HBASECON ASIA2019

THE COMMUNITY EVENT FOR APACHE HBASE™

Further GC optimization for HBase 2.x: Reading HFileBlock into offheap directly

Anoop Sam John / Zheng Hu

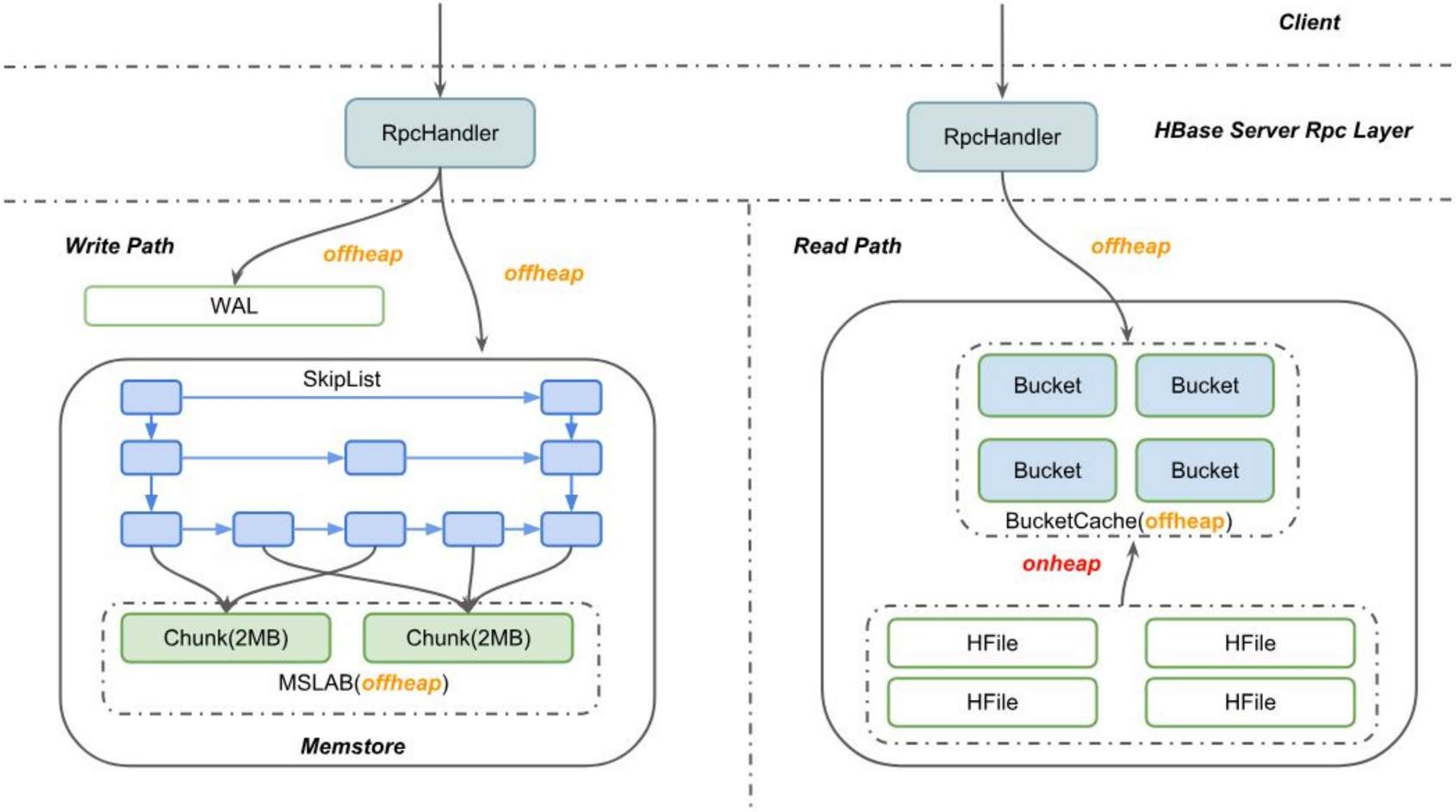
Abstract

- ❏ Background: Why offheap HDFS block reading
- ❏ Idea & Implementation
- ❏ Performance Evaluation
- ❏ Best Practice

Abstract

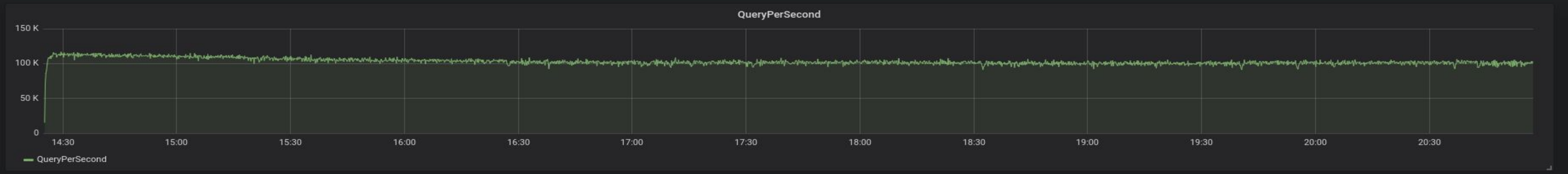
- **Background: Why offheap HDFS block reading**
- Idea & Implementation
- Performance Evaluation
- Best Practice

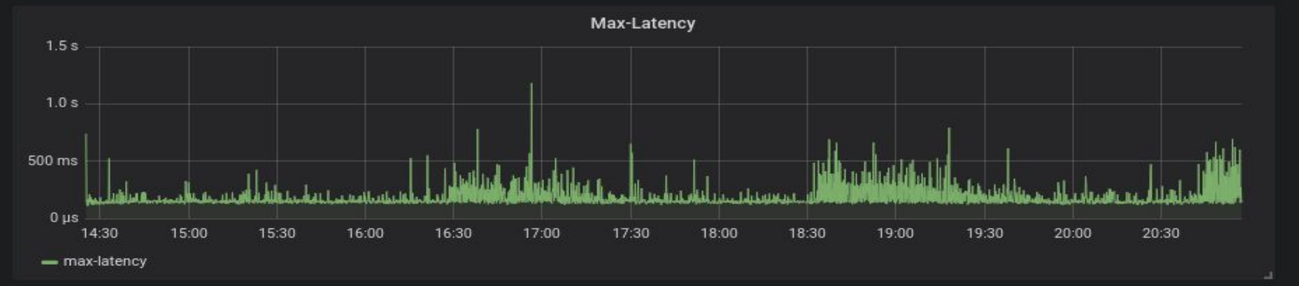
Background: current offheap read/write path

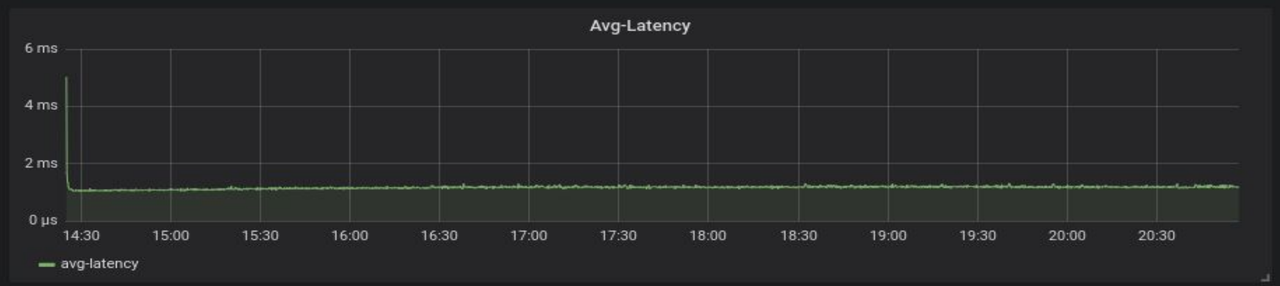


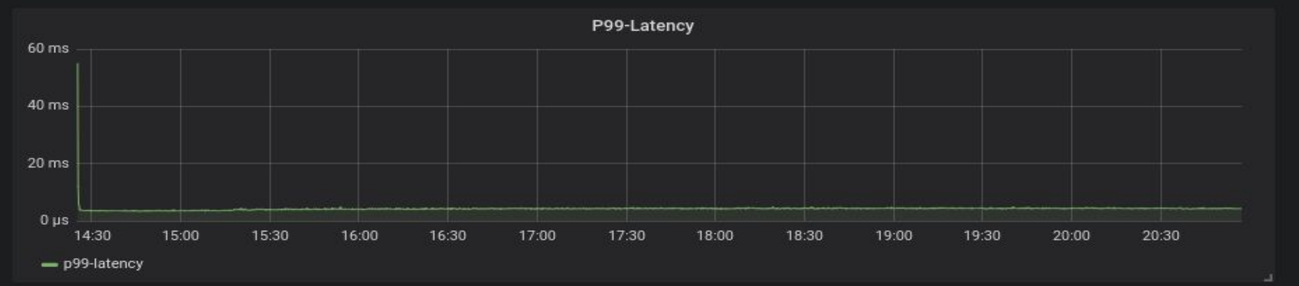
hbase20-ssd-get-10000000000-rows.oraclejdk.case1

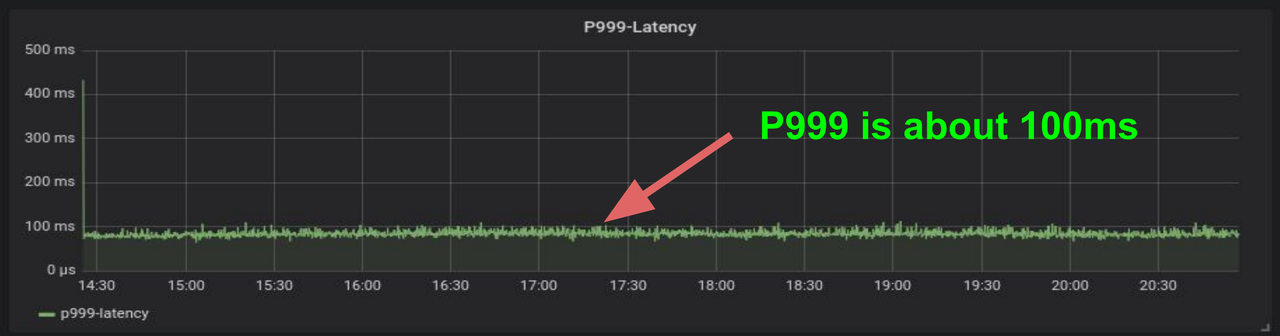
2月 12, 2019 14:24:49 to 2月 12, 2019 20:57:18

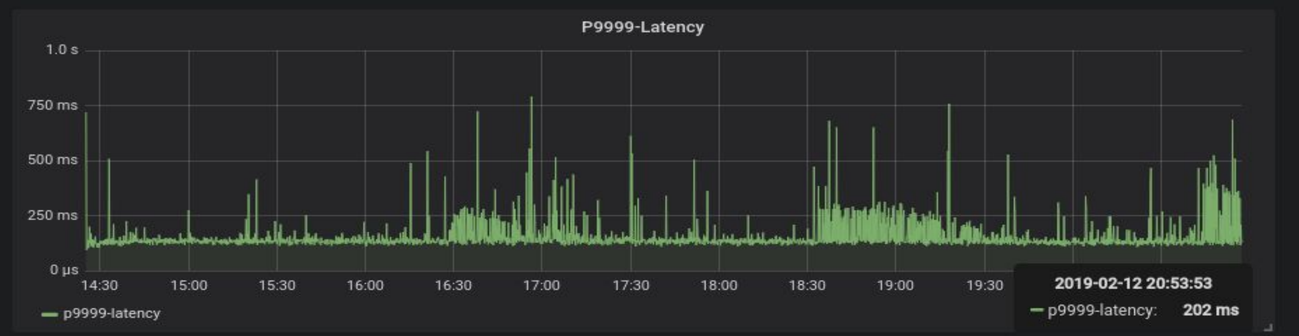




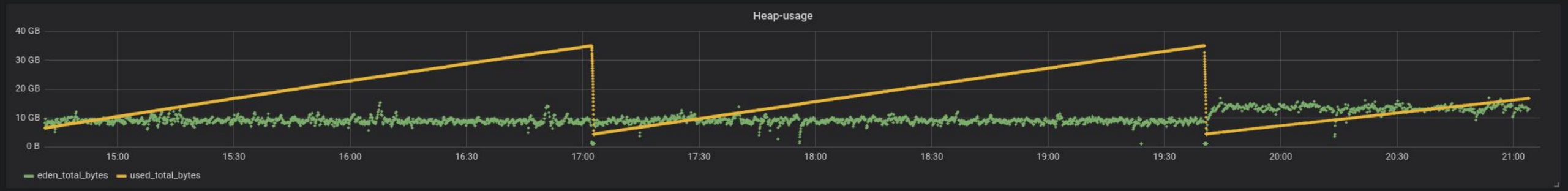


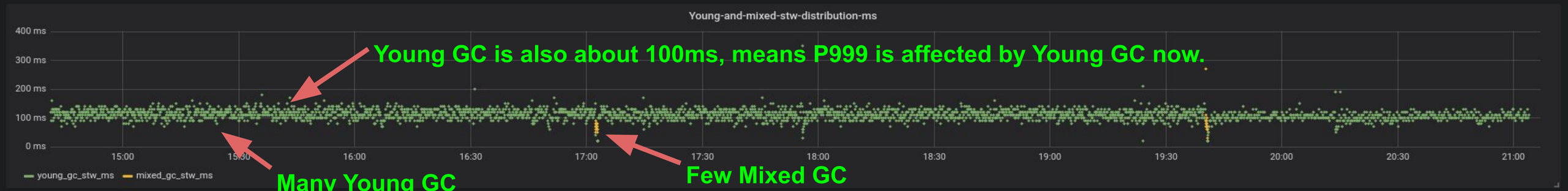


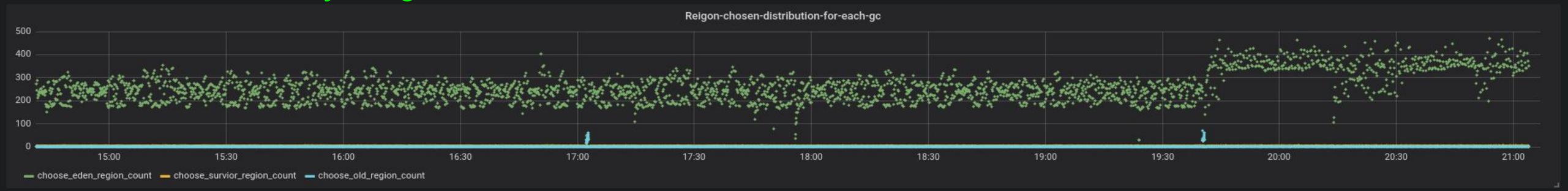




case_name hbase20-ssd-get-10000000000-rows.oraclejdk.case1-gc-data







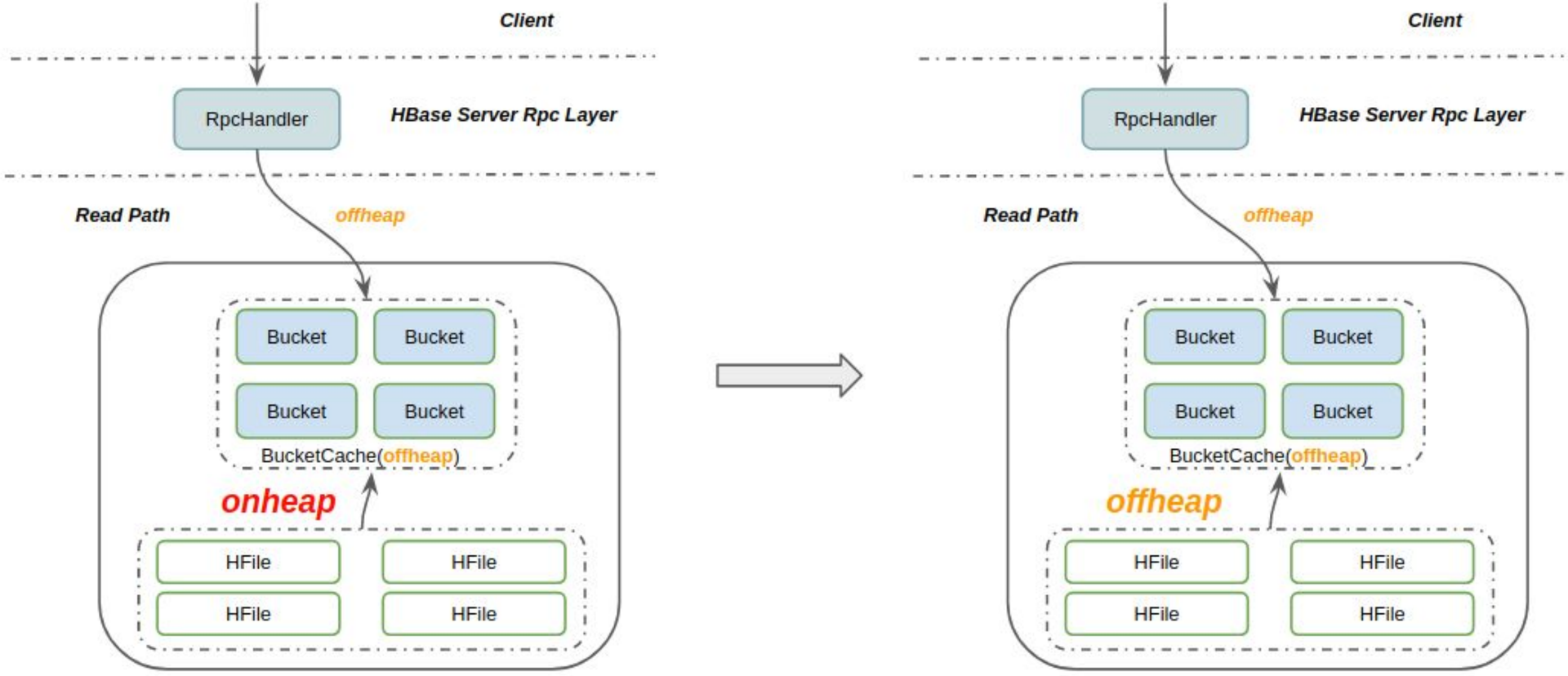
Background: still GC issue in some case ?

- High young GC pressure if cachHitRatio is not 100%
  - Reading the Block from HFile is still copied to the heap firstly
  - The heap block won’t be garbage collected unless:
    - Read is complete and the results been created (CellBlock) in the RPC responder area.
    - the WriterThread of BucketCache flushes the Block to offheap IOEngine successfully
  - A large number of young generation objects, which leads to the raising young GC pressure.

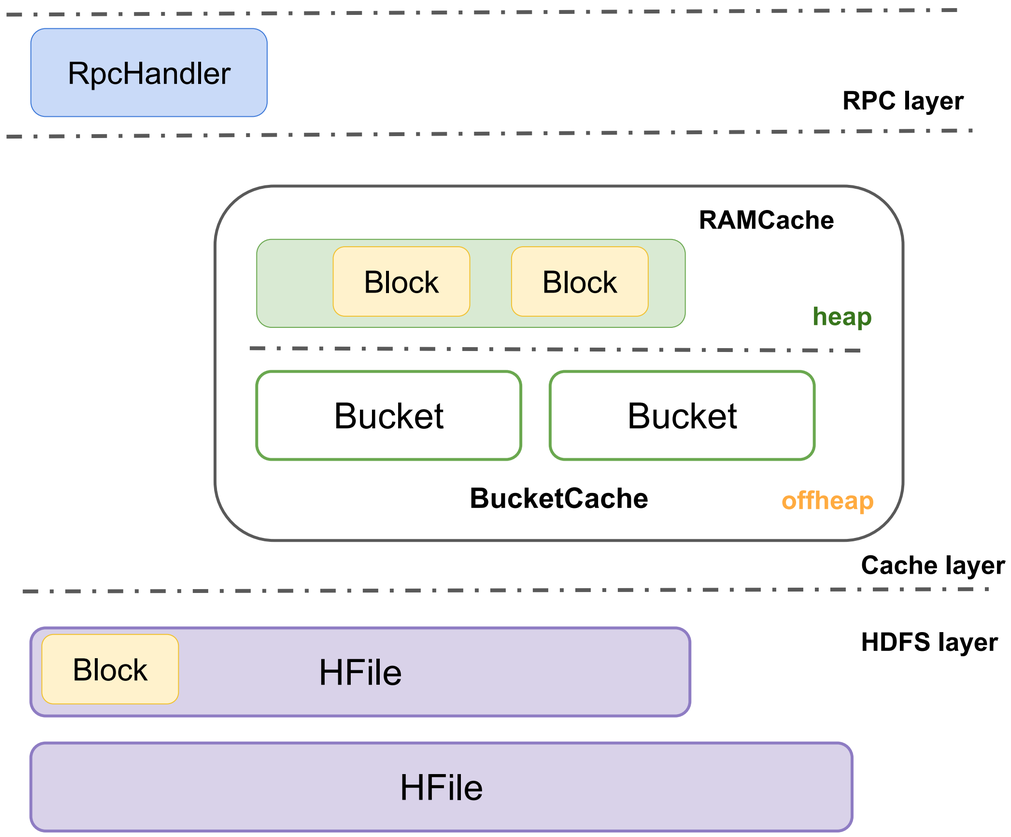
Abstract

- ❏ Background: Why offheap HDFS block reading
- ❏ **Idea & Implementation**
- ❏ Performance Evaluation
- ❏ Best Practice

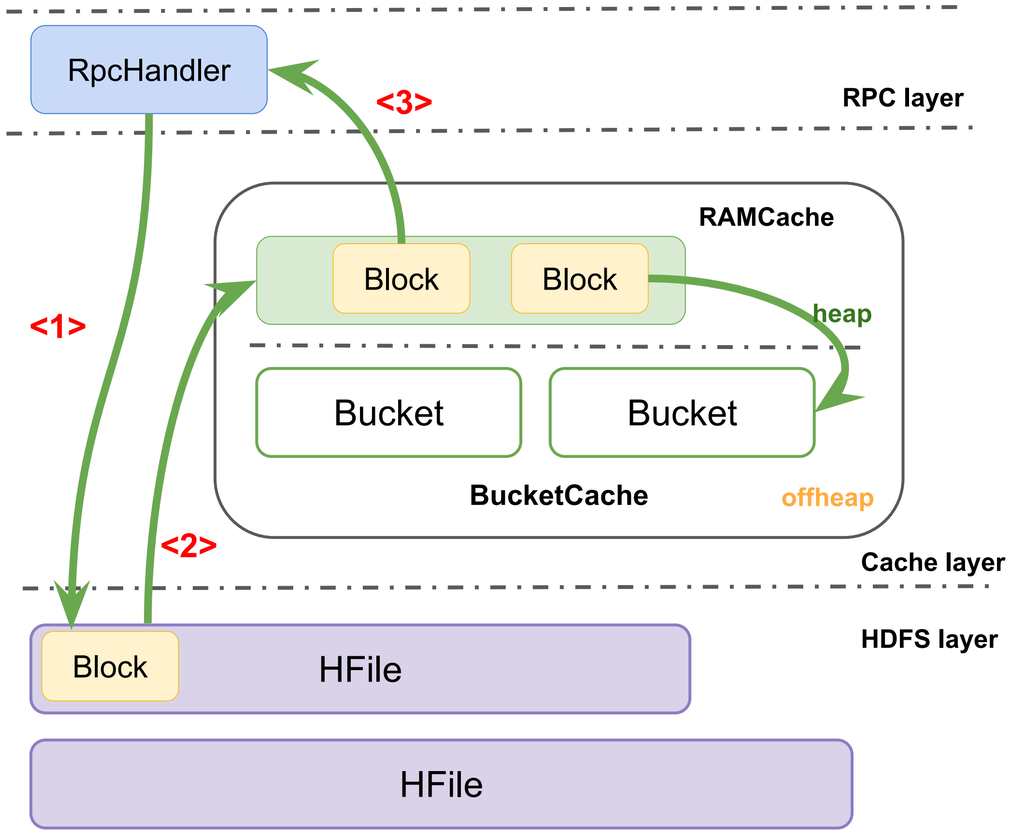
Basic Idea: Just read the block from HFile to **pooled ByteBuffers**



Review: how did we cache a block in BucketCache ?



Review: how did we cache a block in BucketCache ?



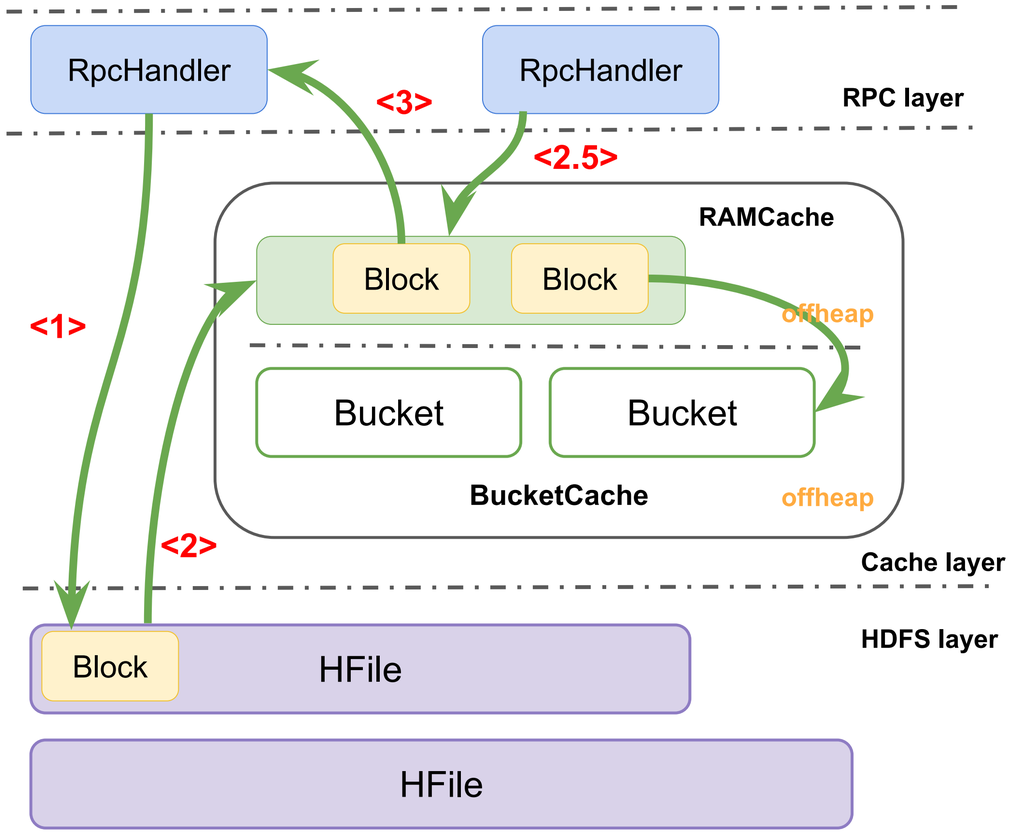
<1> Read block from HFile to pooled ByteBuffers;

<2> Cache the block in a temporary map named **RAMCache** for avoiding the unstable latency if flushing to offheap bucket cache synchronously.

The WriterThreads of BucketCache flush the block to offheap array asynchronously once #2 finished.

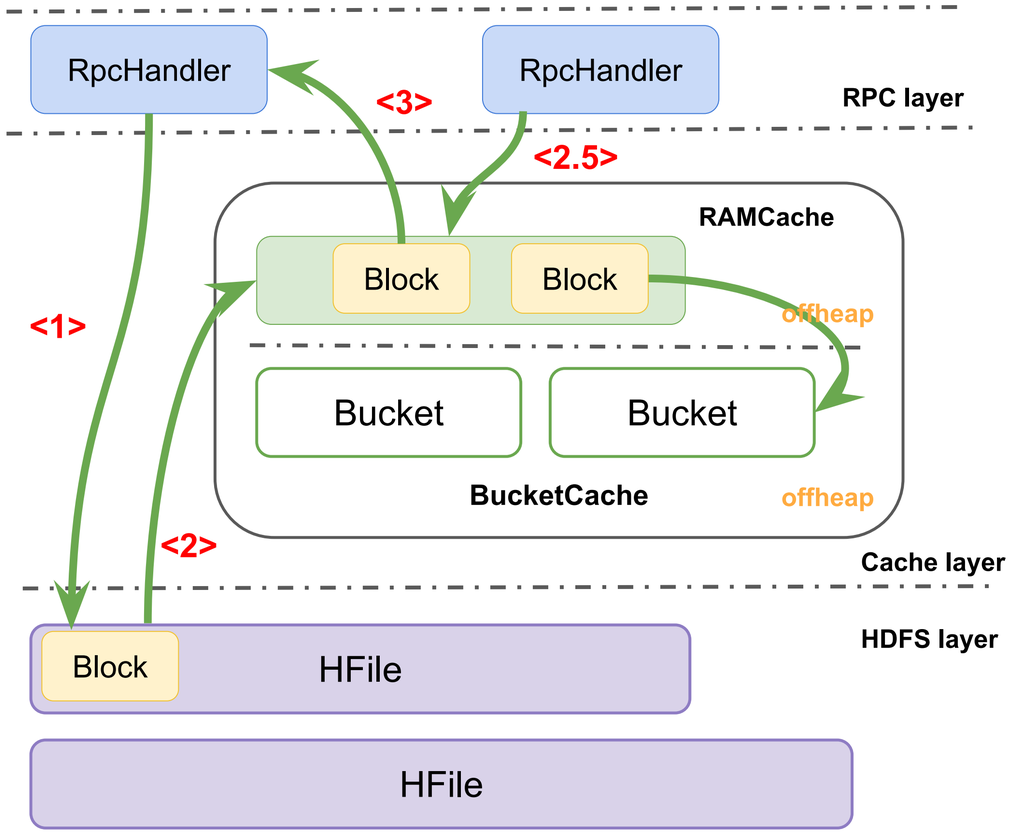
<3> Encoded the cells from block and shipped to RPC client.

Problem: Block may be accessed by other RPC Handlers ?



Once #2 load the block into RAMCache, then other RpcHandler may hit the block and reference to it.

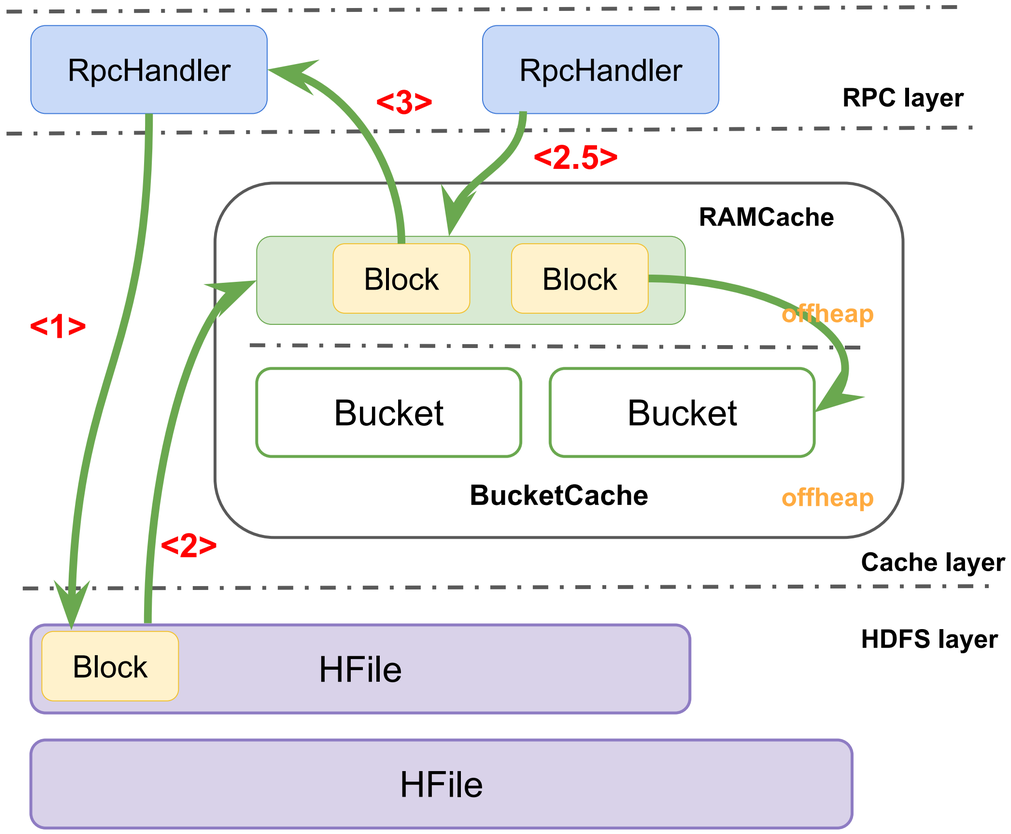
Problem: Block may be accessed by other RPC Handlers ?



Once #2 load the block into RAMCache, then other RpcHandler may hit the block and reference to it.

Then how should we release the offheap block back to the pool without causing any memory leak issues?

Problem: Block may be accessed by other RPC Handlers ?



Once #2 load the block into RAMCache, then other RpcHandler may hit the block and reference to it.

Then how should we release the offheap block back to the pool without causing any memory leak issues?

Reference count:
1. Consider the RAMCache as a separate reference path;
2. RPC handlers are another separate reference paths.

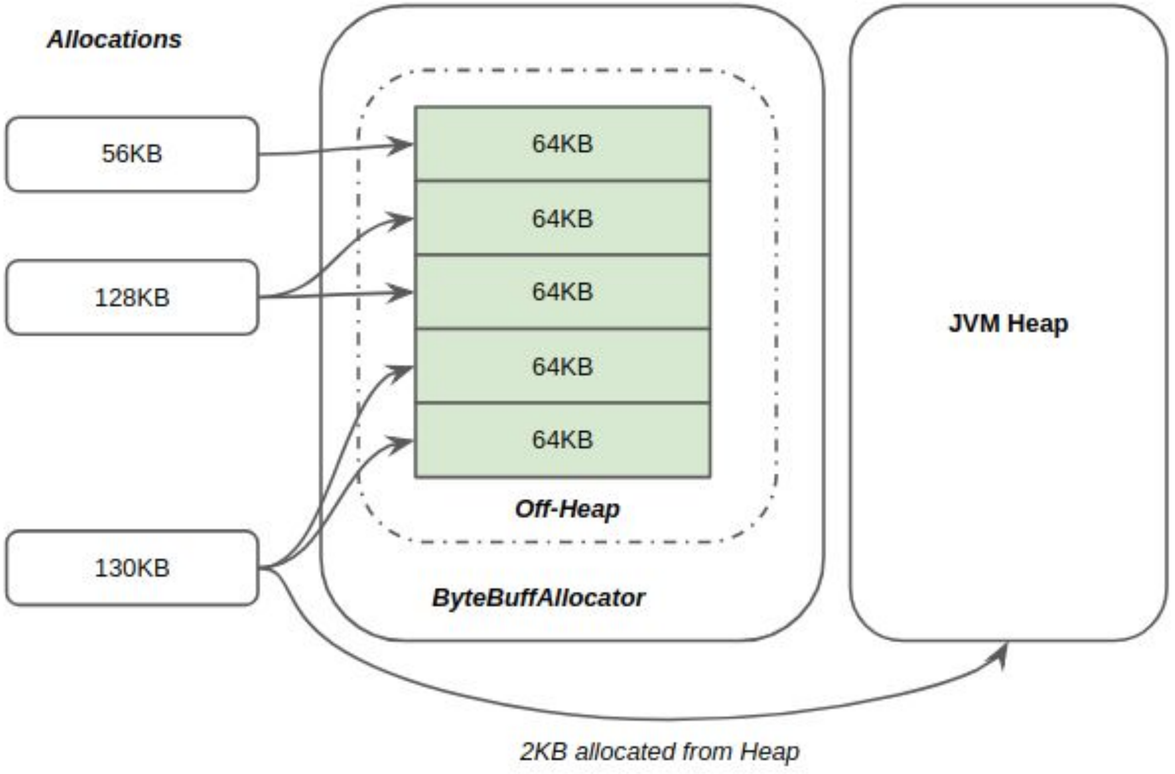
Core Idea

1. Maintain a refCount in block's ByteBuff, once allocated, its refCount will be initialized to 1;
2. If put it into **RAMCache**, then refCount ++;
3. If removed from **RAMCache**, then refCount --;
4. If some RPC hit the ByteBuff in RAMCache, then refCount ++;
5. Once RPC finished, then ByteBuff's refCount --;
6. If its refCount decrease to zero, we MUST deallocate the ByteBuff which means putting its NIO ByteBuffers back to ByteBuffAllocator. Besides, nobody can access the ByteBuff with refCount = 0.

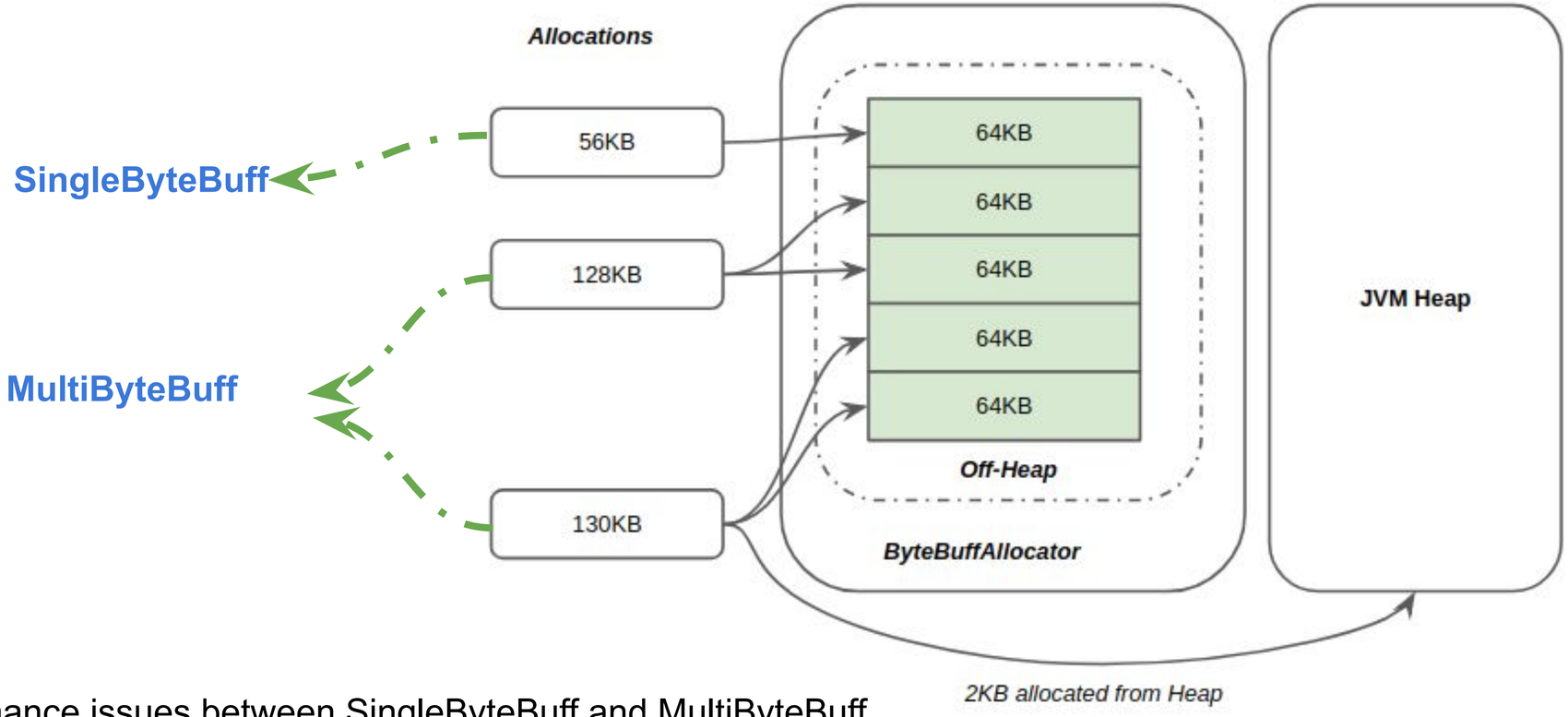
Implementation

- General ByteBuffer Allocator
- Reference Count
- DownStream API Support
- Other issues:
  - Unify the refCnt of BucketEntry and HFileBlock into one
  - Combined the BucketEntry sub-classes into one

General Allocator



General Allocator



Performance issues between SingleByteBuff and MultiByteBuff

Reference Count

- Use Netty's AbstractReferenceCounted
  - Use unsafe/safe method to maintain the reference count value.
  - Once the reference count value decreasing to zero, will trigger **the registered Recycler** to deallocate.
  - The duplicate or slice ByteBuff will share the same RefCnt with the original one.
    - if want to retain the buffer even if original one did a release, can do as the following:

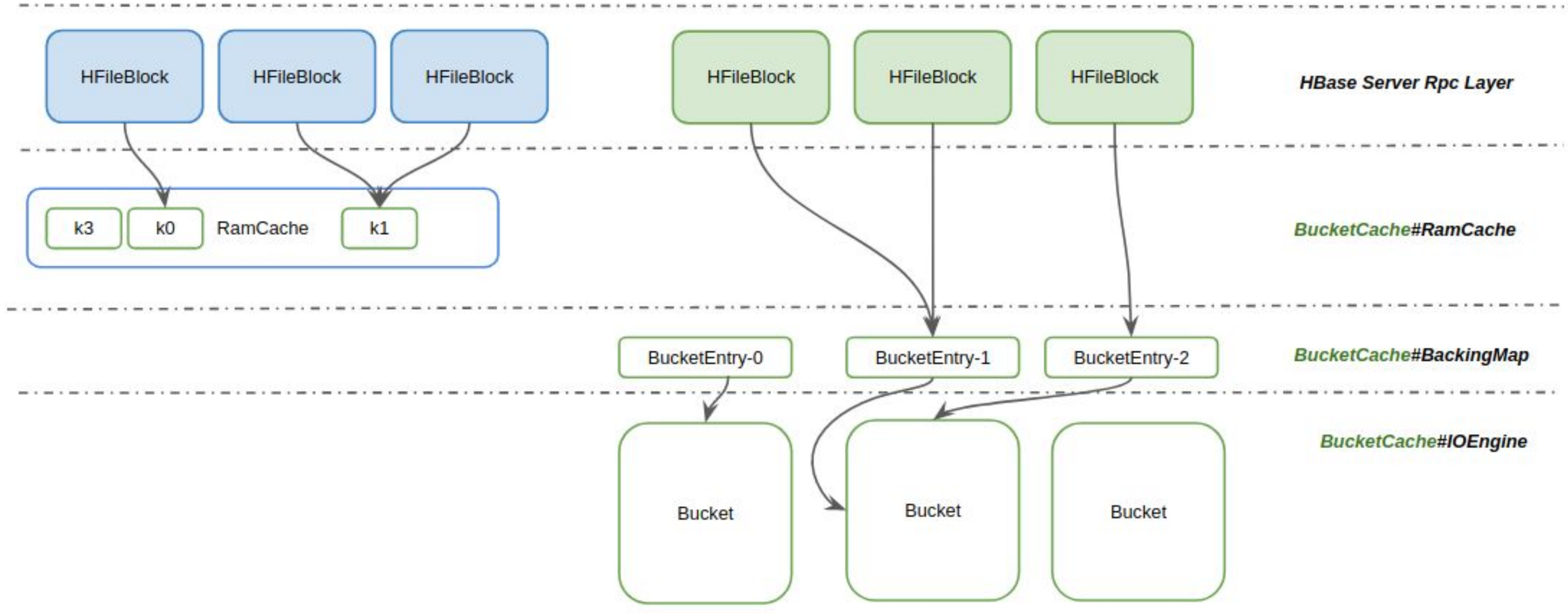
```
ByteBuff original = ...;
ByteBuff dup = original.duplicate();
dup.retain();
original.release();
// The NIO buffers can still be accessed unless release the duplicated one
dup.get(...);
dup.release();
// Both the original and dup can not access the NIO buffers any more.
```

Retain the duplicated one before release the original one

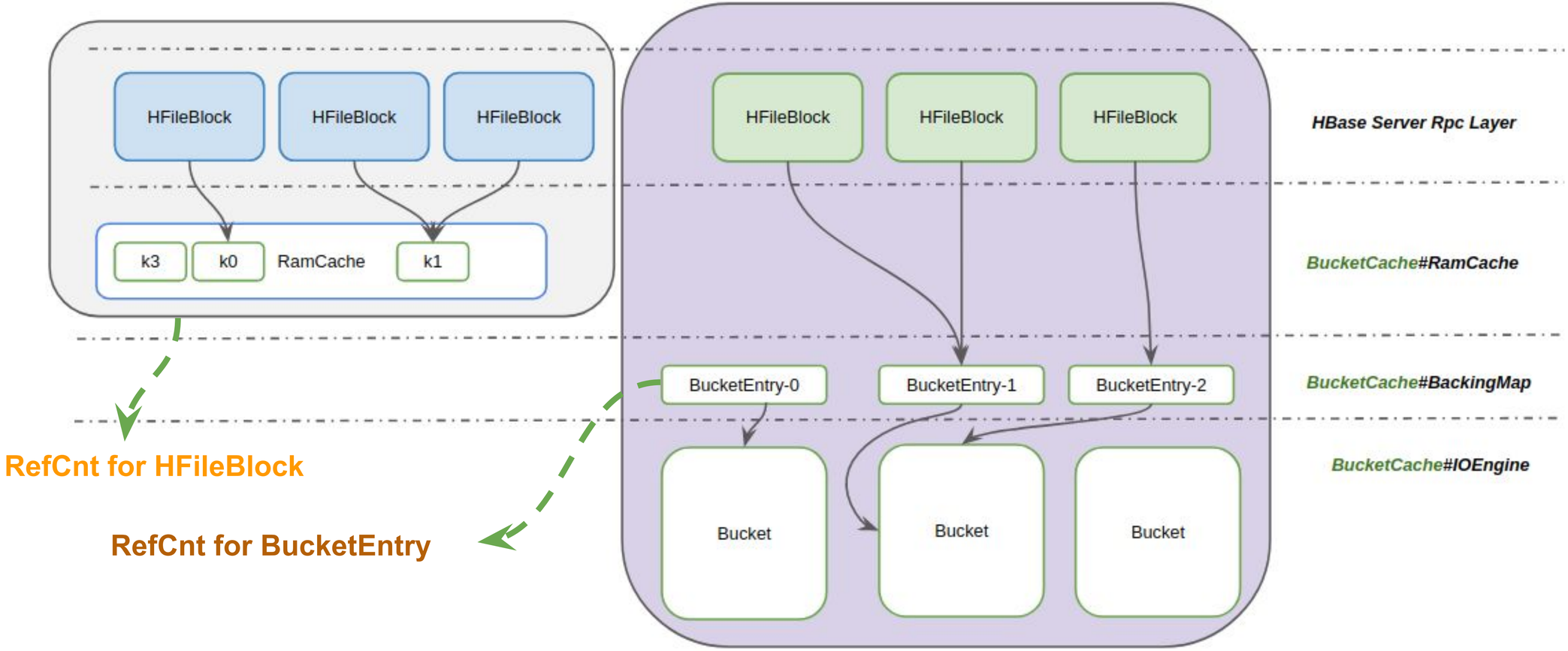
Downstream API Support

- ByteBuffer positional read interface (HBASE-21946)
  - Need support from Apache Hadoop (**Hadoop >=2.9.3**)
- Checksum validation methods (HBASE-21917)
  - SingleByteBuff will use the hadoop native lib
  - MultiByteBuff will copy to heap and validate the checksum.
- Block decompression methods (HBASE-21937)
  - Copy to an temporary small heap buffer and decompression

Unify the refCnt of BucketEntry and HFileBlock into one



Unify the refCnt of BucketEntry and HFileBlock into one



Just pass the BucketEntry's refCnt to HFileBlock. Then BucketEntry and HFileBlock will share an single RefCnt.

Combined the BucketEntry sub-classes into one

- BucketEntry
  - Exclusive heap block
  - No reference count.
- SharedMemoryBucketEntry
  - Shared offheap block
  - Use a AtomicInteger to maintain the reference count.
- UnsafeSharedMemoryBucketEntry
  - Shared offheap block
  - Use integer and unsafe CAS to maintain the reference count.

Before HBASE-21879

BucketEntry

With netty's RefCnt inside, which will use safe/unsafe way to update the refCnt.

After HBASE-21879

Abstract

- ❏ Background: Why offheap HDFS block reading
- ❏ Idea & Implementation
- ❏ **Performance Evaluation**
- ❏ Best Practice

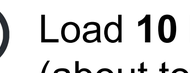
Test Cases

Three test cases to prove the performance improvement after HBASE-21879

- Disabled BlockCache cache: CacheHitRatio ~ 0%
- CacheHitRatio~65%
- CacheHitRatio~100%

Environment & Workload

service	job	host	cpu	disk	netowork	comment
YCSB	hbase-client	c3-hadoop-tst-st37.bj	-	-	-	-
HBase	master	c3-hadoop-tst-zk02.bj	-	-	-	-
HBase	master	c3-hadoop-tst-zk03.bj	-	-	-	-
HBase	region server	c3-hadoop-tst-st47.bj	24 core	12*900G SSD	10Gbps	onheap=50g/offheap=50g
HBase	region server	c3-hadoop-tst-st48.bj	24 core	12*900G SSD	10Gbps	onheap=50g/offheap=50g
HBase	region server	c3-hadoop-tst-st49.bj	24 core	12*900G SSD	10Gbps	onheap=50g/offheap=50g
HBase	region server	c3-hadoop-tst-st50.bj	24 core	12*900G SSD	10Gbps	onheap=50g/offheap=50g
HBase	region server	c3-hadoop-tst-st51.bj	24 core	12*900G SSD	10Gbps	onheap=50g/offheap=50g
HDFS	namenode	c3-hadoop-tst-zk02.bj	-	-	-	onheap=10g
HDFS	namenode	c3-hadoop-tst-zk03.bj	-	-	-	onheap=10g
HDFS	journal node	c3-hadoop-tst-zk01.bj	-	-	-	-
HDFS	journal node	c3-hadoop-tst-zk02.bj	-	-	-	-
HDFS	journal node	c3-hadoop-tst-zk03.bj	-	-	-	-
HDFS	zkfc	same as namenode0	-	-	-	-
HDFS	zkfc	same as namenode1	-	-	-	-
HDFS	datanode	c3-hadoop-tst-st47.bj	24 core	12*900G SSD	10Gbps	onheap=2g
HDFS	datanode	c3-hadoop-tst-st48.bj	24 core	12*900G SSD	10Gbps	onheap=2g
HDFS	datanode	c3-hadoop-tst-st49.bj	24 core	12*900G SSD	10Gbps	onheap=2g
HDFS	datanode	c3-hadoop-tst-st50.bj	24 core	12*900G SSD	10Gbps	onheap=2g
HDFS	datanode	c3-hadoop-tst-st51.bj	24 core	12*900G SSD	10Gbps	onheap=2g



Load **10 billion** rows , each row with **size=100 byte**.
(about total **700GB** in the clusters)
Major compaction to ensure the locality is 1.0

Case#1: Disabled BlockCache, CacheHitRatio~0%

	Before HBASE-21879	After HBASE-21879	Delta %
GC Overview			
Young GC Count in 3 hour	768	664	-13.6%
STW per GC (ms)	160 ms	150 ms	-6.25%
Eden Usage (GB)	25.2 GB	4.6 GB	-81.7%
GC Details			
Update RememberSet (ms)	0.9987 ms	0.9940 ms	-0.4%
Region Chosen Each GC	809	150	-81.5%
G1 Object Copy (ms)	116.98 ms	108.98 ms	-6.8%
QPS & Latency			
Get QPS	22844 op/s	26779 op/s	+17.2%
Avg Latency (ms)	5.253 ms	4.481 ms	-14.7%
P99 Latency (ms)	56 ms	50 ms	-10.7%
P999 Latency (ms)	172.41 ms	169.98 ms	-1.4%

Case#1: Disabled BlockCache, CacheHitRatio~0%

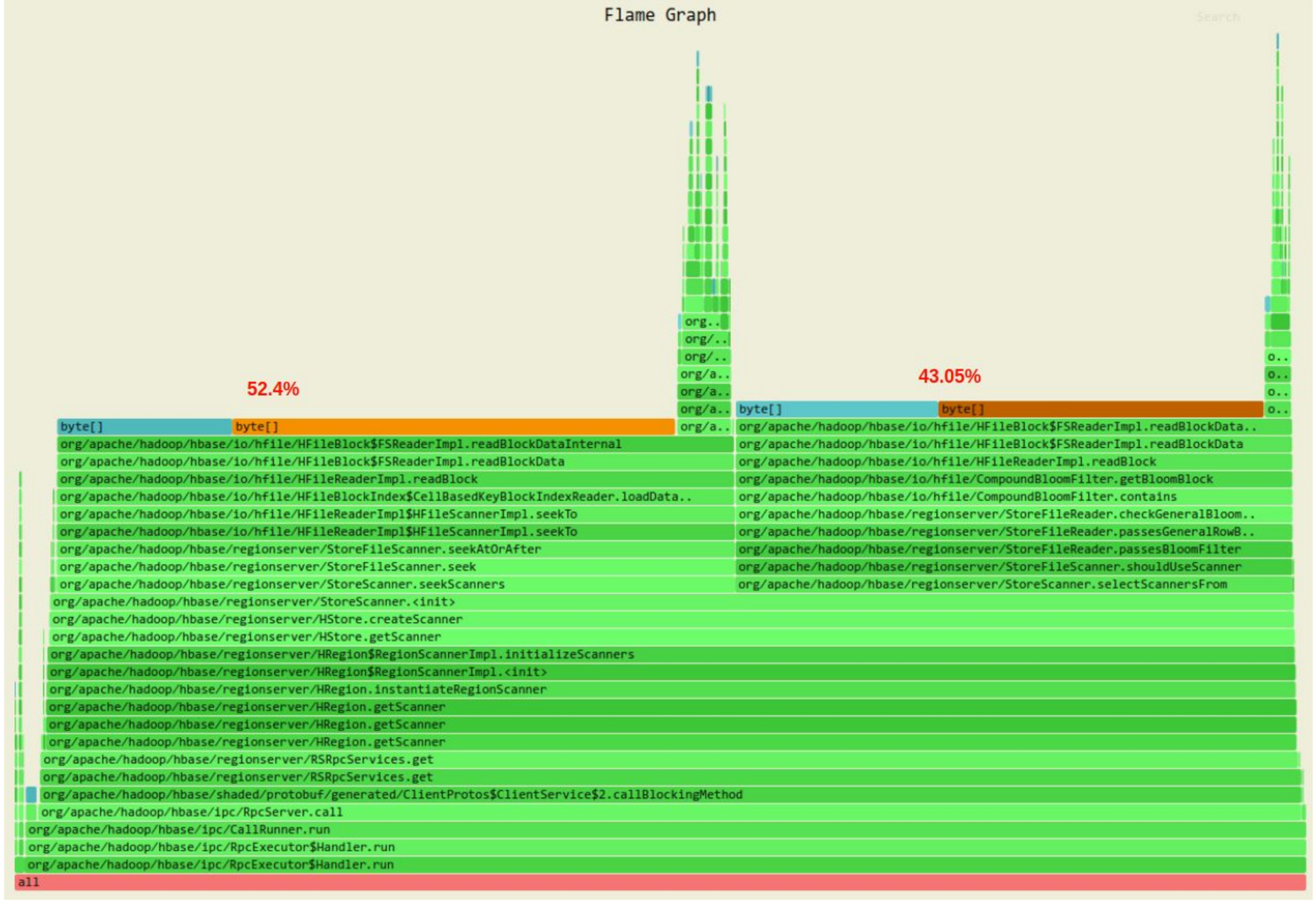
	Before HBASE-21879	After HBASE-21879	Delta %
GC Overview			
Young GC Count in 3 hour	768	664	-13.6%
STW per GC (ms)	160 ms	150 ms	-6.25%
Eden Usage (GB)	25.2 GB	4.6 GB	-81.7%
GC Details			
Update RememberSet (ms)	0.9987 ms	0.9940 ms	-0.4%
Region Chosen Each GC	809	150	-81.5%
G1 Object Copy (ms)	116.98 ms	108.98 ms	-6.8%
QPS & Latency			
Get QPS	22844 op/s	26779 op/s	+17.2%
Avg Latency (ms)	5.253 ms	4.481 ms	-14.7%
P99 Latency (ms)	56 ms	50 ms	-10.7%
P999 Latency (ms)	172.41 ms	169.98 ms	-1.4%

Decreased almost 81.7% young usage.

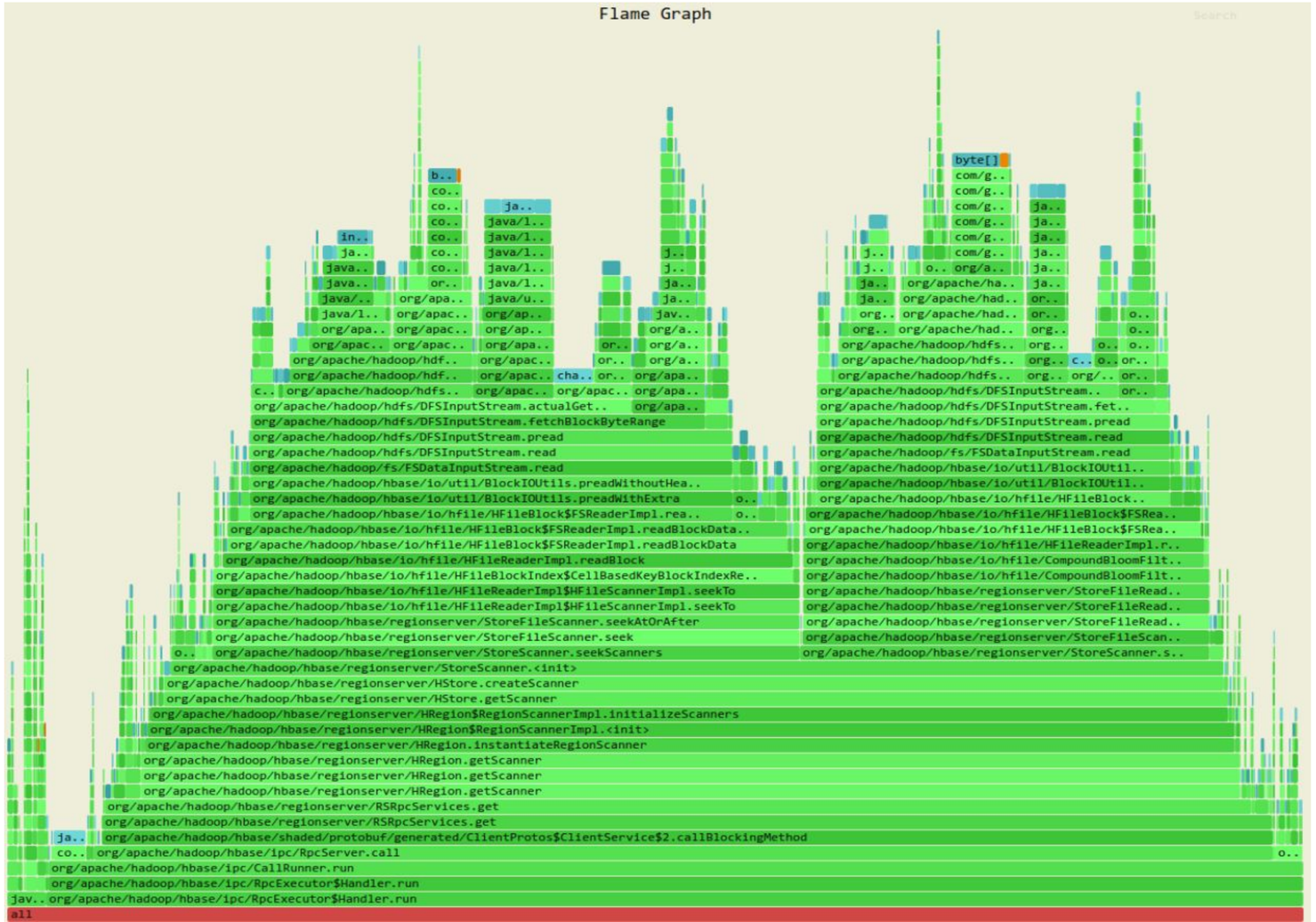
Heap occupation after GC decreased a bit.

Throughtput increased 17.2%, latency decreased about 14.7%

Case#1: Before HBASE-21879



Case#1: After HBASE-21879



Decreased almost 95.5% heap allocation.

Case#2: CacheHitRatio ~ 65%

	Before HBASE-21879	After HBASE-21879	Delta %
GC Overview			
Young GC Count in 3 hour	2709	2167	-20%
STW per GC (ms)	53.58 ms	55.5 ms	+3.5%
Eden Usage (GB)	3.3	2.2	-33%
GC Details			
Update RememberSet (ms)	7.48 ms	7.03 ms	-6%
Region Chosen Each GC	91	58	-36%
G1 Object Copy (ms)	21 ms	20 ms	-4%
QPS & Latency			
Get QPS	101530 op/s	97224 op/s	-3%
Avg Latency (ms)	1.181 ms	1.223 ms	+3.4%
P99 Latency (ms)	5.468 ms	5.579 ms	+2%
P999 Latency (ms)	113.29 ms	115.54 ms	+1%

Case#2: CacheHitRatio ~ 65%

	Before HBASE-21879	After HBASE-21879	Delta %
GC Overview			
Young GC Count in 3 hour	2709	2167	-20%
STW per GC (ms)	53.58 ms	55.5 ms	+3.5%
Eden Usage (GB)	3.3	2.2	-33%
GC Details			
Update RememberSet (ms)	7.48 ms	7.03 ms	-6%
Region Chosen Each GC	91	58	-36%
G1 Object Copy (ms)	21 ms	20 ms	-4%
QPS & Latency			
Get QPS	101530 op/s	97224 op/s	-3%
Avg Latency (ms)	1.181 ms	1.223 ms	+3.4%
P99 Latency (ms)	5.468 ms	5.579 ms	+2%
P999 Latency (ms)	113.29 ms	115.54 ms	+1%

Young GC count decreased about 20%

Object copying when GC decreased about 4%

Latency increased a bit because of the offheap bytes reading & cell decoding.

Case#3: CacheHitRatio ~100%

	Before HBASE-21879	After HBASE-21879	Delta %
GC Overview			
Young GC Count in 3 hour	296	306	+3.8%
STW per GC (ms)	29 ms	25 ms	-13.8%
Eden Usage (GB)	30GB	30GB	0.0%
GC Details			
Update RememberSet (ms)	1.010 ms	1.000 ms	-0.9%
Region Chosen Each GC	960	960	0.0%
G1 Object Copy (ms)	0.294 ms	0.252 ms	-14.2%
QPS & Latency			
Get QPS	257580 op/s	262900 op/s	+2.1%
Avg Latency (ms)	0.4634 ms	0.4540 ms	-2.0%
P99 Latency (ms)	2.141 ms	1.331 ms	-37.8%
P999 Latency (ms)	12 ms	9 ms	-25.0%

Case#3: CacheHitRatio ~100%

	Before HBASE-21879	After HBASE-21879	Delta %
GC Overview			
Young GC Count in 3 hour	296	306	+3.8%
STW per GC (ms)	29 ms	25 ms	-13.8%
Eden Usage (GB)	30GB	30GB	0.0%
GC Details			
Update RememberSet (ms)	1.010 ms	1.000 ms	-0.9%
Region Chosen Each GC	960	960	0.0%
G1 Object Copy (ms)	0.294 ms	0.252 ms	-14.2%
QPS & Latency			
Get QPS	257580 op/s	262900 op/s	+2.1%
Avg Latency (ms)	0.4634 ms	0.4540 ms	-2.0%
P99 Latency (ms)	2.141 ms	1.331 ms	-37.8%
P999 Latency (ms)	12 ms	9 ms	-25.0%

No difference in young generation usage.

Throughput increased about 2.1%

Won’t affect the latency & throughput of normal read path.

Abstract

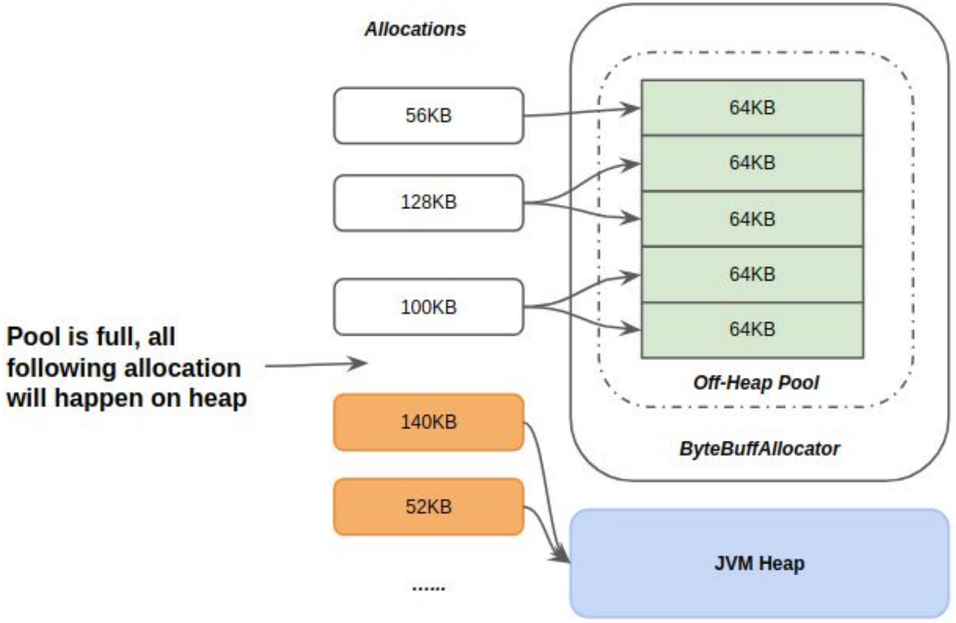
- ❏ Background: Why offheap HDFS block reading
- ❏ Idea & Implementation
- ❏ Performance Evaluation
- ❏ **Best Practice**

The introduced config keys in 3.0.0 & 2.3.0

- hbase.server.allocator.pool.enabled
  - Whether the region server will use the pooled offheap ByteBuffer allocator;
  - Default: true
  - The ~~hbase.ipc.server.reservoir.enabled~~ is deprecated one since 2.3.0.
- hbase.server.allocator.minimal.allocate.size
  - Allocated as a pooled offheap ByteBuff if desired size >= this value, otherwise just use heap ByteBuff.
  - Default: hbase.server.allocator.buffer.size / 6
- hbase.server.allocator.max.buffer.count
  - How many buffers are there in the pool
  - Default: 1890 (2MB * 2 * hbase.regionserver.handler.count / 65KB)
  - The ~~hbase.ipc.server.reservoir.initial.max~~ is deprecated since 2.3.0
- hbase.server.allocator.buffer.size
  - The byte size of each ByteBuffer
  - Default: 65KB (why not 64 KB ?)

Practice#1

- Please make sure that there are enough pooled DirectByteBuffer in your ByteBuffAllocator.



Practice#1

➢ Please make sure that there are enough pooled DirectByteBuffer in your ByteBuffAllocator.

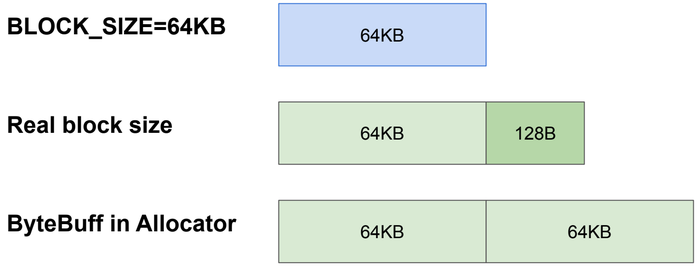
$$heapAllocationRatio = \frac{heapAllocationBytes}{heapAllocationBytes + poolAllocationBytes} \times 100\%$$

ⓘ Need to **enlarge max.buffer.count** or **decrease minimal.allocation.size** if meet the following condition.

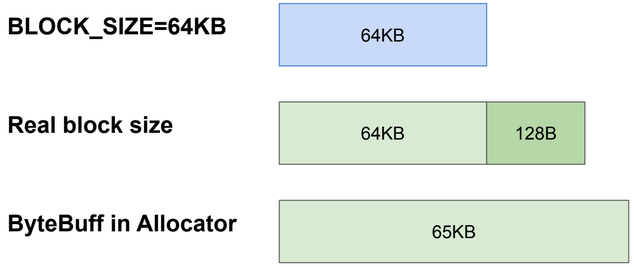
$$heapAllocationRatio \geq \frac{minimal.allocation.size}{allocator.buffer.size + minimal.allocation.size} \times 100\%$$

Practice#2

- Please make sure the buffer size of allocator is greater than your block size.
  - Assume block size=64KB, your block will be 64KB + delta.
    - delta come from: checksum / header / other meta data.
  - If buffer size is also 64KB, then the block will be composited by two 64KB ByteBuffers.
  - SingleByteBuff have simple data structure and access faster than MultiByteBuff.



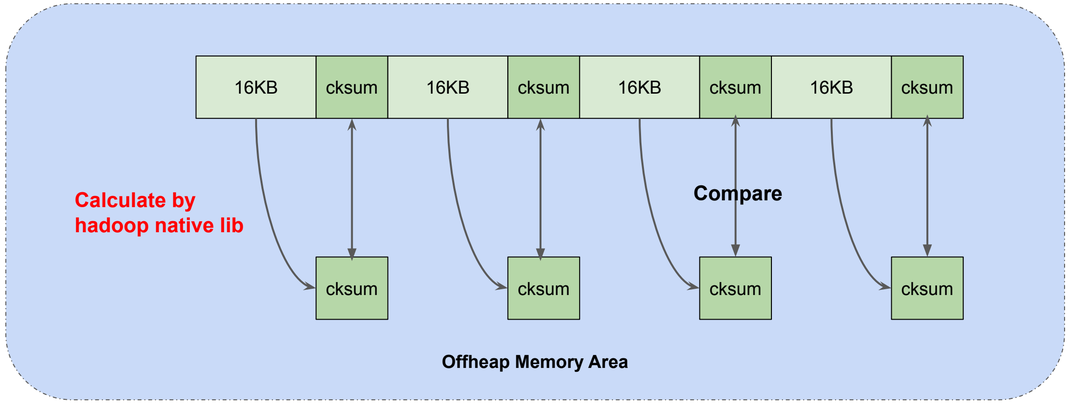
hbase.server.allocator.buffer.size=64KB



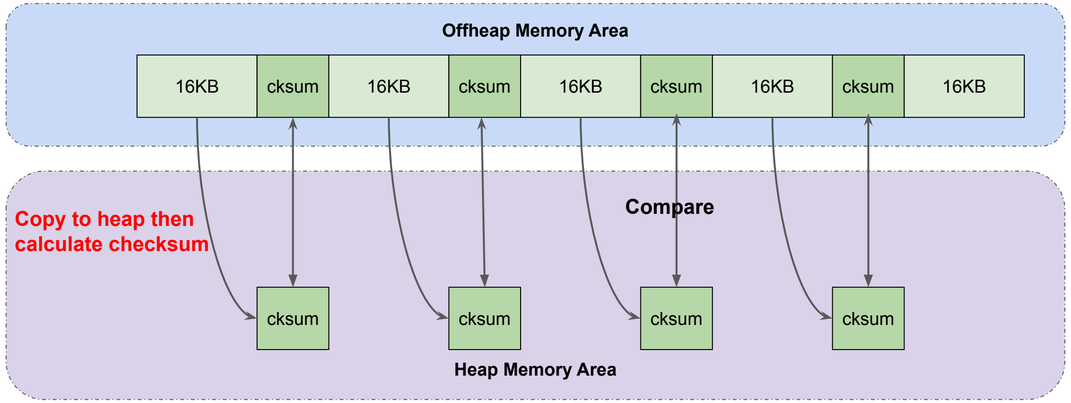
hbase.server.allocator.buffer.size=65KB

Practice#2

- Please make sure the buffer size of allocator is greater than your block size.
  - SingleByteBuff's checksum can speed by using hadoop native lib, while MultiByteBuff can not.



Calculate checksum for Block backended by SingleByteBuff



Calculate checksum for Block backended by MultiByteBuff

Practice#3

- If disabled block cache, need to consider the index/bloom block size.
  - The default hfile.index.block.max.size is 128KB.
  - buffer size set to 130KB will be better if disabled block cache (based on *practice#2*)

Practice#4

- Prevent to OOM or full GC for huge cell reading
  - Huge cell won’t cache in BucketCache, because BucketCache only cache <513KB block by default.
  - All reads will direct to HDFS, reading them into pooled BB will protect the RS from OOM or full GC.

Thanks !