Evaluating NVMe drives for accelerating HBase

Nicolas Poggi and David Grier

BSC Data Centric Computing – Rackspace collaboration

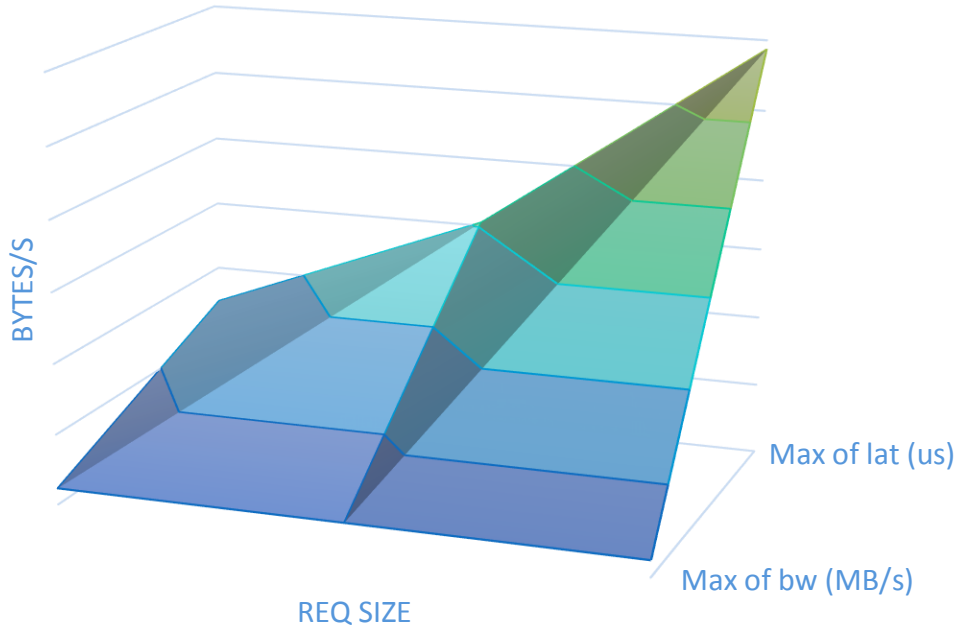
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

FOSDEM Jan 2017

aloja

Outline

1. Intro on BSC and ALOJA
2. Cluster specs and disk benchmarks
3. **HBase use case with NVE**
    1. Read-only workload
        - Different strategies
    2. Mixed workload
4. Summary



Barcelona Supercomputing Center (BSC)

aloja

- Spanish national supercomputing center 22 years history in:
  - Computer Architecture, networking and distributed systems research
  - Based at BarcelonaTech University (UPC)
  - Large ongoing life science computational projects

- Prominent body of research activity around **Hadoop**
  - 2008-2013: SLA Adaptive Scheduler, Accelerators, Locality Awareness, Performance Management. 7+ publications
  - 2013-Present: **C**ost-efficient upcoming Big Data architectures (ALOJA) 6+ publications

GOBIERNO DE ESPAÑA MINISTERIO DE ECONOMÍA Y COMPETITIVIDAD

Generalitat de Catalunya
Departament d'Economia i Coneixement

UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

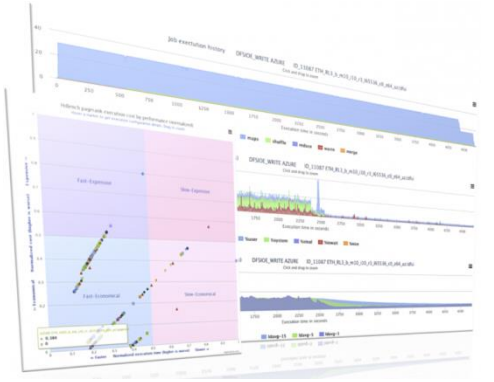
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

EXCELENCIA SEVERO OCHOA

ALOJA: towards cost-effective Big Data

aloja

- Research project for automating characterization and optimization of **Big Data** deployments



http://aloja.bsc.es

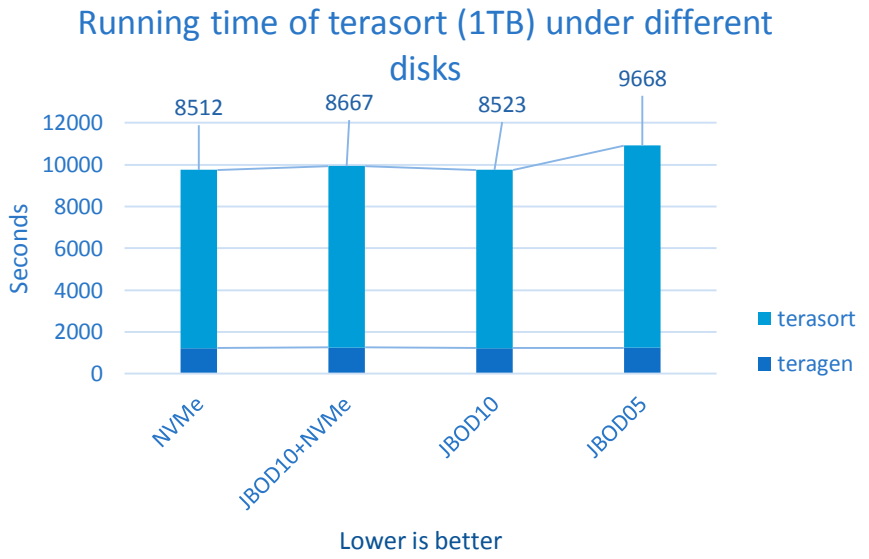
- Open source *Benchmarking-to-Insights* platform and tools

- Largest Big Data public repository (70,000+ jobs)
  - Community collaboration with industry and academia

Motivation and objectives

- Explore use cases where NVMe devices can speedup Big Data apps
  - Poor initial results…
  - HBase (this study) based on Intel report
- Measure the possibilities of NVMe devices
  - System level benchmarks (FIO, IO Meter)
  - WiP towards tiered-storage for Big data status
  - Extend ALOJA into low-level I/O
- Challenge
  - benchmark and stress high-end Big Data clusters
    - In reasonable amount of time (and cost)

First tests:



Marginal improvement!!!

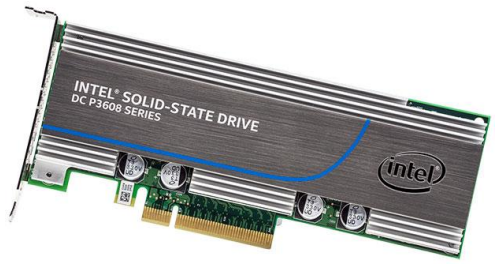
Cluster and drive specs

All nodes (x5)	
Operating System	CentOS 7.2
Memory	128GB
CPU	Single Octo-core (16 threads)
Disk Config OS	2x600GB SAS RAID1
Network	10Gb/10Gb redundant

Master node (x1, extra nodes for HA not used in these tests)	
Disk Config Master storage	4x600GB SAS RAID10 XSF partition

Data Nodes (x4)	
NVMe (cache)	1.6TB Intel DC P3608 NVMe SSD (PCIe)
Disk config HDFS data storage	10x 0.6TB NL-SAS/SATA JBOD PCIe3 x8, 12Gb/s SAS RAID SEAGATE ST3600057SS 15K (XFS partition)

1. Intel DC P3608 (current 2015)
   - PCI-Express 3.0 x8 lanes
   - Capacity: 1.6TB (two drives)
   - Seq R/W BW:
     - 5000/2000 MB/s, (128k req)
   - Random 4k R/W IOPS:
     - 850k/150k
   - Random 8k R/W IOPS:
     - 500k/60k, 8 workers
   - Price $10,780
     - (online search 02/2017)

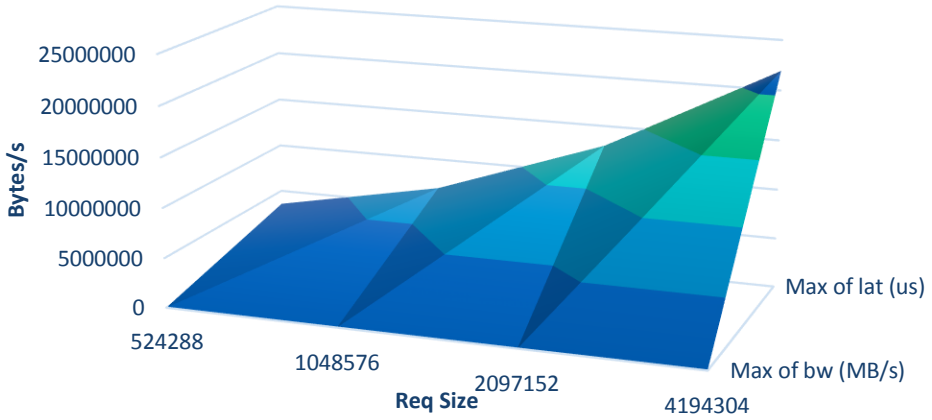


2. LSI Nytro WarpDrive 4-400 (old gen 2012)
   - PCI-Express 2.0 x8 lanes
   - Capacity: 1.6TB (two drives)
   - Seq. R/W BW:
     - 2000/1000 MB/s (256k req)
   - R/W IOPS:
     - 185k/120k (8k req)
   - Price $4,096
     - (online search 02/2017)
     - $12,195 MSRP 2012

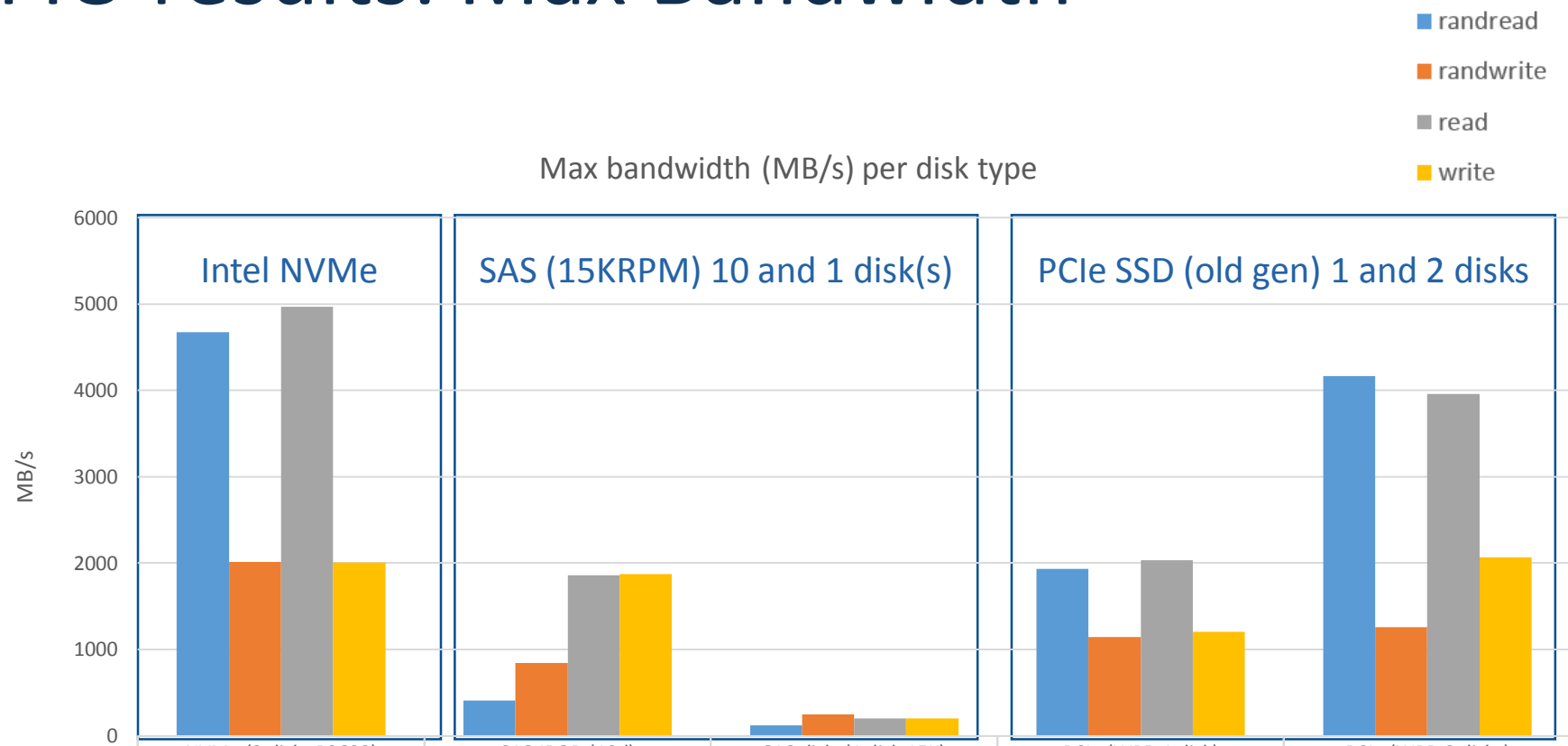
FIO Benchmarks

Objectives:

- Assert vendor specs (BW, IOPS, Latency)
- Seq R/W, Random R/W
- Verify driver/firmware and OS
- Set performance expectations



FIO results: Max Bandwidth



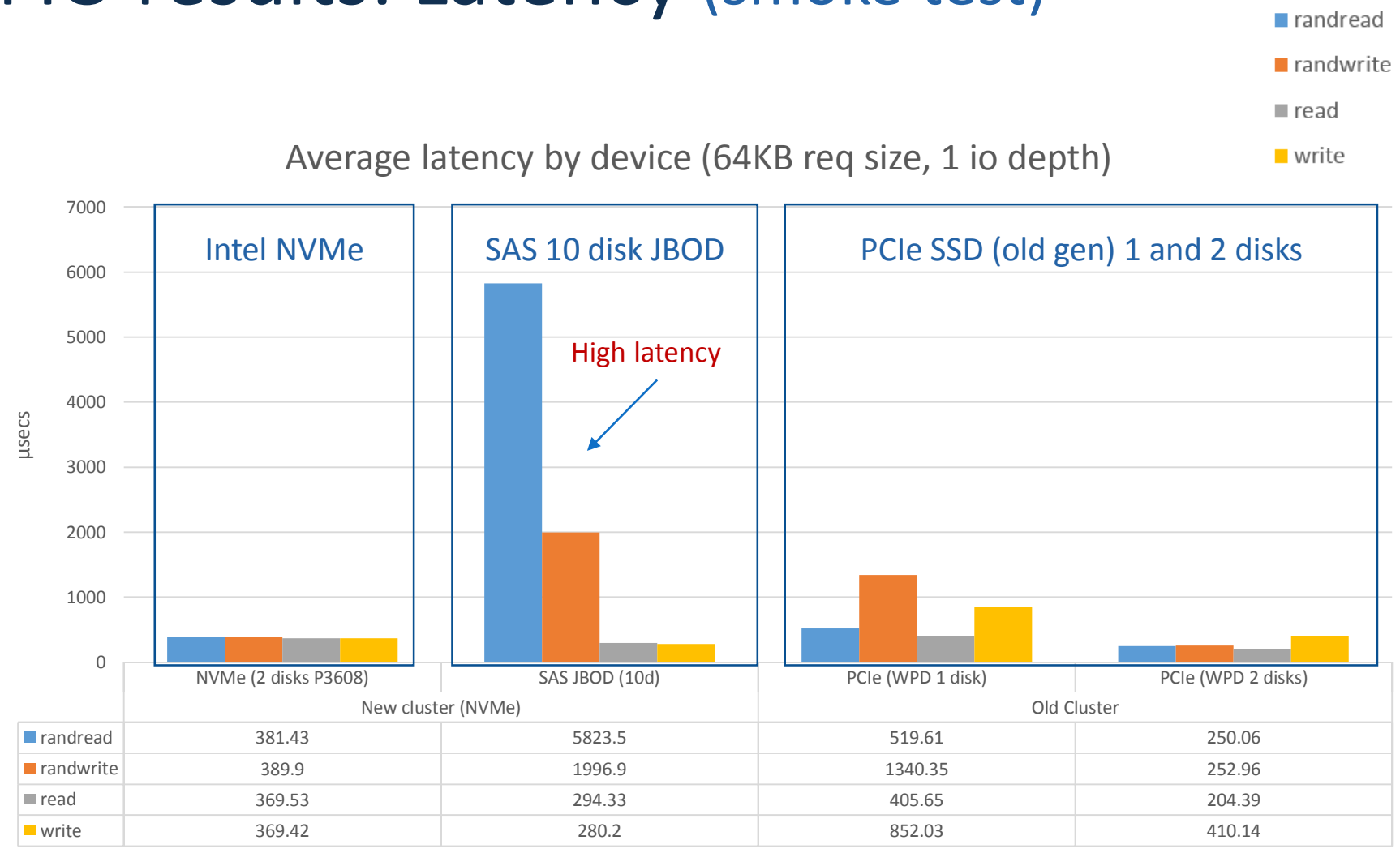
	NVMe (2 disks P3608)	SAS JBOD (10d)	SAS disk (1 disk 15K)	PCIe (WPD 1 disk)	PCIe (WPD 2 disks)
randread	4674.99	409.95	118.07	1935.24	4165.65
randwrite	2015.27	843	249.06	1140.96	1256.12
read	4964.44	1861.4	198.54	2033.52	3957.42
write	2006	1869.49	204.17	1201.52	2066.48

Results:

- Random R/W similar in both PCIe SSDs
  - But not for the SAS JBOD
- SAS JBOD achieves high both seq R/W
  - 10 disks 2GB/s
- Achieved both PCIe vendor numbers
  - Also on IOPS
- Combined WPD disks only improve in W performance

Higher is better.
Max BW recorded for each device under different settings: req size, io depth, threads. Using libaio.

FIO results: Latency (smoke test)



	NVMe (2 disks P3608)	SAS JBOD (10d)	PCIe (WPD 1 disk)	PCIe (WPD 2 disks)
randread	381.43	5823.5	519.61	250.06
randwrite	389.9	1996.9	1340.35	252.96
read	369.53	294.33	405.65	204.39
write	369.42	280.2	852.03	410.14

Results:

- JBOD has highest latency for random R/W (as expected)
  - But very low for seq
- Combined WPD disks lower the latency
  - Lower than P3608 disks.

Notes:

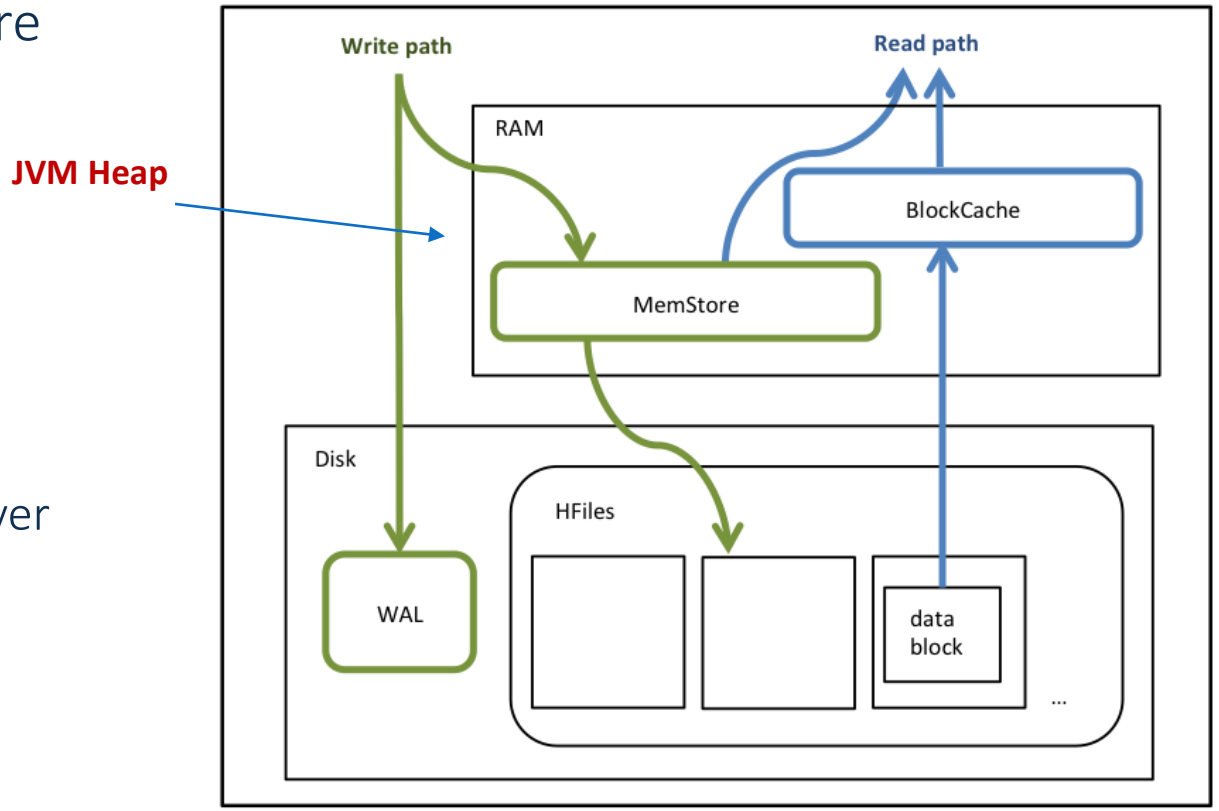
- Need to more thorough comparison and at different settings.

Higher is better.
Average latency for req size 64KB and 1 io depth (varying workers). Using libaio.

HBase in a nutshell

- Highly scalable Big data key-value store
  - On top of Hadoop (HDFS)
  - Based on Google's Bigtable
- Real-time and **random access**
  - Indexed
  - **Low-latency**
  - Block cache and Bloom Filters
- Linear, modular scalability
  - Automatic sharding of tables and failover
- Strictly consistent reads and writes.
  - failover support
- Production ready and battle tested
  - Building block of other projects

HBase R/W architecture

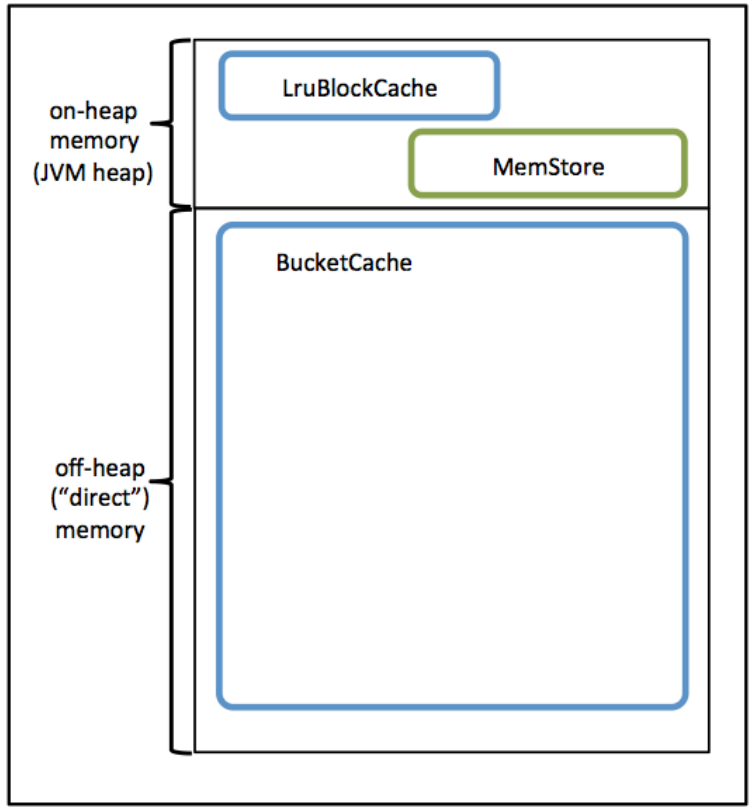


Source: HDP doc

L2 Bucket Cache (BC) in HBase

- Adds a second “block” storage for HFiles
- Use case: L2 cache and replaces OS buffer cache
  - Does copy-on-read
- Fixed sized, reserved on startup
- 3 different modes:
  - Heap
    - Marginal improvement
    - Divides mem with the block cache
  - Offheap (in RAM)
    - Uses Java NIO’s Direct ByteBuffer
  - File
    - Any local file / device
    - Bypasses HDFS
    - Saves RAM

Region server (worker) memory with BC



L2-BucketCache experiments summary

Tested configurations for HBase v1.24

1. HBase default (**baseline**)
2. HBase w/ Bucket cache **Offheap**
   1. Size: **32GB** /work node
3. HBase w/ Bucket cache in **RAM disk**
   1. Size: **32GB** /work node
4. HBase w/ Bucket cache in **NVMe disk**
   1. Size: 250GB / worker node

- All using same Hadoop and HDFS configuration
  - On JBOD (10 SAS disks, /grid/{0,9})
  - 1 Replica, short-circuit reads

Experiments

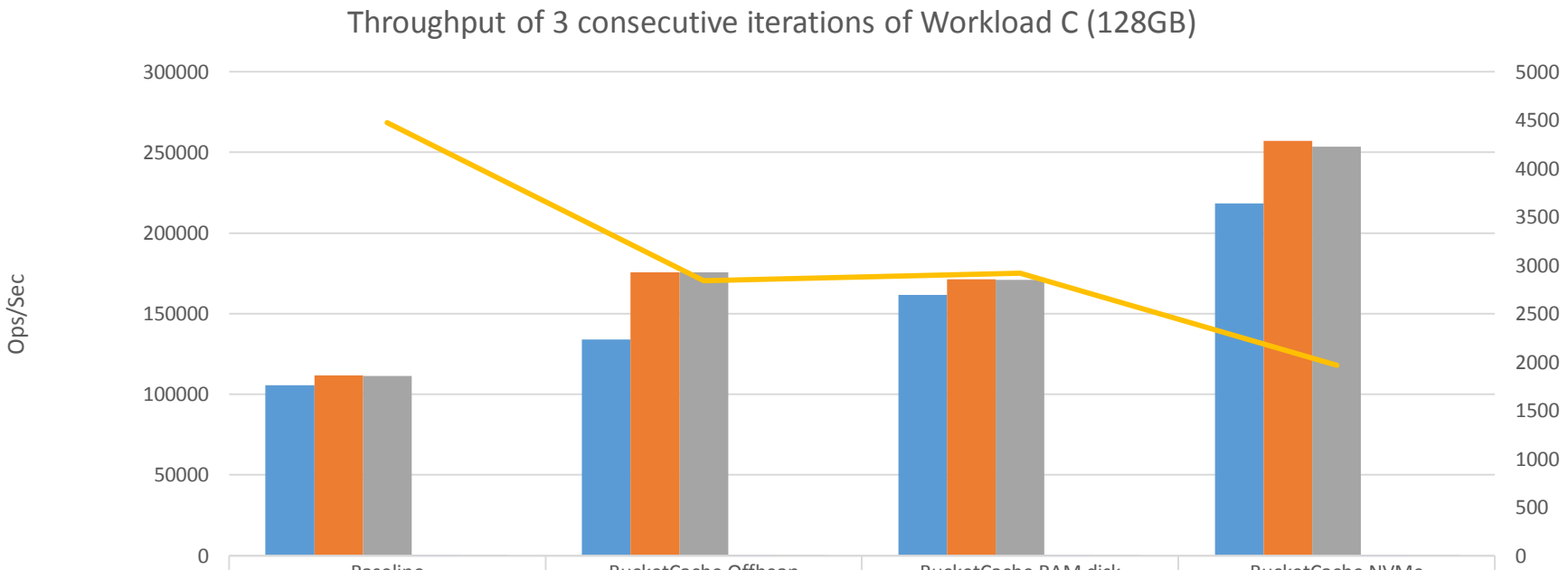
1. Read-only (workload C)
   1. RAM at 128GB / node
   2. RAM at 32GB / node
   3. Clearing buffer cache
2. Full YCSB (workloads A-F)
   4. RAM at 128GB / node
   5. RAM at 32GB / node

- Payload:
  - **YCSB** 250M records.
  - ~2TB HDFS raw

Experiment 1: Read-only (gets)

YCSB Workload C: 25M records, 500 threads, ~2TB HDFS

E 1.1: Throughput of the 4 configurations (128GB RAM)



	Baseline	BucketCache Offheap	BucketCache RAM disk	BucketCache NVMe
WorkloadC_run1	105610	133981	161712	218342
WorkloadC_run2	111530	175483	171236	257017
WorkloadC_run3	111422	175483	170889	253625
Cache Time %	5.5	31	5.6	16.2
Speeup (run3)	1	1.57	1.53	2.28
Latency µs (run3)	4476	2841	2917	1964

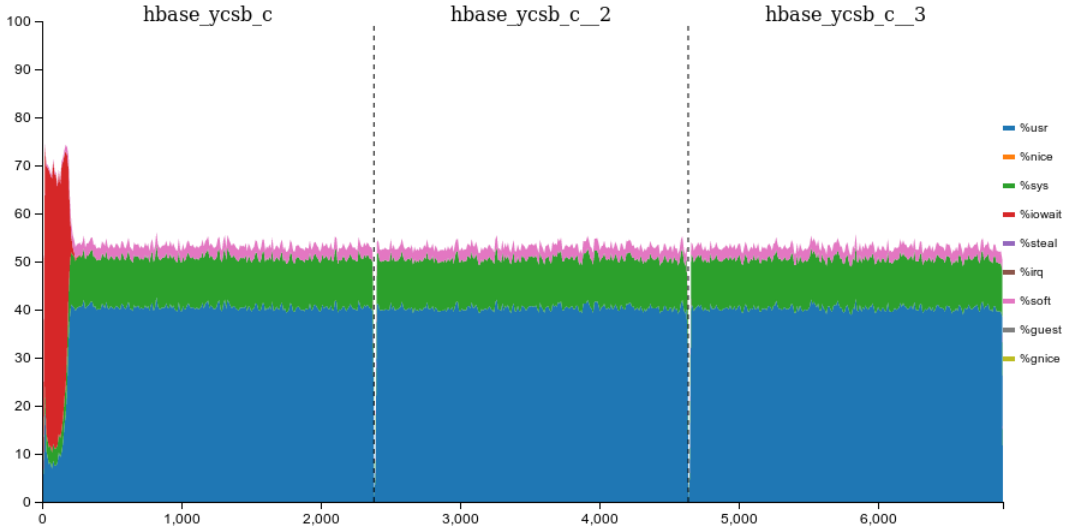
Results:

- Ops/Sec improve with BC
  - Offheap 1.6x
  - RAMd 1.5x
  - NVMe 2.3x
- AVG latency as well
  - (req time)
- First run slower on 4 cases (writes OS cache and BC)
  - Baseline and RAMd only 6% faster after
  - NVMe 16%
- 3rd run not faster, cache already loaded
- Tested onheap config:
  - > 8GB failed
  - 8GB slower than baseline

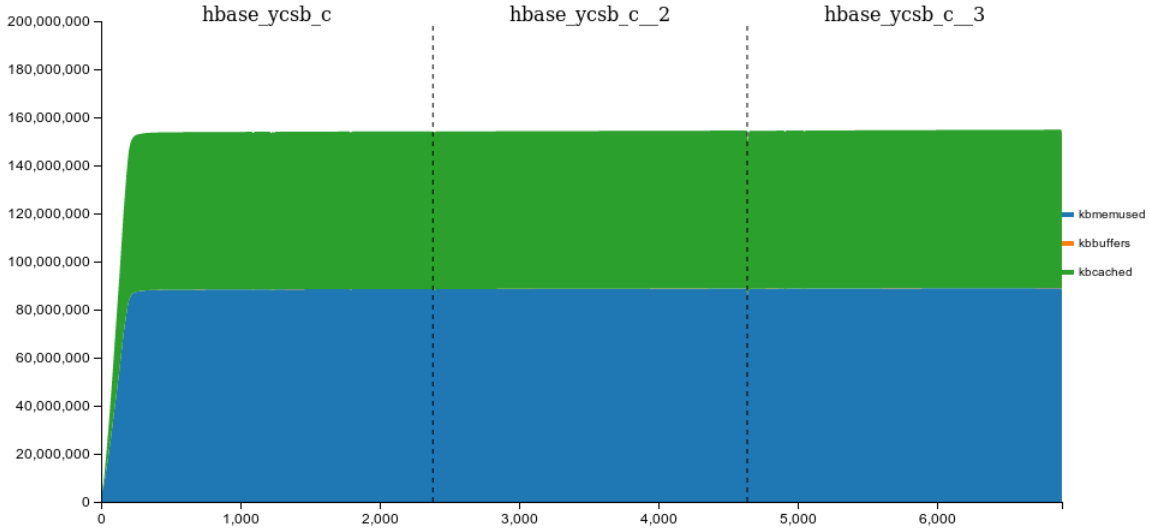
Higher is better for Ops (Y1), lower for latency (Y2)
(Offheap run2 and 3 the same run)

E 1.1 Cluster resource consumption: Baseline

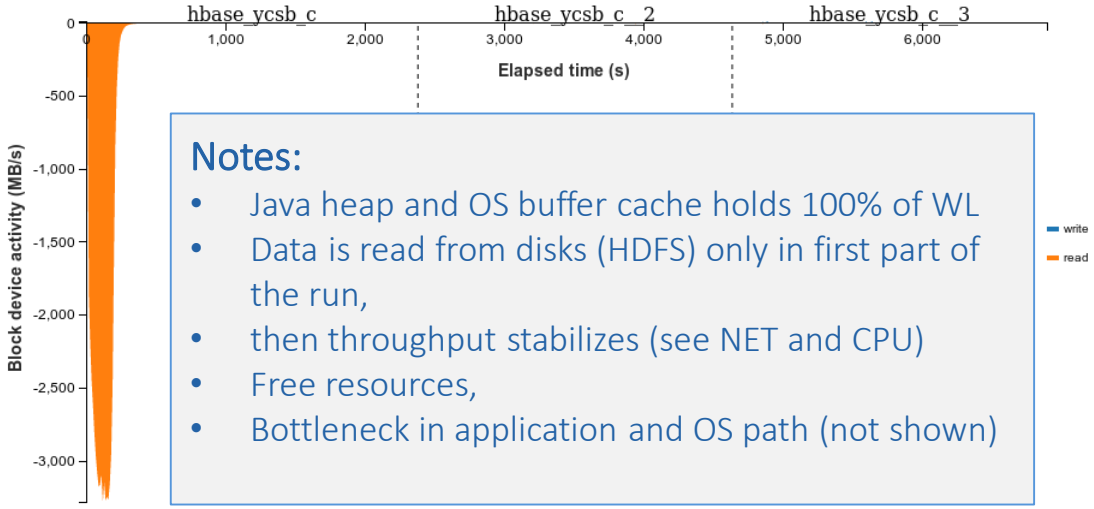
CPU % (AVG)



Mem Usage KB (AVG)



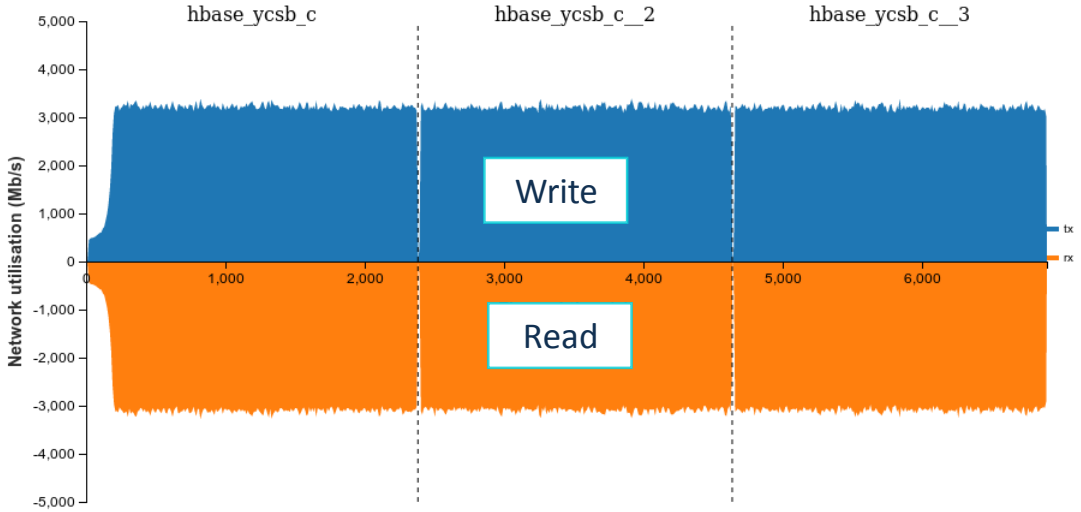
Disk R/W MB/s (SUM)



Notes:

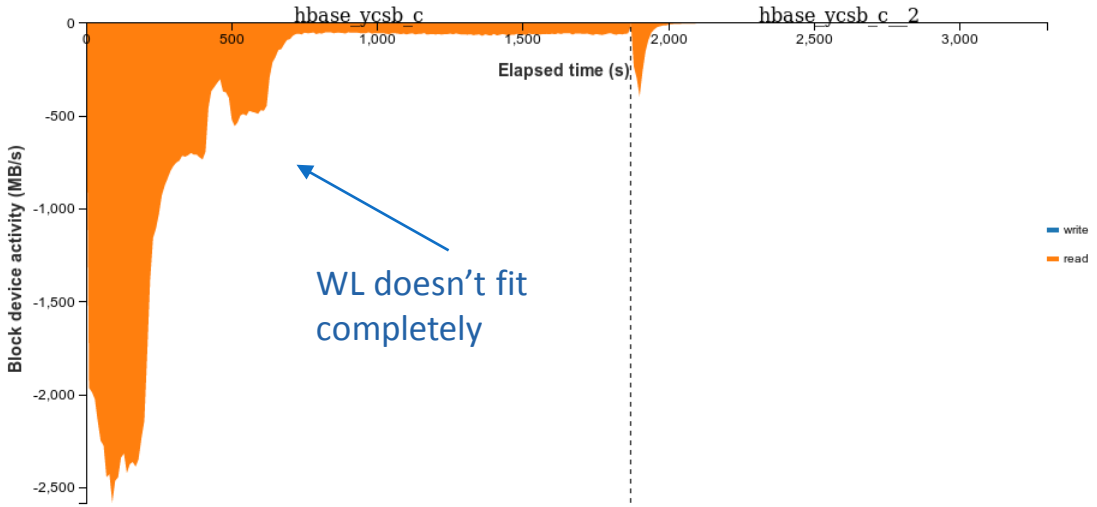
- Java heap and OS buffer cache holds 100% of WL
- Data is read from disks (HDFS) only in first part of the run,
- then throughput stabilizes (see NET and CPU)
- Free resources,
- Bottleneck in application and OS path (not shown)

NET R/W Mb/s (SUM)

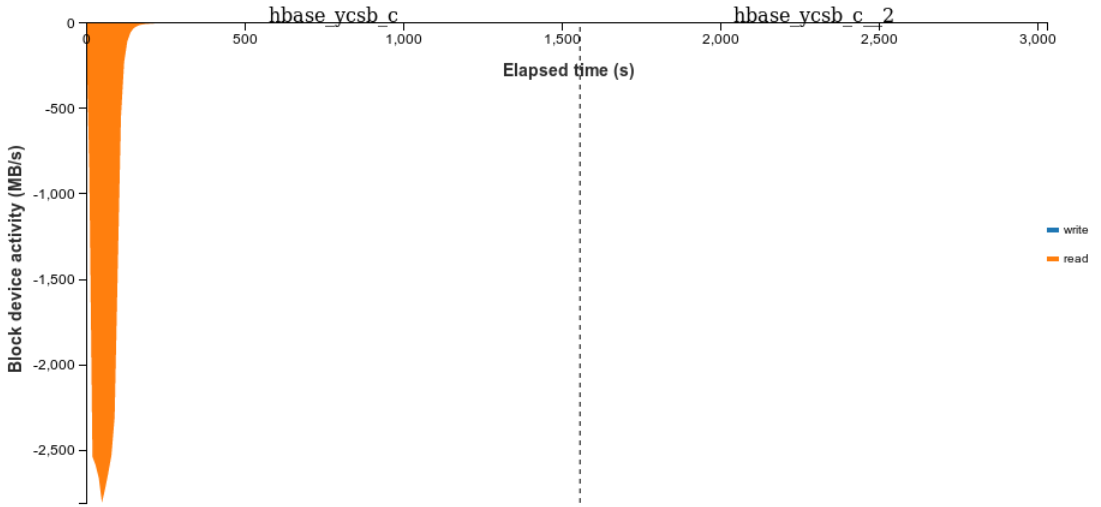


E 1.1 Cluster resource consumption: Bucket Cache strategies

Offheap (32GB) Disk R/W



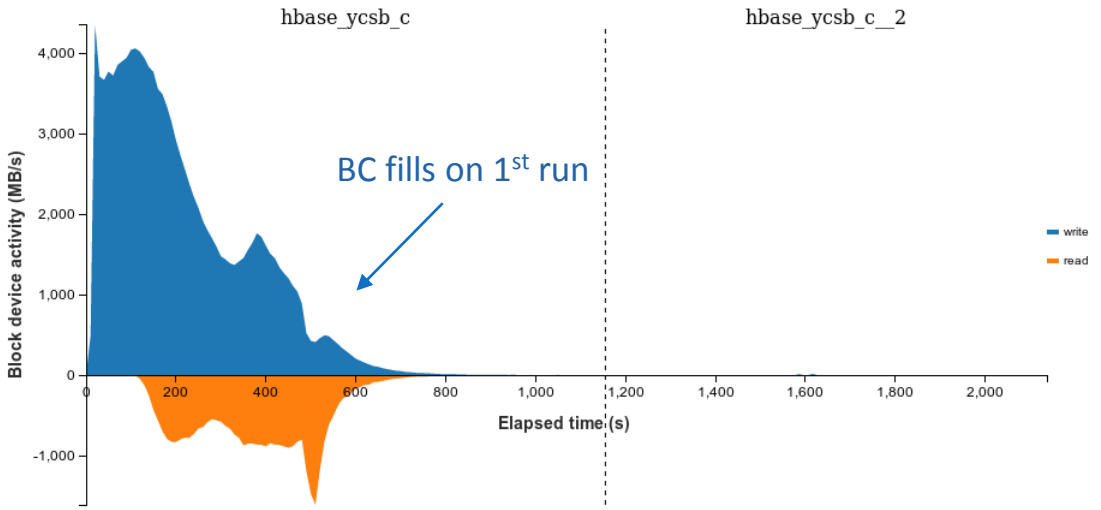
RAM disk (tmpfs 32GB) Disk R/W



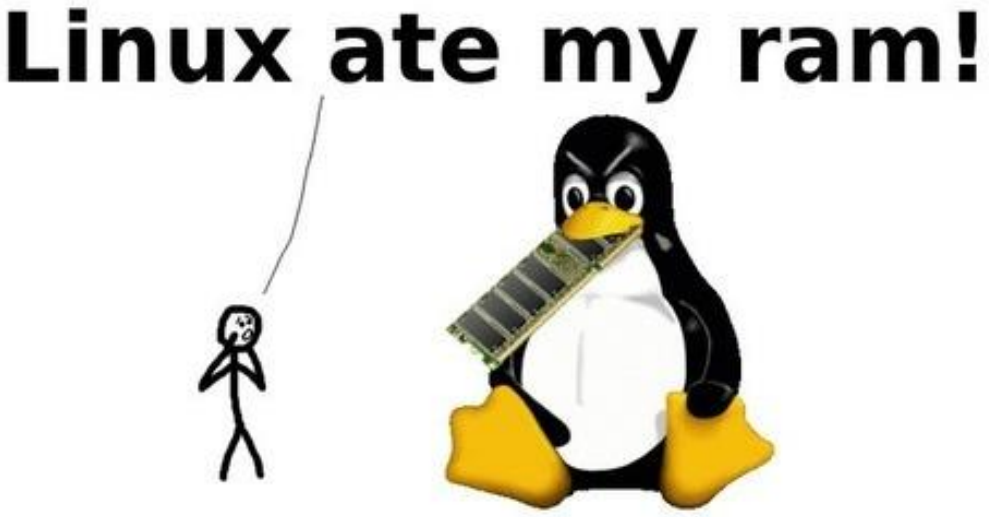
Notes:

- 3 BC strategies faster than baseline
- BC LRU more effective than OS buffer
- Offheap slightly more efficient ran RAMd (same size)
  - But seems to take longer to fill (different per node)
  - And more capacity for same payload (plus java heap)
- NVM can hold the complete WL in the BC
- Read and Writes to MEM not captured by charts

NVMe (250GB) Disk R/W



E 1.2-3



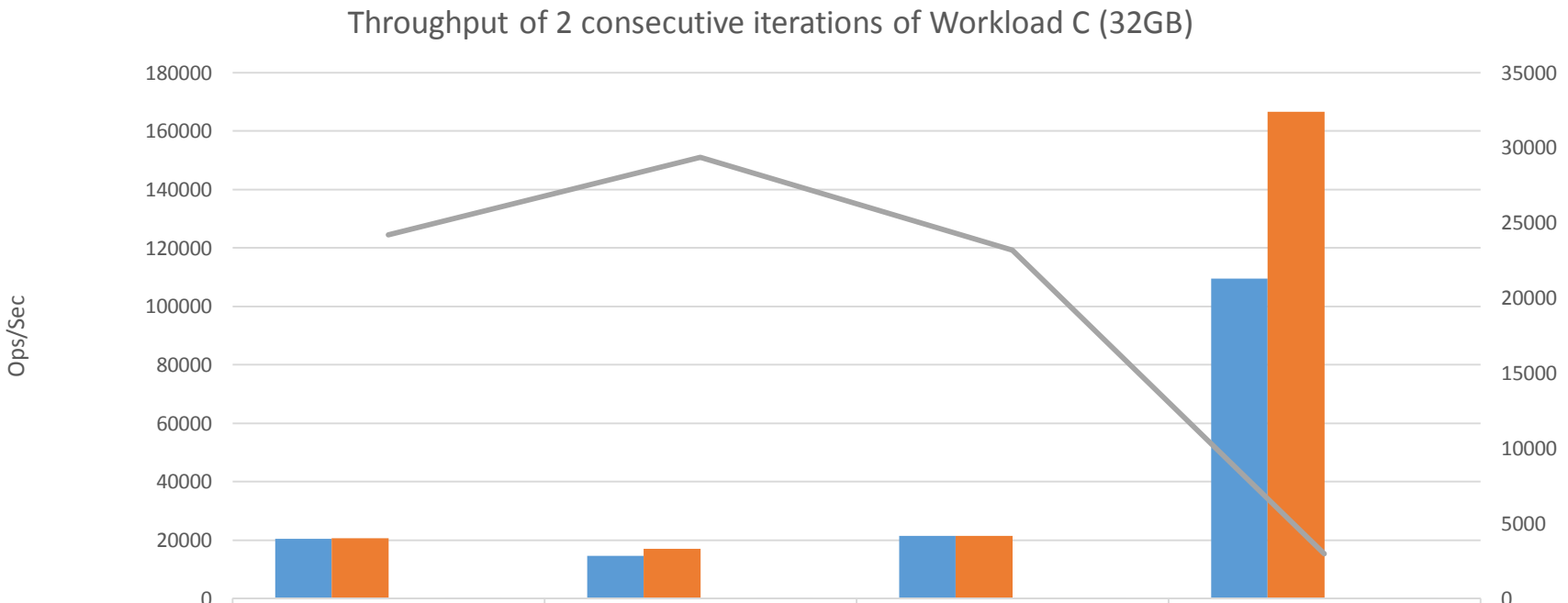
Challenge:
Limit OS buffer cache effect on experiments

1st approach, larger payload. Cons: high execution time

2nd limit available RAM (using *stress* tool)

3rd clear buffer cache periodically (drop caches)

E1.2: Throughput of the 4 configurations (32GB RAM)



	Baseline	BucketCache Offheap	BucketCache RAM disk	BucketCache NVMe
WorkloadC_run1	20578	14715.493	21520	109488
WorkloadC_run2	20598	16995	21534	166588
Speeup (run2)	1	0.83	1.05	8.09
Cache Time %	99.9	99.9	99.9	48
Latency µs (run3)	24226	29360	23176	2993

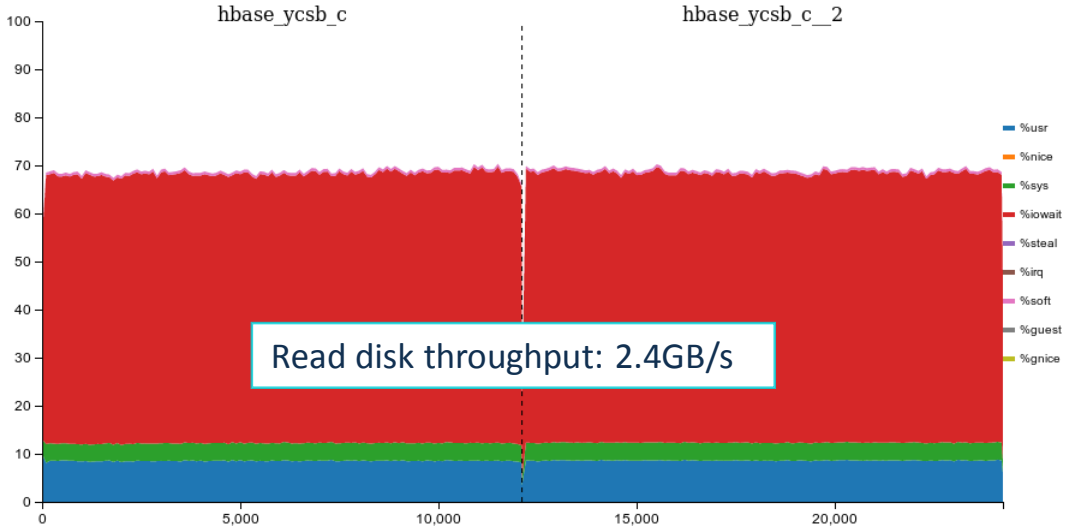
Results:

- Ops/Sec improve only with NVMe up to 8X
  - RAMd performs close to baseline
- First run same on baseline and RAMd
  - tmpfs "blocks" as RAM is needed
- At lower capacity, external BC shows more improvement

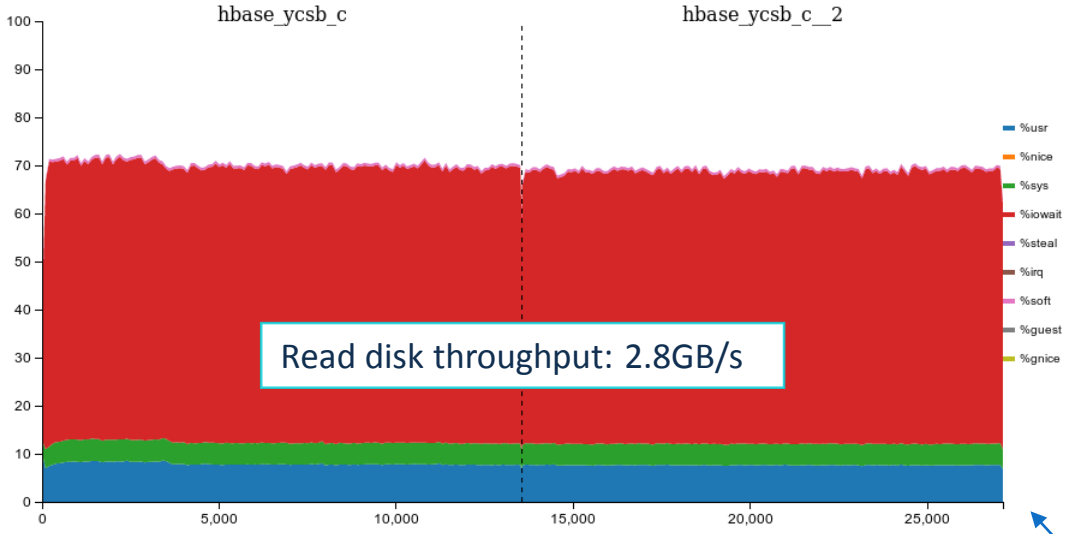
Higher is better for Ops (Y1), lower for latency (Y2)

E 1.1 Cluster CPU% AVG: Bucket Cache (32GB RAM)

Baseline

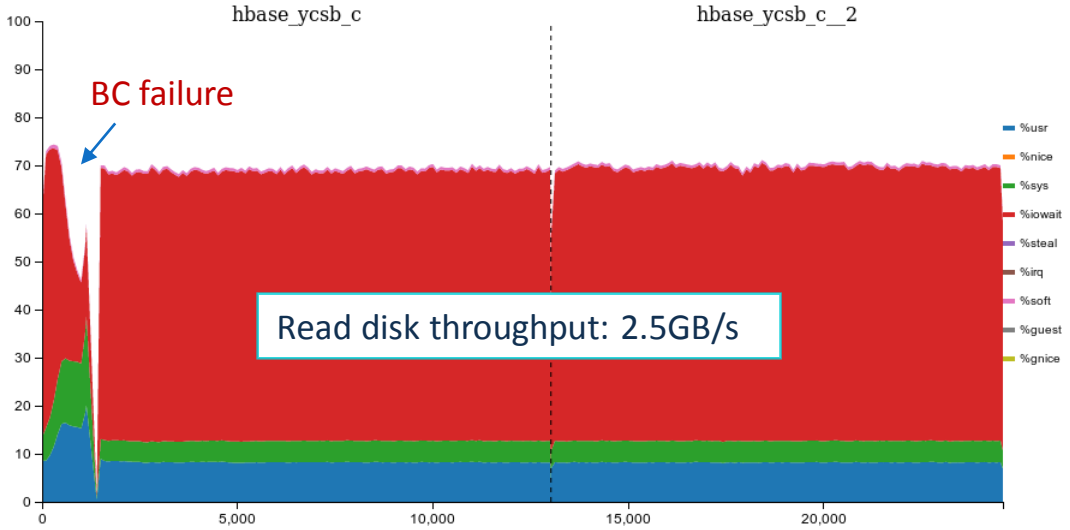


Offheap (4GB)

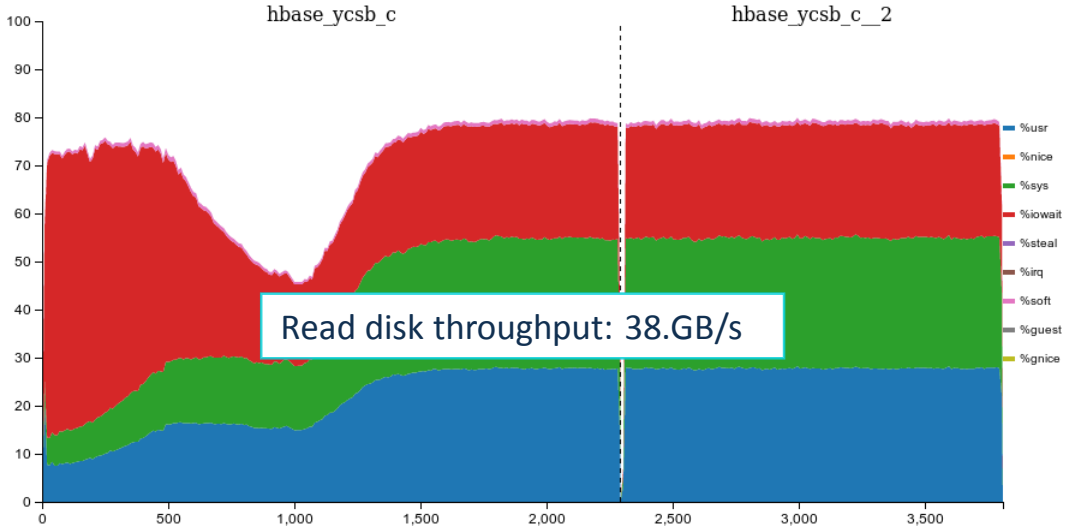


Slowest

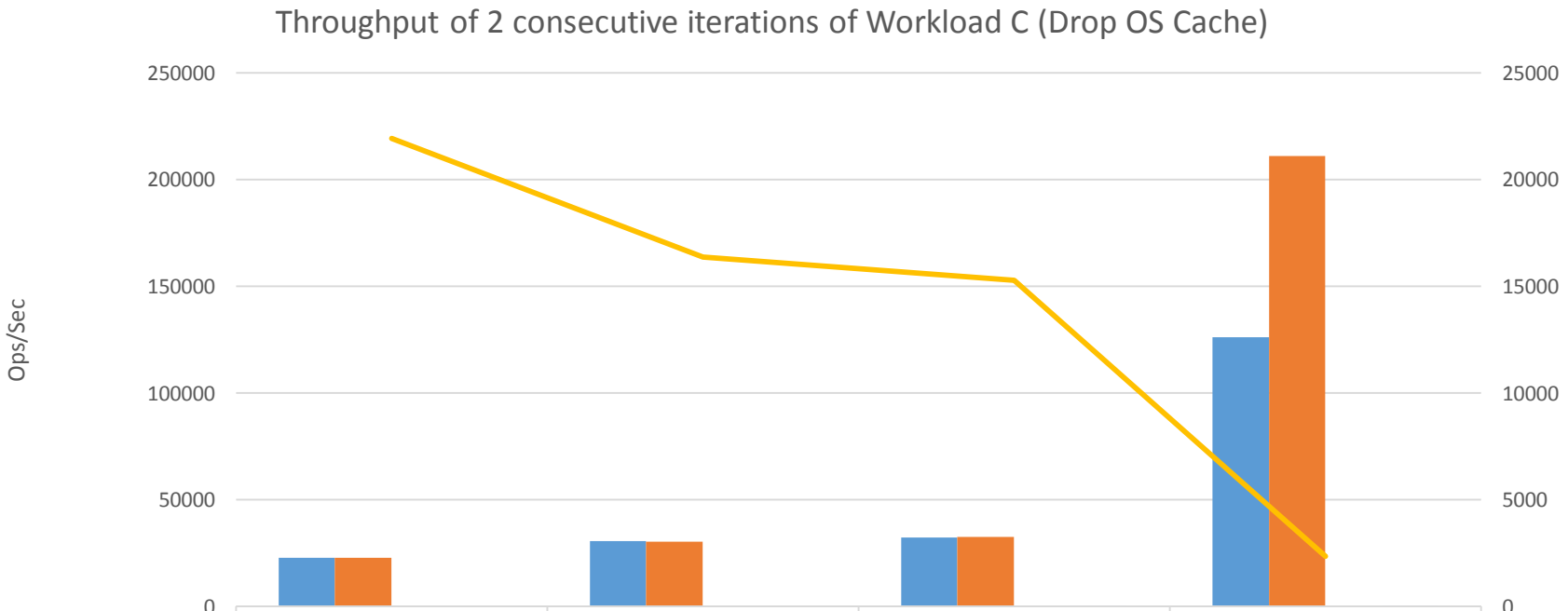
RAM disk (/dev/shm 8GB)



NVMe (250GB)



E1.3: Throughput of the 4 configurations
(Drop OS buffer cache)



	Baseline	BucketCache Offheap	BucketCache RAM disk	BucketCache NVMe
WorkloadC_run1	22780	30593	32447	126306
WorkloadC_run2	22770	30469	32617	210976
Speeup (run3)	1	1.34	1.43	9.27
Cache Time %	-0.1	-0.1	0.5	67
Latency µs (run2)	21924	16375	15293	2361

Results:

- Ops/Sec improve only with NVMe up to 9X
- RAMd performs 1.43X better this time
- First run same on baseline and RAMd
  - But RAMd worked fine
- Having a larger sized BC improves performance over RAMd

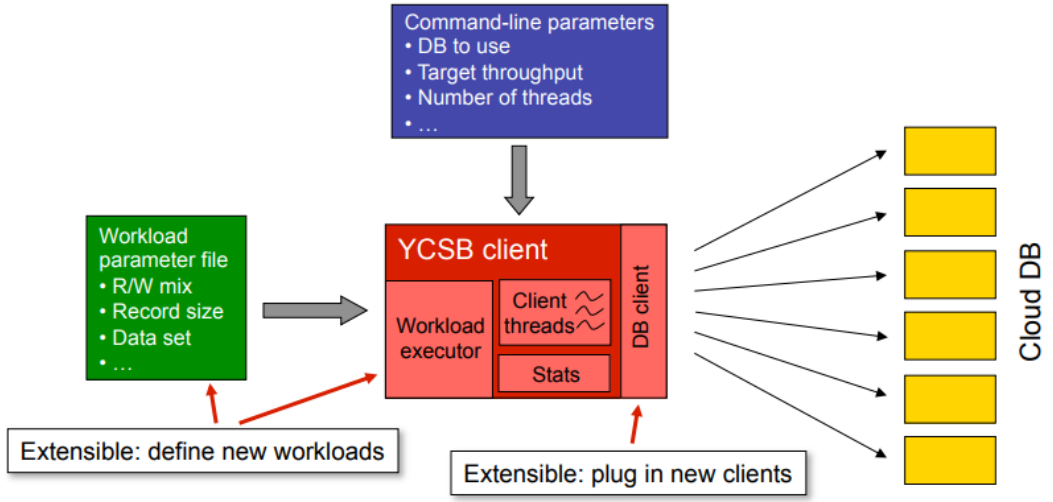
Higher is better for Ops (Y1), lower for latency (Y2).
Dropping cache every 10 secs.

Experiment 2: All workloads (-E)

YCSB workloads A-D, F: 25M records, 1000 threads

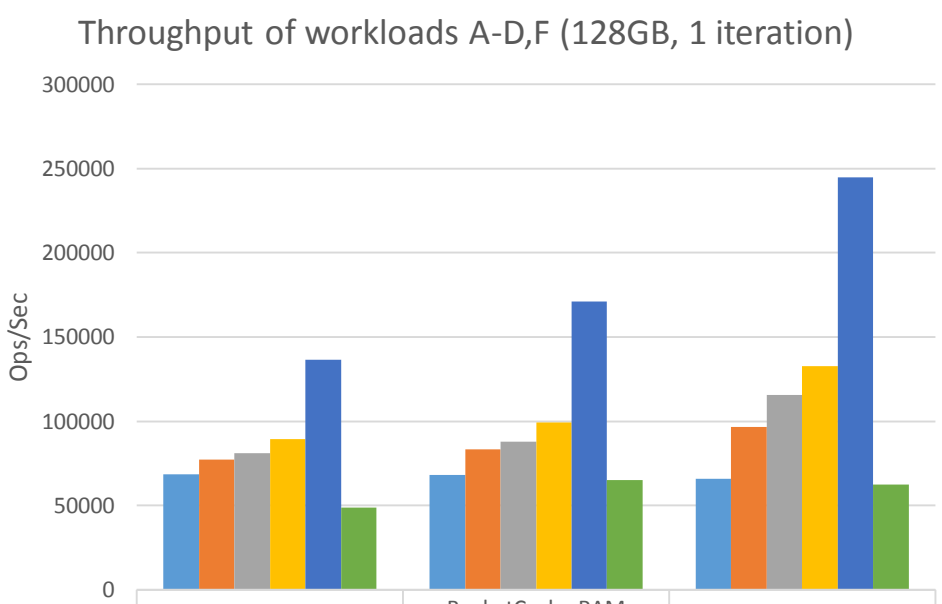
Benchmark suite: The Yahoo! Cloud Serving Benchmark (YCSB)

- Open source specification and kit, for comparing NoSQL DBs. (since 2010)
- Core workloads:
  - A: Update heavy workload
    - 50/50 R/W.
  - B: Read mostly workload
    - 95/5 R/W mix.
  - C: Read only
    - 100% read.
  - D: Read latest workload
    - Inserts new records and reads them
  - Workload E: Short ranges (Not used, takes too long to run SCAN type)
    - Short ranges of records are queried, instead of individual records
  - F: Read-modify-write
    - read a record, modify it, and write back.

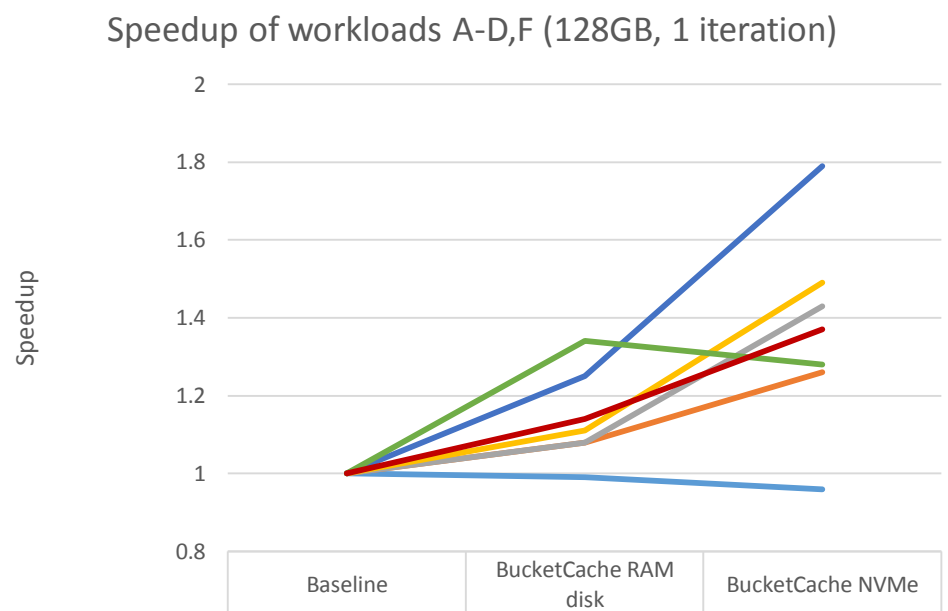


https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads

E2.1: Throughput and Speedup **ALL** (128GB RAM)



	Baseline	BucketCache RAM disk	BucketCache NVMe
Datagen	68502	67998	65933
WL A	77049	83379	96752
WL B	80966	87788	115713
WL C	89372	99403	132738
WL D	136426	171123	244759
WL F	48699	65223	62496



	Baseline	BucketCache RAM disk	BucketCache NVMe
Speedup Data	1	0.99	0.96
Speedup A	1	1.08	1.26
Speedup B	1	1.08	1.43
Speedup C	1	1.11	1.49
Speedup D	1	1.25	1.79
Speedup F	1	1.34	1.28
Total	1	1.14	1.37

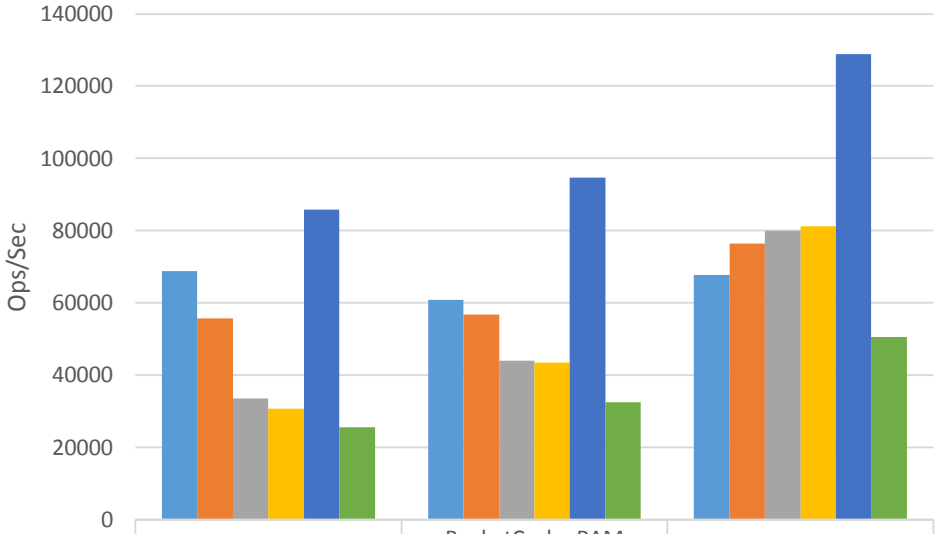
Results:

- Datagen same in all
  - (write-only)
- Overall: Ops/Sec improve with BC
  - RAMd 14%
  - NVMe 37%
- WL D gets higher speedup with NVMe
- WL F 6% faster on RAMd than in NVMe
- Need to run more iterations to see max improvement

Higher is better

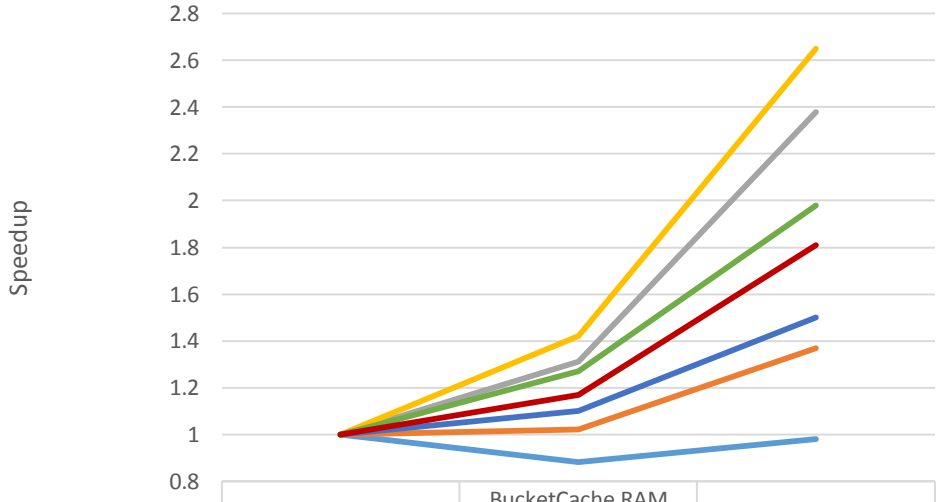
E2.1: Throughput and Speedup **ALL**
(32GB RAM)

Throughput of workloads A-D,F (128GB, 1 iteration)



	Baseline	BucketCache RAM disk	BucketCache NVMe
Datagen	68821	60819	67737
WL A	55682	56702	76315
WL B	33551	43978	79895
WL C	30631	43420	81245
WL D	85725	94631	128881
WL F	25540	32464	50568

Speedup of workloads A-D,F (128GB, 1 iteration)



	Baseline	BucketCache RAM disk	BucketCache NVMe
Speedup Data	1	0.88	0.98
Speedup A	1	1.02	1.37
Speedup B	1	1.31	2.38
Speedup C	1	1.42	2.65
Speedup D	1	1.1	1.5
Speedup F	1	1.27	1.98
Total	1	1.17	1.81

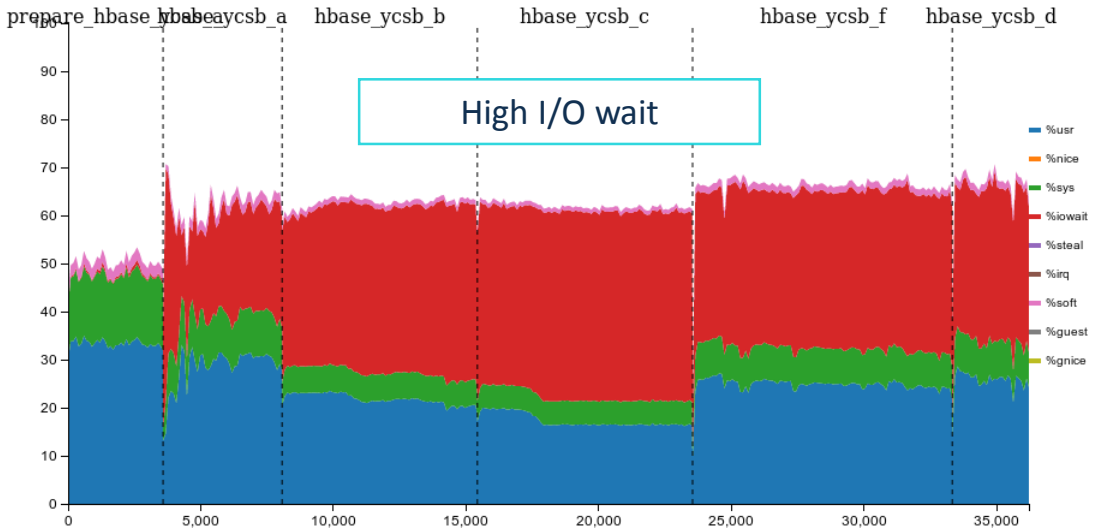
Results:

- Datagen slower with the RAMd (less OS RAM)
  Overall: Ops/Sec improve with BC
  - RAMd 17%
  - NVMe 87%
- WL C gets higher speedup with NVMe
- WL F now faster with NVMe
- Need to run more iterations to see max improvement

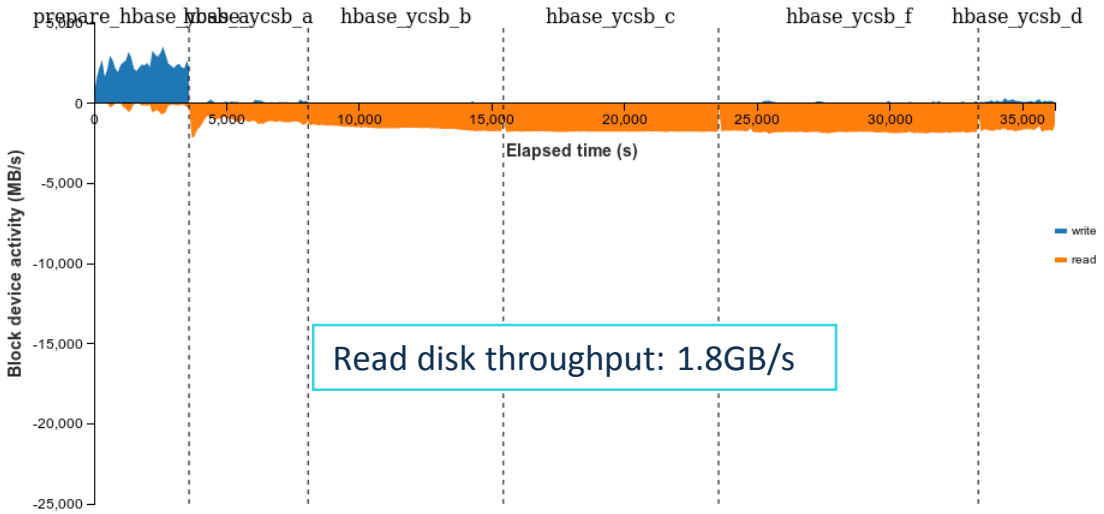
Higher is better

E2.1: **CPU** and **Disk** for ALL (32GB RAM)

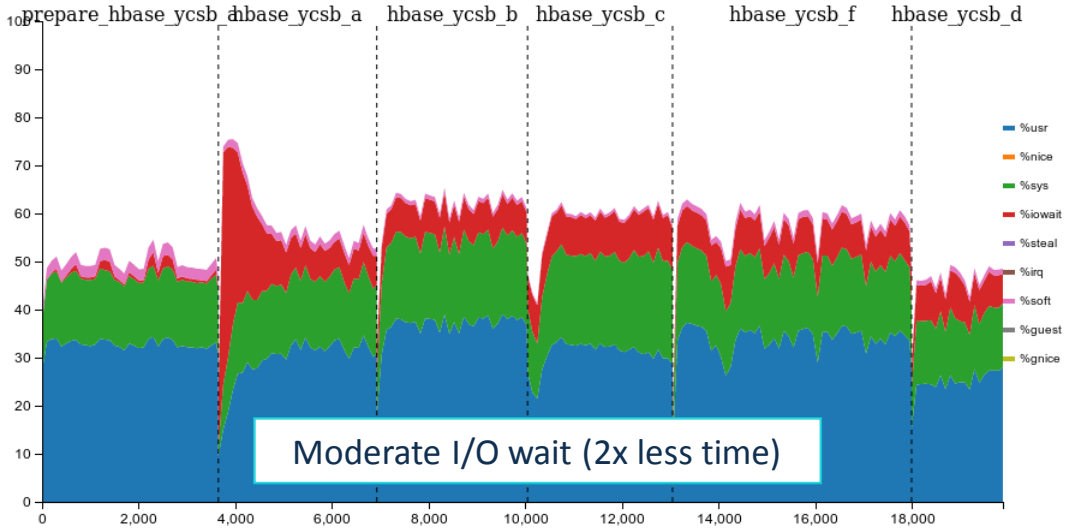
Baseline CPU %



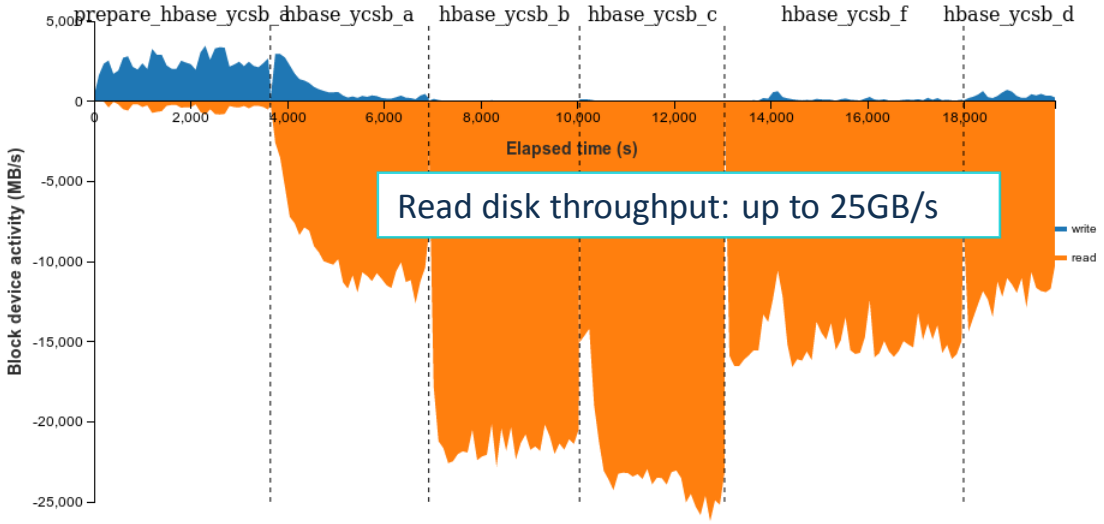
Baseline Disk R/w



NVMe CPU%

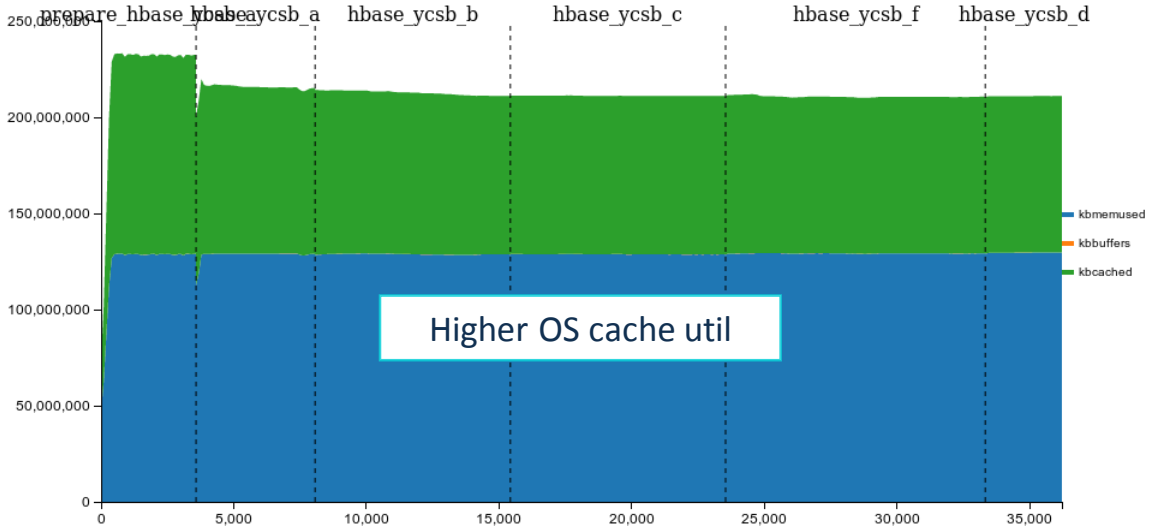


NVMe Disk R/W

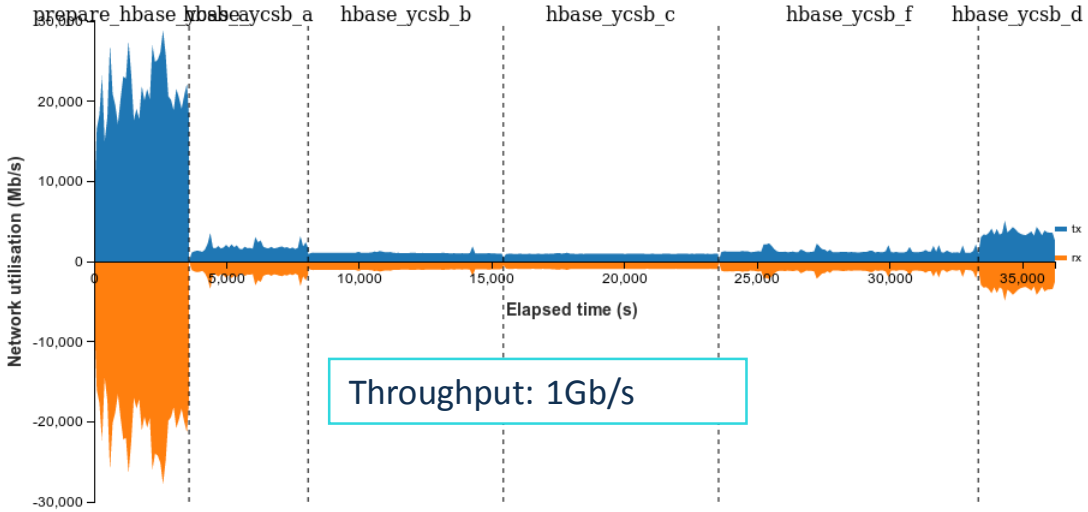


E2.1: **NET** and **MEM** ALL (32GB RAM)

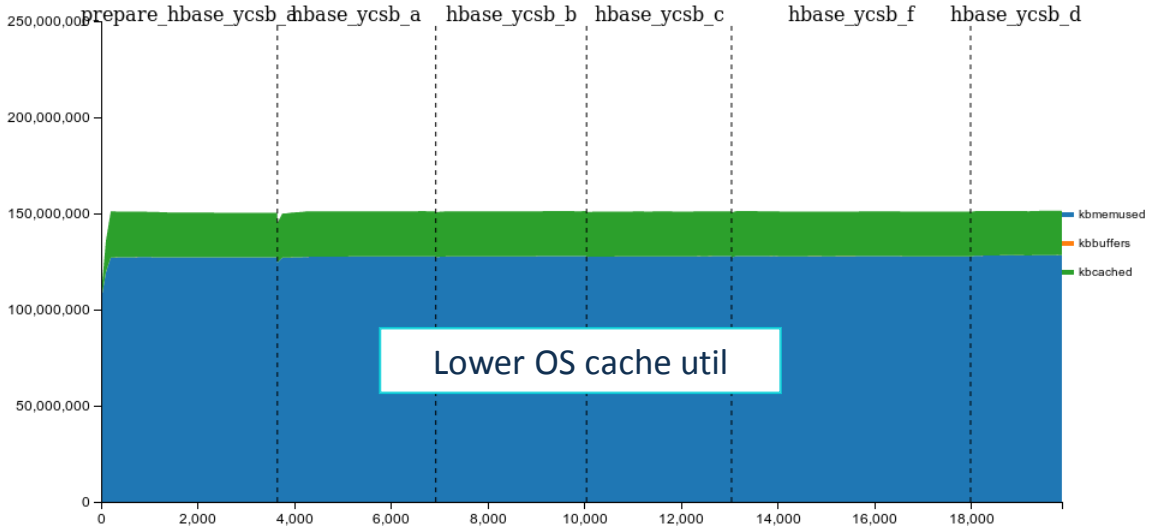
Baseline MEM



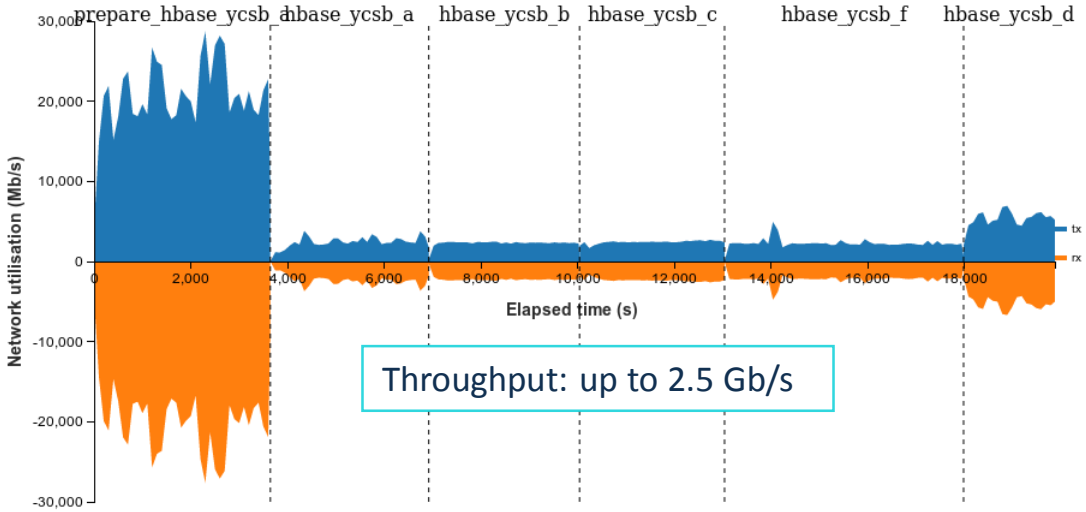
Baseline NET R/W



NVMe MEM

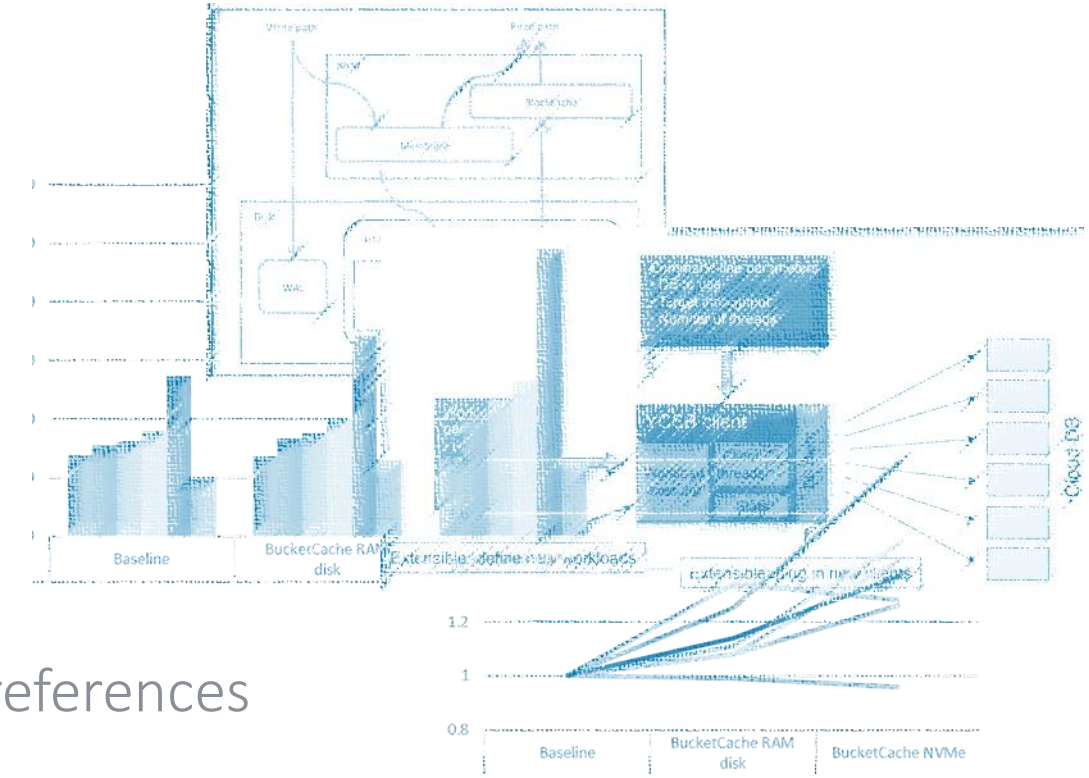


NVMe NET R/W



Summary

Lessons learned, findings, conclusions, references

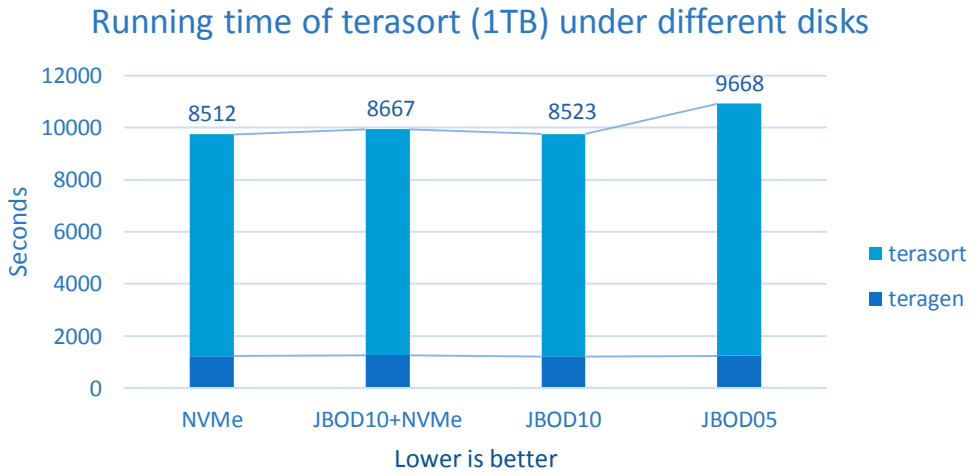


Bucket Cache results recap (medium sized WL)

- Full cluster (128GB RAM / node)
  - WL-C up to **2.7x** speedup (warm cache)
  - Full benchmark (CRUD) from 0.3 to 0.9x speedup (cold cache)
- Limiting resources
  - 32GB RAM / node
    - WL-C NVMe gets up to **8X** improvement (warm cache)
      - Other techniques failed/poor results
    - Full benchmark between **0.4 and 2.7x** speedup (cold cache)
  - Drop OS cache WL-C
    - Up to **9x** with NVMe, only < 0.5x with other techniques (warm cache)
- Latency reduces significantly with cached results
- Onheap BC not recommended
  - just give more RAM to BlockCache

Open challenges / Lessons learned

- Generating app level workloads that **stresses newer HW**
  - At acceptable time / cost
- Still need to
  - Run micro-benchmarks
    - Per DEV, node, cluster
  - Large working sets
    - > RAM (128GB / Node)
    - > NMVe (1.6TB / Node)
- OS buffer cache **highly effective**
  - at least with HDFS and HBase
  - Still, a RAM app "L2 cache" is able to speedup
    - App level LRU more effective
      - YCSB **Zipfian distribution** (popular records)
  - The larger the WL, higher the gains
    - Can be simulated by limiting resources or dropping caches effectively



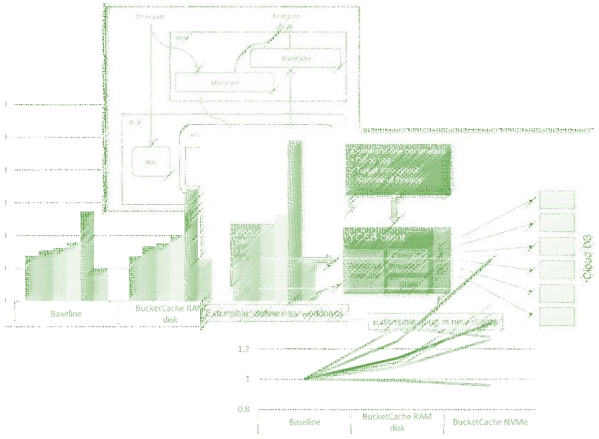
To conclude…

- **NVMe** offers **significant BW and Latency improvement** over SAS/SATA, **but**
  - **JBODs still perform well for seq R/W**
  - Also **cheaper €/TB**
  - Big Data apps still **designed for rotational** (avoid random I/O)
- Full tiered-storage support is missing by Big Data frameworks
  - Byte addressable vs. block access
    - Research shows improvements
  - Need to rely on external tools/file systems
    - Alluxio (Tachyon), Triple-H, New file systems (SSDFS), …
- Fast devices speedup, but still caching is the simple use case…

References

- ALOJA
  - http://aloja.bsc.es
  - https://github.com/aloja/aloja
- Bucket cache and HBase
  - BlockCache (and Bucket Cache 101) http://www.n10k.com/blog/blockcache-101/
  - Intel brief on bucket cache: http://www.intel.com/content/dam/www/public/us/en/documents/solution-briefs/apache-hbase-block-cache-testing-brief.pdf
  - http://www.slideshare.net/larsgeorge/hbase-status-report-hadoop-summit-europe-2014
  - HBase performance: http://www.slideshare.net/bijugs/h-base-performance
- Benchmarks
  - FIO https://github.com/axboe/fio
  - Brian F. Cooper, et. Al. 2010. Benchmarking cloud serving systems with YCSB. http://dx.doi.org/10.1145/1807128.1807152

Evaluating NVMe drives for accelerating HBase



Thanks, questions?

Follow up / feedback : Nicolas.Poggi@bsc.es

Twitter: @ni_po

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

aloja

FIO commands:

- Sequential read
  - fio --name=read --directory=${dir} --ioengine=libaio --direct=1 --bs=${bs} --rw=read --iodepth=${iodepth} --numjobs=1 --buffered=0 --size=2gb --runtime=30 --time_based --randrepeat=0 --norandommap --refill_buffers --output-format=json
- Random read
  - fio --name=randread --directory=${dir} --ioengine=libaio --direct=1 --bs=${bs} --rw=randread --iodepth=${iodepth} --numjobs=${numjobs} --buffered=0 --size=2gb --runtime=30 --time_based --randrepeat=0 --norandommap --refill_buffers --output-format=json
- Sequential read on raw devices:
  - fio --name=read --filename=${dev} --ioengine=libaio --direct=1 --bs=${bs} --rw=read --iodepth=${iodepth} --numjobs=1 --buffered=0 --size=2gb --runtime=30 --time_based --randrepeat=0 --norandommap --refill_buffers --output-format=json

- Sequential write
  - fio --name=write --directory=${dir} --ioengine=libaio --direct=1 --bs=${bs} --rw=write --iodepth=${iodepth} --numjobs=1 --buffered=0 --size=2gb --runtime=30 --time_based --randrepeat=0 --norandommap --refill_buffers --output-format=json
- Random write
  - fio --name=randwrite --directory=${dir} --ioengine=libaio --direct=1 --bs=${bs} --rw=randwrite --iodepth=${iodepth} --numjobs=${numjobs} --buffered=0 --size=2gb --runtime=30 --time_based --randrepeat=0 --norandommap --refill_buffers --output-format=json
- Random read on raw devices:
  - fio --name=randread --filename=${dev} --ioengine=libaio --direct=1 --bs=${bs} --rw=randread --iodepth=${iodepth} --numjobs=${numjobs} --buffered=0 --size=2gb --runtime=30 --time_based --randrepeat=0 --norandommap --refill_buffers --offset_increment=2gb --output-format=json

https://github.com/axboe/fio