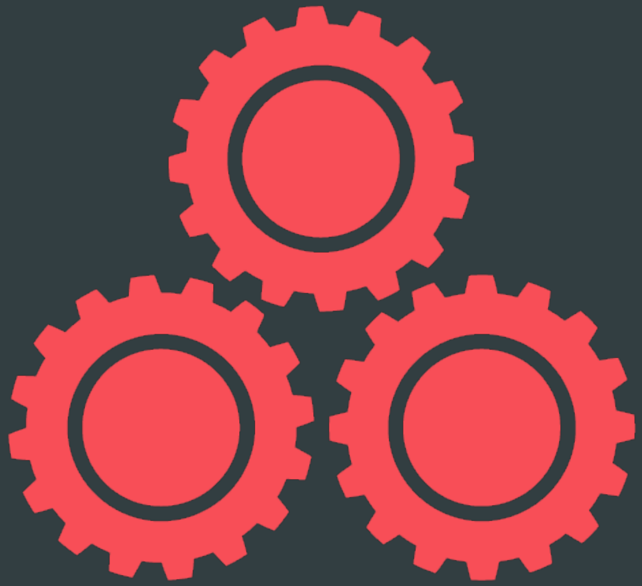
# GOODBYE, XFS: BUILDING A NEW, FASTER STORAGE BACKEND FOR CEPH

SAGE WEIL – RED HAT

2017.09.12

# OUTLINE

- Ceph background and context
  - FileStore, and why POSIX failed us
- BlueStore – a new Ceph OSD backend
- Performance
- Recent challenges
- Current status, future
- Summary

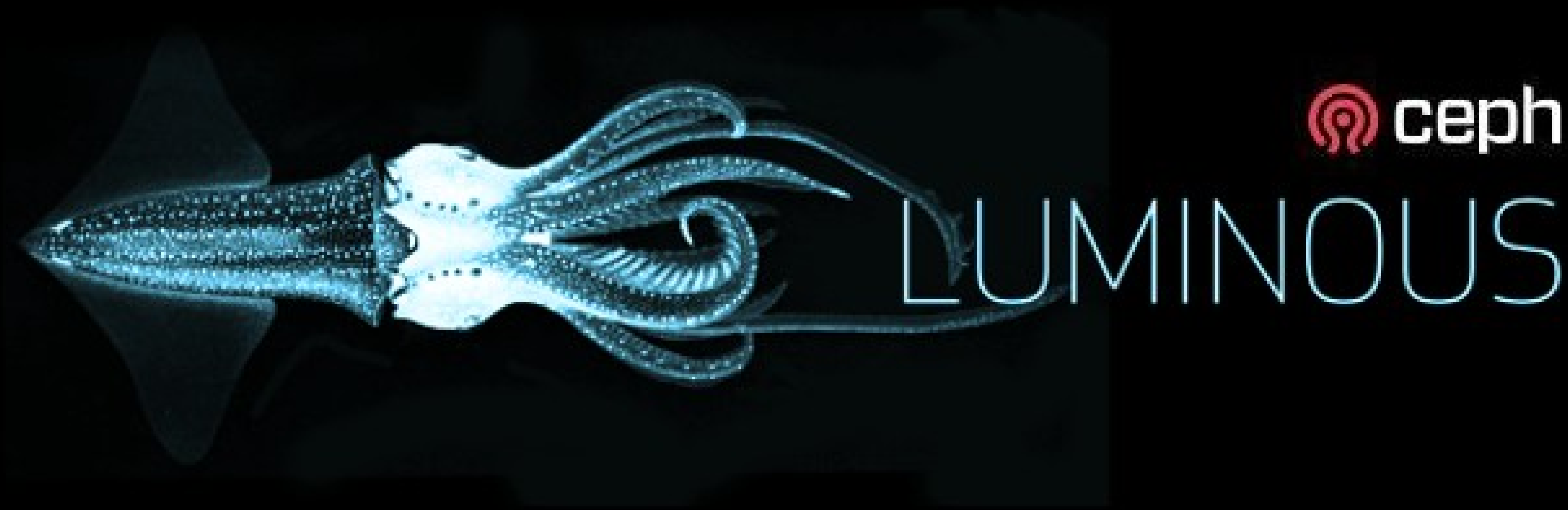


# MOTIVATION

# 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”




- Released two weeks ago
- Erasure coding support for block and file (and object)
- BlueStore (our new storage backend)

# CEPH COMPONENTS

OBJECT

BLOCK

FILE

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

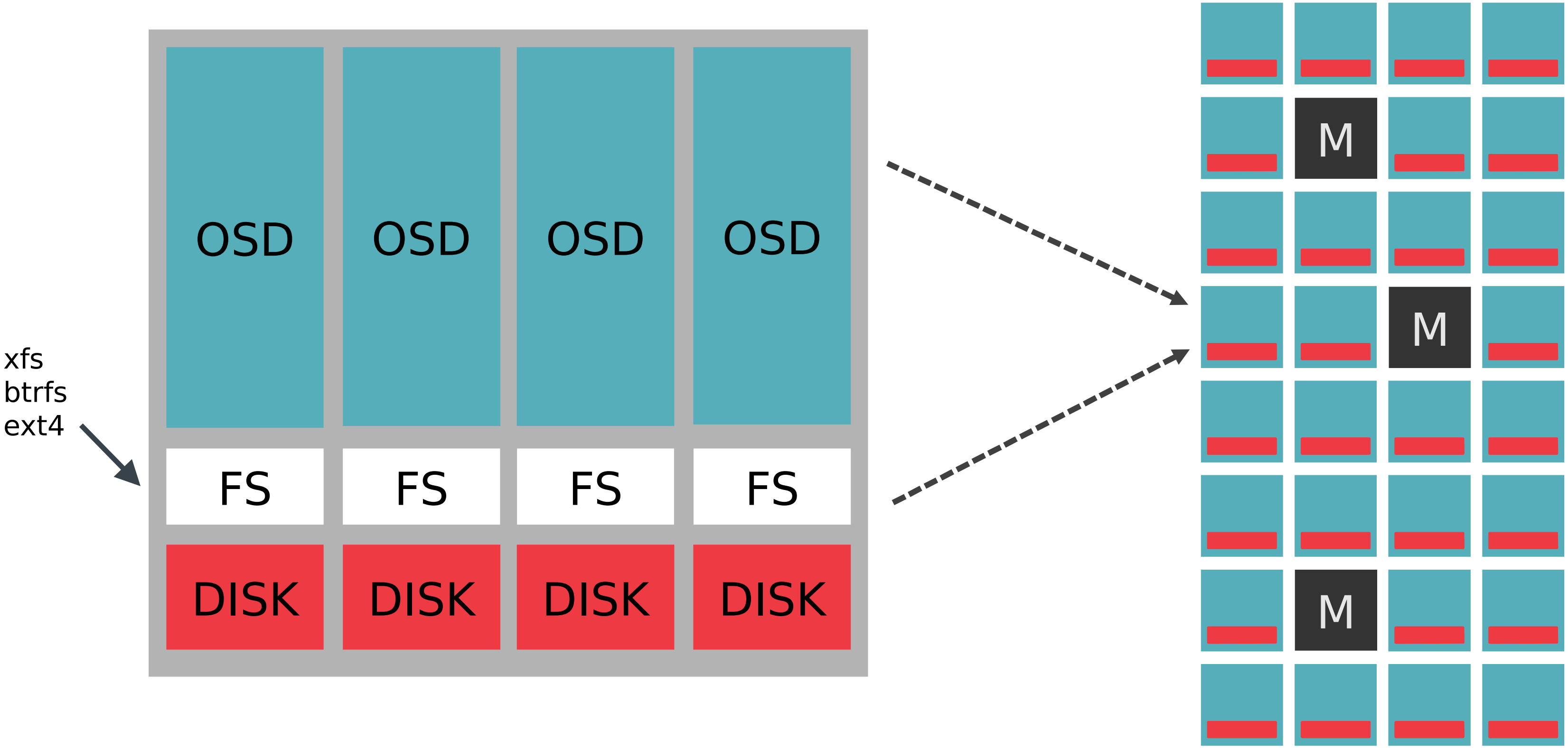
**RBD**
A reliable, fully-distributed block device with cloud platform integration

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

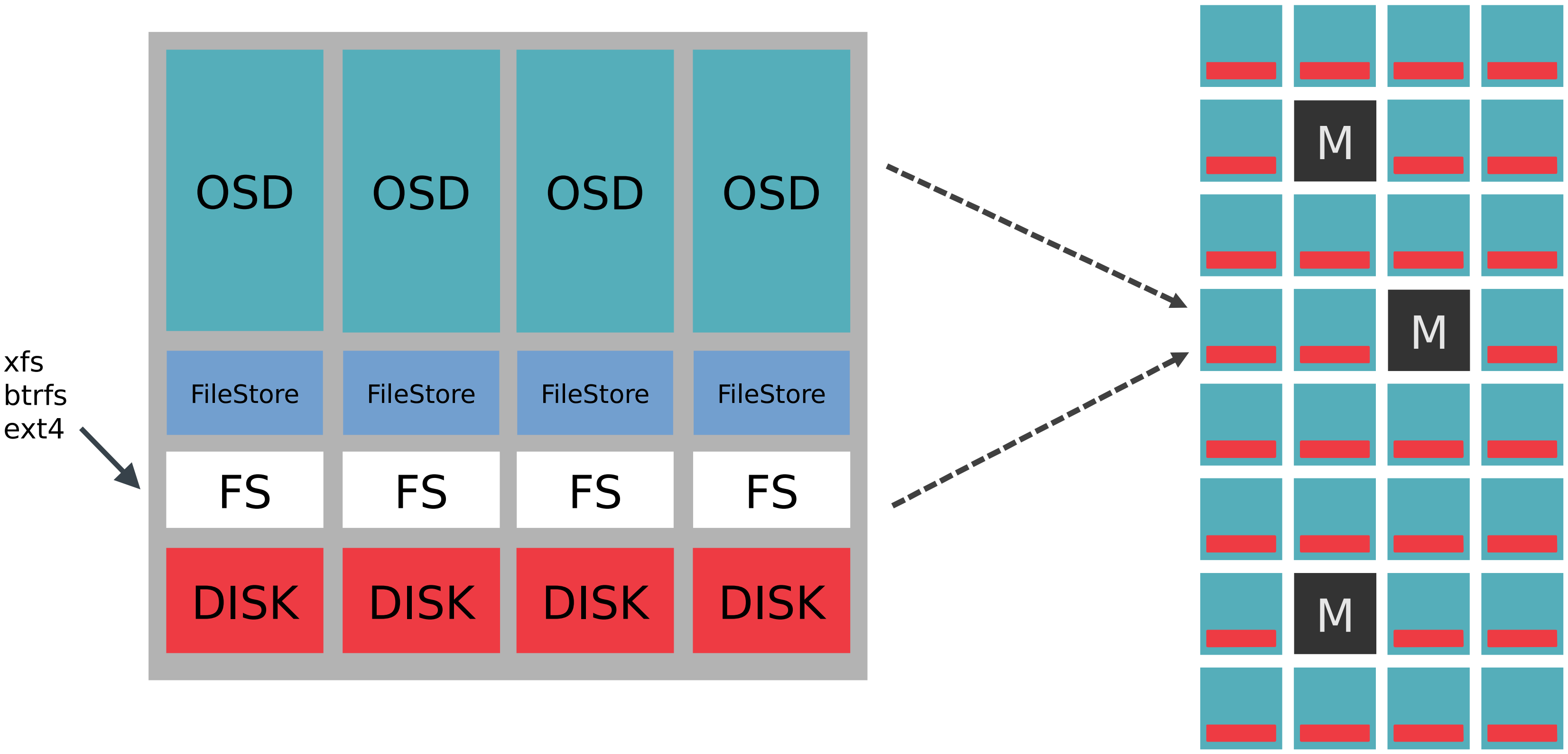
**LIBRADOS**
A library allowing apps to directly access RADOS (C, C++, Java, Python, Ruby, PHP)

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

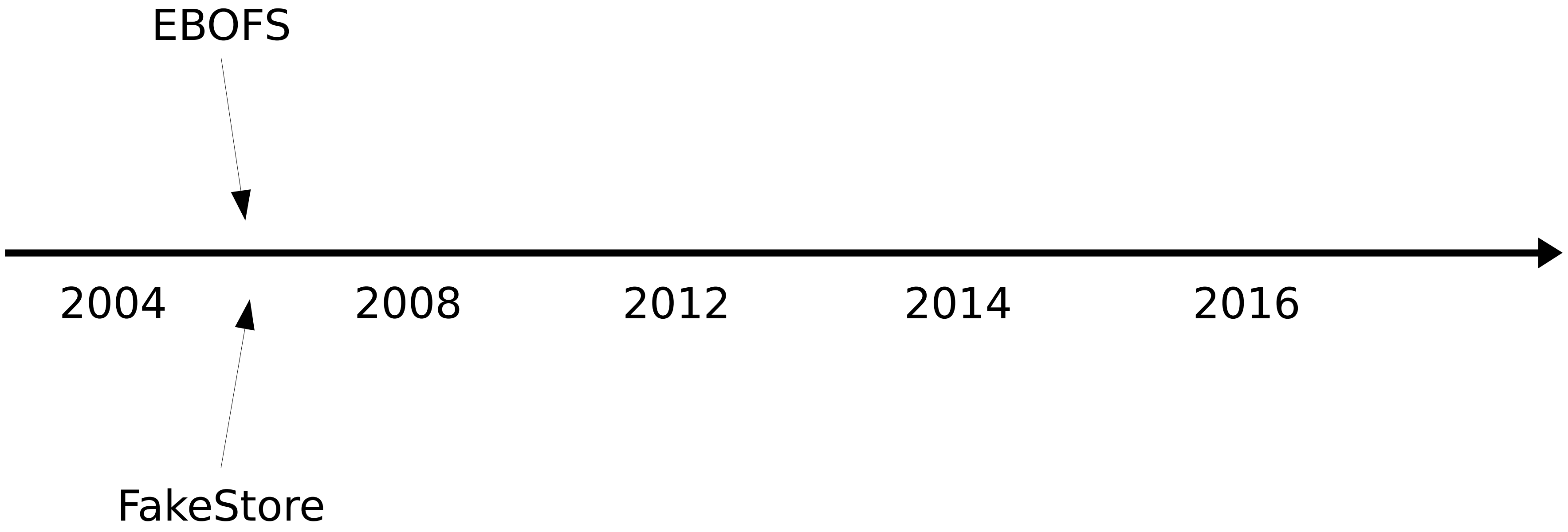
# OBJECT STORAGE DAEMONS (OSDS)



# OBJECT STORAGE DAEMONS (OSDS)



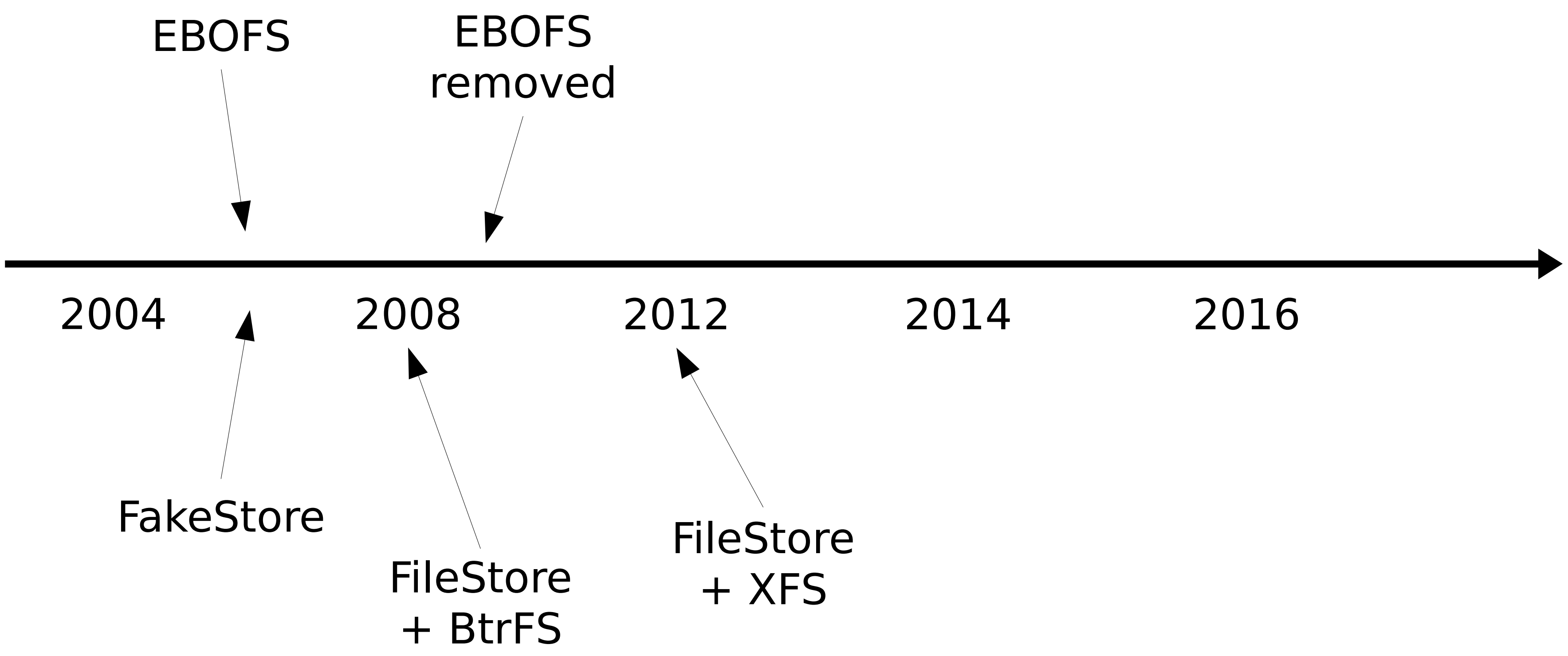
# IN THE BEGINNING



# OBJECTSTORE AND DATA MODEL

- ObjectStore
  - abstract interface for storing local data
  - EBOFS, FakeStore
- Object – “file”
  - data (file-like byte stream)
  - attributes (small key/value)
  - omap (unbounded key/value)
- Collection – “directory”
  - actually a shard of RADOS pool
  - Ceph placement group (PG)
- All writes are transactions
  - **A**tomic + **C**onsistent + **D**urable
  - **I**solation provided by OSD

# LEVERAGE KERNEL FILESYSTEMS!



# FILESTORE

- FileStore
  - evolution of FakeStore
  - PG = collection = directory
  - object = file
- Leveldb
  - large xattr spillover
  - object omap (key/value) data

- `/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: KERNEL CACHE

- Kernel cache is mostly wonderful
  - automatically sized to all available memory
  - automatically shrinks if memory needed
  - efficient
- No cache implemented in app, yay!

- Read/modify/write vs write-ahead
  - write must be persisted
  - then applied to file system
  - only then you can read it

# POSIX FAILS: ENUMERATION

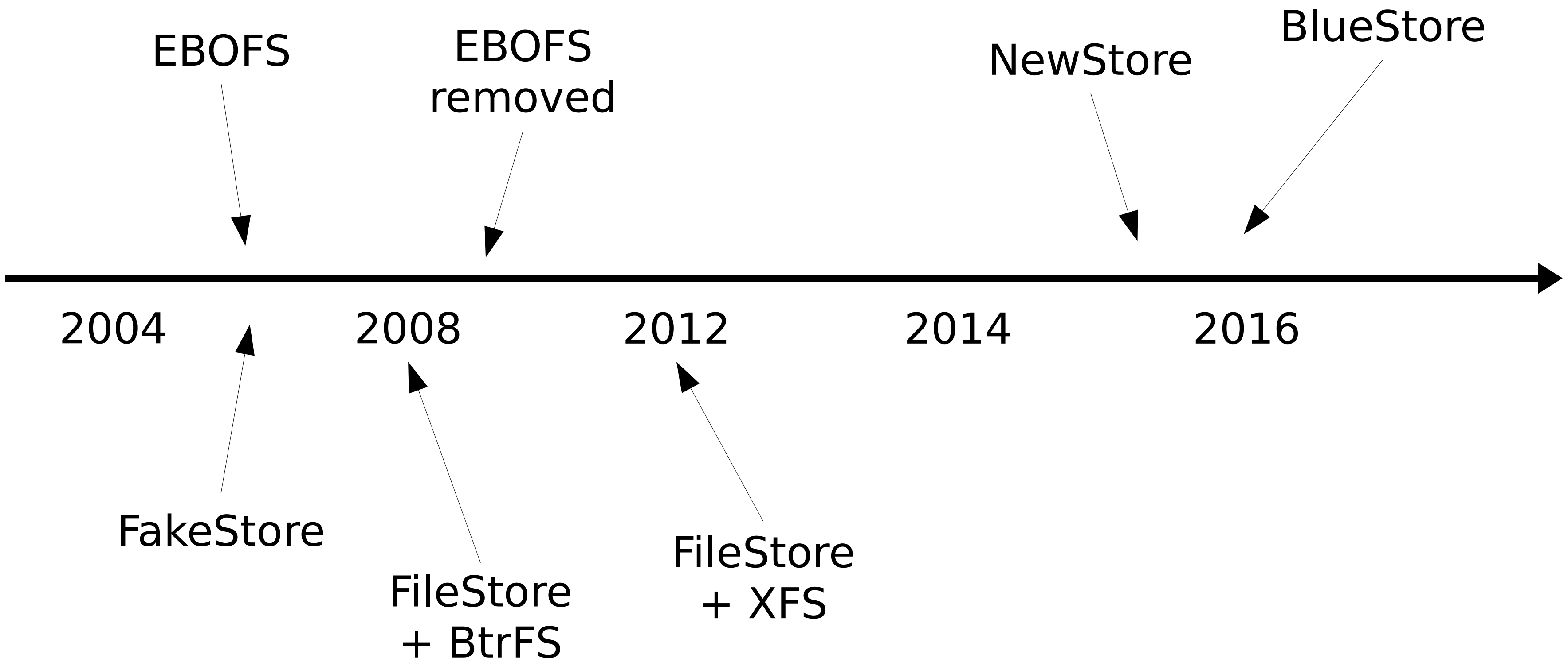
- Ceph objects are distributed by a 32-bit hash
- Enumeration is in hash order
  - scrubbing
  - data rebalancing, recovery
  - enumeration via librados client API
- POSIX readdir is not well-ordered
- Need O(1) “split” for a given shard/range
- Build directory tree by hash-value prefix
  - split big directories, merge small ones
  - 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+XFS problems continue to surface
  - FileStore directory splits lead to throughput collapse when an entire pool's PG directories split in unison
  - Cannot bound deferred writeback work, even with fsync(2)
  - QoS efforts thwarted by deep queues and periodicity in FileStore throughput
  - {RBD, CephFS} snapshots triggering inefficient 4MB object copies to create object clones

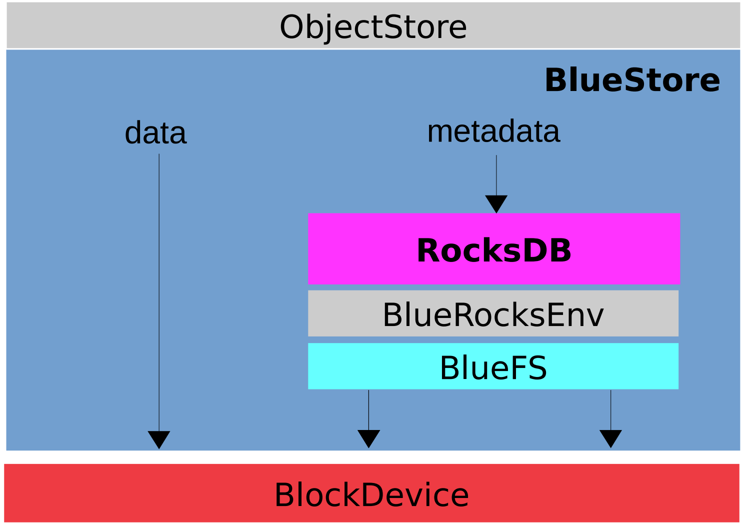
# ACTUALLY, OUR FIRST PLAN WAS BETTER



# BLUESTORE

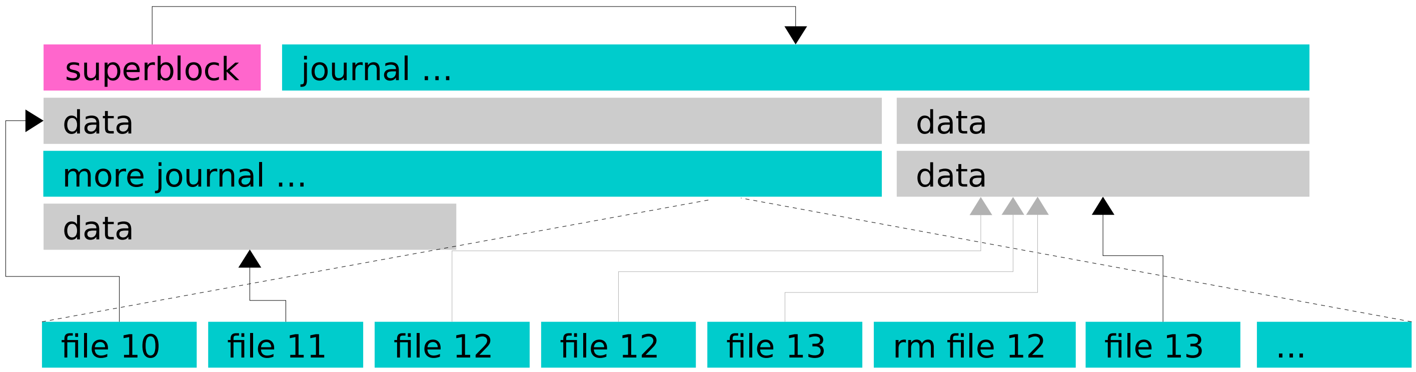
# BLUESTORE

- BlueStore = **Bl**ock + **NewStore**
  - consume raw block device(s)
  - key/value database (RocksDB) for metadata
  - data written directly to block device
  - pluggable inline compression
  - full data checksums
- Target hardware
  - Current gen HDD, SSD (SATA, SAS, NVMe)
- 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
  - journal rewritten/compacted when it gets large
  - no need to store block free list
- 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



# METADATA

# ONODE – OBJECT

- 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;

  // extent map metadata
  struct shard_info {
    uint32_t offset;
    uint32_t bytes;
  };
  vector<shard_info> shards;

  bufferlist inline_extents;
  bufferlist spanning_blobs;
};
```

# CNODE – COLLECTION

- Collection metadata
  - Interval of object namespace

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


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

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

struct bluestore_cnode_t {
  uint32_t bits;
};
```

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

# BLOB AND SHAREDBLOB

- Blob
  - Extent(s) on device
  - Checksum metadata
  - Data may be compressed

```
struct bluestore_blob_t {
  uint32_t flags = 0;

  vector<bluestore_pextent_t> extents;
  uint16_t unused = 0; // bitmap

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

  uint32_t compressed_length_orig = 0;
  uint32_t compressed_length = 0;
};
```

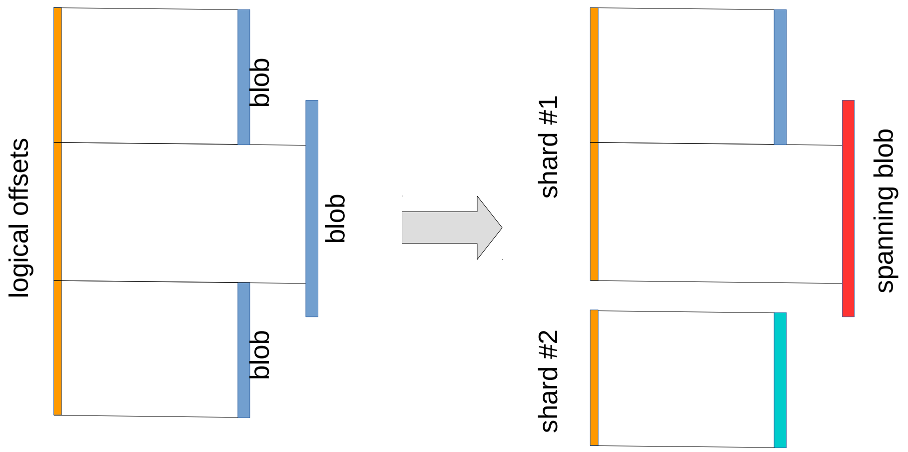
- SharedBlob
  - Extent ref count on cloned blobs

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

# EXTENT MAP

- Map object extents → blob extents
- 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

```
...
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

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

## Three ways to write

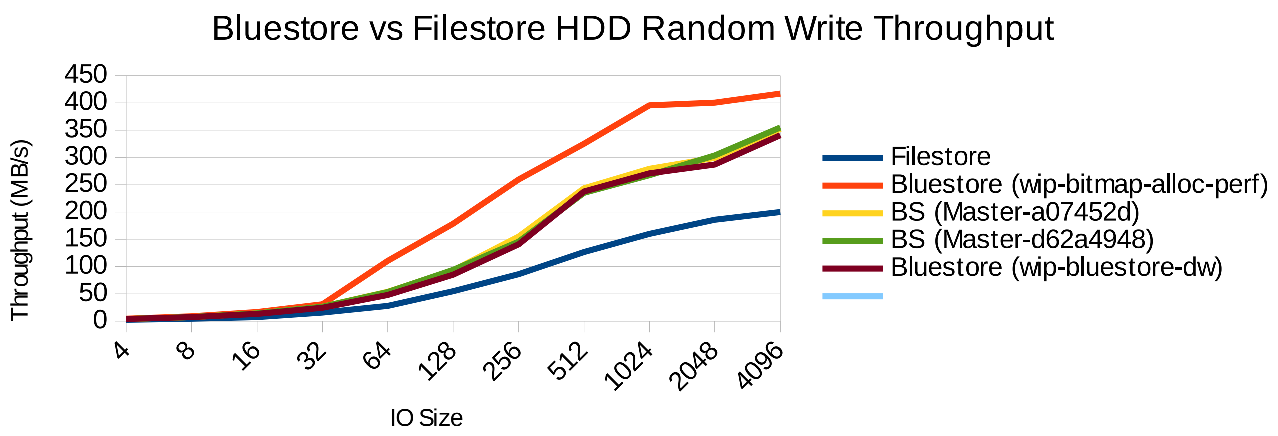
- New allocation
  - Any write larger than **min_alloc_size** goes to a new, unused extent on disk
  - Once that IO completes, we commit the transaction
- Unused part of existing blob
- Deferred writes
  - Commit temporary promise to (over)write data with transaction
    - includes data!
  - Do async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

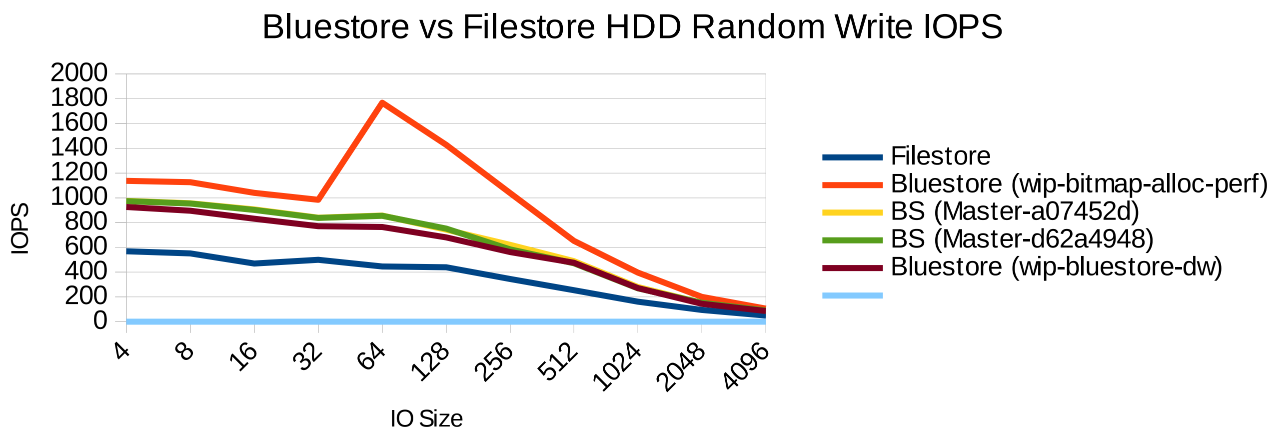
# IN-MEMORY CACHE

- *OnodeSpace* per collection
  - in-memory name → *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*
  - *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

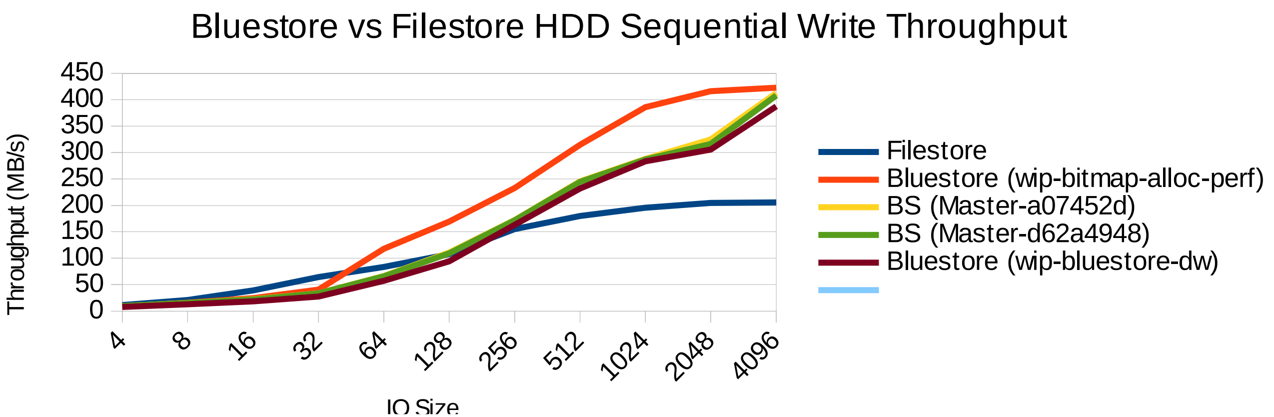
# PERFORMANCE

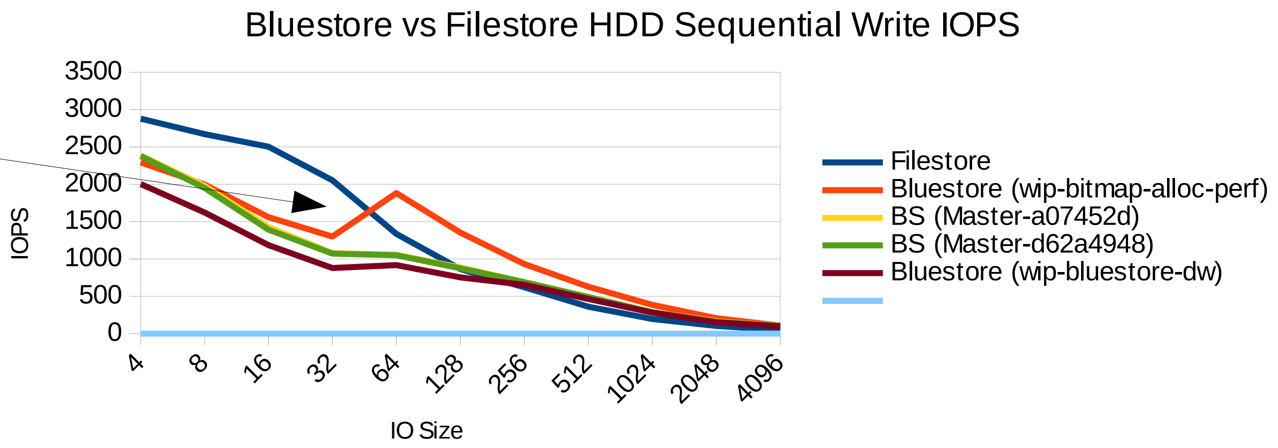
# HDD: RANDOM WRITE



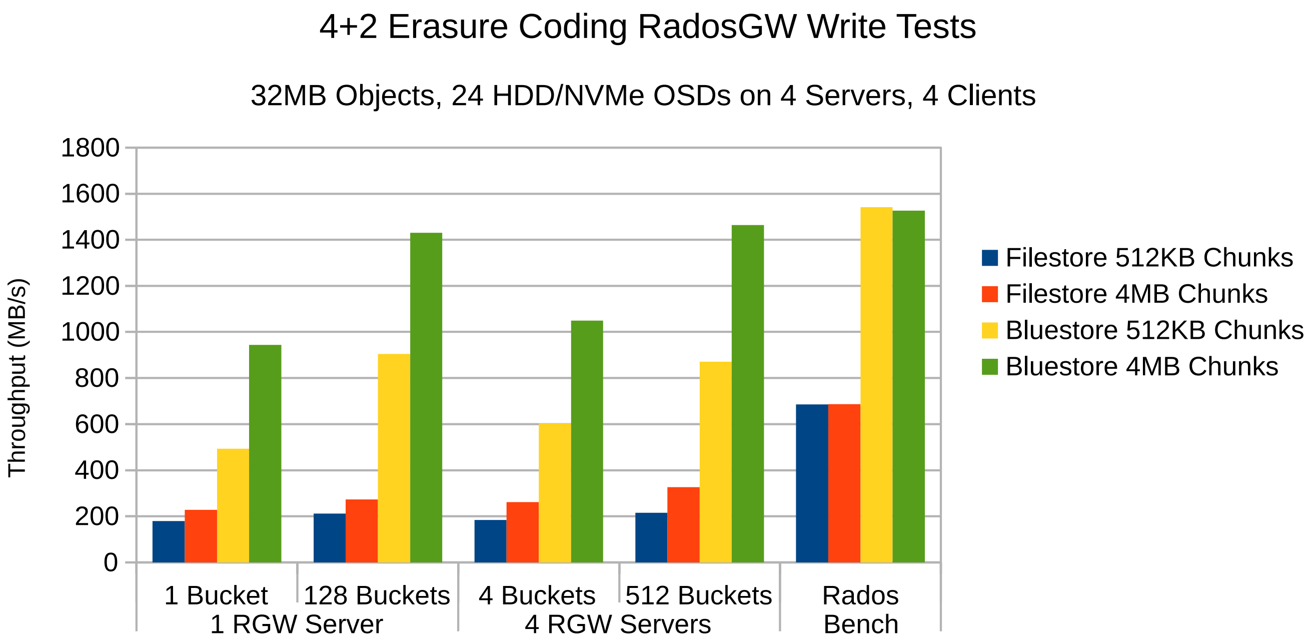
# HDD: SEQUENTIAL WRITE




WTH

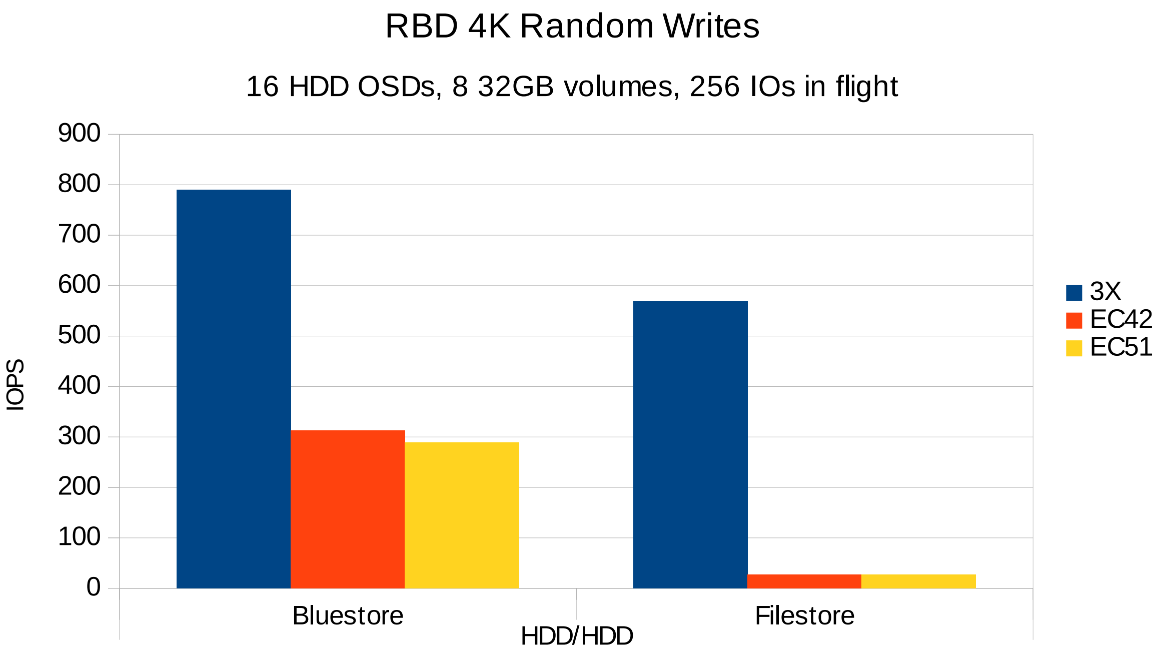
# RGW ON HDD+NVME, EC 4+2



# ERASURE CODE OVERWRITES

- Luminous allows overwrites of EC objects
  - Requires two-phase commit to avoid “RAID-hole” like failure conditions
  - OSD creates temporary rollback objects
    - clone_range $extent to temporary object
    - overwrite $extent with new data
- BlueStore can do this easily…
- FileStore literally copies (reads and writes to-be-overwritten region)

# 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
- Requires configuration
  - bluestore_cache_size (default 1GB)
    - not as convenient as auto-sized kernel caches
- Finally have meaningful implementation of fadvise NOREUSE (vs DONTNEED)

# MEMORY EFFICIENCY

- Careful attention to struct sizes: packing, redundant fields
- 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 (e.g., std::map<>) bad for CPUs
  - low temporal write locality → many CPU cache misses, failed prefetches
  - use btrees where appropriate
  - per-onode slab allocators for extent and blob structs

# ROCKSDB

- Compaction
  - Overall impact grows as total metadata corpus grows
  - Invalidates rocksdb block cache (needed for range queries)
  - Awkward to control priority on kernel libaio interface
- 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

STATUS

# STATUS

- Stable and recommended default in Luminous v12.2.z (just released!)
- Migration from FileStore
  - Reprovision individual OSDs, let cluster heal/recover

- Current efforts
  - Optimizing for CPU time
  - SPDK support!
  - Adding some repair functionality to fsck

# FUTURE

# FUTURE WORK

- Tiering
  - hints to underlying block-based tiering device (e.g., dm-cache, bcache)?
  - implement directly in BlueStore?
- host-managed SMR?
  - maybe...

- Persistent memory + 3D NAND future?
- NVMe extensions for key/value, blob storage?

# 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
- Designing internal storage interfaces without a POSIX bias is a win...

  ...eventually
- We are definitely not done yet
  - Performance!
  - Other stuff
- We can finally solve our IO problems ourselves

# BASEMENT CLUSTER

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

- Luminous 12.2.0
- CephFS
- Cache tiering
- Erasure coding
- BlueStore

- Untrained IT staff!

# THANK YOU!

Sage Weil

CEPH PRINCIPAL ARCHITECT

sage@redhat.com

@liewegas