Introduction

Block cache is mechanism with which we can get better latency and throughput for read operations in HBase. It avoids the need to fetch HFile block data from underlying FS. HBase gives 2 types of Block caches now. These are

1. LRU cache (aka L1 cache) which caches block data in on heap memory area.
2. Bucket cache (aka L2 cache) which can cache data in on heap/ off heap/ local file depending the IOEngine being configured.

L1 cache size can be configured as a percentage of max heap size for RS process. Default percentage is 40%. The higher the max heap size for RS process, the more L1 cache we get. . If we allocate a very large value for the Xmx we might suffer from large full GC delays. Typical used value for max heap size is 32-48 GB. Said this, L2 cache (with off heap mode) is a way to have larger block cache and cache more data. In off heap bucket cache, we reserve memory blocks of 4KB size in off heap area. This is known as Buckets. When an HFile block is been cached, the block data is been split and stored in these fixed sizes buckets. So each block data might be residing in one or more buckets in block cache. The read path in HBase expects the HFileBlock data to be available in an on heap buffer. (Contiguous byte array) This makes a need to create a new on heap buffer and copy the data from buckets to it during read from bucket cache. This is not so efficient especially when there is a huge read load in the system. The work here tries to remove such an expectation from the read path (data has to be backed by single on heap buffer).

As part of HBASE-11425, we want to support E2E off heap for the HBase read path. When we implement E2E off heap support, we have to make sure to select the best performing data structure/framework. The below tests are done in this regard.

Possible options

The 2 best possible options comes are

* + Java NIO ByteBuffer
  + Netty ByteBuf

NIO ByteBuffer has its pros such as, our RPC layer already deals with it. Also there is an HDFS read API which takes ByteBuffer and if we can pass an off heap buffer, we can avoid some data copies. But NIO ByteBuffers can be slow (boundary checks and/or some methods may not inline). This makes us look to Netty ByteBuf as a possibly better performing alternative.

As a first step, I am comparing read performance of NIO ByteBuffer over Netty ByteBuf. (Netty version is 4.0.23) First the test was conducted on Java 7 (u71). We will repeat the test for on heap as well as off heap buffers. The test is a JMH micro benchmark test (attached below) which will just read a long, int and short values added to buffer.



This is the result for on heap buffer comparison

Result "nettyOnheap":

54425661.568 Â±(99.9%) 605586.471 ops/s [Average]

(min, avg, max) = (54296903.902, 54425661.568, 54520593.217), stdev = 93715.177

CI (99.9%): [53820075.098, 55031248.039] (assumes normal distribution)

Result "nioOnheap":

42555946.872 Â±(99.9%) 636228.094 ops/s [Average]

(min, avg, max) = (42456142.234, 42555946.872, 42665845.979), stdev = 98457.002

CI (99.9%): [41919718.779, 43192174.966] (assumes normal distribution)

So we can see netty on heap buffer is having ~28% gain on avg throughput.

The next test is our main focus which compares the 2 off heap buffers (The test is there in the above attached file – NettyVsNio.java)

Result "nettyOffheap":

63870175.131 Â±(99.9%) 995092.439 ops/s [Average]

(min, avg, max) = (63639532.370, 63870175.131, 63956565.261), stdev = 153991.656

CI (99.9%): [62875082.692, 64865267.569] (assumes normal distribution)

Result "nioOffheap":

73432240.675 Â±(99.9%) 875770.928 ops/s [Average]

(min, avg, max) = (73230736.849, 73432240.675, 73512274.300), stdev = 135526.521

CI (99.9%): [72556469.747, 74308011.603] (assumes normal distribution)

The result is opposite when it comes to off heap. The NIO has ~15% better throughput. We can also see the off heap reads are better than on heap in both NIO and netty.

The same tests are repeated on Java 8 (u45)

Result "nettyOnheap":

54316427.593 Â±(99.9%) 630883.110 ops/s [Average]

(min, avg, max) = (54181522.411, 54316427.593, 54402760.769), stdev = 97629.860

CI (99.9%): [53685544.483, 54947310.703] (assumes normal distribution)

Result "nioOnheap":

43986070.263 Â±(99.9%) 592908.851 ops/s [Average]

(min, avg, max) = (43874471.123, 43986070.263, 44074777.524), stdev = 91753.301

CI (99.9%): [43393161.412, 44578979.113] (assumes normal distribution)

Result "nettyOffheap":

60294244.232 Â±(99.9%) 535211.991 ops/s [Average]

(min, avg, max) = (60190199.038, 60294244.232, 60389283.049), stdev = 82824.648

CI (99.9%): [59759032.242, 60829456.223] (assumes normal distribution)

Result "nioOffheap":

73624934.040 Â±(99.9%) 1199369.976 ops/s [Average]

(min, avg, max) = (73359137.517, 73624934.040, 73774723.887), stdev = 185603.831

CI (99.9%): [72425564.064, 74824304.016] (assumes normal distribution)

This shows Netty is still better when it comes to on heap buffers but off heap buffer case NIO is clearly winning.(Over 20% better throughput)

Test based on Unsafe based reads

In current HBase code we use Unsafe reads wherever/whenever possible. This gives much better throughput. For on heap buffer we do unsafe based reads from the underlying byte[] of the buffer. (Refer toXXX APIs and Compare APIs in org.apache.hadoop.hbase.util.Bytes )

Similarly we can do Unsafe based reads from off heap buffer as well. (See attached ByteBufferUtils.java for sample implementations)



The Unsafe based read avoids different range checks and not inlining issues with NIO ByteBuffers.

The next test compares Unsafe based reads for both on heap and off heap buffers for NIO and netty. This is done on Java 8

Result "nettyOnheapUnsafe":

67043879.587 Â±(99.9%) 936655.012 ops/s [Average]

(min, avg, max) = (66895527.439, 67043879.587, 67220494.435), stdev = 144948.400

CI (99.9%): [66107224.574, 67980534.599] (assumes normal distribution)

Result "nioOnheapUnsafe":

85020139.348 Â±(99.9%) 958398.850 ops/s [Average]

(min, avg, max) = (84852536.471, 85020139.348, 85152231.117), stdev = 148313.283

CI (99.9%): [84061740.499, 85978538.198] (assumes normal distribution)

Result "nettyOffheapUnsafe":

72470705.850 Â±(99.9%) 976023.259 ops/s [Average]

(min, avg, max) = (72254169.128, 72470705.850, 72605629.000), stdev = 151040.680

CI (99.9%): [71494682.591, 73446729.109] (assumes normal distribution)

Result "nioOffheapUnsafe":

85117459.317 Â±(99.9%) 1734771.838 ops/s [Average]

(min, avg, max) = (84721261.570, 85117459.317, 85297460.074), stdev = 268457.862

CI (99.9%): [83382687.478, 86852231.155] (assumes normal distribution)

Here the NIO is outperforming Netty for both on heap and off heap.

Conclusion:

This makes it clear that it is better to select Java NIO especially considering we will move to Java 8 soon.

Average Throughput Table (ops/s)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | JAVA 7 | | JAVA 8 | |
| Netty | NIO | Netty | NIO |
| On heap | 54425661.568 | 42555946.872 | 54316427.593 | 43986070.263 |
| On heap Unsafe | 78899841.105 | 84773518.918 | 67043879.587 | 85020139.348 |
| Off heap | 63870175.131 | 73432240.675 | 60294244.232 | 73624934.040 |
| Off heap Unsafe | 73382366.181 | 85149536.280 | 72470705.850 | 85117459.317 |

Note: Ideally the Unsafe based reads should yield similar performance numbers in Netty and NIO. Still we see significant difference (Netty slower). So we went ahead and checked the source code of Netty. In the version I tested with (4.0.23) there is a referencing counting and memory leak detection mode is running by default. This causes some extra ops and extra call level indirection (actual ByteBuf object needs to be wrapped in a wrapper instance). In case of on heap Unsafe reads, the call to ByteBuf#array() is not so cheap as its counterpart in NIO. There is reference counting happening with every call. For Off heap ByteBuf#isDirect() and ByteBuf#memoryAddress() calls seem not so cheap. Again the check for memory leak detection causes extra ops. If this wrapper was not in place, The C2 compiler would have inlined the memoryAddress() call. The wrapper prevents it. (With an older version, 4.0.4, this extra cost was not seen)

When I have repeated the above tests with ResourceLeakDetector level as DISABLED, I am able to get almost similar performance for offheap with Unsafe. (Even without Unsafe case also same throughput)

Average Throughput Table (ops/s) with ResourceLeakDetector disabled

|  |  |  |
| --- | --- | --- |
|  | JAVA 8 | |
| Netty | NIO |
| On heap | 54096550.334 | 43986974.529 |
| On heap Unsafe | 78655647.254 | 85024718.935 |
| Off heap | 72599007.565 | 73132087.933 |
| Off heap Unsafe | 83613659.416 | 84514777.734 |

NIO On heap vs off heap

The next test is for comparing the read performance for NIO on heap buffer (HeapByteBuffer – HBB) vs NIO off heap buffer (DirectByteBuffer – DBB). (The results are already there above – On Java 8)

Result "nioOnheap":

43986070.263 Â±(99.9%) 592908.851 ops/s [Average]

(min, avg, max) = (43874471.123, 43986070.263, 44074777.524), stdev = 91753.301

CI (99.9%): [43393161.412, 44578979.113] (assumes normal distribution)

Result "nioOffheap":

73624934.040 Â±(99.9%) 1199369.976 ops/s [Average]

(min, avg, max) = (73359137.517, 73624934.040, 73774723.887), stdev = 185603.831

CI (99.9%): [72425564.064, 74824304.016] (assumes normal distribution)

This is plain reads from buffer using ByteBuffer APIs (not Unsafe based optimization). We can see that the offheap buffer is giving ~60% better avg throughput over the on heap.

The next test compares the Unsafe based reads on NIO on heap buffer vs off heap buffer

Result "nioOnheapUnsafe":

85020139.348 Â±(99.9%) 958398.850 ops/s [Average]

(min, avg, max) = (84852536.471, 85020139.348, 85152231.117), stdev = 148313.283

CI (99.9%): [84061740.499, 85978538.198] (assumes normal distribution)

Result "nioOffheapUnsafe":

85117459.317 Â±(99.9%) 1734771.838 ops/s [Average]

(min, avg, max) = (84721261.570, 85117459.317, 85297460.074), stdev = 268457.862

CI (99.9%): [83382687.478, 86852231.155] (assumes normal distribution)

Both throughputs come out almost same.

The improvement of DBB over HBB for the plain read case (no Unsafe) comes from the fact that for DBB we do read long/int etc using Unsafe only and it is a direct single instrinsic call. The HBB reads bytes one after the other and construct the long/int using the math within the HBB code.

Ops/sec

|  |  |  |
| --- | --- | --- |
|  | On Heap | Off Heap |
| Plain reads | 43986070.263 | 73624934.040 |
| Unsafe reads | 85020139.348 | 85117459.317 |

Byte[] UnsafeComparer vs DBB UnsafeComparer

We use UnsafeComparer which can compare byte arrays in an efficient way. This comparison operation is very critical in the HBase read path and most of the CPU time is spent in this. So the next test that I have done is comparing the throughput of the existing Comparer against a new one which acts on ByteBuffers. (To be precise, on DBBs) We just now saw that we can do Unsafe based reads from the Direct ByteBuffers. For this test I am taking a key of 135 bytes long and keys are equal.

Doing the test on Java 7, the Unsafe comparison based on byte[] seems better performing wrt avg throughput.

Result "offheap":

14444367.534 Â±(99.9%) 404076.245 ops/s [Average]

(min, avg, max) = (14361441.649, 14444367.534, 14500027.800), stdev = 62531.24

CI (99.9%): [14040291.289, 14848443.779] (assumes normal distribution)

Result "onheap":

28525186.418 Â±(99.9%) 265776.648 ops/s [Average]

(min, avg, max) = (28469492.636, 28525186.418, 28557006.901), stdev = 41129.23

CI (99.9%): [28259409.770, 28790963.066] (assumes normal distribution)

When the same test is repeated with Java 8, the DBB based Unsafe comparison looks better. It gives more than 22% throughput improvement over byte[] based.

Result "offheap":

38205893.545 Â±(99.9%) 265309.769 ops/s [Average]

(min, avg, max) = (38164316.169, 38205893.545, 38246801.754), stdev = 41056.980

CI (99.9%): [37940583.776, 38471203.313] (assumes normal distribution)

Result "onheap":

31166847.740 Â±(99.9%) 430242.970 ops/s [Average]

(min, avg, max) = (31069672.467, 31166847.740, 31214107.795), stdev = 66580.576

CI (99.9%): [30736604.770, 31597090.710] (assumes normal distribution)

|  |  |  |
| --- | --- | --- |
| UnsafeComparer | Java 7 | Java 8 |
| ByteArray based | 28525186.418 | 31166847.740 |
| DBB based | 14444367.534 | 38205893.545 |

Based on these tests, when we use Java 8, the NIO is giving better results and the off heap reads/compare is better than the on heap based.

Conclusion

Based on these micro benchmark tests, we would like to select NIO off heap bytebuffer as data structure for our off heap path in reads. We already have off heap based BlockCache. But we will do an on heap copy of block data, every time we read from this cache. HBASE-11425 will avoid any such copy and reads can happen over this off heap buffer.