LDAP at Lightning Speed

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

2014-11-20

HTTP://SYMAS.COM

OpenLDAP Project

- Open source code project
- Founded 1998
- Three 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) API Overview

(4) Design Approach

(5) Internals

(6) Special Features

(7) Results

(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

(3) API Overview

- Based on BDB Transactional API
  - BDB apps can easily be migrated to LMDB
- Written in C
  - C/C++ supported directly
  - Wrappers for all popular languages available
- All functions return 0 on success or a non-zero error code on failure
  - except some void functions which cannot fail

API Overview

- All DB operations are transactional
  - There is no non-transactional interface
- Results fetched from the DB are owned by the DB
  - Point directly to the mmap contents, not memcpy'd
  - Need no disposal, callers can use the data then forget about it
  - Read-only by default, attempts to overwrite data will trigger a SIGSEGV

API Overview

- Most function names are grouped by purpose:
  - Environment:
    - mdb_env_create, mdb_env_open, mdb_env_sync, mdb_env_close
  - Transaction:
    - mdb_txn_begin, mdb_txn_commit, mdb_txn_abort
  - Cursor:
    - mdb_cursor_open, mdb_cursor_close, mdb_cursor_get, mdb_cursor_put, mdb_cursor_del
  - Database/Generic:
    - mdb_dbi_open, mdb_dbi_close, mdb_get, mdb_put, mdb_del

API Overview

LMDB Sample

```
#include <stdio.h>
#include <lmdb.h>

int main(int argc, char *argv[])
{
    int rc;
    MDB_env *env;
    MDB_txn *txn;
    MDB_cursor *cursor;
    MDB_dbi dbi;
    MDB_val key, data;
    char sval[32];

    rc = mdb_env_create(&env);
    rc = mdb_env_open(env,
      "./testdb", 0, 0664);
    rc = mdb_txn_begin(env, NULL,
      0, &txn);
    rc = mdb_open(txn, NULL, 0,
      &dbi);

    key.mv_size = sizeof(int);
    key.mv_data = sval;
    data.mv_size = sizeof(sval);
    data.mv_data = sval;

    sprintf(sval, "%03x %d foo bar", 32, 3141592);
    rc = mdb_put(txn, dbi, &key, &data, 0);
    rc = mdb_txn_commit(txn);
    if (rc) {
        fprintf(stderr, "mdb_txn_commit: (%d) %s\n",
          rc, mdb_strerror(rc));
        goto leave;
    }
    rc = mdb_txn_begin(env, NULL, MDB_RDONLY, &txn);
    rc = mdb_cursor_open(txn, dbi, &cursor);
    while ((rc = mdb_cursor_get(cursor, &key, &data,
      MDB_NEXT)) == 0) {
        printf("key: %p %.*s, data: %p %.*s\n",
            key.mv_data,
            (int) key.mv_size,
            (char *) key.mv_data,
            data.mv_data,
            (int) data.mv_size,
            (char *) data.mv_data);
    }
    mdb_cursor_close(cursor);
    mdb_txn_abort(txn);
leave:
    mdb_close(env, dbi);
    mdb_env_close(env);
    return rc;
}
```

API Overview

BDB Sample

```
#include <stdio.h>
#include <string.h>
#include <db.h>

int main(int argc, char *argv[])
{
    int rc;
    DB_ENV *env;
    DB_TXN *txn;
    DBC *cursor;
    DB *dbi;
    DBT key, data;
    char sval[32], kval[32];

#define FLAGS (DB_INIT_LOCK|DB_INIT_LOG|DB_INIT_TXN|
 DB_INIT_MPOOL|DB_CREATE|DB_THREAD)
    rc = db_env_create(&env, 0);
    rc = env>open(env, "./testdb", FLAGS,
      0664);
    rc = db_create(&dbi, env, 0);
    rc = env>txn_begin(env, NULL, &txn, 0);
    rc = dbi>open(dbi, txn, "test.bdb", NULL,
      DB_BTREE, DB_CREATE, 0664);

    memset(&key, 0, sizeof(DBT));
    memset(&data, 0, sizeof(DBT));
    key.size = sizeof(int);
    key.data = sval;
    data.size = sizeof(sval);
    data.data = sval;
    sprintf(sval, "%03x %d foo bar", 32, 3141592);
    rc = dbi>put(dbi, txn, &key, &data, 0);
    rc = txn>commit(txn, 0);
    if (rc) {
        fprintf(stderr, "txn>commit: (%d) %s\n",
          rc, db_strerror(rc));
        goto leave;
    }
    rc = env>txn_begin(env, NULL, &txn, 0);
    rc = dbi>cursor(dbi, txn, &cursor, 0);
    key.flags = DB_DBT_USERMEM;
    key.data = kval;
    key.ulen = sizeof(kval);
    data.flags = DB_DBT_USERMEM;
    data.data = sval;
    data.ulen = sizeof(sval);
    while ((rc = cursor>c_get(cursor, &key, &data,
      DB_NEXT)) == 0) {
        printf("key: %p %.*s, data: %p %.*s\n",
            key.data,
            (int) key.size,
            (char *) key.data,
            data.data,
            (int) data.size,
            (char *) data.data);
    }
    rc = cursor>c_close(cursor);
    rc = txn>abort(txn);
leave:
    rc = dbi>close(dbi, 0);
    rc = env>close(env, 0);
    return rc;
}
```

API Overview

LMDB Sample

```
#include <stdio.h>

#include <lmdb.h>

int main(int argc, char *argv[])
{
    int rc;
    MDB_env *env;
    MDB_txn *txn;
    MDB_cursor *cursor;
    MDB_dbi dbi;
    MDB_val key, data;
    char sval[32];




    rc = mdb_env_create(&env);
    rc = mdb_env_open(env, "./testdb", 0,
      0664);

    rc = mdb_txn_begin(env, NULL, 0, &txn);
    rc = mdb_open(txn, NULL, 0, &dbi);
```

BDB Sample

```
#include <stdio.h>
#include <string.h>
#include <db.h>

int main(int argc, char *argv[])
{
    int rc;
    DB_ENV *env;
    DB_TXN *txn;
    DBC *cursor;
    DB *dbi;
    DBT key, data;
    char sval[32], kval[32];

#define FLAGS (DB_INIT_LOCK|DB_INIT_LOG|DB_INIT_TXN|DB_INIT_MPOOL|
 DB_CREATE|DB_THREAD)
    rc = db_env_create(&env, 0);
    rc = env>open(env, "./testdb", FLAGS,
      0664);
    rc = db_create(&dbi, env, 0);
    rc = env>txn_begin(env, NULL, &txn, 0);
    rc = dbi>open(dbi, txn, "test.bdb",
      NULL, DB_BTREE, DB_CREATE, 0664);
```

API Overview

LMDB Sample

```
key.mv_size = sizeof(int);
key.mv_data = sval;
data.mv_size = sizeof(sval);
data.mv_data = sval;
sprintf(sval, "%03x %d foo bar",
  32, 3141592);
rc = mdb_put(txn, dbi, &key, &data, 0);
rc = mdb_txn_commit(txn);
if (rc) {
    fprintf(stderr, "mdb_txn_commit: (%d) %s\n", rc,
mdb_strerror(rc));
    goto leave;
}
```

BDB Sample

```
memset(&key, 0, sizeof(DBT));
memset(&data, 0, sizeof(DBT));
key.size = sizeof(int);
key.data = sval;
data.size = sizeof(sval);
data.data = sval;
sprintf(sval, "%03x %d foo bar",
  32, 3141592);
rc = dbi>put(dbi, txn, &key, &data, 0);
rc = txn>commit(txn, 0);
if (rc) {
    fprintf(stderr, "txn>commit: (%d) %s\n", rc, db_strerror(rc));
    goto leave;
}
```

API Overview

LMDB Sample

```
    rc = mdb_txn_begin(env, NULL, MDB_RDONLY,
      &txn);
    rc = mdb_cursor_open(txn, dbi, &cursor);







    while ((rc = mdb_cursor_get(cursor, &key,
      &data, MDB_NEXT)) == 0) {
        printf("key: %p %.*s, data: %p %.*s\n",
          key.mv_data,
          (int)key.mv_size,
          (char *)key.mv_data,
          data.mv_data,
          (int)data.mv_size,
          (char *)data.mv_data);
    }
    mdb_cursor_close(cursor);
    mdb_txn_abort(txn);
leave:
    mdb_close(env, dbi);
    mdb_env_close(env);
    return rc;
}
```

BDB Sample

```
    rc = env>txn_begin(env, NULL, &txn,
      0);
    rc = dbi>cursor(dbi, txn, &cursor, 0);
    key.flags = DB_DBT_USERMEM;
    key.data = kval;
    key.ulen = sizeof(kval);
    data.flags = DB_DBT_USERMEM;
    data.data = sval;
    data.ulen = sizeof(sval);
    while ((rc = cursor>c_get(cursor, &key,
      &data, DB_NEXT)) == 0) {
        printf("key: %p %.*s, data: %p %.*s\n",
          key.data,
          (int)key.size,
          (char *)key.data,
          data.data,
          (int)data.size,
          (char *)data.data);
    }
    rc = cursor>c_close(cursor);
    rc = txn>abort(txn);
leave:
    rc = dbi>close(dbi, 0);
    rc = env>close(env, 0);
    return rc;
}
```

API Overview

- LMDB naming is simple and consistent
  - MDB_xxx for all typedefs
  - BDB uses DB_XXX, DBX, DB_DBX_...
- LMDB environment setup is simple
  - BDB requires multiple subsystems to be initialized
- LMDB database setup is simple and reliable
  - BDB creates a file per DB
    - If the transaction containing the DB Open is aborted, rollback is very complicated because the filesystem operations to create the file cannot be rolled back atomically
    - Likewise during recover and replay of a transaction log

API Overview

- LMDB data is simple
  - BDB requires DBT structure to be fully zeroed out before use
  - BDB requires the app to manage the memory of keys and values returned from the DB
- LMDB teardown is simple
  - BDB *-close functions can fail, and there's nothing the app can do if a failure occurs

API Overview

- 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
```

(4) Design Approach

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

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

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 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

- 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

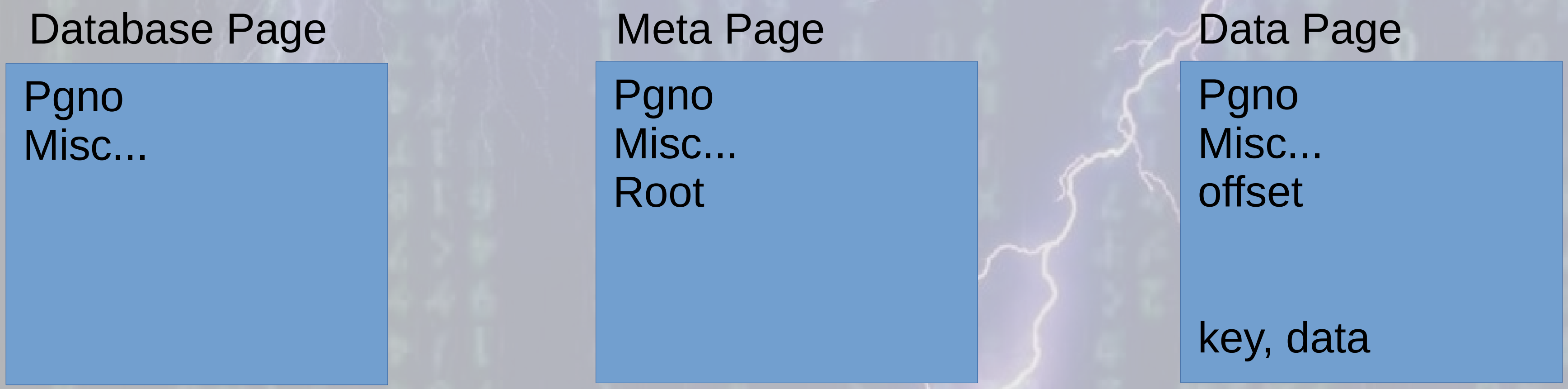
- Resulting library was under 32KB of object code
  - Compared to the original btree.c at 39KB
  - Compared to BDB at 1.5MB
- API is loosely modeled after the BDB API to ease migration of back-bdb code

(5) 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

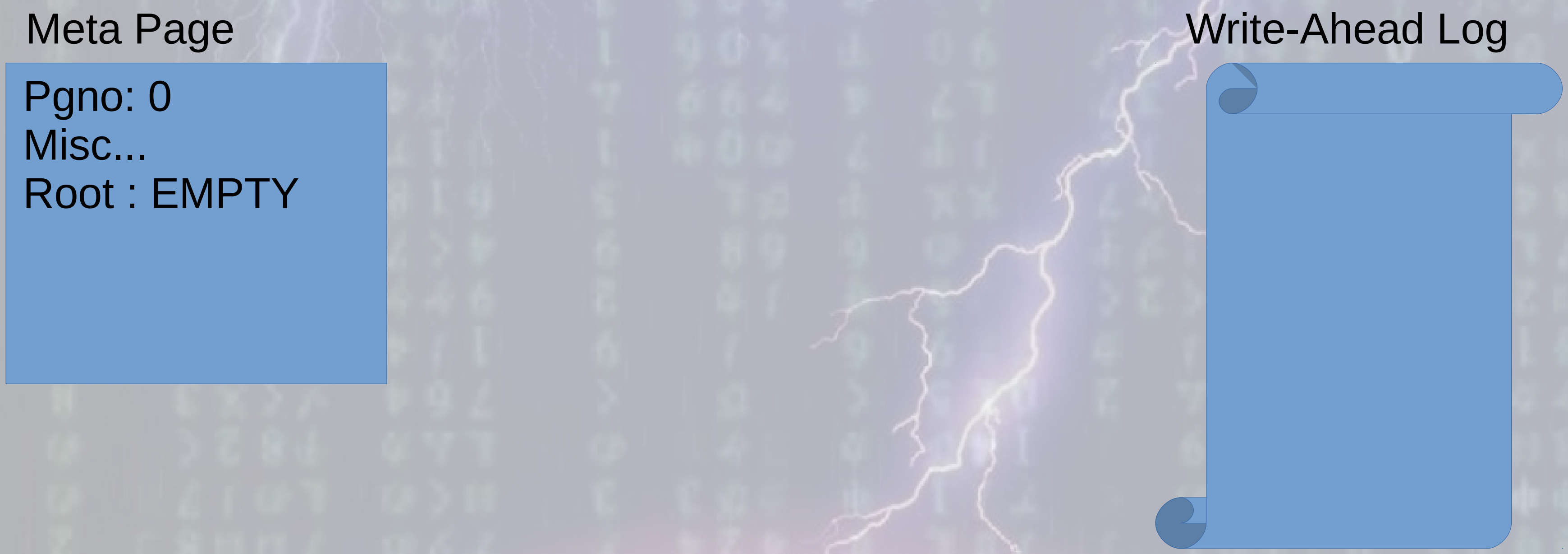
Data Page

Pgno
Misc...
offset

key, data

Btree Operation

Write-Ahead Logger



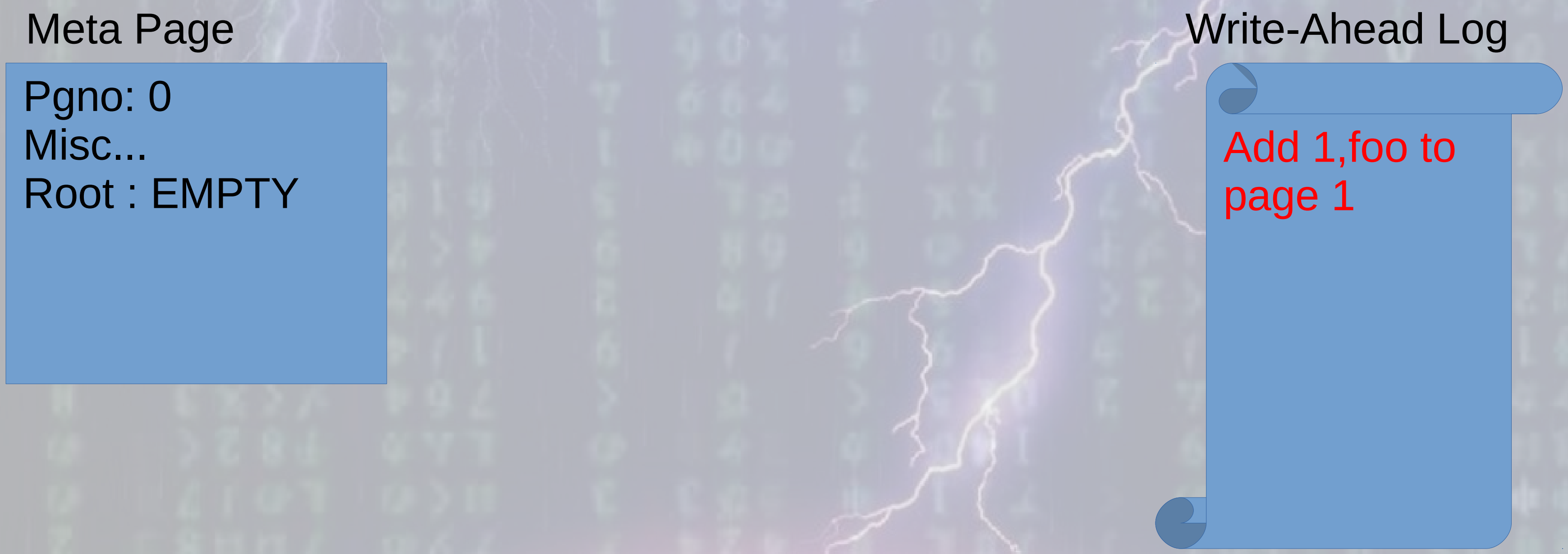
Meta Page

Pgno: 0
Misc...
Root : EMPTY

Write-Ahead Log

Btree Operation

Write-Ahead Logger



Meta Page

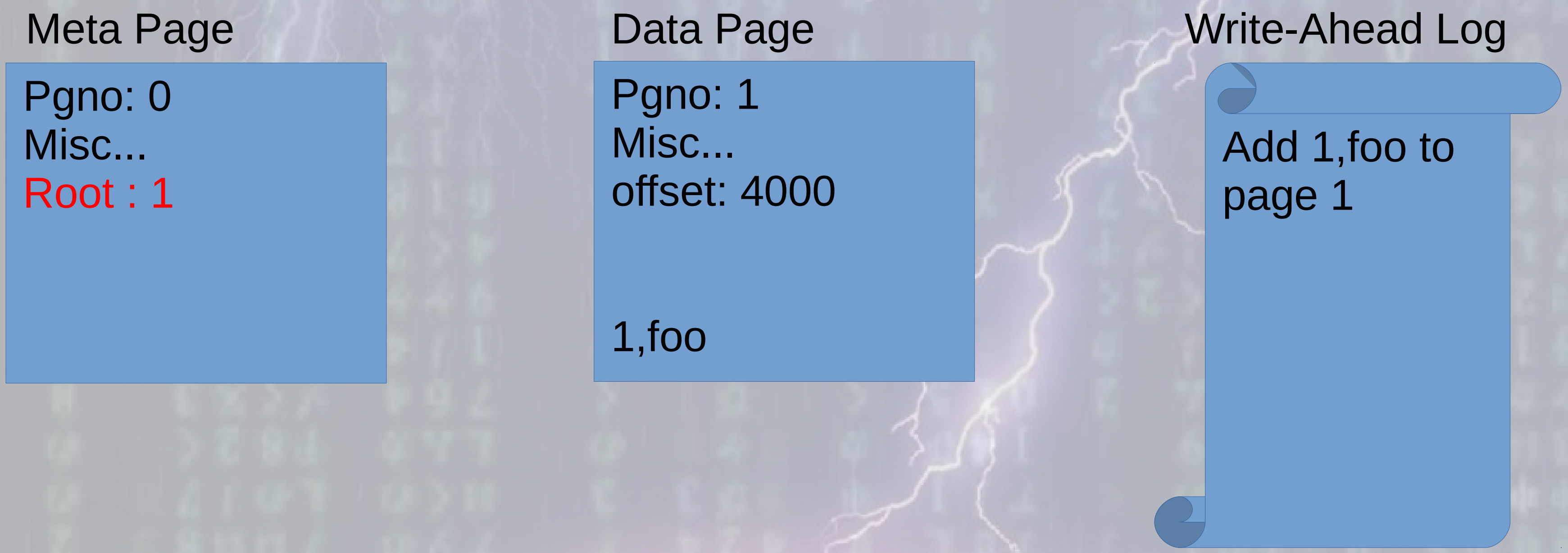
Pgno: 0
Misc...
Root : EMPTY

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

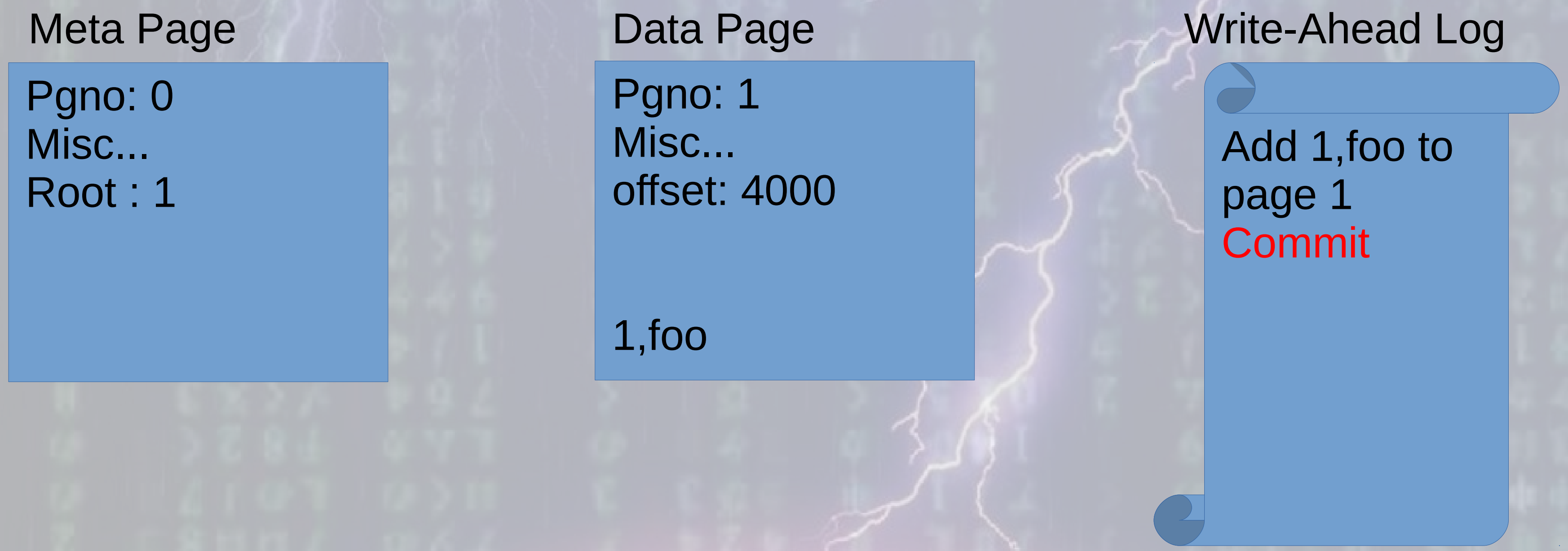
1,foo

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

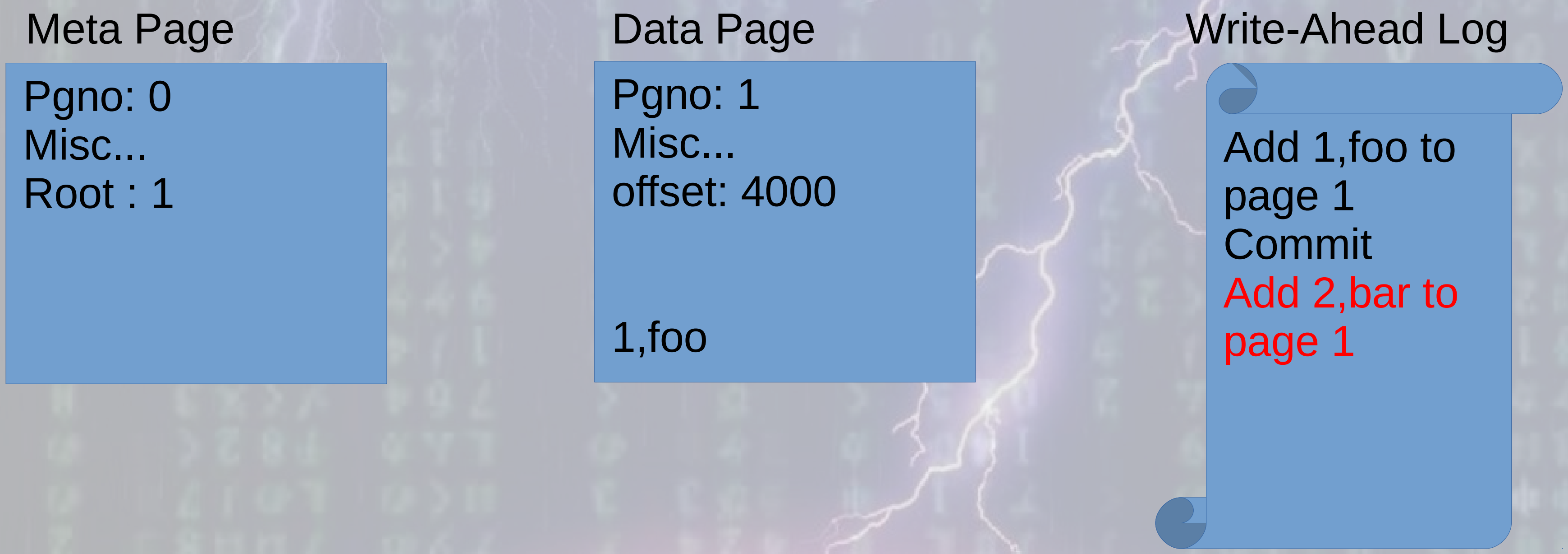
1,foo

Write-Ahead Log

Add 1,foo to page 1
Commit

Btree Operation

Write-Ahead Logger



Meta Page

Pgno: 0
Misc...
Root : 1

Data Page

Pgno: 1
Misc...
offset: 4000

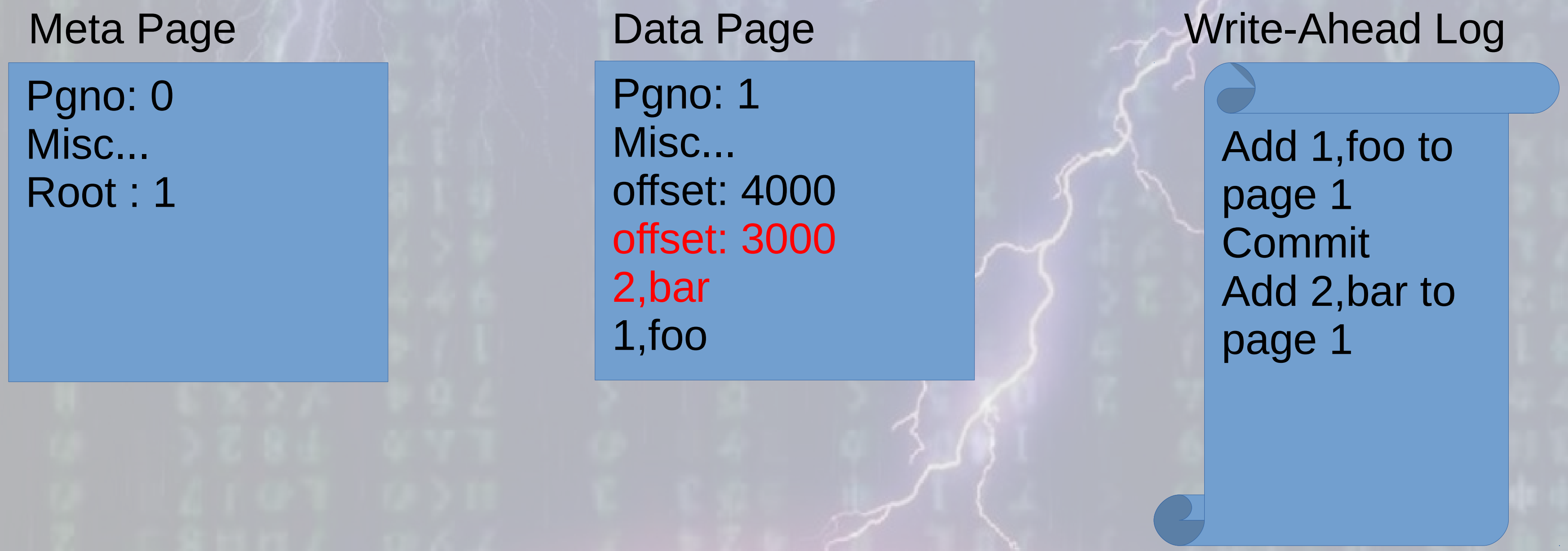
1,foo

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

Btree Operation

Write-Ahead Logger

Meta Page

Pgno: 0
Misc...
Root : 1

Data Page

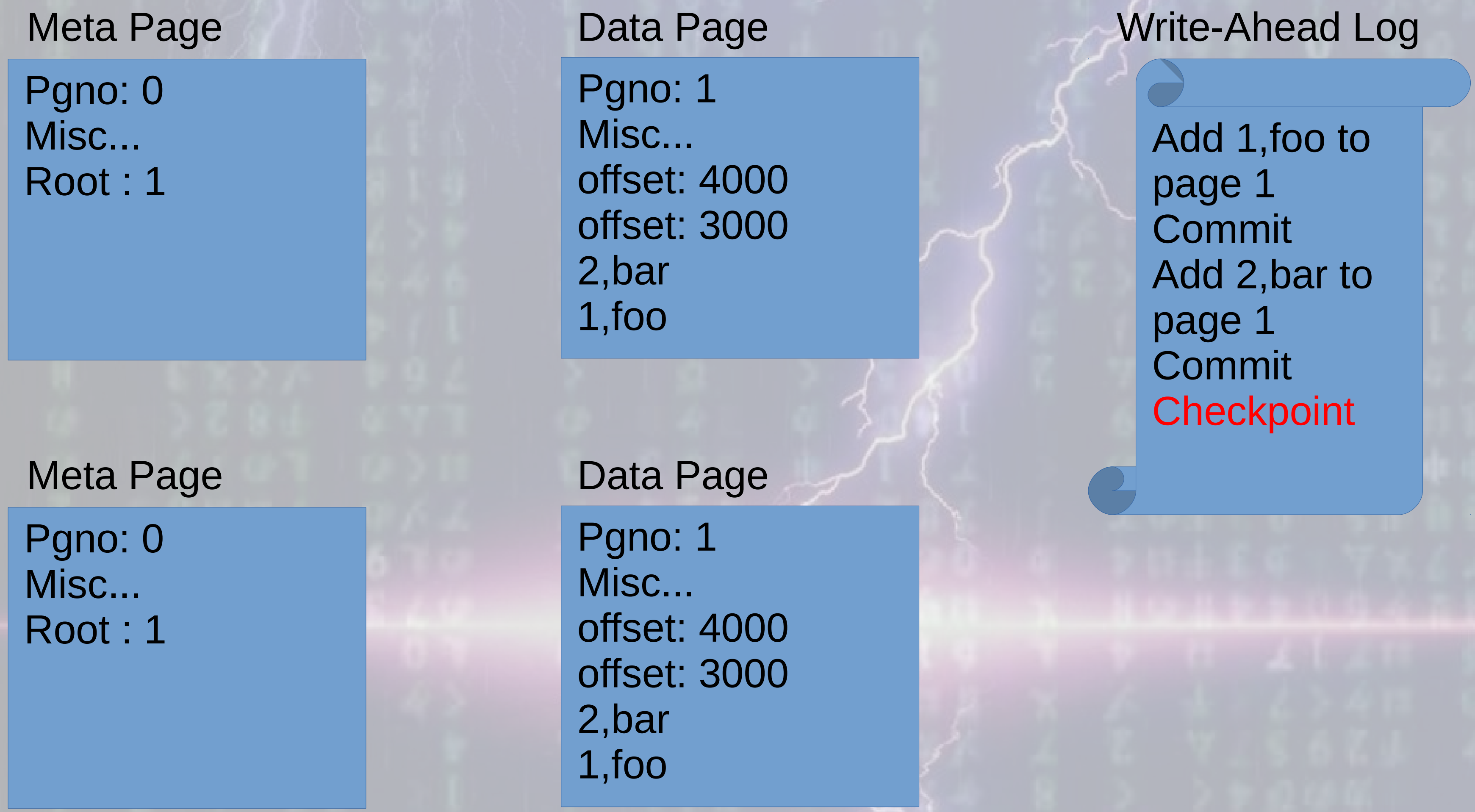
Pgno: 1
Misc...
offset: 4000
offset: 3000
2,bar
1,foo

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

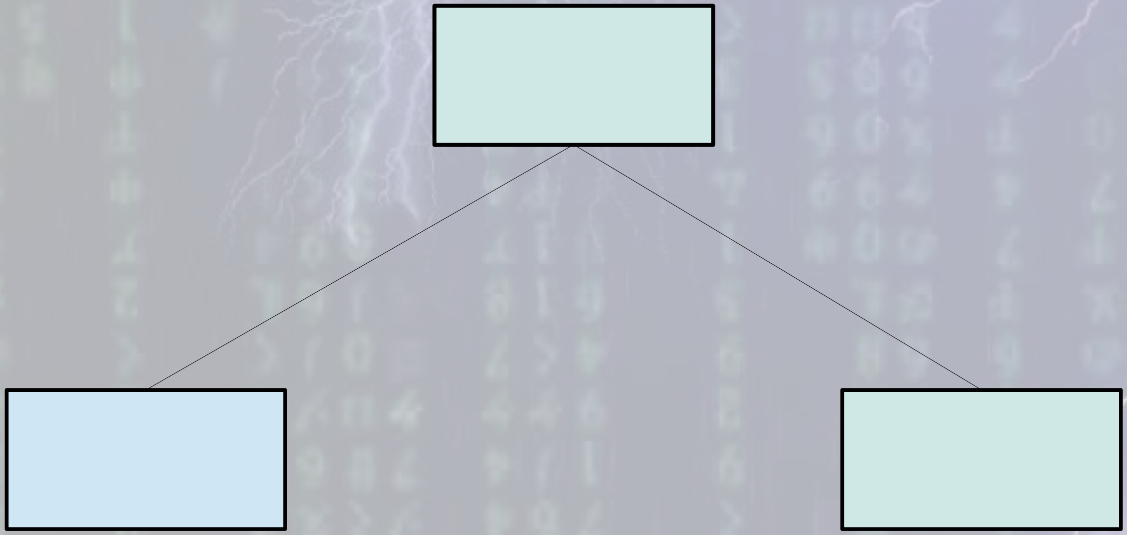
Btree Operation

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

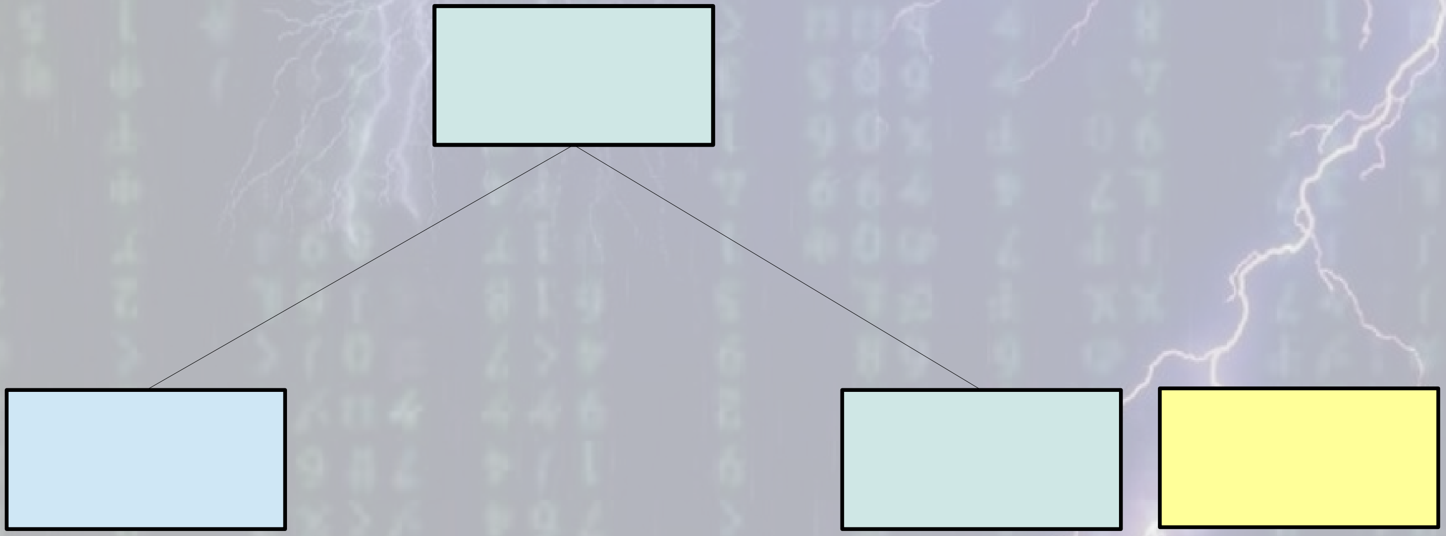
Append-Only



Start with a simple tree

Btree Operation

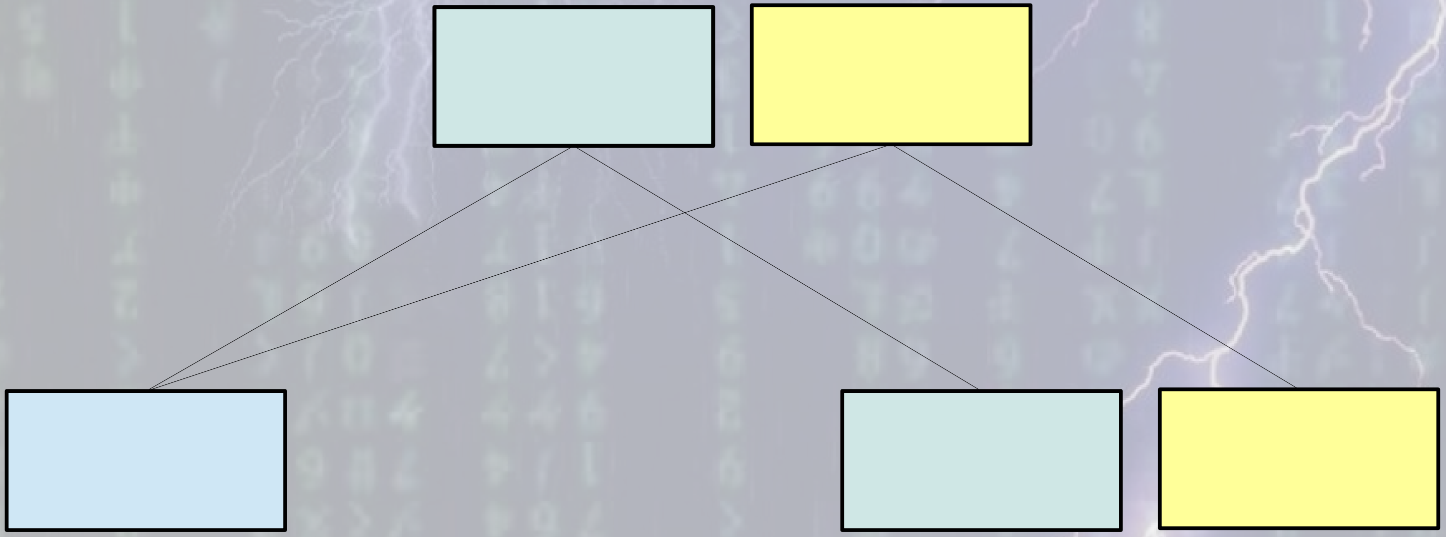
Append-Only



Update a leaf node by copying it and updating the copy

Btree Operation

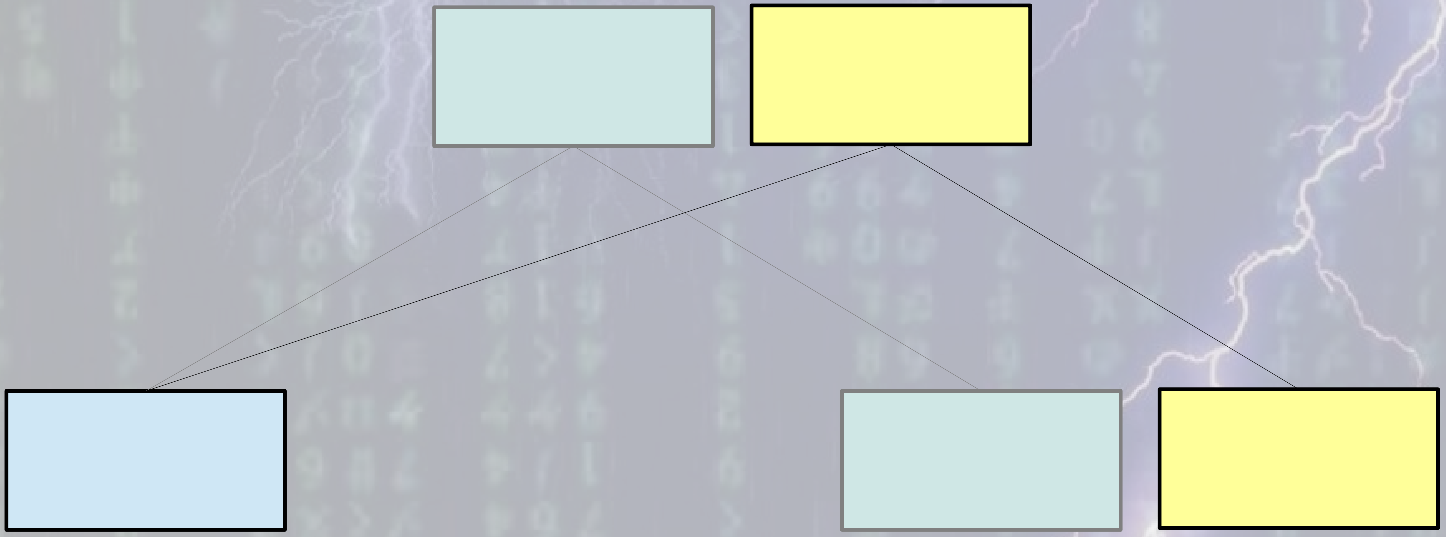
Append-Only



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

Btree Operation

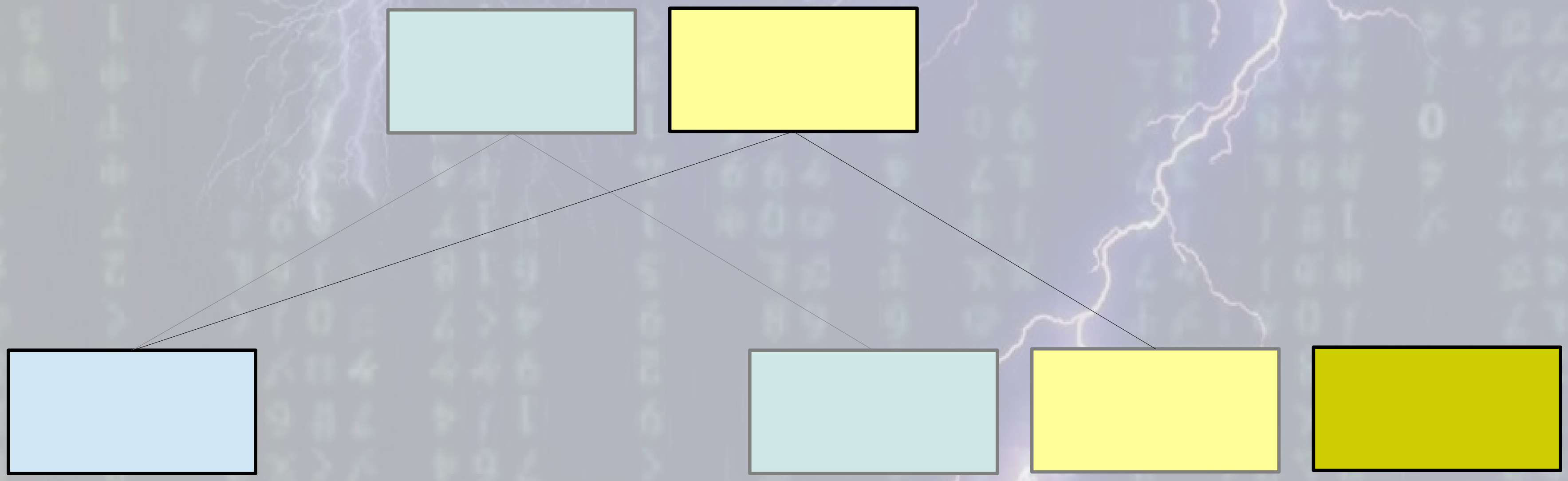
Append-Only



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

Btree Operation

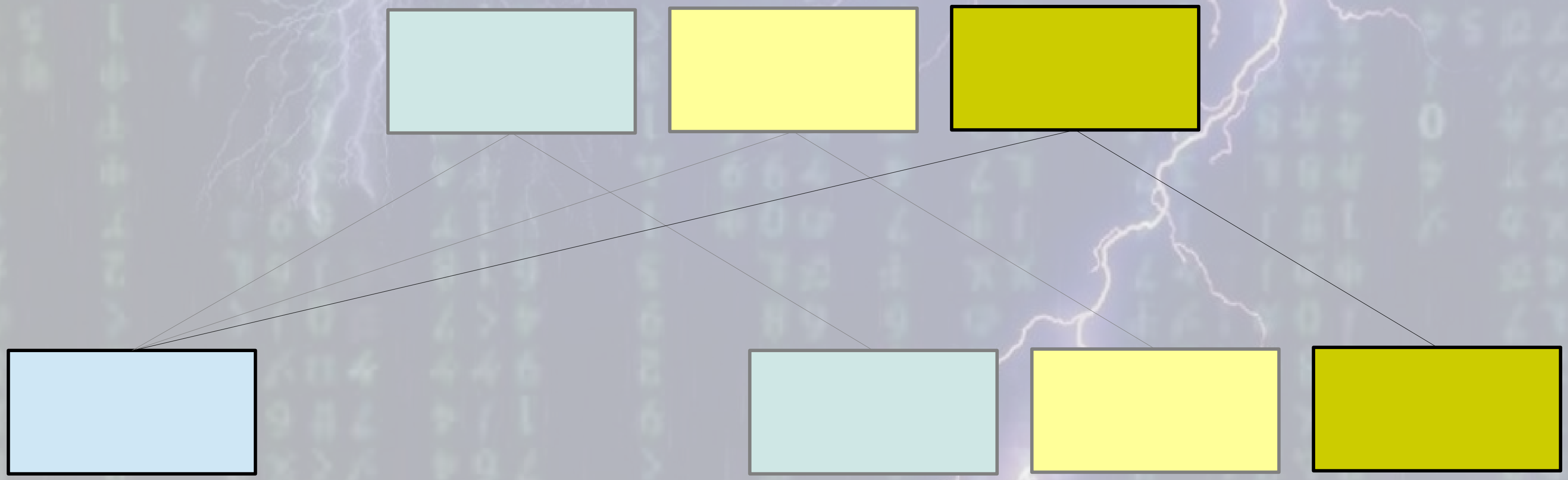
Append-Only



Further updates create additional versions

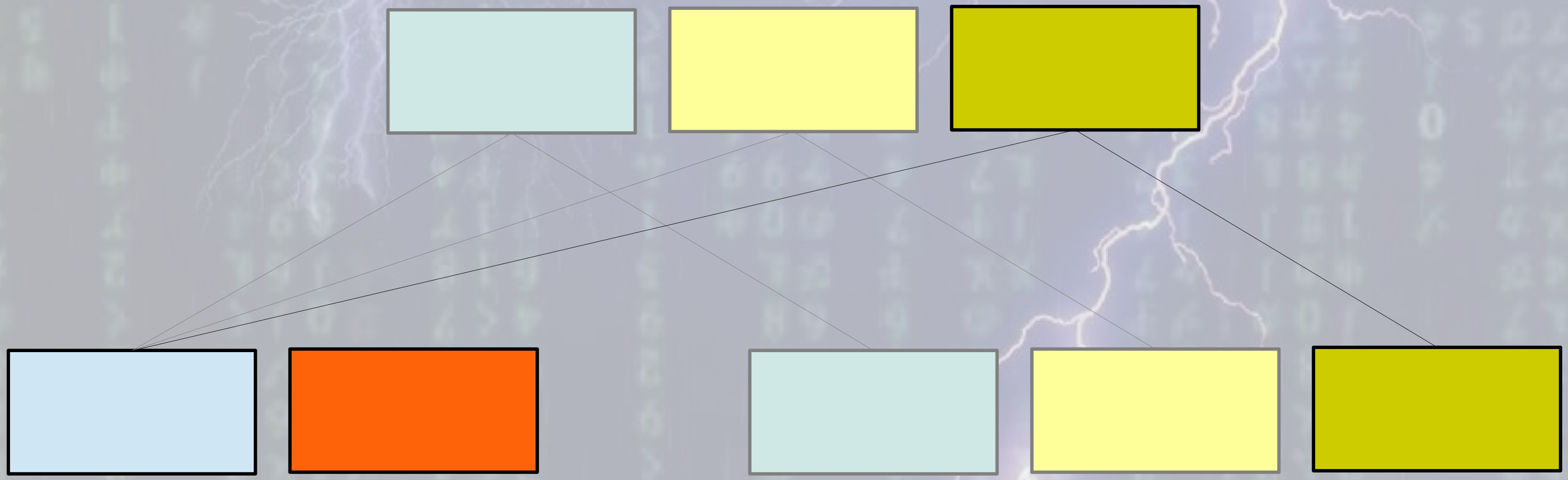
Btree Operation

Append-Only



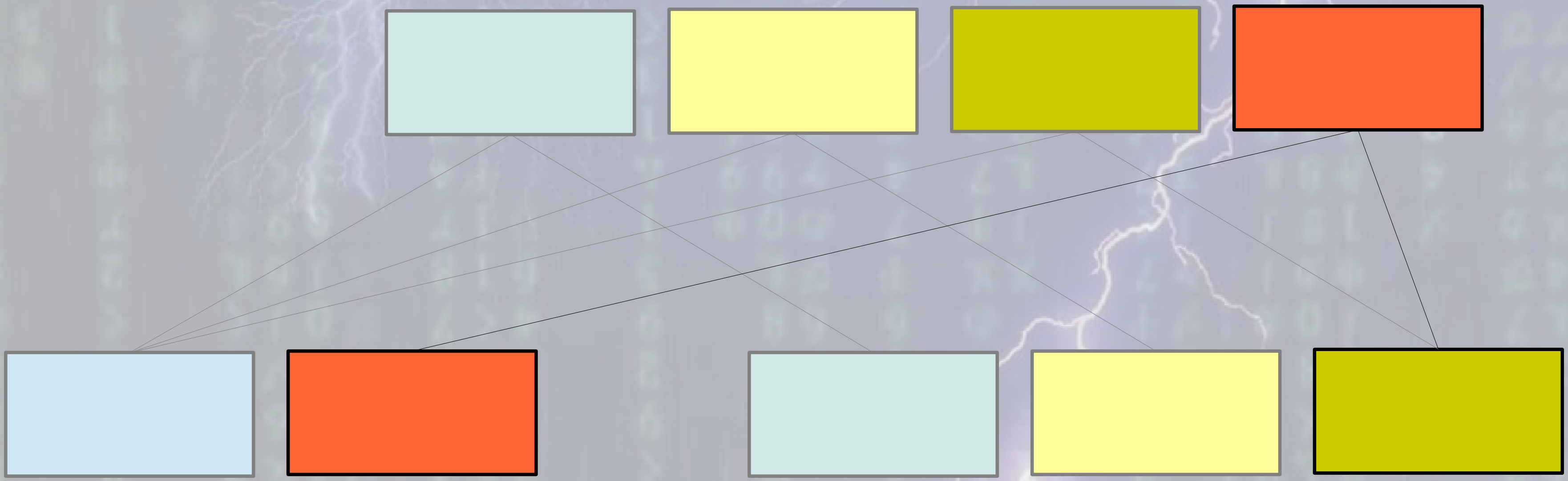
Btree Operation

Append-Only



Btree Operation

Append-Only



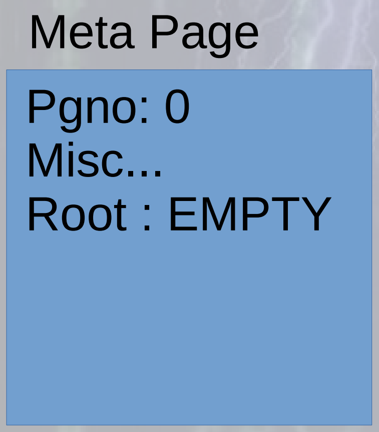
Btree Operation

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

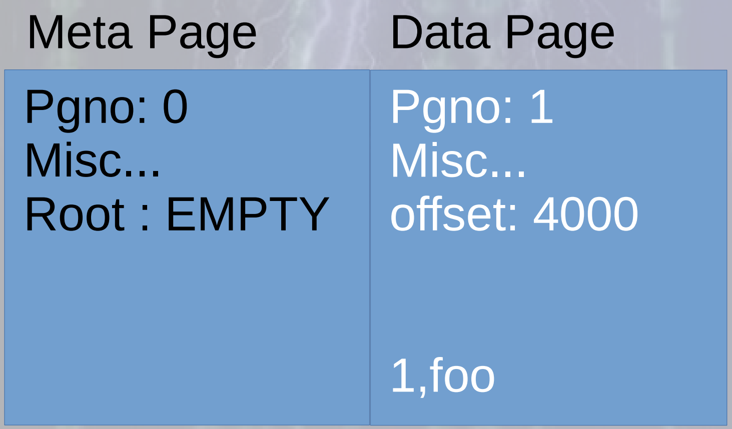


Meta Page

Pgno: 0
Misc...
Root : EMPTY

Btree Operation

Append-Only



Meta Page

Pgno: 0
Misc...
Root : EMPTY

Data Page

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

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

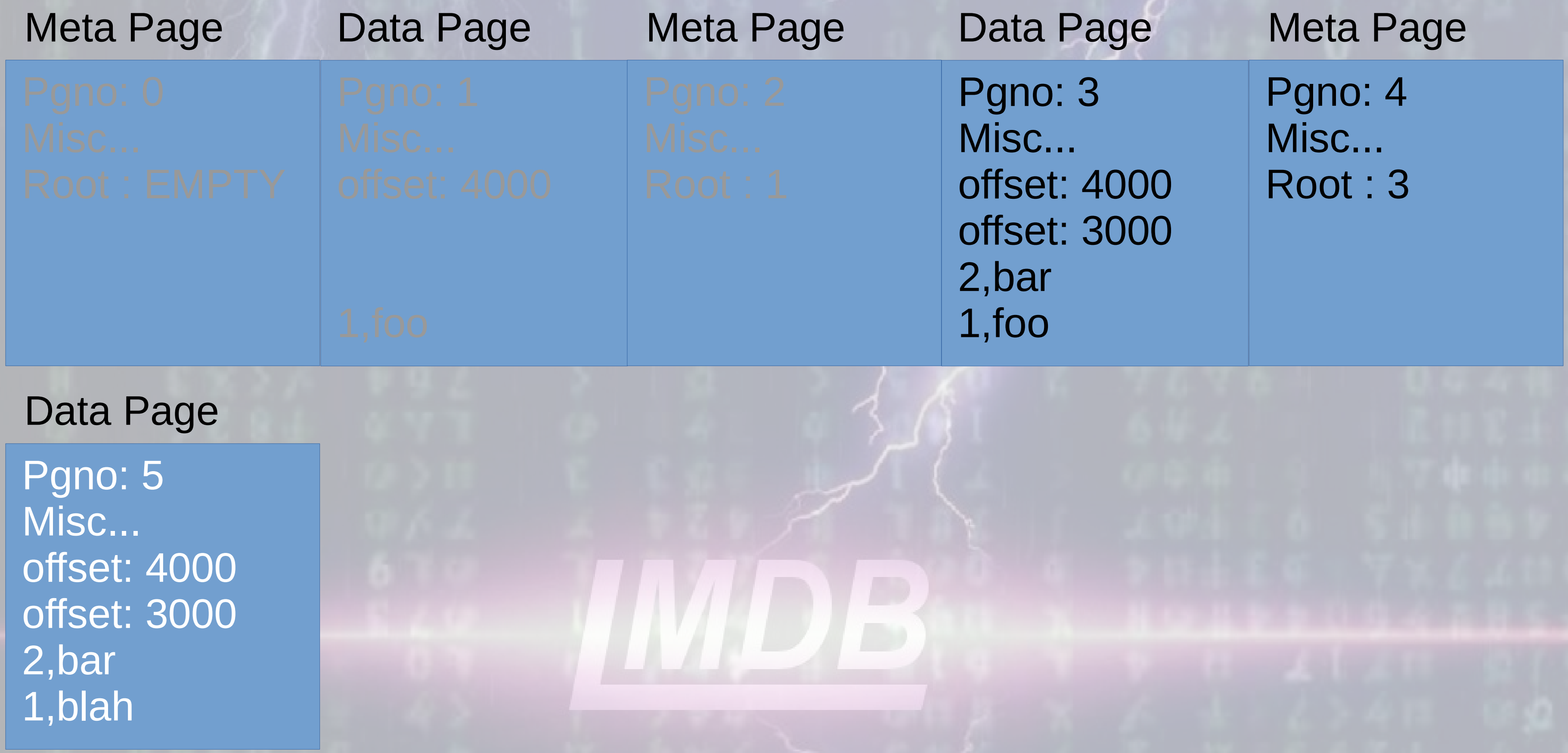
Btree Operation

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

Btree Operation

Append-Only



Meta Page

Pgno: 0
Misc...
Root : EMPTY

Data Page

Pgno: 1
Misc...
offset: 4000
1,foo

Meta Page

Pgno: 2
Misc...
Root : 1

Data Page

Pgno: 3
Misc...
offset: 4000
offset: 3000
2,bar
1,foo

Meta Page

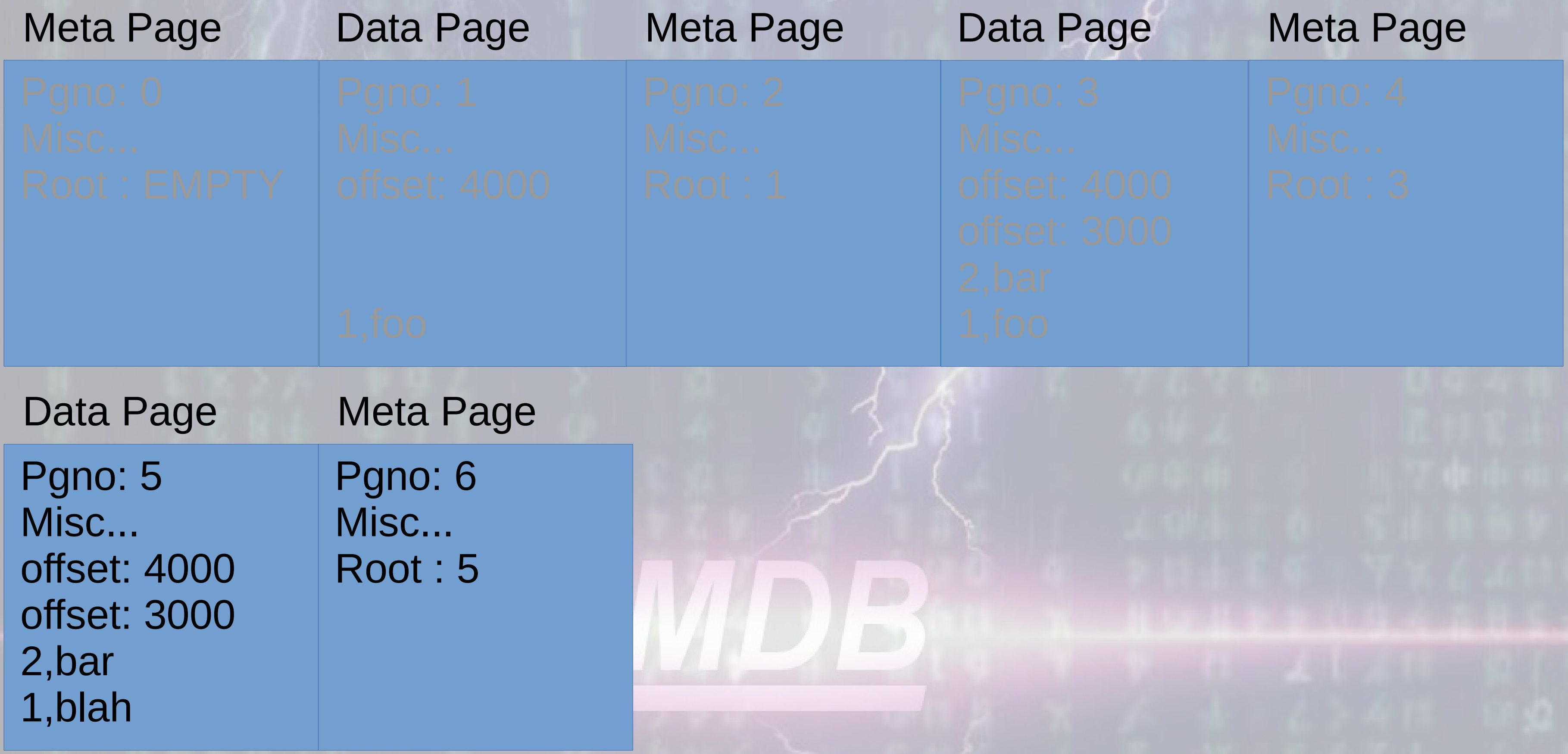
Pgno: 4
Misc...
Root : 3

Data Page

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

Btree Operation

Append-Only



Meta Page

Pgno: 0
Misc...
Root : EMPTY

Data Page

Pgno: 1
Misc...
offset: 4000
1,foo

Meta Page

Pgno: 2
Misc...
Root : 1

Data Page

Pgno: 3
Misc...
offset: 4000
offset: 3000
2,bar
1,foo

Meta Page

Pgno: 4
Misc...
Root : 3

Data Page

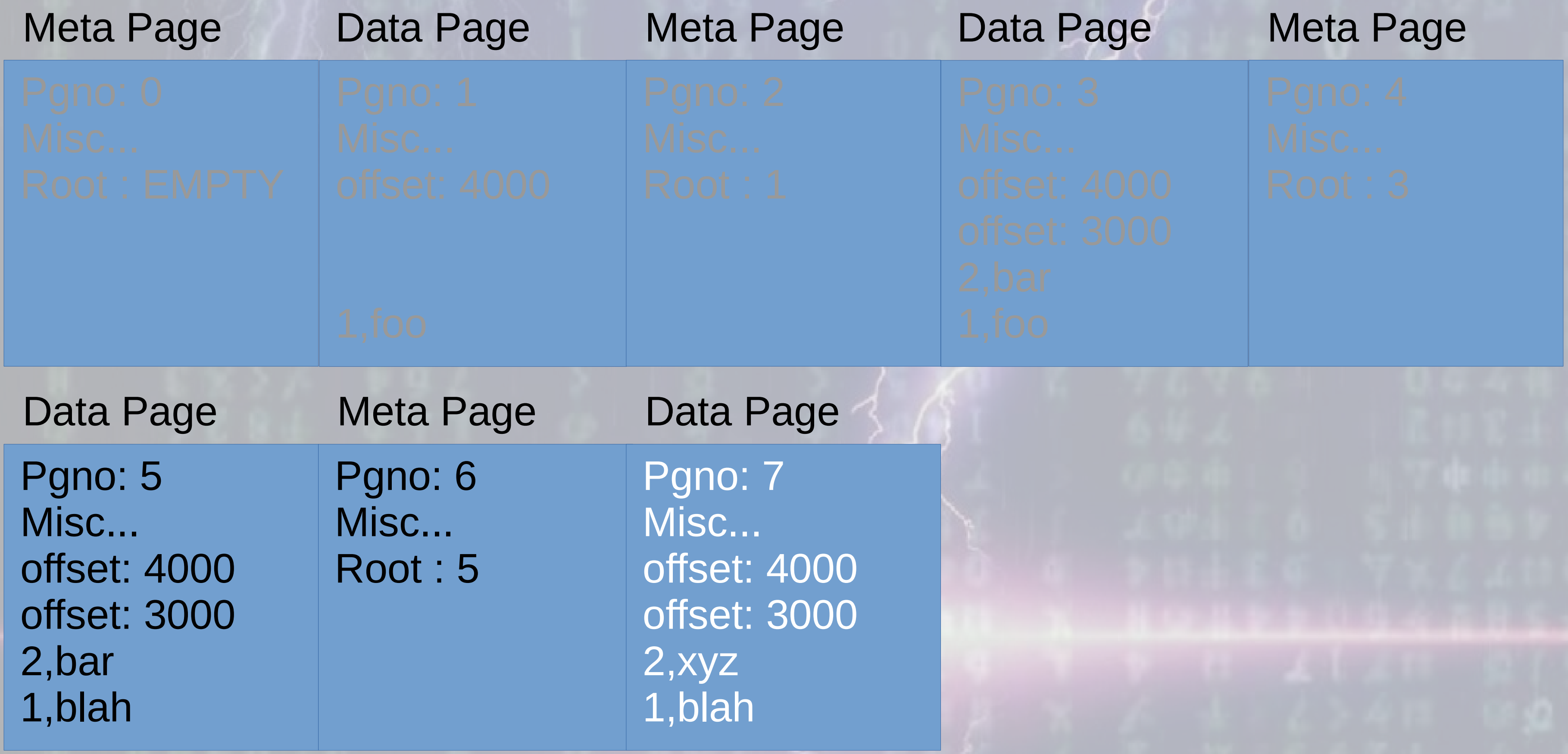
Pgno: 5
Misc...
offset: 4000
offset: 3000
2,bar
1,blah

Meta Page

Pgno: 6
Misc...
Root : 5

Btree Operation

Append-Only



Meta Page

Pgno: 0
Misc...
Root : EMPTY

Data Page

Pgno: 1
Misc...
offset: 4000
1,foo

Meta Page

Pgno: 2
Misc...
Root : 1

Data Page

Pgno: 3
Misc...
offset: 4000
offset: 3000
2,bar
1,foo

Meta Page

Pgno: 4
Misc...
Root : 3

Data Page

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

Meta Page

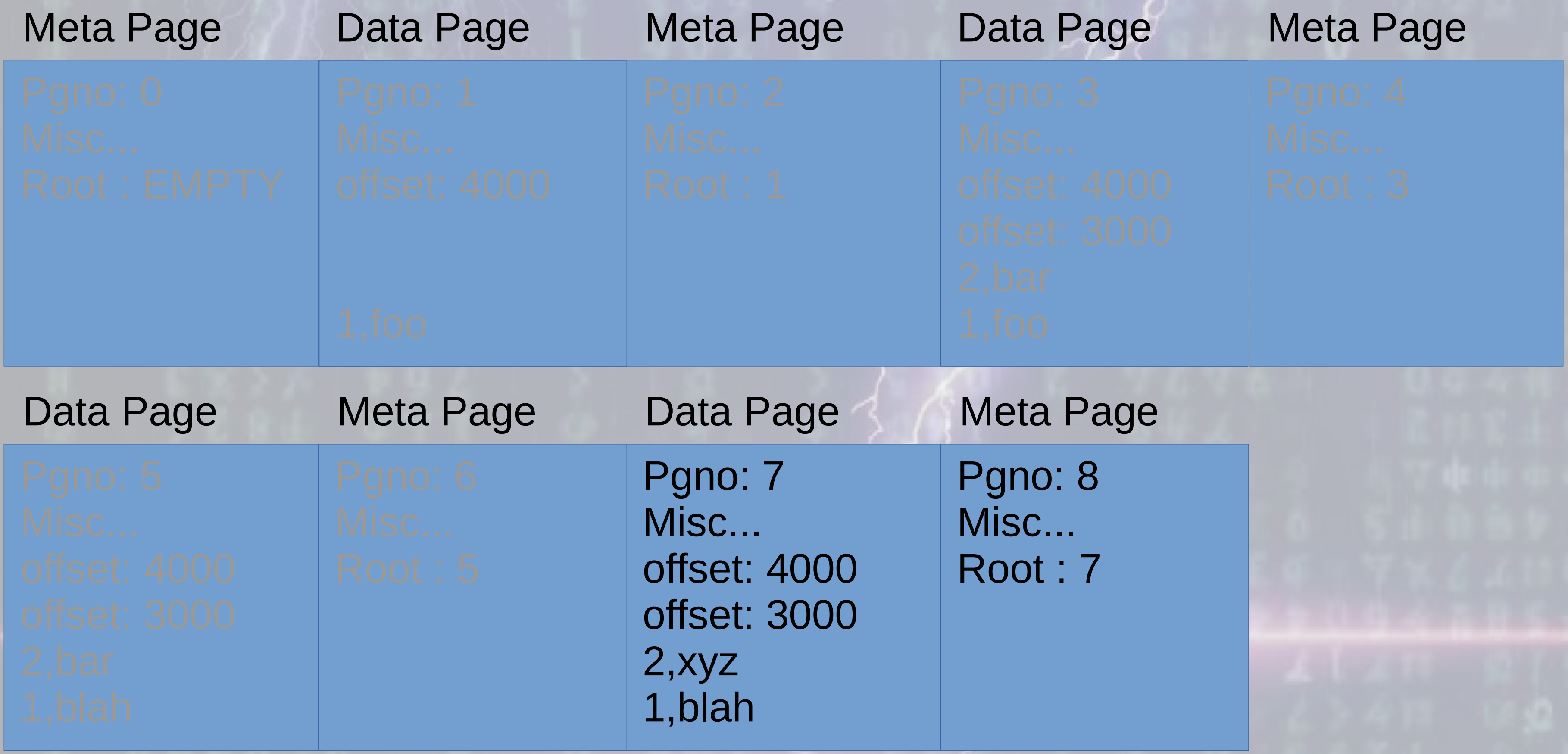
Pgno: 6
Misc...
Root : 5

Data Page

Pgno: 7
Misc...
offset: 4000
offset: 3000
2,xyz
1,blah

Btree Operation

Append-Only



Meta Page

Pgno: 0
Misc...
Root : EMPTY

Data Page

Pgno: 1
Misc...
offset: 4000
1,foo

Meta Page

Pgno: 2
Misc...
Root : 1

Data Page

Pgno: 3
Misc...
offset: 4000
offset: 3000
2,bar
1,foo

Meta Page

Pgno: 4
Misc...
Root : 3

Data Page

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

Meta Page

Pgno: 6
Misc...
Root : 5

Data Page

Pgno: 7
Misc...
offset: 4000
offset: 3000
2,xyz
1,blah

Meta Page

Pgno: 8
Misc...
Root : 7

Btree Operation

Append-Only disk usage is very inefficient

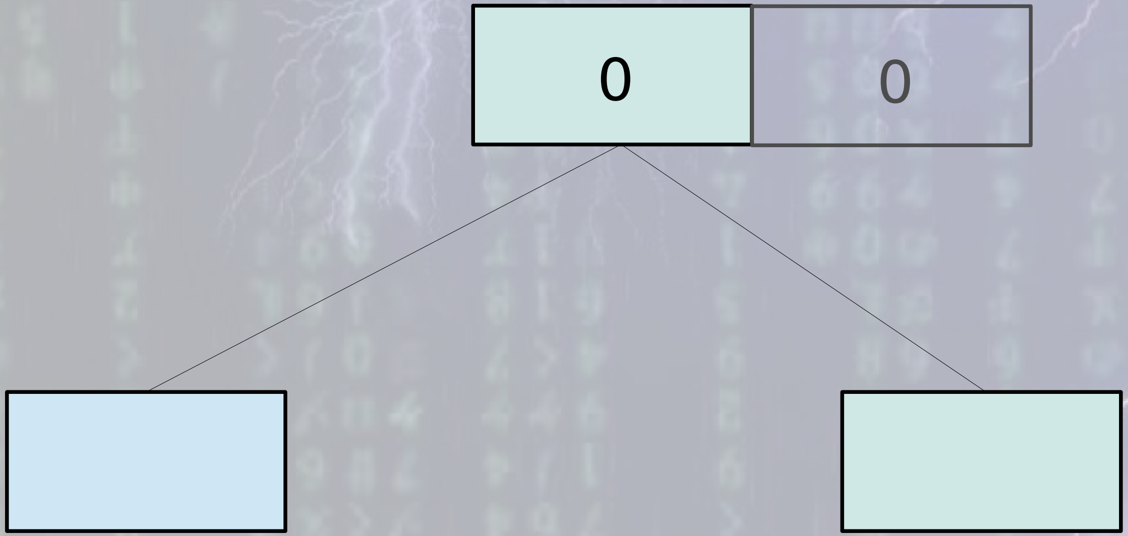
- Disk space usage grows without bound
- 99+% of the space will be occupied by old versions of the data
- The old versions are usually not interesting
- Reclaiming the old space requires a very expensive compaction phase
- New updates must be throttled until compaction completes

Btree Operation

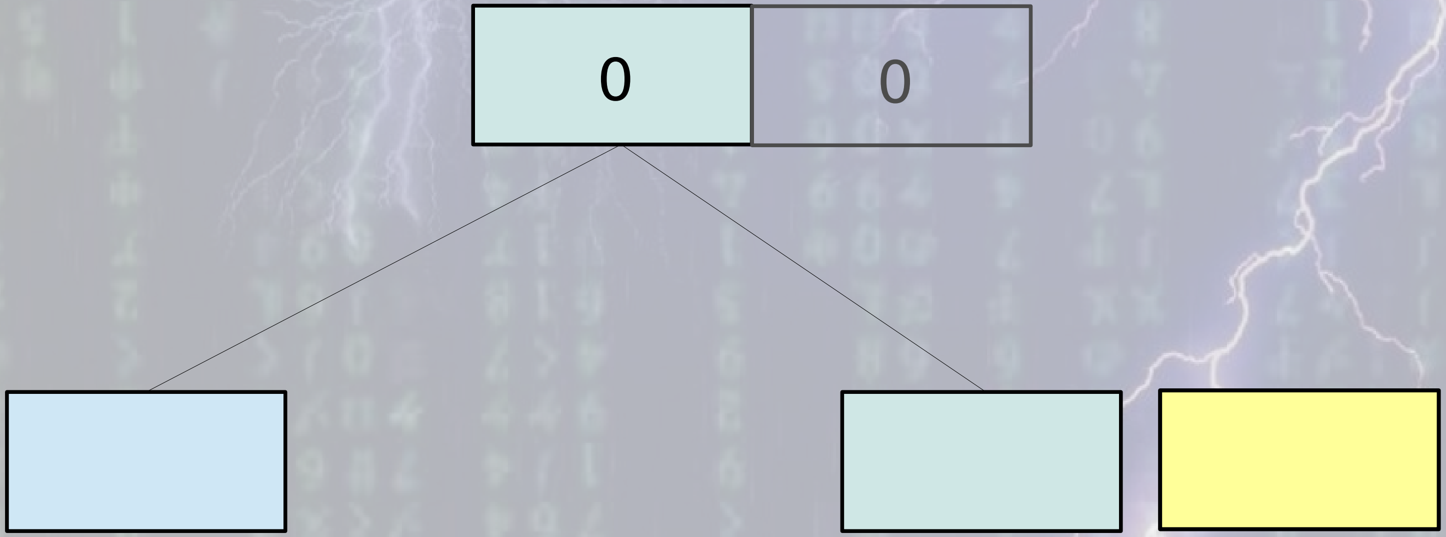
The LMDB Approach

- 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"

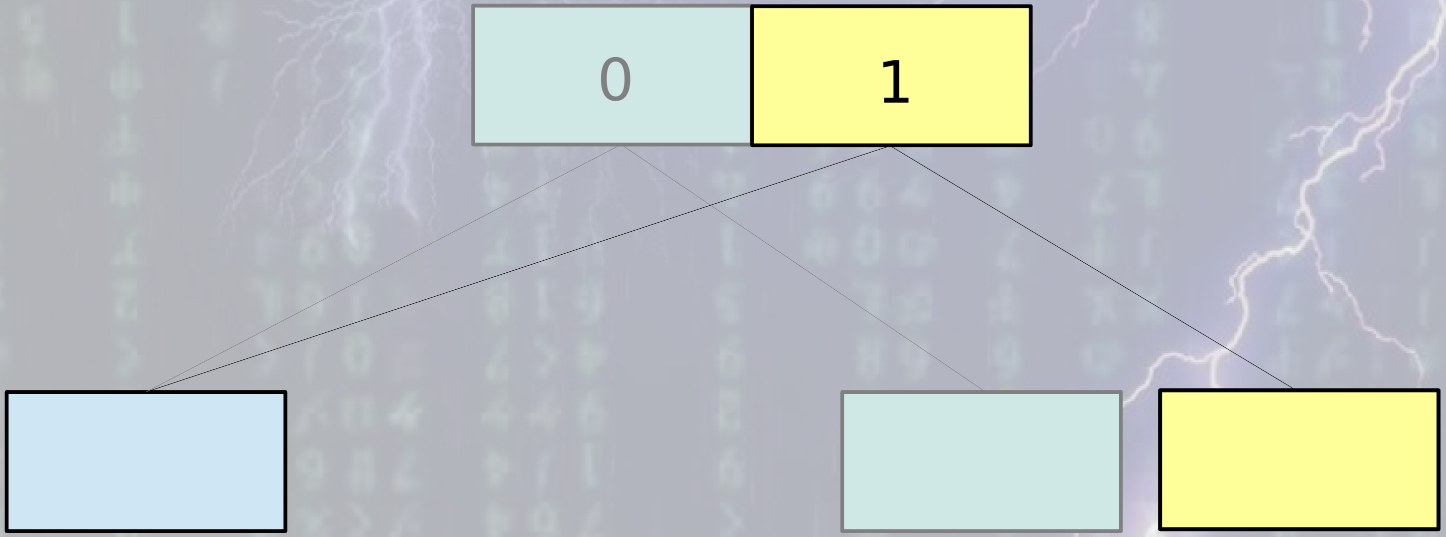
Btree Operation



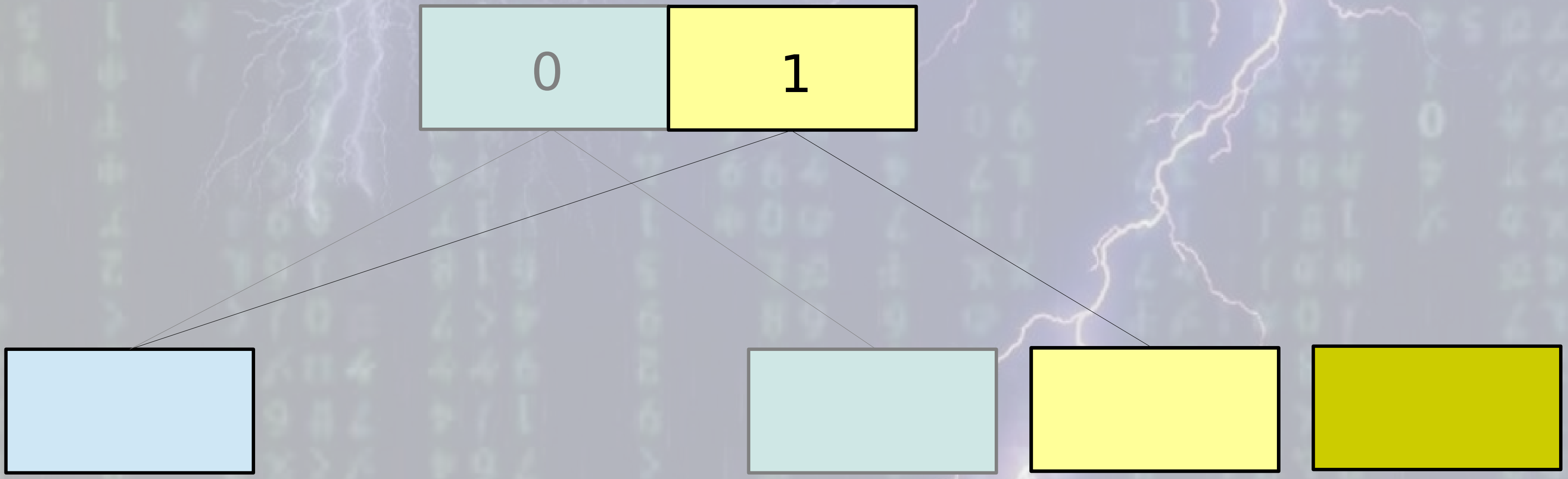
Btree Operation



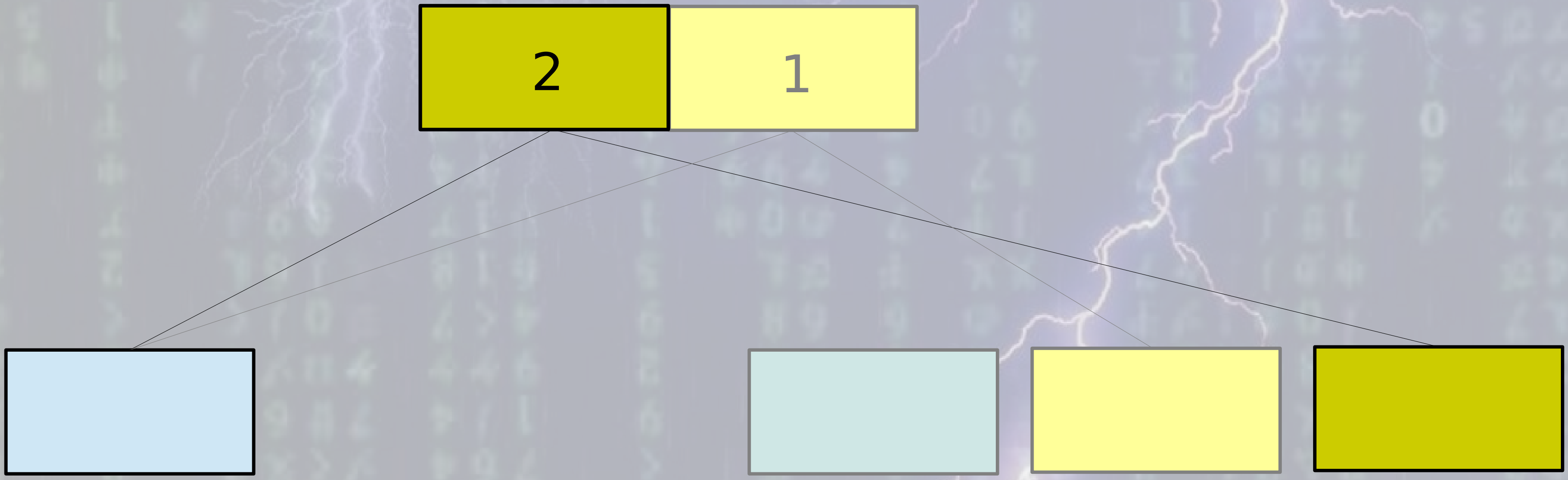
Btree Operation



Btree Operation

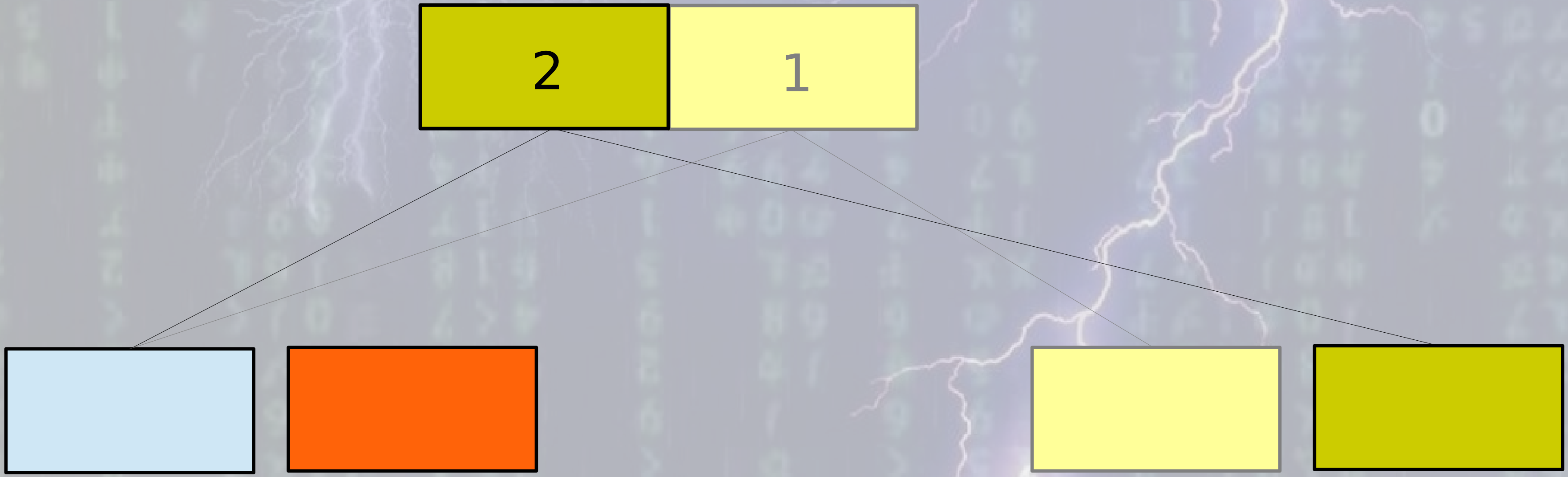


Btree Operation

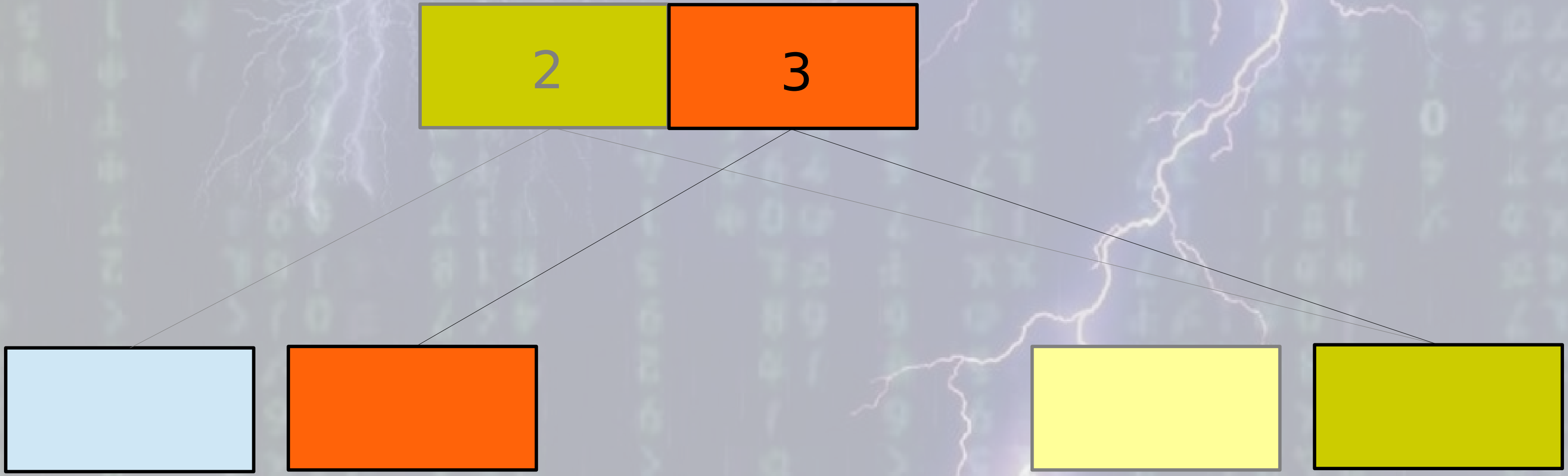


After this step the old blue page is no longer referenced by anything else in the database, so it can be reclaimed

Btree Operation



Btree Operation



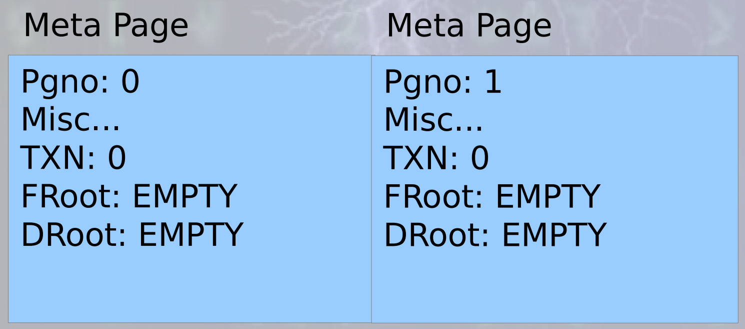
After this step the old yellow page is no longer referenced by anything else in the database, so it can also be reclaimed

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

Pgno: 0
Misc...
TXN: 0
FRoot: EMPTY
DRoot: EMPTY

Meta Page

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

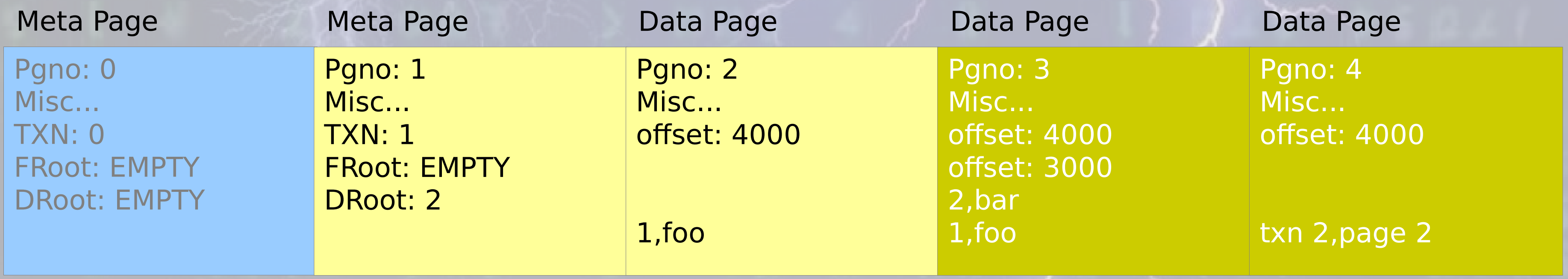
Free Space Management

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	Pgno: 1	Pgno: 2	Pgno: 3	Pgno: 4
Misc...	Misc...	Misc...	Misc...	Misc...
TXN: 2	TXN: 1	offset: 4000	offset: 4000	offset: 4000
FRoot: 4	FRoot: EMPTY		offset: 3000	
DRoot: 3	DRoot: 2		2,bar	
		1,foo	1,foo	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

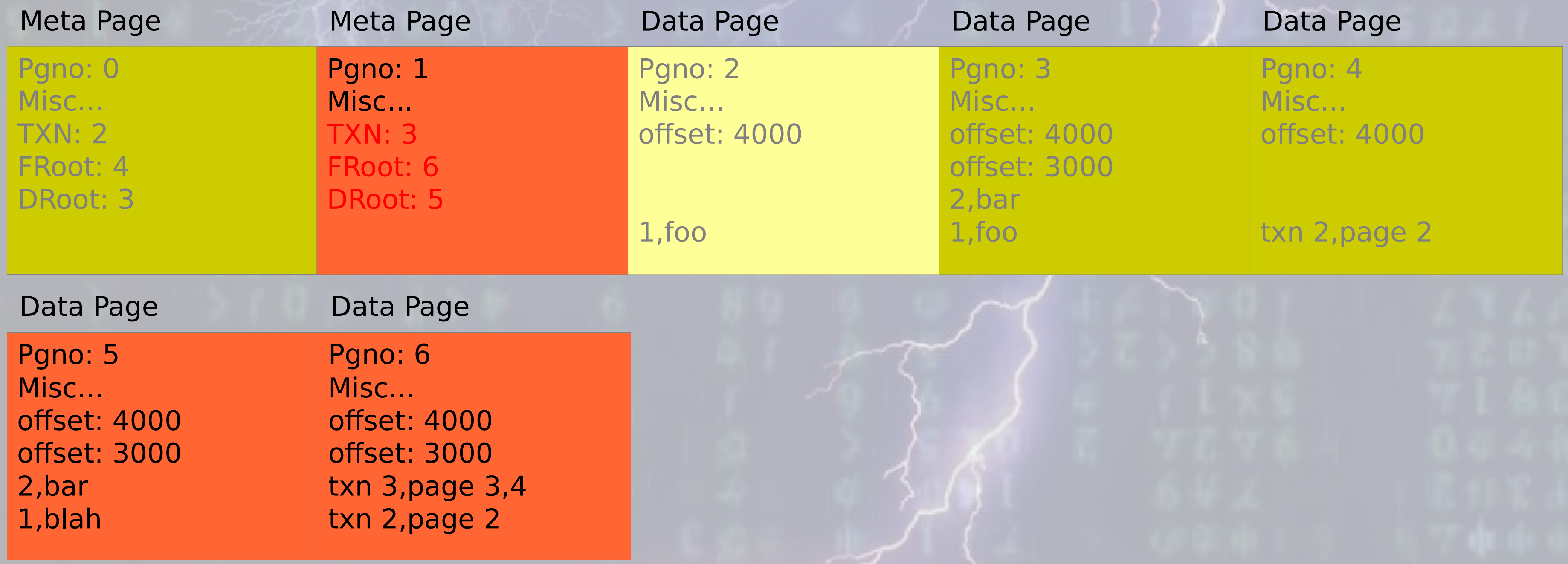
Data Page
Pgno: 5 Misc... offset: 4000 offset: 3000 2,bar 1,blah

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
Pgno: 0
Misc...
TXN: 2
FRoot: 4
DRoot: 3

Meta Page
Pgno: 1
Misc...
TXN: 3
FRoot: 6
DRoot: 5

Data Page
Pgno: 2
Misc...
offset: 4000
1,foo

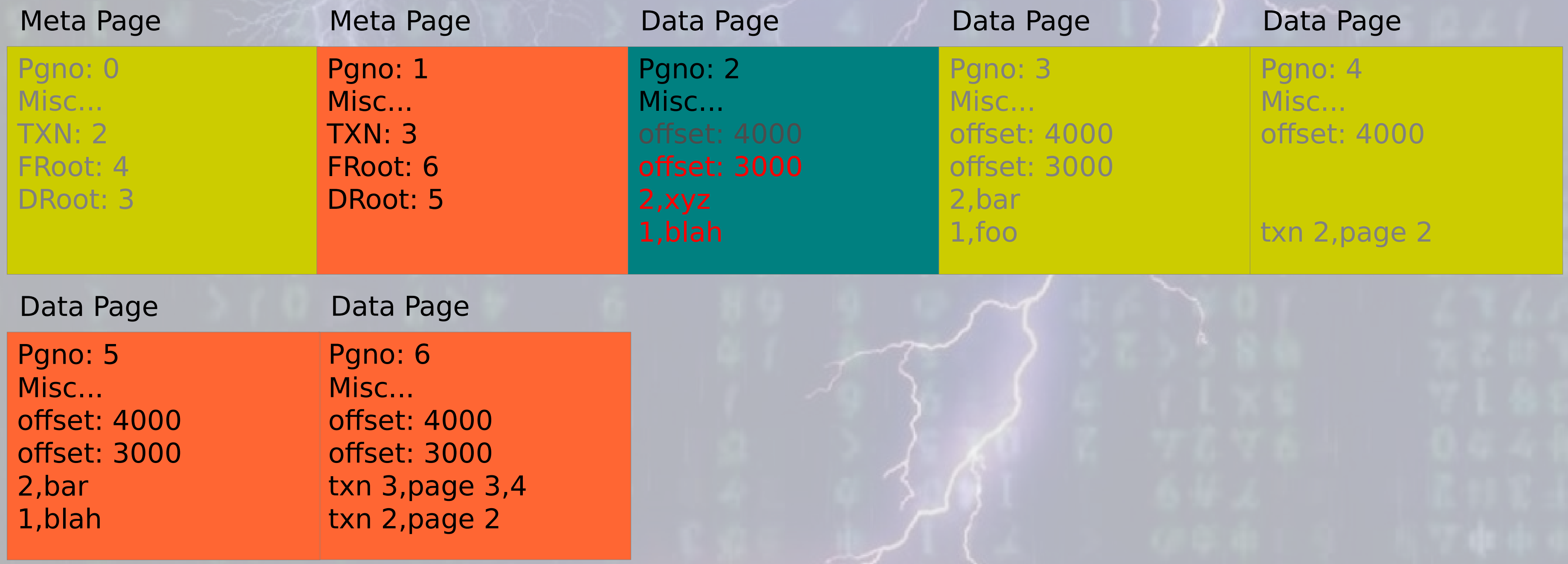
Data Page
Pgno: 3
Misc...
offset: 4000
offset: 3000
2,bar
1,foo

Data Page
Pgno: 4
Misc...
offset: 4000
txn 2,page 2

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

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

Free Space Management



Meta Page

Pgno: 0
Misc...
TXN: 2
FRoot: 4
DRoot: 3

Meta Page

Pgno: 1
Misc...
TXN: 3
FRoot: 6
DRoot: 5

Data Page

Pgno: 2
Misc...
offset: 4000
offset: 3000
2,xyz
1,blah

Data Page

Pgno: 3
Misc...
offset: 4000
offset: 3000
2,bar
1,foo

Data Page

Pgno: 4
Misc...
offset: 4000
txn 2,page 2

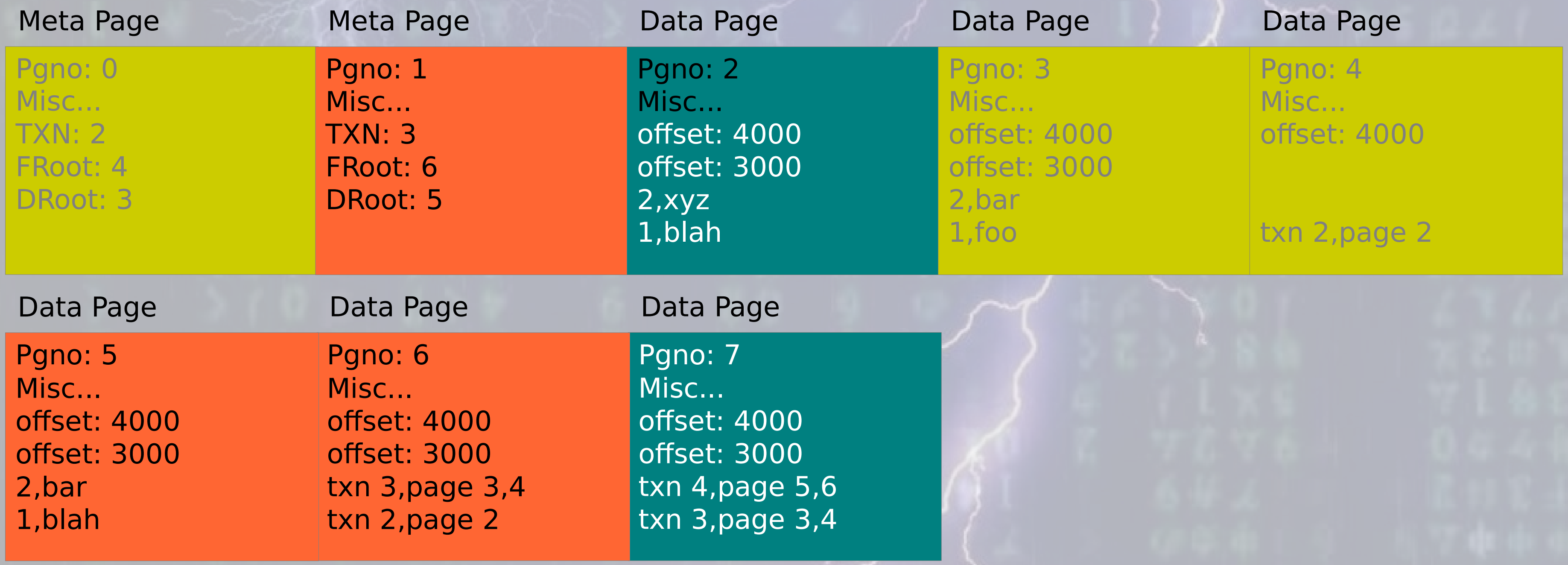
Data Page

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

Data Page

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	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

(6) 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

(7) Results

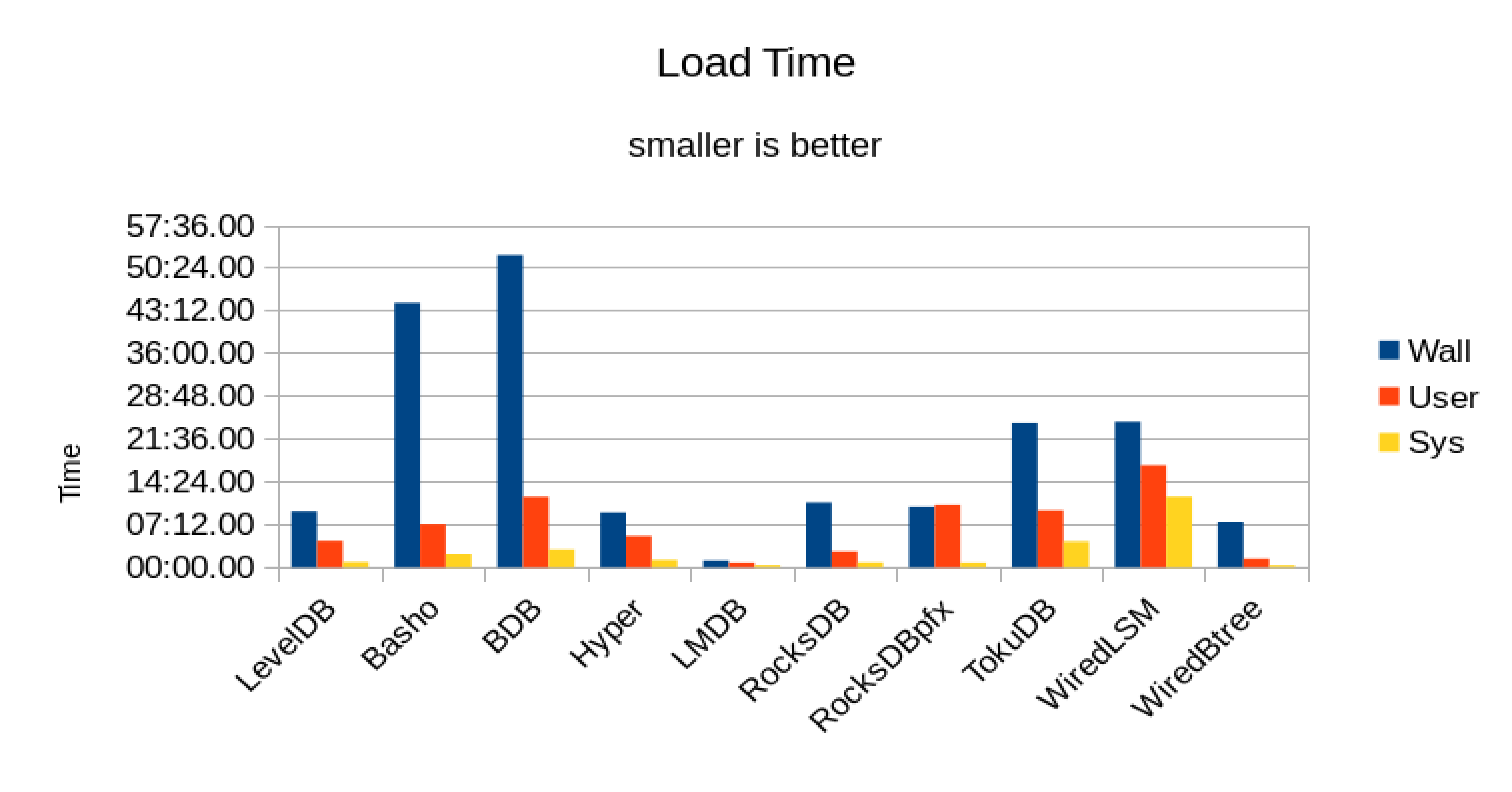
- Support for LMDB is available in scores of open source projects and all major Linux and BSD distros
- In OpenLDAP slapd
  - LMDB reads are 5-20x faster than BDB
  - Writes are 2-5x faster than BDB
  - Consumes 1/4 as much RAM as BDB
- In MemcacheDB
  - LMDB reads are 2-200x faster than BDB
  - Writes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

Results

- LMDB has been tested exhaustively by multiple parties
  - Symas has tested on all major filesystems: btrfs, ext2, ext3, ext4, jfs, ntfs, reiserfs, xfs, zfs
  - ext3, ext4, jfs, reiserfs, xfs also tested with external journalling
  - Testing on physical servers, VMs, HDDs, SSDs, PCIe NVM
  - Testing crash reliability as well as performance and efficiency - LMDB is proven corruption-proof in real world conditions

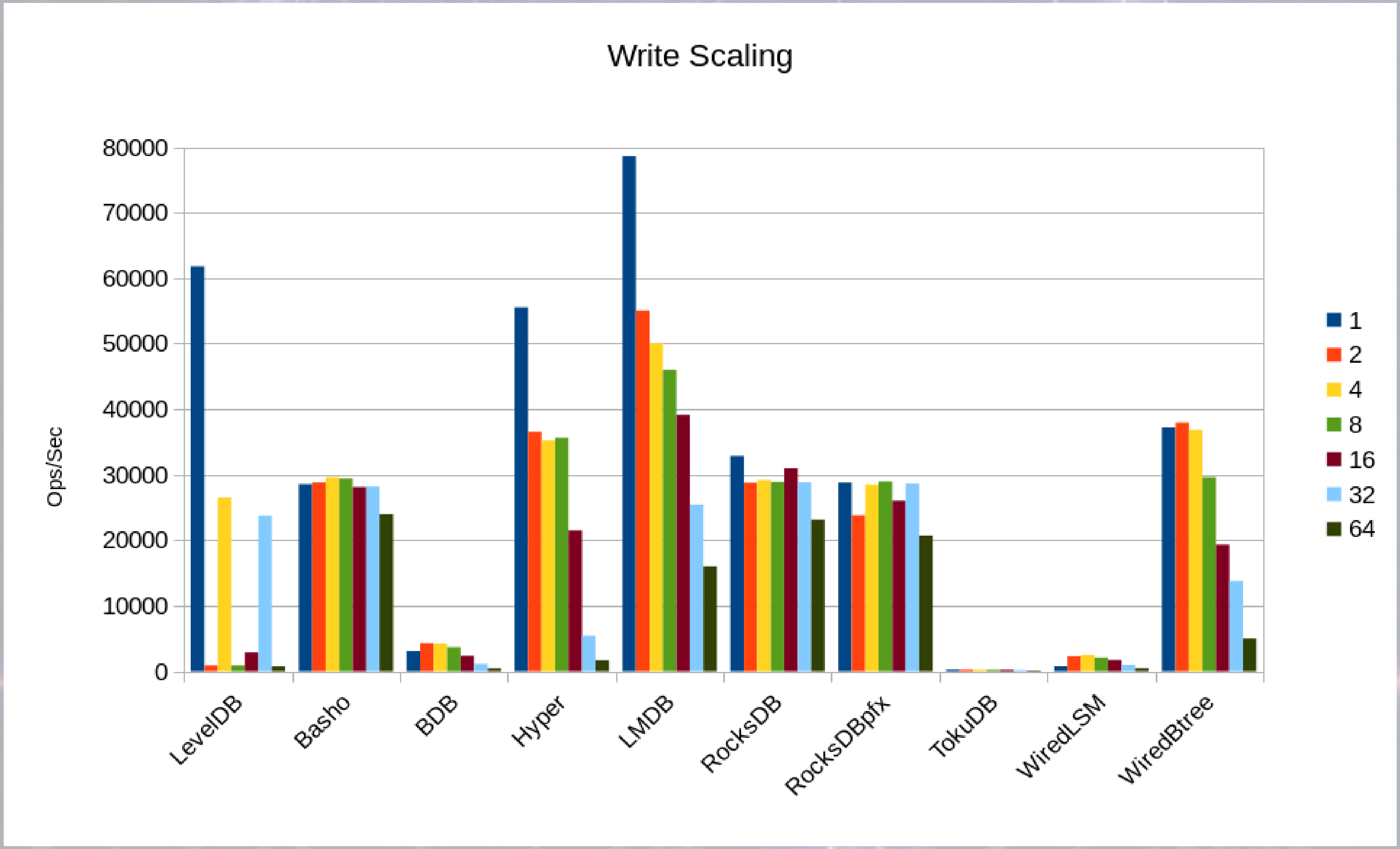
Results

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



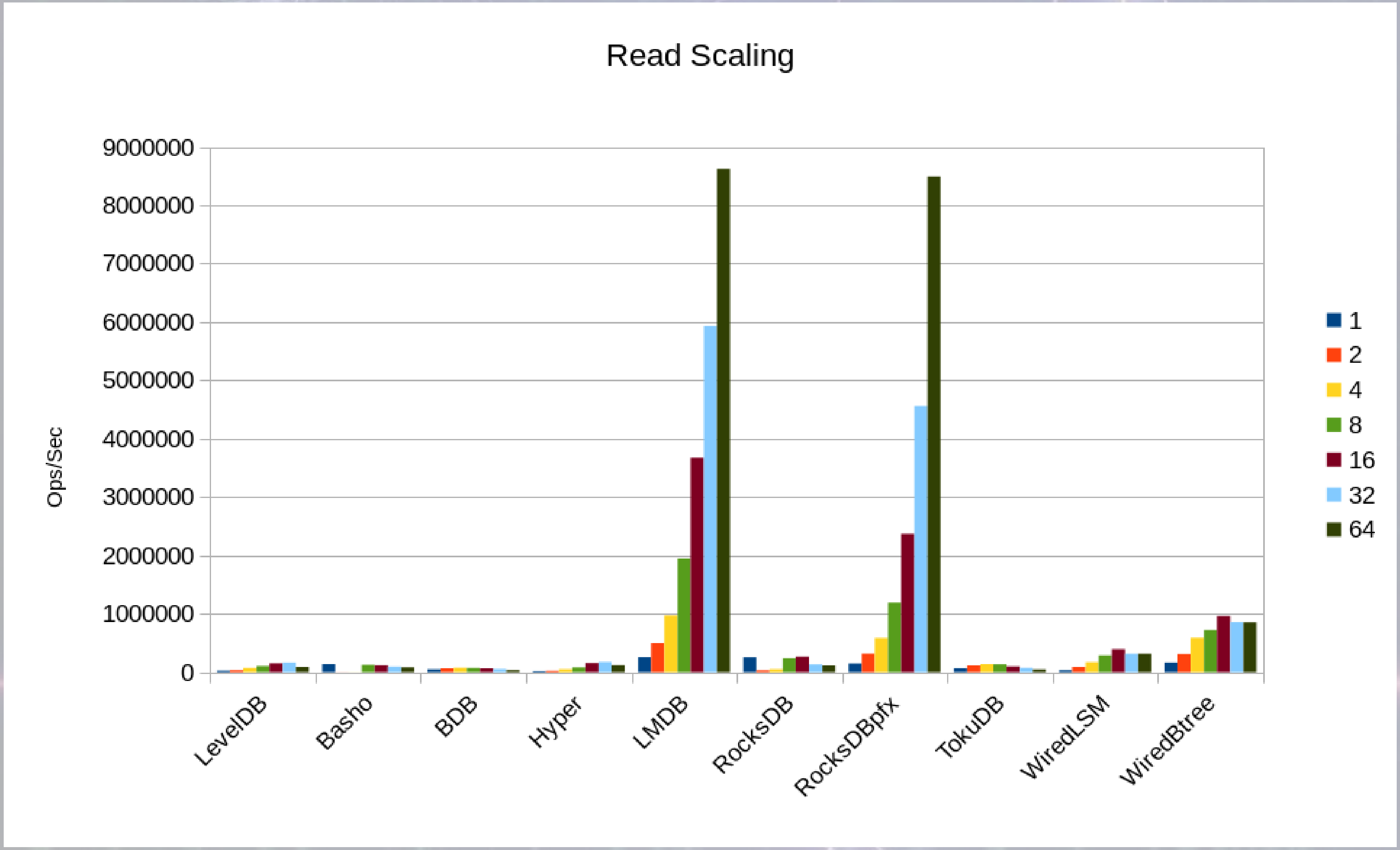
Results

- Scaling up to 64 CPUs, 64 concurrent readers



Results

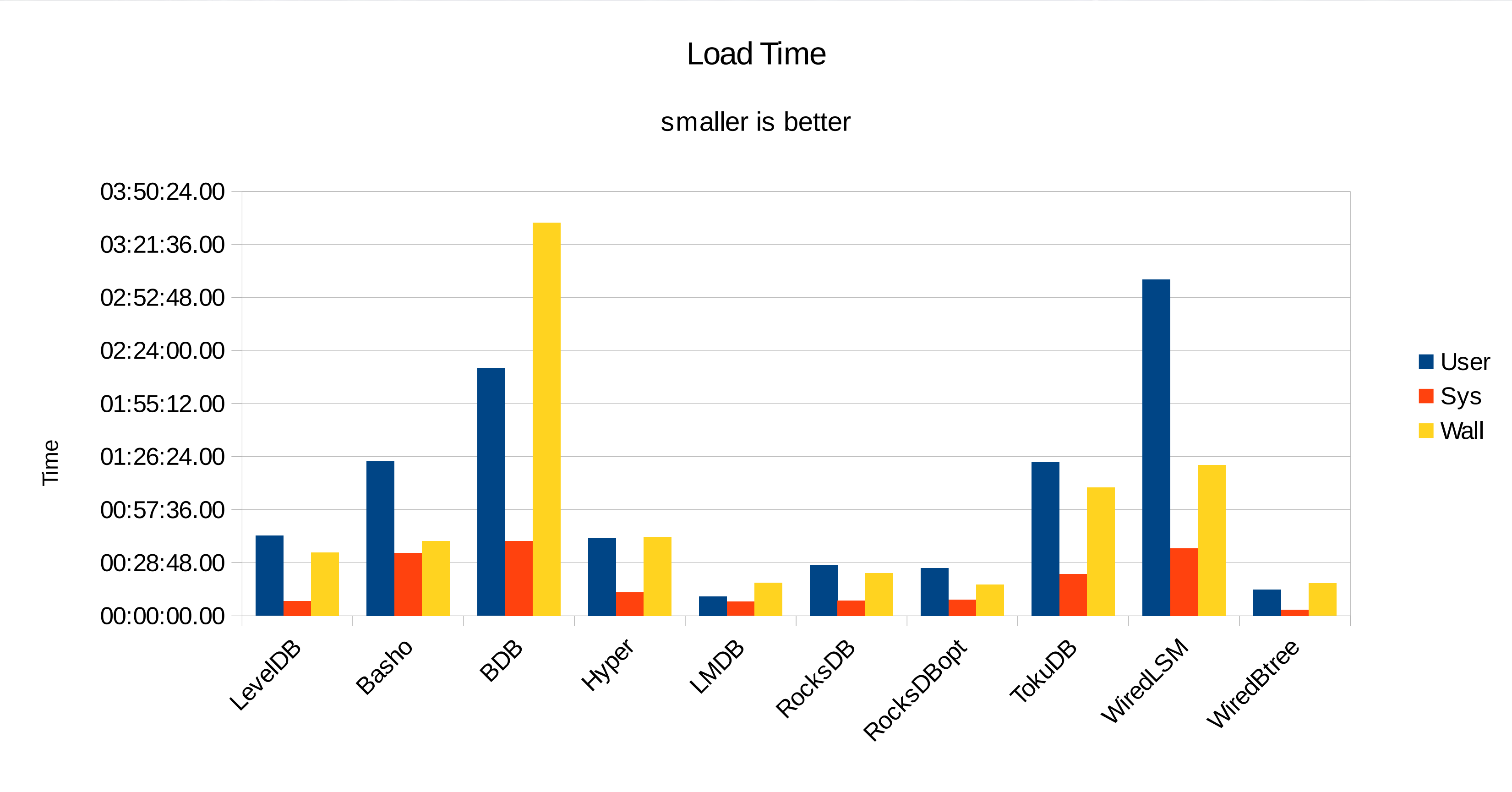
- Scaling up to 64 CPUs, 64 concurrent readers



HTTP://SYMAS.COM

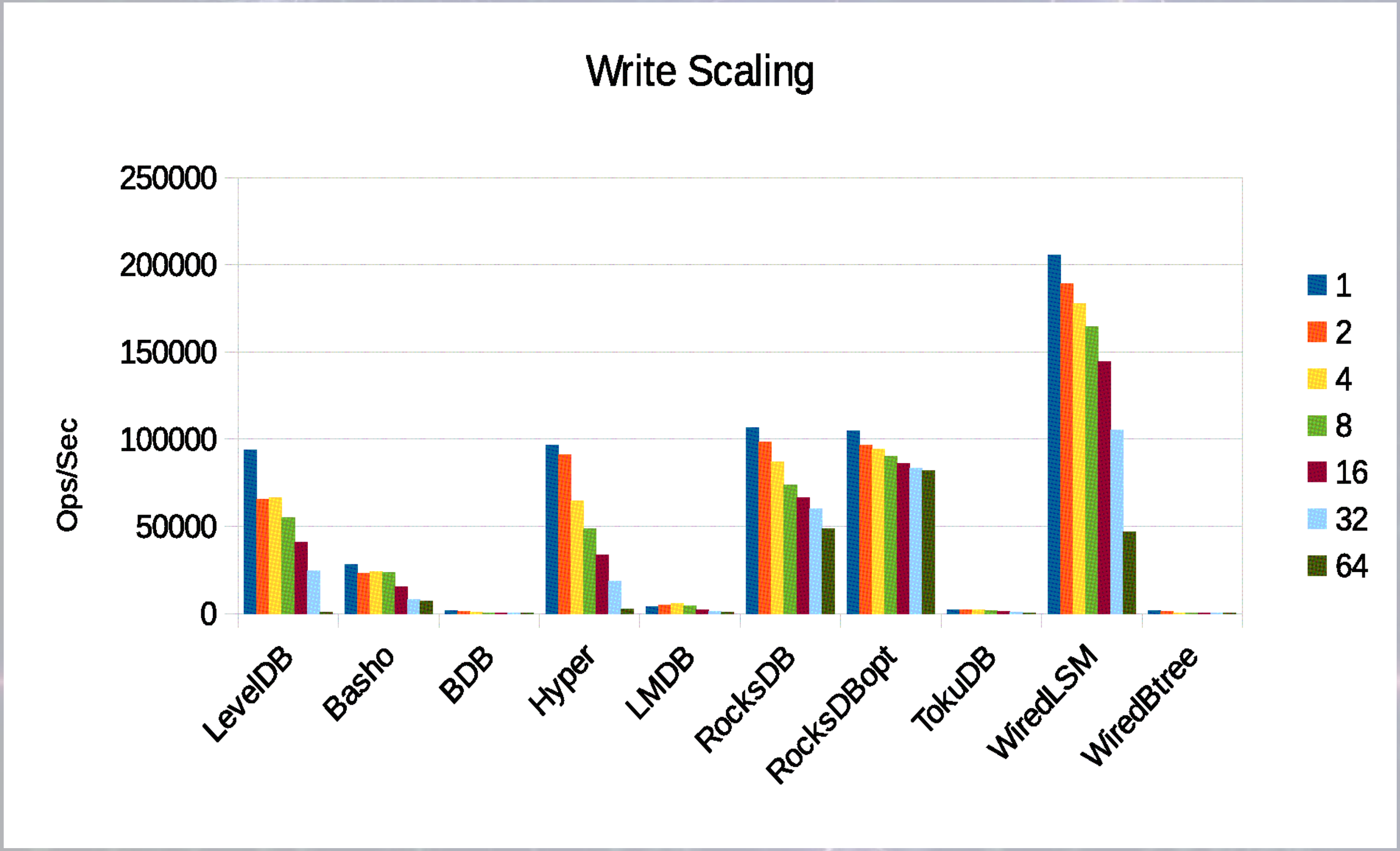
Results

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



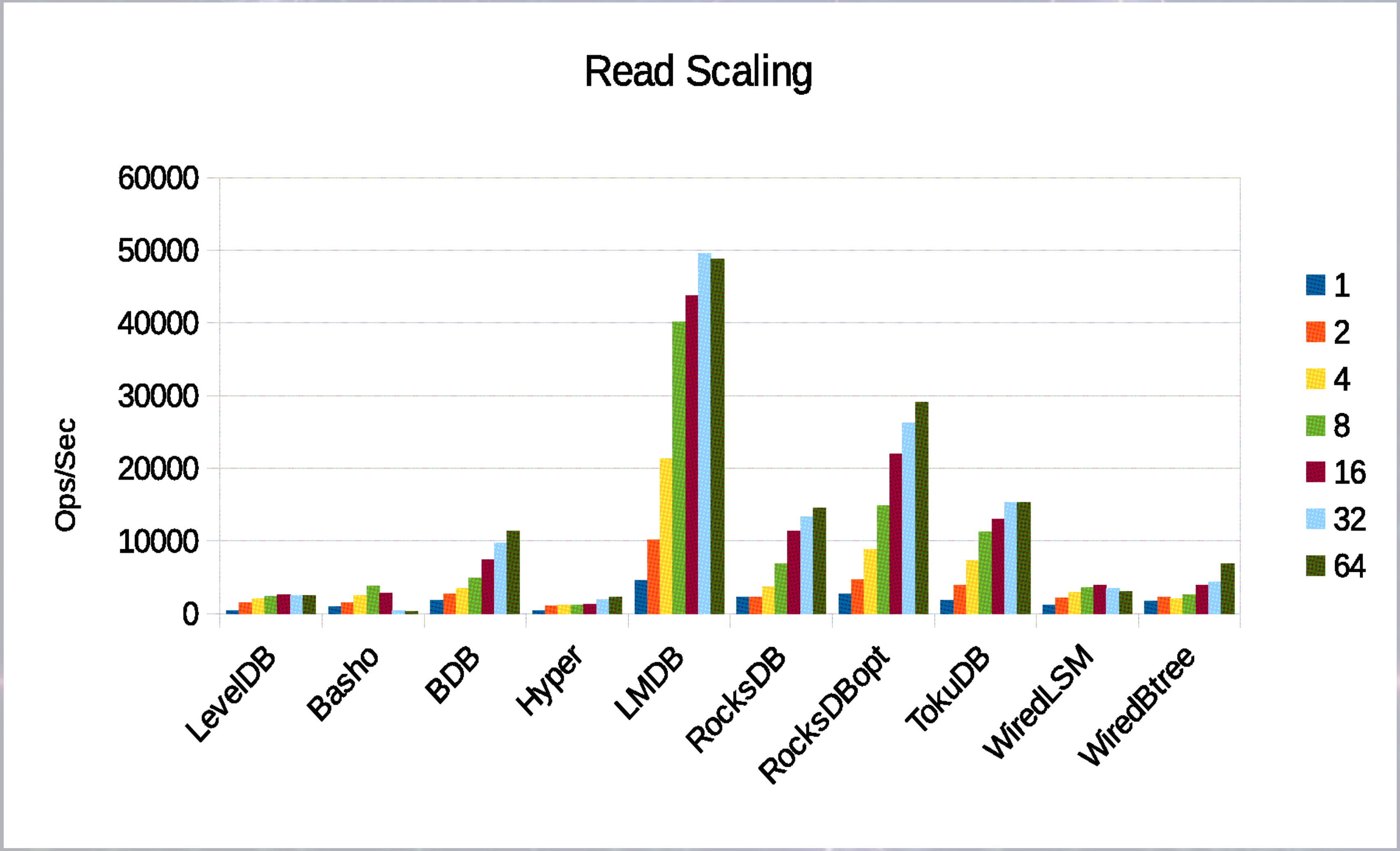
Results

- VM with 16 CPU cores, 64 concurrent readers



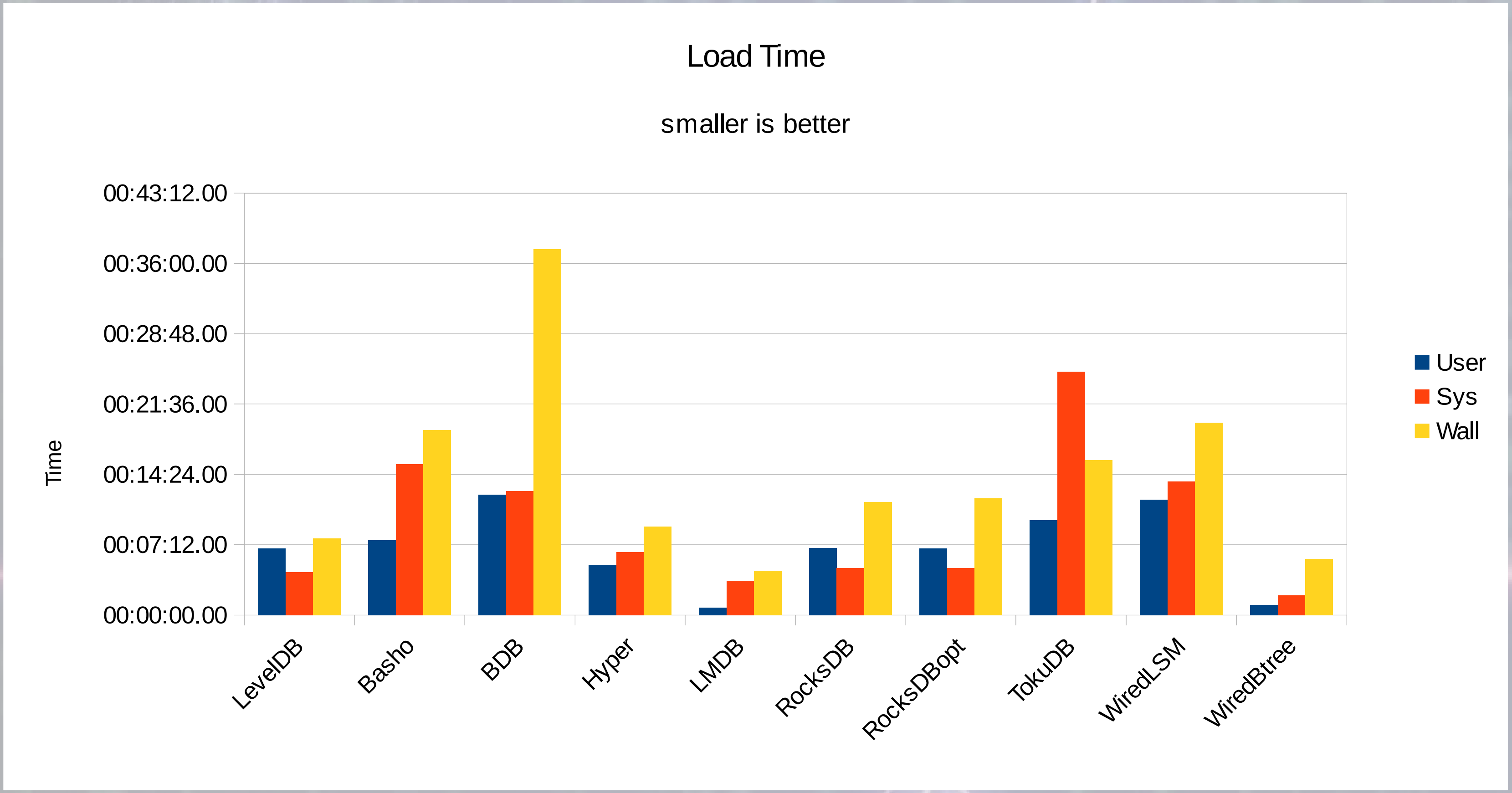
Results

- VM with 16 CPU cores, 64 concurrent readers



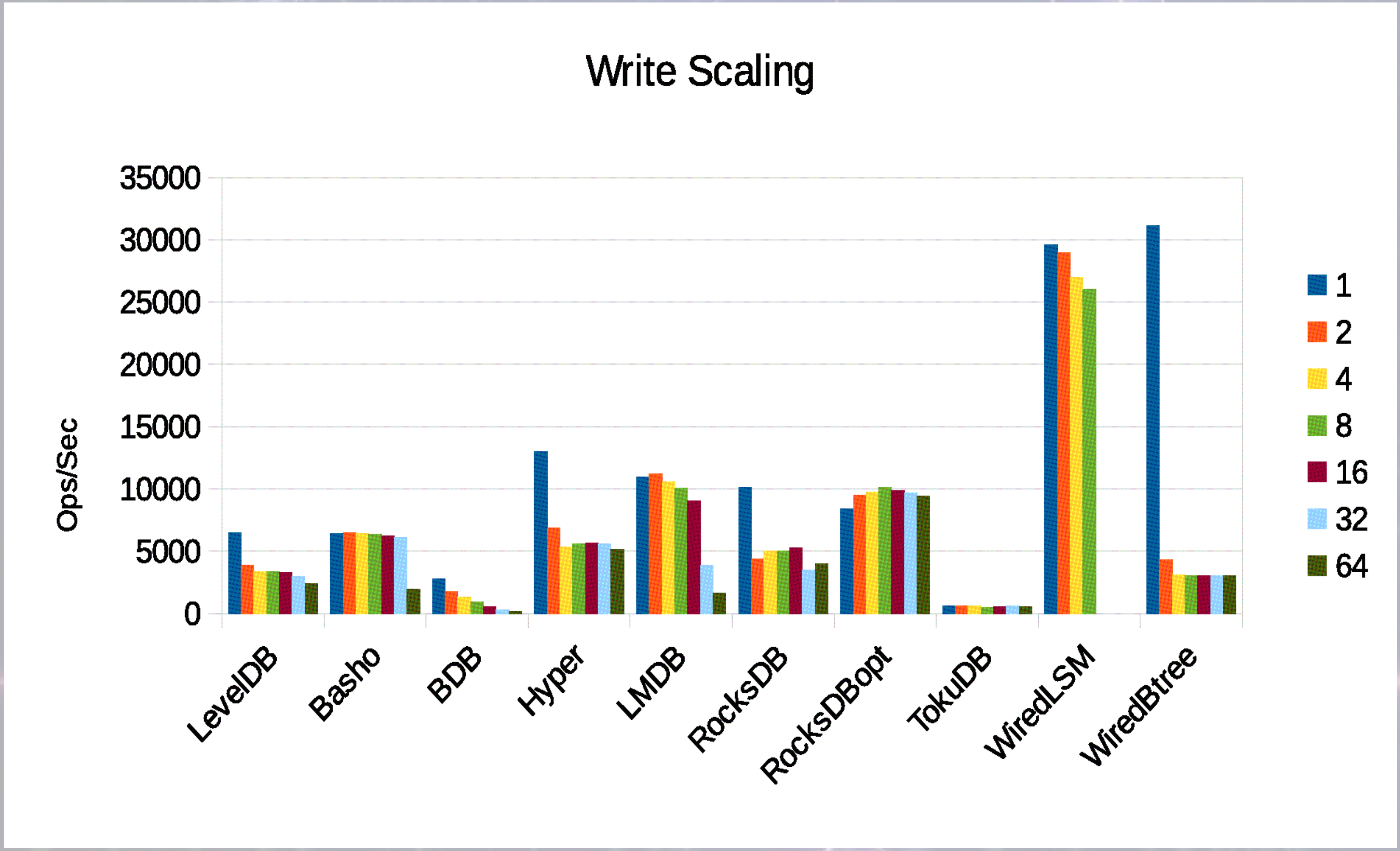
Results

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



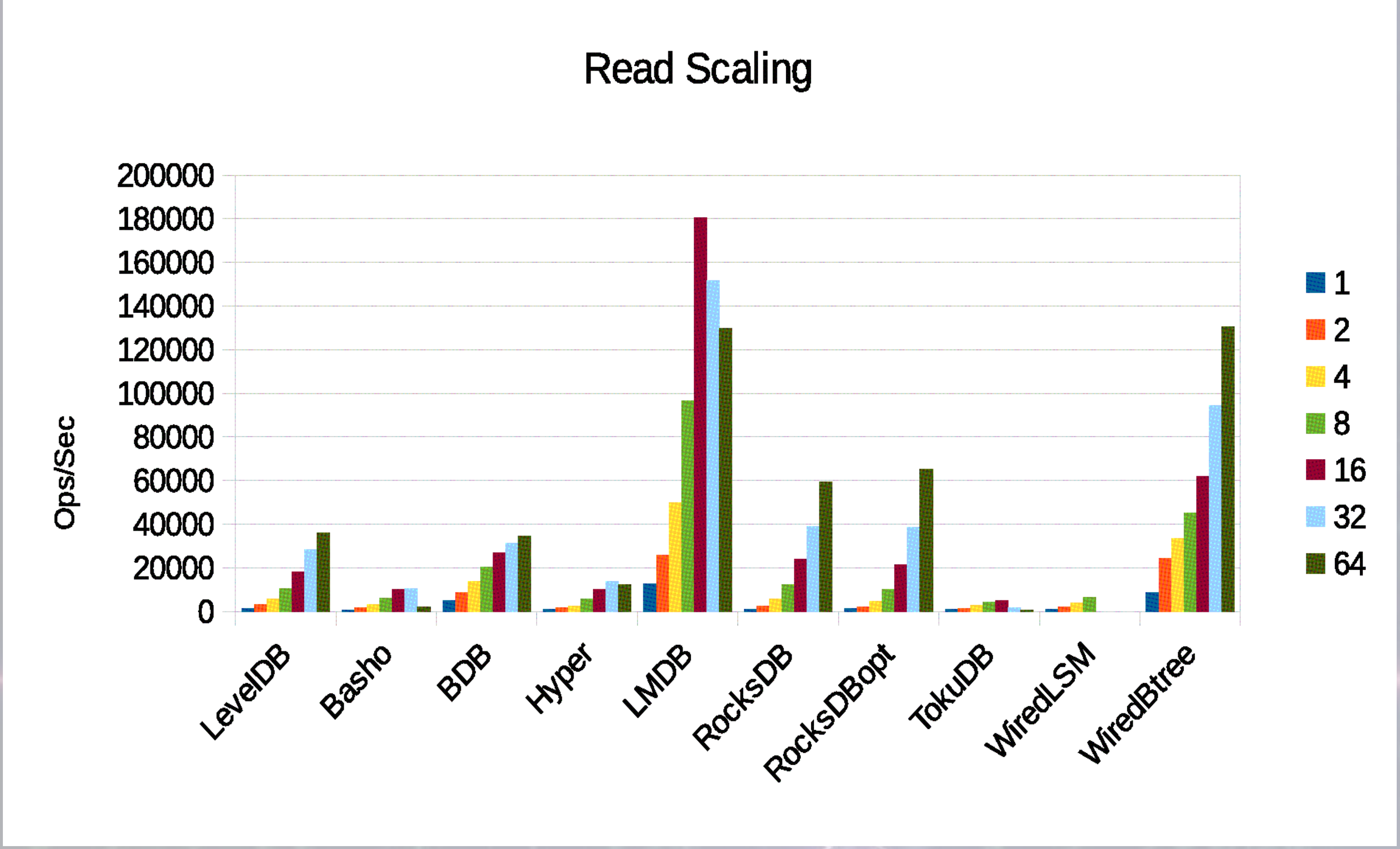
Results

- 16 CPU cores, 64 concurrent readers



Results

- 16 CPU cores, 64 concurrent readers

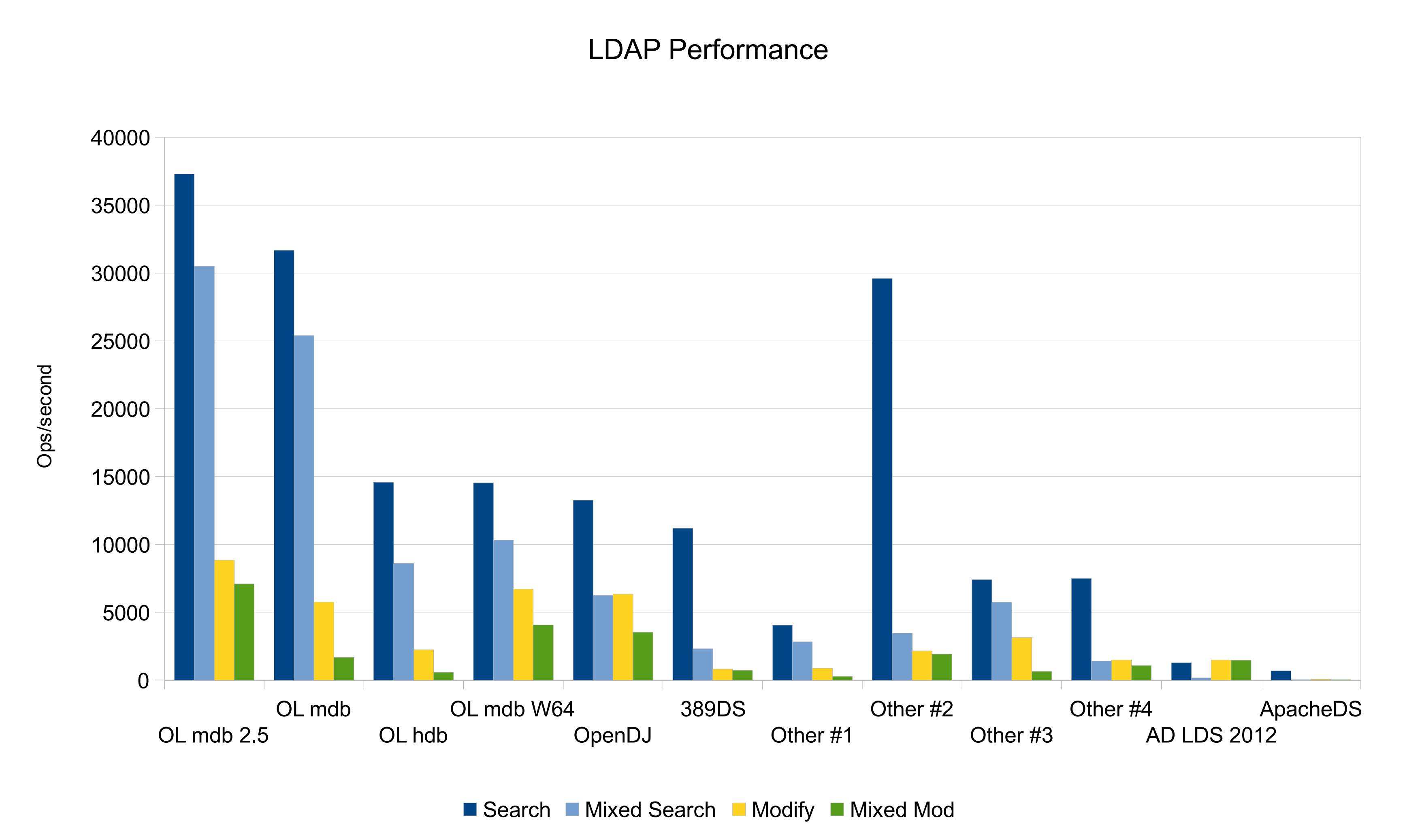


Results

- 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

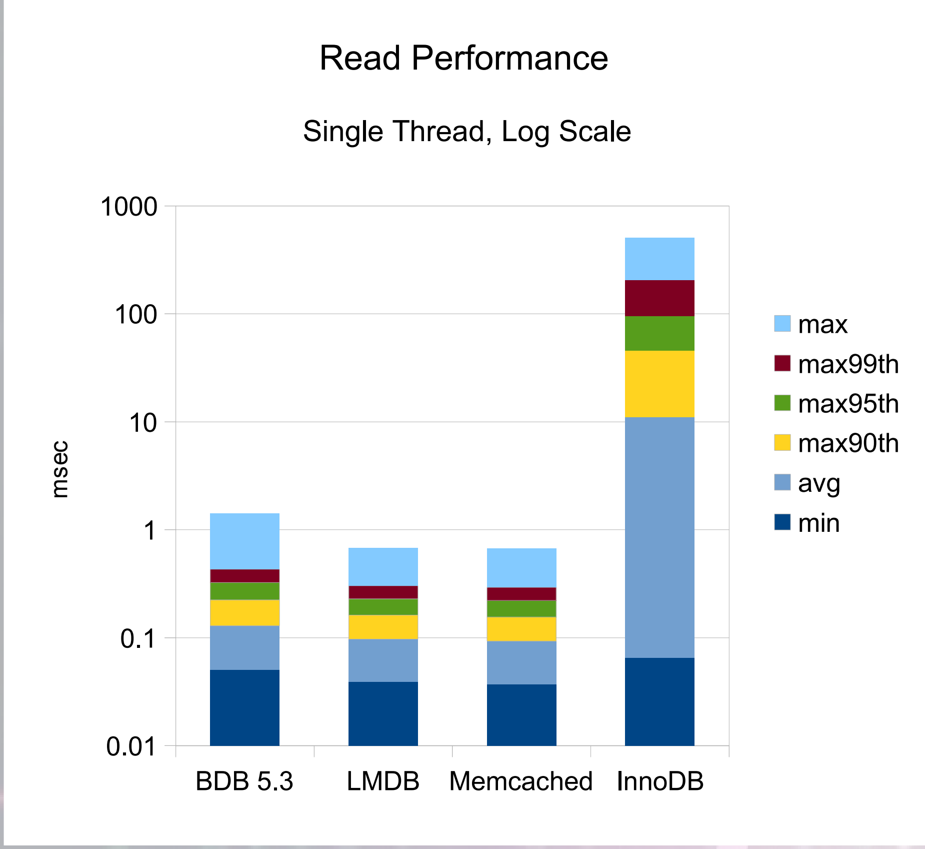
Results

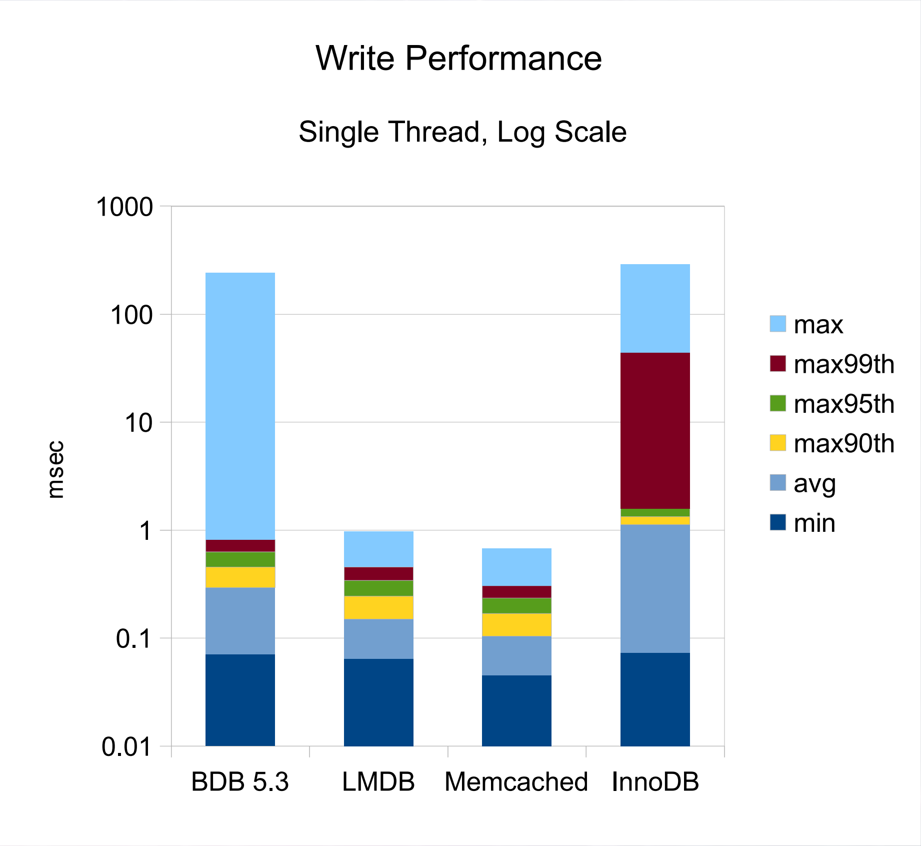
- LDAP Benchmarks



Results

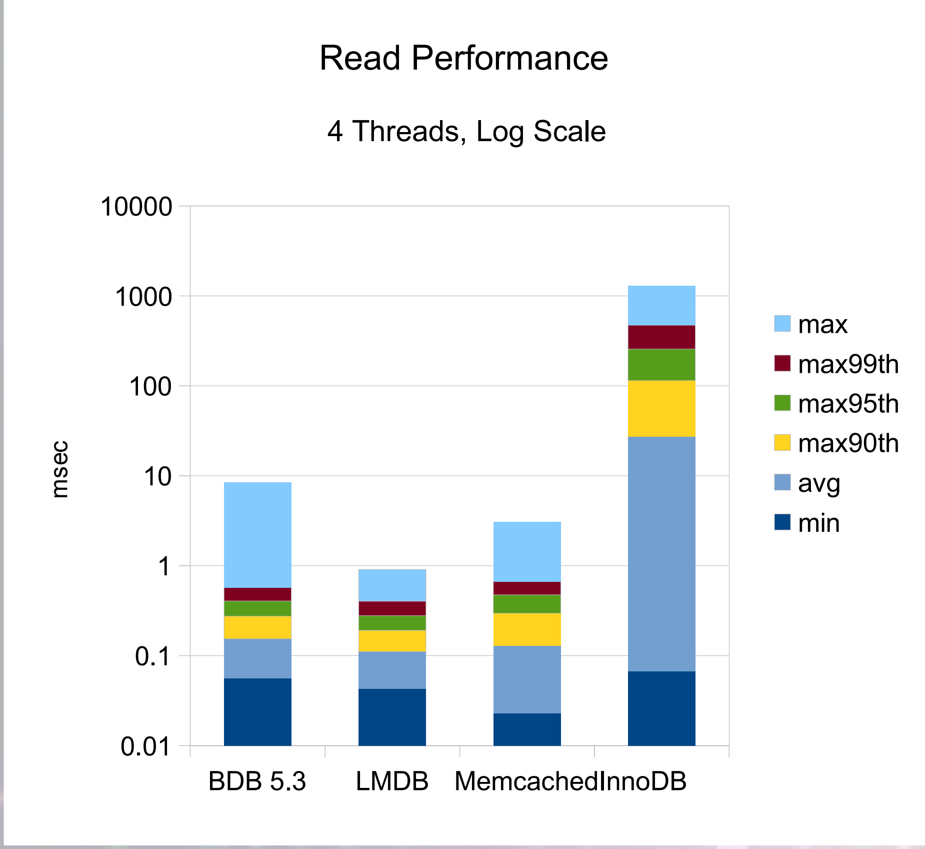
- Memcached

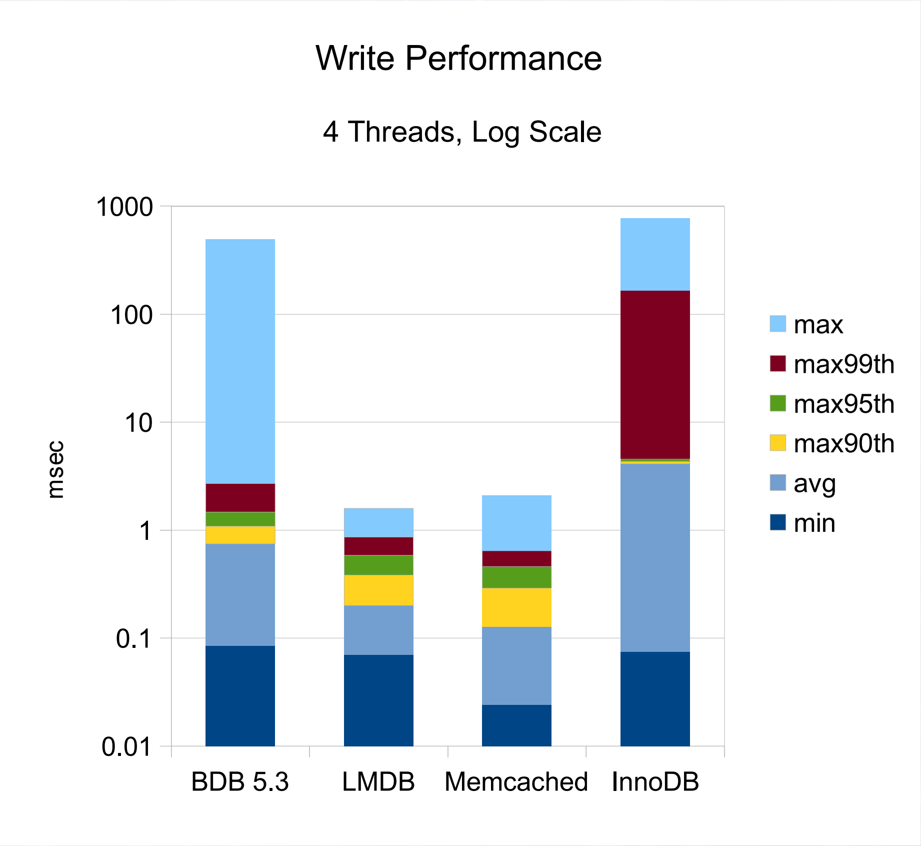




Results

- Memcached





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

Questions?