SDC19
September 23-26, 2019
Santa Clara, CA

Spark-PMoF: Accelerating Bigdata Analytics with Persistent Memory over Fabrics

Haodong Tang (haodong.tang@intel.com)
Jian Zhang (jian.zhang@intel.com)
Yuan Zhou (yuan.zhou@intel.com)

Agenda

- Background and motivation
- Persistent Memory over Fabrics (PMoF)
- Spark-PMoF design
- Spark-PMoF performance evaluation
- Next-step

Background and motivation

Challenges of scaling Hadoop* Storage

BOUNDED Storage and Compute resources on Hadoop Nodes brings challenges

Data Capacity

Silos

Costs

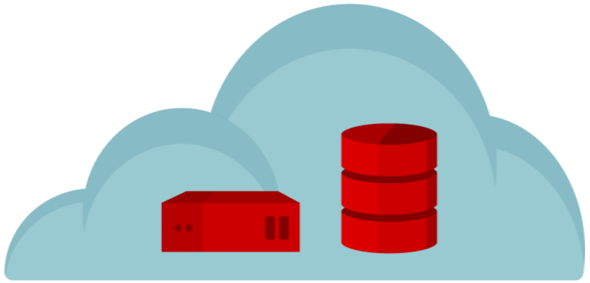
Performance & efficiency

Typical Challenges

Data/Capacity	Multiple Storage Silos
Space, Power, Utilization	Inadequate Performance
Upgrade Cost	Provisioning and Configuration

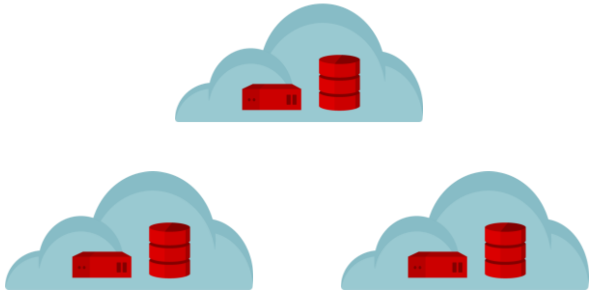
*Other names and brands may be claimed as the property of others.

Discontinuity in bigdata infrastructure makes different solution



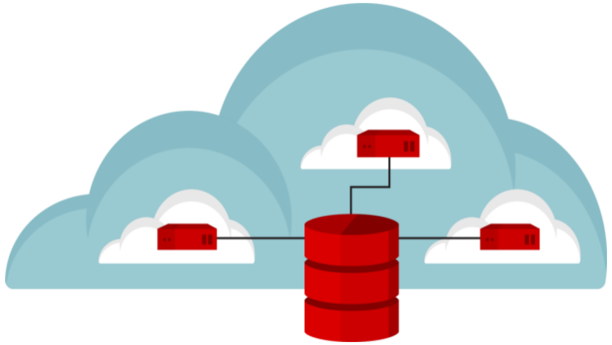
SINGLE LARGE CLUSTER

Get a bigger cluster for many teams to share.



MULTIPLE SMALL CLUSTERS

Give each team their own dedicated cluster, each with a copy of PBs of data.



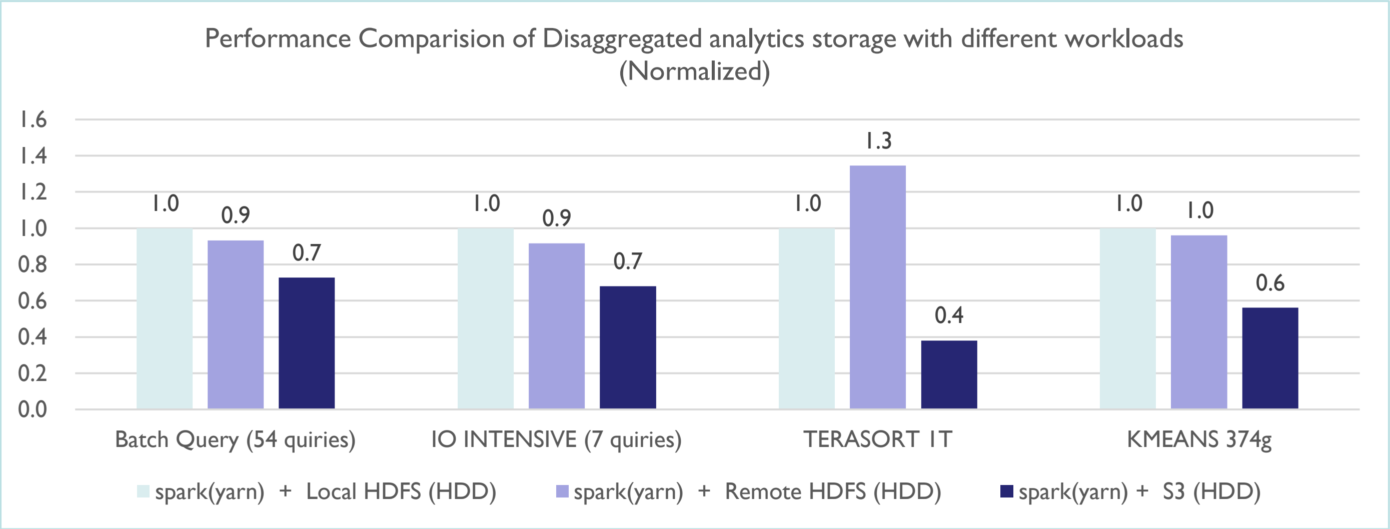
ON DEMAND ANALYTIC CLUSTERS

Give teams ability to spin-up/spin-down clusters which can share data sets.

Benefits of compute and storage disaggregation

Independent scale of CPU and storage capacity	Single copy of data	Enable Agile application development	Hybrid cloud deployment	Simple and flexible software management
• Right size HW for each layer • Reduce resource wastage • Cost saving	• Multiple compute cluster share common data repo/lake • Simplified data management • Reduced provisioning overhead • Improve security	• In-memory cloning • Snapshot service • Quick & efficient copies	• Mix and match resources depending on workload nature and life cycle	• Avoid software version management • Upgrade compute software only

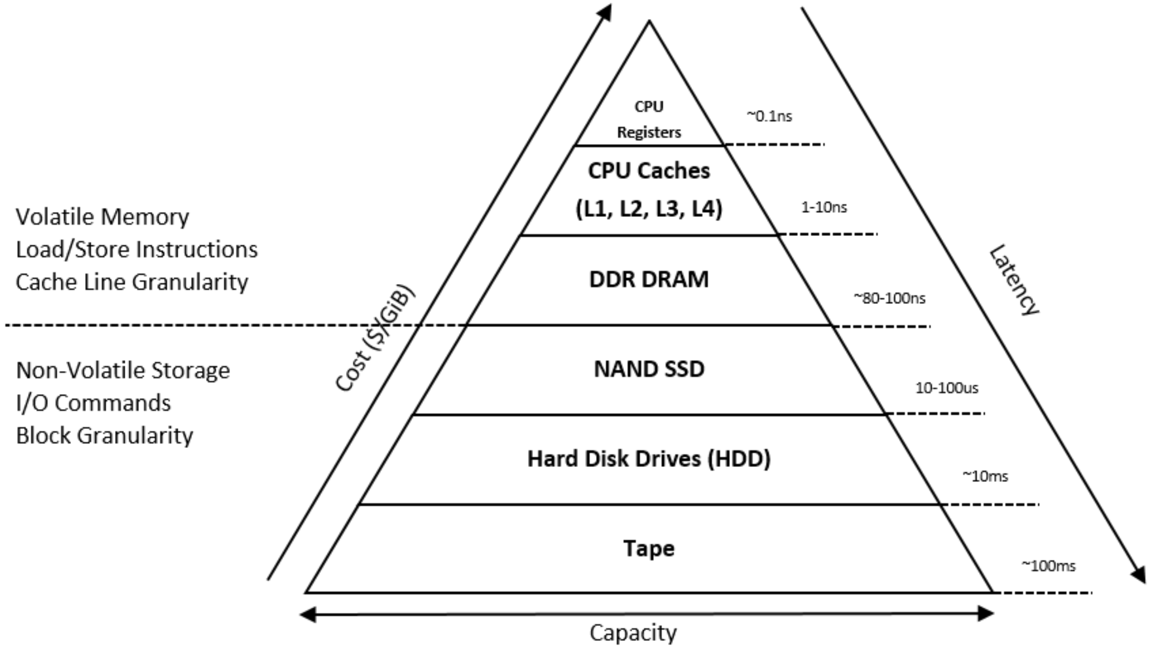
Disaggregation leads to performance regression



- Storage disaggregation leads to performance regression
  - Up to 10% for remote HDFS, Terasort performance is higher as usable memory increased
  - Up to 60% for S3 object storage (optimized results with tunings)
- One important cause for the performance gap: s3a does not support Transactional Writes
  - Most of bigdata software (Spark, Hive) relies on HDFS's atomic rename feature to support atomic writes
  - During job submit, *commit protocol* is used to specify how results should be written at the end of job
    - First stage task output into temporary locations, and only moving (*renaming*) data to final location upon task or job completion
    - S3a implements this with: COPY+DELETE+HEAD+POST
- The gap in public cloud will be much smaller
  - It is not an on-premise configuration
  - Compute are running in side VMs/containers, while HDFS was running on elastic block volumes

Persistent memory over fabrics (PMoF)

Why persistent memory



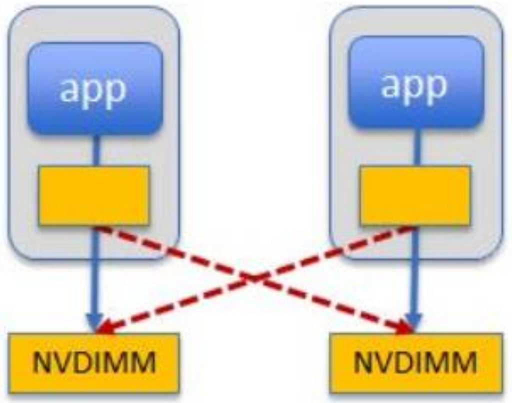
- Persistent Memory
  - PMEM represents a new class of memory and storage technology architected specifically for data center usage
  - Combination of high-capacity, affordability and persistence.

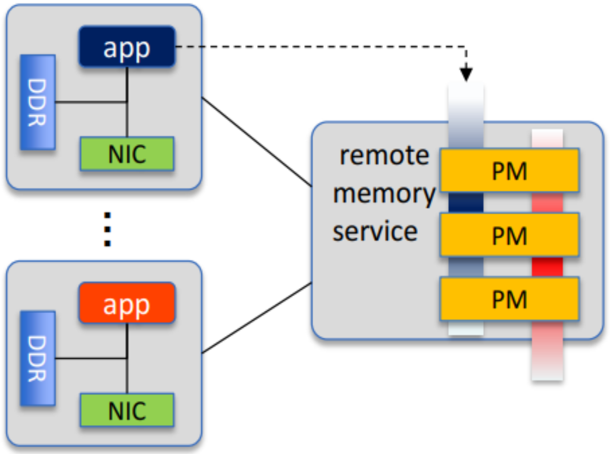
Picture source: https://docs.pmem.io/getting-started-guide/introduction

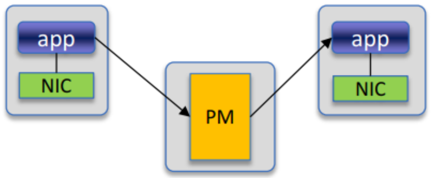
Why RDMA

- Remote persistent memory requirement
  - PM is really fast (Especially for Read)
    - Needs ultra low-latency networking
  - PM has very high bandwidth per socket
    - Needs ultra efficient protocol, transport offload, high BW
  - Remote access must not add significant latency
    - Network switches & adaptors deliver predictability, fairness, zero packet loss
- RDMA offers
  - Low latency
  - High BW
    - zero-copy, kernel bypass, HW offered one side memory to remote memory operations
  - Reliable credit base data and control delivered by HW
    - Network resiliency, scale-out

Persistent Memory over Fabrics (PMoF)







- Replicate Data in local PM across Fabric and Store in remote PM
- DRBD

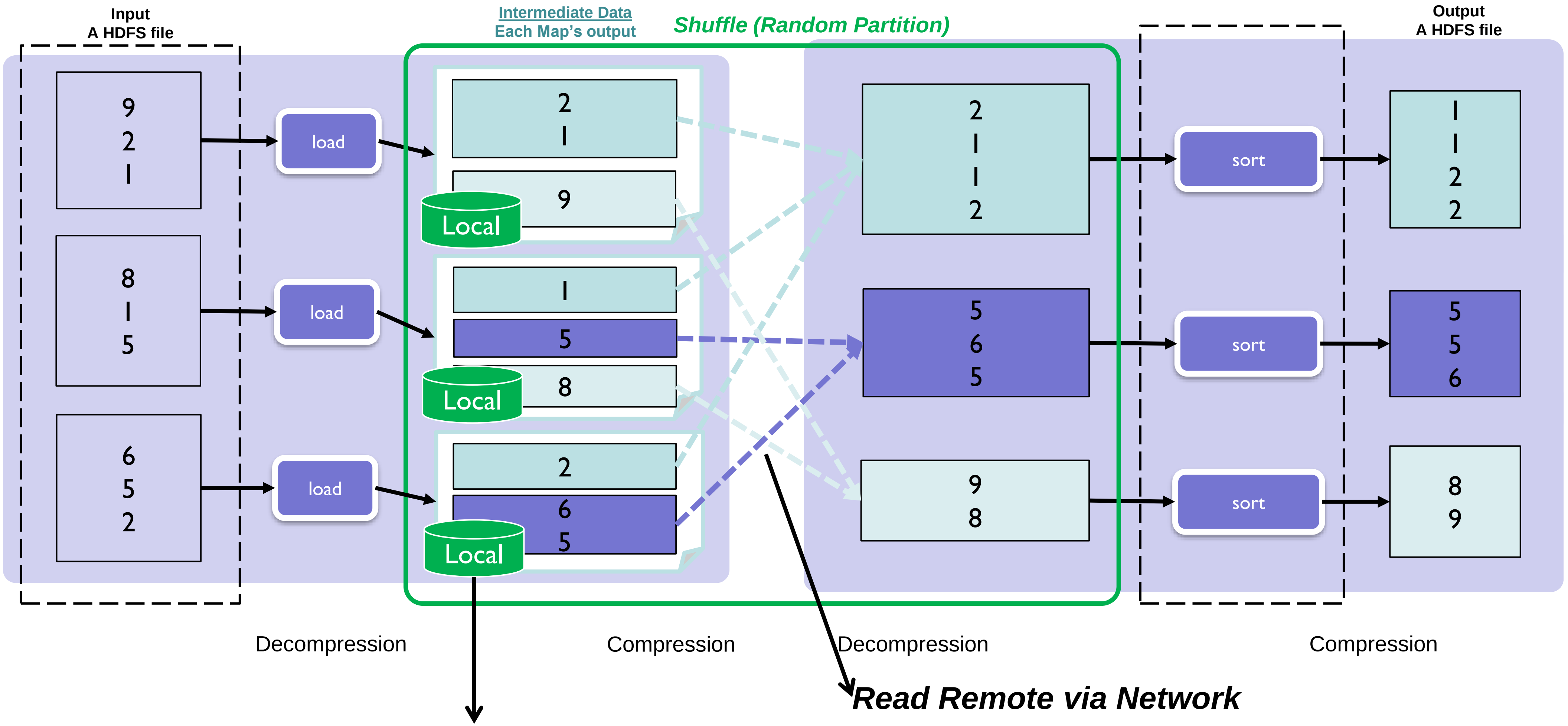
- Expand on-node memory capacity (w/ or w/o persistency) in a disaggregated architecture

- PM holds shared data among distributed application
- Spark-PMoF

*Picture source: https://www.snia.org/sites/default/files/PM-Summit/2018/presentations/05_PM_Summit_Grun_PM_%20Final_Post_CORRECTED.pdf

Spark PMoF Design

Shuffle recap

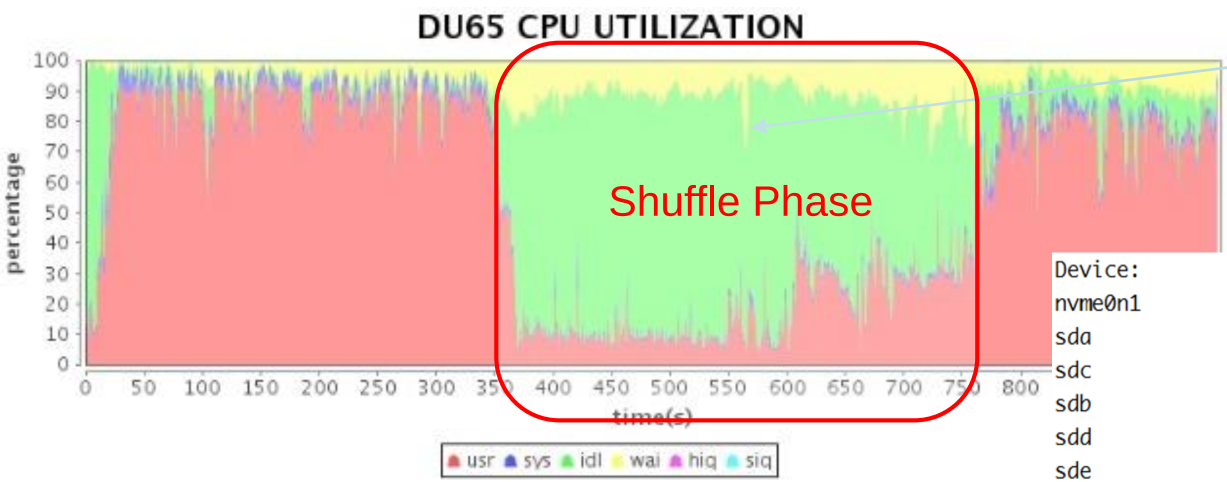


Write Local, can use shuffle service to cache the data.

https://github.com/intel-hadoop/HiBench/blob/master/sparkbench/micro/src/main/scala/com/intel/sparkbench/micro/ScalaSort.scala

Spark Shuffle Bottlenecks – Disk

- **Spark Shuffle (nWeight – a Graph Computation Workload)**
- **Context**: Iterative graph-parallel algorithm, implemented with GraphX, to compute the association for 2 vertices in 2-3 hops distance in the graph. (e.g. recommend a video for my friends' friends)
- **H/W Configuration**: 1+4 cluster / E5 2680 v2@2.8GHz / 192GB DDR3 1600 MHz / 11 HDD, 11 SSD, 1 PCI-E SSD (P3600)
- **S/W Configuration**: Redhat 6.2 / Spark 1.4.1 / Hadoop 2.5.0-CDH5.3.2/Scala 2.10.4
- **Benchmark Analysis**:

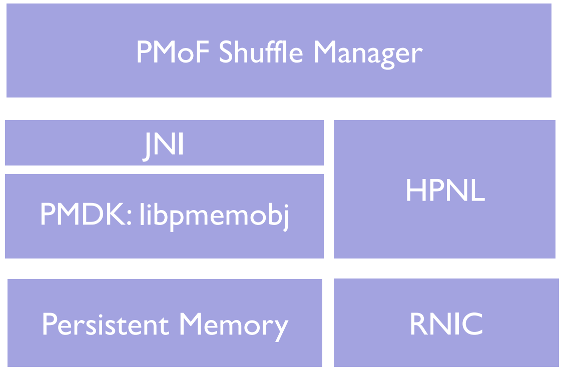


Significant IO bottleneck in Shuffle

Device:	rrqm/s	wrqm/s	r/s	w/s	rkB/s	wkB/s	avgrq-sz	avgqu-sz	await	svctm	%util
nvme0n1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
sda	0.00	0.00	0.00	9.00	0.00	36.00	8.00	0.00	0.00	0.00	0.00
sdc	2.00	0.00	398.00	0.00	37268.00	0.00	187.28	7.09	15.63	2.46	97.90
sdb	19.00	347.00	389.00	8.00	35236.00	1420.00	184.66	10.37	25.28	2.45	97.30
sdd	19.00	0.00	326.00	23.00	29592.00	11776.00	237.07	153.35	137.34	2.87	100.00
sde	0.00	0.00	317.00	0.00	30344.00	0.00	191.44	2.87	9.02	2.54	80.40
sdf	11.00	0.00	267.00	1.00	25332.00	4.00	189.07	12.98	50.60	3.73	100.00
sdg	18.00	332.00	384.00	0.00	34684.00	0.00	180.65	25.56	47.58	2.60	100.00
sdh	4.00	183.00	334.00	5.00	33392.00	752.00	201.44	4.86	14.35	2.69	91.30

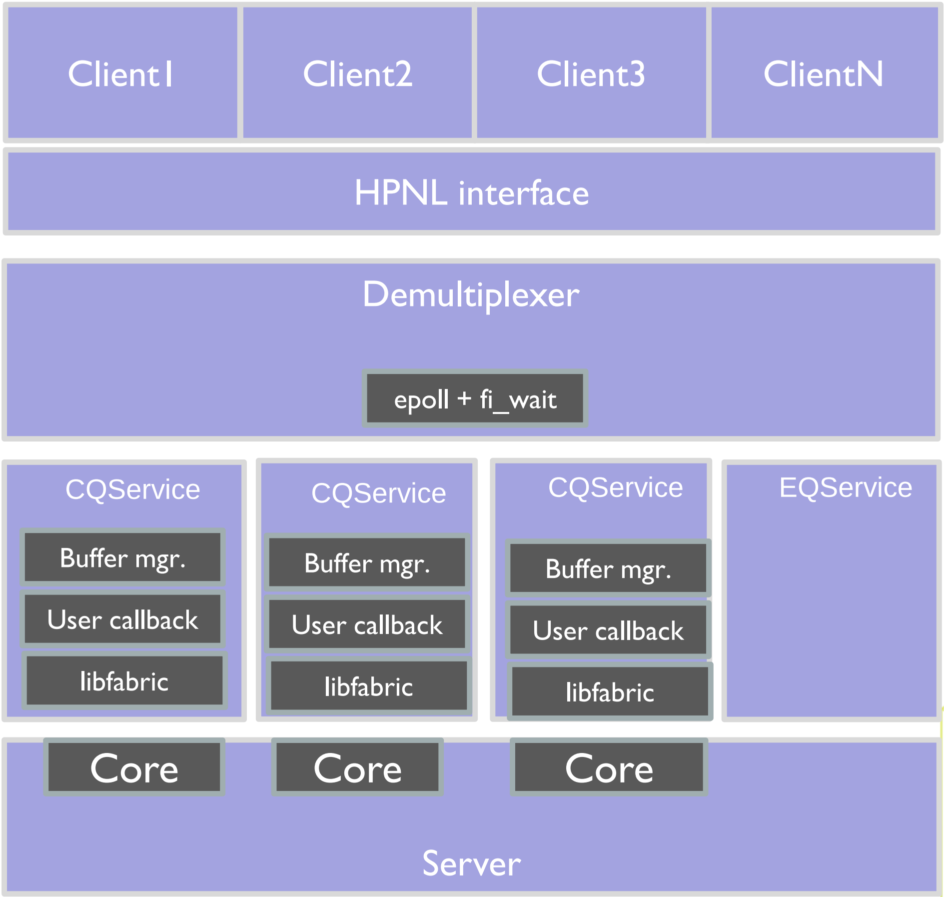
Spark-PMoF design

- A new Spark Shuffle Manager (based on Spark 2.3)
  - Efficient PMDK Java wrapper.
    - Leverage PMDK (libpmemobj) for write
    - ensure data consistency
  - Failover
    - support multiple executor processes to get multiple PMEM namespace in devdax mode and also be able to re-open the same device when failover
  - Pmem based external sorter
    - Support shuffle data spill
    - Support map side combine
  - RDMA
    - Using HPNL (high performance network library) for RDMA networking

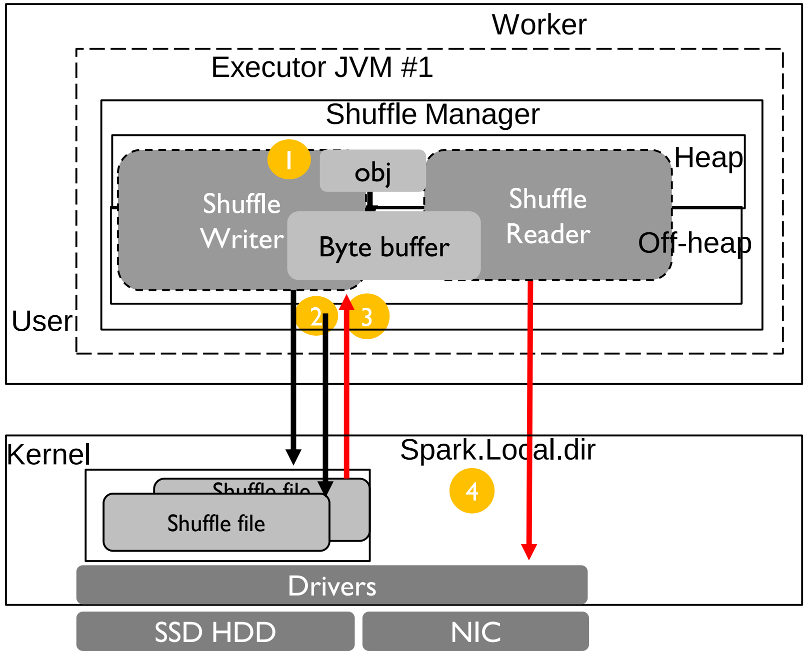


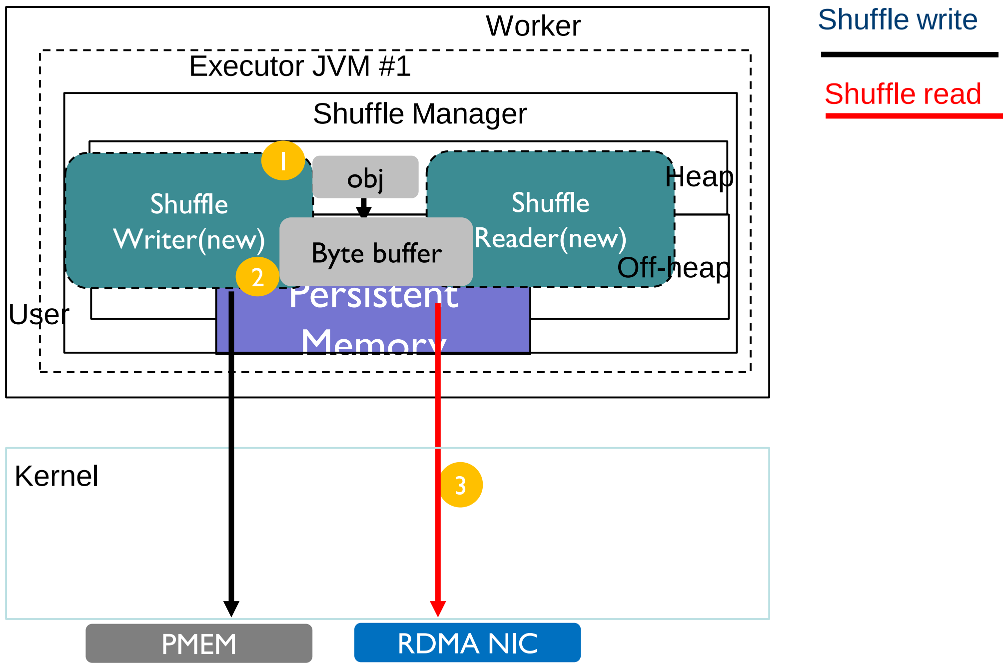
HPNL – High performance network library for bigdata application

- Zero-copy approach
  - Maintain memory pool, no memory copy between HPNL buffer and application buffer.
  - Thanks to RDMA, it supports user-space to kernel-space zero-copy.
- Threading model
  - Implements the Proactor model.
  - Supports thread binding specific core.
- HPNL interface
  - C++ and Java binding.
  - Supports RDMA send, receive, remote read semantics.
  - Pluggable buffer management.
  - Capable of using persistent memory (devdax for now) as RDMA region.
- Open source
  - HPNL is open source in Q2, 2019.



Spark-PMoF design





1. Serialize obj to off-heap memory
2. Write to local shuffle dir
3. Read from local shuffle dir
4. Send to remote reader through TCP-IP

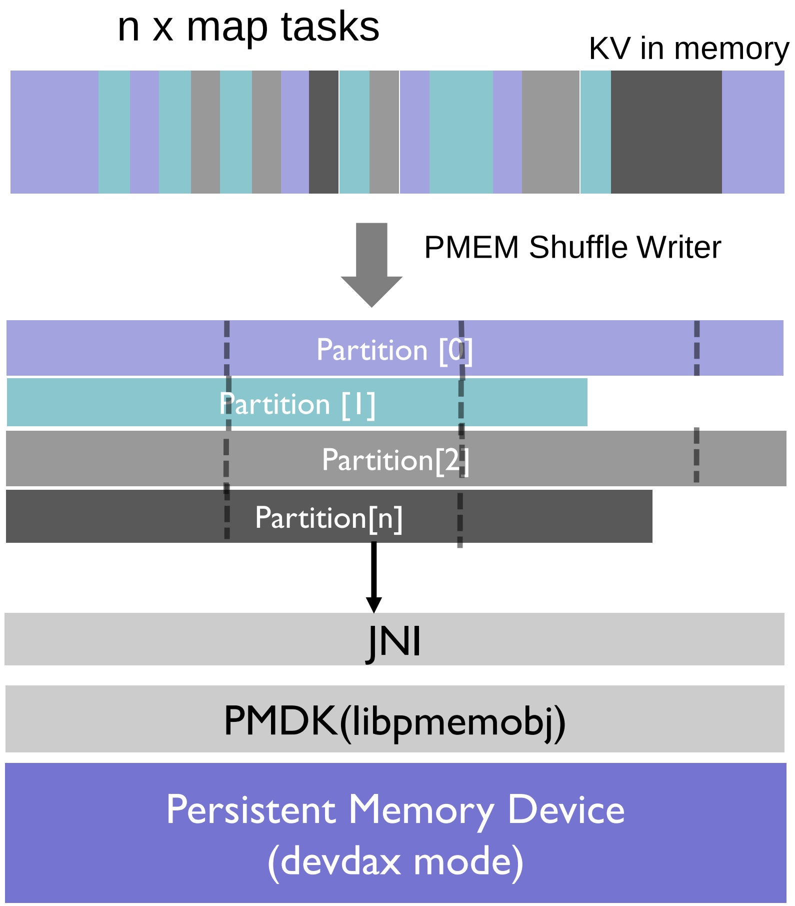
- Lots of context switch
- POSIX buffered read/write on shuffle disk
- TCP/IP based socket send for remote shuffle read

1. Serialize obj to off-heap memory
2. Persistent to PMEM
3. Read from remote PMEM through RDMA, PMEM is used as RDMA memory buffer

- No context switch
- Efficient read/write on PMEM
- RDMA read for remote shuffle read

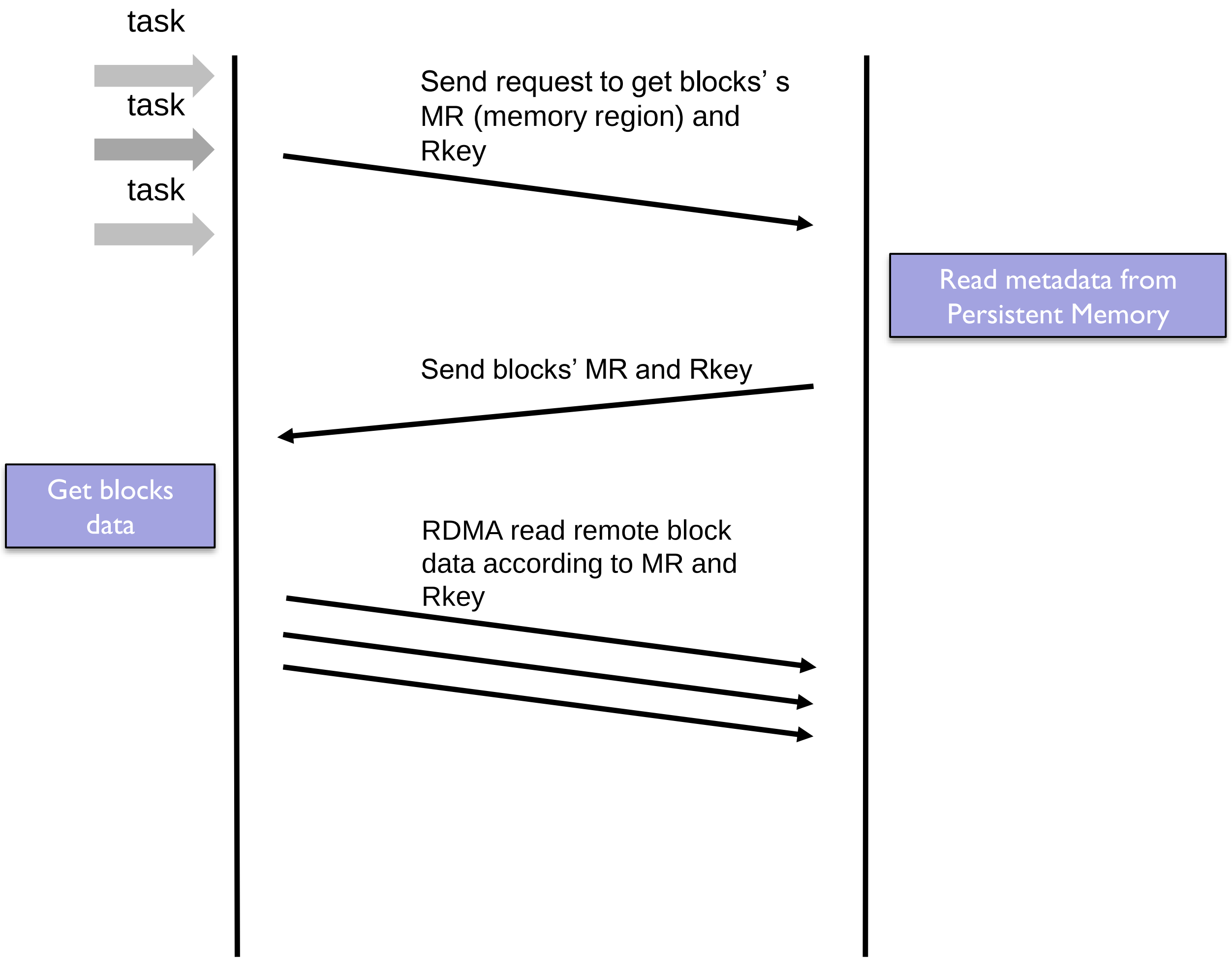
Disk IO traffic in Shuffle map stage

- Provision Persistent Memory name space in advance.
- No filesystem involvement.
  - Serialized data write to off-heap buffer. Once hit threshold, create a block via libpmemobj then flush shuffle data to Persistent Memory.
  - Append write, only write/read once.
- No index file. Metadata and data are collocated in Persistent Memory.
- Sort in PMEM.



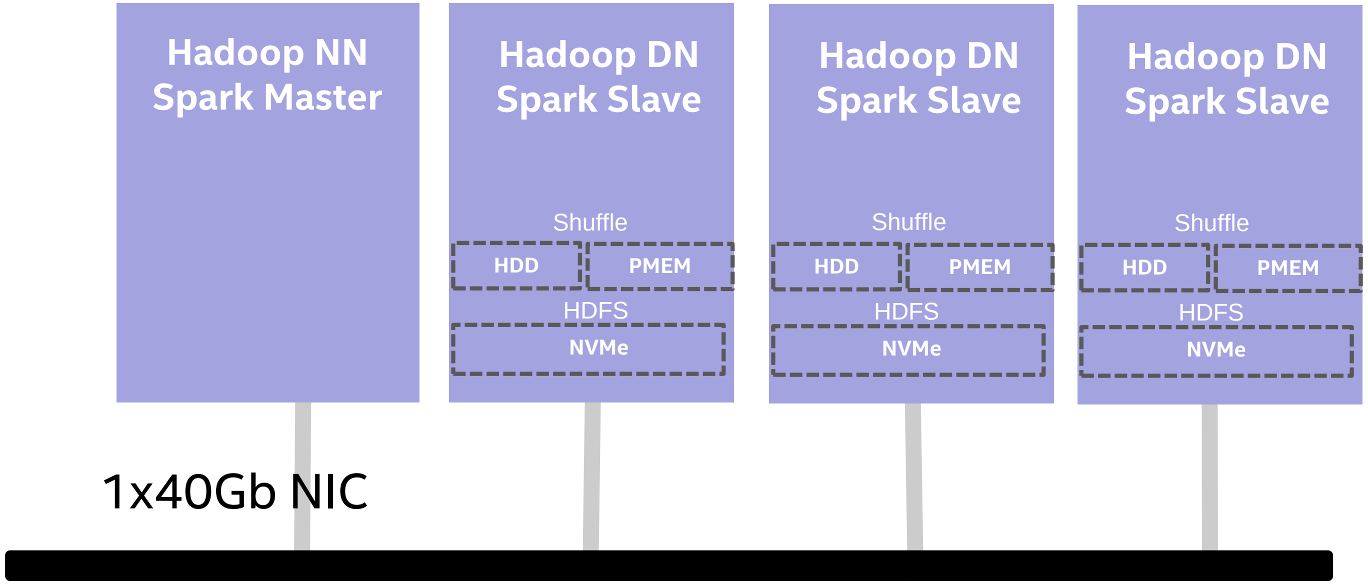
Network traffic in Shuffle reduce stage

- Use RDMA RMA semantics to read data in shuffle reduce stage.
  - 2+n times network transfer per task.
  - Use off-heap memory as RDMA region on one side, and Persistent Memory as RDMA region on the other side.
  - Leverage RDMA to achieve kernel bypass.



Spark PMoF performance evaluation

Benchmark configuration



3 Node cluster

Hardware:

- Intel® Xeon™ processor Gold 6240 CPU @ 2.60GHz, 384GB Memory
- 1x Mellanox ConnectX-4 40Gb NIC
- Shuffle Devices:
  - 1x HDD for shuffle
  - 4x 128GB Persistent Memory for shuffle
- 4x 1T NVMe for HDFS

Software:

- Hadoop 2.7
- Spark 2.3
- Fedora 27 with WW26 BKC

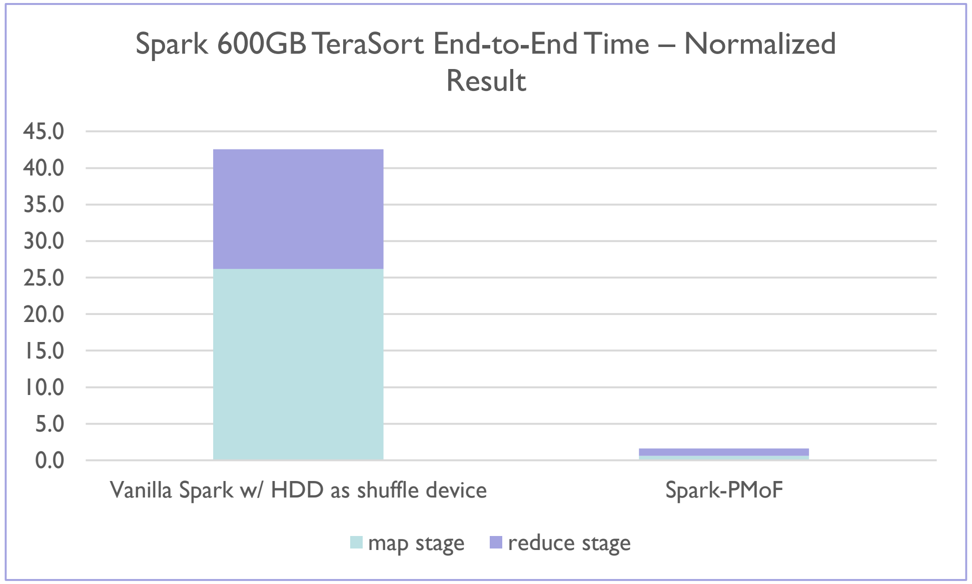
Workloads

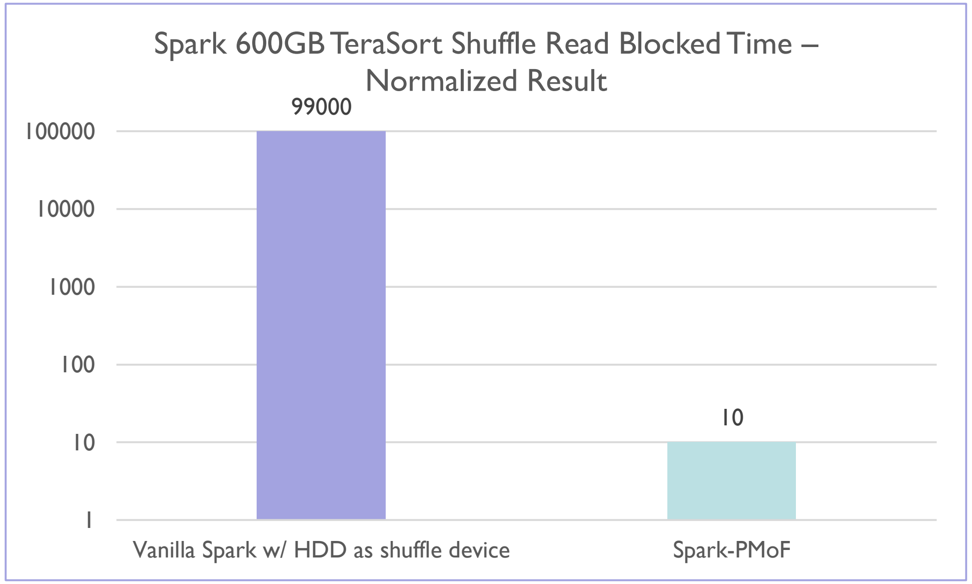
Terasort 600GB

- hibench.spark.master yarn-client
- hibench.yarn.executor.num 12
- yarn.executor.num 12
- hibench.yarn.executor.cores 8
- yarn.executor.cores 8
- spark.shuffle.compress false
- spark.shuffle.spill.compress false
- spark.executor.memory 60g
- spark.executor.memoryoverhead 10G
- spark.driver.memory 80g
- spark.eventLog.compress = false
- spark.executor.extraJavaOptions=-XX:+UseG1GC
- spark.hadoop.yarn.timeline-service.enabled false
- spark.serializer org.apache.spark.serializer.KryoSerializer
- hibench.default.map.parallelism 200
- hibench.default.shuffle.parallelism 1000

Spark PMoF end-to-end time evaluation

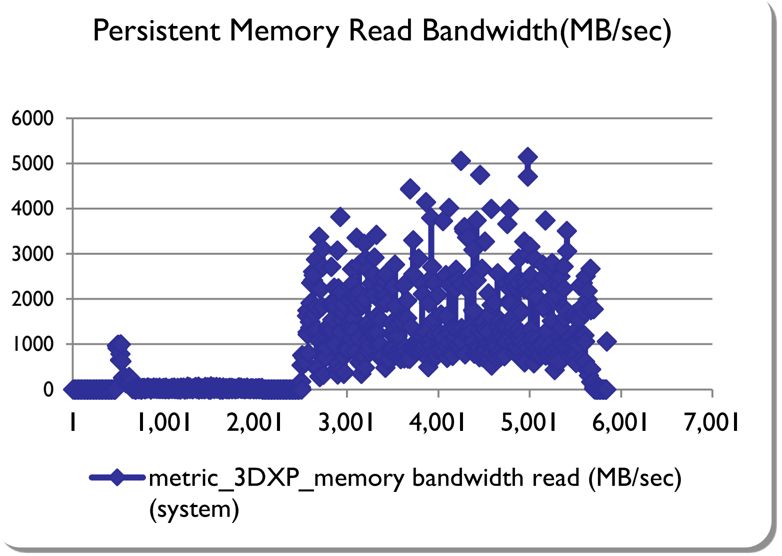
- Vanilla Spark
  - Input/output data to HDFS (NVMe)
  - Shuffle data to local HDD
- Spark-PMoF
  - Input/output data to HDFS (NVMe)
  - Shuffle data to Spark-PMoF
- Spark-PMoF end-to-end time gains: 24.8x.
  - Persistent Memory provides higher write bandwidth per node than HDD.
- Spark-PMoF shuffle remote read latency gains: 9900x.
  - PMoF extremely shorten the remote read latency.
  - PMEM provides higher read bandwidth per node than HDD.
- Optimization headroom
  - Registering PMEM address as RDMA region is time consuming.
  - Need PMEM provisioned on every Spark executor node.
  - Currently just support PMEM devdax with RDMA.

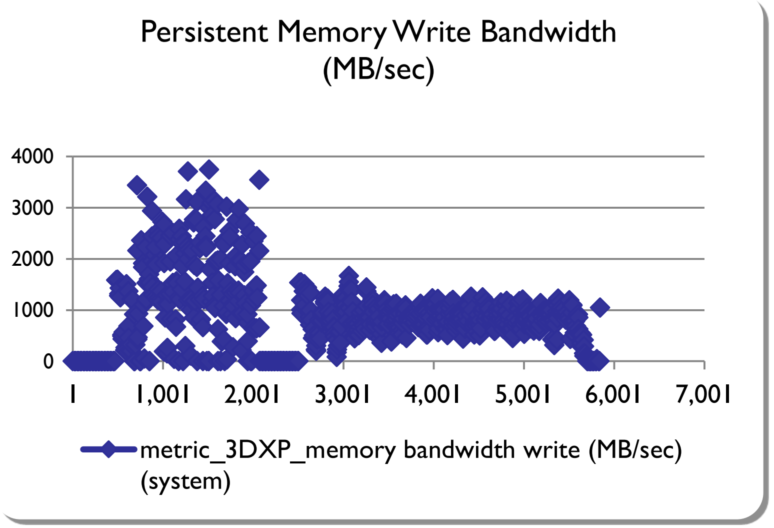




Persistent Memory IO

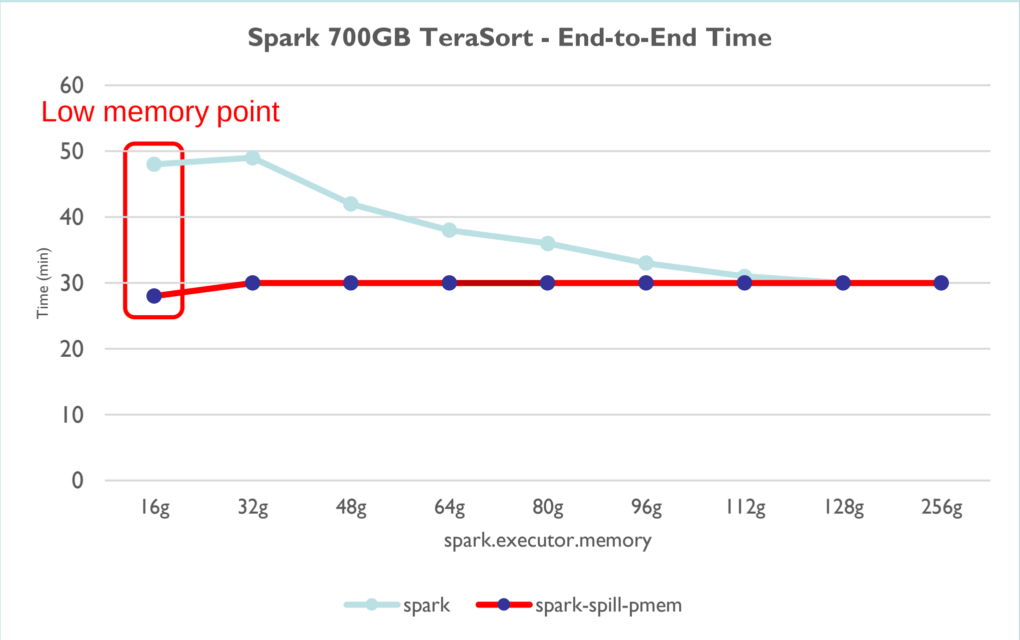
- Deliver up to 4GB/s write bandwidth per node.
  - Didn’t hit PMEM theoretical peak write bandwidth.
  - Performance was limited by read bandwidth from HDFS in map stage.
- Deliver up to 5GB/s read bandwidth per node.
  - Didn’t hit PMEM theoretical peak read bandwidth.
  - Performance was limited by sort operation in reduce stage and write bandwidth to HDFS.





PMoF Memory footprint benefit

- Also enables Spill to persistent memory.
- Spark-PMoF significantly reduces memory footprint by **~4.7x under the same performance**
  - 11 GB persistent memory spill with 16 GB DRAM as executor memory vs 128 GB DRAM as executor memory
- Spark-PMoF shows excellent performance in low memory environment.
  - ~1.7x performance benefit for end-to-to time.
  - ~2.2x performance benefit in reduce stage.
- Spark-PMoF optimized:
  - GC overhead.
  - Shuffle storage IO overhead (mixed read and write IO when spill happens).

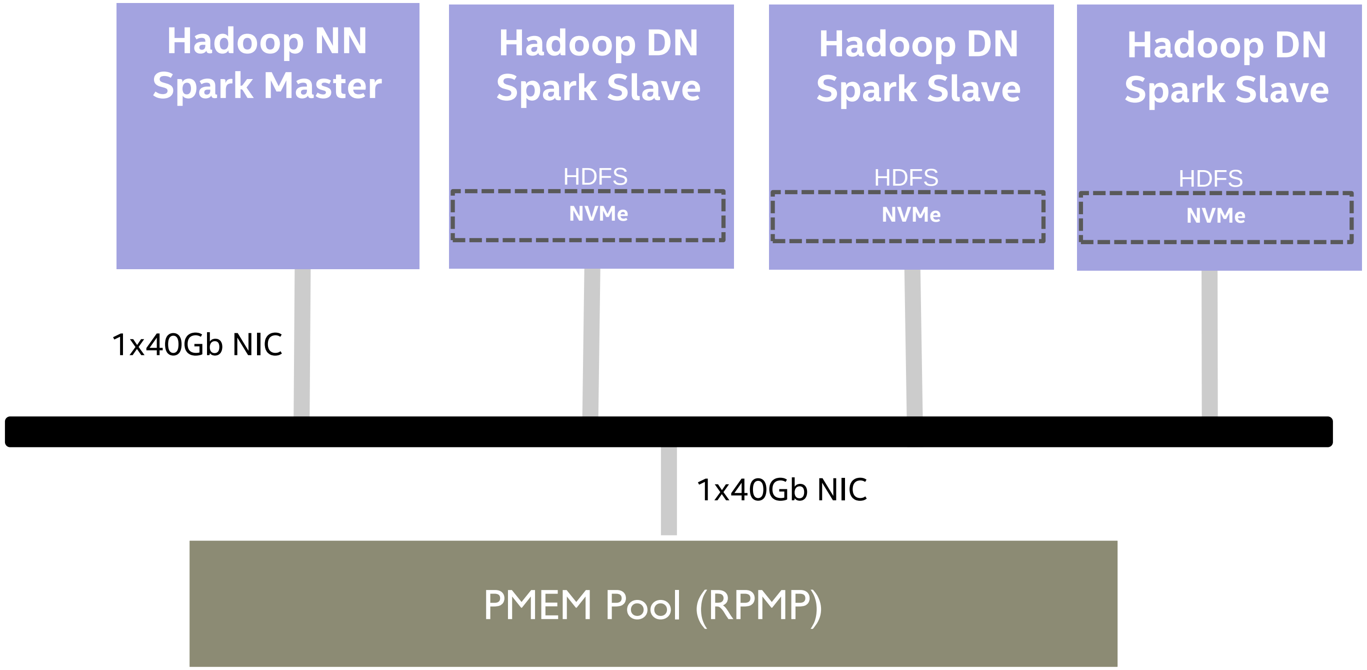


Spark-PMoF performance summary

- PMEM changes the traditional memory/storage hierarchy with high capacity and high bandwidth. PMoF combines PMEM and RDMA technology to provides high bandwidth and ultra-low latency for Spark Shuffle.
- Spark-PMoF is good:
  - If you expect high capacity, high bandwidth and low latency Spark Shuffle solution.
  - If the Spark Shuffle is DRAM based. Migrating DRAM based shuffle to PMoF based shuffle is more cost-effective and brings comparable performance benefit.
- Spark-PMoF is not needed:
  - If the Spark Shuffle is not IO-intensive, disk IO and latency is not the bottleneck.

PMoF in other scenarios & Next step

External PM Pool for Spark Shuffle



- Working on extending Spark-PMoF to Spark Shuffle with RPMP (Remote Persistent Memory Pool) to solve some of issues addressed before.
- Current status
  - Able to saturate 40GB RDMA NIC, will try100GB RDMA NIC in the near future.
- An independent shuffle layer is becoming increasingly important for large CSPs to deliver consistent latency for critical workloads

Summary

Summary

- A new high performance, low latency In Memory Data Accelerator will be needed to close the performance gap and improve scale out capabilities
- Persistent Memory over Fabrics extending PM new usage mode to new scenarios
- Leveraging persistent memory and RDMA, Spark PMoF enables a high performance, low latency shuffle solution to accelerate spark shuffle, and delivers 24.8x performance improvement for TeraSort compared with traditional HDD based shuffle and brings three orders of magnitude reduction in shuffle block ready latency
- PMoF components integration to Spark external shuffle services and RL framework to be explored.

Notices and Disclaimers

- No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.
- Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.
- This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.
- The products and services described may contain defects or errors known as errata which may cause deviations from published specifications. Current characterized errata are available on request.
- Intel, the Intel logo, Xeon, Optane, Optane DC Persistent Memory are trademarks of Intel Corporation in the U.S. and/or other countries.
- *Other names and brands may be claimed as the property of others
- © Intel Corporation.

Legal Information: Benchmark and Performance Disclaimers

- Performance results are based on testing as of Feb. 2019 and may not reflect all publicly available security updates. See configuration disclosure for details. No product can be absolutely secure.
- Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information, see Performance Benchmark Test Disclosure.
- Configurations: see performance benchmark test configurations.