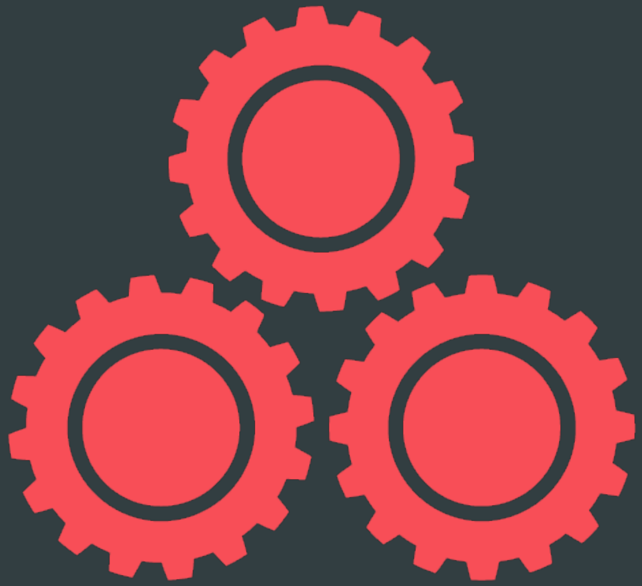
BLUESTORE: A NEW STORAGE BACKEND FOR CEPH – ONE YEAR IN

SAGE WEIL

2017.03.23

OUTLINE

- Ceph background and context
  - FileStore, and why POSIX failed us
- BlueStore – a new Ceph OSD backend
- Performance
- Recent challenges
- Future
- Status and availability
- Summary



MOTIVATION

CEPH

- Object, block, and file storage in a single cluster
- All components scale horizontally
- No single point of failure
- Hardware agnostic, commodity hardware
- Self-manage whenever possible
- Open source (LGPL)

- “A Scalable, High-Performance Distributed File System”
- “performance, reliability, and scalability”

CEPH COMPONENTS

OBJECT | BLOCK | FILE

RGW
A web services gateway for object storage, compatible with S3 and Swift

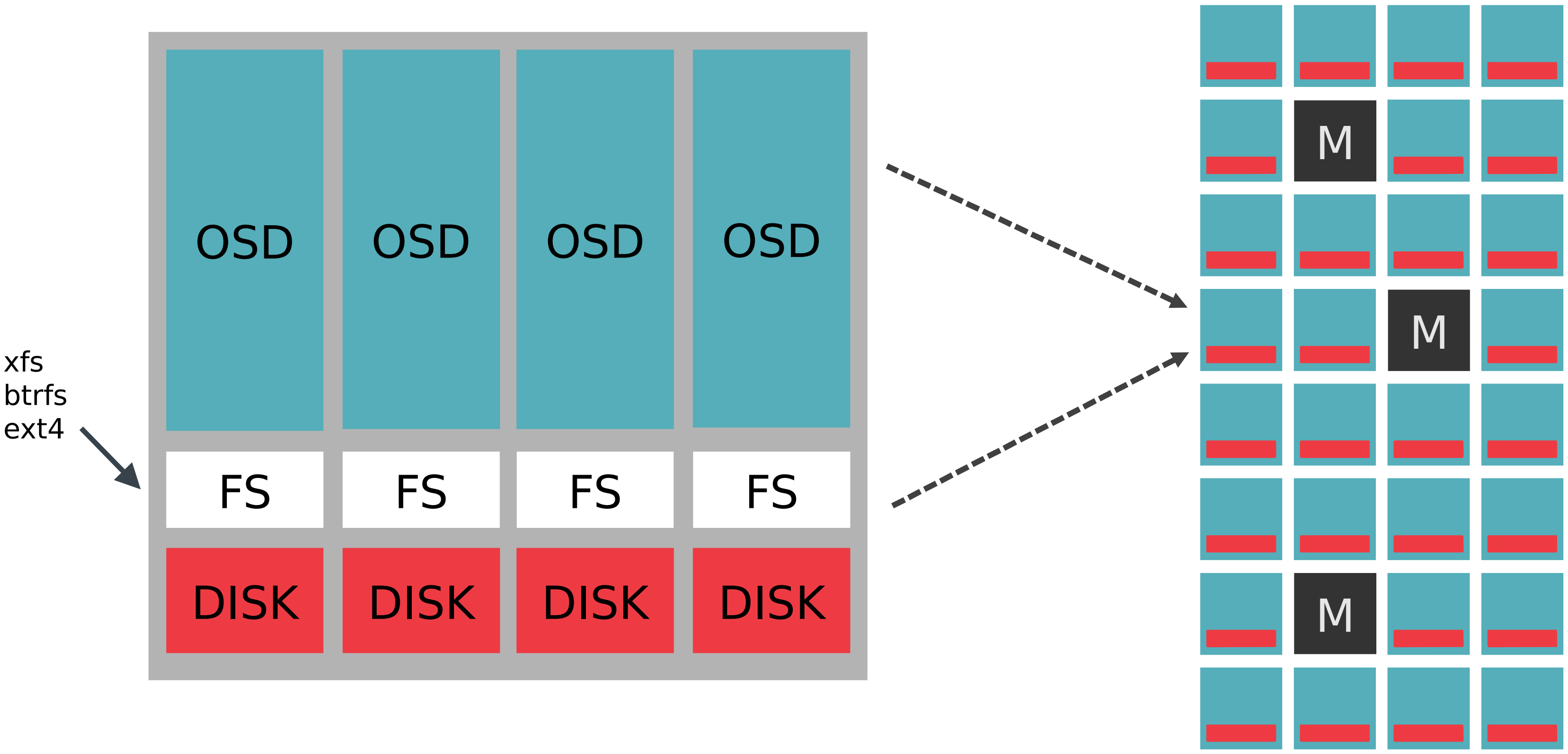
RBD
A reliable, fully-distributed block device with cloud platform integration

CEPHFS
A distributed file system with POSIX semantics and scale-out metadata management

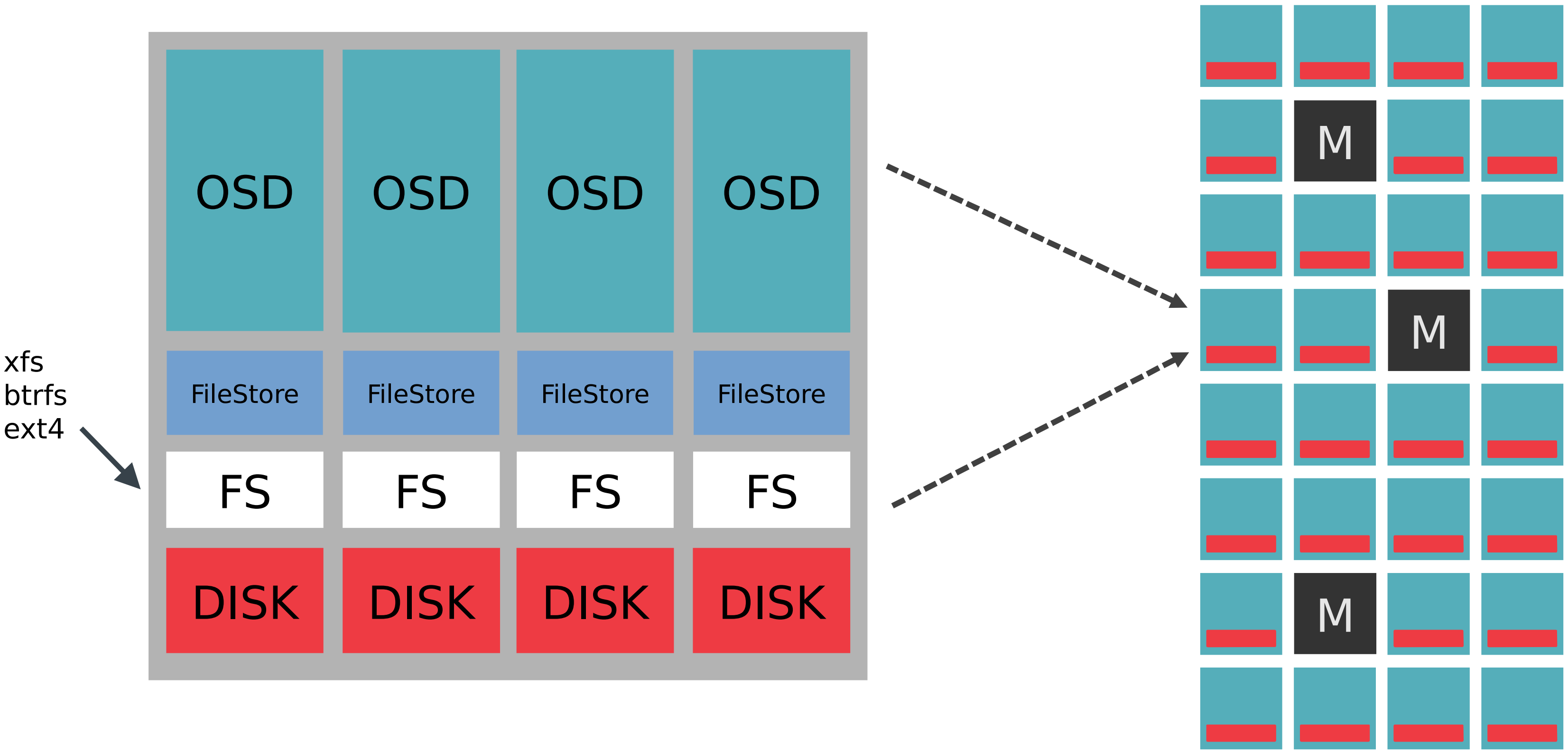
LIBRADOS
A library allowing apps to directly access RADOS (C, C++, Java, Python, Ruby, PHP)

RADOS
A software-based, reliable, autonomous, distributed object store comprised of self-healing, self-managing, intelligent storage nodes and lightweight monitors

OBJECT STORAGE DAEMONS (OSDS)



OBJECT STORAGE DAEMONS (OSDS)



OBJECTSTORE AND DATA MODEL

- ObjectStore
  - abstract interface for storing local data
  - EBOFS, FileStore
- EBOFS
  - a user-space **e**xtent-**b**ased **o**bject **f**ile **s**ystem
  - deprecated in favor of FileStore on btrfs in 2009
- Object – “file”
  - data (file-like byte stream)
  - attributes (small key/value)
  - omap (unbounded key/value)
- Collection – “directory”
  - placement group shard (slice of the RADOS pool)
- All writes are transactions
  - **A**tomic + **C**onsistent + **D**urable
  - **I**solation provided by OSD

FILESTORE

- FileStore
  - PG = collection = directory
  - object = file
- Leveldb
  - large xattr spillover
  - object omap (key/value) data
- Originally just for development...
  - later, only supported backend (on XFS)

- /var/lib/ceph/osd/ceph-123/
  - current/
    - meta/
      - osdmap123
      - osdmap124
    - 0.1_head/
      - object1
      - object12
    - 0.7_head/
      - object3
      - object5
    - 0.a_head/
      - object4
      - object6
    - omap/
      - <leveldb files>

POSIX FAILS: TRANSACTIONS

- Most transactions are simple
  - write some bytes to object (file)
  - update object attribute (file xattr)
  - append to update log (kv insert)

  ...but others are arbitrarily large/complex
- Serialize and write-ahead txn to journal for atomicity
  - We double-write everything!
  - Lots of ugly hackery to make replayed events idempotent


```
[
    {
        "op_name": "write",
        "collection": "0.6_head",
        "oid": "#0:73d87003:::benchmark_data_gnit_10346_object23:head#",
        "length": 4194304,
        "offset": 0,
        "bufferlist length": 4194304
    },
    {
        "op_name": "setattrs",
        "collection": "0.6_head",
        "oid": "#0:73d87003:::benchmark_data_gnit_10346_object23:head#",
        "attr_lens": {
            "_": 269,
            "snapset": 31
        }
    },
    {
        "op_name": "omap_setkeys",
        "collection": "0.6_head",
        "oid": "#0:60000000::::head#",
        "attr_lens": {
            "0000000005.00000000000000000006": 178,
            "_info": 847
        }
    }
]
```

POSIX FAILS: ENUMERATION

- Ceph objects are distributed by a 32-bit hash
- Enumeration is in hash order
  - scrubbing
  - “backfill” (data rebalancing, recovery)
  - enumeration via librados client API
- POSIX readdir is not well-ordered
  - And even if it were, it would be a different hash
- Need O(1) “split” for a given shard/range
- Build directory tree by hash-value prefix
  - split any directory when size > ~100 files
  - merge when size < ~50 files
  - read entire directory, sort in-memory

```
…
DIR_A/
DIR_A/A03224D3_qwer
DIR_A/A247233E_zxcv
…
DIR_B/
DIR_B/DIR_8/
DIR_B/DIR_8/B823032D_foo
DIR_B/DIR_8/B8474342_bar
DIR_B/DIR_9/
DIR_B/DIR_9/B924273B_baz
DIR_B/DIR_A/
DIR_B/DIR_A/BA4328D2_asdf
…
```

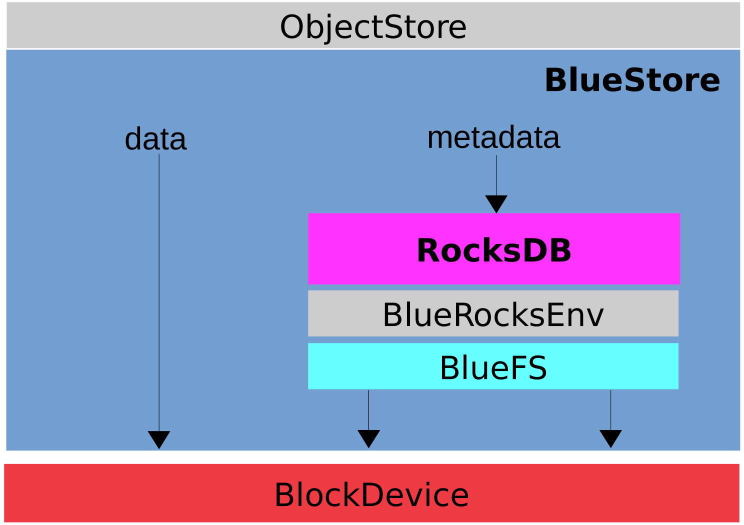
THE HEADACHES CONTINUE

- New FileStore problems continue to surface as we approach switch to BlueStore
  - Recently discovered bug in FileStore omap implementation, revealed by new CephFS scrubbing
  - FileStore directory splits lead to throughput collapse when an entire pool’s PG directories split in unison
  - Read/modify/write workloads perform horribly
    - RGW index objects
    - RBD object bitmaps
  - QoS efforts thwarted by deep queues and periodicity in FileStore throughput
  - Cannot bound deferred writeback work, even with fsync(2)
  - {RBD, CephFS} snapshots triggering inefficient 4MB object copies to create object clones

BLUESTORE

BLUESTORE

- BlueStore = **Bl**ock + **NewStore**
  - consume raw block device(s)
  - key/value database (RocksDB) for metadata
  - data written directly to block device
  - pluggable block Allocator (policy)
  - pluggable compression
  - checksums, ponies, ...
- We must share the block device with RocksDB

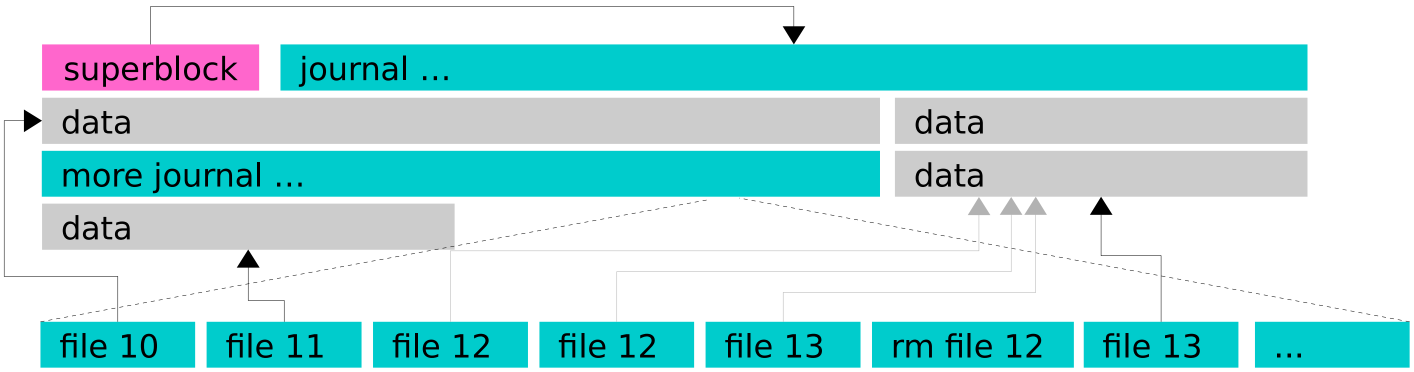


ROCKSDB: BLUEROCKSENV + BLUEFS

- class BlueRocksEnv : public rocksdb::EnvWrapper
  - passes “file” operations to BlueFS
- BlueFS is a super-simple “file system”
  - all metadata lives in the journal
  - all metadata loaded in RAM on start/mount
  - no need to store block free list
  - coarse allocation unit (1 MB blocks)
  - journal rewritten/compacted when it gets large

- Map “directories” to different block devices
  - db.wal/ – on NVRAM, NVMe, SSD
  - db/ – level0 and hot SSTs on SSD
  - db.slow/ – cold SSTs on HDD

- BlueStore periodically balances free space



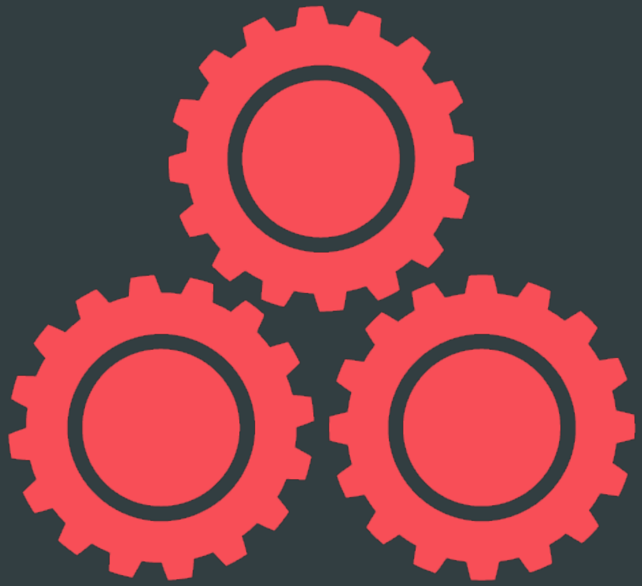
MULTI-DEVICE SUPPORT

- Single device
  - HDD or SSD
    - Bluefs db.wal/ + db/ (wal and sst files)
    - object data blobs

- Two devices
  - 512MB of SSD or NVRAM
    - bluefs db.wal/ (rocksdb wal)
  - big device
    - bluefs db/ (sst files, spillover)
    - object data blobs

- Two devices
  - a few GB of SSD
    - bluefs db.wal/ (rocksdb wal)
    - bluefs db/ (warm sst files)
  - big device
    - bluefs db.slow/ (cold sst files)
    - object data blobs

- Three devices
  - 512MB NVRAM
    - bluefs db.wal/ (rocksdb wal)
  - a few GB SSD
    - bluefs db/ (warm sst files)
  - big device
    - bluefs db.slow/ (cold sst files)
    - object data blobs



METADATA

BLUESTORE METADATA

- Everything in flat kv database (rocksdb)
- Partition namespace for different metadata
  - S* – “superblock” properties for the entire store
  - B* – block allocation metadata (free block bitmap)
  - T* – stats (bytes used, compressed, etc.)
  - C* – collection name → cnode_t
  - O* – object name → onode_t or bnode_t
  - X* – shared blobs
  - L* – deferred writes (promises of future IO)
  - M* – omap (user key/value data, stored in objects)

CNODE

- Collection metadata
  - Interval of object namespace

```
  shard  pool  hash     name       bits
C<NOSHARD,12,3d3e0000> “12.e3d3” = <19>


  shard  pool  hash   name snap   gen
O<NOSHARD,12,3d3d880e,foo,NOSNAP,NOGEN> = …
O<NOSHARD,12,3d3d9223,bar,NOSNAP,NOGEN> = …
O<NOSHARD,12,3d3e02c2,baz,NOSNAP,NOGEN> = …
O<NOSHARD,12,3d3e125d,zip,NOSNAP,NOGEN> = …
O<NOSHARD,12,3d3e1d41,dee,NOSNAP,NOGEN> = …
O<NOSHARD,12,3d3e3832,dah,NOSNAP,NOGEN> = …
```

```
struct spg_t {
  uint64_t pool;
  uint32_t hash;
  shard_id_t shard;
};

struct bluestore_cnode_t {
  uint32_t bits;
};
```

- Nice properties
  - Ordered enumeration of objects
  - We can “split” collections by adjusting collection metadata only

ONODE

- Per object metadata
  - Lives directly in key/value pair
  - Serializes to 100s of bytes
- Size in bytes
- Attributes (user attr data)
- Inline extent map (maybe)

```
struct bluestore_onode_t {
  uint64_t size;
  map<string,bufferptr> attrs;
  uint64_t flags;

  struct shard_info {
    uint32_t offset;
    uint32_t bytes;
  };
  vector<shard_info> shards;

  bufferlist inline_extents;
  bufferlist spanning_blobs;
};
```

BLOBS

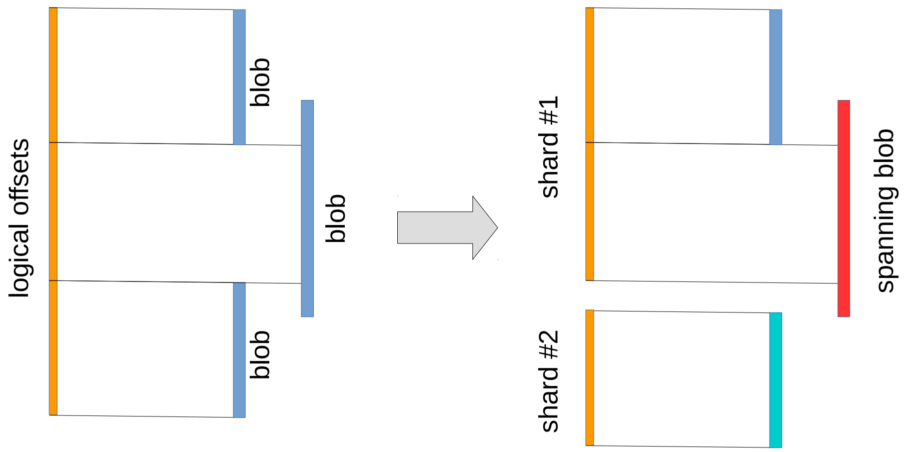
- Blob
  - Extent(s) on device
  - Lump of data originating from same object
  - May later be referenced by multiple objects
  - Normally checksummed
  - May be compressed
- SharedBlob
  - Extent ref count on cloned blobs
  - In-memory buffer cache

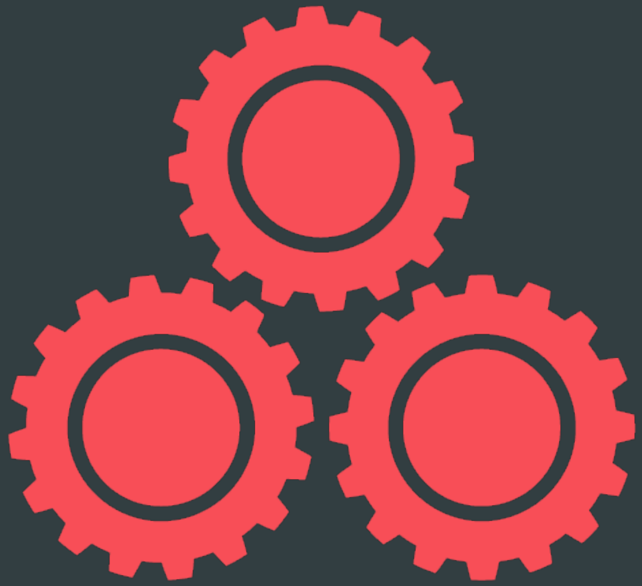
```
struct bluestore_blob_t {
  vector<bluestore_pextent_t> extents;
  uint32_t compressed_length_orig = 0;
  uint32_t compressed_length = 0;
  uint32_t flags = 0;
  uint16_t unused = 0; // bitmap

  uint8_t csum_type = CSUM_NONE;
  uint8_t csum_chunk_order = 0;
  bufferptr csum_data;
};

struct bluestore_shared_blob_t {
  uint64_t sbid;
  bluestore_extent_ref_map_t ref_map;
};
```

EXTENT MAP

- Map object extents → blob extents
- Extent map serialized in chunks
  - stored inline in onode value if small
  - otherwise stored in adjacent keys
- Blobs stored inline in each shard
  - unless it is referenced across shard boundaries
  - "spanning" blobs stored in onode key
- If blob is "shared" (cloned)
  - ref count on allocated extents stored in external key
  - only needed (loaded) on deallocations

```
...
O<,,foo,,>  =onode + inline extent map
O<,,bar,,>  =onode + spanning blobs
O<,,bar,,0> =extent map shard
O<,,bar,,4> =extent map shard
O<,,baz,,>  =onode + inline extent map
...
```





DATA PATH

DATA PATH BASICS

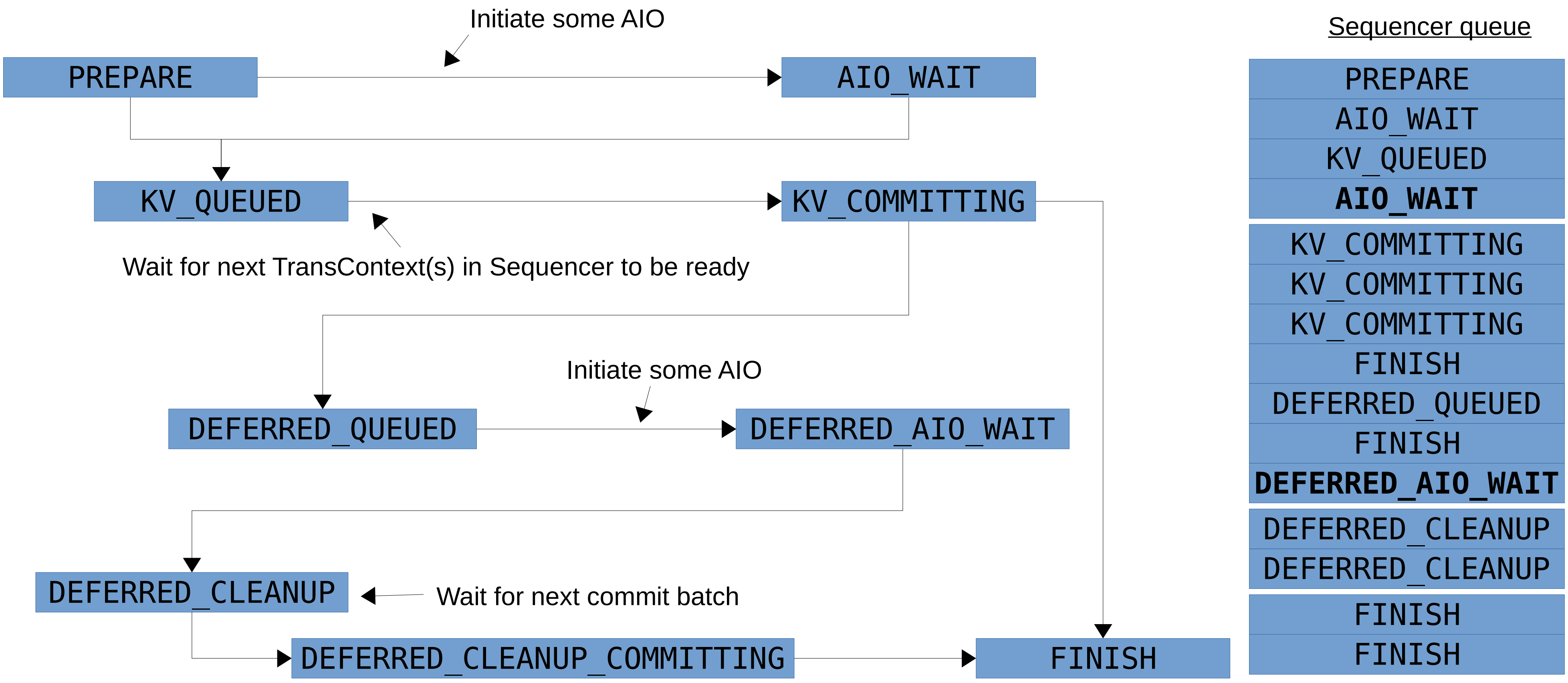
Terms

- Sequencer
  - An independent, totally ordered queue of transactions
  - One per PG
- TransContext
  - State describing an executing transaction

Three ways to write

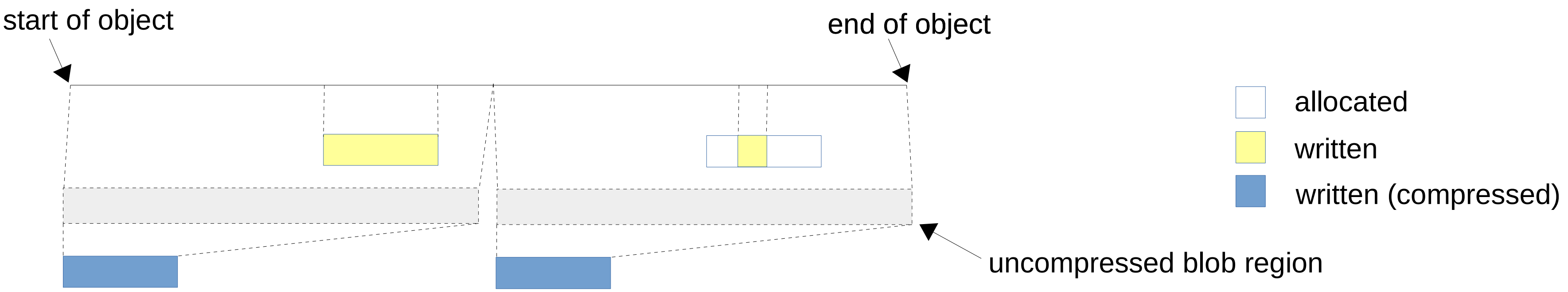
- New allocation
  - Any write larger than **min_alloc_size** goes to a new, unused extent on disk
  - Once that IO completes, we commit the transaction
- Unused part of existing blob
- Deferred writes
  - Commit temporary promise to (over)write data with transaction
    - includes data!
  - Do async (over)write
  - Then clean up temporary k/v pair

TRANSCONTEXT STATE MACHINE



INLINE COMPRESSION

- Blobs can be compressed
  - Tunables target min and max sizes
  - Min size sets ceiling on size reduction
  - Max size caps max read amplification
- Garbage collection to limit occluded/wasted space



  - compacted (rewritten) when waste exceeds threshold

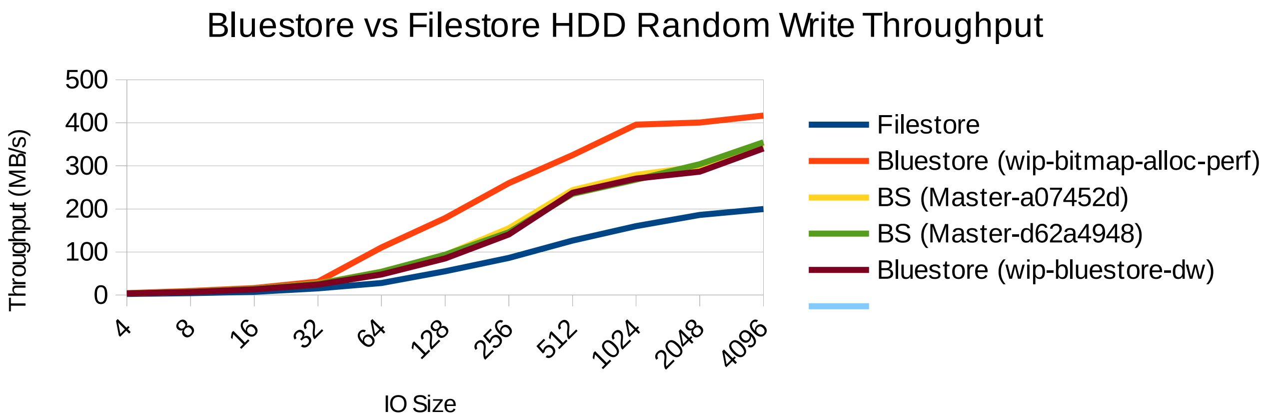
IN-MEMORY CACHE

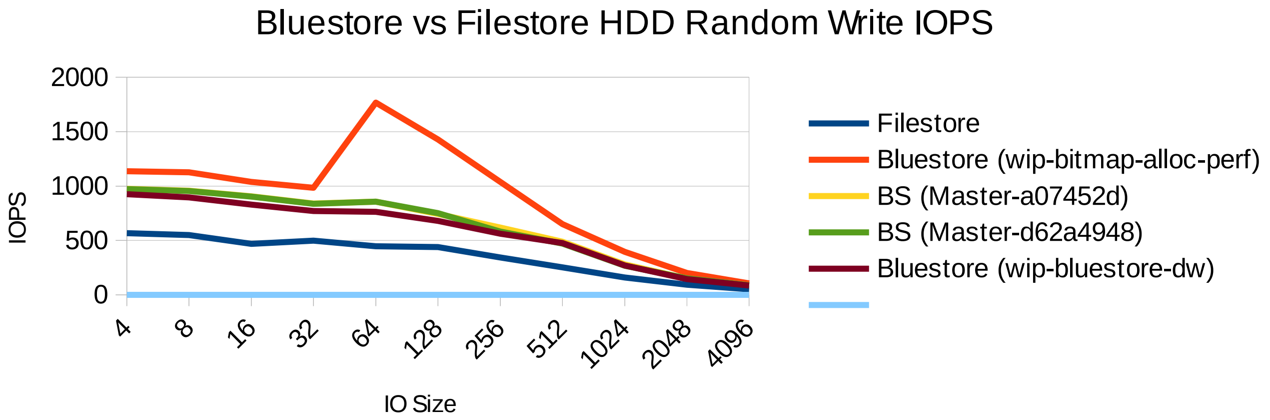
- OnodeSpace per collection
  - in-memory ghobject_t → Onode map of decoded onodes
- BufferSpace for in-memory blobs
  - all in-flight writes
  - may contain cached on-disk data

- Both buffers and onodes have lifecycles linked to a Cache
  - LRUCache – trivial LRU
  - TwoQCache – implements 2Q cache replacement algorithm (default)
- Cache is sharded for parallelism
  - Collection → shard mapping matches OSD's op_wq
  - same CPU context that processes client requests will touch the LRU/2Q lists
  - aio completion execution not yet sharded – TODO?

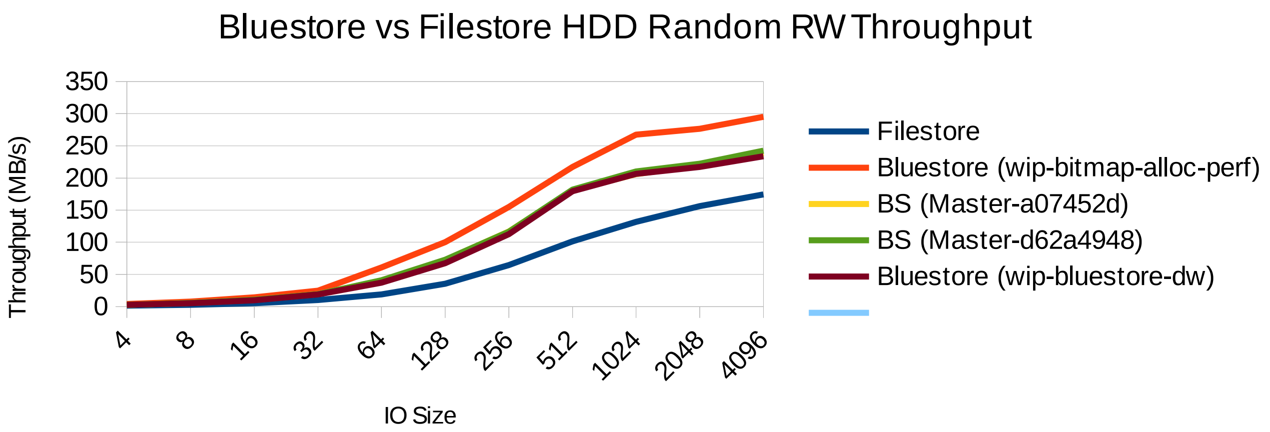
PERFORMANCE

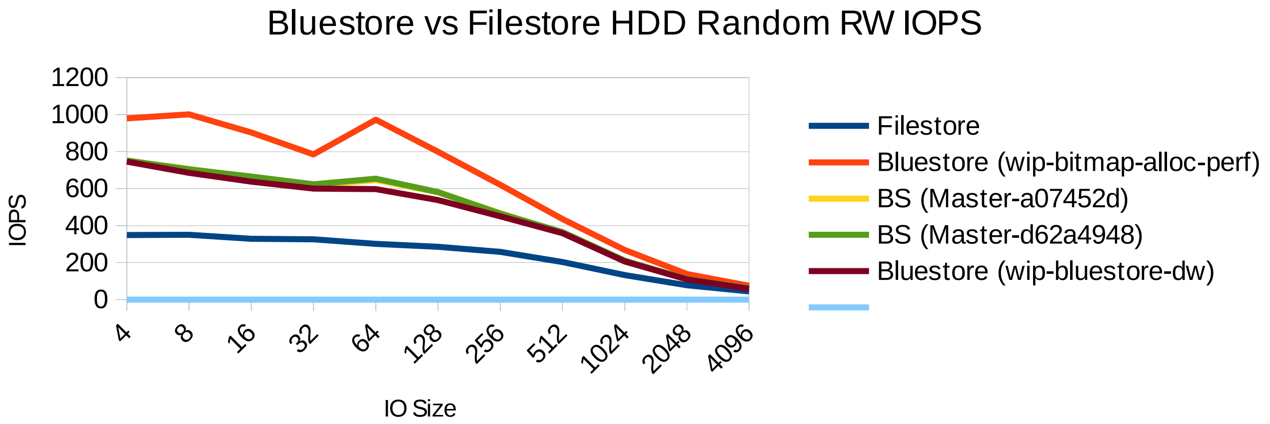
HDD: RANDOM WRITE



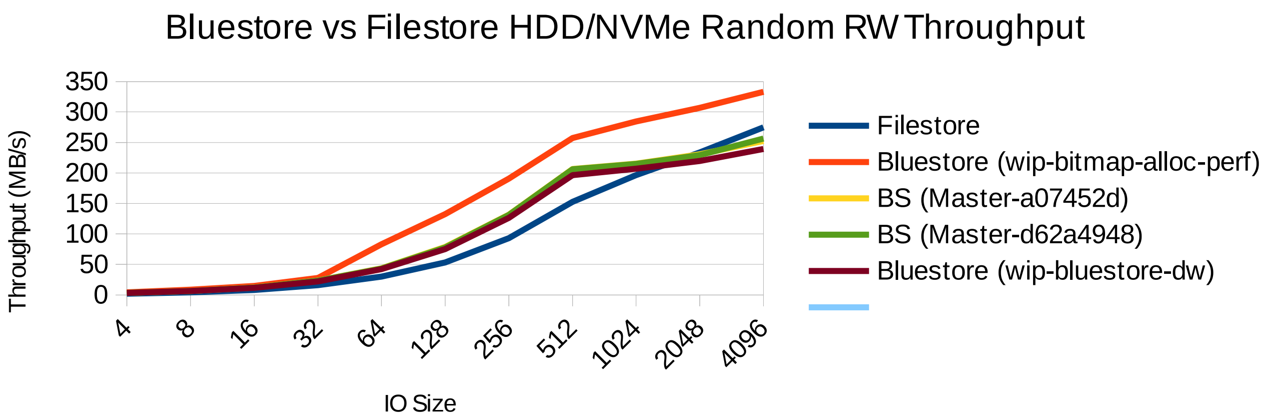


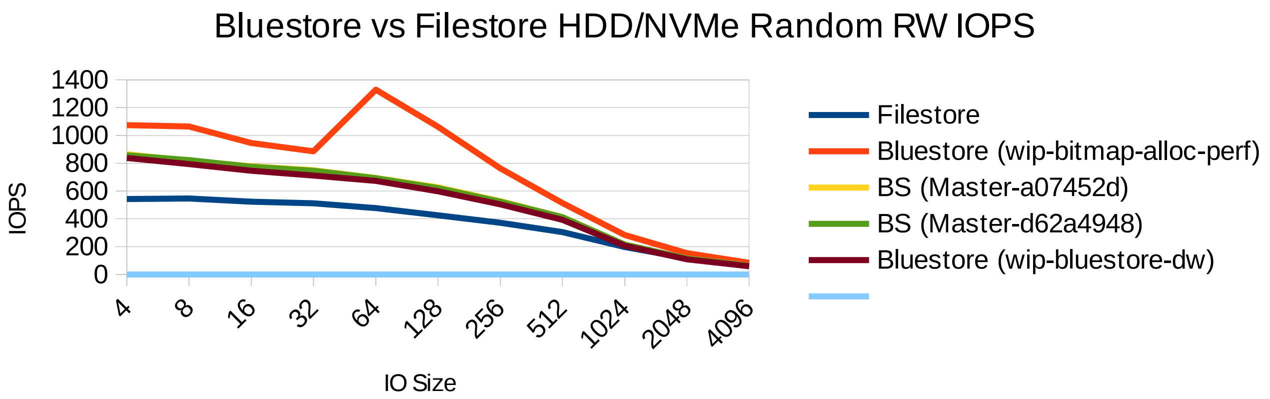
HDD: MIXED READ/WRITE



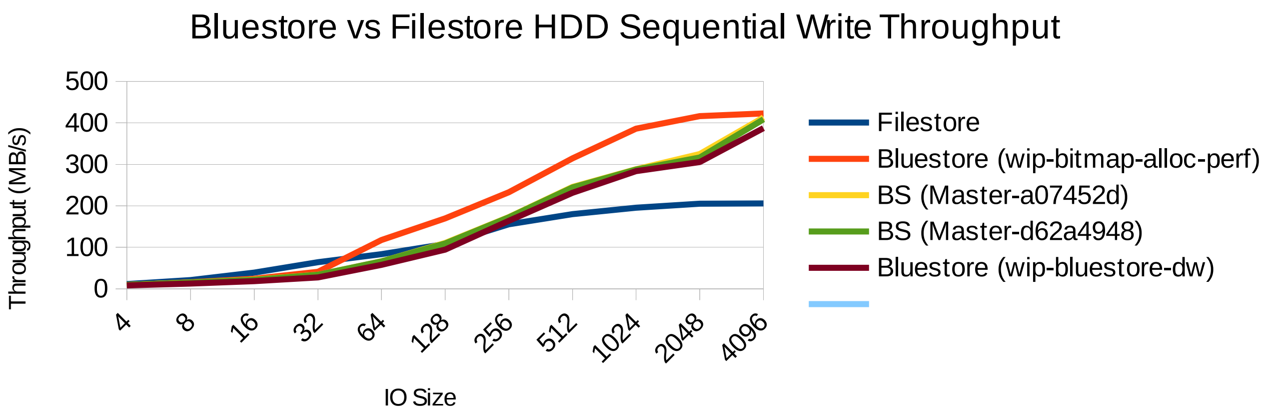


HDD+NVME: MIXED READ/WRITE

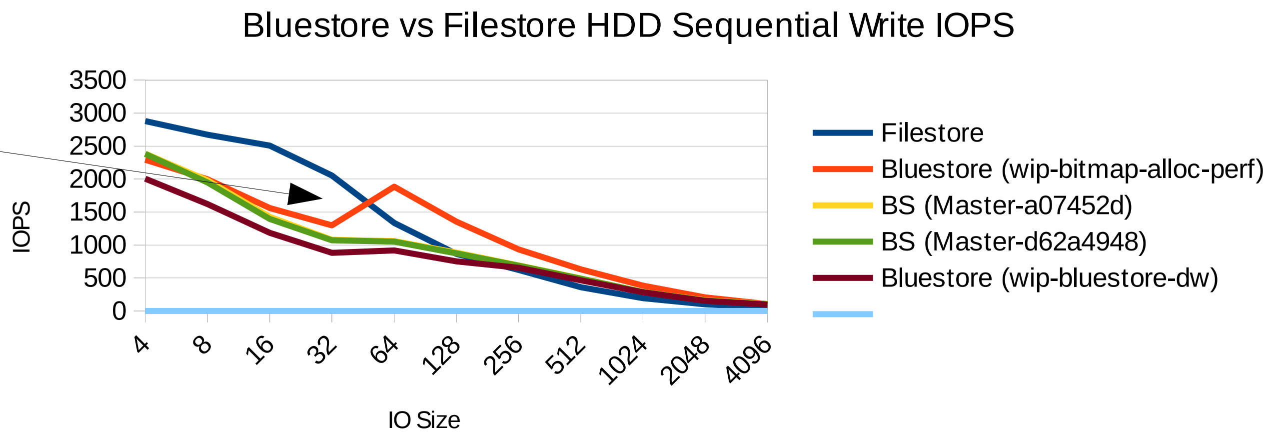




HDD: SEQUENTIAL WRITE



WTF?



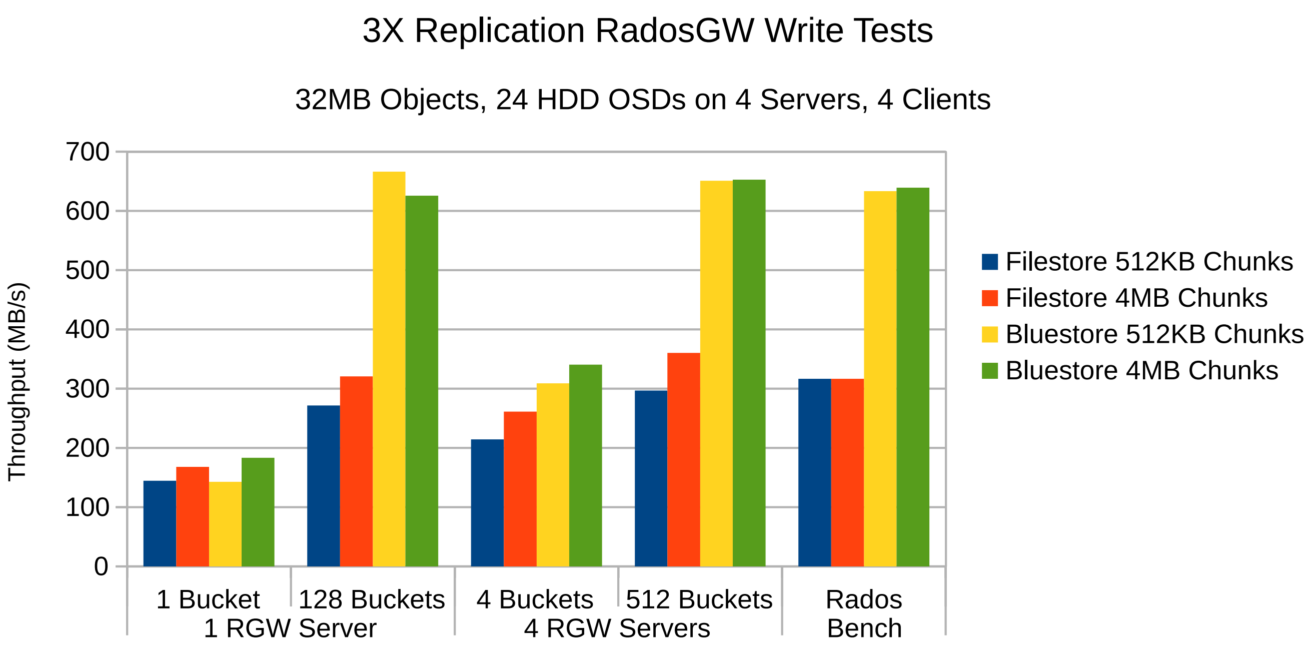
WHEN TO JOURNAL WRITES

- min_alloc_size – smallest allocation unit (16KB on SSD, 64KB on HDD)
  - >= send writes to newly allocated or unwritten space
  - < journal and deferred small overwrites
- Pretty bad for HDDs, especially sequential writes

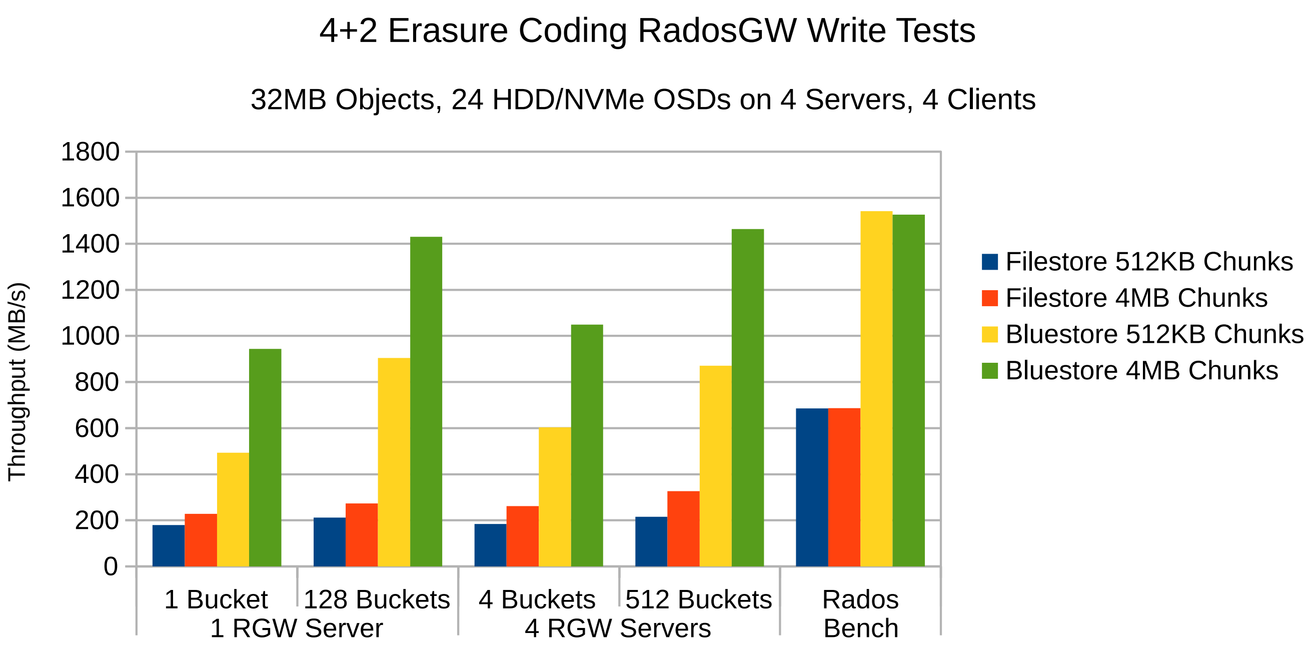
- New tunable threshold for direct vs deferred writes
  - Separate default for HDD and SSD
- Batch deferred writes
  - journal + journal + … + journal + many deferred writes + journal + …

- TODO: plenty more tuning and tweaking here!

RGW ON HDD, 3X REPLICATION



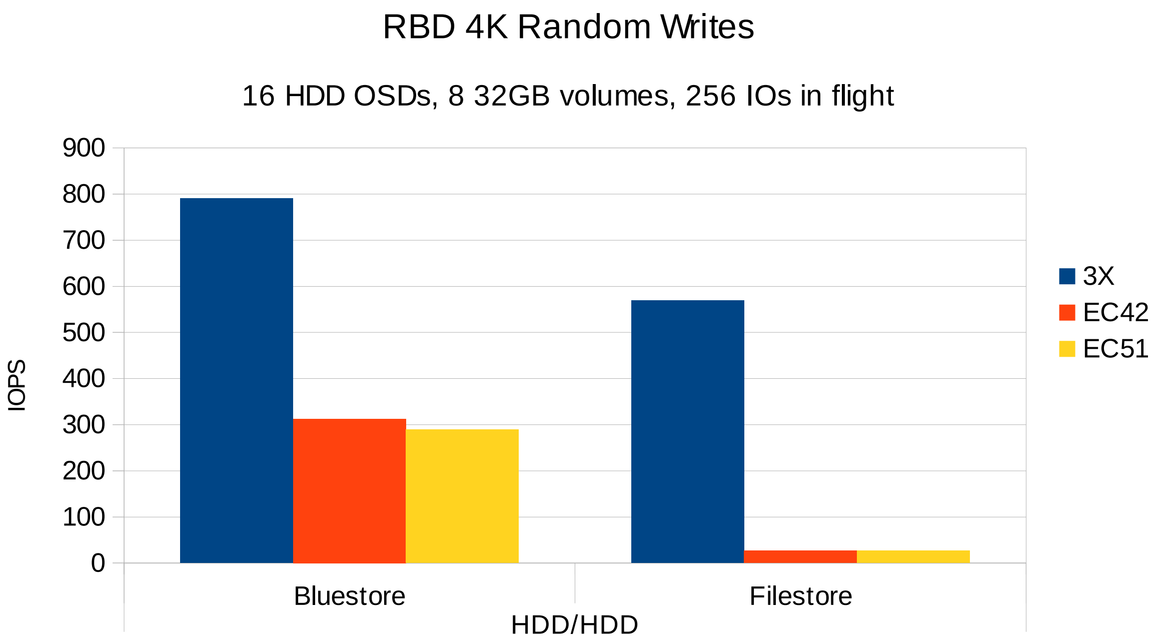
RGW ON HDD+NVME, EC 4+2



ERASURE CODE OVERWRITES

- Luminous allows overwrites of EC objects
  - Requires two-phase commit to avoid “RAID-hole” like failure conditions
  - OSD creates rollback objects
    - clone_range $extent to temporary object
    - write $extent with overwrite data
- clone[_range] marks blobs immutable, creates SharedBlob record
  - Future small overwrites to this blob disallowed; new allocation
  - Overhead of SharedBlob ref-counting record
- TODO: un-share and restore mutability in EC case
  - Either hack since (in general) all references are in cache
  - Or general un-sharing solution (if it doesn’t incur any additional cost)

BLUESTORE vs FILESTORE
3X vs EC 4+2 vs EC 5+1



OTHER CHALLENGES

USERSPACE CACHE

- Built ‘mempool’ accounting infrastructure
  - easily annotate/tag C++ classes and containers
  - low overhead
  - debug mode provides per-type (vs per-pool) accounting (items and bytes)
- All data managed by ‘bufferlist’ type
  - manually track bytes in cache
  - ref-counting behavior can lead to memory use amplification
- Requires configuration
  - bluestore_cache_size (default 1GB)
    - not as convenient as auto-sized kernel caches
  - bluestore_cache_meta_ratio (default .9)
- Finally have meaningful implementation of fadvise NOREUSE (vs DONTNEED)

MEMORY EFFICIENCY

- Careful attention to struct sizes: packing, redundant fields
- Write path changes to reuse and expand existing blobs
  - expand extent list for existing blob
  - big reduction in metadata size, increase in performance
- Checksum chunk sizes
  - client hints to expect sequential read/write → large csum chunks
  - can optionally select weaker checksum (16 or 8 bits per chunk)
- In-memory red/black trees (std::map<>, boost::intrusive::set<>)
  - low temporal write locality → many CPU cache misses, failed prefetches
  - per-onode slab allocators for extent and blob structs

ROCKSDB

- Compaction
  - Awkward to control priority
  - Overall impact grows as total metadata corpus grows
  - Invalidates rocksdb block cache (needed for range queries)
    - we prefer O_DIRECT libaio – workaround by using buffered reads and writes
    - Bluefs write buffer
- Many deferred write keys end up in L0
- High write amplification
  - SSDs with low-cost random reads care more about total write overhead
- Open to alternatives for SSD/NVM
  - ZetaScale (recently open sourced by SanDisk)
  - Or let Facebook et al make RocksDB great?

FUTURE

MORE RUN TO COMPLETION (RTC)

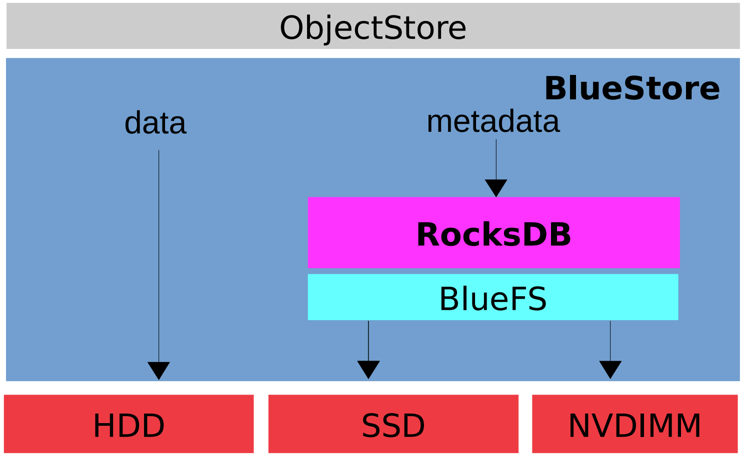
- “Normal” write has several context switches
  - A: prepare transaction, initiate any aio
  - B: io completion handler, queue txc for commit
  - C: commit to kv db
  - D: completion callbacks
- Metadata-only transactions or deferred writes?
  - skip B, do half of C
- Very fast journal devices (e.g., NVDIMM)?
  - do C commit synchronously
- Some completion callbacks back into OSD can be done synchronously
  - avoid D
- Pipeline nature of each Sequencer makes this all opportunistic

SMR

- Described high-level strategy at Vault’16
  - GSoC project
- Recent work shows less-horrible performance on DM-SMR
  - Evolving ext4 for shingled disks (FAST’17, Vault’17)
  - “Keep metadata in journal, avoid random writes”
- Still plan SMR-specific allocator/freelist implementation
  - Tracking released extents in each zone useless
  - Need reverse map to identify remaining objects
  - Clean least-full zones
- Some speculation that this will be good strategy for non-SMR devices too

“TIERING” TODAY

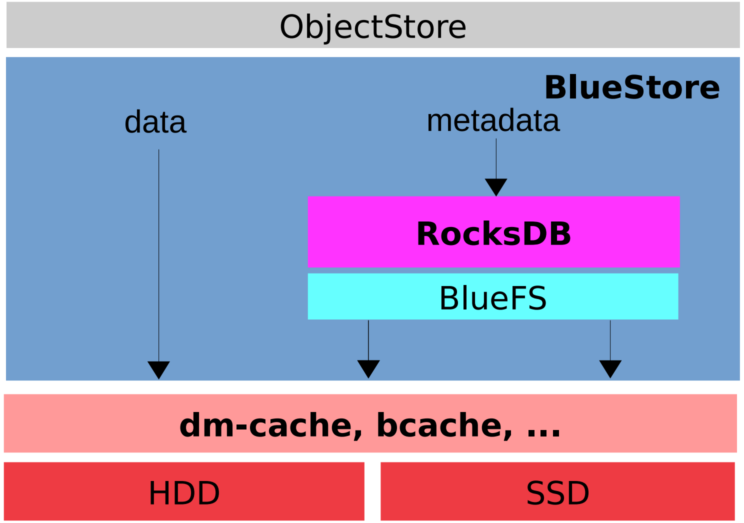
- We do only very basic multi-device
  - WAL, KV, object data
- Not enthusiastic about doing proper tiering within BlueStore
  - Basic multi-device infrastructure not difficult, but
  - Policy and auto-tiering are complex and unbounded



TIERING BELOW!

- Prefer to defer tiering to block layer
  - bcache, dm-cache, FlashCache
- Extend libaio interface to enable hints
  - HOT and COLD flags
  - Add new IO_CMD_PWRITEV2 with a usable flags field
    - and eventually DIF/DIX for passing csum to device?
- Modify bcache, dm-cache, etc to respect hints

- (And above! This is unrelated to RADOS cache tiering and future tiering plans across OSDs)



SPDK – KERNEL BYPASS FOR NVME

- SPDK support is in-tree, but
  - Lack of caching for bluefs/rocksdb
  - DPDK polling thread per OSD not practical
- Ongoing work to allow multiple logical OSDs to coexist in same process
  - Share DPDK infrastructure
  - Share some caches (e.g., OSDMap cache)
  - Multiplex across shared network connections (good for RDMA)
  - DPDK backend for AsyncMessenger
- Blockers
  - msgr2 messenger protocol to allow multiplexing
  - some common shared code cleanup (g_ceph_context)

STATUS

STATUS

- Early prototype in Jewel v10.2.z
  - Very different than current code; no longer useful or interesting
- Stable (code and on-disk format) in Kraken v11.2.z
  - Still marked ‘experimental’
- Stable and recommended default in Luminous v12.2.z (out this Spring)

- Current efforts
  - Workload throttling
  - Performance anomalies
  - Optimizing for CPU time

MIGRATION FROM FILESTORE

- Fail in place
  - Fail FileStore OSD
  - Create new BlueStore OSD on same device

    → period of **reduced redundancy**
- Disk-wise replacement
  - Format new BlueStore on spare disk
  - Stop FileStore OSD on same host
  - Local host copy between OSDs/disks

    → **reduce *online* redundancy**, but data still available offline
  - Requires extra drive slot per host
- Host-wise replacement
  - Provision new host with BlueStore OSDs
  - Swap new host into old hosts CRUSH position

    → **no reduced redundancy** during migration
  - Requires spare host per rack, or extra host migration for each rack

SUMMARY

- Ceph is great at scaling out
- POSIX was poor choice for storing objects
- Our new BlueStore backend is **so** much better
  - Good (and rational) performance!
  - Inline compression and full data checksums
- We are definitely not done yet
  - Performance, performance, performance
  - Other stuff
- We can finally solve our IO problems

BASEMENT CLUSTER

- 2 TB 2.5” HDDs
- 1 TB 2.5” SSDs (SATA)
- 400 GB SSDs (NVMe)

- Kraken 11.2.0
- CephFS
- Cache tiering
- Erasure coding
- BlueStore
- CRUSH device classes

- Untrained IT staff!

THANK YOU!

Sage Weil
CEPH PRINCIPAL ARCHITECT

sage@redhat.com

@liewegas

BLOCK FREE LIST

- FreelistManager
  - persist list of free extents to key/value store
  - prepare incremental updates for allocate or release
- Initial implementation
  - extent-based

    `<offset> = <length>`
  - kept in-memory copy
  - small initial memory footprint, very expensive when fragmented
  - imposed ordering constraint on commits :(
- Newer bitmap-based approach

  `<offset> = <region bitmap>`
  - where region is N blocks
    - 128 blocks = 8 bytes
  - use k/v **merge** operator to XOR allocation or release

    `merge 10=0000000011`

    `merge 20=1110000000`
  - RocksDB log-structured-merge tree coalesces keys during compaction
  - no in-memory state or ordering

BLOCK ALLOCATOR

- Allocator
  - abstract interface to allocate blocks
- StupidAllocator
  - extent-based
  - bin free extents by size (powers of 2)
  - choose sufficiently large extent closest to hint
  - highly variable memory usage
    - btree of free extents
  - implemented, works
  - based on ancient ebofs policy
- BitmapAllocator
  - hierarchy of indexes
    - L1: 2 bits = 2^6 blocks
    - L2: 2 bits = 2^12 blocks
    - ...

      00 = all free, 11 = all used,

      01 = mix
  - fixed memory consumption
    - ~35 MB RAM per TB

CHECKSUMS

- We scrub... periodically
  - window before we detect error
  - we may read bad data
  - we may not be sure which copy is bad
- We want to validate checksum on **every** read

- Blobs include csum metadata
  - crc32c (default), xxhash{64,32}
- Overhead
  - 32-bit csum metadata for 4MB object and 4KB blocks = 4KB
  - larger csum blocks (compression!)
  - smaller csums
    - crc32c_8 or 16
- IO hints
  - seq read + write → big chunks
  - compression → big chunks
- Per-pool policy