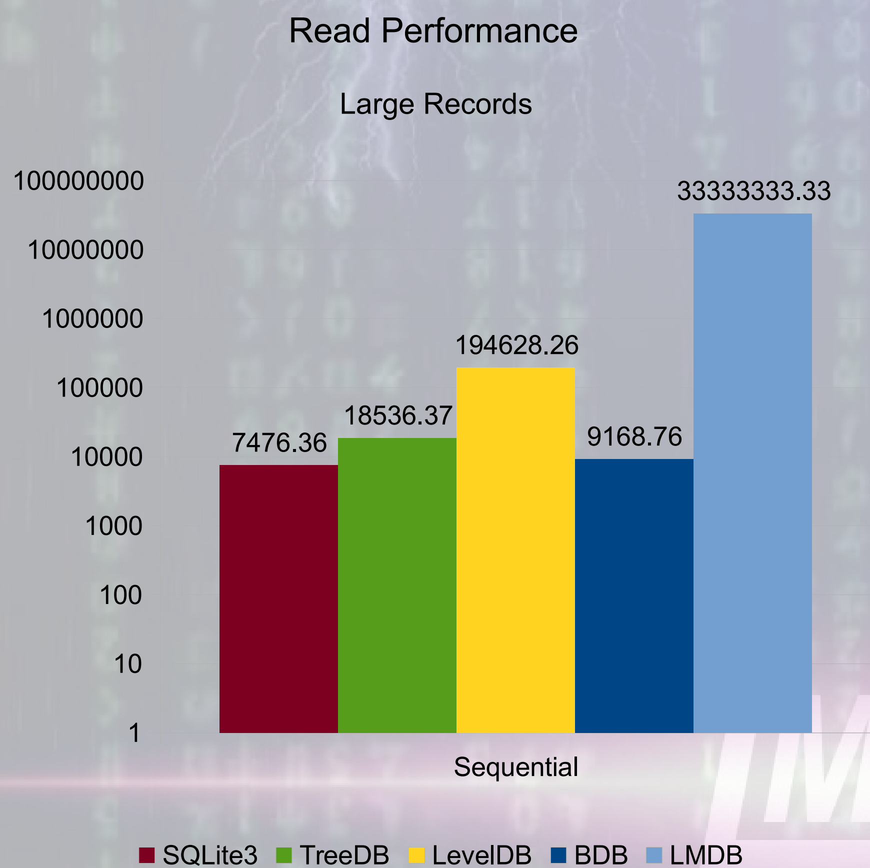


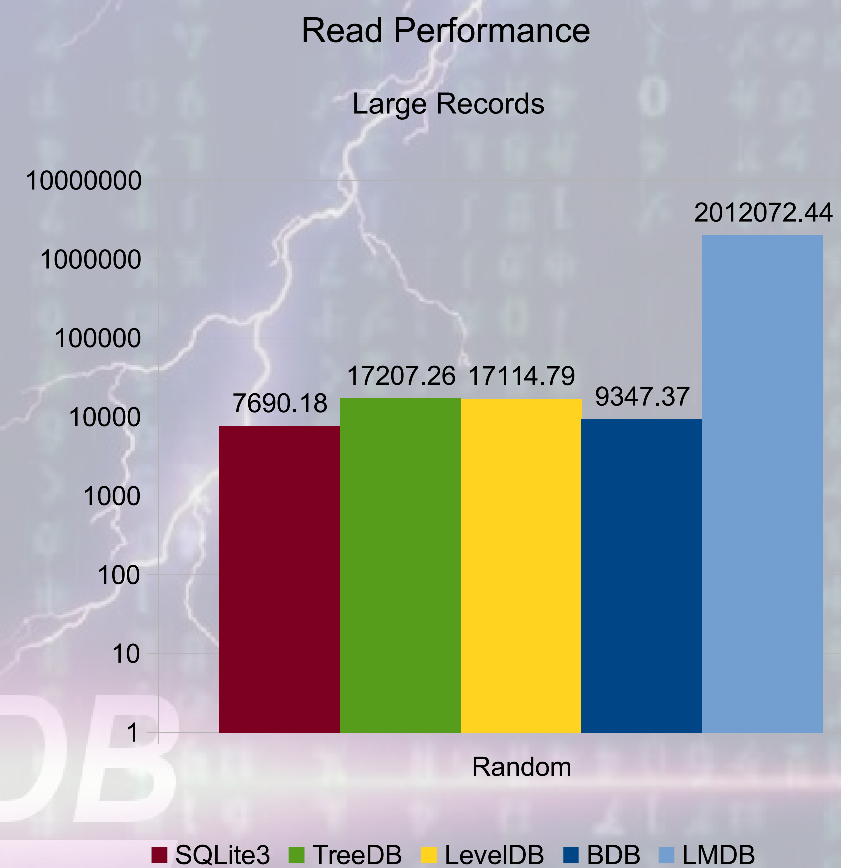
HTTP://SYMAS.COM

Features

- World's fastest for reads - nothing else comes close

Log Scale





Features

- Benchmarked exhaustively across multiple dimensions
  - Multiple filesystems
    - btrfs, ext2, ext3, ext4, jfs, ntfs, reiserfs, xfs, and zfs
    - HDD, SSD, PCIe NVME, etc.
  - Multiple malloc libraries
    - glibc, tcmalloc, jemalloc
  - Multiple compressors
    - bz2, zlib, lz4, lzma, lzo, snappy, zlib
  - Multiple server configs
    - VMs, physical servers
    - small to large RAM (32GB to 1TB)
    - 1 to 64 CPU cores
  - Multiple record sizes - 16 bytes to 100KB
  - Multiple DB sizes - in-memory to 50x RAM size

Features

- Proven crash-proof in independent studies
  - Usenix OSDI 2014
    - https://www.usenix.org/conference/osdi14/technical-sessions/presentation/pillai
    - https://www.usenix.org/conference/osdi14/technical-sessions/presentation/zheng_mai

(3) Design Approach

- Motivation - problems dealing with BDB
- Obvious Solutions
- Approach

Motivation

- Tuning complexity
- Cache inefficiency
- Lock management and deadlocks
- Logging and reliability

Motivation

- BDB slapd backend always required careful, complex tuning
  - Data comes through 3 separate layers of caches
  - Each layer has different size and speed traits
  - Balancing the 3 layers against each other can be a difficult juggling act
  - Performance without the backend caches is unacceptably slow - over an order of magnitude

Motivation

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

Motivation

- The caches were not always beneficial, and were sometimes detrimental
  - Data could exist in 3 places at once - filesystem, DB, and backend cache - 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 fragmentation in libc malloc

Motivation

- Cache management requires lock management
  - You must ensure that no one is modifying a cache entry while others are reading
  - Particularly painful if accesses collide with evictions
  - Deadlocks are a routine occurrence

Motivation

- BDB transactions are actually unreliable
  - BDB uses a file per database
  - File creation/deletion/rename ops in the filesystem cannot be atomically tied to the txn logs
  - Filesystem operation ordering cannot be guaranteed
  - Log file creation can fail, file descriptors can be exhausted, etc.

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

- File management is a hassle, so just use one file
  - No dependency on filesystem operation ordering
  - No danger of running out of descriptors in middle of operations

Obvious Solutions

- BDB supports MVCC, but still requires complex caching and locking
- To get the desired results, we need to abandon BDB
- Surveying the landscape revealed no other DB libraries with the desired characteristics
- Thus LMDB was created in 2011
  - "Lightning Memory-Mapped Database"
  - BDB is now deprecated in OpenLDAP

Design Approach

- Based on the "Single-Level Store" concept
  - Not new, first implemented in Multics in 1964
  - Access a database by mapping the entire DB into memory
  - Data fetches are satisfied by direct reference to the memory map; there is no intermediate page or buffer cache

Single-Level Store

- 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

Design Approach

- Uses a read-only memory map
  - Protects the DB structure from corruption due to stray writes in memory
  - Any attempts to write to the map will cause a SEGV, allowing immediate identification of software bugs
- Can optionally use a read-write mmap
  - Slight performance gain for fully in-memory data sets
  - Should only be used on fully-debugged application code

Design Approach

- Keith Bostic (BerkeleyDB author, personal email, 2008)
  - "The most significant problem with building an mmap'd back-end is implementing write-ahead-logging (WAL). (You probably know this, but just in case: the way databases usually guarantee consistency is by ensuring that log records describing each change are written to disk before their transaction commits, and before the database page that was changed. In other words, log record X must hit disk before the database page containing the change described by log record X.)
  - In Berkeley DB WAL is done by maintaining a relationship between the database pages and the log records. If a database page is being written to disk, there's a look-aside into the logging system to make sure the right log records have already been written. In a memory-mapped system, you would do this by locking modified pages into memory (mlock), and flushing them at specific times (msync), otherwise the VM might just push a database page with modifications to disk before its log record is written, and if you crash at that point it's all over but the screaming."

Design Approach

- Implement MVCC using copy-on-write
  - In-use data is never overwritten, modifications are performed by copying the data and modifying the copy
  - Since updates never alter existing data, the DB structure can never be corrupted by incomplete modifications
    - Write-ahead transaction logs are unnecessary
  - Readers always see a consistent snapshot of the DB, they are fully isolated from writers
    - Read accesses require no locks

MVCC Details

- "Full" MVCC can be extremely resource intensive
  - DBs typically store complete histories reaching far back into time
  - The volume of data grows extremely fast, and grows without bound unless explicit pruning is done
  - Pruning the data using garbage collection or compaction requires more CPU and I/O resources than the normal update workload
    - Either the server must be heavily over-provisioned, or updates must be stopped while pruning is done
  - Pruning requires tracking of in-use status, which typically involves reference counters, which require locking

Design Approach

- LMDB nominally maintains only two versions of the DB
  - Rolling back to a historical version is not interesting for OpenLDAP
  - Older versions can be held open longer by reader transactions
- LMDB maintains a free list tracking the IDs of unused pages
  - Old pages are reused as soon as possible, so data volumes don't grow without bound
- LMDB tracks in-use status without locks

Implementation Highlights

- LMDB library started from the append-only btree code written by Martin Hedenfalk for his ldapd, which is bundled in OpenBSD
  - Stripped out all the parts we didn't need (page cache management)
  - Borrowed a couple pieces from slapd for expedience
  - Changed from append-only to page-reclaiming
  - Restructured to allow adding ideas from BDB that we still wanted

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

LMDB

HTTP://SYMAS.COM

Implementation Highlights

Footprint

size db_bench*						
text	data	bss	dec	hex	filename	Lines of Code
285306	1516	352	287174	461c6	db_bench	39758
384206	9304	3488	396998	60ec6	db_bench_basho	26577
1688853	2416	312	1691581	19cfbd	db_bench_bdb	1746106
315491	1596	360	317447	4d807	db_bench_hyper	21498
121412	1644	320	123376	1e1f0	db_bench_mdb	7955
1014534	2912	6688	1024134	fa086	db_bench_rocksdb	81169
992334	3720	30352	1026406	fa966	db_bench_tokudb	227698
853216	2100	1920	857236	d1494	db_bench_wiredtiger	91410

(4) Internals

- Btree Operation
  - Write-Ahead Logging
  - Append-Only
  - Copy-on-Write, LMDB-style
- Free Space Management
  - Avoiding Compaction/Garbage Collection
- Transaction Handling
  - Avoiding Locking

Btree Operation

Basic Elements

Database Page

Pgno
Misc...

Meta Page

Pgno
Misc...
Root

Data Page

Pgno
Misc...
offset

key, data

Btree Operation

Write-Ahead Logger

Meta Page

Pgno: 0
Misc...
Root : EMPTY

Write-Ahead Log

Btree Operation

Write-Ahead Logger

Meta Page

Pgno: 0
Misc...
Root : EMPTY

Write-Ahead Log

Add 1,foo to page 1

Btree Operation

Write-Ahead Logger

Meta Page

Pgno: 0
Misc...
Root : 1

Data Page

Pgno: 1
Misc...
offset: 4000

1,foo

Write-Ahead Log

Add 1,foo to page 1

Btree Operation

Write-Ahead Logger

Meta Page

Pgno: 0
Misc...
Root : 1

Data Page

Pgno: 1
Misc...
offset: 4000

1,foo

Write-Ahead Log

Add 1,foo to
page 1
Commit

Btree Operation

Write-Ahead Logger

Meta Page	Data Page	Write-Ahead Log
Pgno: 0 Misc... Root : 1	Pgno: 1 Misc... offset: 4000 1,foo	Add 1,foo to page 1 Commit Add 2,bar to page 1

Btree Operation

Write-Ahead Logger

Meta Page

Pgno: 0
Misc...
Root : 1

Data Page

Pgno: 1
Misc...
offset: 4000
offset: 3000
2,bar
1,foo

Write-Ahead Log

Add 1,foo to page 1
Commit
Add 2,bar to page 1

Btree Operation

Write-Ahead Logger

Meta Page

Pgno: 0
Misc...
Root : 1

Data Page

Pgno: 1
Misc...
offset: 4000
offset: 3000
2,bar
1,foo

Write-Ahead Log

Add 1,foo to page 1
Commit
Add 2,bar to page 1
Commit

Btree Operation

Write-Ahead Logger

Meta Page

Pgno: 0
Misc...
Root : 1

Data Page

Pgno: 1
Misc...
offset: 4000
offset: 3000
2,bar
1,foo

Write-Ahead Log

Add 1,foo to page 1
Commit
Add 2,bar to page 1
Commit
Checkpoint

Meta Page

Pgno: 0
Misc...
Root : 1

Data Page

Pgno: 1
Misc...
offset: 4000
offset: 3000
2,bar
1,foo

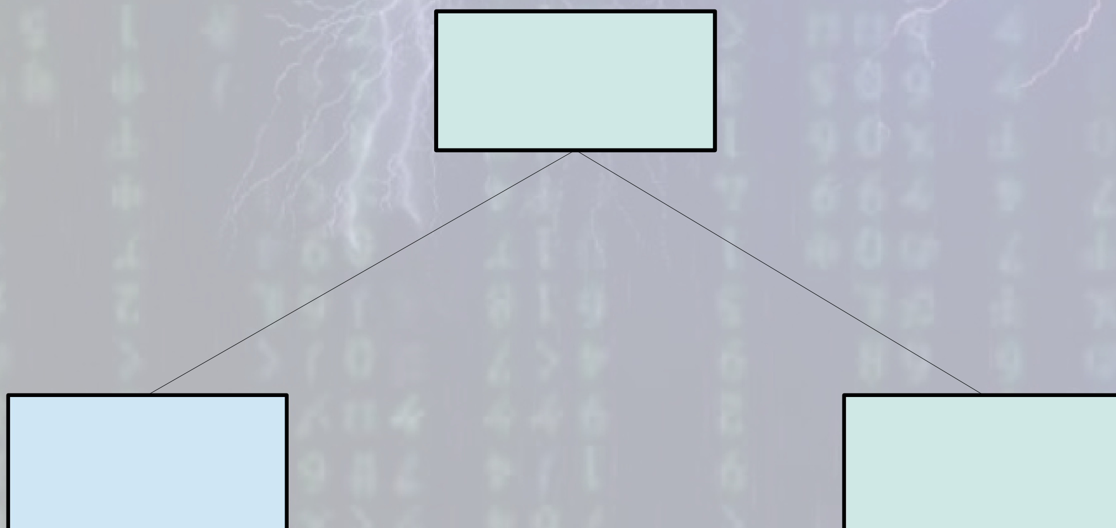
Btree Operation

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 unaltered
- The root node is always written last

Btree Operation

Append-Only



Start with a simple tree

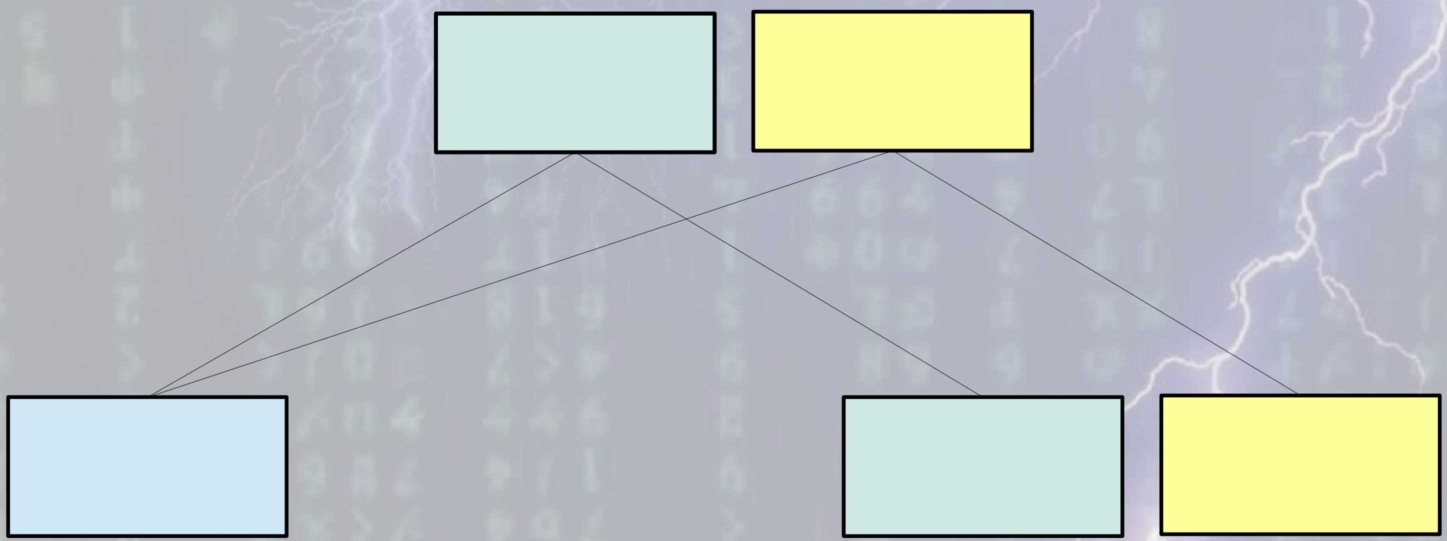
Btree Operation

Append-Only

Update a leaf node by copying it and updating the copy

Btree Operation

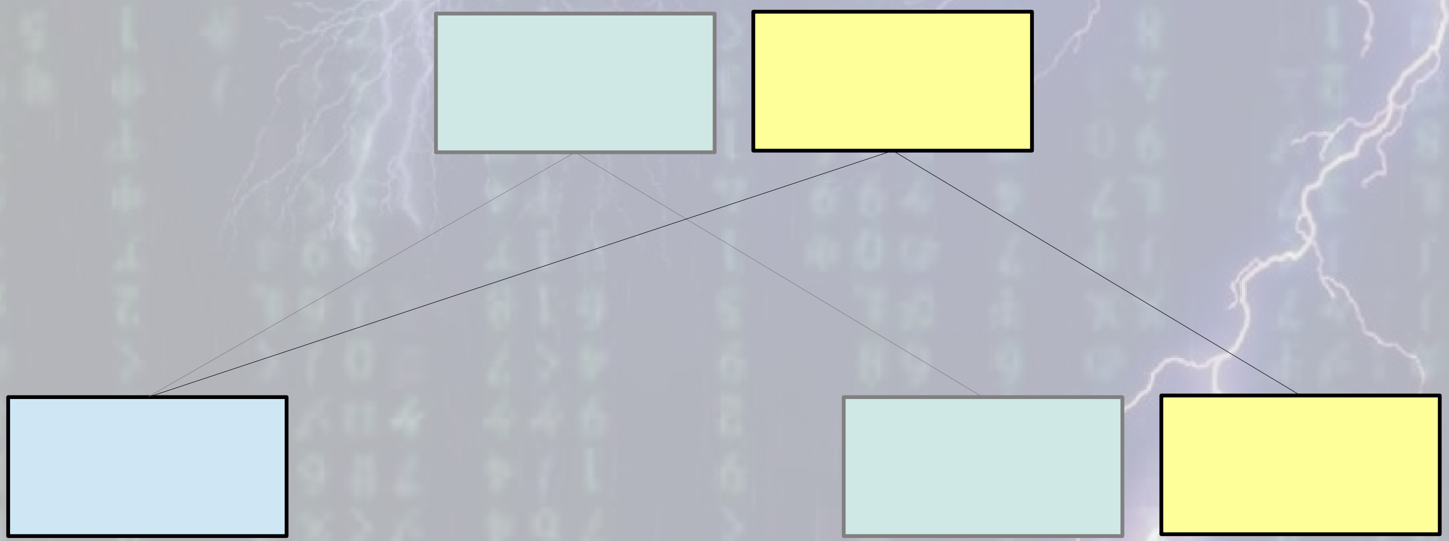
Append-Only



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

Btree Operation

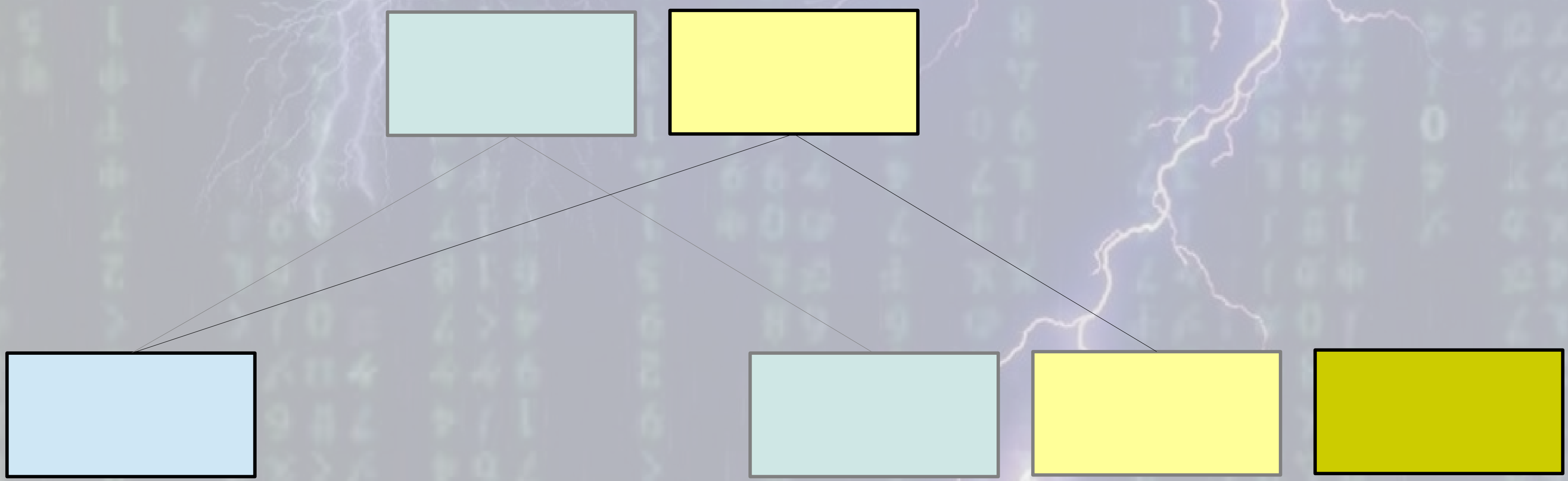
Append-Only



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

Btree Operation

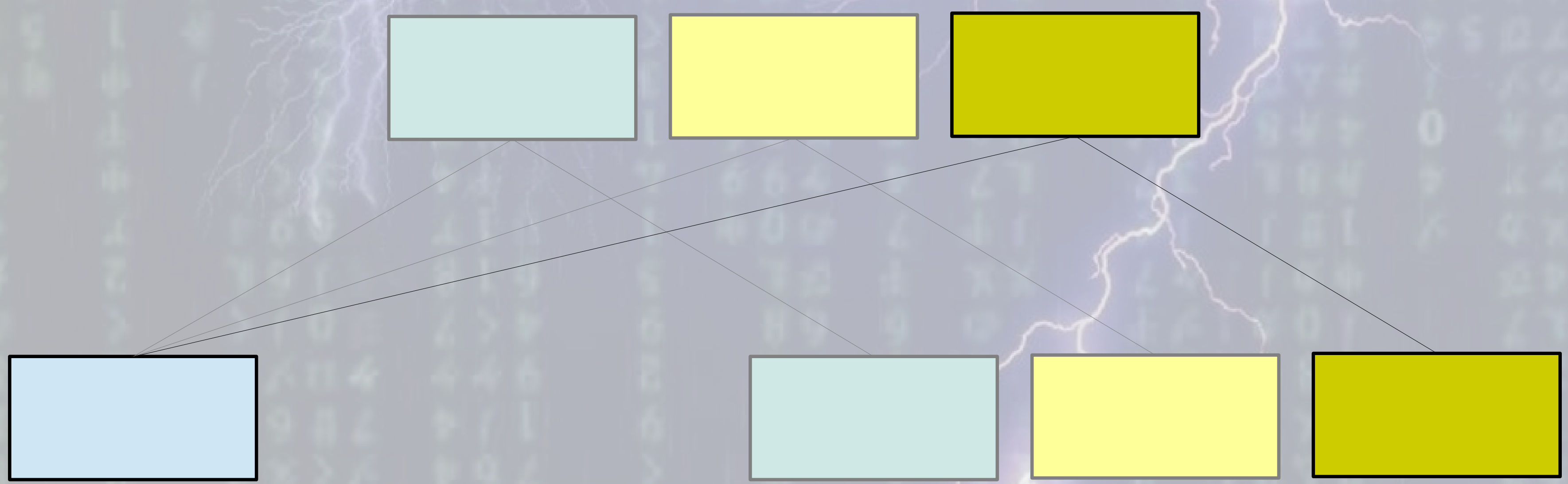
Append-Only



Further updates create additional versions

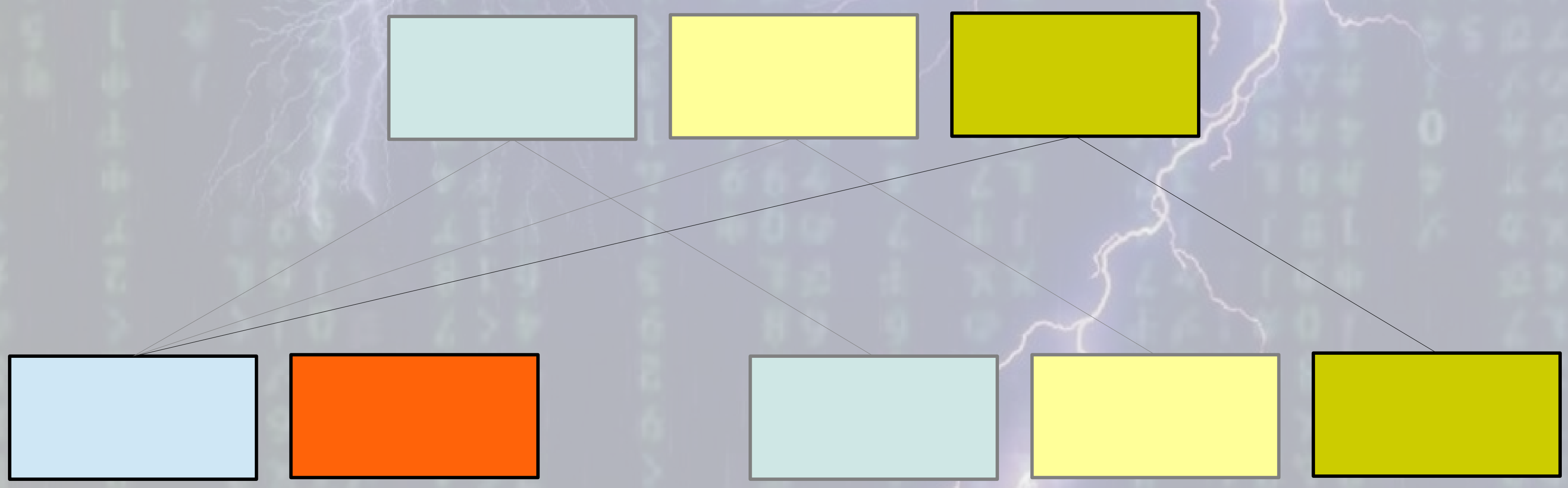
Btree Operation

Append-Only



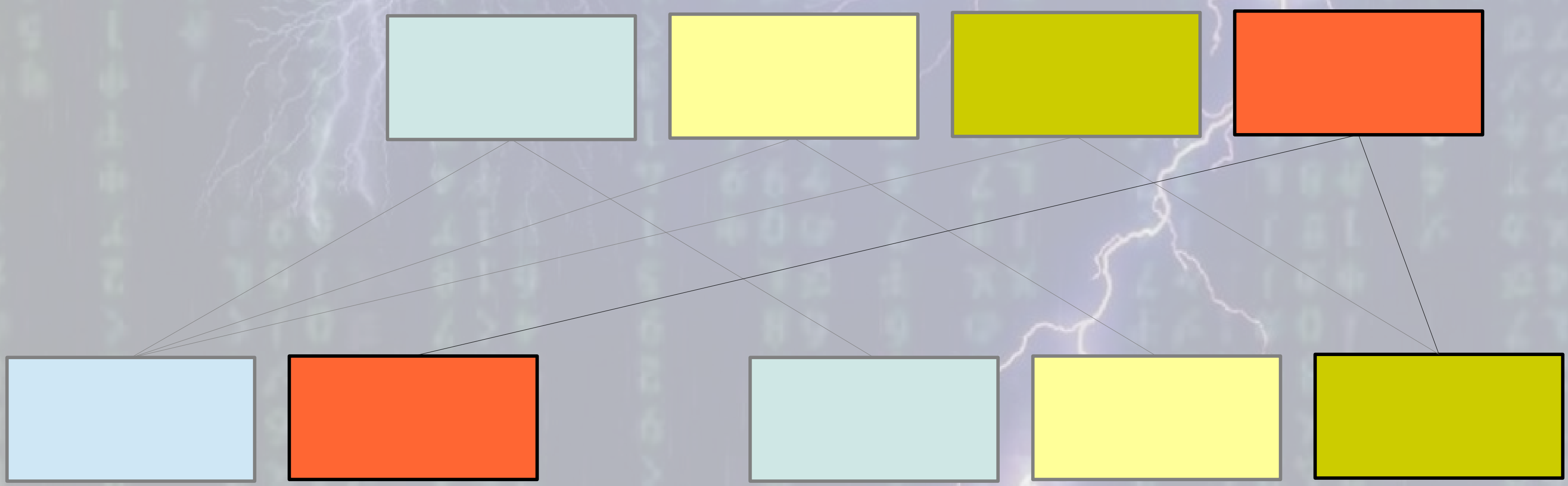
Btree Operation

Append-Only



Btree Operation

Append-Only



Btree Operation

In the Append-Only tree, new pages are always appended sequentially to the DB 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 crash
- Any root node can be found by seeking backward from the end of the file, and checking the page's header
- Recovery from a 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

Btree Operation

Append-Only

Meta Page

Pgno: 0
Misc...
Root : EMPTY

HTTP://SYMAS.COM

Btree Operation

Append-Only

Meta Page	Data Page
Pgno: 0 Misc... Root : EMPTY	Pgno: 1 Misc... offset: 4000 1,foo

Btree Operation

Append-Only

Meta Page	Data Page	Meta Page
Pgno: 0 Misc... Root : EMPTY	Pgno: 1 Misc... offset: 4000 1,foo	Pgno: 2 Misc... Root : 1

LMDB

HTTP://SYMAS.COM

Btree Operation

Append-Only

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

Btree Operation

Append-Only

Meta Page	Data Page	Meta Page	Data Page	Meta Page
Pgno: 0 Misc... Root : EMPTY	Pgno: 1 Misc... offset: 4000 1,foo	Pgno: 2 Misc... Root : 1	Pgno: 3 Misc... offset: 4000 offset: 3000 2,bar 1,foo	Pgno: 4 Misc... Root : 3

LMDB

HTTP://SYMAS.COM

Btree Operation

Append-Only

Meta Page	Data Page	Meta Page	Data Page	Meta Page
Pgno: 0 Misc... Root : EMPTY	Pgno: 1 Misc... offset: 4000 1,foo	Pgno: 2 Misc... Root : 1	Pgno: 3 Misc... offset: 4000 offset: 3000 2,bar 1,foo	Pgno: 4 Misc... Root : 3

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

Btree Operation

Append-Only

Meta Page	Data Page	Meta Page	Data Page	Meta Page
Pgno: 0 Misc... Root : EMPTY	Pgno: 1 Misc... offset: 4000 1,foo	Pgno: 2 Misc... Root : 1	Pgno: 3 Misc... offset: 4000 offset: 3000 2,bar 1,foo	Pgno: 4 Misc... Root : 3

Data Page	Meta Page
Pgno: 5 Misc... offset: 4000 offset: 3000 2,bar 1,blah	Pgno: 6 Misc... Root : 5

Btree Operation

Append-Only

Meta Page	Data Page	Meta Page	Data Page	Meta Page
Pgno: 0 Misc... Root : EMPTY	Pgno: 1 Misc... offset: 4000 1,foo	Pgno: 2 Misc... Root : 1	Pgno: 3 Misc... offset: 4000 offset: 3000 2,bar 1,foo	Pgno: 4 Misc... Root : 3

Data Page	Meta Page	Data Page
Pgno: 5 Misc... offset: 4000 offset: 3000 2,bar 1,blah	Pgno: 6 Misc... Root : 5	Pgno: 7 Misc... offset: 4000 offset: 3000 2,xyz 1,blah

Btree Operation

Append-Only

Meta Page	Data Page	Meta Page	Data Page	Meta Page
Pgno: 0 Misc... Root : EMPTY	Pgno: 1 Misc... offset: 4000 1,foo	Pgno: 2 Misc... Root : 1	Pgno: 3 Misc... offset: 4000 offset: 3000 2,bar 1,foo	Pgno: 4 Misc... Root : 3

Data Page	Meta Page	Data Page	Meta Page
Pgno: 5 Misc... offset: 4000 offset: 3000 2,bar 1,blah	Pgno: 6 Misc... Root : 5	Pgno: 7 Misc... offset: 4000 offset: 3000 2,xyz 1,blah	Pgno: 8 Misc... Root : 7

Btree Operation

Append-Only disk usage is very inefficient

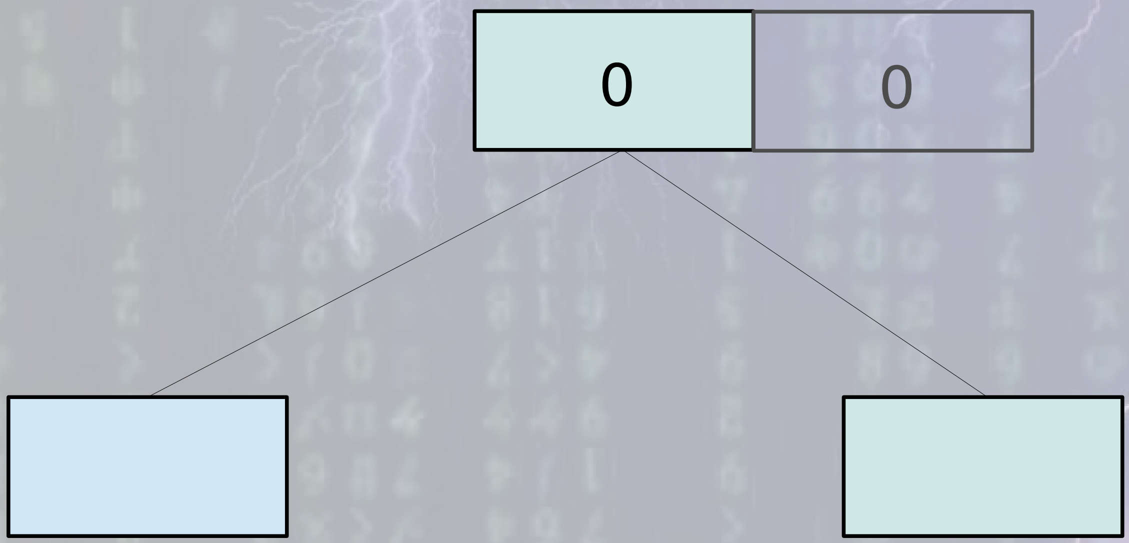
- Disk space usage grows without bound
- 99+% of the space will be occupied by old versions of the data
- The old versions are usually not interesting
- Reclaiming the old space requires a very expensive compaction phase
- New updates must be throttled until compaction completes

Btree Operation

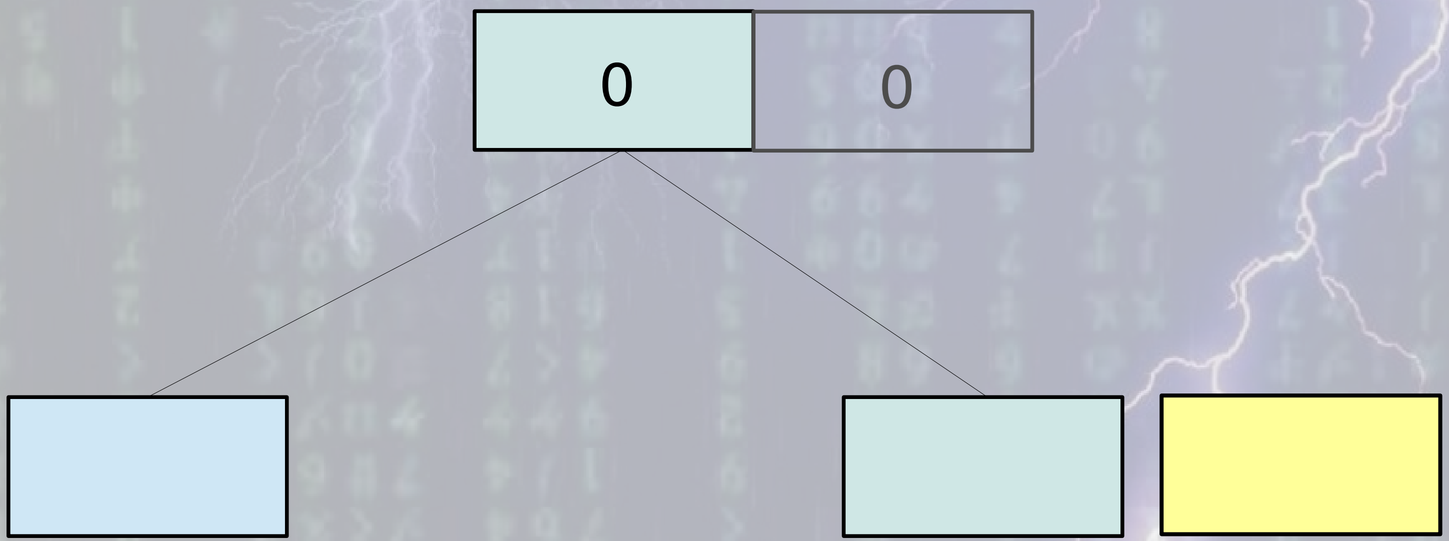
The LMDB Approach

- 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"

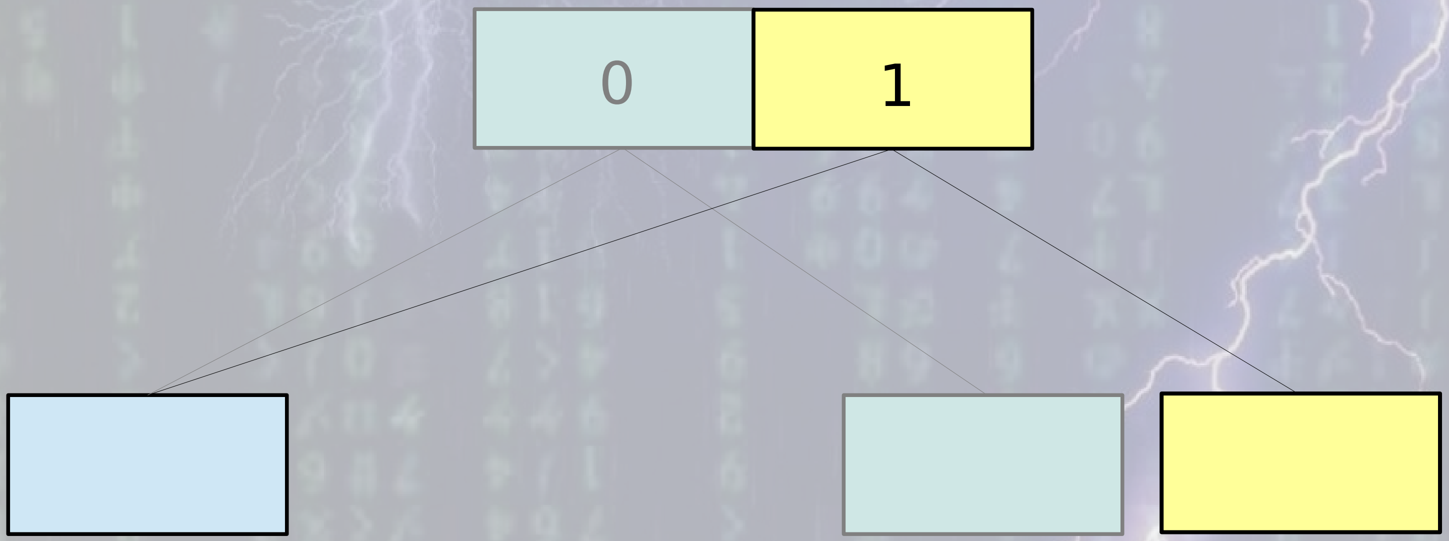
Btree Operation



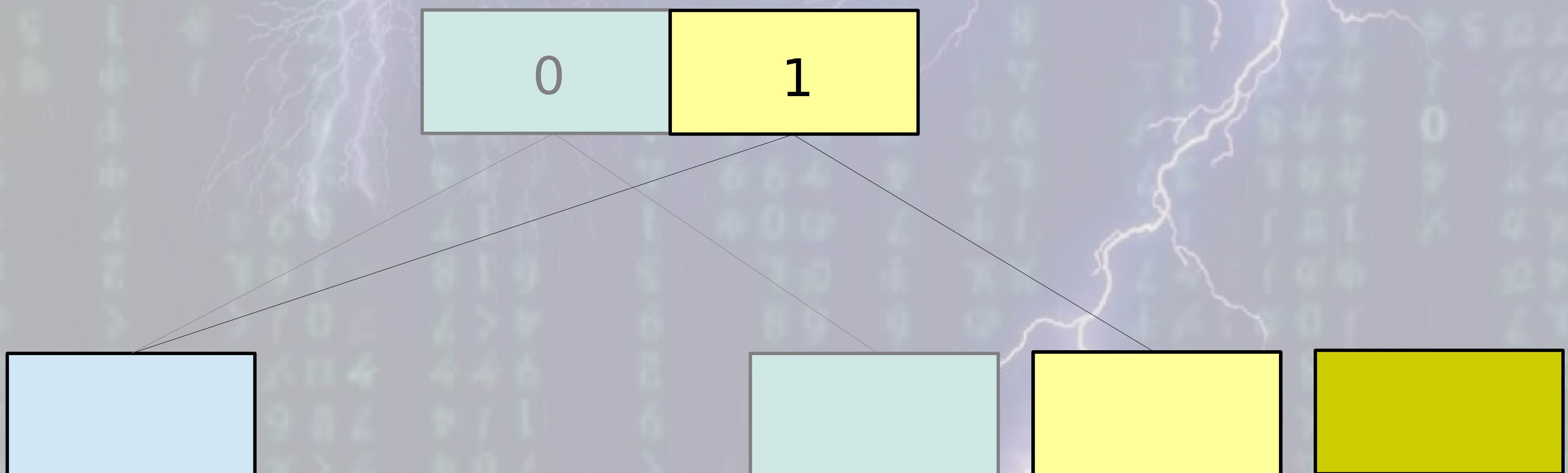
Btree Operation



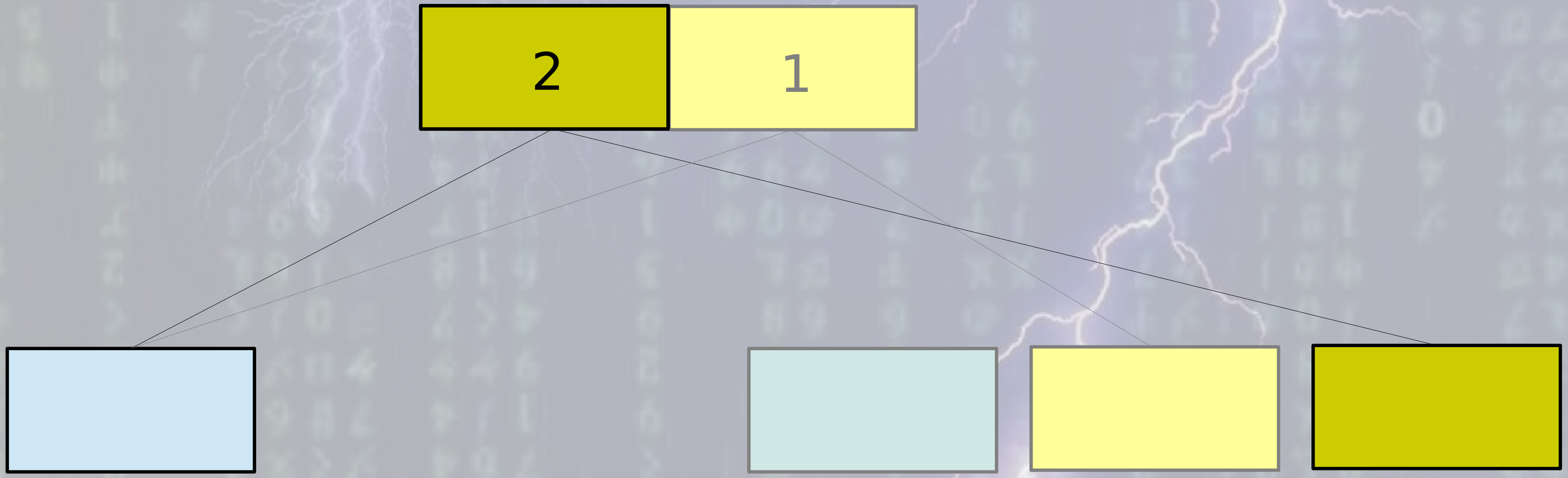
Btree Operation



Btree Operation

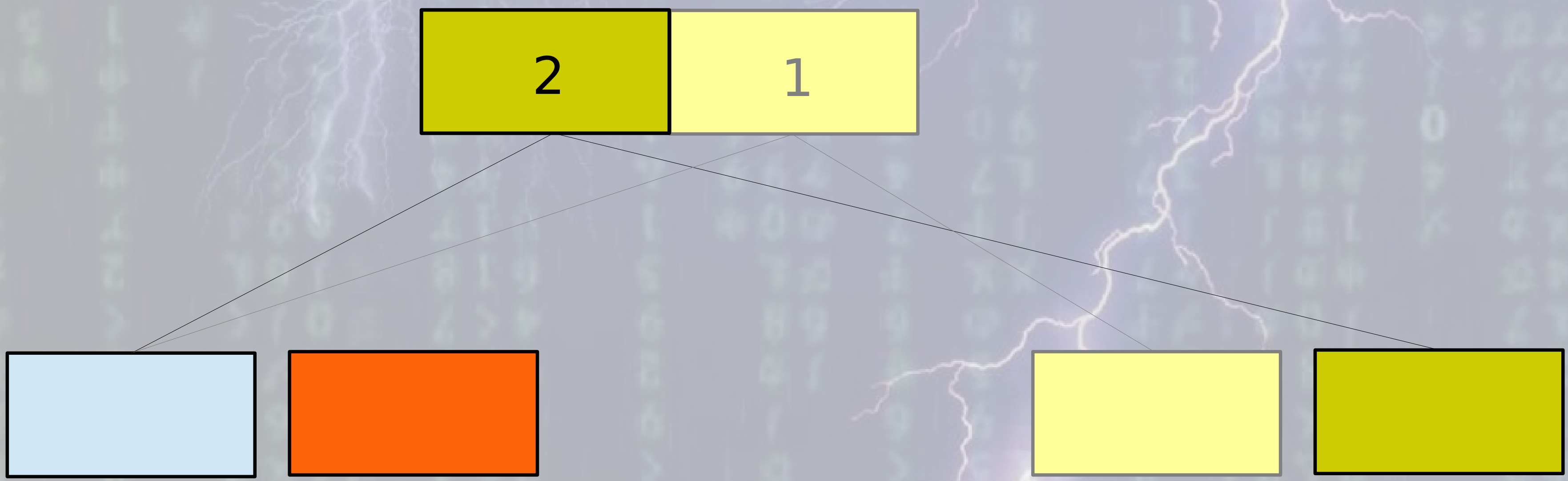


Btree Operation

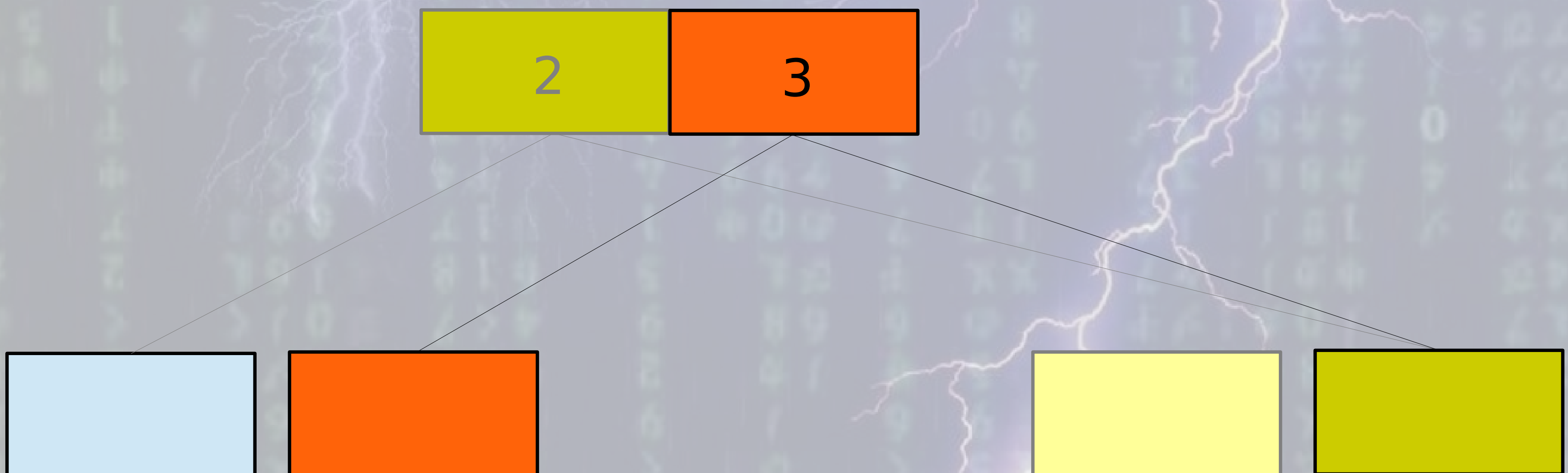


After this step the old blue page is no longer referenced by anything else in the database, so it can be reclaimed

Btree Operation



Btree 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

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

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

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

HTTP://SYMAS.COM

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	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

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 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

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

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 array elements are CPU cache aligned
  - 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

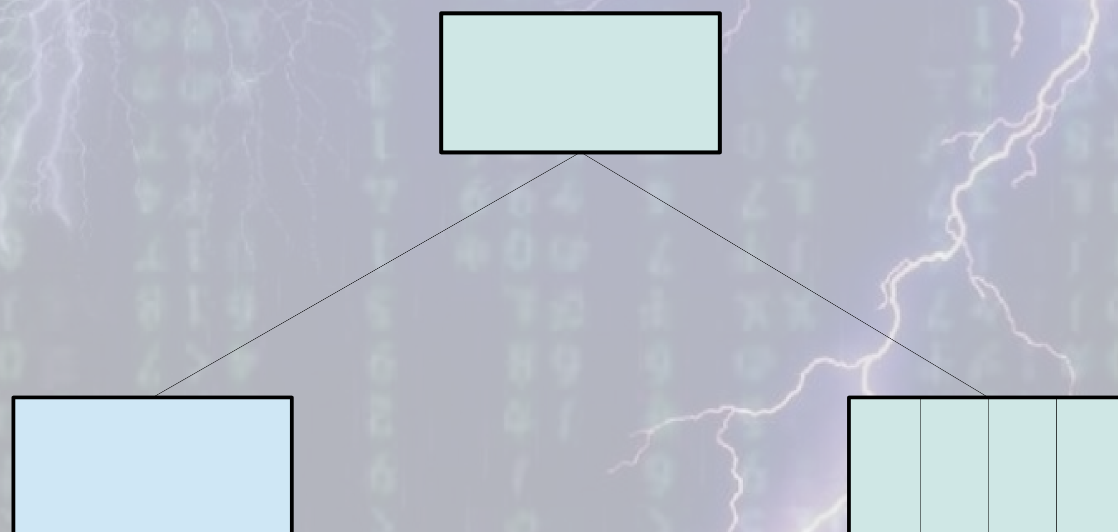
- Write transactions use pages from the free list before allocating new disk pages
  - Pages in the free list are used in order, oldest transaction first
  - The readers table must be scanned to see if any reader is referencing an old transaction
  - The writer doesn't need to lock the reader table when 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

(Dark Underside)

- Complications that textbook B+tree descriptions never mention
  - Textbook examples assume uniform record sizes
    - In real life, records are variable size
  - Textbook examples use parent and sibling pointers in each page
    - Supports fast iteration, range lookups
    - With COW, only child pointers are allowed

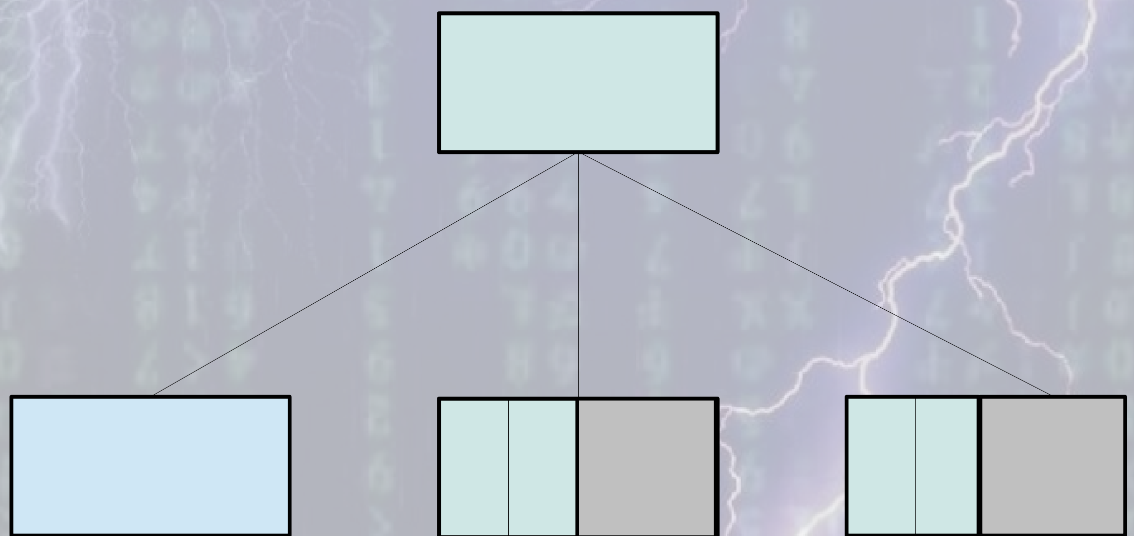
Page Splitting

Uniform Records



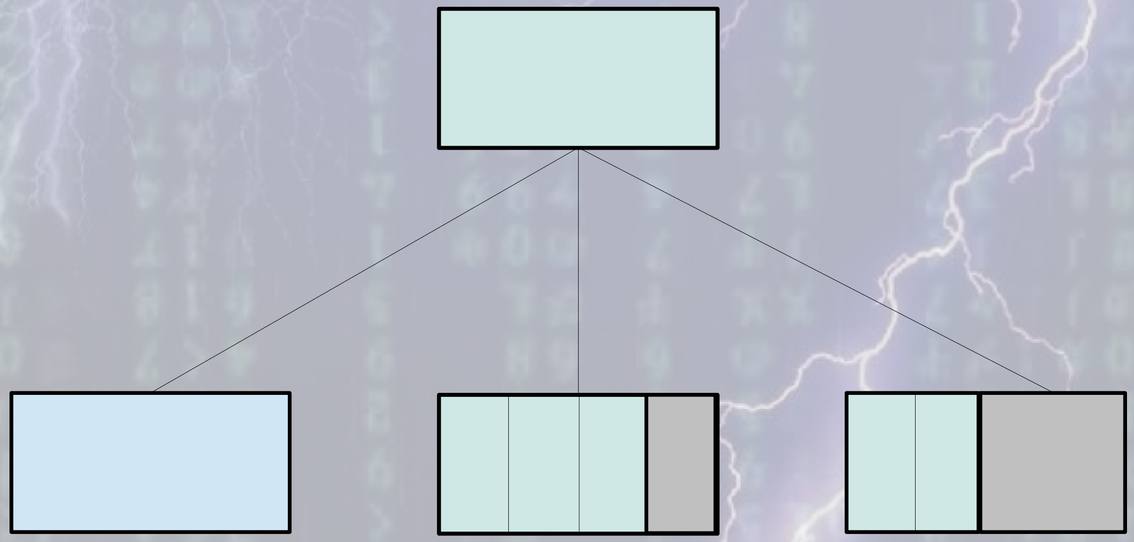
Page Splitting

Uniform Records



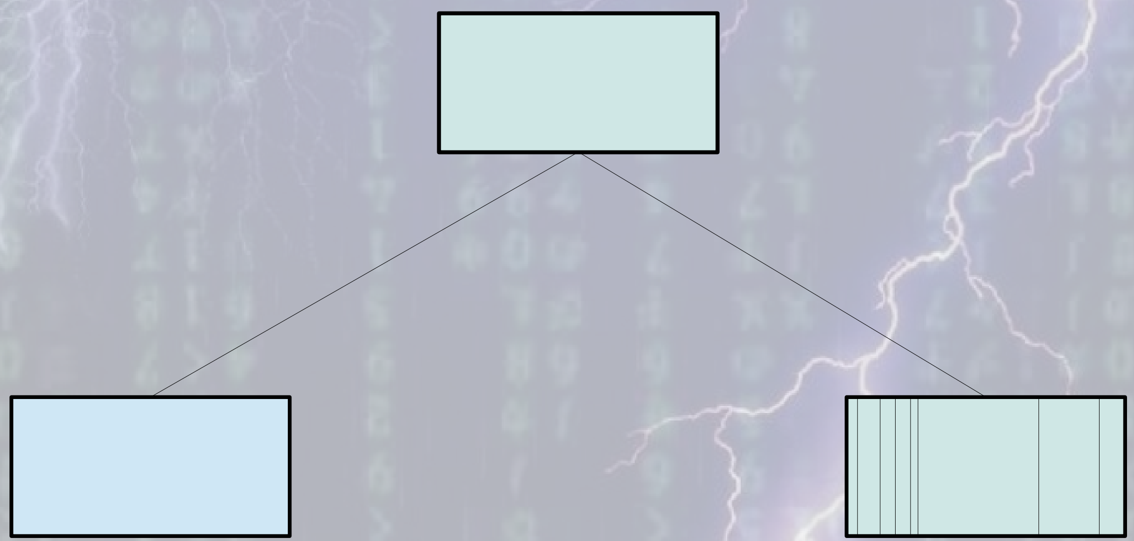
Page Splitting

Uniform Records



Page Splitting

Non-Uniform Records



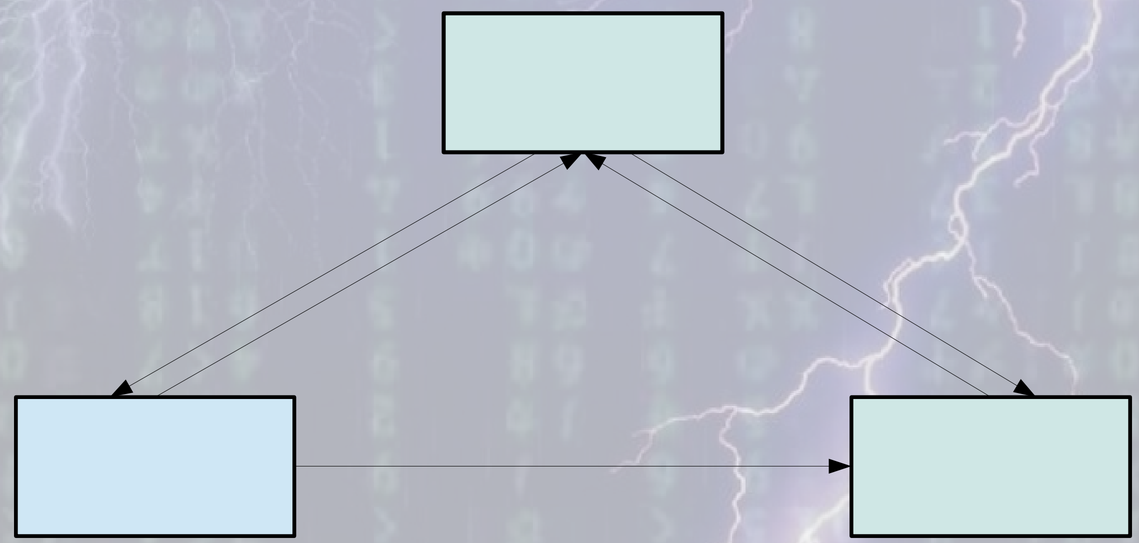
8 nodes, but where do we split?

Page Splitting

- Where do we split? "It depends"
  - For pages with "small" number of records, we tally up the sizes of each node up to the insertion point and split wherever things fit
  - For pages with more records, we assume they are all small enough that just taking the N/2 slot will work

Page Pointers

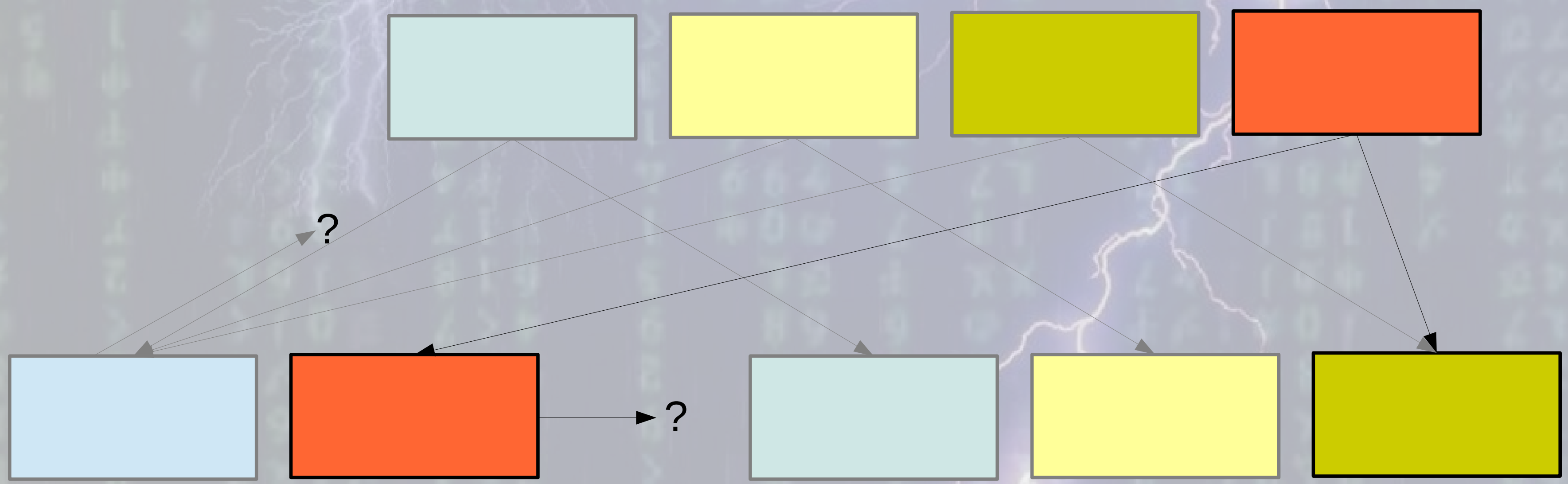
Regular B+tree



Pages point to their parent as well as sibling pages

Page Pointers

Copy-on-Write



Pages no longer have unique parent or sibling

Page Pointers

- Naive code mallocs pointer structs to point to pages and their parents
  - Messy, requires extensive malloc/free activity while navigating up/down tree
  - Refcounting overhead, NULL checks, etc...
- LMDB uses a cursor struct with a fixed-size stack to record position in tree
  - No malloc/free needed for navigation

Page Pointers

Aside - QuickSort

```
void qSort1(int a[], int p, int q) {

const int max = 50; int lb[max], rb[max], n = 0;

L: // to be sorted: a[p..q-1] and the n items on the stack

  if (p < q-1) { // at least two elements in a[p..q-1]

    int m = partition(a, p, q);

    if (m-p < q-(m+1)) { //push smaller (left) half

      lb[n] = p; rb[n] = m; n++;

      // sort right half

      p = m+1; goto L;

    } else { // push smaller (right) half

      lb[n] = m+1; rb[n] = q; n++;

      // sort left half

      q = m; goto L;

    }
  } // p >= q-1, so a[p..q-1] already sorted
  if (n == 0) return; // because nothing on the
stack
  n--; p = lb[n]; q = rb[n];
  goto L;
}
void qSort(int a[], int n) { // sorts a[0..n-1]
  qSort1(a, 0, n);
}
```

- Why is max = 50?

(Dark Underside)

- Cursor Tracking
  - When multiple cursors are open in the same transaction on the same DB, changes made by one may affect all the others
    - Obvious example - Deleting a record that others point to
- Freespace DB
  - Special challenges unique to LMDB

Cursor Tracking

- Deleting a record that others point to
  - BerkeleyDB sets a tombstone
    - Other cursors need no special help, but you must schedule periodic compactions to reclaim space
  - SQLite3 invalidates all open cursors
    - Other cursors must seek to their old position again on next use
  - LMDB updates all open cursors
    - Other cursors are always immediately usable

Free Space DB

- Using an additional B+tree to track the free pages in the B+tree presents special challenges
  - The act of recording a free page in the tree may consume a free page
  - Updating the freespace DB requires careful thought

(5) 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, 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 and used directly, with no deserialization

Special Features

- Sub-Databases
  - Store multiple independent named B+trees in a single LMDB environment
  - A Sub-DB 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
    - Transactions may span all of the Sub-DBs
    - Used in back-mdb for the main data and all of the indices
    - Used in SQLightning for multiple tables and indices

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
  - Can be used to efficiently implement inverted indices and sets

Special Features

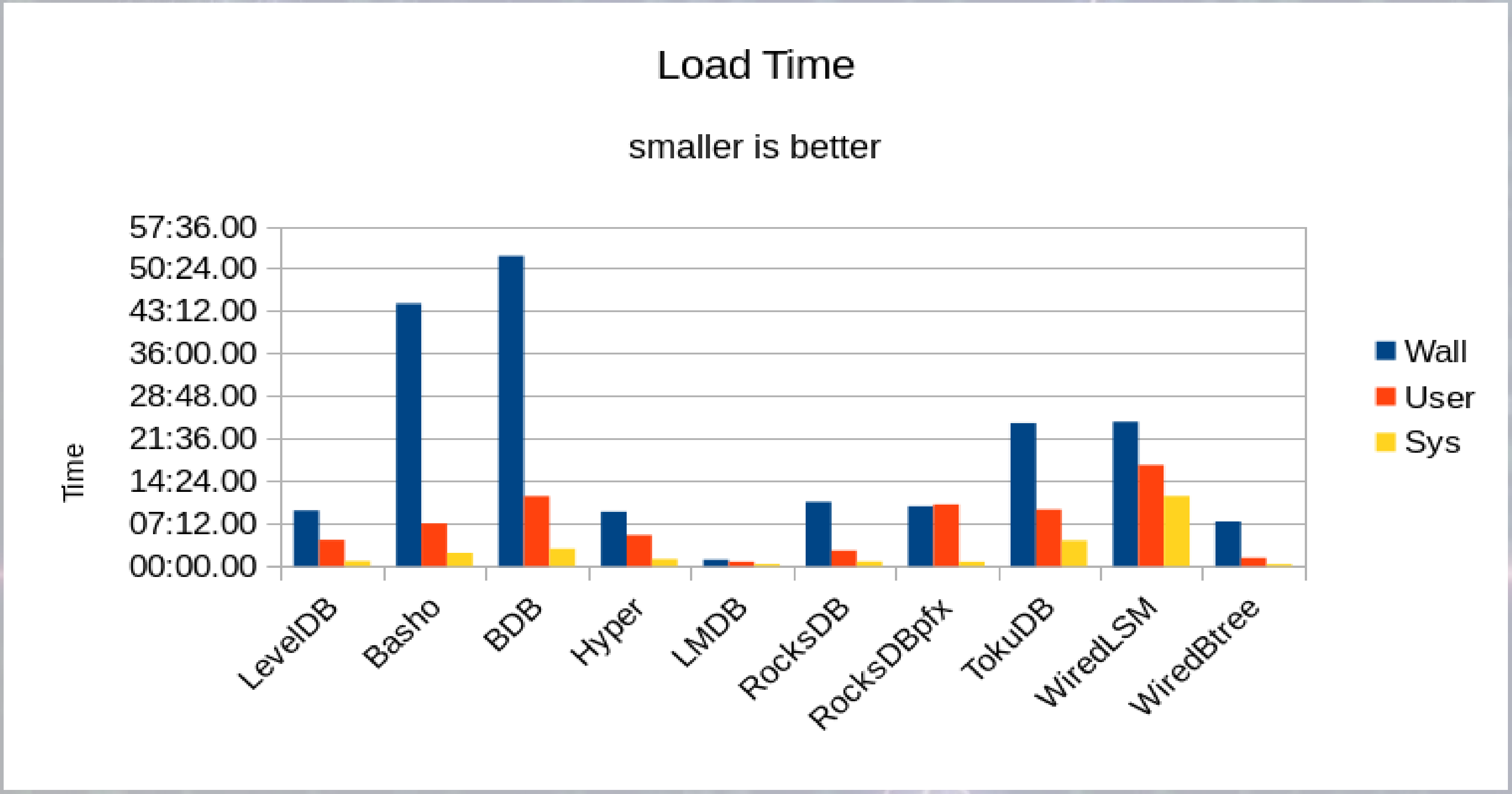
- Atomic Hot Backup
  - The entire database can be backed up live
  - No need to stop updates while backups run
  - The backup runs at the maximum speed of the target storage medium
  - Essentially: write(outfd, map, mapsize);
    - No memcpy's in or out of user space
    - Pure DMA from the database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

(6) Results

- Support for LMDB is available in scores of open source projects and all major Linux and BSD distros
- In OpenLDAP slapd
  - LMDB reads are 5-20x faster than BDB
  - Writes are 2-5x faster than BDB
  - Consumes 1/4 as much RAM as BDB
- In MemcacheDB
  - LMDB reads are 2-200x faster than BDB
  - Writes are 5-900x faster than BDB
  - Multi-thread reads are 2-8x faster than pure-memory Memcached

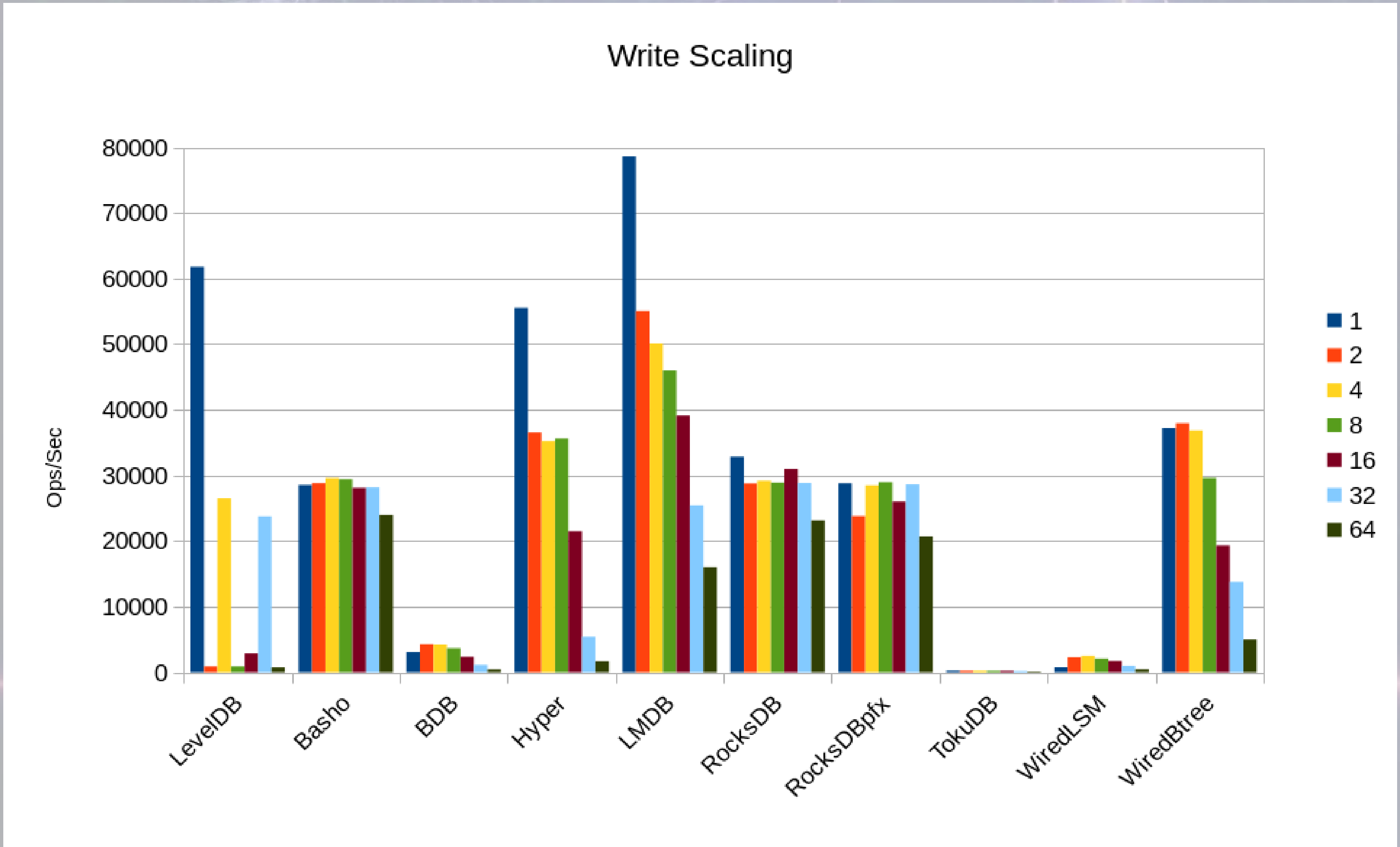
Results

- Microbenchmarks
  - In-memory DB with 100M records, 16 byte keys, 100 byte values



Results

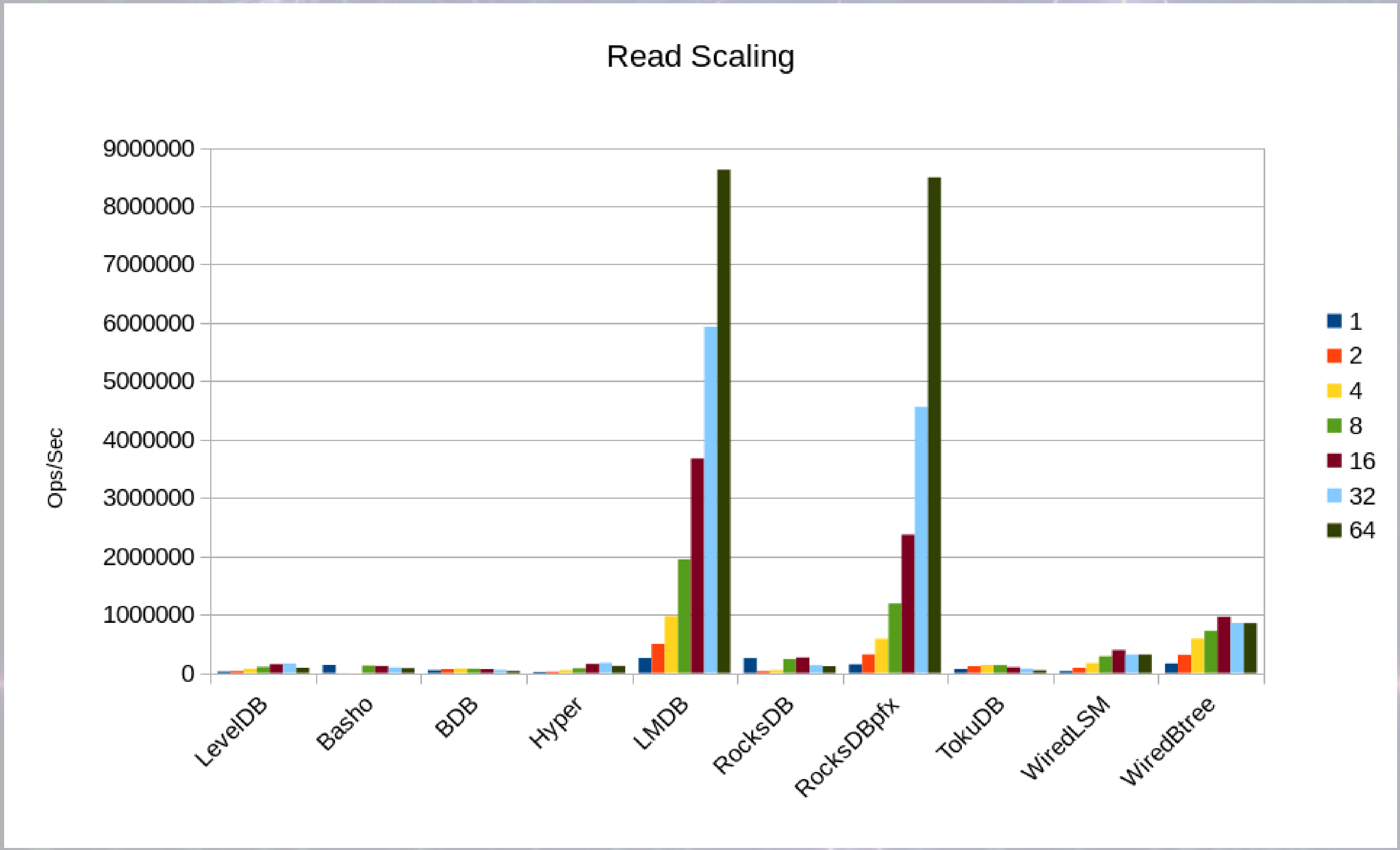
- Scaling up to 64 CPUs, 64 concurrent readers



HTTP://SYMAS.COM

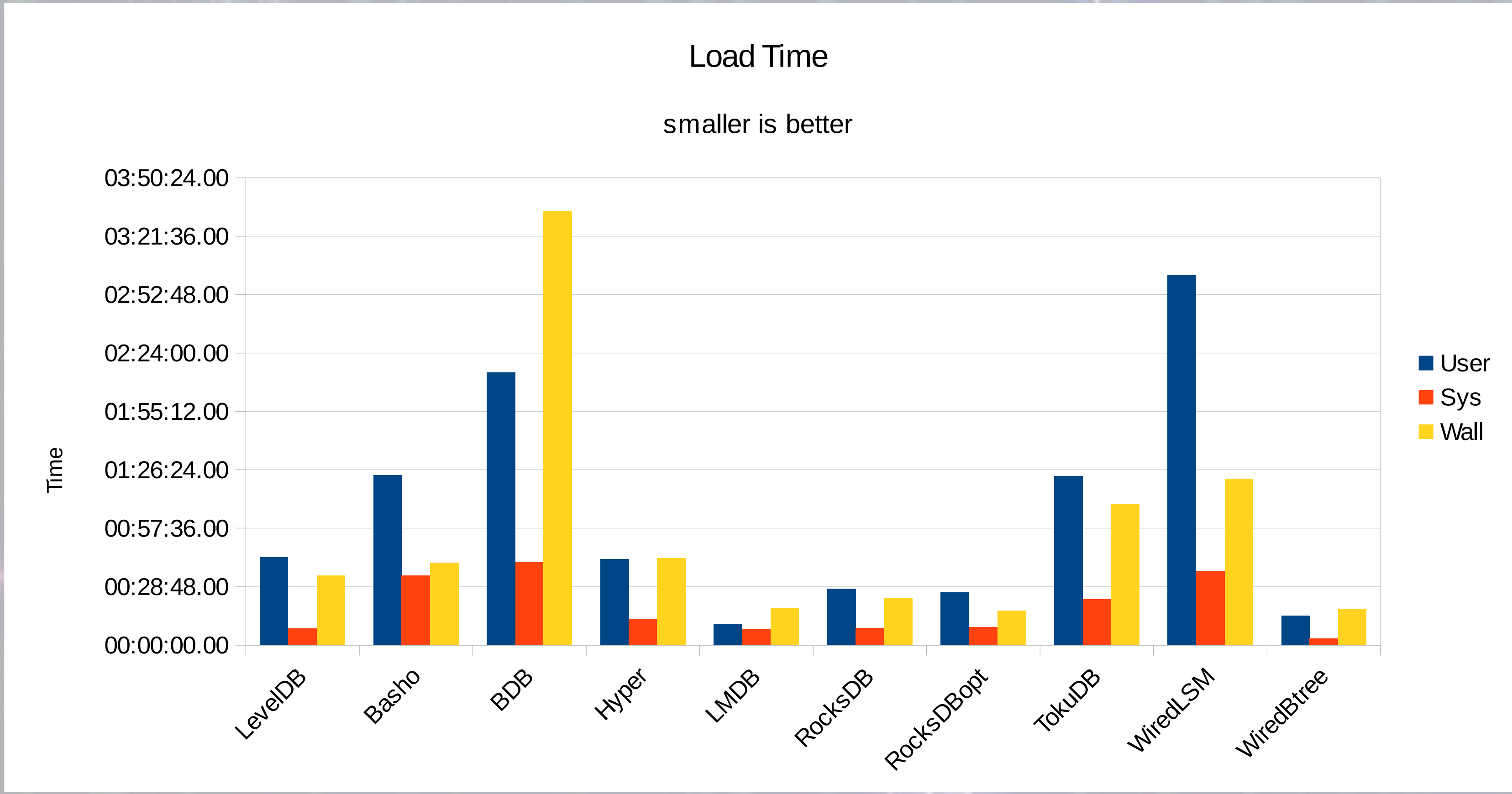
Results

- Scaling up to 64 CPUs, 64 concurrent readers



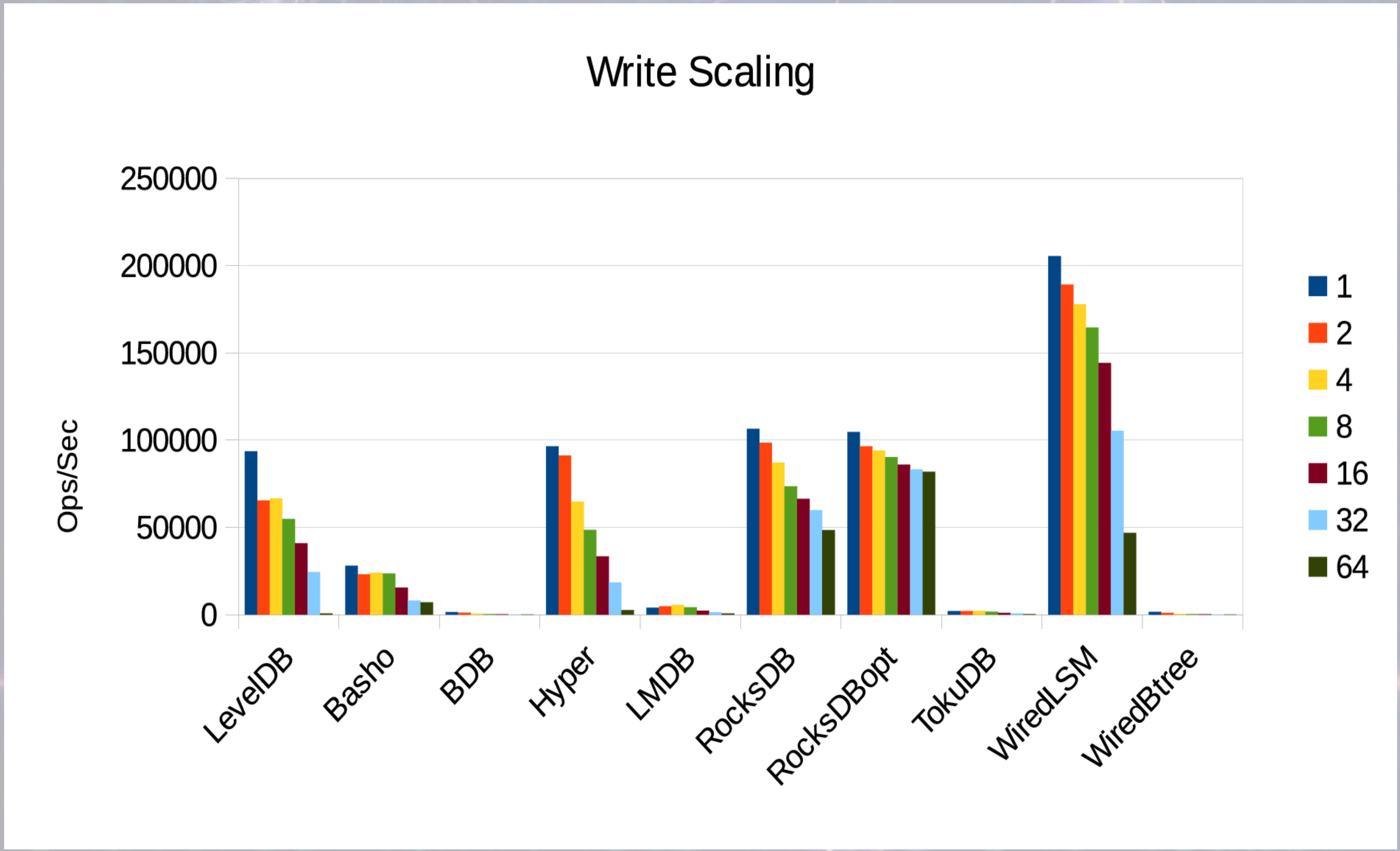
Results

- Microbenchmarks
  - On-disk, 1.6Billion records, 16 byte keys, 96 byte values, 160GB on disk with 32GB RAM, VM



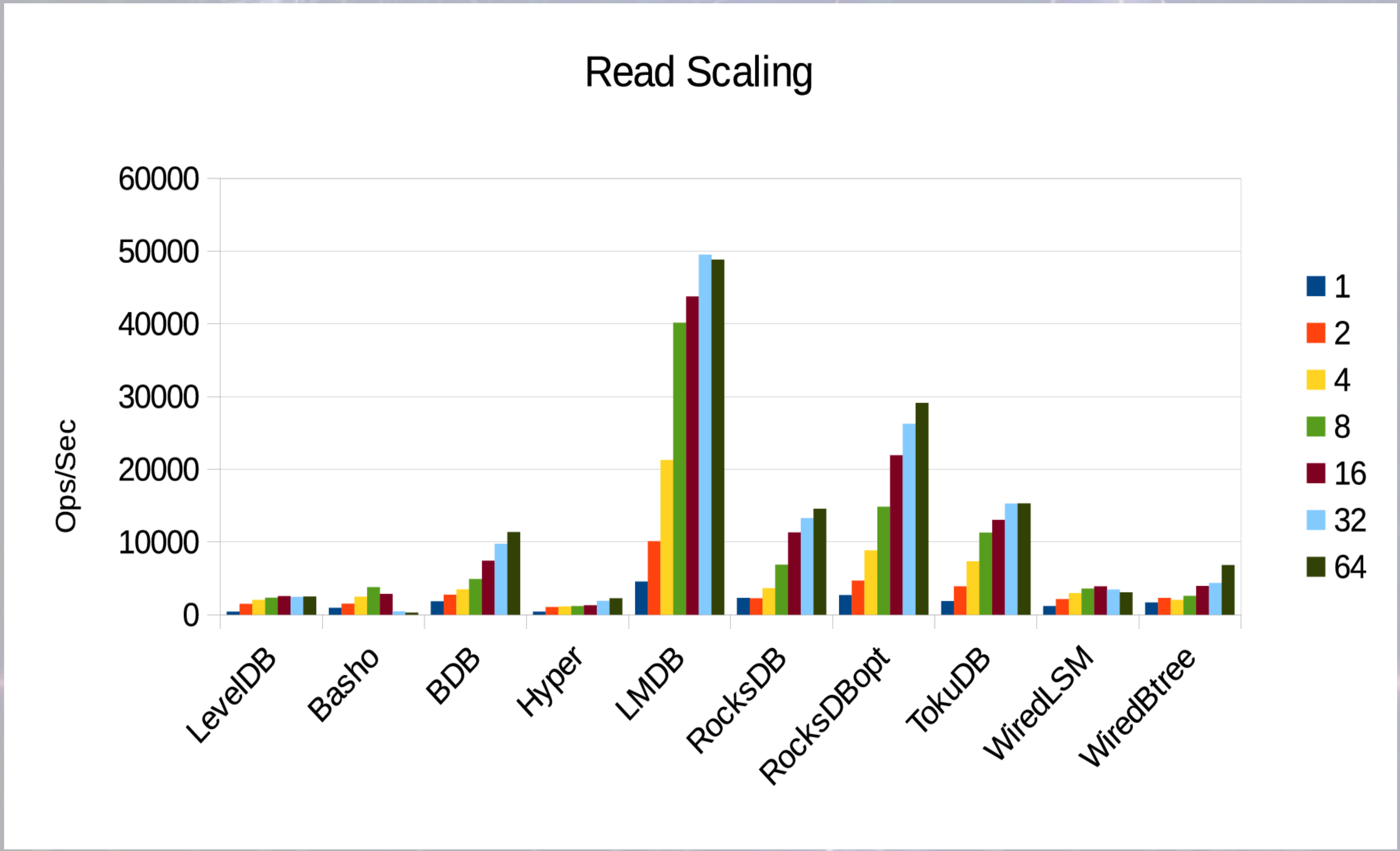
Results

- VM with 16 CPU cores, 64 concurrent readers



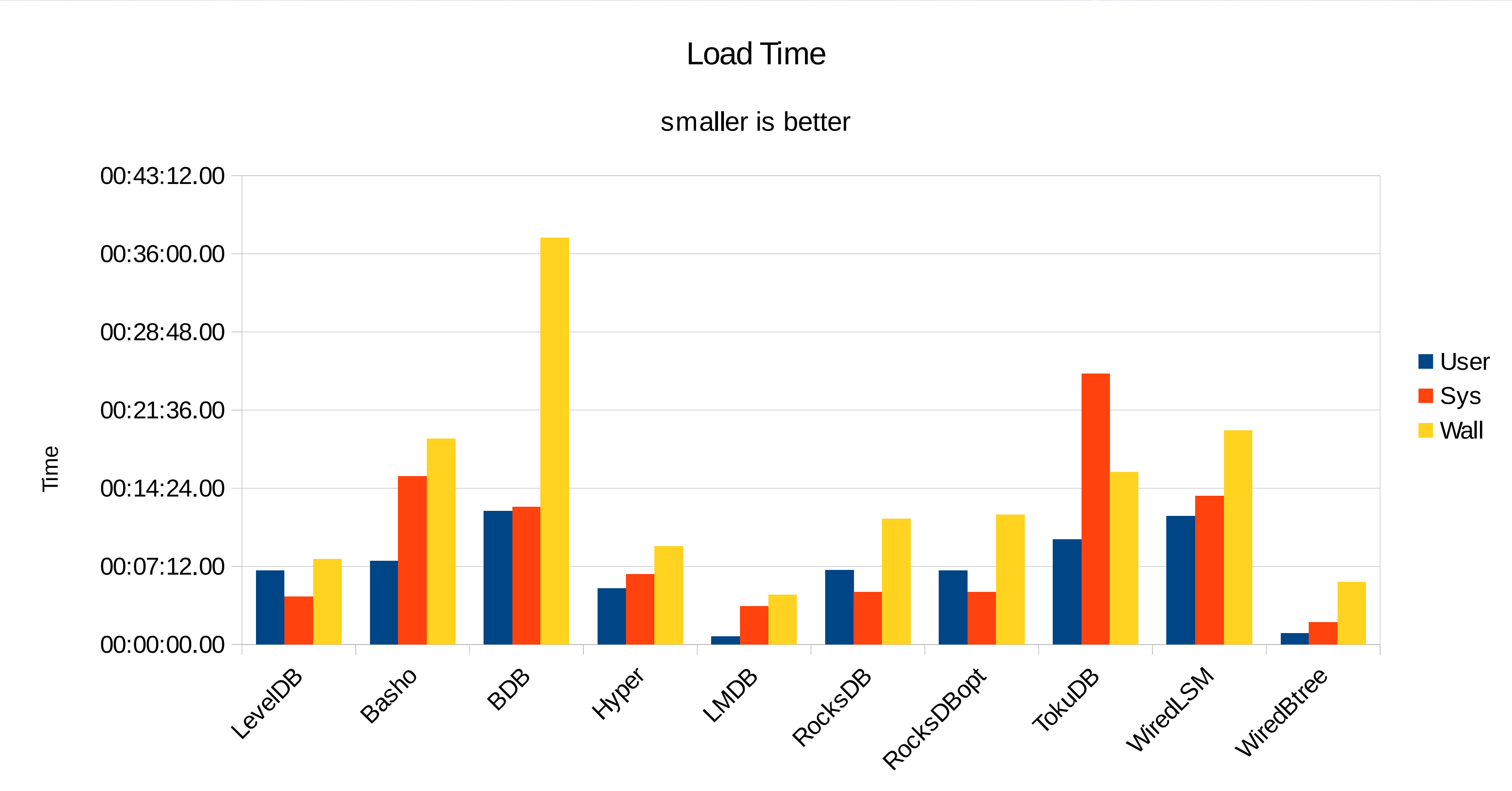
Results

- VM with 16 CPU cores, 64 concurrent readers



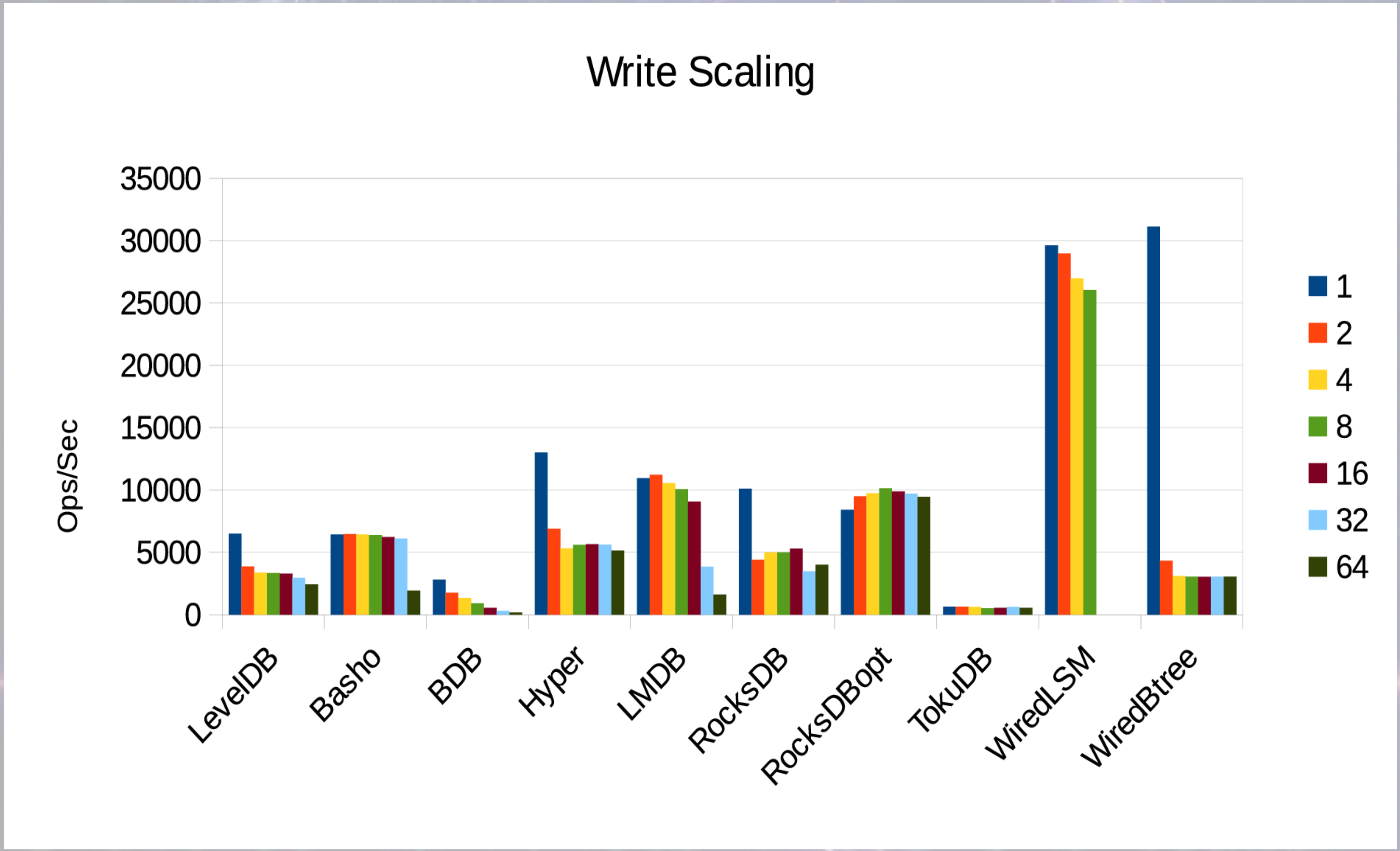
Results

- Microbenchmark
  - On-disk, 384M records, 16 byte keys, 4000 byte values, 160GB on disk with 32GB RAM



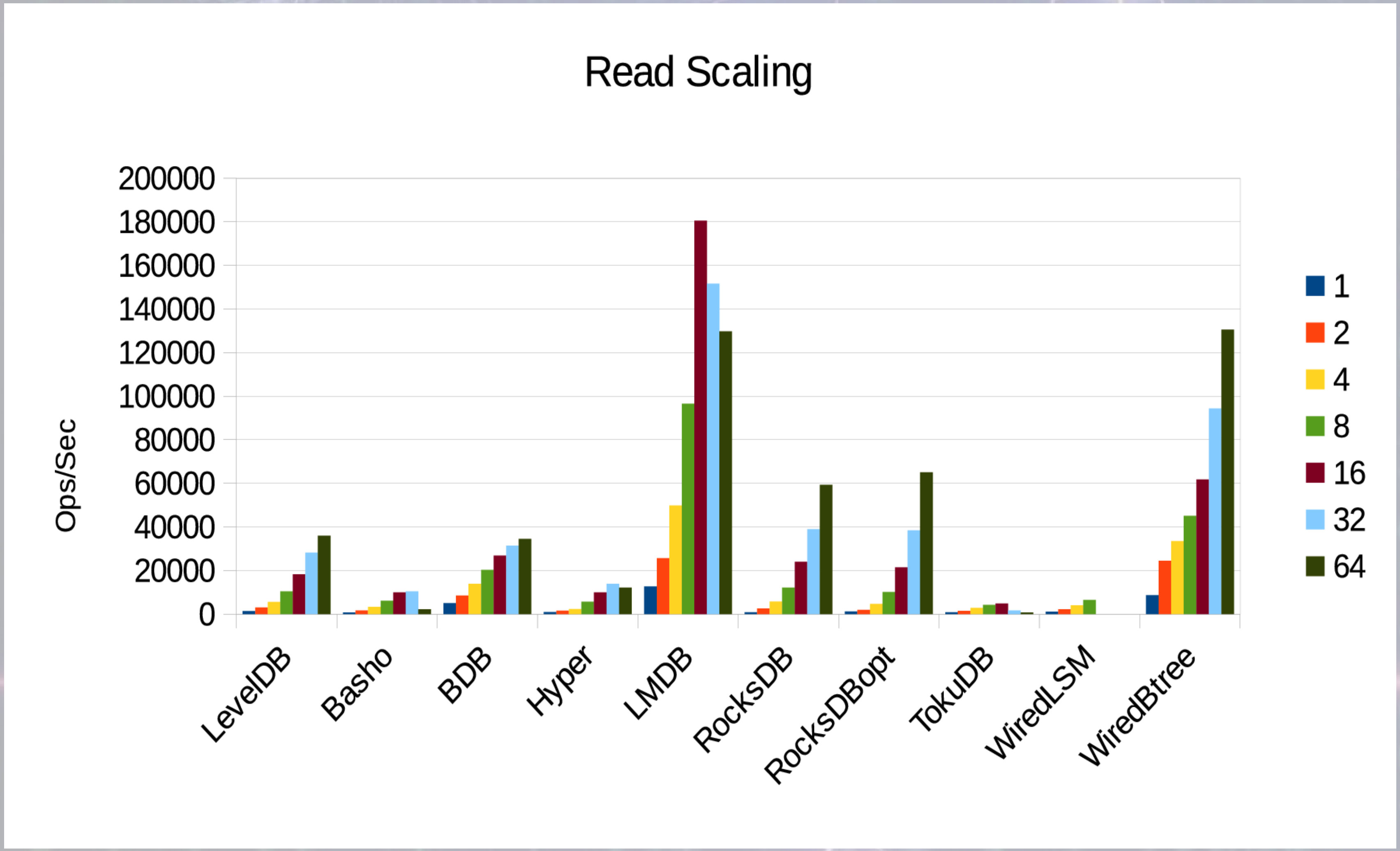
Results

- 16 CPU cores, 64 concurrent readers



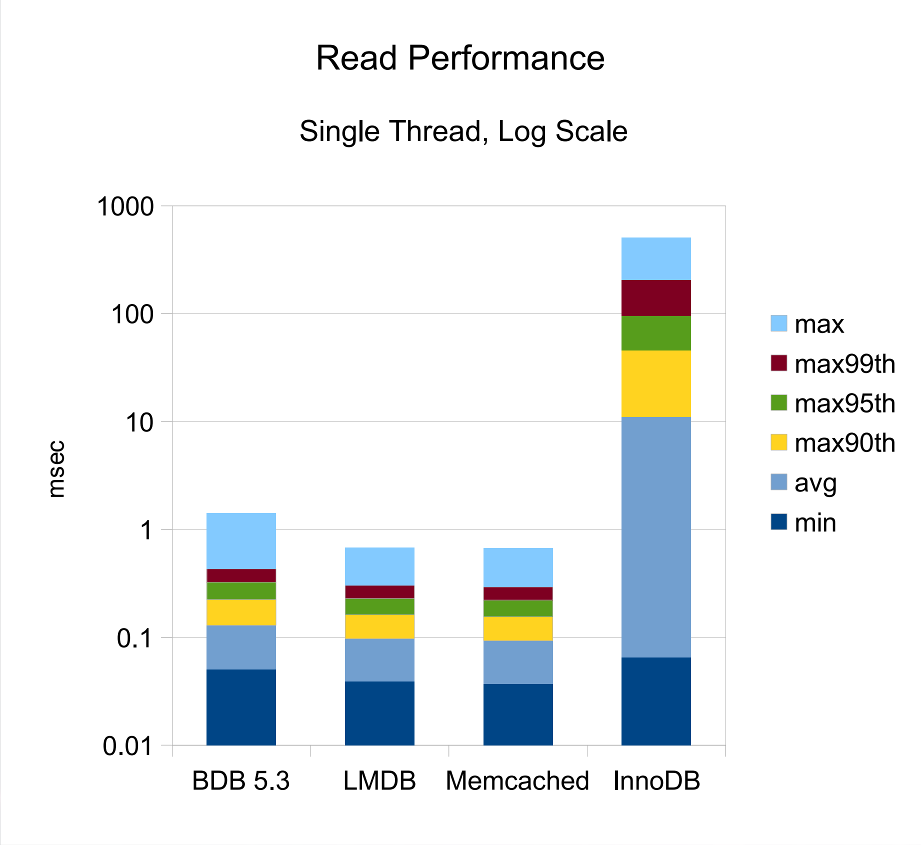
Results

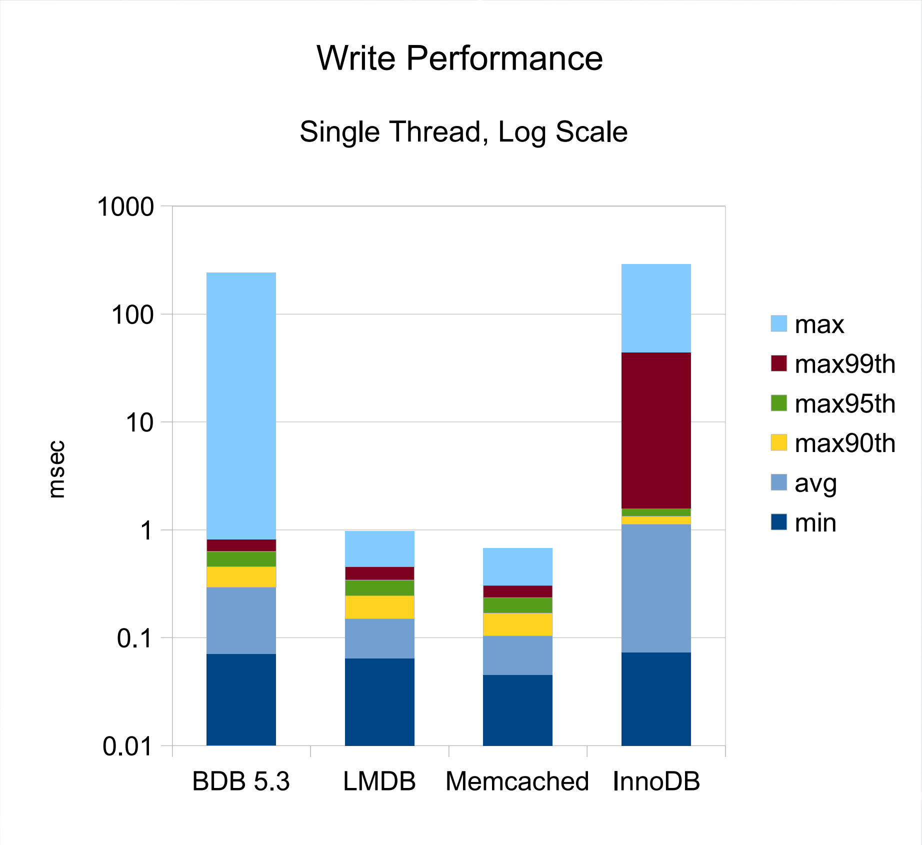
- 16 CPU cores, 64 concurrent readers



Results

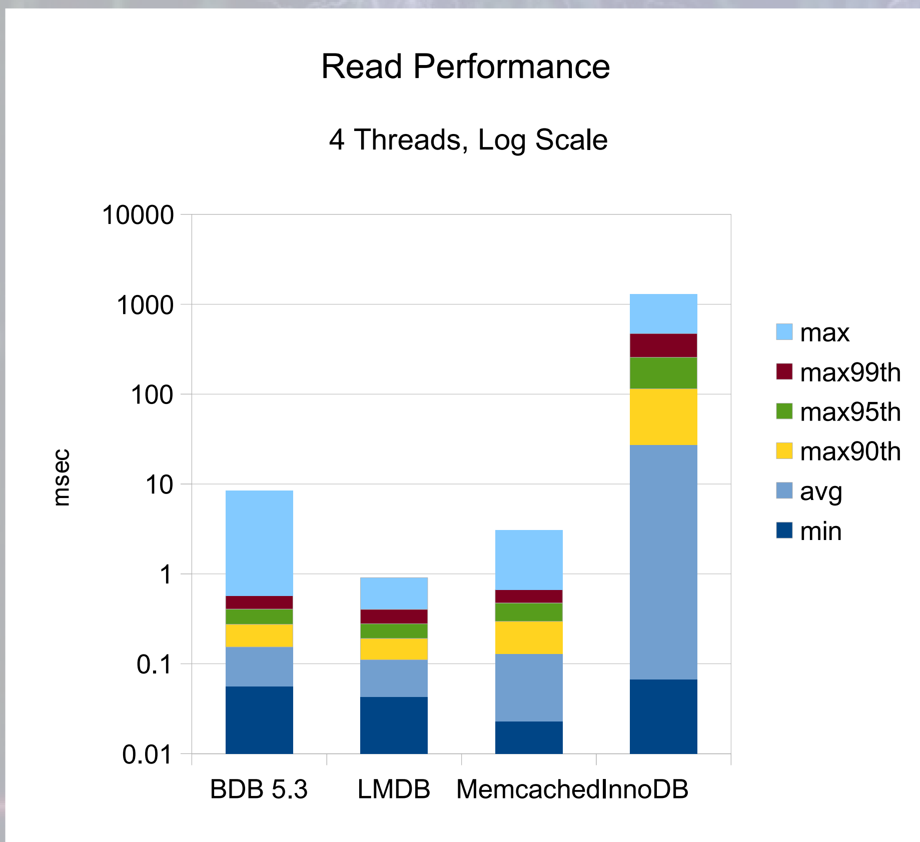
- Memcached

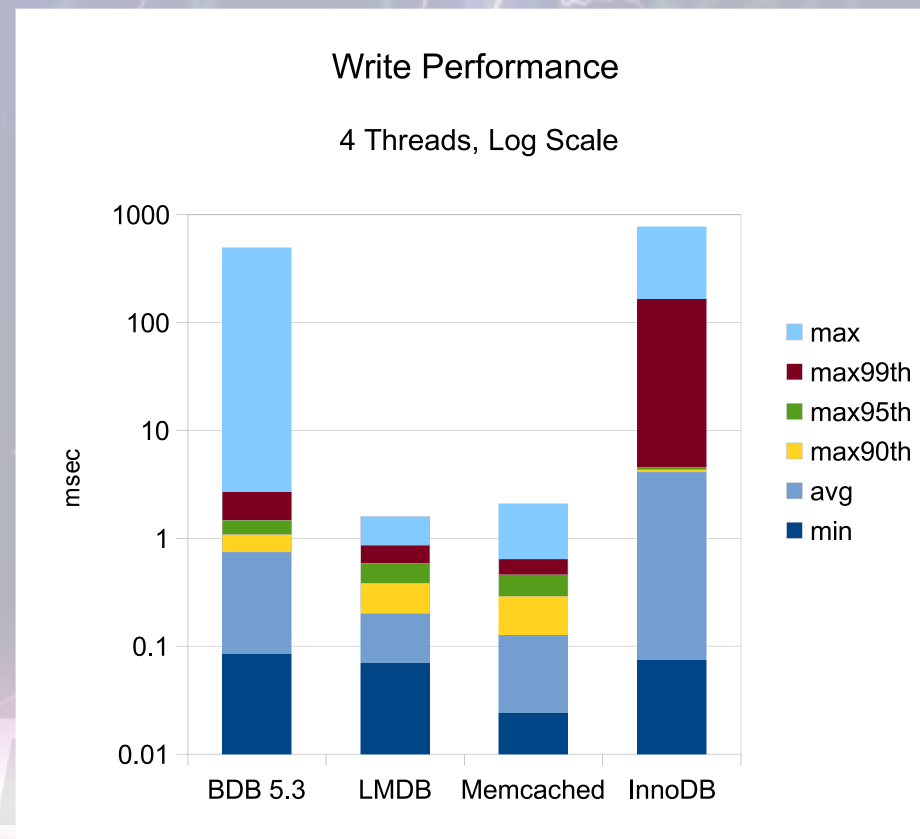




Results

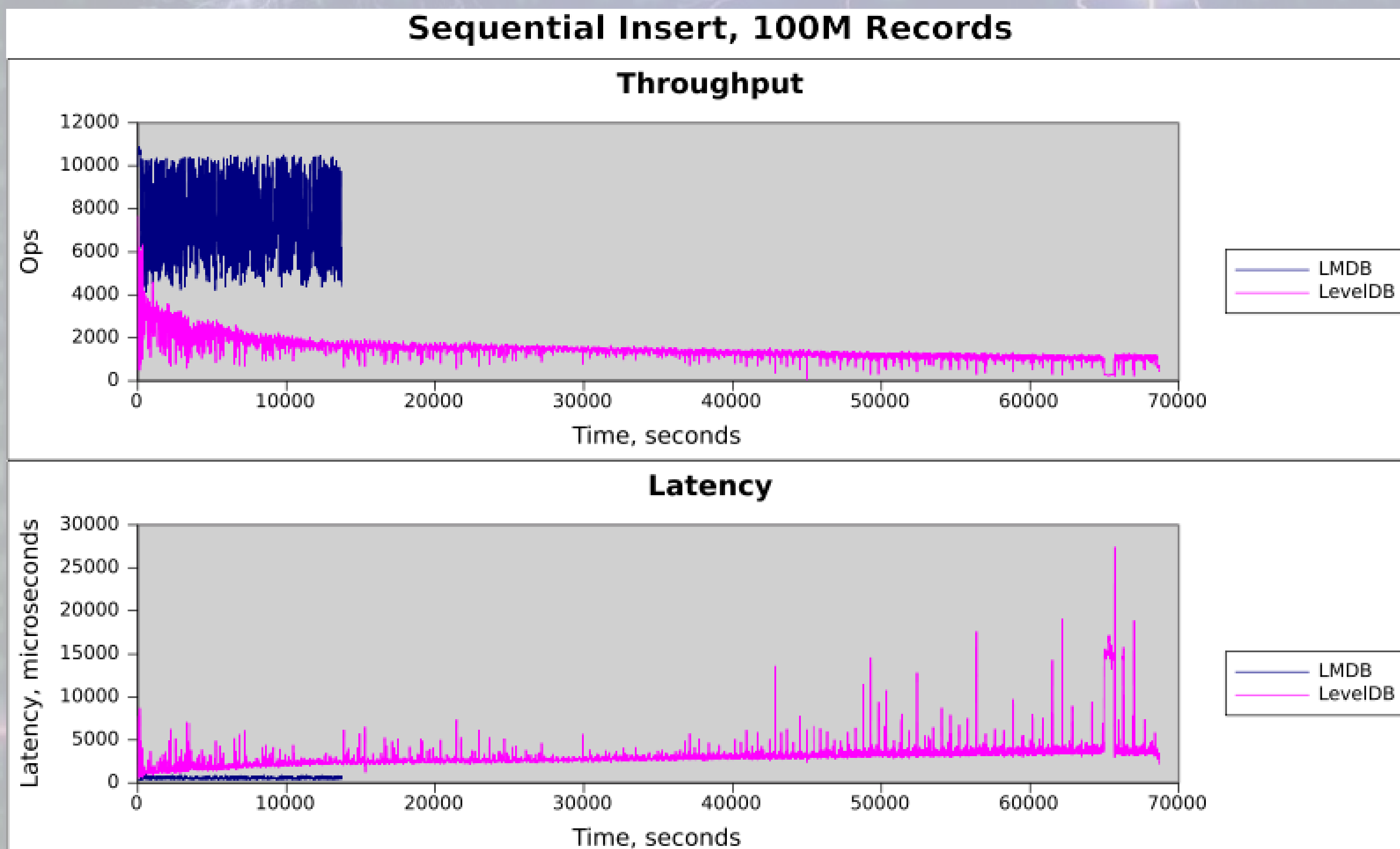
- Memcached





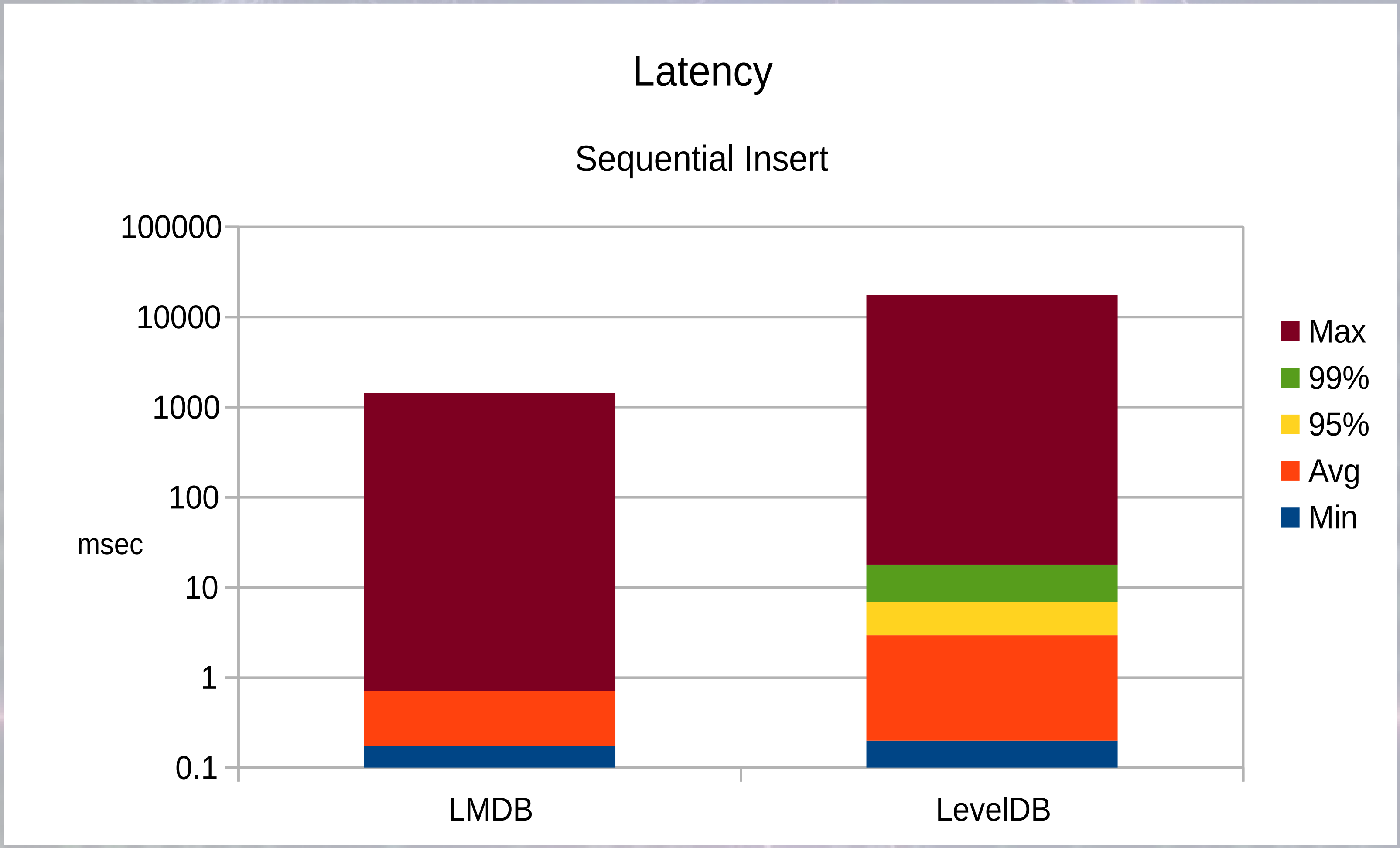
Results

- HyperDex/YCSB



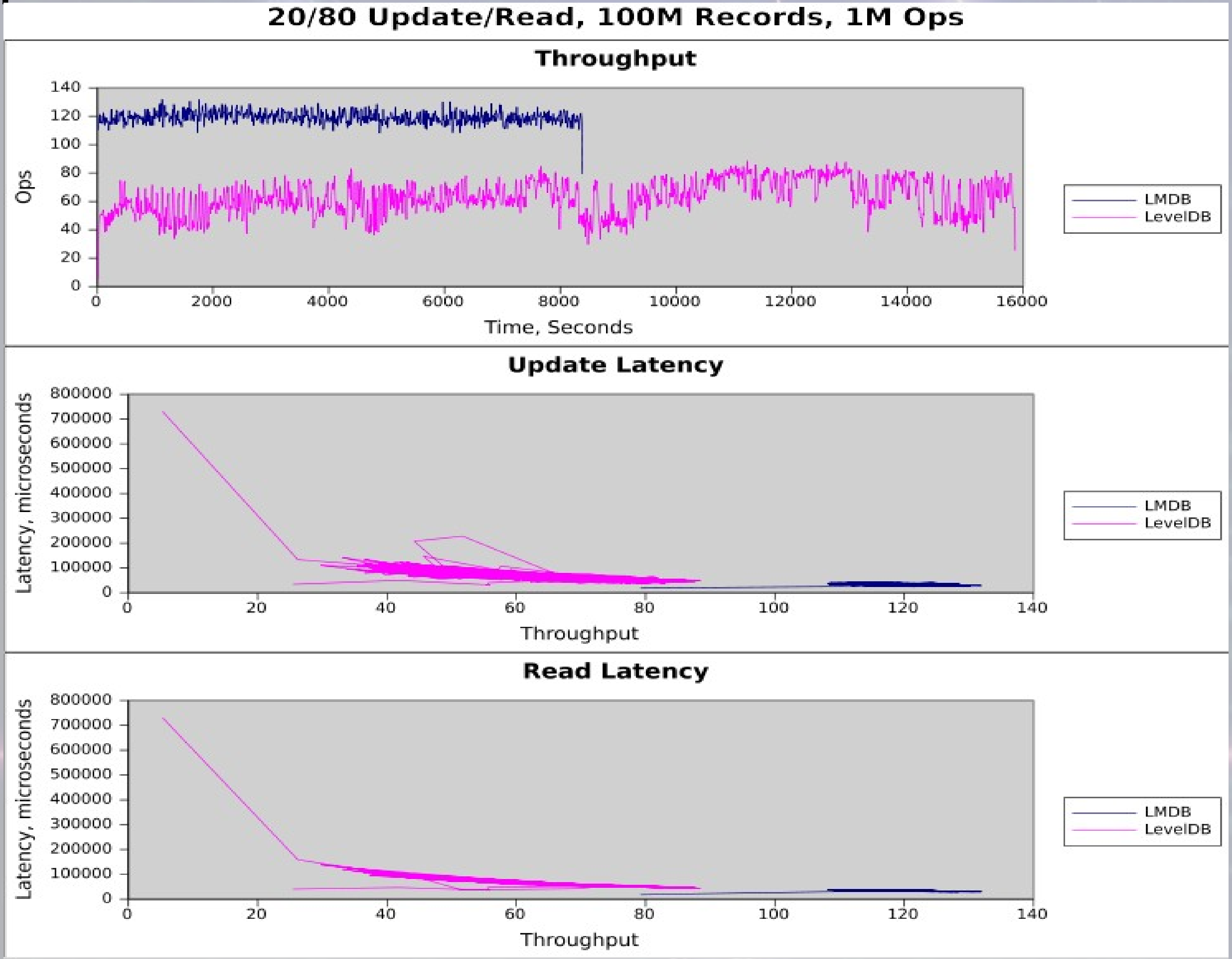
Results

- HyperDex/YCSB



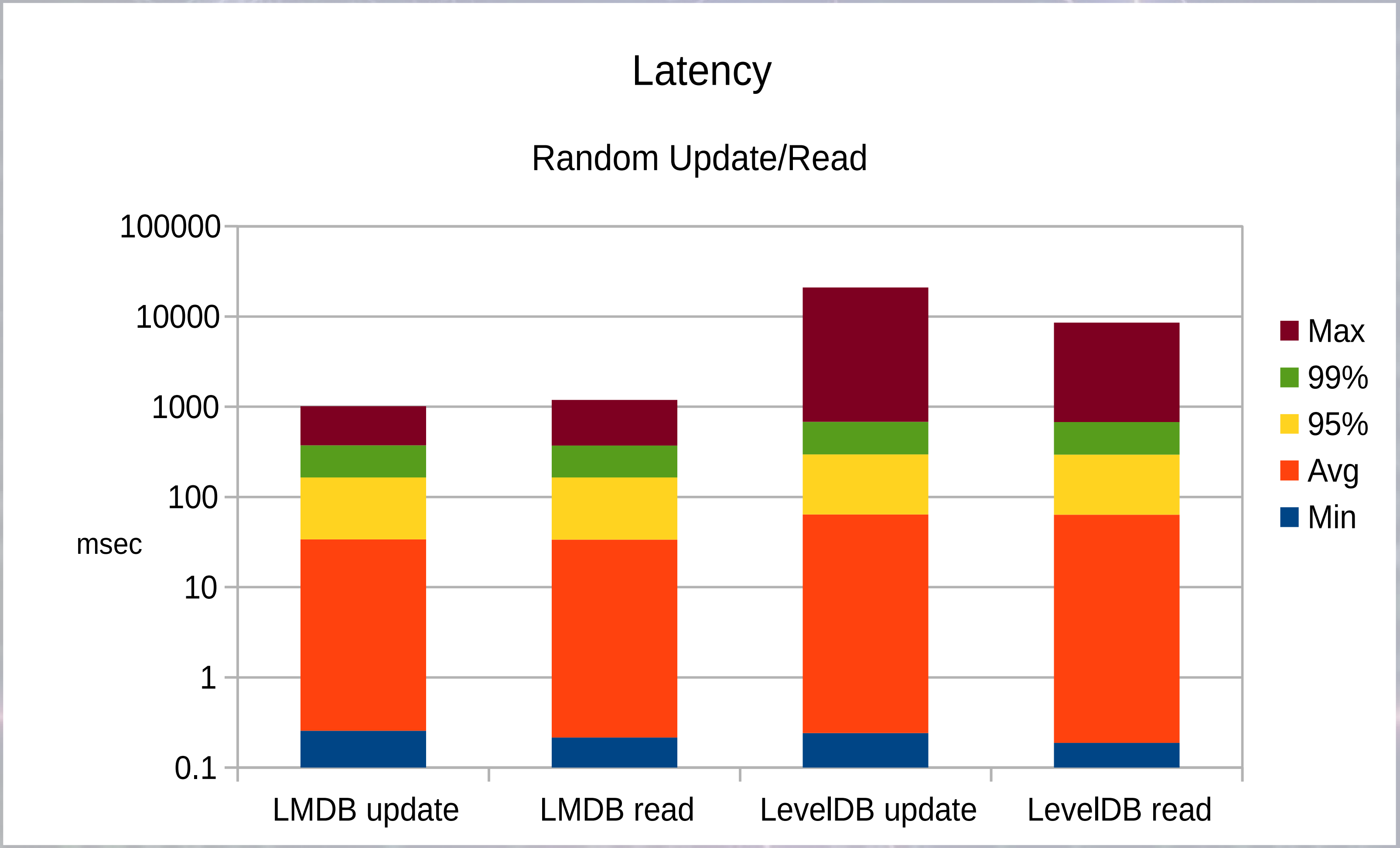
Results

- HyperDex/YCSB



Results

- HyperDex/YCSB



Results

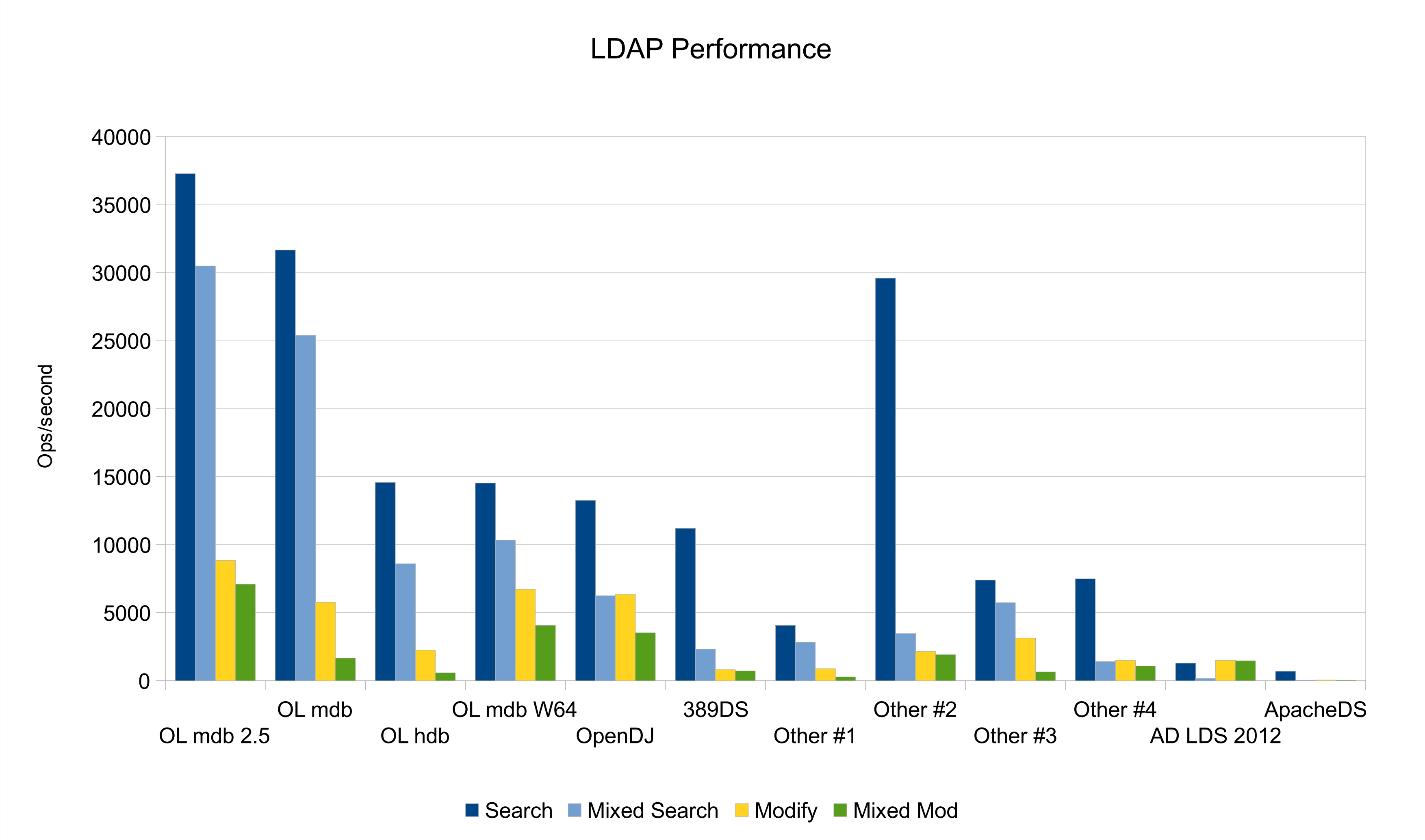
- An Interview with Armory Technologies CEO Alan Reiner
  - *JMC For more normal users, who have been frustrated with long load times. In my testing of the latest beta build, using bitcoin 0.10 and the new headers first format, I've seen you optimise the load time from 3 days, to less than 2 hours now. Well done! Can you talk us through how you did this?*
  - *AR. It really comes down to the new database engine (LMDB instead of LevelDB) and really hard [work] by some of our developers to reshape the architecture and the optimizations of the databases*
- http://bitcoinsinireland.com/an-interview-with-armory-technologies-ceo-alan-reiner/

Results

- LDAP Benchmarks - compared to:
  - OpenLDAP 2.4 back-mdb and -hdb
  - OpenLDAP 2.4 back-mdb on Windows 2012 x64
  - OpenDJ 2.4.6, 389DS, ApacheDS 2.0.0-M13
  - Latest proprietary servers from CA, Microsoft, Novell, and Oracle
  - Test on a VM with 32GB RAM, 10M entries

Results

- LDAP Benchmarks



Results

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

Conclusions

- The combination of memory-mapped operation with MVCC is extremely potent
  - Reduced administrative overhead
    - no periodic cleanup / maintenance required
    - no particular tuning required
  - Reduced developer overhead
    - code size and complexity drastically reduced
  - Enhanced efficiency
    - minimal CPU and I/O use
      - allows for longer battery life on mobile devices
      - allows for lower electricity/cooling costs in data centers
      - allows more work to be done with less hardware

Futures

- Work in progress
  - Incremental backup
    - Add a txnID stamp to the page header
    - Copy all pages with txnID newer than N
  - Write-Ahead Log
    - Despite misgivings from BDB experience
    - Prototype shows 30x improvement on synchronous write speed
    - fsync() on a small log file is much faster than on large DB file

Questions?