HBASE OFFHEAP READ PATH SUPPORT (HBASE-11425)

John Anoop S

Ramkrishna S Vasudevan

(Intel Corporation)

Table of Contents

Introduction

HBASE-11425 is aimed at making HBase reads serve directly from the off heap memory. Currently HBase code path expects the HFileBlock data to be available in an on heap buffer. Block cache is an efficient way to improve the read latency and throughput. We know that the L1 cache (LRU Cache) size is limited by the max heap memory which can be allocated for the RS java program. 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 between 16GB – 32GB.

We have L2 cache, Bucket Cache, which can be backed by off heap memory area. When the block is served from L2 cache, we will have to copy the entire data to a newly created on heap buffer. This is not efficient. Even when the Bucket Cache is backed by on heap, we cannot avoid the copy. This is because within BC one block data will be cached by multiple buckets. Each bucket's size is fixed to 4KB. In read path, one HFileBlock data is expected to be in one BB.

To avoid this, we try to support reads to be backed by off heap memory and enable the Cell interface to be backed by off heap buffers. So this JIRA indirectly aims at avoiding the multiple data copying happening in the read path.

MultiByteBuffer and Prevention of Block Eviction when in use

As stated earlier, Bucket cache recreates an Hfileblock from the multiple BB buckets backing it. So in order to avoid this copy, we try to serve this multiple BB buckets as a single unit with the help of MultiByteBuffer. It is nothing but a wrapper on a set of BBs. Provides all APIs in BB like getXXX, putXXX, duplicate(), slice() etc. HFileBlock data is represented by this MBB.

Bucket cache evicts blocks and frees the associated buckets when it is having no space for caching new blocks. In the existing way, any blocks can be evicted as we will copy the buckets to a new BB when a block is being read from BC. But now we will avoid any such copy. The data is served directly from the BC's buckets. So we have to prevent the eviction of actively read blocks. **We use a simple reference counting mechanism to do so**. The reference is incremented whenever the block is being read from the BC (Using getBlock API). We have added a new API to BC to return a block back to BC after the read on that block is over. The reference count is decremented then. The block can be evicted iff its reference count is zero.

Note that the LRU Cache does not have this block reference counting happening as that does not deal with BBs and deals with the HFileblock objects directly.

Considering ByteRanges and BBs

There are some thoughts in using ByteRange instead of BB for HFileBlock backing data structure. As a first step we tried to check the usage of ByteRanges and ByteBuffers. In HBase we already have ByteRanges.

Advantages of BR

- Manipulate byte arrays with offset and index
- Ensures that we don't do lot of boundary value condition checks (Range checks)
- There is a clear position based tracking on BRs.
- Easily in-linable methods whereas the same is not the case with BB.

Decision to use BB

Initially proceeding on with BR we found that when we need a structure that could work with off heap, then it should be something like a BB (internally we have Direct BB and on heap BB) or we should have our own implementation of the BB. The new BR implementation will act as a wrapper.

First point to be noted is that **BBs cannot be sub classed** as they are package private and hence there is no way of having a 'lesser overhead' type of BB. In that case we would end up in writing our own BB type implementation.

- RPC layer interacts with BB only. So any native/new implementation of BB would end up in converting them to BB in RPC layer. Or find a newer RPC implementation that would deal with new types of implementation.
- As mentioned in the introduction para, if later we need to serve BB directly from HDFS then BB would be the easiest convention and currently HDFS provides APIs using BBs where we could ask HDFS to populate the BB that we pass, with the data being read.
- Wrapping a BB inside BR is nothing but a level of indirection only. Because BB will anyway have its own offset and index, and BR will add one more to it. Any operation like duplicate(), slice() would internally call the BB APIs only.

Cell APIs with BB

Currently Cells work with byte [] using the getXXXArray() APIs along with getXXXOffset() and getXXXLength(). When the read paths are off heap BB backed, we will have to add buffer based APIs also to cell.

New APIs to Cell

- getRowBuffer()
- getFamilyBuffer()
- getQualifierBuffer()
- getValueBuffer()
- getTagsBuffer()
- hasArray()

The new API hasArray() would have a major role in deciding whether the cell is backed by BB or byte[]. Users have to use hasArray() API and decide whether to go with getXXXBuffer() or getXXXArray().

Please note that the Cells in the memstore are still KV based (byte [] backed).

Take a case of a cell that comes from an off heap backed HFileBlock. In that case **hasArray() would be false** and it is recommended that we use getXXXBuffer() in that case. Using getXXXArray() would throw UnSupportedOperationException. Note that even if the HFileBlock is on heap BB we do not support getXXXArray() APIs. This is because, in such a case the getXXXOffset() API might clash with the usage of getXXXArray() or getXXXBuffer(). The offset within the BB/ byte[] might be different. The BB itself can

carry offset and length info via position and limit. But this will need we have to duplicate the Cell backing BB object every time the getXXXBuffer() is being called. We have tested this and the BB duplicate will cost us. The APIs are heavily used in Comparators at Store and Region levels. This will make so many short living objects creation also. That is why we decided to go with usage of getXXXOffset() and getXXXLength() API usage also along with buffer based APIs.

In case of cell in memstore which is a plain KeyValue object, **hasArray() would return true.** In such cases it is prudent to use getXXXArray() APIs along with the getXXXOffset() and getXXXLength(). Here again if user uses getXXXBuffer() things would work but there would be BB.wrap() that happens on the byte[] that is backing the KV.

Advantages

What are the advantages by having only the above defined APIs

- The same getXXXOffset() and getXXXLength() would work with getXXXBuffer() and getXXXArray() APIs.
- No confusion on the user side rather than having more APIs like getXXXBufferOffset() and getXXXBufferLength() along with the existing getXXXOffset and getXXXLength() APIs.

Disadvantages

- Exposed APIs in CP and filters would have to be changed and their implementation should be based on the Cell.hasArray(). So we could target this on HBase 2.0?
- Using getXXXBuffer when hasArray() is true will have lot of BB.wrap() happening and multiple short lived objects would be created whenever the cell is referenced in the comparators. One must be prudent before using these new getXXXBuffer() APIs of cell.

Comparators

All the Comparators will have two paths – hasArray() true and hasArray() false.

This is the best way considering the profiler outputs that we got. If we know that a cell has hasArray() true (a KV) then it is better to use array manipulations using the getXXXArray() rather than getXXXBuffer() which would create multiple smaller BB objects.

Usage of Unsafe

So as to avoid the overhead on BB manipulation on the various APIs it provides, we have provided an ***unsafe*** way of manipulating the APIs so that we get better performance and avoid some of the range checks that BB does on its implementation. This is not new because already we have Unsafe support for byte [] backed Cells.

Unsafe comparators works by avoiding the direct BB based API calls. It tries to use the memory offset and manipulates the BB on the native memory itself without having to bring the BB's data on heap.

Considering the above factors we went in with BB usage in the read path and support Cells to accommodate BB along with byte [] as it currently does.

Note: The write path is still KV based.

Bottleneck in RPC layer

We have avoided the data copy from the BC to HFileBlock. There is another copying happening at RPC layer to create Cell Blocks. We initially thought that since our Cells are Bb backed in the RPC layer we could try to write these Cells directly into the socket rather than creating a Cell Block. But on profiling and testing, we found that pushing the data in these BBs backing the Cells to the RPC had a negative impact on performance and the socket was not able to accept multiple small amount of data pushed to it at a faster rate. Instead if we had created a cell Block and then the cell block was pushed to the socket it was found to be better.

The recent work on reusing the buffers at RPC layer (HBASE-13142) has added significant benefits to the performance. The reports below are before these changes were applied.

Performance reports

The below graphs illustrates the Avg, 95th and 99th percentile values under different test scenarios

The Y axis denotes time in 'ms'.

Offheap Bucket cache settings

HEAP_SIZE = 9GB

DirectMemory size for offheap case – 24 G

'hfile.block.cache.size' – 0.2

'hbase.bucketcache.size' – 7000

Onheap Bucket cache settings

HEAP_SIZE = 9GB

'hbase.regionserver.global.memstore.size' – 0.3

'hfile.block.cache.size' – 0.5

LRU cache settings

HEAP_SIZE = 9GB

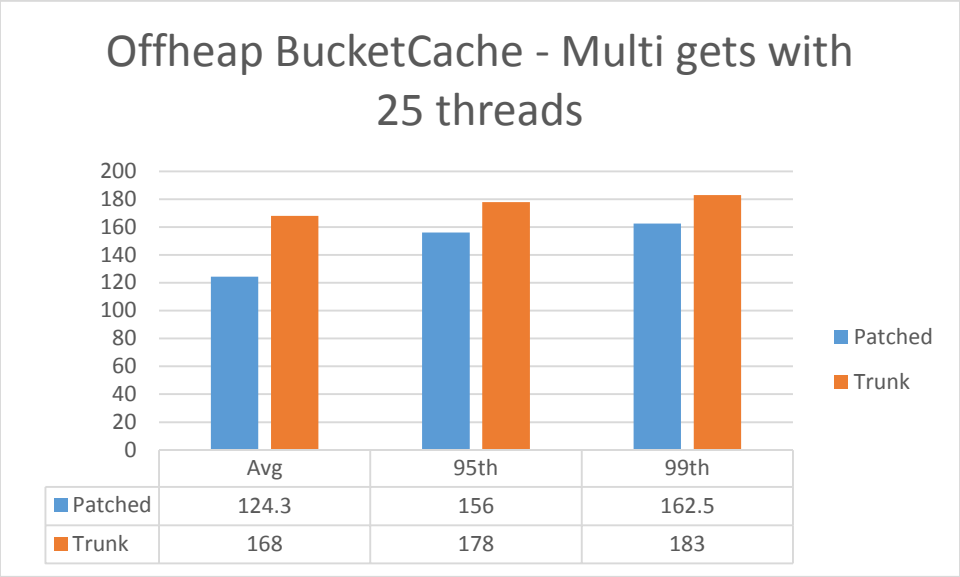
'hbase.regionserver.global.memstore.size' – 0.3

'hfile.block.cache.size' – 0.5

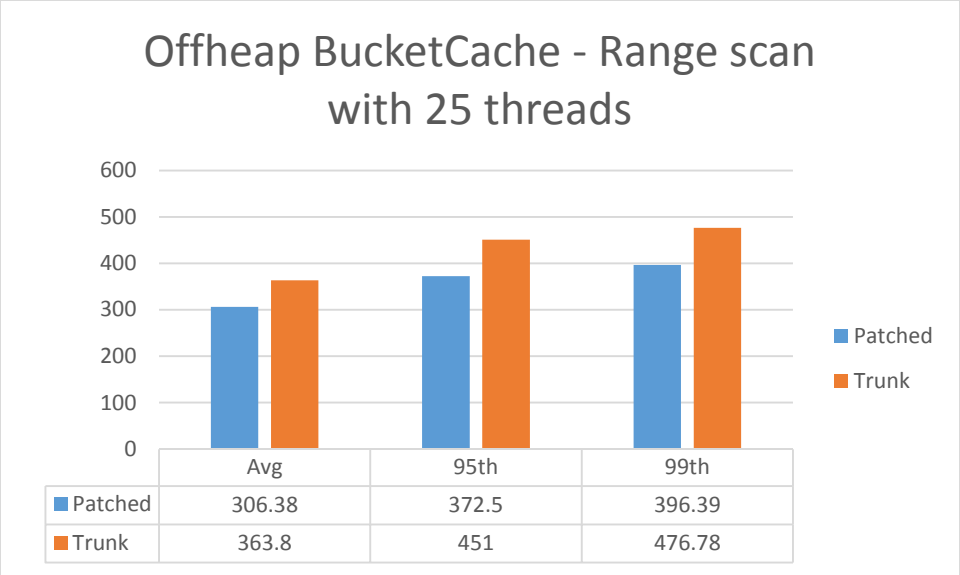
The reports attached here are tests carried out using PE tool. The readings taken are after prepopulating the block cache. In order to take the percentile readings – the multi gets and the range scans were done one SAME set of row keys so that we don't have the random nature in them.

Also note that these tests are done in pure read based scenarios. No other operations were done. All the data are from Block cache and no data from memstore.

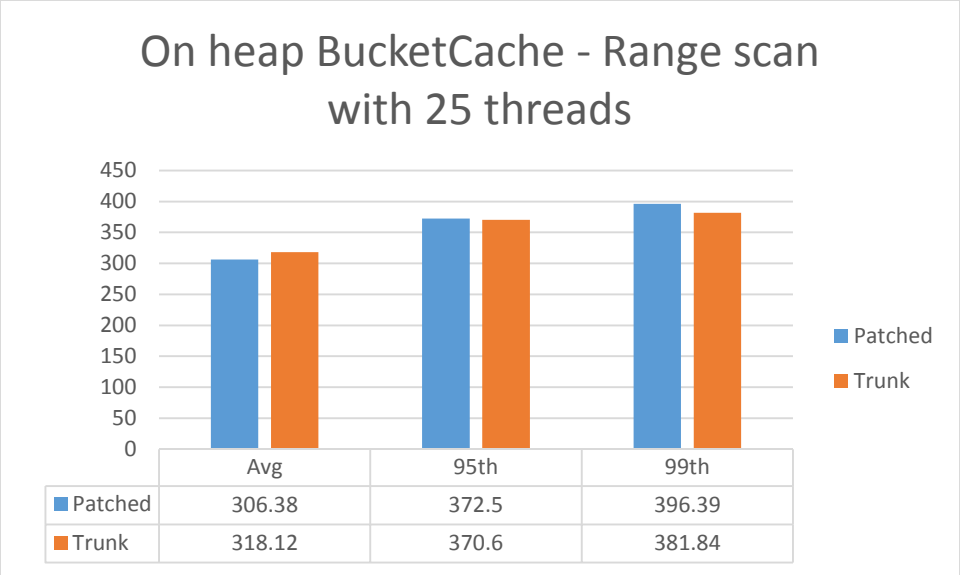
The LRU cache testing does not involve block reference counting because in case of LRU cache it is only the HFileBlock objects that are cached and there is no need to worry about block getting evicted under us.



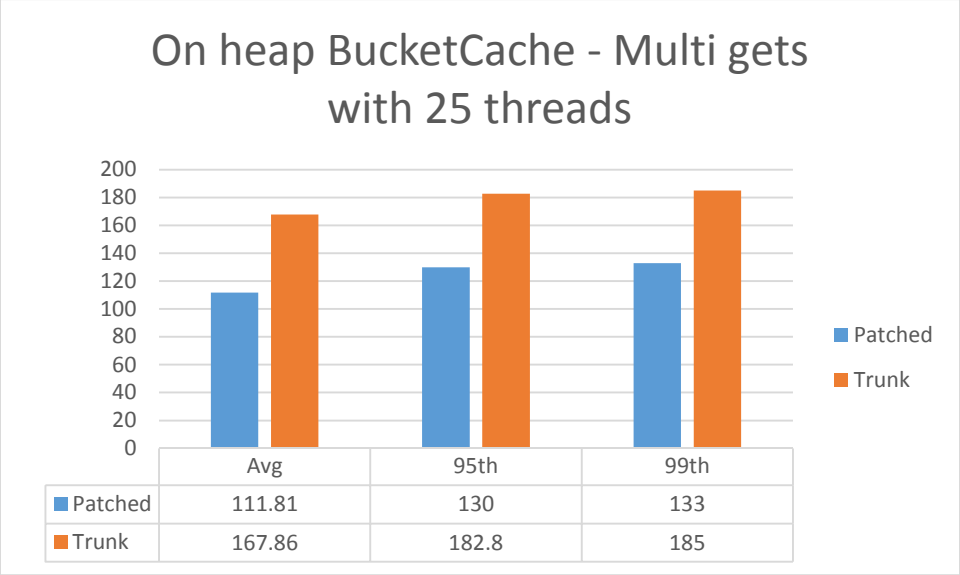
	Avg	95th	99th
Patched	124.3	156	162.5
Trunk	168	178	183



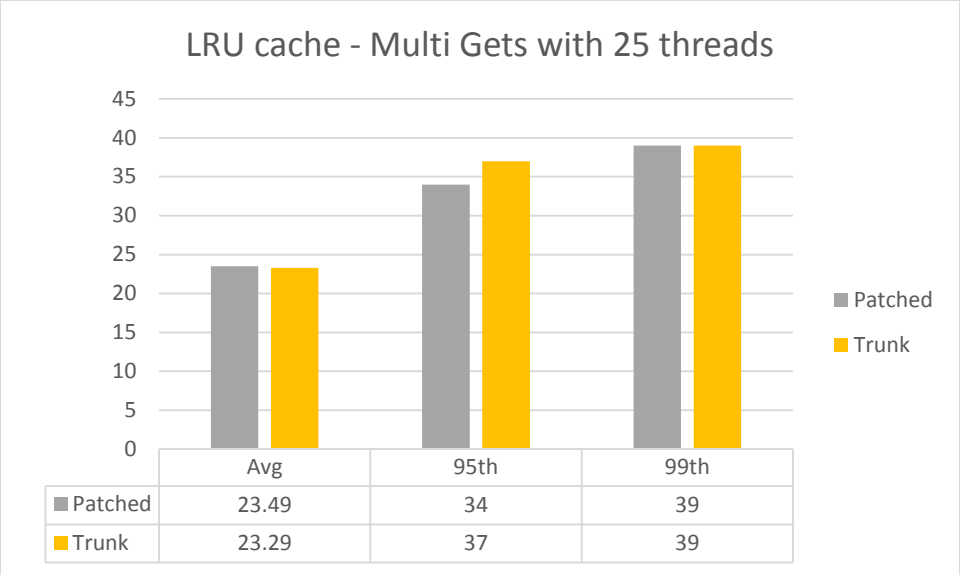
	Avg	95th	99th
Patched	306.38	372.5	396.39
Trunk	363.8	451	476.78

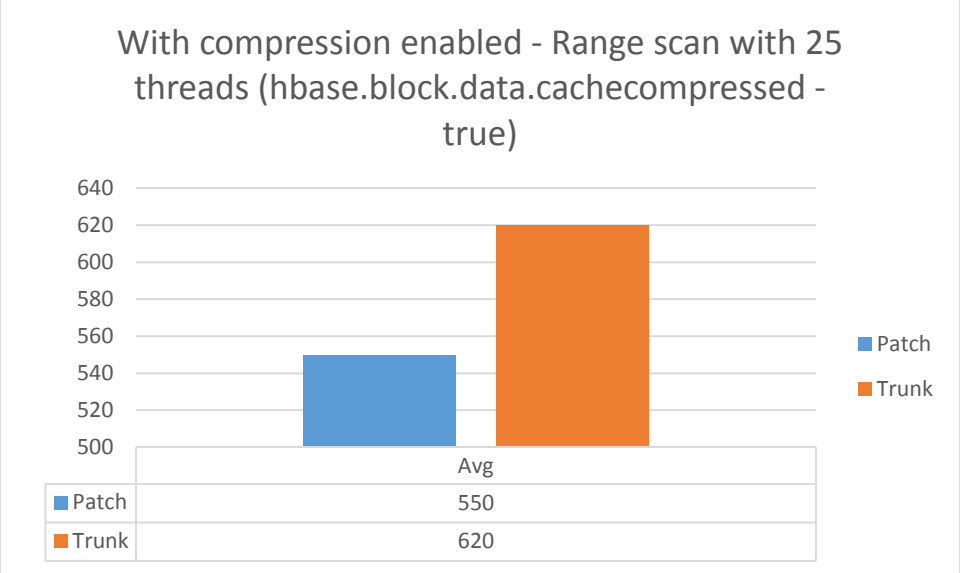


	Avg	95th	99th
Patched	306.38	372.5	396.39
Trunk	318.12	370.6	381.84



	Avg	95th	99th
Patched	111.81	130	133
Trunk	167.86	182.8	185





Comments/suggestions that can be taken up

Usage of ServerCell

If we would want the Client facing APIs that use Cell, not to have the BB based APIs in them, then we can extend the Cell to create a ServerCell (a cell that would be used only the server side) which will have all the BB based APIs. This would mean that the Client facing Cell API would remain clean and only the APIs in filters and CPs need to be changed to use the newly added ServerCell. But this has got lot of changes

involved and also we need to see the impact of these changes. This change is not done in the POC. After brainstorming and further analysis on the pros and cons we could do this change.