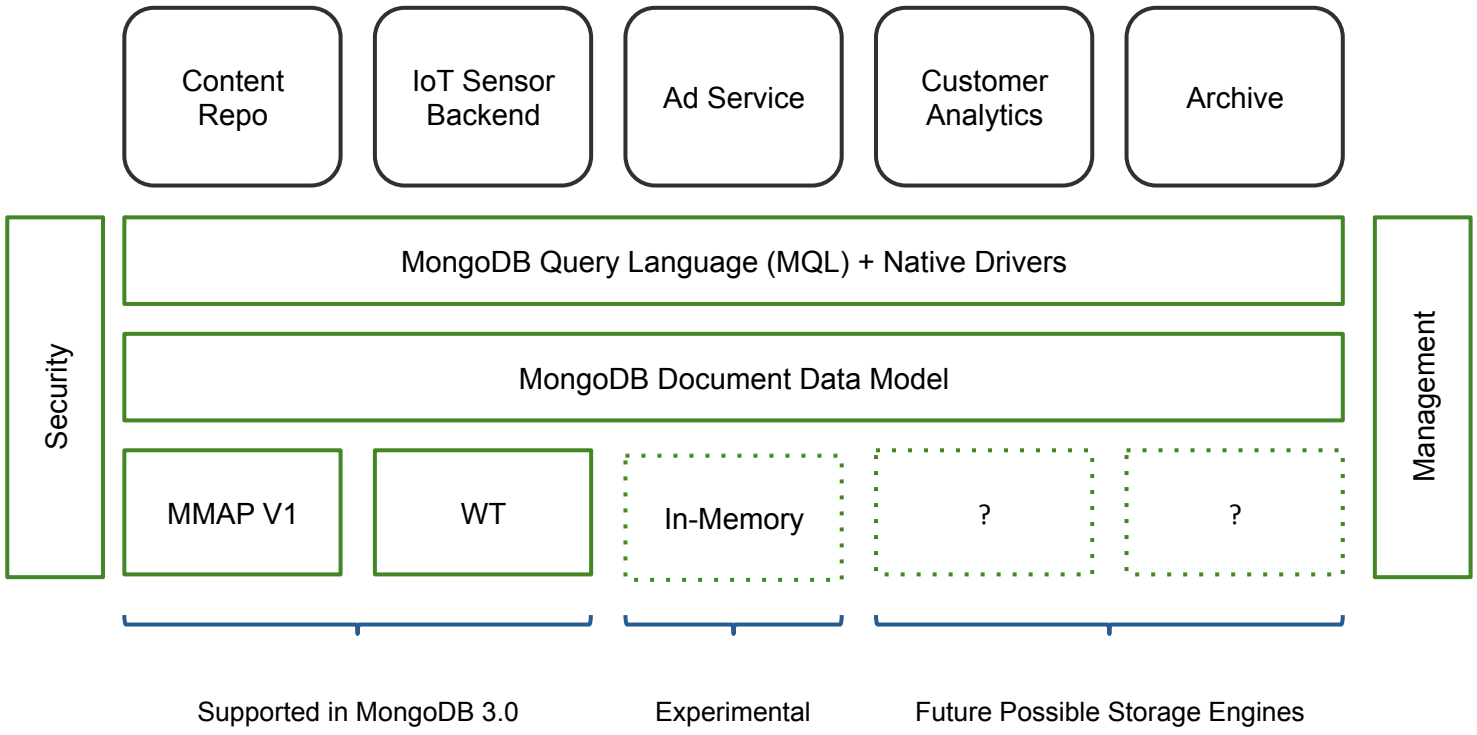
Storage Engines

Outline

- Introduction
- MMAPv1
- WiredTiger
- Benchmarks

Storage Engine Layer



At a glance

MMAPv1

- MongoDB >= 1.0
- OS manages memory
- No compression
- All collections and indexes within one set of db files
- Padding/PowerOf2Sizes, in-place updates
- Supports 32-bit and 64-bit

WiredTiger

- MongoDB >= 3.0
- Server manages memory
- Compression (snappy, zlib)
- One collection or index per file
- No padding, always reallocate for writes
- Supports 64-bit only

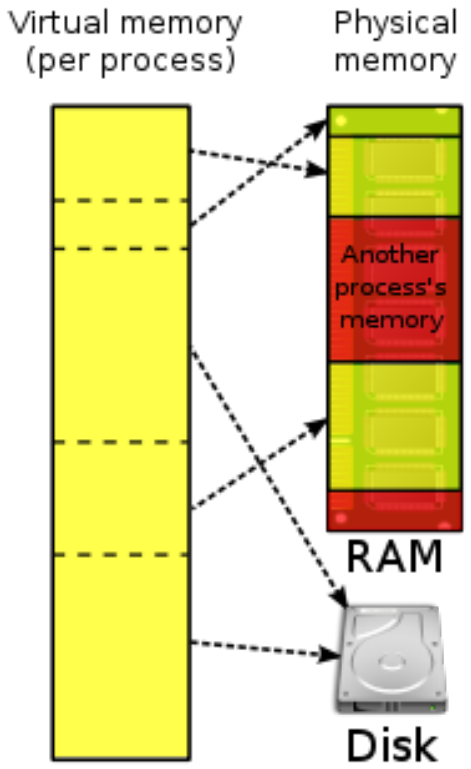
Locking

- Lock modes – compatibility matrix
  - https://github.com/mongodb/mongo/blob/r3.2.0/src/mongo/db/concurrency/lock_manager_defs.h#L46-L57

MMAPv1

MMAPv1 in a nutshell

- Memory map all data files to virtual memory
  - mmap in Linux
  - MapViewOfFile in Windows
- Let the OS handle everything



Pros and cons

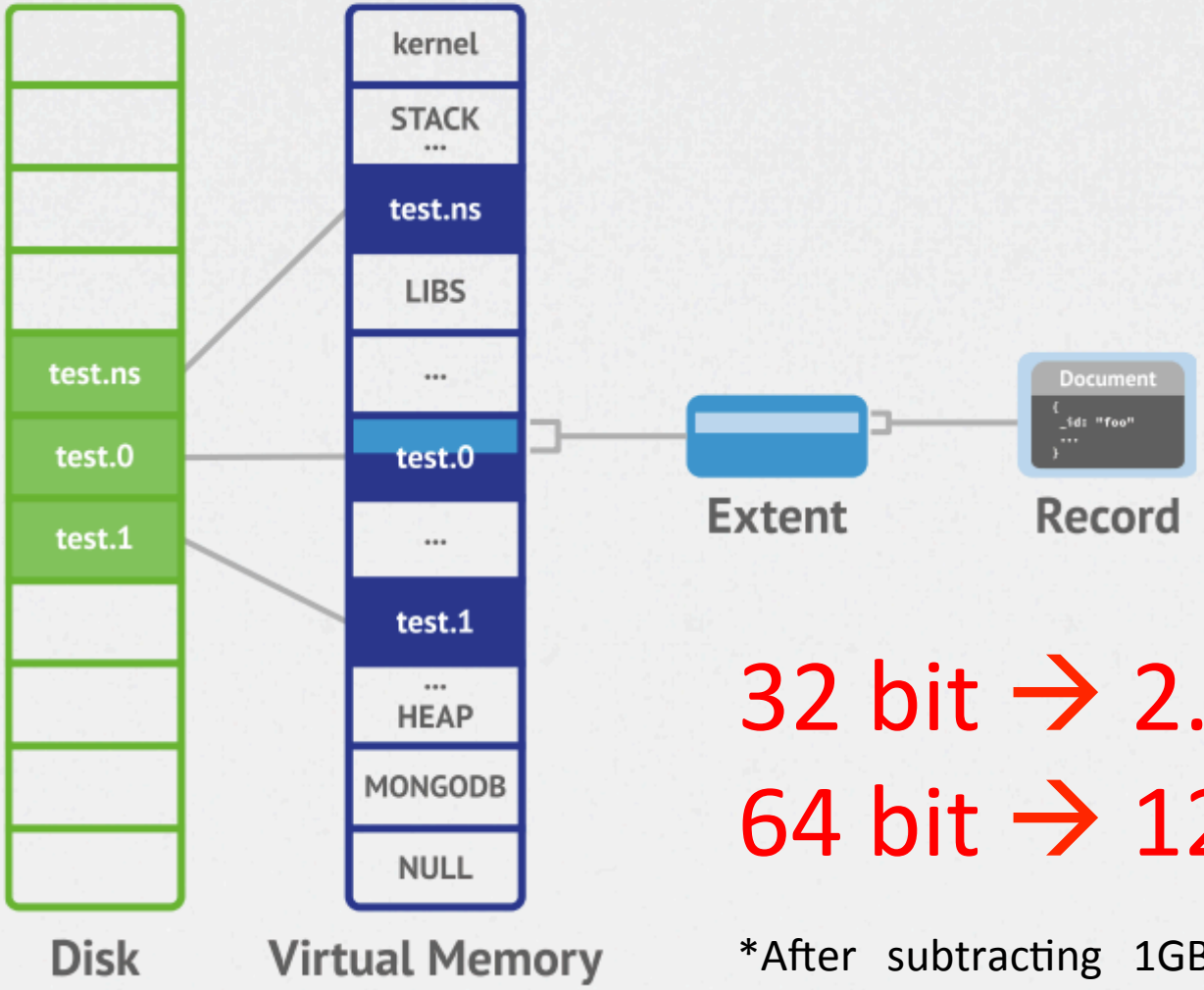
Pros:

- No complex memory / disk code in MongoDB, huge win!
- The OS is very good at caching for any type of storage
- Least Recently Used behavior
- Cache stays warm across MongoDB restarts

Cons:

- RAM usage is affected by disk fragmentation
- RAM usage is affected by high read-ahead
- LRU behavior does not prioritize things (like indexes)

Virtual Address Space



32 bit → 2.5GB

64 bit → 128TB

*After subtracting 1GB for the kernel, 0.5GB for the stack, space for the mongod binary…

How much data is in RAM?

- Resident memory the best indicator of how much data in RAM
- Resident is: process overhead (connections, heap) + FS pages in RAM that were accessed
- Means that it resets to 0 upon restart even though data is still in RAM due to FS cache
- Use free command to check on FS cache size
- Can be affected by fragmentation and read-ahead

Lifecycle of a write

- Changes are written to disk when:
  - msync called
  - File closed, process ends
  - Memory pressure from the OS

- When do we call msync?
  - Once every 60 seconds (--syncdelay)
  - DataFileSync thread does background async flushes for each open file

ls -l

```
total 12644368
drwxr-xr-x  2 jribnik  staff          68 Apr 20 16:50 _tmp
drwxr-xr-x  4 jribnik  staff         136 Apr 20 16:44 journal
-rw-------  1 jribnik  staff    67108864 Apr 20 16:43 local.0
-rw-------  1 jribnik  staff    16777216 Apr 20 16:43 local.ns
-rwxr-xr-x  1 jribnik  staff           6 Apr 20 16:42 mongod.lock
-rw-r--r--  1 jribnik  staff          69 Apr 20 16:42 storage.bson
-rw-------  1 jribnik  staff    67108864 Apr 20 16:45 test.0
-rw-------  1 jribnik  staff   134217728 Apr 20 16:51 test.1
-rw-------  1 jribnik  staff   268435456 Apr 20 16:45 test.2
-rw-------  1 jribnik  staff   536870912 Apr 20 16:46 test.3
-rw-------  1 jribnik  staff  1073741824 Apr 20 16:51 test.4
-rw-------  1 jribnik  staff  2146435072 Apr 20 16:51 test.5
-rw-------  1 jribnik  staff  2146435072 Apr 20 16:51 test.6
-rw-------  1 jribnik  staff    16777216 Apr 20 16:51 test.ns
```

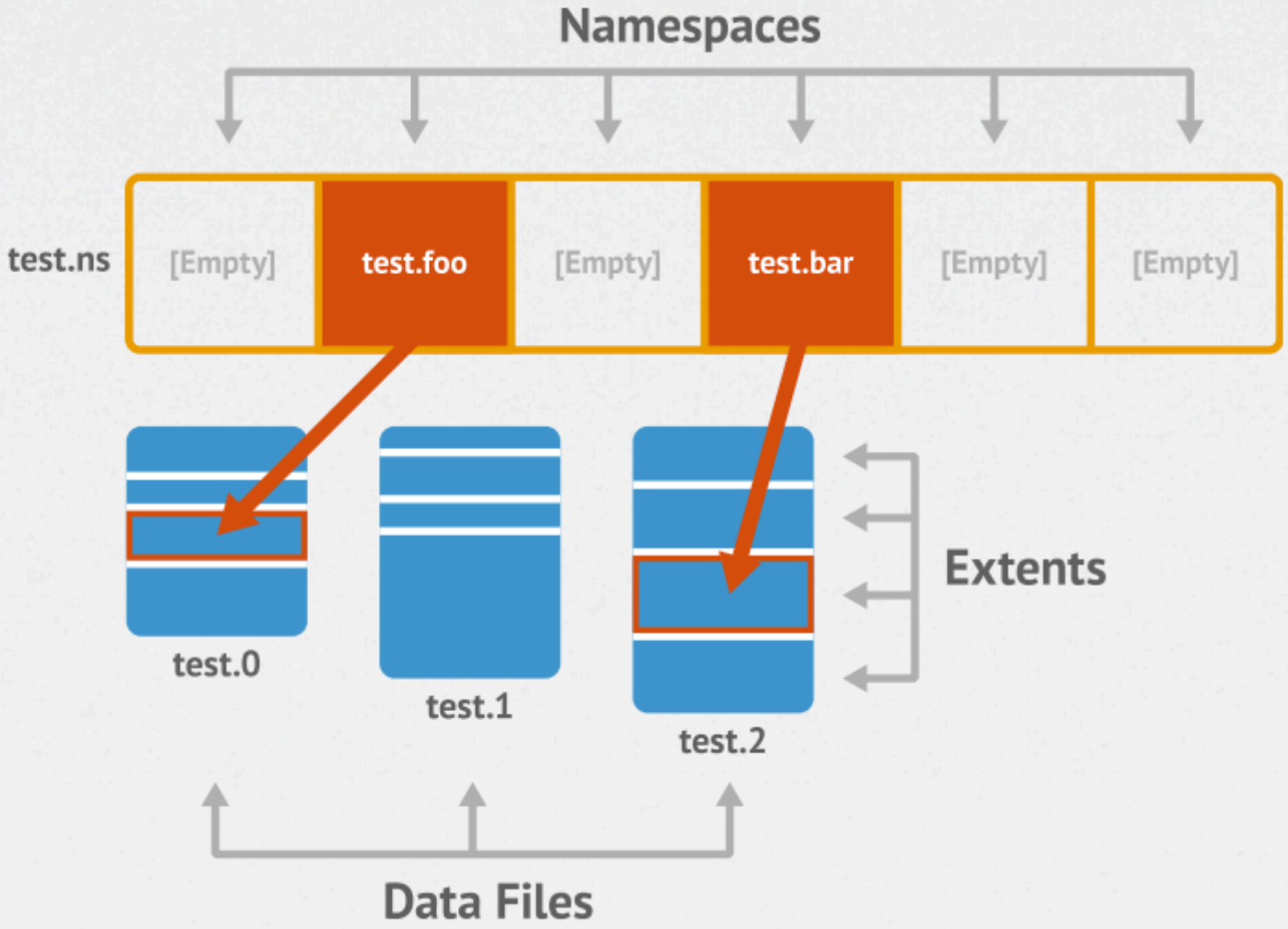
Naming restrictions

- Db name in file name, so no OS forbidden characters, e.g. \0, /, \, .
  - dbname.extension
- Case insensitive
- Cannot rename db just by renaming files

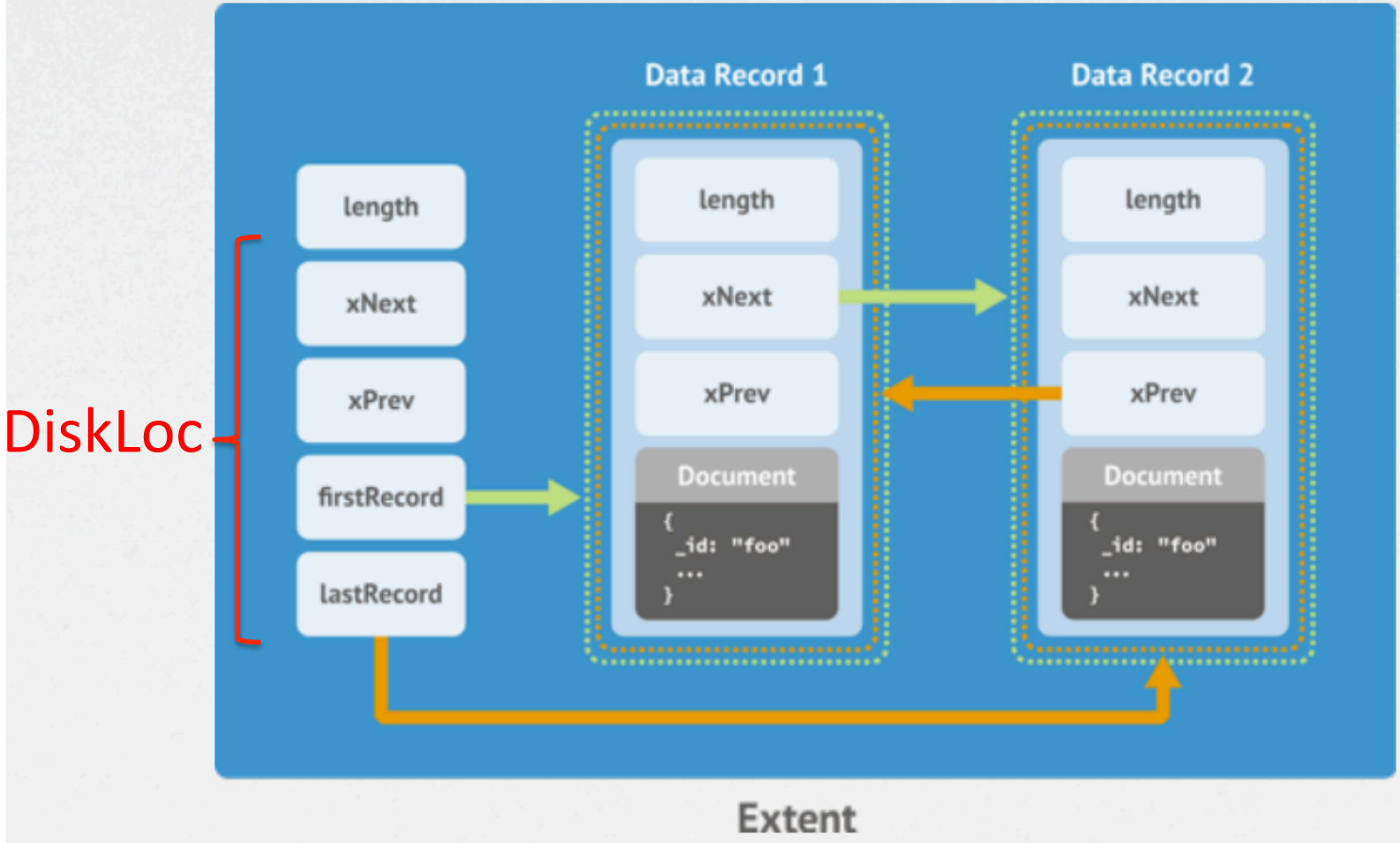
Namespace file

- Giant hash table, fixed in size, 16MB by default (--nssize)
- class NamespaceDetails (628 bytes):
  - DiskLoc firstExtent, lastExtent, deletedList[18]
  - Stats; // fast count()
  - Index data
- DiskLoc is a pointer to a location on disk (8 bytes)
  - int fileNum; // the '0' in test.0
  - int offset; // position in the file
- Limit of ~24K collections and indexes per database

Internal File Format



Extents and Records



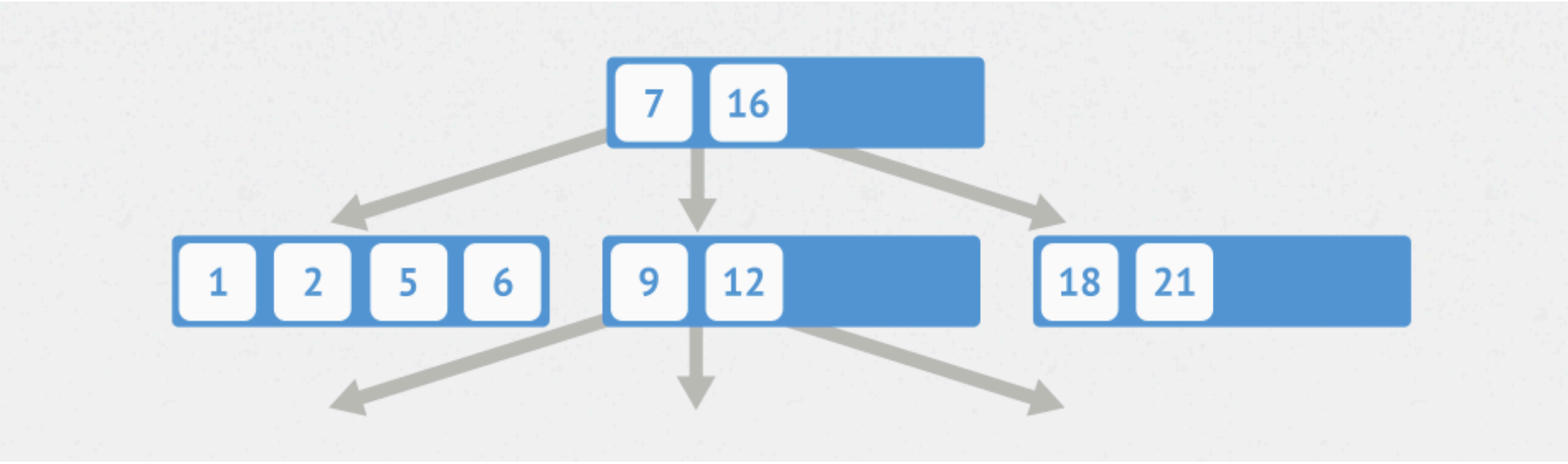
Corruption!?

- Is there a way to check data for corruption?
  - Not easily as there are no checksums
  - DiskLoc contains file number that doesn't exist

- mongodump --repair
  - Does not look for circular references

Indexes

- Btree structures serialized to disk
- Stored in same files as data, but record has index node instead of document



Preallocation

- We are aggressive, always create 1 spare file (why?)
- (0) 64MB, (1) 128MB, … , (5) 2GB, (6) 2GB, …

- Can keep data files smaller with --smallfiles
  - Divide numbers by 4, i.e. double at 512MB
  - This does not actually make your data smaller

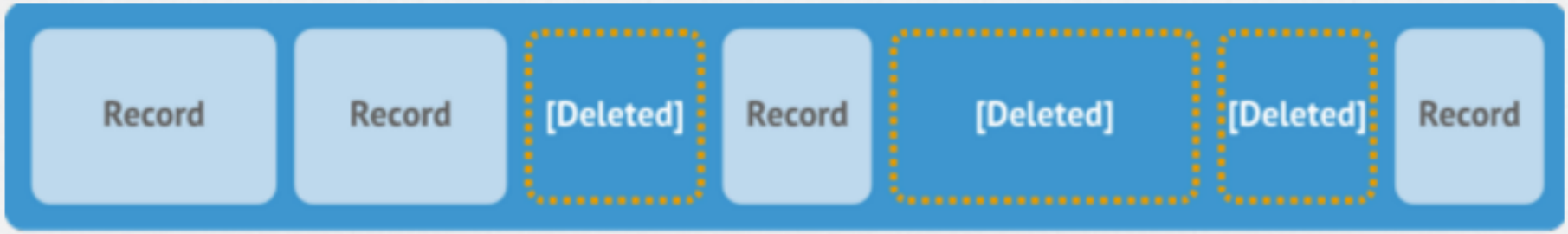
Preallocation continued

- Namespace and numbered files allocated with OS specific implementation of posix_fallocate
  - OS reserves the space, bytes may be zero'ed lazily
- On Linux, fallocate only implemented in a few file drivers (ext4, xfs, …)
- On Windows, NTFS has equivalent, but not FAT
- ZFS (used in Solaris, BSD...) not recommended in Linux due to inefficient preallocation

Their fault, not ours (?)

- Very recently, a 3.x Linux kernel (used in RedHat 6.5) + VMWare caused lazy zeroing to fail
- mongod saw random bytes
- Particularly affected ns files, which are on disk hashes
  - Zeroes → free space
  - Non-zeroes → random garbage
- Since 2.6.5(?) mongod bypasses posix_fallocate to write zero to every block of ns file only

Fragmentation



- Files get fragmented over time from remove() and update() operations
- Wastes disk space AND RAM
- Makes writes scattered and slower

Fragmentation continued

- 2.0 → Added compact command
  - Not automatically performed, maintenance op
- 2.2 → Added PowerOf2Sizes
  - Makes deletedList buckets more reusable
- 2.6 → PowerOf2Sizes enabled by default

The DB Stats

```
> db.stats()
{
    "db" : "test",
    "collections" : 22,
    "objects" : 17000383, ## number of documents
    "avgObjSize" : 44.336902762720l1,
    "dataSize" : 753744328,  ## size of data
    "storageSize" : 1159569408, ## size of all containing
extents
    "numExtents" : 81,
    "indexes" : 85,
    "indexSize" : 624204896, ## separate index storage
size
    "fileSize" : 4176478208, ## size of data files on disk
    "nsSizeMB" : 16,
    "ok" : 1
}
```

The Collection Stats

```
> db.large.stats()
{
    "ns" : "test.large",
    "count" : 5000000, ## number of documents
    "size" : 280000024, ## size of data
    "avgObjSize" : 56.0000048,
    "storageSize" : 409206784, ## size of all containing
extents
    "numExtents" : 18,
    "nindexes" : 1,
    "lastExtentSize" : 74846208,
    "paddingFactor" : 1, ## amount of padding
    "systemFlags" : 0,
    "userFlags" : 0,
    "totalIndexSize" : 162228192, ## separate index storage
size
    "indexSizes" : {
        "_id_" : 162228192
    },
    "ok" : 1
}
```

When mongod starts up…

1. Check for mongod.lock
   - If exists and has length > 0, then there is already a mongod running with this dbpath
   - Contains PID of the running mongod, truncated to length 0 on clean shutdown

2. Check for write-ahead log in journal

Before journaling (1.8), lock file was the only way to detect a clean shutdown.

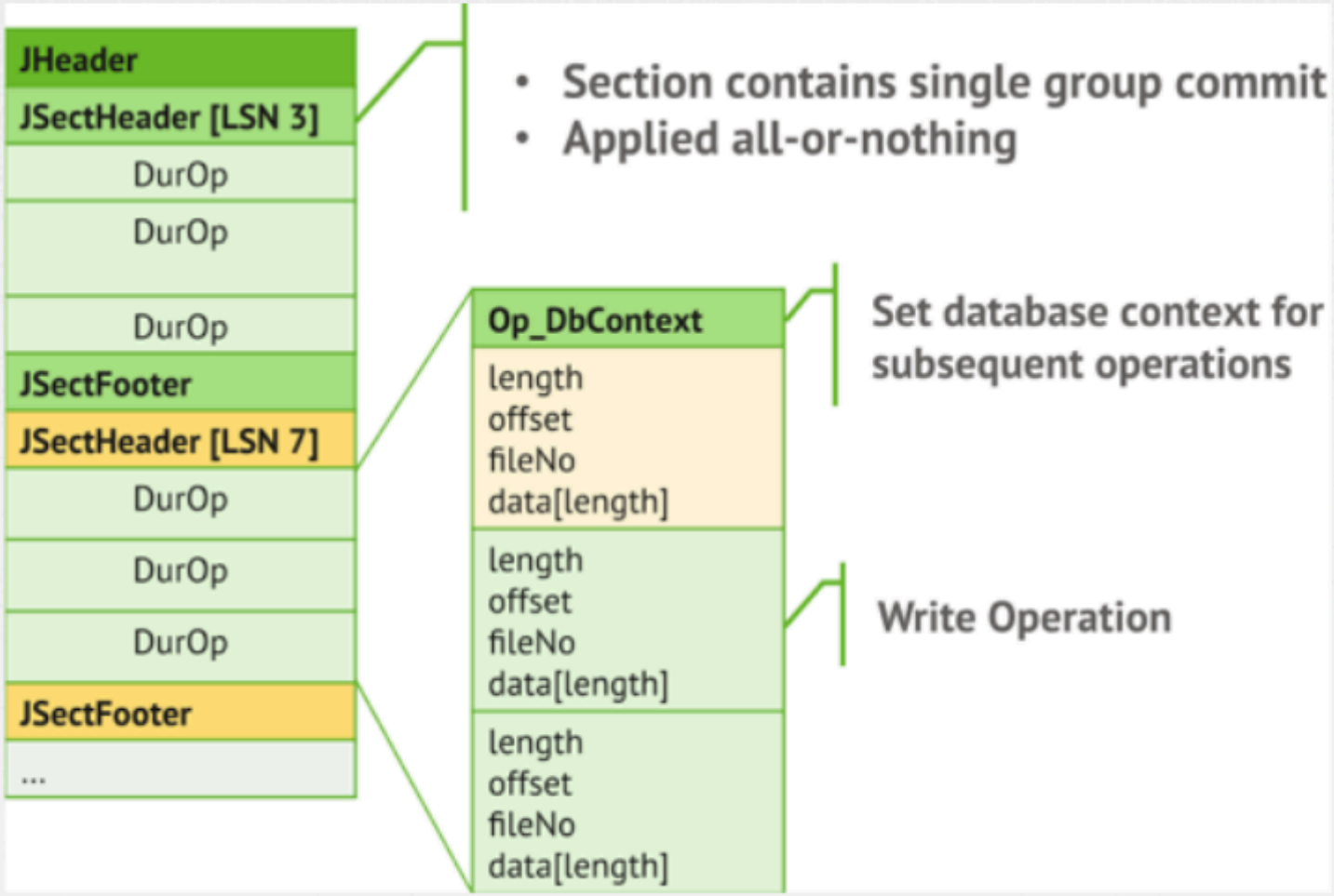
The problem

- Changes in memory mapped files are not applied in order and different parts of the file can be from different points in time
- You want a consistent point-in-time snapshot when restarting after a crash
- In Dec 2010 (version 1.8?), we wanted to introduce a 'writeAhead' mechanism like the RDBMS
- We needed to build a pre-image of what we were going to write to the data file

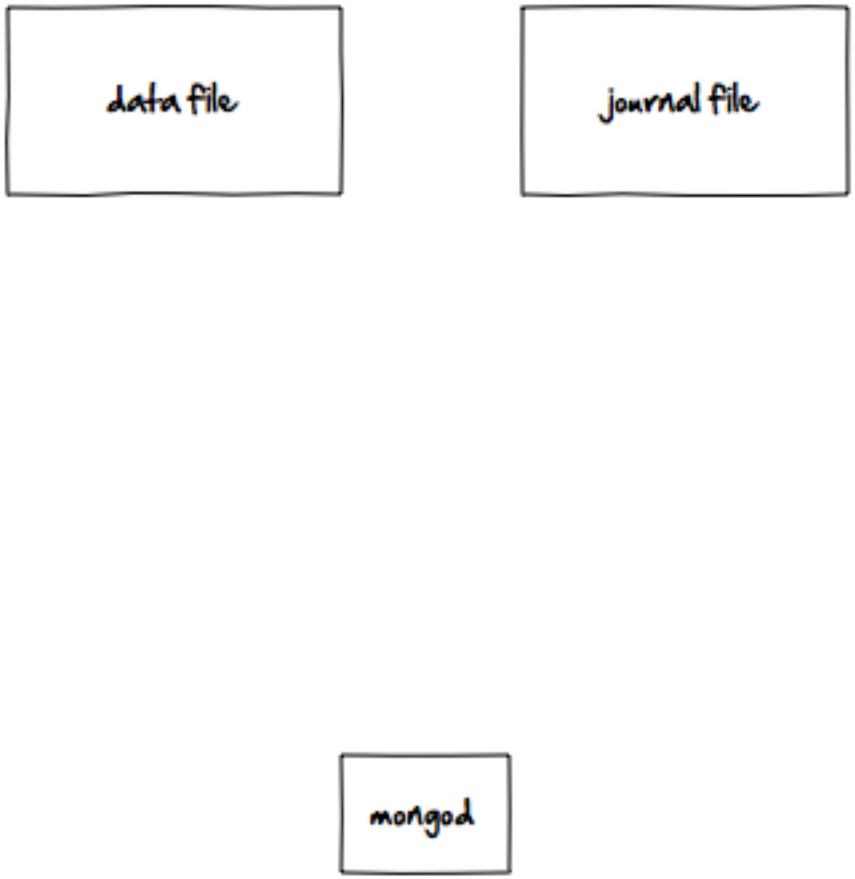
The solution

- Data gets written to a journal before making it to the data files
- Operations written to a journal buffer in RAM that gets flushed every 100ms by default or 100MB
- Once the journal is written to disk, the data is safe
- Journal prevents corruption and allows durability
- Can be turned off, but don't!

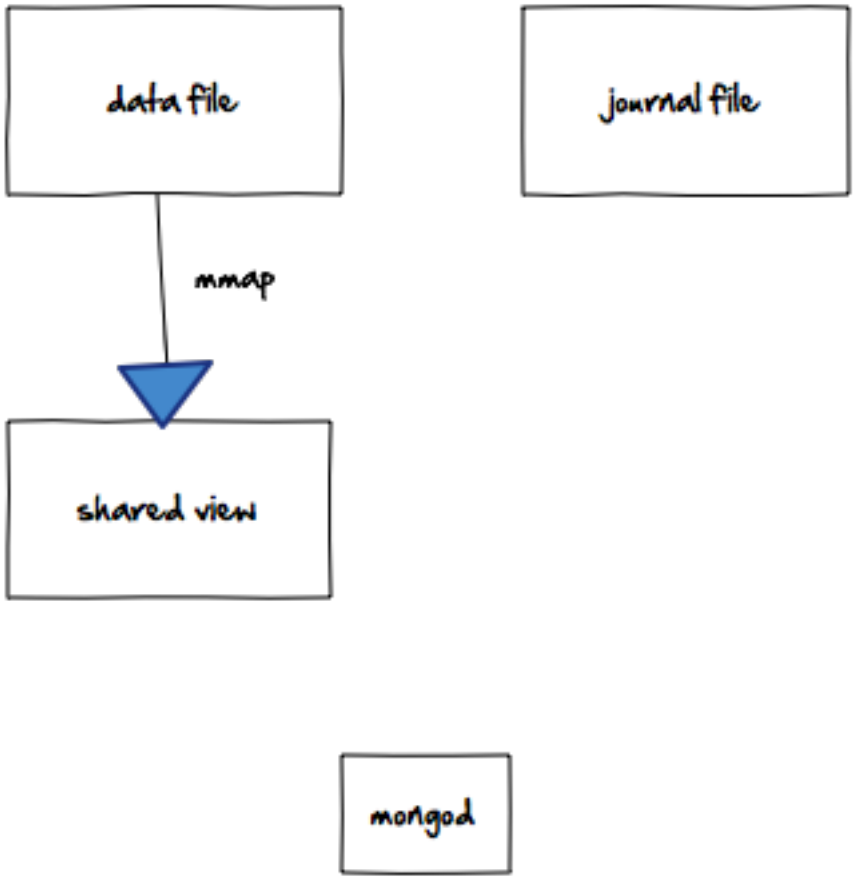
Journal Format

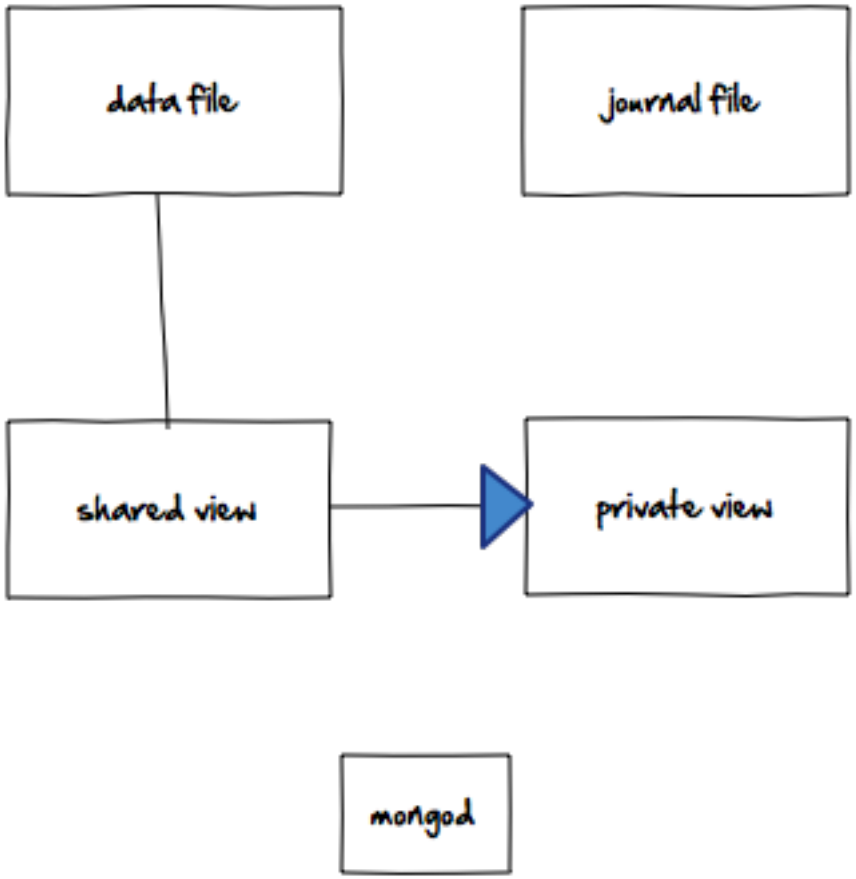


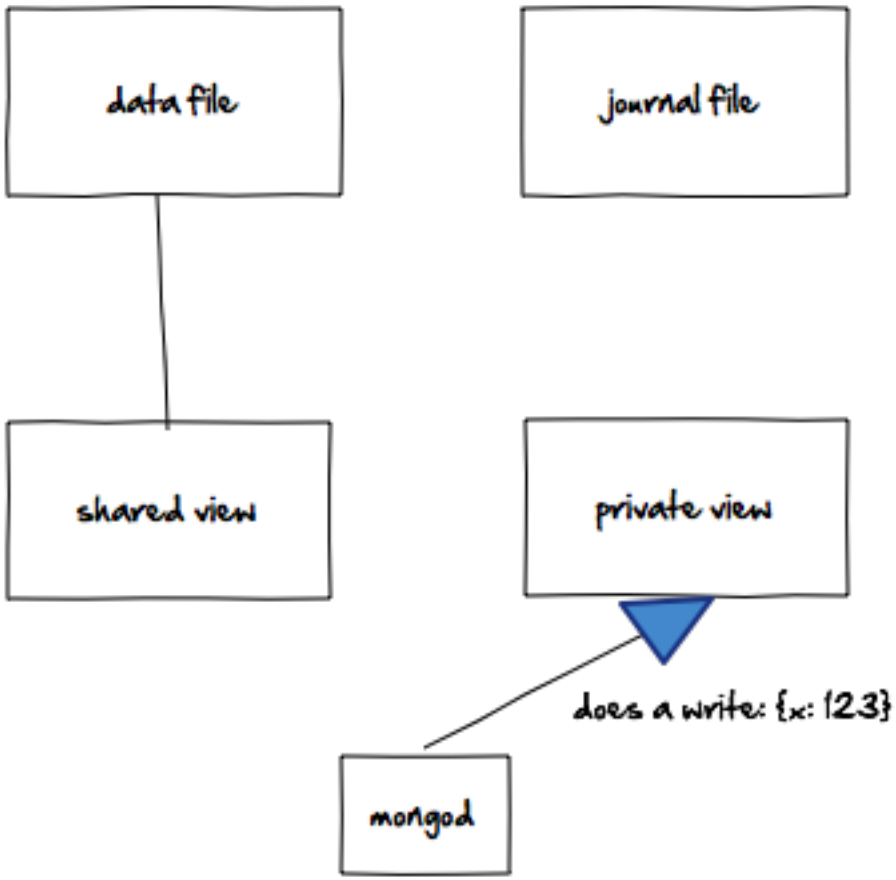
How MongoDB's Journaling Works

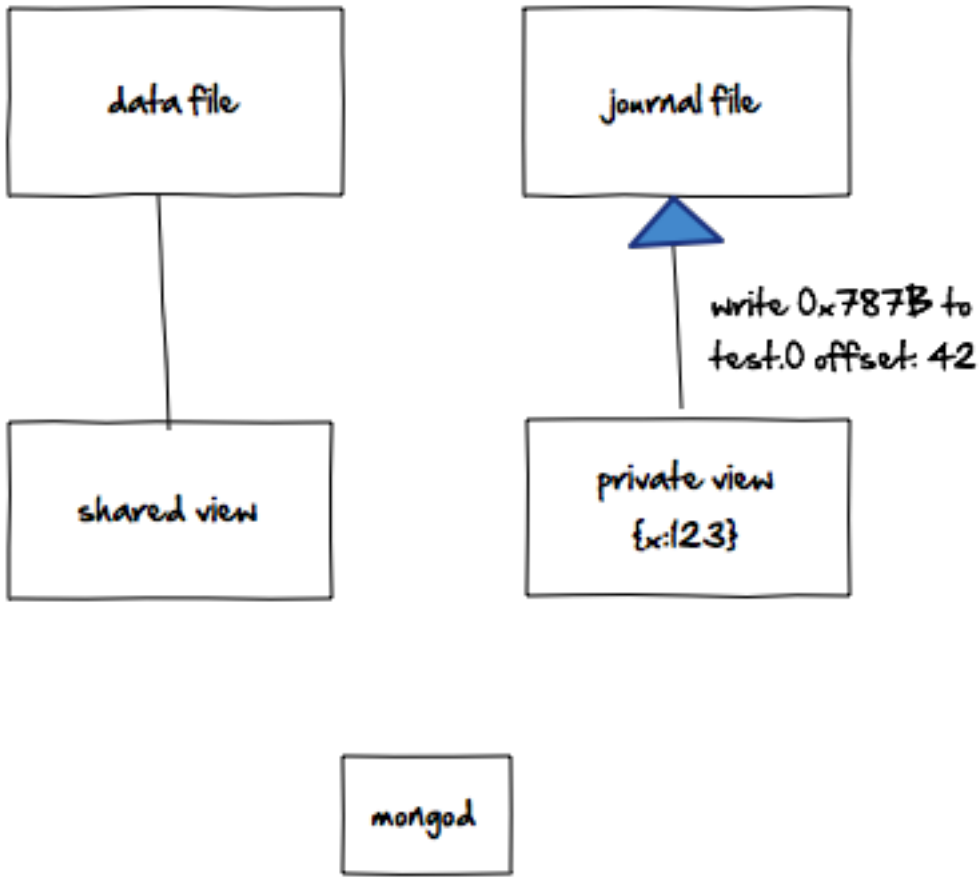


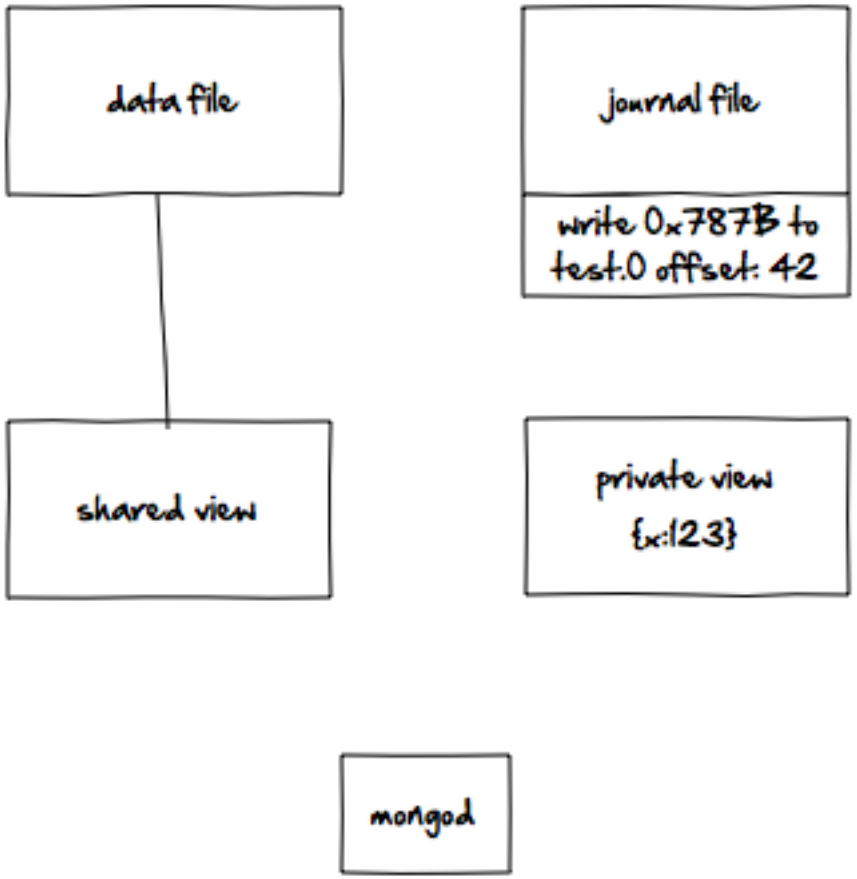
http://www.kchodorow.com/blog/2012/10/04/how-mongodbs-journaling-works/

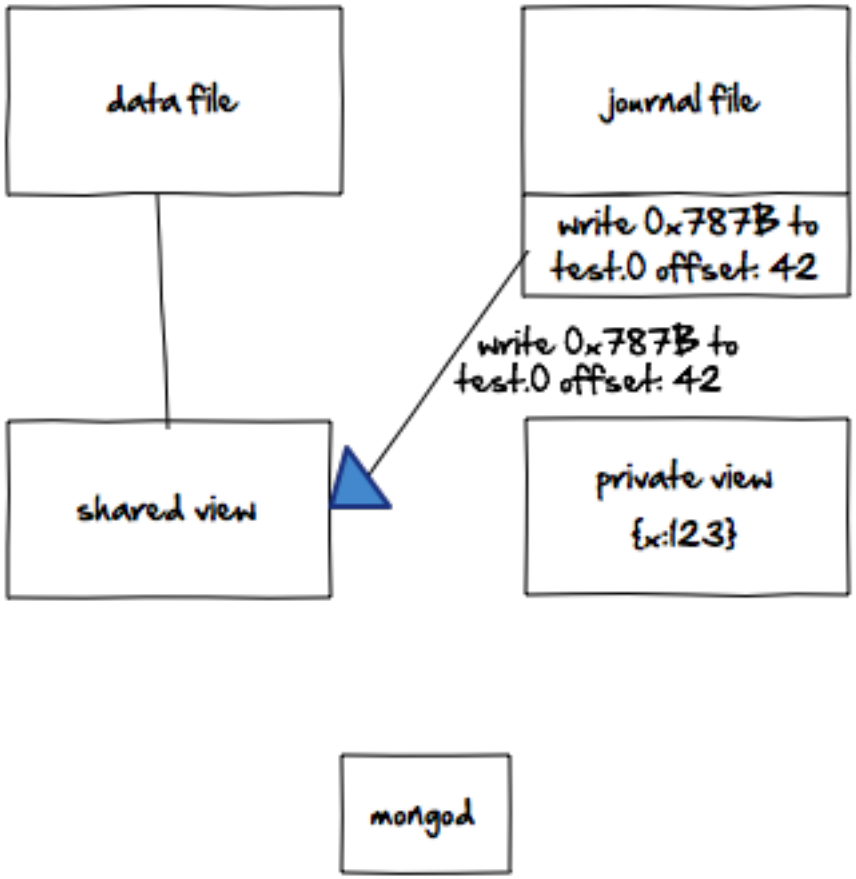

data file
journal file
mmap
shared view
mongod

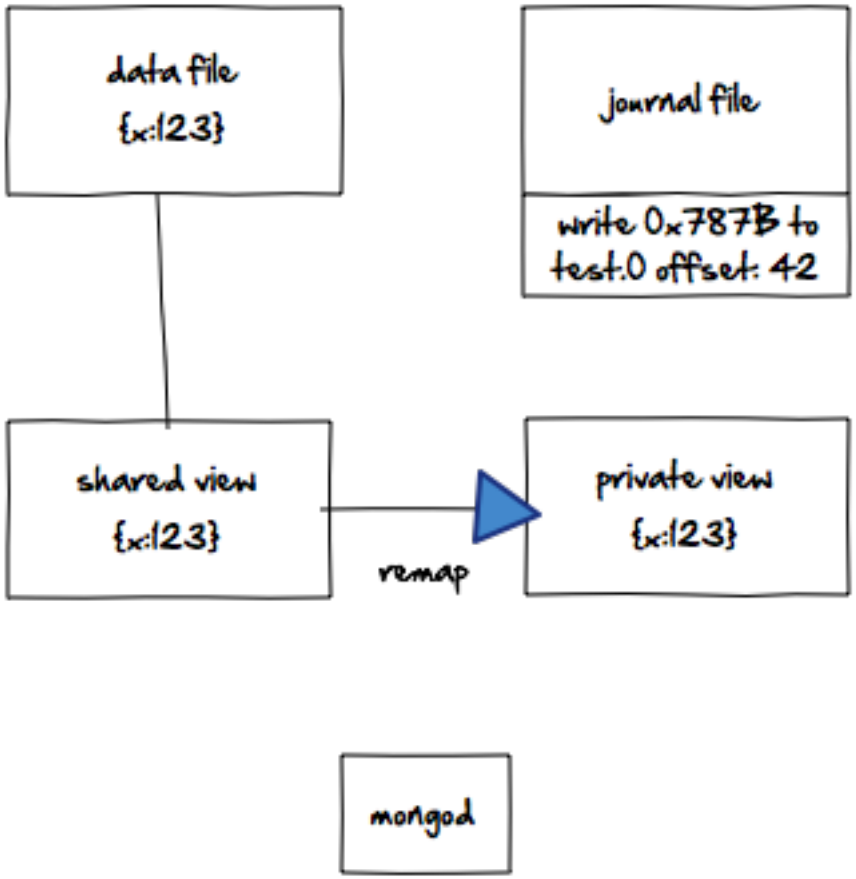

data file
journal file
shared view
private view
mongod

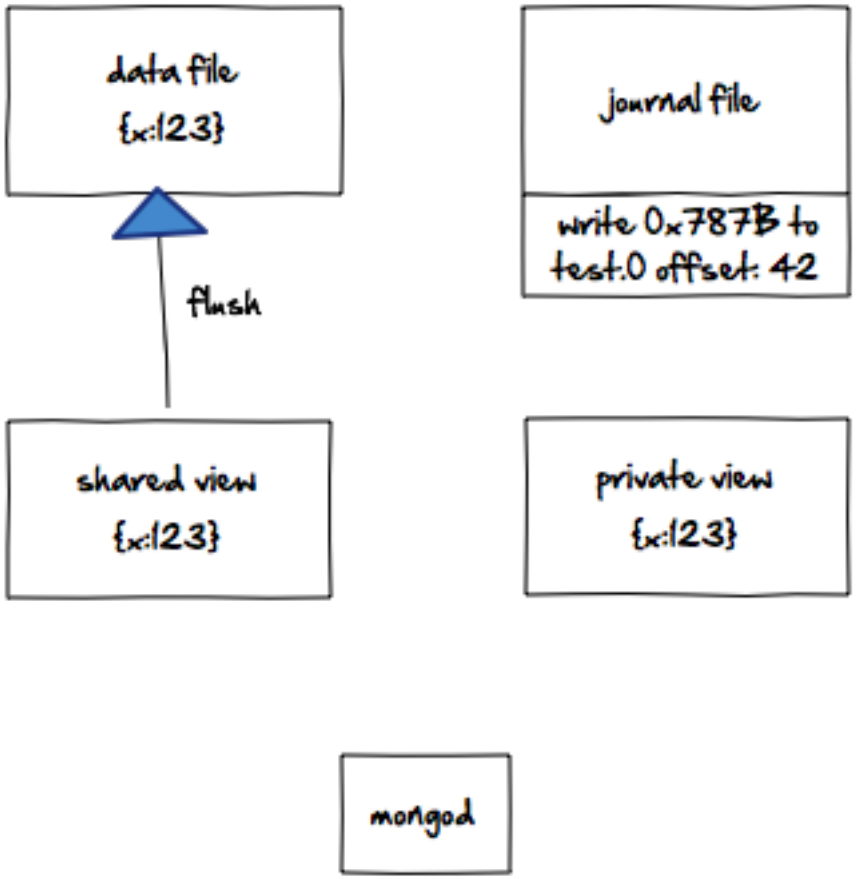

data file
journal file
shared view
private view
does a write: {x: 123}
mongod


data file
journal file
write 0x787B to
test.0 offset: 42
shared view
private view
{x:123}
mongod


data file
journal file
write 0x787B to
test.0 offset: 42
shared view
private view
{x:123}
mongod


data file
journal file
write 0x787B to test.0 offset: 42
write 0x787B to test.0 offset: 42
shared view
private view
{x:123}
mongod


data file
{x:123}
journal file
write 0x787B to
test.0 offset: 42
shared view
{x:123}
remap
private view
{x:123}
mongod


data file
{x:123}
journal file
write 0x787B to
test.0 offset: 42
flush
shared view
{x:123}
private view
{x:123}
mongod

Journal FAQ

- Can I lose data on a hard drive crash?
  - Max of 100ms worth (--journalCommitInterval)
- What are the performance impacts?
  - No impact for read-heavy systems
  - Reduced by 5-30% for writes
  - Use a separate drive for write-heavy apps

Concurrency

- We say we use locks, which scares DBAs as a lock is a heavyweight thing, but really we use latches.

- In earlier versions, there was a single global reader/writer latch. Starting with 2.2, there was a reader/writer latch for each database.

- The latch is multiple-reader, single-writer, and is writer-greedy:
  - There can be an unlimited number of simultaneous readers on a database
  - There can be only one writer at a time on any collection in any one db
  - Writers block out readers
  - "writer-greedy" means that once a write request comes in, all readers are blocked until the write completes

- In 3.0 there is a reader/writer latch for each collection in MMAPv1, and document level locking of some technology with WiredTiger.

WiredTiger

WT Team is located in Boston (USA)
and Sydney (Australia)

WiredTiger Background

- Minimize contention between threads
  - lock-free algorithms
  - minimize blocking due to concurrency control

- Hotter cache and more work per I/O
  - compact file formats
  - compression
  - big-block I/O

Advantages

- Addresses some of the weaknesses of MMAP v1. The storage engine complements MMAP v1 for various use-cases rather than replacing MMAP v1 for all use-cases.
- Facilities for compression, online compaction.
  - zlib and snappy compression are supported.
  - snappy compression is enabled by default and has lower cpu cost with some reasonable compression factor.
  - zlib provides better compression factor compared to snappy but has higher cpu cost.
- Higher concurrency and scaling with document level locking and more liberal use of lock-free data structures to reduce locking contentions.
- Since there are lot of open areas for improvements, it has higher ceiling for improvement compared to MMAP v1.
- Used in various high profile environments like Amazon.
- With document level locking, writes will no longer block other writes.

WiredTiger Storage Engine Features

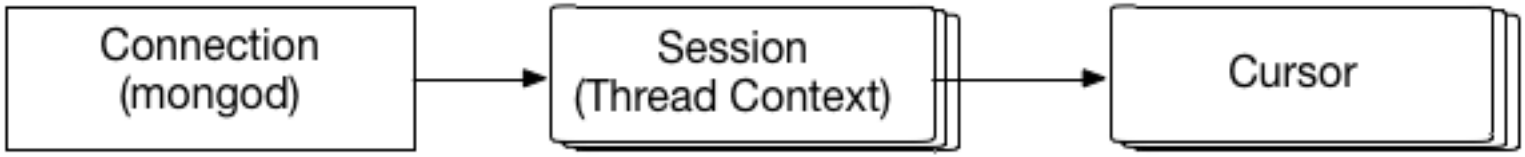
- Document-level concurrency
- Disk Compression
- Consistency without journaling
- Better performance on certain workloads
  - e.g., multi-threaded, write heavy
- Supported platforms
  - Linux/Windows (64 bits)/Mac OSX
- Not supported
  - No Solaris, not yet
  - No Windows 32 bits, likely never

WiredTiger Storage Engine Implementation

Two parts:

- A mapping layer between the WiredTiger API and the MongoDB Storage Engine API.
- The WiredTiger data store.

WiredTiger API

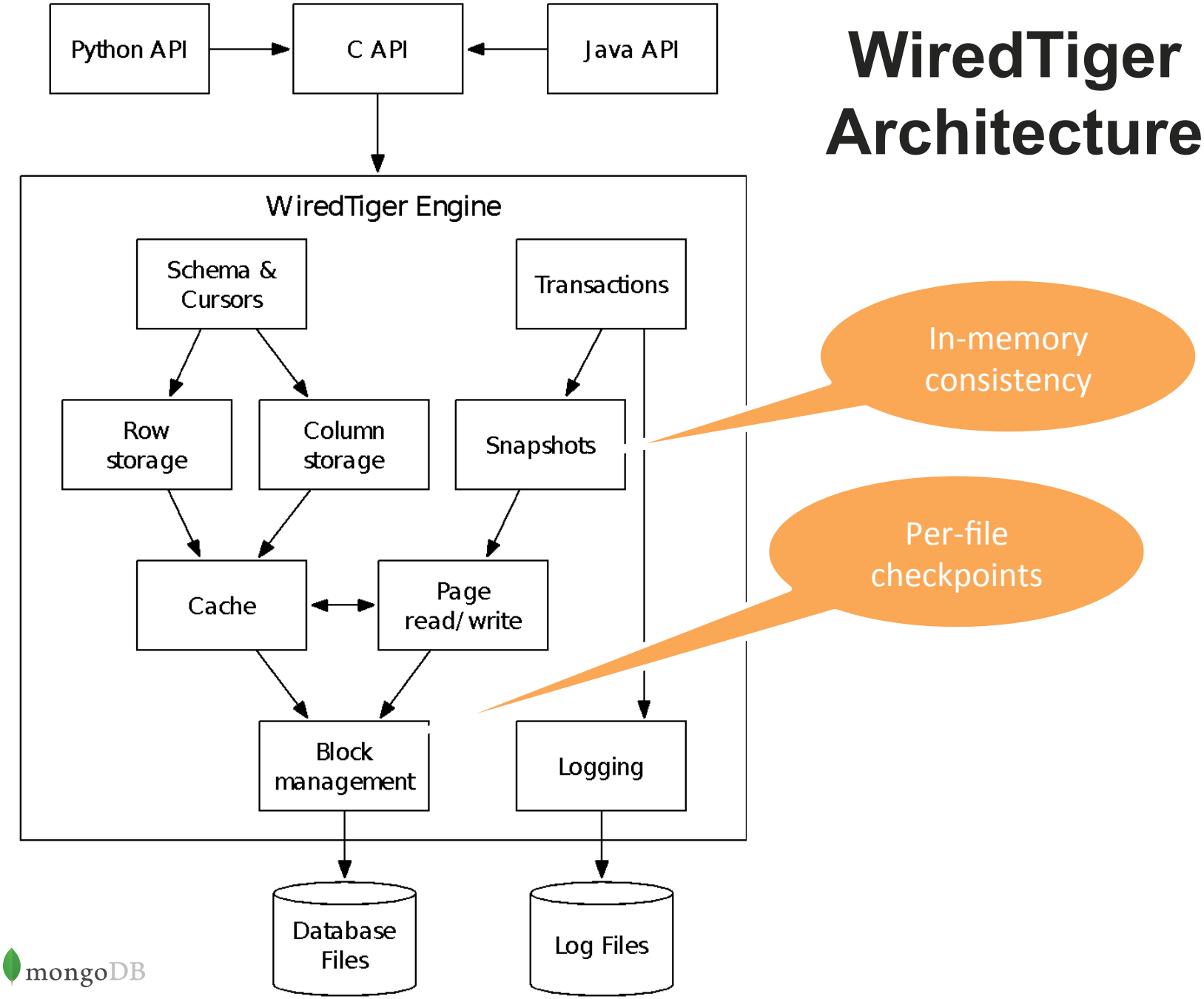


- There is a single connection handle. It owns and manages mongod wide resources.
- There can be multiple session handles.
- Each session can have multiple cursor handles.

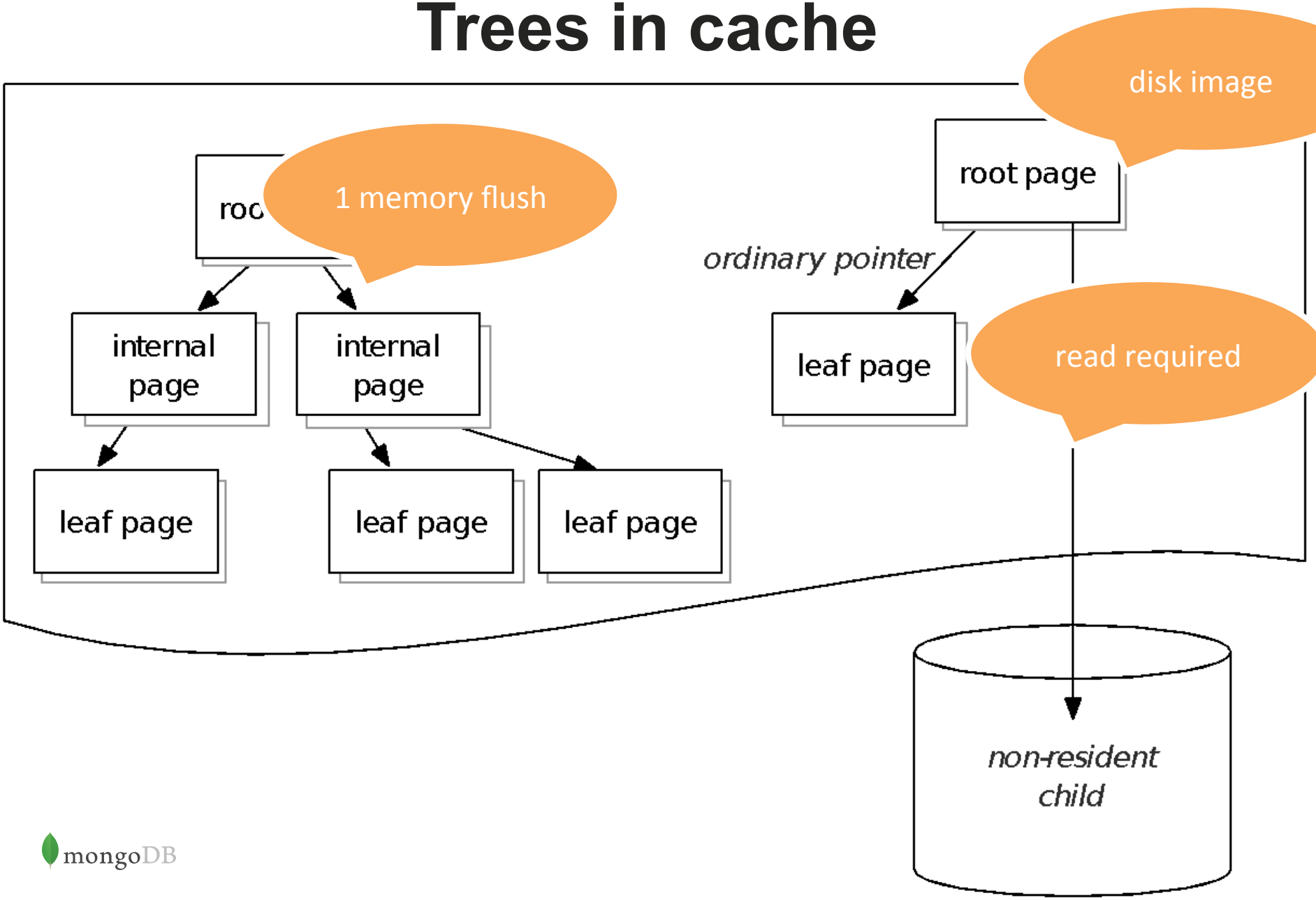
WiredTiger API

- The session handle can be thought of as a thread context.
- Sessions have three main purposes:
  - Schema operations
    create, drop, truncate, verify, etc
  - Transaction management
    begin, commit, rollback
  - Cursor creation
    Table, statistics, etc

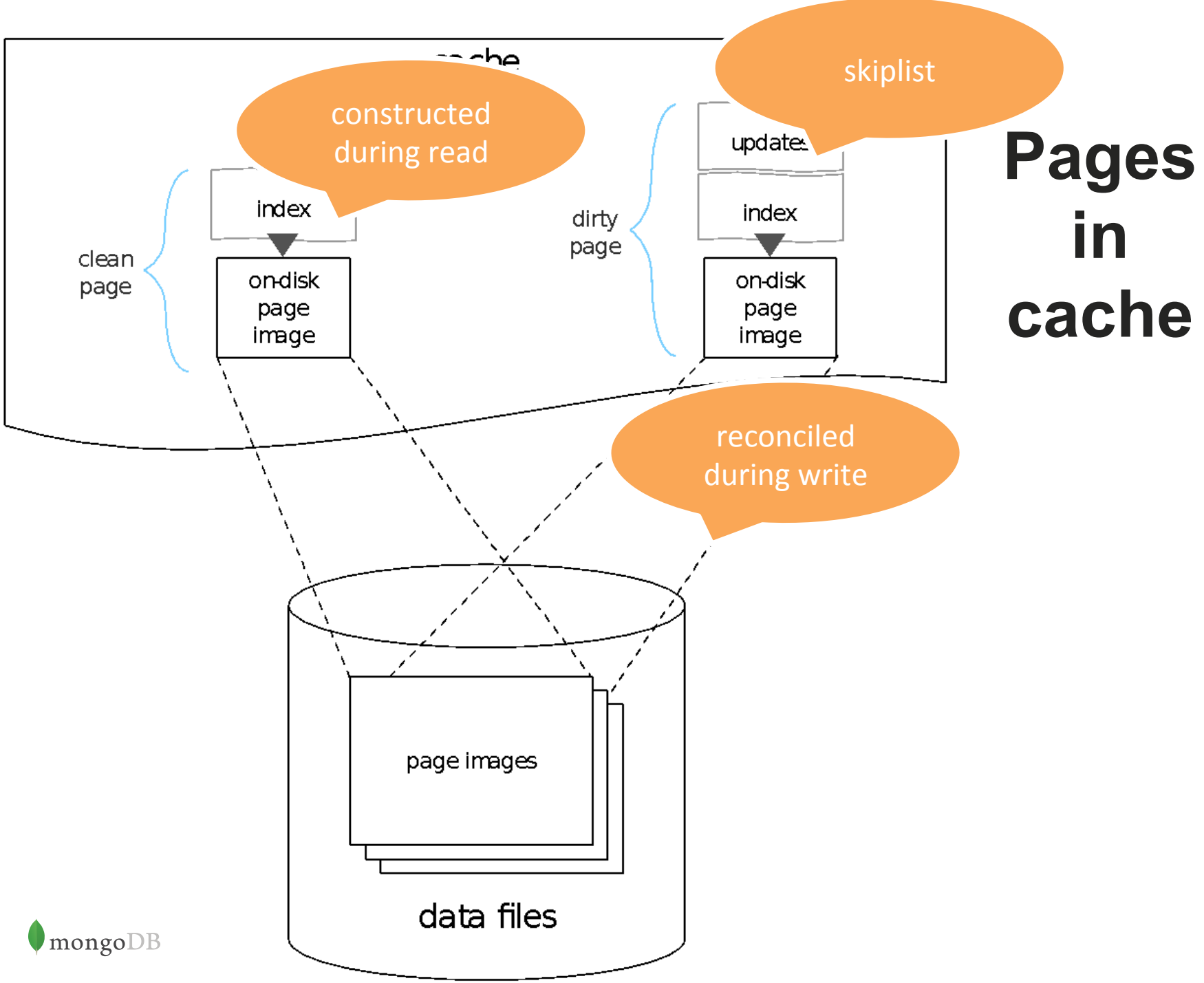
WiredTiger Architecture



Trees in cache



Pages in cache

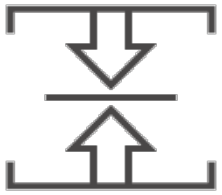


Write a page

- Combine the previous page image with in-memory changes
- Allocate new space in the file
- If the result is too big, split
- Always write multiples of the allocation size
- Can configure direct I/O to keep reads and writes out of the filesystem cache

Record Store

- This is the default format; regular B+ tree on disk.
- Inserts are into one big file for the collection. Collection and Index are self-contained in separate single files.
- Mostly optimized for read loads.
- Has variable length pages on disk for storage.
- Dataset is mostly limited by the RAM available.
- The in-memory representation is different than the on-disk representation. The in-memory organization is in a skip-list.



Compression

- WiredTiger uses snappy compression by default in MongoDB
- Supported compression algorithms
  - snappy [default]: good compression, low overhead
  - zlib: better compression, more CPU
  - none
- Indexes are compressed using prefix compression
  - allows compression in memory

Consistency without Journaling

- MMAPv1 uses a write-ahead log (journal) to guarantee consistency
- WT doesn't have this need: no in-place updates
  - Write-ahead log committed at checkpoints
    - 2GB or 60sec by default – configurable!
  - Updates are written to optional journal as they commit
    - Not flushed by default
    - Recovery rolls forward from last checkpoint
- Replication guarantees the durability

Journal and recovery

- Optional write-ahead logging
- Only written at transaction commit
- snappy compression by default
- Group commit
- Automatic log archive / removal
- On startup, we rely on finding a consistent checkpoint in the metadata
- Check LSNs in the metadata to figure out where to roll forward from

Checkpoint durability

1. Write the leaves
2. Write the internal pages, including the root
   – the old checkpoint is still valid
3. Sync the file
4. Write the new root's address to the metadata
   – free pages from old checkpoints once the metadata is durable

Log Structured Merge trees (LSM)

- Optimized for write workloads.
- Data sets can be much larger than RAM available.
- Writes to multiple files (trees) that are merged in the background.
- Reads search multiple trees using bloom filters to narrow down lookups.
- In-memory and on-disk format are different and unrelated.

Other storage formats

- Column Store: Not supported yet and no time-line
- Index Format: Support both Record Store and LSM. Can mix and match document and index formats

File layout

- No in place updates: Always re-allocated and also no padding factor!
- Will fail to start if mmapv1 files found in dbpath.

Cache - two types

- File system cache
- WT engine cache, by default 50% of the memory or 1GB, whichever is higher

Storage scaling

- Use LVM.

When is data written to disk

- Every 60 seconds or after 2 GB is written
- When WT cache is full, pages will be written to disk

Durability

- Write ahead logging - cannot use journalCommitInterval, but can use j:true and —journal / —no journal
- Old journal files truncated after each checkpoint
- Data files are always consistent, however data loss possible for up to 60 seconds journal flush

Understanding Configurations

What's running?

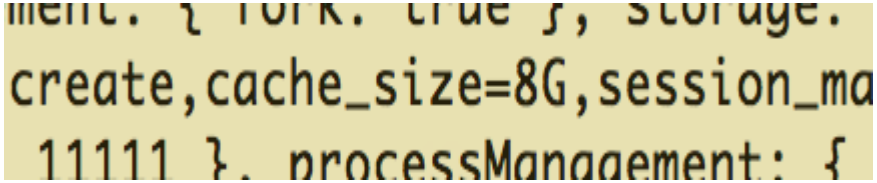
log file
db.server commands
db.collection.stats
mongostat

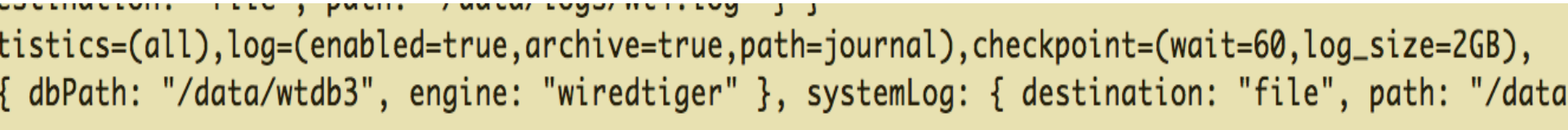
Understanding Configurations

What's running?

log file

- db.server commands
- db.collection.stats
- mongostat





Understanding Configurations

What's running?

- log file
  db.server commands
- db.collection.stats
- mongostat


```
"block-manager" : {
        "mapped bytes read" : 0,
        "bytes read" : 8822272000,
        "bytes written" : 40963559424,
        "mapped blocks read" : 0,
        "blocks pre-loaded" : 0,
        "blocks read" : 360784,
        "blocks written" : 1690173
},
"cache" : {
        "tracked dirty bytes in the cache" : 12843638787,
        "bytes currently in the cache" : 16174048251,
        "maximum bytes configured" : 50465865728,
        "bytes read into cache" : 8780081321,
        "bytes written from cache" : 40729563391,
        "pages evicted by application threads" : 0,
        "checkpoint blocked page eviction" : 0,
        "unmodified pages evicted" : 0,
        "page split during eviction deepened the tree" : 23,
        "modified pages evicted" : 198,
        "pages selected for eviction unable to be evicted" : 55,
        "pages evicted because they exceeded the in-memory maximum" : 286,
        "failed eviction of pages that exceeded the in-memory maximum" : 55,
        "hazard pointer blocked page eviction" : 55,
        "internal pages evicted" : 0,
        "eviction server candidate queue empty when topping up" : 0,
        "eviction server candidate queue not empty when topping up" : 0,
        "eviction server evicting pages" : 0,
        "eviction server populating queue, but not evicting pages" : 0,
        "eviction server unable to reach eviction goal" : 0,
        "pages split during eviction" : 282,
        "pages walked for eviction" : 0,
        "in-memory page splits" : 88,
        "tracked dirty pages in the cache" : 277225,
        "pages currently held in the cache" : 370853,
        "pages read into cache" : 360583,
        "pages written from cache" : 1690124
```

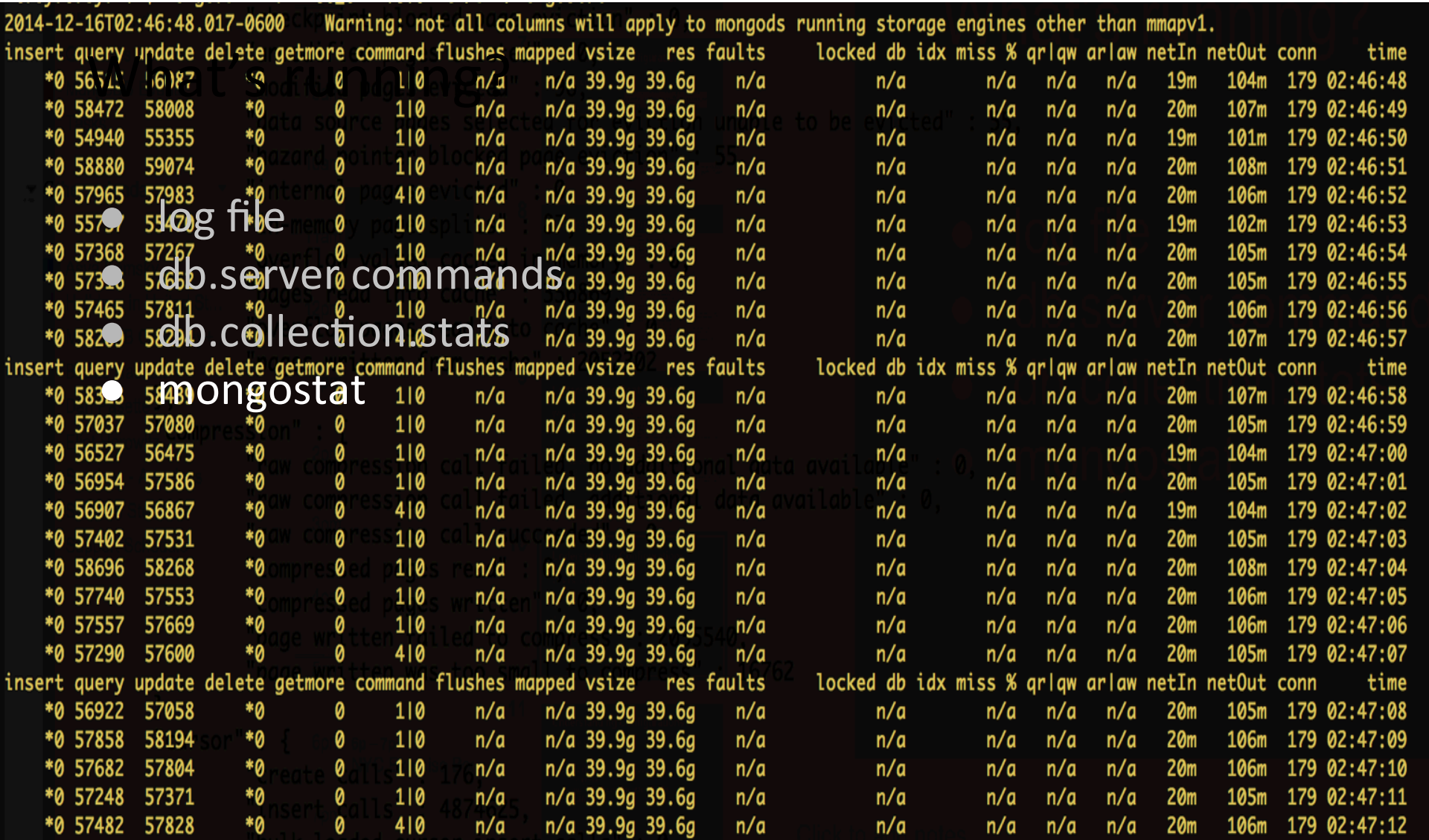
Understanding Configurations

What's running?

- log file
- db.server commands
  db.collection.stats
- mongostat


```
        "fixed-record size" : 0,
        "maximum tree depth" : 20,
        "maximum internal page item size" : 384,
        "maximum internal page size" : 4096,
        "maximum leaf page item size" : 3072,
        "maximum leaf page size" : 32768,
        "overflow pages" : 0,
        "row-store internal pages" : 0,
        "row-store leaf pages" : 0
    },
    "cache" : {
        "bytes read into cache" : 8735074872,
        "bytes written from cache" : 49866161654,
        "checkpoint blocked page eviction" : 0,
        "unmodified pages evicted" : 0,
        "modified pages evicted" : 96,
        "data source pages selected for eviction unable to be evicted" : 55,
        "hazard pointer blocked page eviction" : 55,
        "internal pages evicted" : 0,
        "in-memory page splits" : 87,
        "overflow values cached in memory" : 0,
        "pages read into cache" : 356869,
        "overflow pages read into cache" : 0,
        "pages written from cache" : 2052302
    },
    "compression" : {
        "raw compression call failed, no additional data available" : 0,
        "raw compression call failed, additional data available" : 0,
        "raw compression call succeeded" : 0,
        "compressed pages read" : 0,
        "compressed pages written" : 0,
        "page written failed to compress" : 2035540,
        "page written was too small to compress" : 16762
    },
```

Understanding Configurations



- log file
- db.serverCommands
- db.collection.stats
- mongostat

How Do I Tune WT?

Most important settings:

- cache_size
- checkpoint interval
- logging

cache_size

default size: max(1GB, ½ RAM)

This is where wiredTiger keeps its “working set”

cache_size

default size: max(1GB, ½ RAM)

may need to be different

pay attention when colocating multiple mongods

This is where wiredTiger keeps its “working set”

cache_size

size of the cache is the single most important tuning knob

checkpoints

checkpoints

committing to disk the data written to memory

checkpoints

committing to disk the data written to memory
default: every 60 seconds (or 2GB logged)

checkpoints

committing to disk the data written to memory

default: every 60 seconds (or 2GB logged)

similar to mmapv1 data “fsync”

checkpoints

committing to disk the data written to memory

default: every 60 seconds (or 2GB logged)

~~similar to~~ mmapv1 data “fsync”

very different from (though similar purpose)

logging

logging

logging in WT == “write-ahead log”

same as mmapv1 “journal”

logging

logging in WT == “write-ahead log”

~~same as~~ mmapv1 “journal”
similar to...

logging

logging in WT == “write-ahead log”

~~same as~~ mmapv1 “journal”

~~similar to…~~ except really different

logging

logging in WT == “write-ahead log”

~~same as~~ mmapv1 “journal”
~~similar to…~~ except really different

Provides immediate write durability
Not needed for “crash recovery”

checkpoint + logging

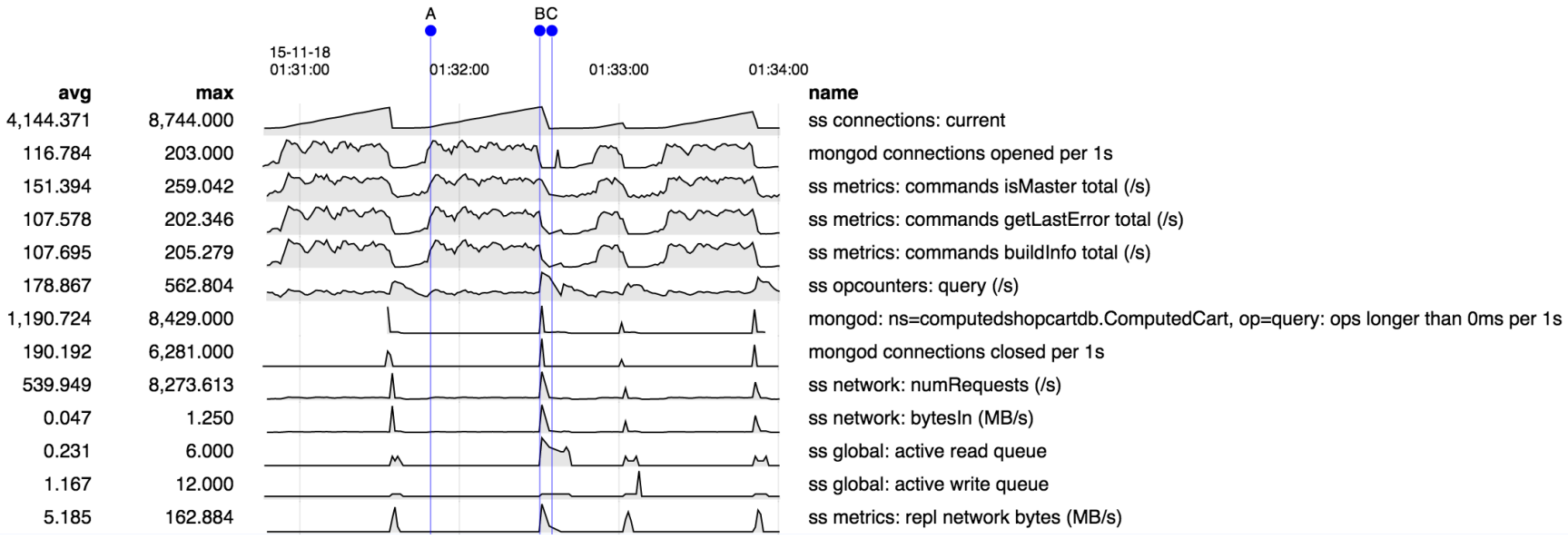
WiredTiger data will be “valid” even if mongod is killed, or crashes, or otherwise not shut down

WAL is not used for “crash recovery” but for recent writes durability - still not necessary if relying on replication for durability, it is not necessary to prevent data corruption

How Do I Administer WT?

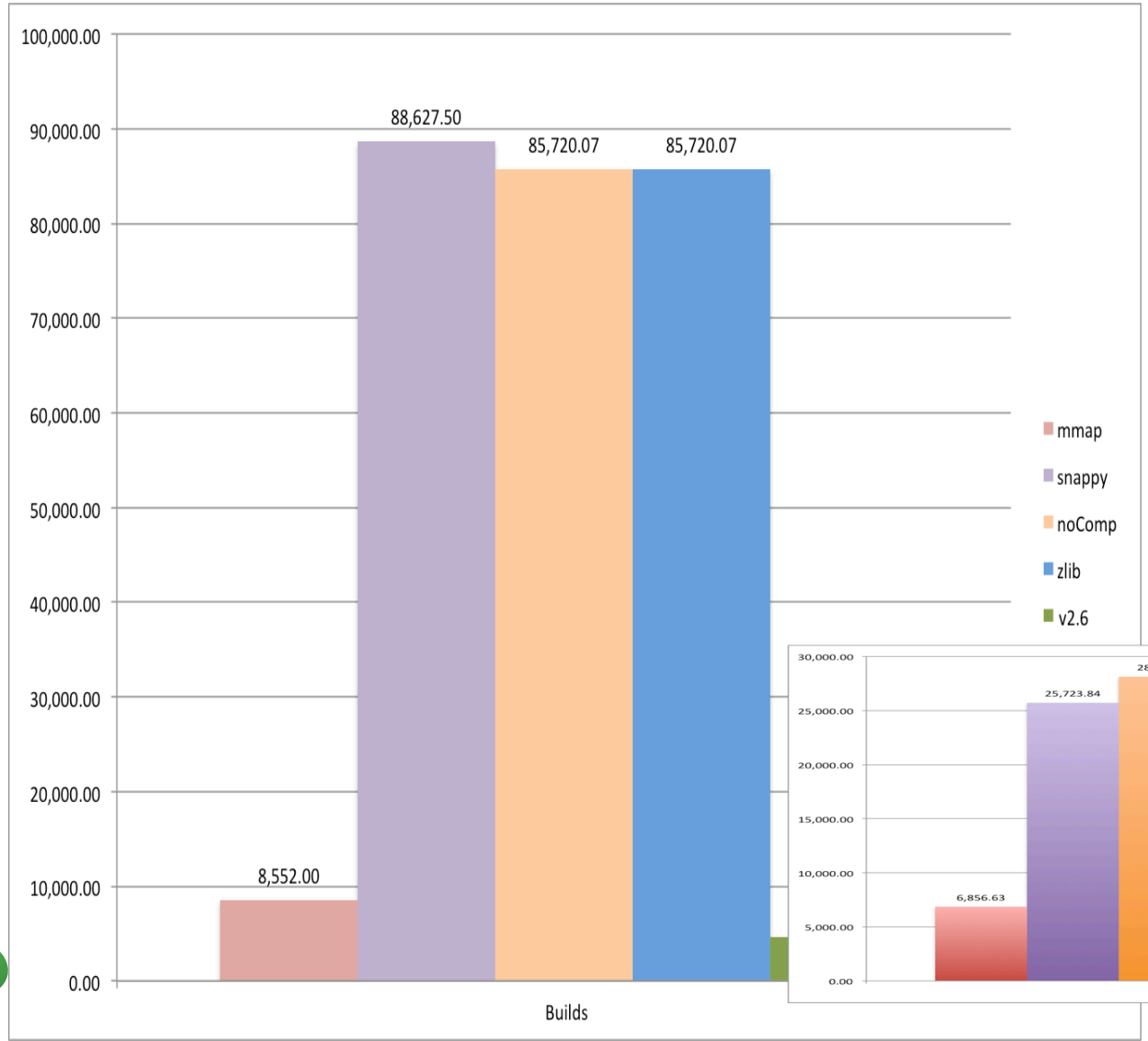
Configure and test (not in production)
Start with defaults
Determine bottleneck
Make corrective configuration changes
Monitor performance

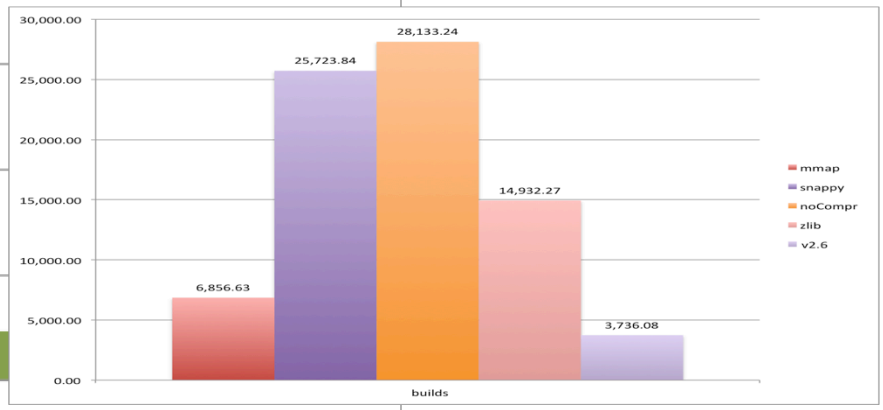
Troubleshooting



avg	max	name
4,144.371	8,744.000	ss connections: current
116.784	203.000	mongod connections opened per 1s
151.394	259.042	ss metrics: commands isMaster total (/s)
107.578	202.346	ss metrics: commands getLastError total (/s)
107.695	205.279	ss metrics: commands buildInfo total (/s)
178.867	562.804	ss opcounters: query (/s)
1,190.724	8,429.000	mongod: ns=computedshopcartdb.ComputedCart, op=query: ops longer than 0ms per 1s
190.192	6,281.000	mongod connections closed per 1s
539.949	8,273.613	ss network: numRequests (/s)
0.047	1.250	ss network: bytesIn (MB/s)
0.231	6.000	ss global: active read queue
1.167	12.000	ss global: active write queue
5.185	162.884	ss metrics: repl network bytes (MB/s)

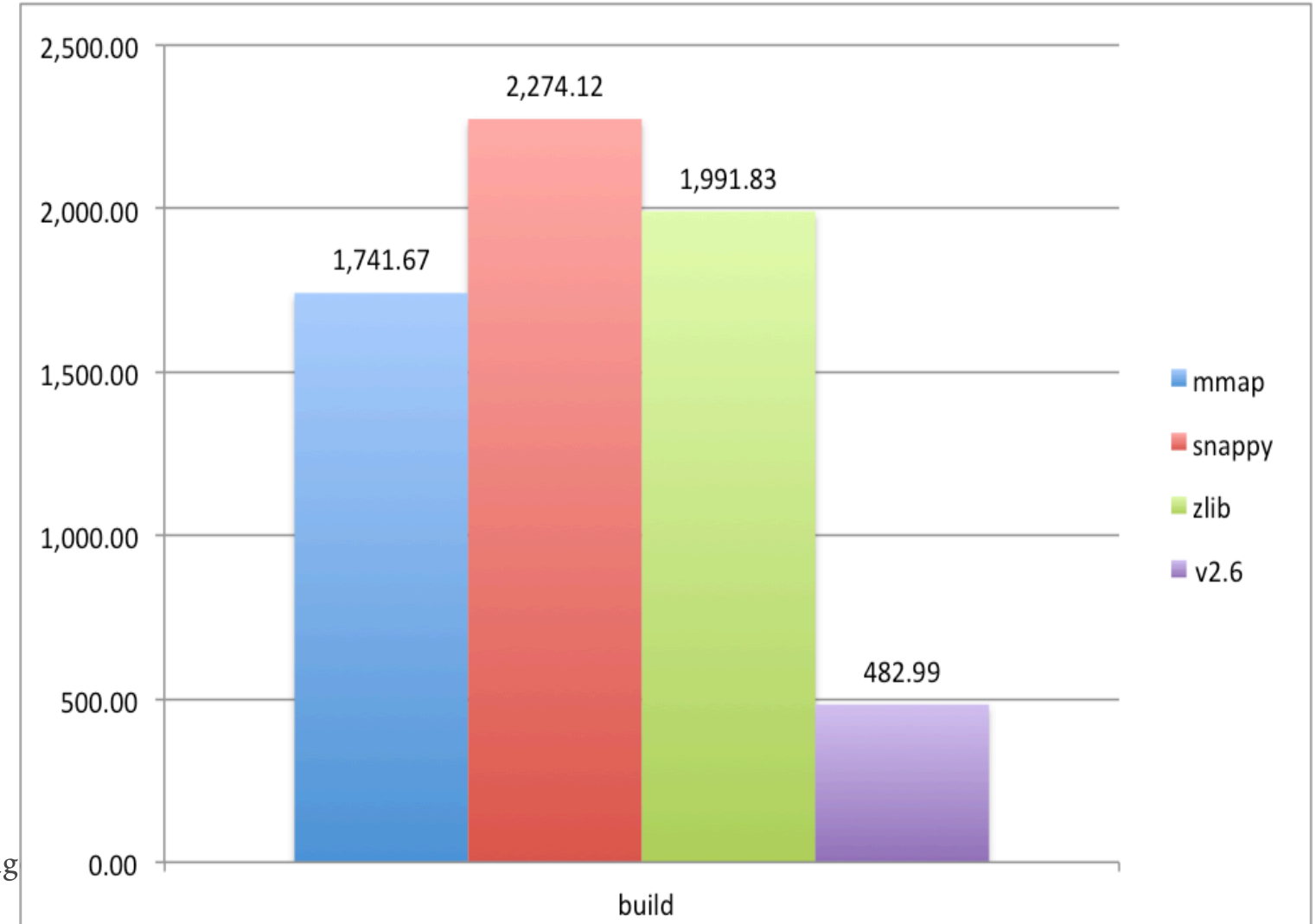
Performance Comparison: iibench





Performance Comparison: sysbench

work phase



Benchmark within WT options

Collection/Index	Btree/Btree	Btree/LSM	LSM/LSM
Average Inserts Per Second (IPS)	31,554	59,208	40,948
Total benchmark runtime (mm:ss)	52:50	28:10	40:43
Average Queries Per Second (QPS)*	261	549	405
On disk size at end of run	46GB	44GB	35GB
Number of latencies over 3 seconds	909	0	608
Max latency	11234	<3000	9780
Total files at end of run	17	382	122

More benchmarking, adding MMapV1

Collection/Index	Btree, Btree	Btree, LSM	MMAP
Average Inserts Per Second (IPS)	124,300	73,998	21,442
Total benchmark runtime (mm:ss)	4:02	6:46	23:20
On disk size at end of run	12G	11G	68G
Number of latencies over 3 seconds	0	0	66
Total files at end of run	17	77	45

Upgrade procedure

- Mongodump/restore
- Initial sync. Replicasets with mixed storage engines are supported
- Cannot copy the data files!

Sizing with WT

Important to take into account that no longer use power of 2 sizes

- more efficient use of disk and memory

Compression ratio of data

- More efficient use of disk

Indexes significantly smaller

- And Compressed both in and out of memory

We now scale with CPU!

Things to look out for

Too many threads for few cores

starves threads, leads to *lower* performance

Not enough threads for many cores

idle CPU, leads to power performance

Workloads that are poor fit for optimistic locking

one collection, majority of updates are to the same document

How to repair a node

- repairDatabase
- compactDatabase

References

- http://docs.mongodb.org/manual/core/storage/