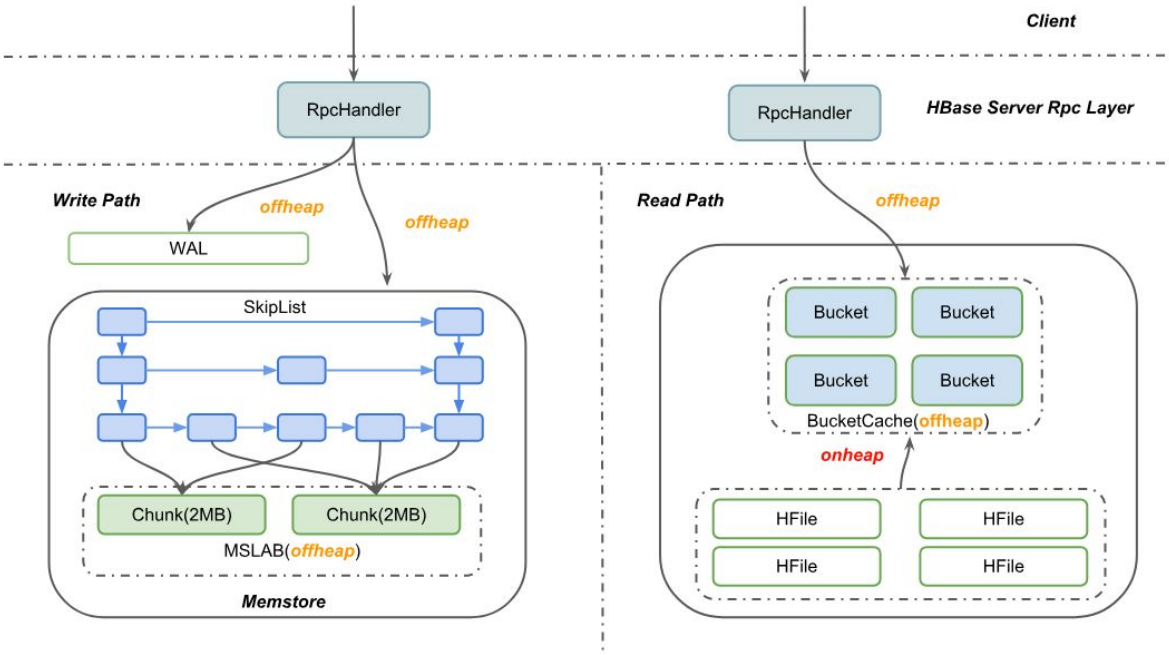
HBASE-21879 Read HFile ’s Block into ByteBuffer directly.

1. Background

For reducing the Java GC impact to p99/p999 RPC latency, HBase 2.x has made an offheap read and write path. The KV are allocated from the JVM offheap, and the memory area in offheap won’t be garbage collected by JVM, so upstreams need to deallocate the memory explicitly by themself. On the write path, the request packet received from client will be allocated offheap and retained until those key values are successfully written to the WAL log and Memstore. The ConcurrentSkipListSet in Memstore does not directly store the Cell data, but reference to cells, which are encoded in multiple Chunks in MSLAB, this is easier to manage the offheap memory. Similarly, on the read path, we’ll try to read the BucketCache firstly, if the Cache misses, go to the HFile and read the corresponding block. The workflow: reading block from cache OR sending cells to client, is basically not involved in heap memory allocations.



However, in our performance test at XiaoMi, we found that the 100% Get case is still seriously affected by Young GC:

Picture.1 QPS and latencies in 100% Get Case (5 Nodes)

Picture.2 G1GC related metrics in 100% Get Case

In above pictures, the p999 latency is almost the same as G1GC STW cost (~100ms). After HBASE-11425 , almost all memory allocations should be in the offheap, there should be rarely heap allocation. However, we found that the process of reading the Block from HFile is still

copied to the heap firstly, the heap block won't free unless the WriterThread of BucketCache flushes the Block to offheap IOEngine successfully. In the pressure test, due to the large amount of data, the cache hit rate is not high (~70%), many blocks are read by disk IO, so a large number of young generation objects are allocated in Heap, which eventually leads to the raising Young GC pressure.

2. Basic Idea

Idea to eliminate the Young GC is: *read the block of HFile to offheap directly*. We didn't accomplish this before because HDFS does not support the ByteBuffer pread interface, also because of its complexity. For the ByteBuffer pread, we have issue HDFS-3246 to track this, so further HDFS version will support this soon.

Now consider the implementation:

Firstly, we need a global ByteBuffAllocator for RPC. when reading a block in RPC, we'll do:

1. Allocate ByteBuff from ByteBuffAllocator, and read the data from HFile to ByteBuff;
2. Cache the HFileBlock in BucketCache which means putting the ByteBuff into RAMCache;
3. WriteThread in BucketCache will flush the ByteBuff into IOEngine in BucketCache asynchronously (don't block the RPC);

In theory, if the RPC finished, we need to free the ByteBuff, but we can't: ByteBuff can also be referenced by other RPC because it's still in RAMCache and can be accessed by others before it's flushed to IOEngine.
Also we cannot free the ByteBuff even if writer thread finished the flushing because some other rpc may still not finish and reference to the ByteBuff; On the other hand we MUST free the ByteBuff once writer thread finished the flushing if nobody ref to it, otherwise memory leak happen.

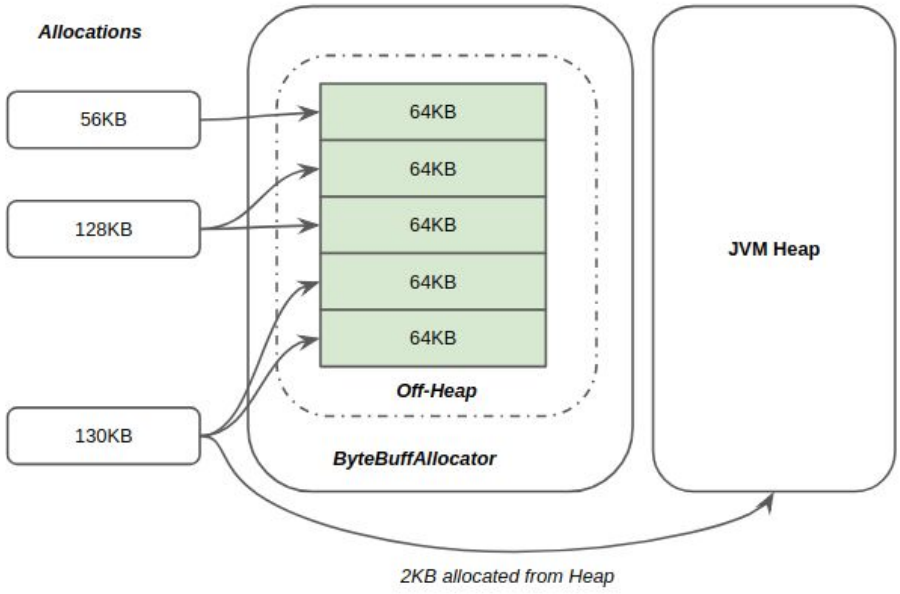
So the solution will be:

1. Maintain a refCount in ByteBuff, once allocated, its refCount will be initialized to 1;
2. If it put 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.

3. Implementation

3.1 General Allocator

As the basic idea part said, the first thing is to design a global ByteBuffAllocator.



In HBASE-11425 , we have introduced an offheap memory management policy as following:

1. Set a max memory size for the off-heap usage, and divide the whole off-heap memory as many small DirectByteBuffer which are composed as a **reservoir**. each DirectByteBuffer has a fixed size, such as 64KB. You can tune this by:
    a. *hbase.server.allocator.buffer.size*
    b. *hbase.server.allocator.max.buffer.count*
2. Set another minimal off-heap allocation size, only when the desired size greater than or equal to the minimal allocation size, we allocate from the off-heap reservoir; otherwise just allocate the desired size from heap memory. Because it's too wasting to allocate an fixed-size DirectByteBuffer.
3. If the desired size is less than 64KB, the allocation will be an SingleByteBuff backend by one fixed-size DirectByteBuffer.
4. If the desired size is greater than 64KB, then the allocation will be an MultiByteBuff composited by multiple fixed-size DirectByteBuffer (or with an extra HeapByteBuffer for the reminder based rule#2);

While the memory management policy was only used for receiving request or sending response in RPC layer before, now we need the policy to pass through the entire reading RPC path. So in HBASE-21916, we wrap all this as a global **ByteBuffAllocator**. The allocator implementation is

simple enough and sufficient to meet our current needs. If we have higher performance requirements in the future, we can turn to Netty 's PooledByteBufAllocator, it has the following advantages:

1. Thread safe , high concurrency and memory utilization.
2. Support reference count to track memory life cycle and detector to debug memory leak;
3. Cache coherence, which means each allocation is mapping to a continuous memory area. It's more efficient to access.

3.2 Reference Count

Each ByteBuff allocated from ByteBuffAllocator will have an RefCnt to track its life cycle, the RefCnt is a class which extends from Netty's AbstractReferenceCounted, use unsafe method(or safe way if unsafe is not available) to maintain the atomic reference count value. Once the reference count value decreasing to zero, the RefCnt will trigger the registered **Recycler** to deallocate the allocated NIO ByteBuffers (Putting those ByteBuffers back to the **reservoir** of ByteBuffAllocator). We also have a check that nobody can access the ByteBuff with an reference count = 0, if someone call the methods, they will get an IllegalArgumentException.

The ByteBuff#duplicate() and ByteBuff#slice() are a bit complex. Releasing either the duplicated one or the original one will free its memory, because they share the same NIO ByteBuffers. If you want to retain the NIO ByteBuffers even if the origin one called ByteBuff#release(), you can do like this:

```
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.
```

With implementing the Netty's ReferenceCounted interface, we can easily replace our hbase defined ByteBuff by Netty's ByteBuf , moving to Netty's PooledByteBufAllocator can also be easy as the 3.1 part described.

3.3 Make downstream API can accept ByteBuff or ByteBuffer as arguments

The first obstacle when implementing the block off-heap reading is: many downstream APIs our HFileBlock depends on are implemented by using byte[], so the first thing is to abstract all of them into ByteBuff or NIO ByteBuffer interfaces.

The three main API we need to refactor are:

#1 ByteBuffer positional read interface (HBASE-21946);

The ByteBuffer pread issue HDFS-3246 is still in progress, so we can not just use the HDFS ByteBuffer pread interface in our HBase project. Here we abstract an ByteBuff pread method in our HBase BlockIOUtils but with an implementation by using byte[] copying. Once the HDFS ByteBuffer pread is available(Need upgrade the HDFS version in our HBase pom.xml), we can re-implement the ByteBuff pread method by using HDFS pread.

#2 Checksum validation methods (HBASE-21917);

An HFileBlock can be backended by an SingleByteBuff or MultiByteBuff:

1. For SingleByteBuff, it has only one NIO ByteBuffer, and the hadoop-common support the NIO ByteBuffer checksum validation now, so can just call the hadoop's interface;
2. For MultiByteBuff, it has multiple NIO ByteBuffers, so we can only update the checksum value incrementally by copying the bytes into a small heap buffer (such as byte[256]).

Actually, almost all of the HFile Block are about 64KB, so it would be a SingleByteBuff because our ByteBuffer chunk is 4MB in ByteBufferArray. The above case#2 is a rare case though higher cost.

#3 Block decompression methods (HBASE-21937);

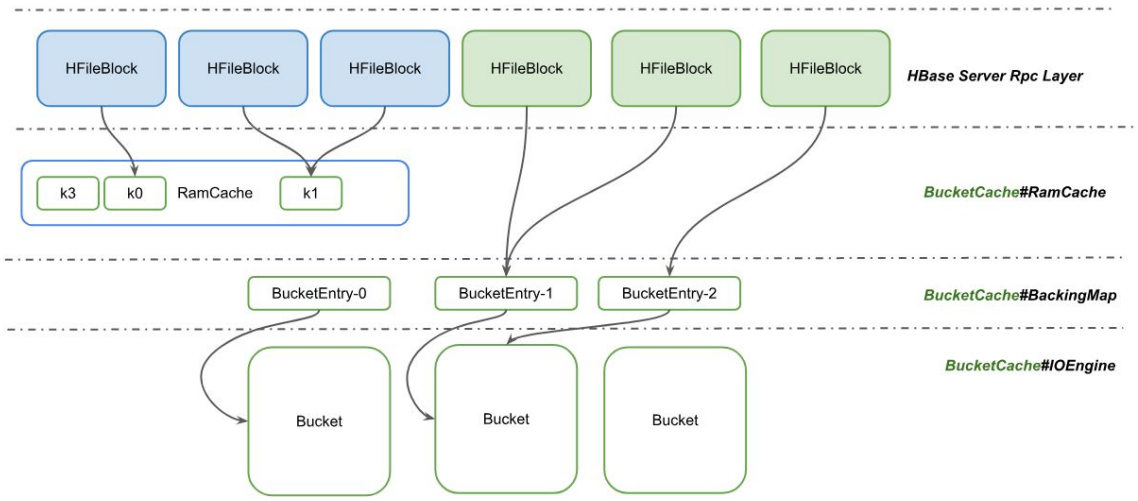
We currently implement it as:

1. Read block data to off-heap ByteBuffers;
2. Uncompress the off-heap ByteBuffers into a small heap buffer incrementally;
3. Copy the bytes from small heap buffer into off-heap ByteBuffers, which is wrapped as the unpack HFileBlock.

In fact, I think we can improve the implementation as a pure ByteBuffer uncompression without any heap->direct or direct->heap copy, it will be a TODO issue.

3.4 Unify the refCnt of BucketEntry and HFileBlock into one.

Now, both HFileBlock and BucketEntry have their own different reference count. The HFileBlock has the RefCnt object inside, it mainly used for tracking the life cycle of the block's ByteBuff; The BucketEntry also maintain an atomic integer by itself, mainly used for counting how many RPC path are referring the BucketEntry, if no reference then can evict the block from BucketCache.
See the below picture, there are two RPC are referring the BucketEntry-1, so the BucketEntry's atomic reference count will be 2; For BucketEntry-0, there's no RPC referring it so BucketCache can evict the block if cache is full.



Now considering how to unify the reference count of BucketEntry and HFileBlock into one.

There are three cases:

Case.1 : HFileBlock is backend by BucketCache#RamCache. For both the off-heap and on-heap case , its nio ByteBuffers are allocated from the global ByteBuffAllocator. So its refCnt will be changed as the following steps:

1. If adding a new HFileBlock into RamCache, then refCnt ++;
2. If removing an existed HFileBlock from RamCache when flushing to IOEngine, then refCnt --;
3. If RPC path ref to block which still in RamCache, then refCnt ++;
4. If RPC path release its blocks, then refCnt --;

Once the refCnt decrease to zero, then its recycler will put all of the nio ByteBuffers back to the ByteBuffAllocator;

Case.2 : HFileBlock is backend by a shared memory IOEngine, such as off-heap IOEngine, SharedMemoryMmapIOEngine and so on. In this case, different HFileBlocks mapping to the same (offset, len) in BucketCache are *sharing the same NIO ByteBuffers,* which are allocated from **BucketAllocator**. Each (offset, len) pair is maintained by an BucketEntry, and cached in BucketCache#backingMap, we do this for lightweight maintaining in BucketCache. In the picture, the BucketEntry-1 is the sharing one. Even if there's no RPC reference to this kind of BucketEntry, we still can not recycle the BucketEntry from BucketCache unless the used memory size reaches the threshold in BucketCache and the BucketCache marked those BucketEntry as EVICTED.

For BucketEntry and HFileBlock, both of them has an refCnt. How do we define the refCnt here ? We'll choose a simple way: if HFileBlock-A and HFileBlock-B is mapped to the same BucketEntry-X, then all of them HFileBlock-A, HFileBlock-B and BucketEntry-X will share a single refCnt instance. The refCnt will be changed as the following steps:

1. When the WriteThread flush an HFileBlock to IOEngine, a new BucketEntry will be created, and its refCnt will be initialized to 1.
2. If remove an HFileBlock from backingMap, such as evicted a block from BucketCache, then the BucketEntry will do refCnt --;
3. If RPC path start to reference to an HFileBlock, then the HFileBlock will do refCnt ++;
4. Once the RPC finished and stop reference to the HFileBlock, then HFileBlock#release will do refCnt --;

In this way, we can unify the refCnt of both HFileBlock and BucketEntry as one refCnt. And for the previous recycle logics:

1. BucketCache#returnBlock;
2. BucketCache#freeSpace;
3. ….

All of them can be done by the refCnt's Recycler#free. we unify the logic here: Once refCnt decrease to zero, then start to deallocate without caring about who or when will accomplish this.

Case.3: HFileBlock is backend by an exclusive memory IOEngine, such as FileIOEngine. In this case, different HFileBlocks mapping to the same (offset, len) in BucketCache are different NIO ByteBuffers, which are allocated from Java Heap directly. In theory, it is easy to handle, because each of HFileBlock have an independent refCnt, once release the HFileBlock, if decreasing to zero, then just do nothing and wait for the JVM Garbage Collection. The block eviction in BucketCache also won't impact the HFileBlocks in RPC. So the refCnt can be changed as the following:

1. If RPC ref to an HFileBlock, create a heap hfile block with an initialized refCnt(=1) based on the (offset, length);
2. Once the RPC finished, then HFileBlock#release will do refCnt --. Its refCnt will decrease to zero, while the Recycler#free will do nothing.
3. The BucketEntry won't need an refCnt;

BUT design like above, would need a separate path to handle the EXCLUSIVE IOEngine, and need to handle the difference seriously. To simplify both the EXCLUSIVE and SHARED case, I plan to use the same design for them, which means *the BucketEntry will also have an refCnt inside and maintain as case.2 described even if it's mapping to cache in EXCLUSIVE IOEngine.*

3.5 Combined the BucketEntry sub-classes into one

Now, we have three kinds of BucketEntry:

1. BucketEntry: it maintains the offset/len for an block, it has no refCnt inside. Actually, we can think that this kind of BucketEntry are usually used to track an **EXCLUSIVE** memory area for IOEngine.

2. SharedMemoryBucketEntry: it extends from BucketEntry, but with an extra AtomicInteger reference count, which have an initialized zero value(It's different from our newly introduced RefCnt object with an initialized value = 1). It's used to track a **SHARED** memory area for IOEngine, which means all HFileBlocks mapping to the same SharedMemoryBucketEntry are sharing the same NIO ByteBuffers. We can give the reference count a definition: how many RPC path are using those shared NIO ByteBuffers.

   Besides, it has a volatile flag **markedForEvict** inside, means whether the eviction policy has chosen this BucketEntry as an evicted one. And if the BucketEntry has been marked, once its reference count decreased to zero, then we'll do the deallocation.

3. UnsafeSharedMemoryBucketEntry: it also extends from BucketEntry, but with an extra volatile int refCount, increase or decrease by using unsafe methods, which is faster than the AtomicInteger. Other parts are similar with SharedMemoryBucketEntry.

In HBASE-21957 , we'll abstract those three sub-classes into one: BucketEntry.
Firstly, we use the same way to handle NIO ByteBuffers for both SHARED and EXCLUSIVE IOEngine, even if not meaningful for EXCLUSIVE IOEngine, but can help to simplify the abstraction and implementation. So we can think all BucketEntry are shared type. About the **markedForEvict,** we removed the flag and instead we redefined the RefCnt as: how many path (NOT only RPC path) are reference to the shared memory area, the BucketCache will also be considered as a separate reference path.
Second, the two shared sub-classes are easy to abstract, because our newly introduced RefCnt class(based on Netty) have handled both the safe and unsafe case.

4. Performance evaluation

Overview

To evaluate the performance of before/after HBASE-21879, we build the following clusters.

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

And we have designed three test cases to prove the performance improvement after HBASE-21879:

1. Disabled BlockCache cache ;
2. CacheHitRatio~65%;
3. CacheHitRatio~100% .

For case.1 and case.2, we will use the following YCSB workload:

```
table=ycsb-test
```

```
columnfamily=C
recordcount=10000000000
operationcount=10000000000
workload=com.yahoo.ycsb.workloads.CoreWorkload
fieldlength=100
fieldcount=1
clientbuffering=true

#debug=true

readallfields=true
writeallfields=true

readproportion=1
updateproportion=0
scanproportion=0
insertproportion=0

requestdistribution=zipfian
```

For case.3, we only need small data set to ensure the cacheHitRatio to be 100%, so we changed the recordcount from 10^10 to 10^7. Accordingly, the data set size decreased from 288GB per node to 5GB per node.

About the testing workflow: we create the table first, then balance regions and have a YCSB load. After that we will do a major compaction to ensure all nodes have a locality with 1.0, then run the 100% Get workload.

In theory, the case.1 will have a great performance improvement because if all Gets read HDFS directly in before-HBASE-21879 case then it will make so many heap allocations and have a very high GC pressure on heap, while after HBASE-21879 all reads will allocate from pooled offheap bytebuffers and almost no heap allocation. As the cacheHitRatio increasing, the difference between before-HBASE-21879 and after-HBASE-21879 will decrease, when cacheHitRatio is 100%, they should have no much difference in both throughput and latency.

Case.1 Disabled BlockCache

Configuration: here we choose the buffer.size=130KB, because the point #3 in best practice section.

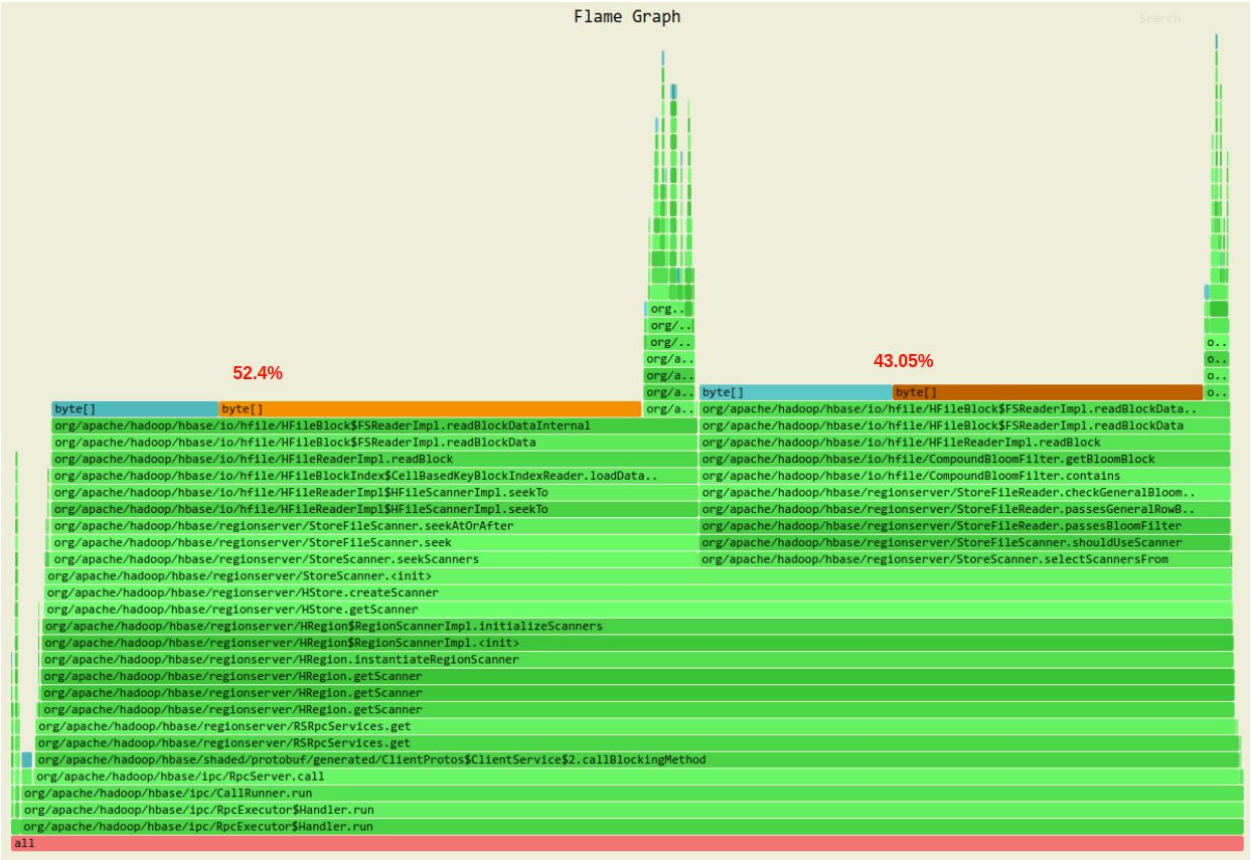
```
# Disable the block cache
hfile.block.cache.size=0

# 2GB~130KB*16131 for ByteBuffAllocator
hbase.server.allocator.max.buffer.count=16131
hbase.server.allocator.buffer.size=133120
hbase.server.allocator.minimal.allocate.size=0
```

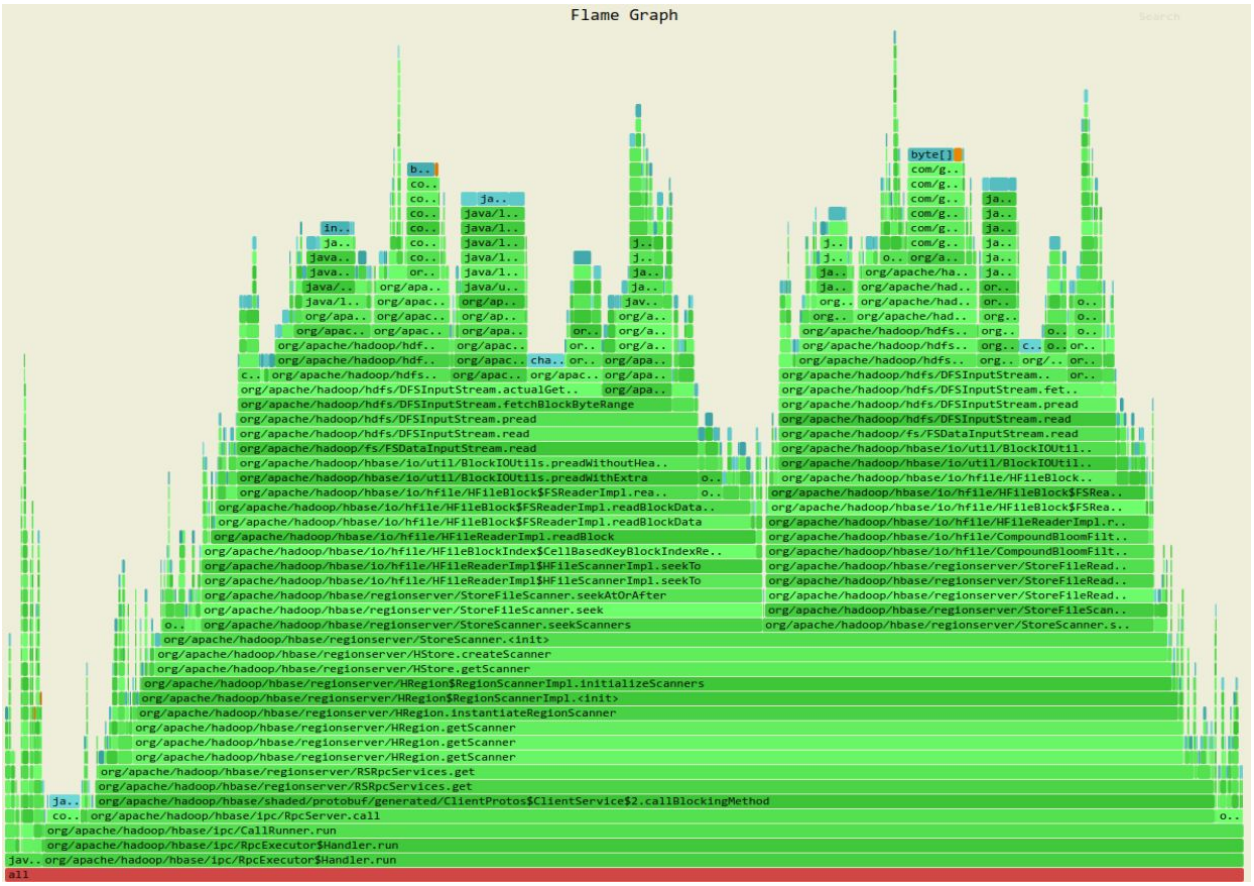
Performance: Note that Delta% column is calculated by the formula: (afterCase - beforeCase) / beforeCase * 100%.

	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%

Heap Allocation FlameGraph - Before HBASE-21879



Heap Allocation FlameGraph - After HBASE-21879



Conclusion:
In the performance table, we can clearly see that the 100% Get throughput with HBASE-21879 increased about 17%, the young generation bytes size decreased about 81%. They are very good points for HBase clusters with HBASE-21879.
Also from the heap allocation flame graph, we can see that we eliminated almost 95% block heap allocation, After HBASE-21879, almost all of the heap allocation are from DFSClient, it may worth us take some time to optimize the heap allocation in DFSClient in the future.

Case.2 Enabled BlockCache: ~ 65% CacheHitRatio

```
# Enabled the combined block cache
hfile.block.cache.size=0.175

# BucketCache
hbase.bucketcache.ioengine=offheap
hbase.bucketcache.size=36864 # UNIT: MB -- 36GB
```

```
# ByteBuffAllocator configuration
# 2GB~130KB*16131 for ByteBuffAllocator
hbase.server.allocator.max.buffer.count=16131
hbase.server.allocator.buffer.size=133120 # 130KB
hbase.server.allocator.minimal.allocate.size=0
```

Performance:

	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%

Conclusion:
We can see that the GC pressure decreased after HBASE-21879, while it's effect is not good as the disabled blockCache benchmark case now. Because as the cache hit ratio increased, we will have much less request to read the HDFS, for example, our QPS is 101530 op/s now, we only have about 101530/5*0.65=7107 Get/second which will allocate the heap memory for its block, it's about 7107*65KB=461MB young generation memory. So the difference in GC pressure will be decreased.
We also observed that the throughput (After HBASE-21879) decreased a bit, I've checked the CPU flame graph and found that the HFileScannerImpl.loadBlockAndSeekToKey increased

from 8.6% (before HBASE-21879) to 10% (after HBASE-21879), seems the encoded offheap cells comparison have some impact on the throughput, we will continue to optimize this in future issues.

Case.3 Enabled BlockCache: ~ 100% CacheHitRatio

To ensure the HBASE-21879 won't make high overhead, we made a benchmark in a small test data set which will have a 100% cacheHitRatio.

	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%

Conclusion:
From the table, we can see that the HBASE-21879 won't affect the throughput and latency, besides the HBASE-21879 have even some advantages when comparing the p99/p999 latency.

5. Best Practice

Firstly, we will introduce several configurations about the ByteBuffAllocator:

1. hbase.server.allocator.minimal.allocate.size
2. hbase.server.allocator.max.buffer.count
3. hbase.server.allocator.buffer.size

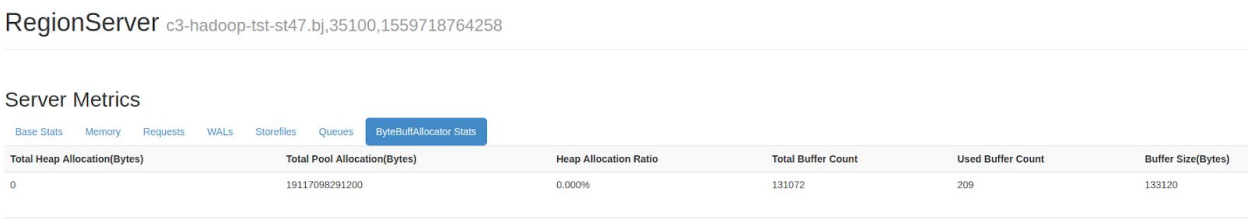
Second, we have some suggestions for HBase administrators:

#1 *Please make sure that there are enough pooled DirectByteBuffer in your ByteBuffAllocator.*

The ByteBuffAllocator will allocate ByteBuffer from DirectByteBuffer pool first, if there's no available ByteBuffer from the pool, then it will just allocate the ByteBuffers from heap, then the GC pressures will increase again.

By default, we will pre-allocate 2MB *2 for each RPC handler (The handler count is determined by the config: hbase.regionserver.handler.count, it has the default value 30) . That's to say, if your hbase.server.allocator.buffer.size is 65KB, then your pool will have 2MB *2 / 65KB * 30 = 1890 DirectByteBuffer. If you have some large scan and have a big caching, say you may have a RPC response whose bytes size is greater than 2MB (another 2MB for receiving rpc request), then it will be better to increase the hbase.server.allocator.max.buffer.count.

The RegionServer web UI also has the statistic about ByteBuffAllocator:



If the following condition meet, you may need to increase your max buffer.count:

$$heapAllocationRatio \geq \frac{hbase.server.allocator..minimal.allocate.size}{hbase.ipc.server.allocator.buffer.size} \times 100\%$$

#2 *Please make sure the buffer size is greater than your block size.*

We have the default block size=64KB, so almost all of the data block have the block size: 64KB + delta, whose delta is very small, depends on the size of last KeyValue. If we use the default hbase.server.allocator.buffer.size=64KB, then each block will be allocated as a MultiByteBuff: one 64KB DirectByteBuffer and one HeapByteBuffer with delta bytes, the HeapByteBuffer will increase the GC pressure. Ideally, we should let the data block to be allocated as a SingleByteBuff, it has simpler data structure, faster access speed, less heap usage.

On the other hand, If the blocks are MultiByteBuff, so we have to validate the checksum by a temporary heap copying (see HBASE-21917), while if it's a SingleByteBuff, we can speed the checksum by calling the hadoop' checksum in native lib, it's more faster.

Please also see: HBASE-22483

#3 *If disabled block cache, need to consider the index/bloom block size.*

Our default hfile.index.block.max.size is 128KB now, which means the index/bloom block size will be a little greater than 128KB. So If still use 65KB buffer size in block cache disabled case, then all index/bloom will be MutiByteBuff again. The problems in #2 will happen again.

Please see: HBASE-22532

6. Related work

1. https://issues.apache.org/jira/browse/HDFS-3246
2. https://issues.apache.org/jira/browse/HDFS-14535
3. https://issues.apache.org/jira/browse/HDFS-14541
4. https://issues.apache.org/jira/browse/HDFS-14483