Programming with Persistent Memory from Java

April 24, 2020

Persistent Memory for Java

Intel open-source libraries

- Low-Level Persistence Library (LLPL) -- version 1.0
- Java bindings to PMDK pmemkv library -- version 0.9
- Persistent Collections for Java (PCJ) -- experimental

OpenJDK enhancements

- JEP 352 Persistent MappedByteBuffer -- in JDK14
- JEP 370 Foreign-Memory Access API -- incubator in JDK14

Links to these are on last slide

Comparison of Libraries

	LLPL	Java bindings to pmemkv	PCJ	Mapped ByteBuffer	Memory Access API
Status	release 1.0	release 0.9	experimental	JDK14	incubator JDK14
Compatibility	JDK 8+	JDK 8+	JDK 8+	JDK14+	JDK14
Persistent data	heaps of memory blocks	key-value store	Java collections and other classes	ByteBuffers	structs, unions, arrays, etc.
Memory mgmt.	manual	manual	automatic	manual	manual
Thread-safe	heap: yes blocks: no	yes - optional	yes	no	no
Data integrity / consistency	developer-defined	developer-defined	ACID objects	developer-defined	developer-defined
Transactions	yes	yes - on puts	yes	no	no

LLPL Overview

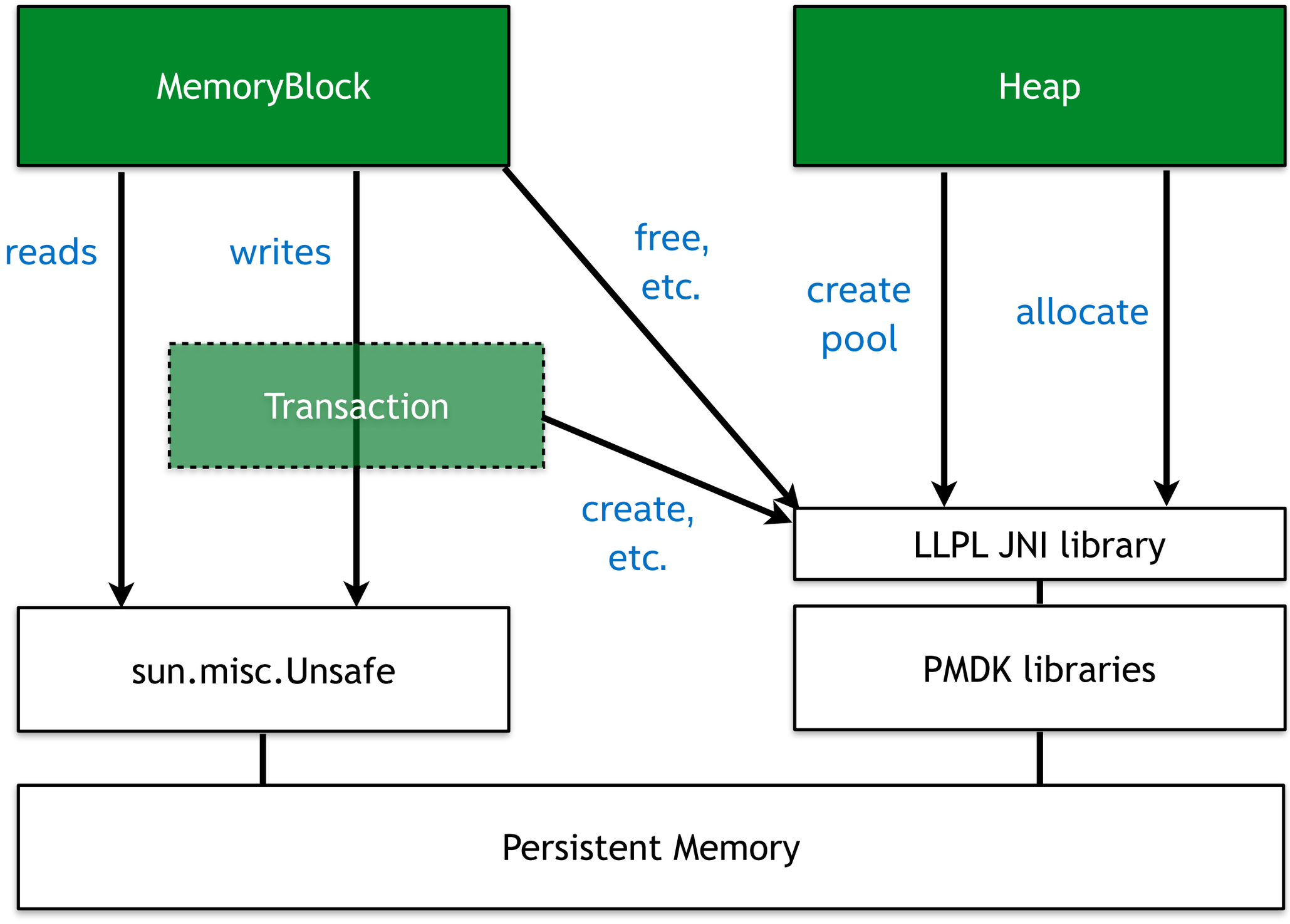
- Intel open-source Java library for persistent memory programming
- Compatible with JDK 8+
- Version 1.0 released at end of 2019
- A component of the Persistent Memory Development Kit
- Depends on PMDK libraries (libpmem, libpmemobj)
- Supports building with Maven or Make
- MavenCentral availability in-progress

- Goals:
  - Flexible, high-performance access to pmem from Java
  - Easy to use, idiomatic Java API (despite low-level nature)
  - Suitable for direct use or as base for higher-level abstractions

LLPL -- Three Primary Elements

- **Heap** -- a pool memory and an allocator for it
  - can create multiple heaps of almost any size
  - re-open a heap to access after restart
  - manual memory management
  - thread-safe API
- **MemoryBlock** -- an accessor for a block of allocated memory
  - low-level setters and getters indexed by offsets within a block
  - can refer to and recover a block long-term using stable (long) handles
  - write handles in another block to link blocks and data structures
  - non-thread-safe API
- **Transaction** -- can group writes for fail-safe data consistency
  - transaction state is thread-local
  - API integrates with try-catch and uses lambdas as bodies
  - nested transactions "flattened" to commit or abort together

LLPL Implementation



MemoryBlock API -- Reading and Writing

Write methods:

1. setByte
2. setShort
3. setInt
4. setLong
5. setMemory
6. copyFromArray
7. copyFromMemoryBlock

Read methods:

1. getByte
2. getShort
3. getInt
4. getLong
5. copyToArray

Other methods:

- flush
- free
- addToTransaction

Data Integrity and Consistency

- Being able to say something clear about the usability of heap data after an event such as:
  - a controlled exit
  - an unhandled exception
  - a power failure
- Durable changes to pmem leave heap data in a known-usable state if writes aren't interrupted, e.g. normal execution, controlled exit
- Transactional changes to pmem leave heap data in a known-usable state after any of the above events
- LLPL offers tools to create your own simple or not-so-simple policies

Three Writes, Two Errors, Three Heaps

Three kinds of writes:

1. volatile: write data
2. durable: write data, **flush data from CPU cache to media**
3. transactional: **add data range to transaction** (back up data), write data

Two new kinds of programming errors:

1. durable write: forget to flush data from cache
2. transactional write: forget to add range to transaction before writing

Three kinds of heaps:

1. Heap: most flexible but both errors are possible
2. PersistentHeap: all durable, opt. transactional -- if code compiles error #1 is not present
3. TransactionalHeap: all transactional -- if code compiles neither error is present

Heap Fragmentation

- Allocated handles are stable for the lifetime of the allocation
- Some allocate / free patterns can fragment the heap
- Same is true for off-heap DRAM; exacerbated here by long lived heaps
- Mitigations and solutions
  - reuse: choose sizes that enable reuse, e.g. when updating existing data
  - sub-allocation: application-managed sub-allocation of large blocks
  - compaction API (exploring now): developer requests that the heap compact a selection of handles returning new handles
  - storage service (like pmemkv): data is copied in and copied out of a data structure; service owns the memory and is free to move it to defragment

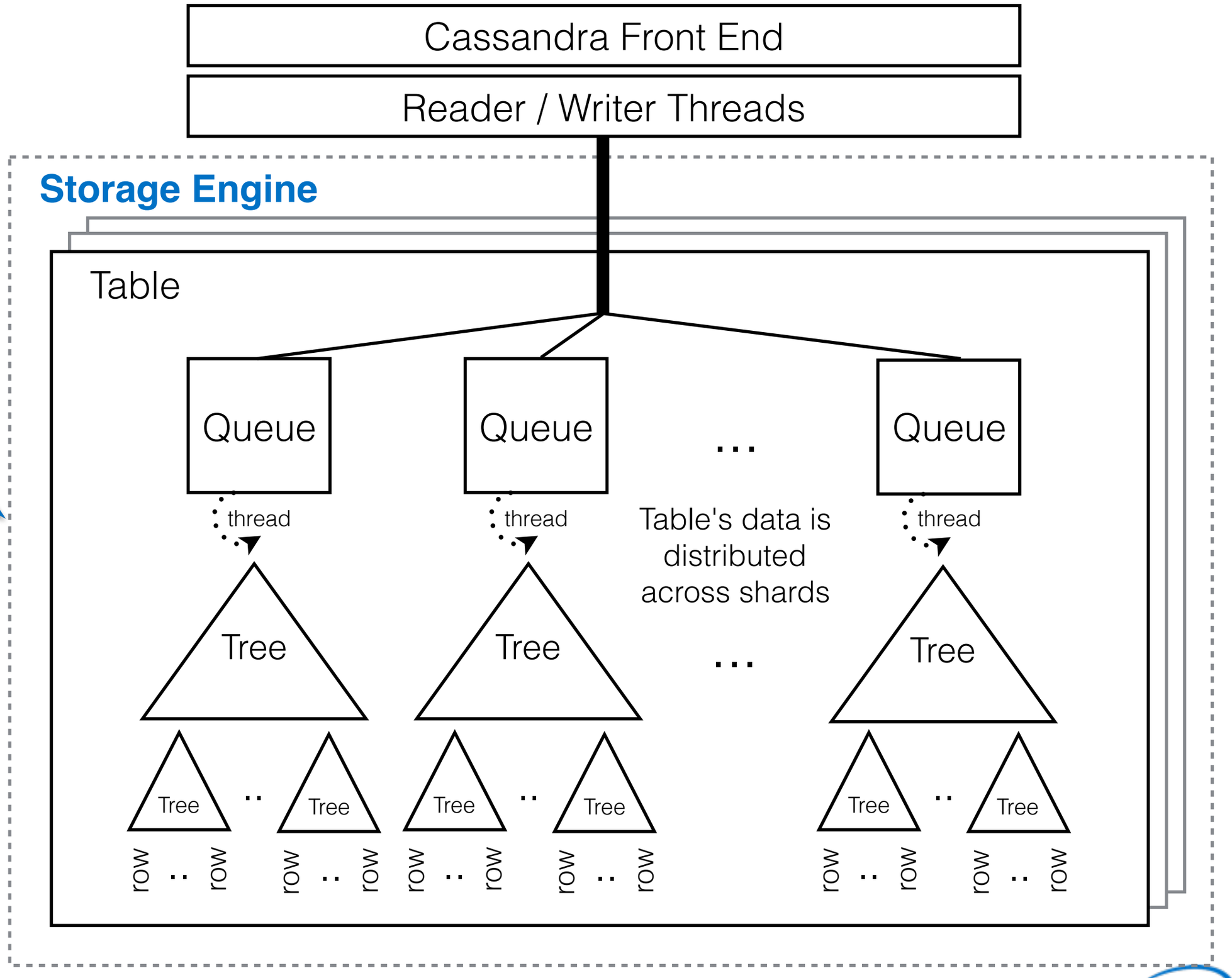
LLPL Performance and Usability Optimizations

- All reads and writes done using Unsafe
- Minimized number of JNI calls
- Easy to use "ranged write" methods automate and minimize flush and transaction operations:
  - on initialization of new memory block
  - developer-defined range of offsets
- Automatically benefit from core PMDK optimizations, e.g.
  - allocator
  - transactions

Example Use: Cassandra Pmem Storage Engine

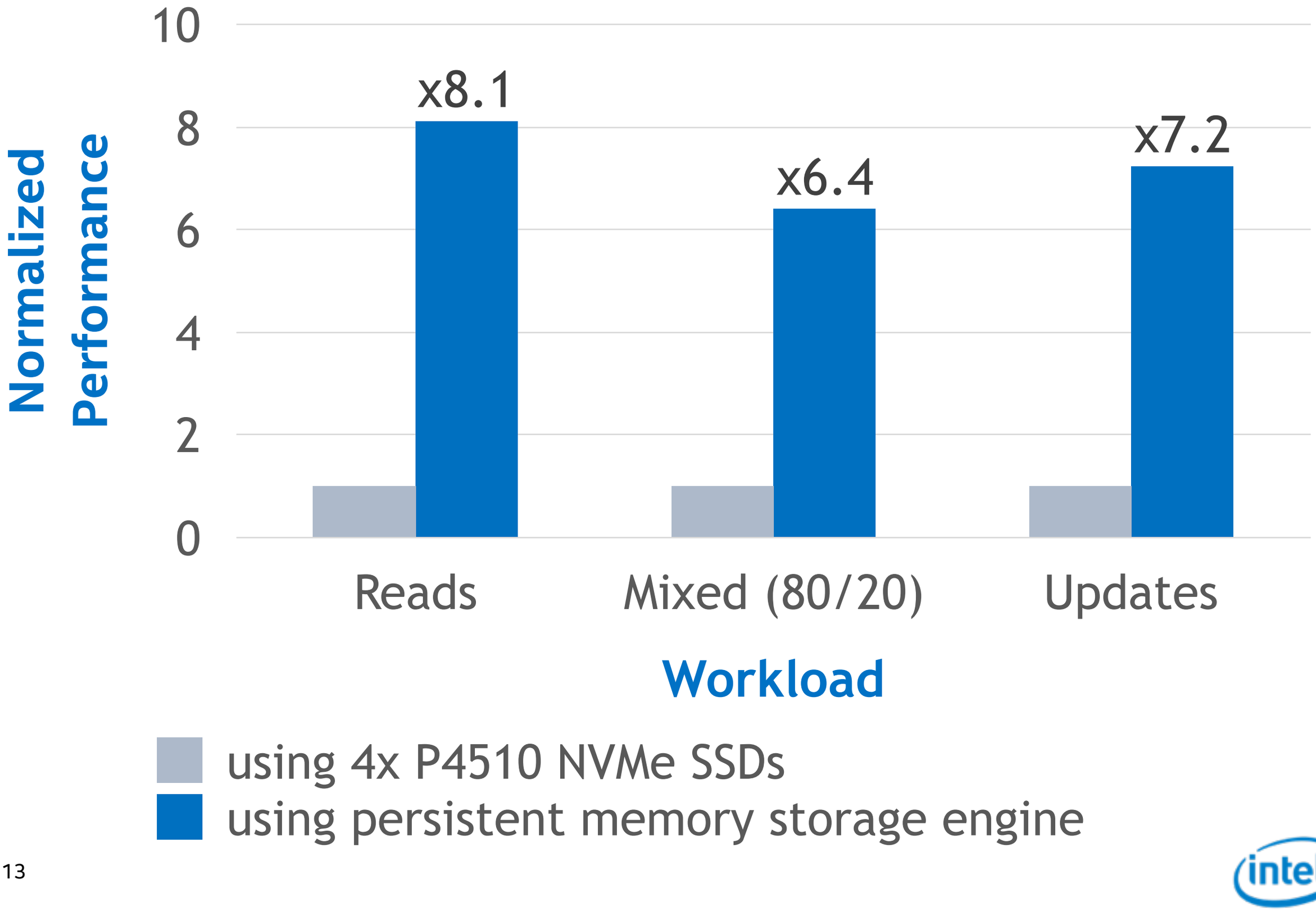
https://github.com/intel/cassandra-pmem

All persistent data structures implemented using LLPL



△ = Adaptive Radix Tree: https://db.in.tum.de/%7Eleis/papers/

Cassandra Throughput



LLPL Workshop and Repository Examples

Workshop

1. getting started
2. sizing heaps
3. using other heaps
4. more on transactions
5. performance idioms
6. wrapping memory blocks

Repository

1. int array
2. handle array
3. linked list
4. adaptive radix tree

Going Forward

- Future of these open-source libraries is steered by customer feedback and requirements
- Examples of LLPL features / optimizations being considered:
  - construction-free allocation
  - re-positionable accessor
  - small library of data structures
  - handle tracking for liveness, etc.
- Please don't hesitate to contact us!

Links

- Intel® Optane™ DC persistent memory
  - https://www.intel.com/content/www/us/en/products/memory-storage/optane-dc-persistent-memory.html
- Low Level Persistence Library (LLPL)
  - https://github.com/pmem/llpl
- Java bindings to PMDK pmemkv library -- version 0.9
  - https://github.com/pmem/pmemkv-java
- Persistent Collections for Java (PCJ) [Experimental]
  - https://github.com/pmem/pcj
- Persistent Memory Development Kit (PMDK)
  - https://github.com/pmem/pmdk
- JEP 370 - java.foreign Memory Access API
  - https://openjdk.java.net/jeps/370
- JEP 352 -- Non-Volatile Mapped Byte Buffers
  - https://openjdk.java.net/jeps/352
- JEP 316 -- Allocation of Java Heap on Alt. Memory Devices
  - https://openjdk.java.net/jeps/316
- Cassandra persistent memory storage engine
  - https://github.com/intel/cassandra-pmem