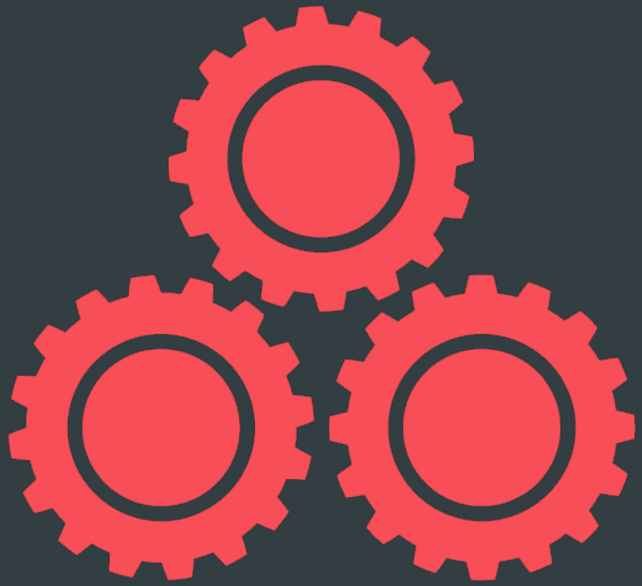
BLUESTORE: A NEW, FASTER STORAGE BACKEND FOR CEPH

SAGE WEIL

VAULT – 2016.04.21

OUTLINE

- Ceph background and context
  - FileStore, and why POSIX failed us
  - NewStore – a hybrid approach
- BlueStore – a new Ceph OSD backend
  - Metadata
  - Data
- Performance
- Upcoming changes
- 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

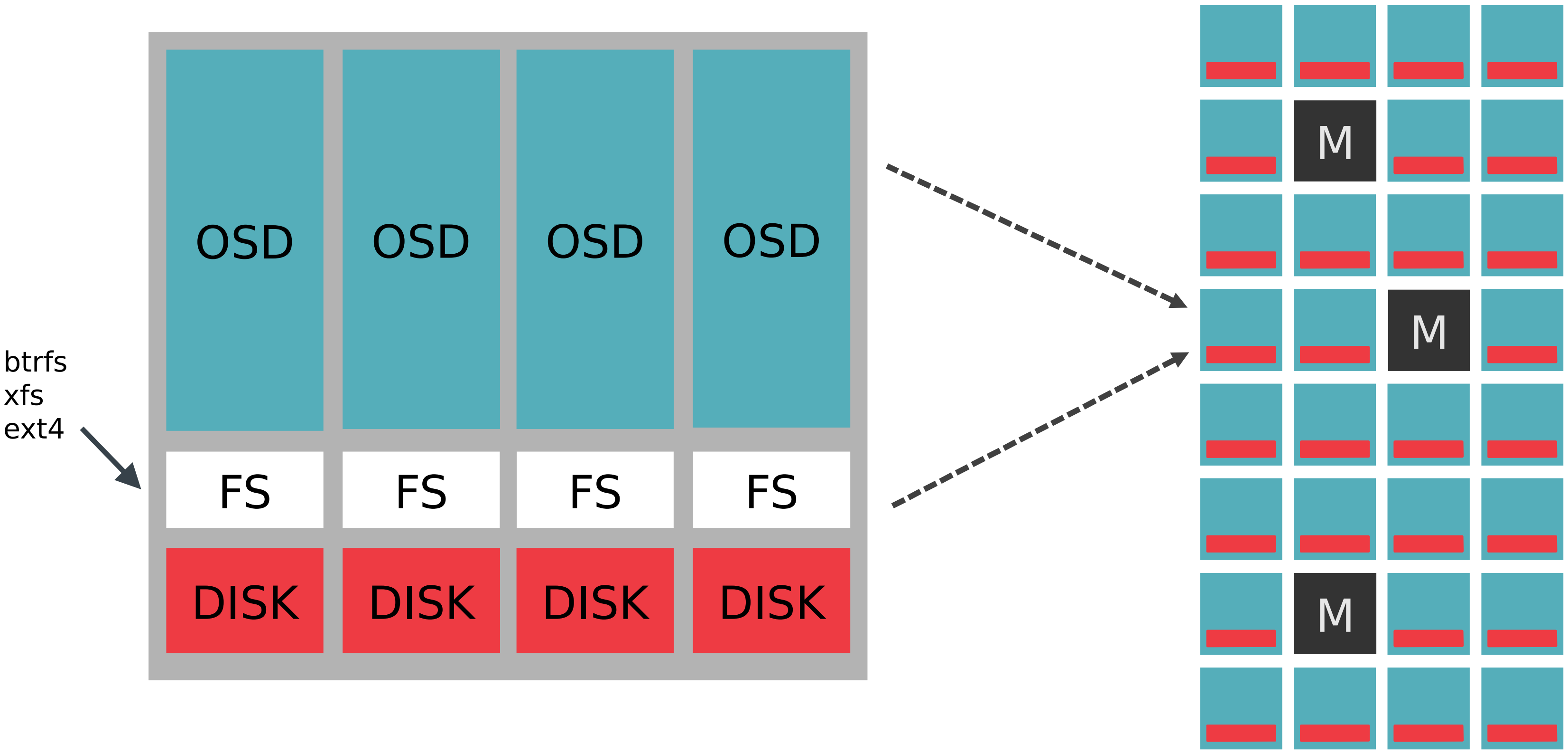
RBD
A reliable, fully-distributed block device with cloud platform integration

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

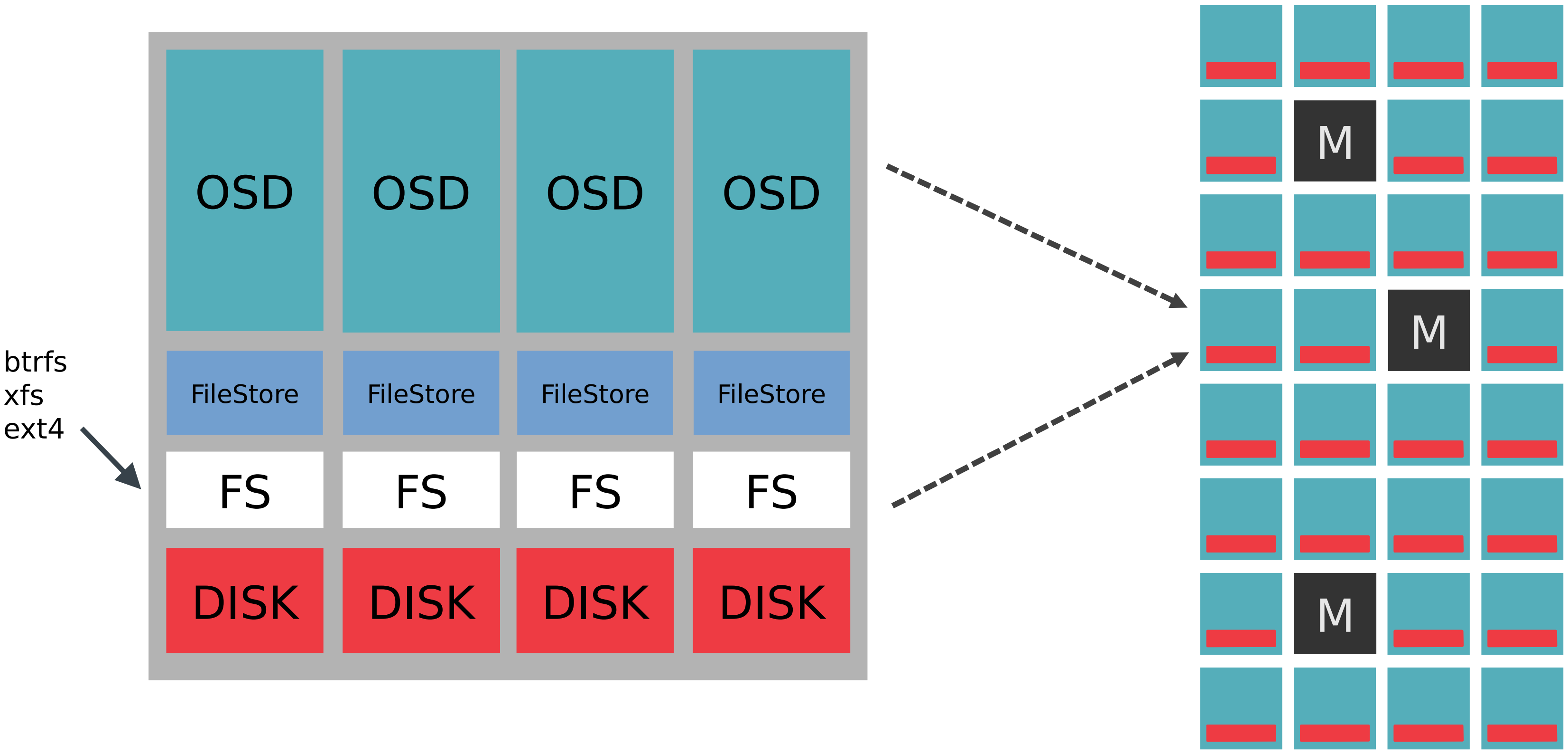
LIBRADOS
A library allowing apps to directly access RADOS (C, C++, Java, Python, Ruby, PHP)

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
  - data (file-like byte stream)
  - attributes (small key/value)
  - omap (unbounded key/value)
- Collection
  - placement group shard (slice of the RADOS pool)
  - sharded by 32-bit hash value
- 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
    - db/
      - <leveldb files>

POSIX FAILS: TRANSACTIONS

- OSD carefully manages consistency of its data
- All writes are transactions
  - we need A+C+D; OSD provides I
- Most are simple
  - write some bytes to object (file)
  - update object attribute (file xattr)
  - append to update log (leveldb insert)

...but others are arbitrarily large/complex


```
[
    {
        "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: TRANSACTIONS

- Btrfs transaction hooks

```
/* trans start and trans end are dangerous, and only for
 * use by applications that know how to avoid the
 * resulting deadlocks
 */
#define BTRFS_IOC_TRANS_START  _IO(BTRFS_IOCTL_MAGIC, 6)
#define BTRFS_IOC_TRANS_END    _IO(BTRFS_IOCTL_MAGIC, 7)
```

- Writeback ordering

```
#define BTRFS_MOUNT_FLUSHONCOMMIT       (1 << 7)
```

- What if we hit an error? ceph-osd process dies?

```
#define BTRFS_MOUNT_WEDGEONTRANSABORT   (1 << …)
```

  – There is no rollback...

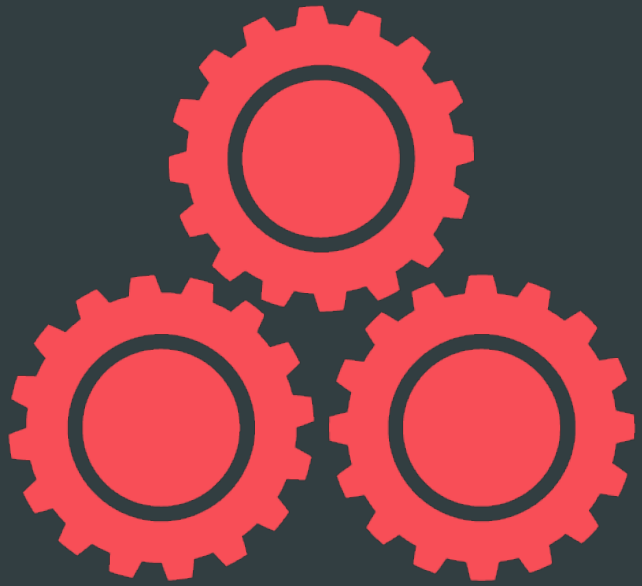
POSIX FAILS: TRANSACTIONS

- Write-ahead journal
  - serialize and journal every ObjectStore::Transaction
  - then write it to the file system
- Btrfs parallel journaling
  - periodic sync takes a snapshot
  - on restart, rollback, and replay journal against appropriate snapshot
- XFS/ext4 write-ahead journaling
  - periodic sync, then trim old journal entries
  - on restart, replay entire journal
  - lots of ugly hackery to deal with events that aren't idempotent
    - e.g., renames, collection delete + create, …
- **full data journal → we double write everything → ~halve disk throughput**

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
- 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
…
```



NEWSTORE

NEW STORE GOALS

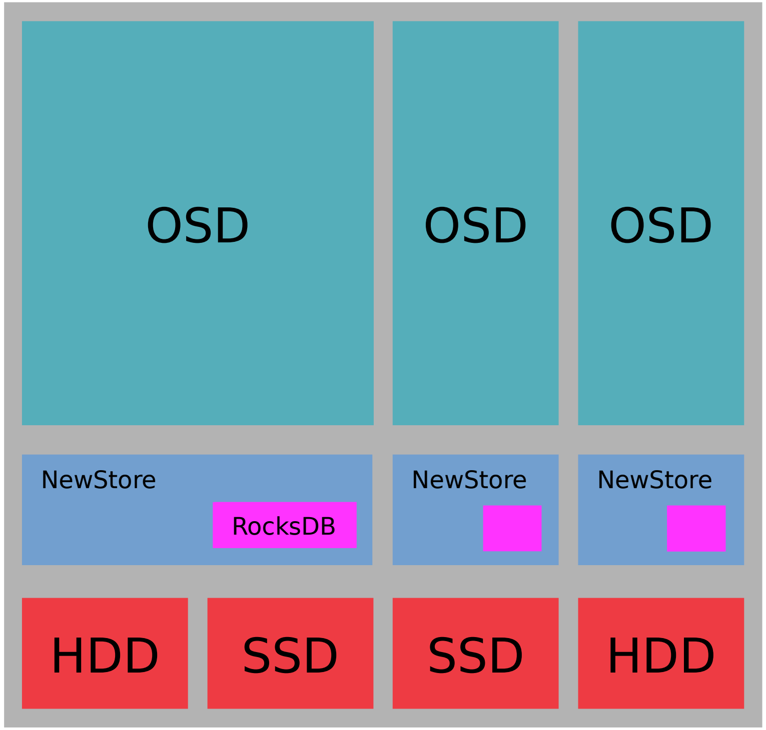
- More natural transaction atomicity
- Avoid double writes
- Efficient object enumeration
- Efficient clone operation
- Efficient splice (“move these bytes from object X to object Y”)

- Efficient IO pattern for HDDs, SSDs, NVMe
- Minimal locking, maximum parallelism (between PGs)

- Advanced features
  - full data and metadata checksums
  - compression

NEWSTORE – WE MANAGE NAMESPACE

- POSIX has the wrong metadata model for us
- Ordered key/value is perfect match
  - well-defined object name sort order
  - efficient enumeration and random lookup
- NewStore = rocksdb + object files
  - `/var/lib/ceph/osd/ceph-123/`
    - `db/`
      - `<rocksdb, leveldb, whatever>`
    - `blobs.1/`
      - `0`
      - `1`
      - `...`
    - `blobs.2/`
      - `100000`
      - `100001`
      - `...`



NEWSTORE FAIL: CONSISTENCY OVERHEAD

- RocksDB has a write-ahead log "journal"
- XFS/ext4(/btrfs) have their own journal (tree-log)
- Journal-on-journal has high overhead
  - each journal manages half of overall consistency, but incurs same overhead

- `write(2) + fsync(2) to new blobs.2/10302`
- `1 write + flush to block device`

  `1 write + flush to XFS/ext4 journal`
- `write(2) + fsync(2) on RocksDB log`

  `1 write + flush to block device`

  `1 write + flush to XFS/ext4 journal`

NEWSTORE FAIL: ATOMICITY NEEDS WAL

- We can't overwrite a POSIX file as part of a atomic transaction
- Writing overwrite data to a new file means many files for each object
- Write-ahead logging?
  - put overwrite data in a “WAL” records in RocksDB
  - commit atomically with transaction
  - then overwrite original file data
  - but we're back to a double-write of overwrites...

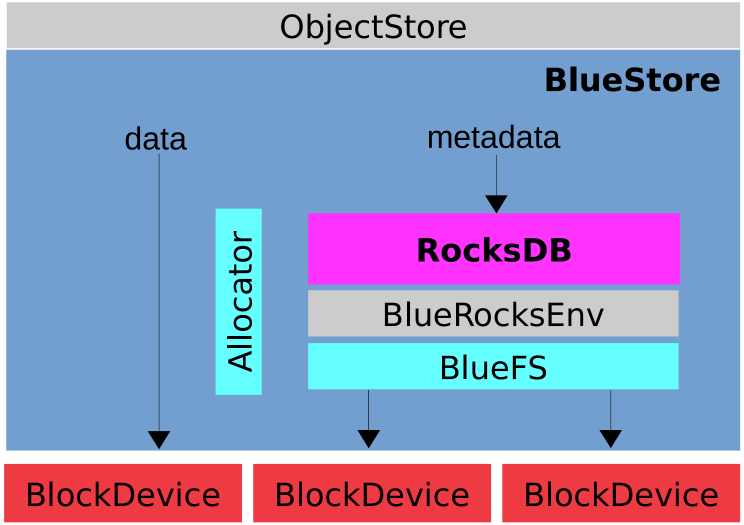
- Performance sucks again
- Overwrites dominate RBD block workloads

BLUESTORE

BLUESTORE

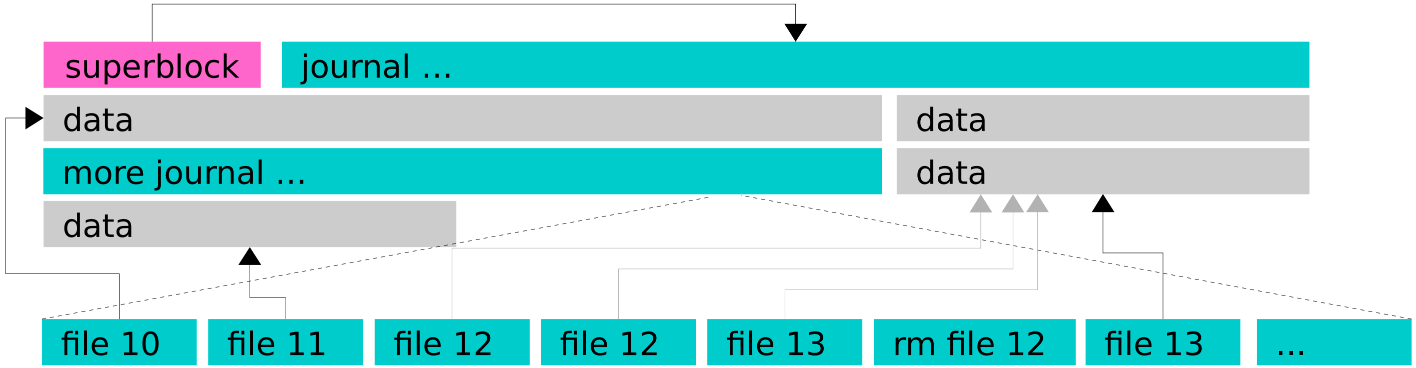
- BlueStore = **Bl**ock + **NewStore**
  - consume raw block device(s)
  - key/value database (RocksDB) for metadata
  - data written directly to block device
  - pluggable block Allocator

- We must share the block device with RocksDB
  - implement our own rocksdb::Env
  - implement tiny “file system” BlueFS
  - make BlueStore and BlueFS share



ROCKSDB: BLUEROCKSENV + BLUEFS

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



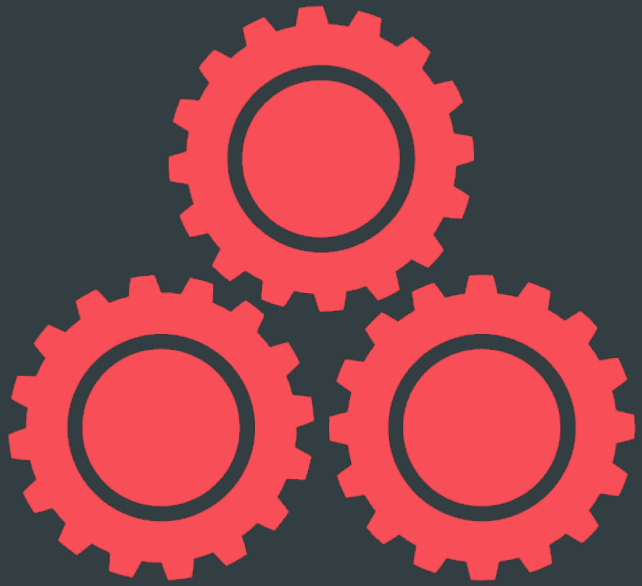
ROCKSDB: JOURNAL RECYCLING

- Problem: 1 small (4 KB) Ceph write → 3-4 disk IOs!
  - BlueStore: write **4 KB of user data**
  - rocksdb: append record to WAL
    - **write update block** at end of log file
    - fsync: XFS/ext4/BlueFS journals **inode size/alloc update** to its journal
- fallocate(2) doesn't help
  - data blocks are not pre-zeroed; fsync still has to update alloc metadata
- rocksdb LogReader only understands two modes
  - read until end of file (need accurate file size)
  - read all valid records, then ignore zeros at end (need zeroed tail)

ROCKSDB: JOURNAL RECYCLING (2)

- Put old log files on recycle list (instead of deleting them)
- LogWriter
  - overwrite old log data with new log data
  - include log number in each record
- LogReader
  - stop replaying when we get garbage (bad CRC)
  - or when we get a valid CRC but record is from a previous log incarnation
- Now we get one log append → one IO!

- Upstream in RocksDB!
  - but missing a bug fix (PR #881)
- Works with normal file-based storage, or BlueFS



METADATA

BLUESTORE METADATA

- Partition namespace for different metadata
  - S* – “superblock” metadata for the entire store
  - B* – block allocation metadata

  - C* – collection name → cnode_t
  - O* – object name → onode_t or enode_t
  - L* – write-ahead log entries, promises of future IO

  - M* – omap (user key/value data, stored in objects)
  - V* – overlay object data (obsolete?)

ONODE

- Per object metadata
  - Lives directly in key/value pair
  - Serializes to ~200 bytes
- Unique object id (like ino_t)
- Size in bytes
- Inline attributes (user attr data)
- Block pointers (user byte data)
  - Overlay metadata
- Omap prefix/ID (user k/v data)

```
struct bluestore_onode_t {
  uint64_t nid;
  uint64_t size;
  map<string,bufferptr> attrs;
  map<uint64_t,bluestore_extent_t> block_map;
  map<uint64_t,bluestore_overlay_t> overlay_map;
  uint64_t omap_head;
};

struct bluestore_extent_t {
  uint64_t offset;
  uint32_t length;
  uint32_t flags;
};
```

CNODE

- Collection metadata
  - Interval of object namespace

```
  shard  pool  hash     name          bits
C<NOSHARD,12,3d321e00> “12.e123d3” = <25>


  shard  pool  hash   name snap   gen
O<NOSHARD,12,3d321d88,foo,NOSNAP,NOGEN> = …
O<NOSHARD,12,3d321d92,bar,NOSNAP,NOGEN> = …
O<NOSHARD,12,3d321e02,baz,NOSNAP,NOGEN> = …
O<NOSHARD,12,3d321e12,zip,NOSNAP,NOGEN> = …
O<NOSHARD,12,3d321e12,dee,NOSNAP,NOGEN> = …
O<NOSHARD,12,3d321e38,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 metadata

ENODE

- Extent metadata
  - *Sometimes* we share blocks between objets (usually clones/snaps)
  - We need to reference count those extents
  - We still want to split collections and repartition extent metadata by hash

```
  shard   pool   hash    name snap    gen
O<NOSHARD,12,3d321d92,bar,NOSNAP,NOGEN> = onode
O<NOSHARD,12,3d321e02>                  = enode
O<NOSHARD,12,3d321e02,baz,NOSNAP,NOGEN> = onode
O<NOSHARD,12,3d321e12>                  = enode
O<NOSHARD,12,3d321e12,zip,NOSNAP,NOGEN> = onode
O<NOSHARD,12,3d321e12,dee,NOSNAP,NOGEN> = onode
O<NOSHARD,12,3d321e38,dah,NOSNAP,NOGEN> = onode
```

```
struct bluestore_enode_t {
  struct record_t {
    uint32_t length;
    uint32_t refs;
  };
  map<uint64_t,record_t> ref_map;

  void add(uint64_t offset, uint32_t len,
           unsigned ref=2);
  void get(uint64_t offset, uint32_t len);
  void put(uint64_t offset, uint32_t len,
           vector<bluestore_extent_t> *release);
};
```

DATA PATH

DATA PATH BASICS

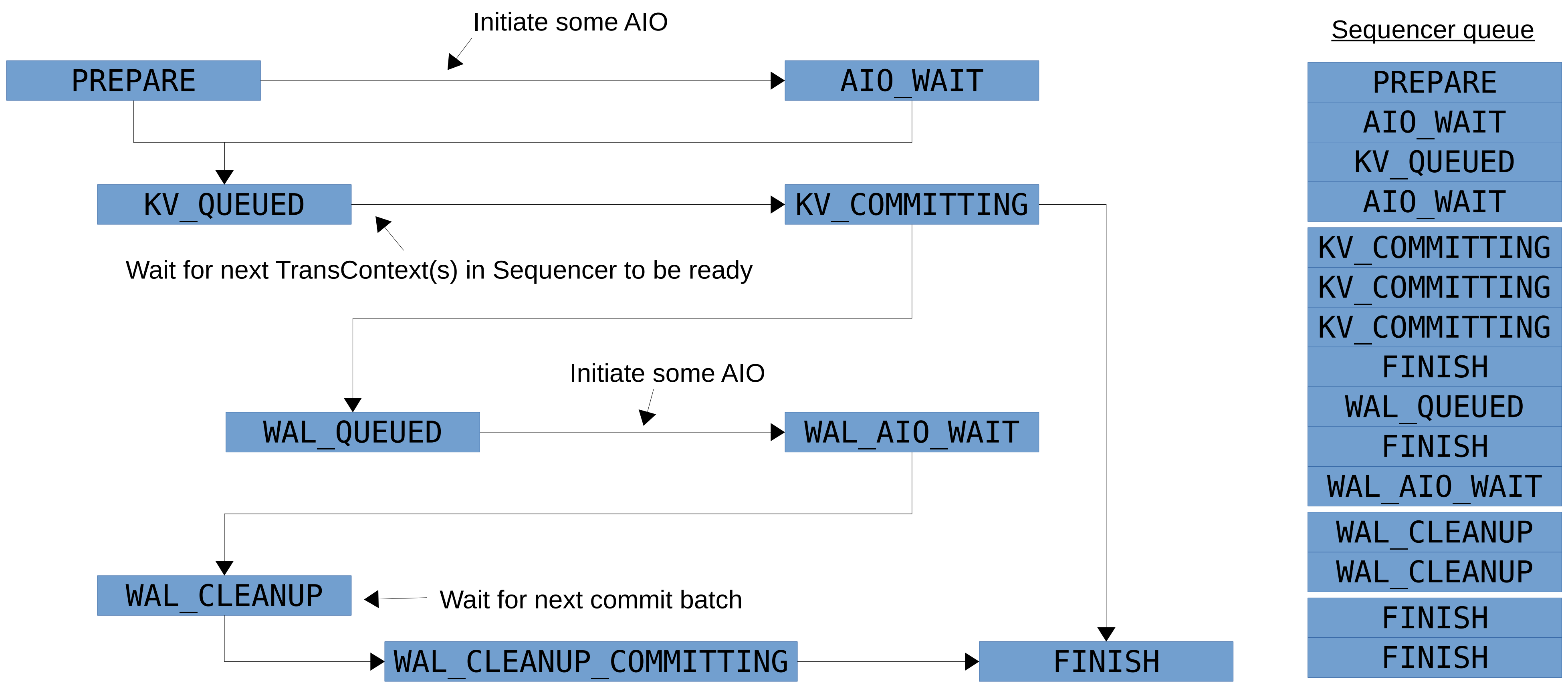
Terms

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

Two 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
- WAL (write-ahead-logged)
  - Commit temporary promise to (over)write data with transaction
    - includes data!
  - Do async overwrite
  - Clean up temporary k/v pair

TRANSCONTEXT STATE MACHINE



AIO, O_DIRECT, AND CACHING

- From open(2) man page

```
Applications  should  avoid  mixing O_DIRECT and normal I/O to the same
file, and especially to overlapping byte  regions  in  the  same  file.
```

- By default, all IO is AIO + O_DIRECT
  - but sometimes we want to cache (e.g., POSIX_FADV_WILLNEED)
- BlueStore mixes direct and buffered IO
  - O_DIRECT read(2) and write(2) invalidate and/or flush pages... racily
  - we avoid mixing them on the same pages
  - disable readahead: `posix_fadvise(fd, 0, 0, POSIX_FADV_RANDOM)`
- But it's not quite enough...
  - moving to fully user-space cache

BLOCK FREE LIST

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

  ```
  <offset> = <length>
  ```

  - keep in-memory copy
  - enforce an ordering on commits

    ```
    del 1600=100000
    put 1700=99990
    ```

  - small initial memory footprint, very expensive when fragmented
- New approach

  ```
  <offset> = <region bitmap>
  ```

  - where region is N blocks (1024?)
  - no in-memory state
  - 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

BLOCK ALLOCATOR

- Allocator
  - abstract interface to allocate new space
- StupidAllocator
  - 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

SMR HDD

- Let's support them natively!
- 256 MB zones / bands
  - must be written sequentially, but not all at once
  - libzbc supports ZAC and ZBC HDDs
  - host-managed or host-aware
- SMRAllocator
  - write pointer per zone
  - used + free counters per zone
  - Bonus: almost no memory!
- IO ordering
  - must ensure allocated writes reach disk in order
- Cleaning
  - store k/v hints

    zone → object hash
  - pick emptiest closed zone, scan hints, move objects that are still there
  - opportunistically rewrite objects we read if the zone is flagged for cleaning soon

FANCY STUFF

WE WANT FANCY STUFF

Full data checksums

- We scrub... periodically
- We want to validate checksum on **every** read

Compression

- 3x replication is expensive
- Any scale-out cluster is expensive

WE WANT FANCY STUFF

Full data checksums

- We scrub... periodically
- We want to validate checksum on **every** read
- More data with extent pointer
  - 4KB for 32-bit csum per 4KB block
  - bigger onode: 300 → 4396 bytes
  - larger csum blocks?

Compression

- 3x replication is expensive
- Any scale-out cluster is expensive
- Need largish extents to get compression benefit (64 KB, 128 KB)
  - overwrites need to do read/modify/write

INDRECT EXTENT STRUCTURES

onode_t

```
map<uint64_t,bluestore_lextent_t> extent_map;

struct bluestore_lextent_t {
  uint64_t blob_id;
  uint64_t b_off, b_len;
};
```

onode may reference a piece of a blob, or multiple pieces

onode_t or enode_t

```
map<uint64_t,bluestore_blob_t> blob_map;

struct bluestore_blob_t {
  vector<bluestore_pextent_t> extents;
  uint32_t logical_length; ///< uncompressed length
  uint32_t flags;          ///< FLAGS_*
  uint16_t num_refs;
  uint8_t csum_type;       ///< CSUM_*
  uint8_t csum_block_order;
  vector<char> csum_data;
};
```

ref counted (when in enode)

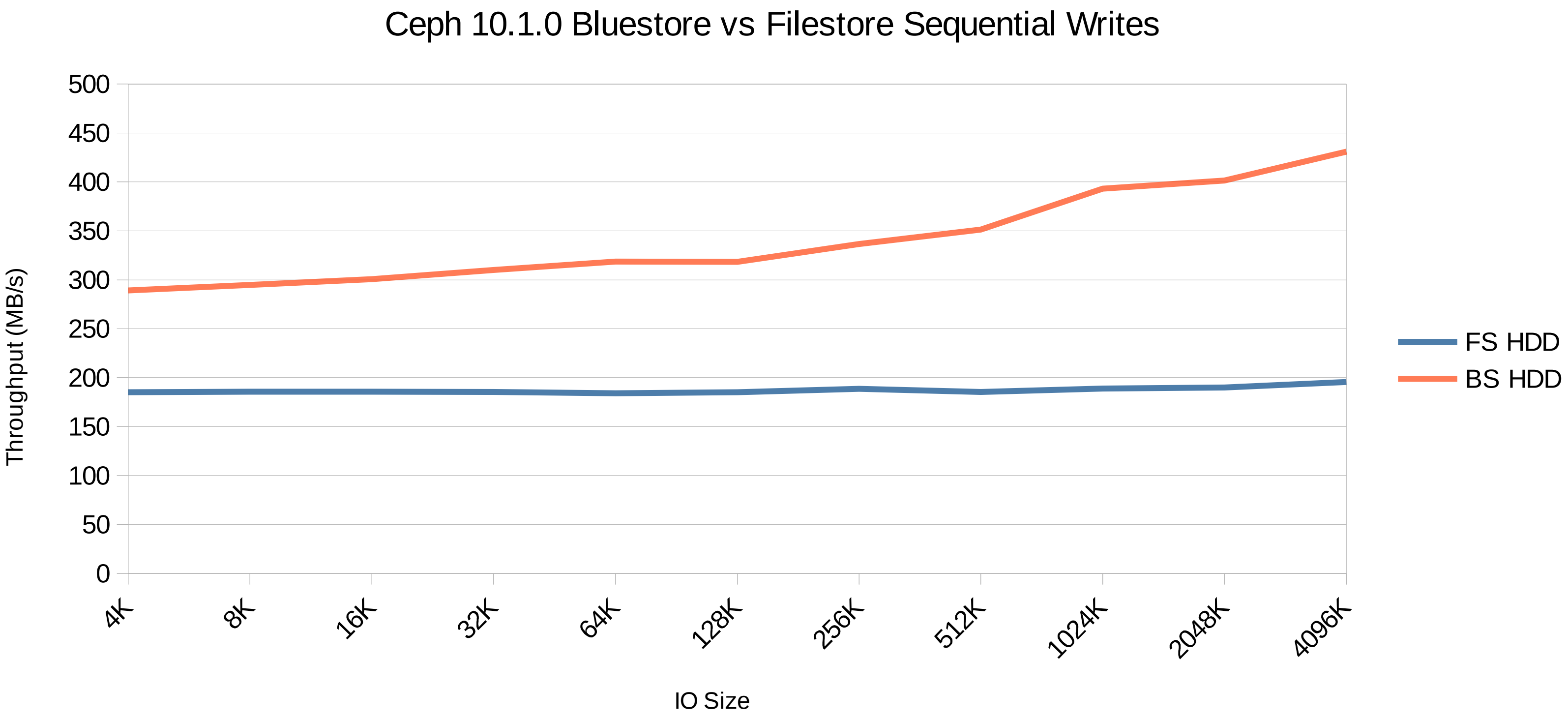
csum block size can vary

```
struct bluestore_pextent_t {
  uint64_t offset, length;
};
```

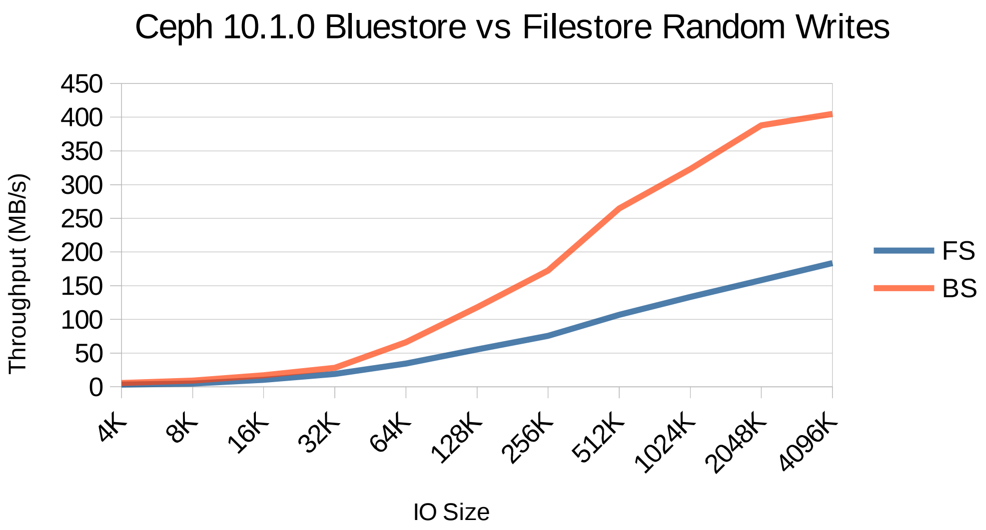
data on block device

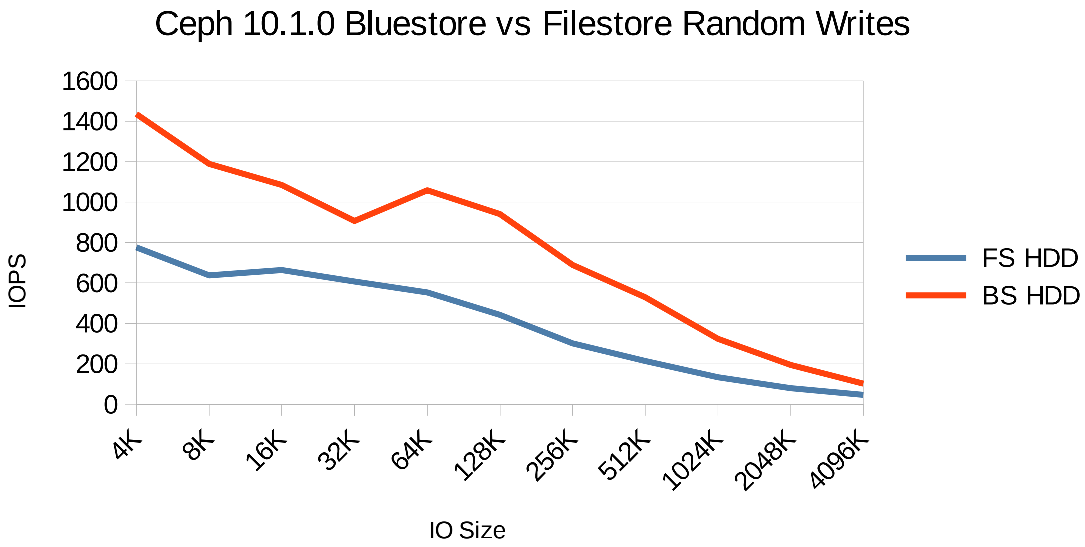
PERFORMANCE

HDD: SEQUENTIAL WRITE

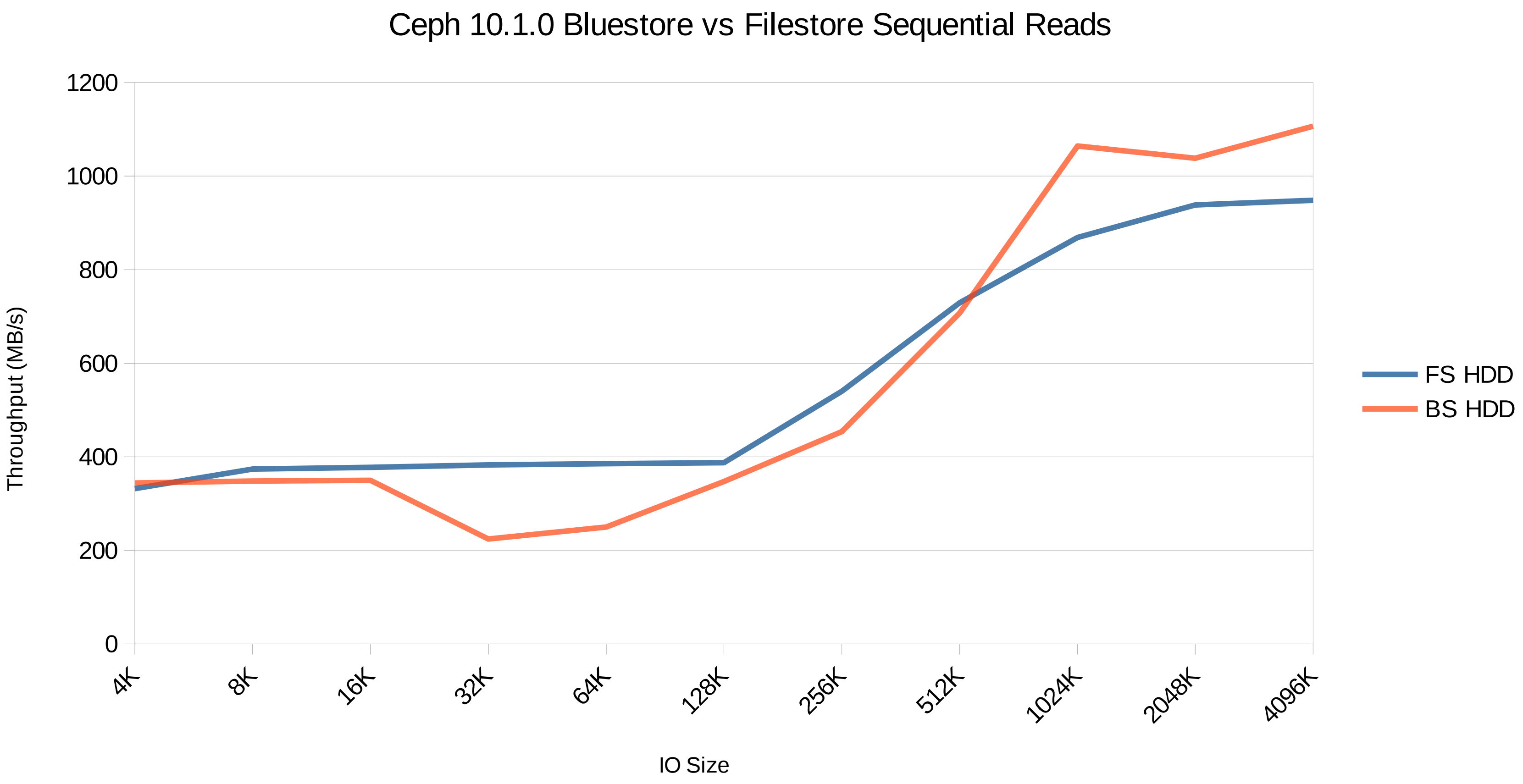


HDD: RANDOM WRITE

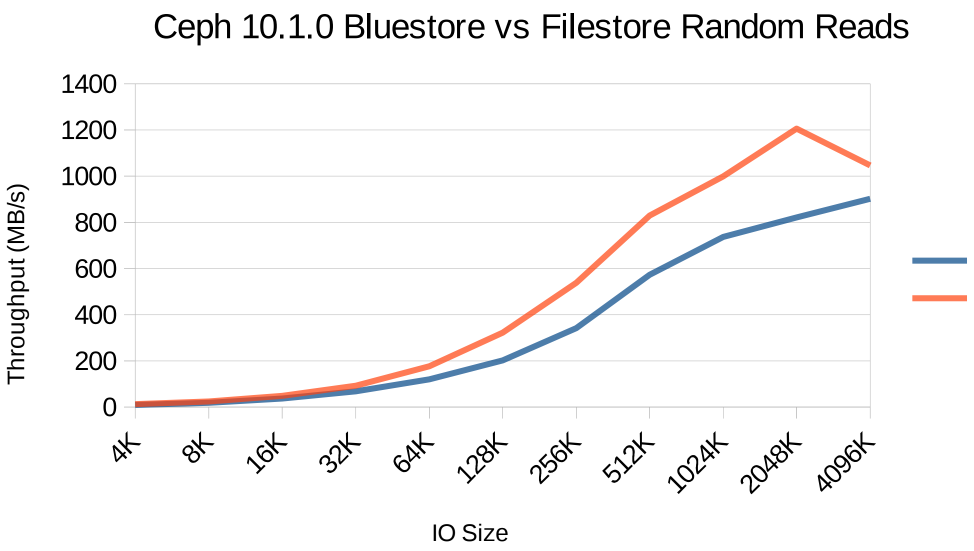


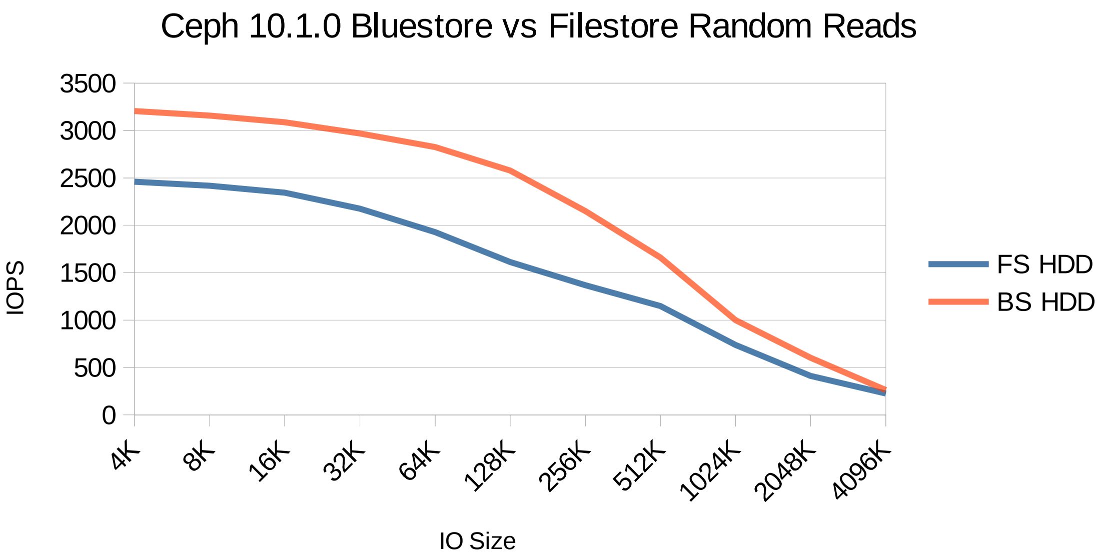


HDD: SEQUENTIAL READ



HDD: RANDOM READ





SSD AND NVME?

- NVMe journal
  - random writes ~2x faster
  - some testing anomalies (problem with test rig kernel?)
- SSD only
  - similar to HDD result
  - small write benefit is more pronounced
- NVMe only
  - more testing anomalies on test rig.. WIP

AVAILABILITY

- Experimental backend in Jewel v10.2.z (just released)
  - enable experimental unrecoverable data corrupting features = bluestore rocksdb
  - ceph-disk --bluestore DEV
    - no multi-device magic provisioning just yet

- The goal...
  - stable in Kraken (Fall '16)?
  - default in Luminous (Spring '17)?

SUMMARY

- Ceph is great
- POSIX was poor choice for storing objects
- Our new BlueStore backend is awesome
- RocksDB rocks and was easy to embed
- Log recycling speeds up commits (now upstream)
- Delayed merge will help too (coming soon)

THANK YOU!

Sage Weil
CEPH PRINCIPAL ARCHITECT

sage@redhat.com

@liewegas