HBASE-12295

Preventing block eviction while serving reads from offheap memory

Anoop Sam John

Ramkrishna S Vasudevan

Problem statement

In Bucket Cache an HFileBlock's data is stored by splitting it in multiple buckets. And when block is read from cache, we are copying these bucket data into a temporary on heap buffer. So immediately after this copy, any of these buckets are available for possible eviction.

But with HBASE-11425 we try to serve the reads directly from the off heap/on heap cache which means that when the cells in the results have references to these blocks, those blocks cannot be evicted till we are sure that these cells are consumed, in other words, we remove the references to the memory area which constitute these blocks.

Solution

Introduction

We try to solve this by introducing a simple framework where we try to track the blocks getting used in a scan/get using **reference counting** mechanism. Whenever a block is retrieved from Bucket Cache, its ref count gets incremented. Whenever a read operation is completed the ref count is decremented.

Whenever eviction thread selects blocks for eviction, we will consider the ref count also and won't allow eviction of any block with ref count>0. This mechanism can ensure that none of the blocks that are currently getting served in a read are evicted, thus avoiding result corruption.

Reference counting in block cache layer

The reference counting happens purely in the block cache layer. We also try to ensure that the incrementing and decrementing operation happens on those blocks which are coming out of L2_CACHE. For L1_CACHE (LRU) only the objects are cached and there should be no issues with eviction and memory references.

The scanners involved in the read path ensures that all the blocks that it has referred to in its path are safely returned back and the cache implementation ensures that the ref count is decremented on those blocks.

A new API is added in the Block cache layer

returnBlock(BlockCacheKey cacheKey, HFileBlock block)

Which internally decrements the ref count depending on the implementation of the cache.

Pls note that, in the following paragraphs when we say return block it means that we try to decrement the reference count on that block.

How does reference counting work

For the reference counting mechanism to work we need to keep track of all the blocks that are getting used in the current read flow. A single scan can involve multiple blocks. We collect all the blocks referred in a read flow at the HFileScanner layer and also specifically holds the reference to the current block (latest block) in use. (Note that these blocks can be from HDFS or from the block cache).

This HFileScanner scanner directly calls the Block Cache's new returnBlock() API when a read is completed – the call to which happens from the upper layers of scan.

We need to consider the following cases for read paths

- Gets issued from clients
- Scans issued from clients
- Reads involving CPs

The clients are of two types

- java based client
- non-java based clients.

In all of the above cases – our aim to is to get rid of the references to the blocks that the cells in the result hold on to.

Java client using PayloadCarryingRpcController

Any scan/get operation issued from a java based client returns the result using a PayloadCarryingRpcController. (For gets it is being added as part of HBASE-13158). A CellScanner is created over the results and the CellScanner is iterated over to create a Cellblock. This cellblock creation happens in the RPC layer where the cells are written to a ByteBufferOutputStream. The cells that are in the results are referring to the blocks from the block cache until the cell blocks are created. (This is where we get the shared memory area – cells referring to the memory area of the blocks which is prone to eviction).

Gets

A get() call will open up a scanner, gets the row cells and closes the scanner in HRegion level. The results are sent to the RpcServer via the RSRpcServer and write back to client via Responder. For the block eviction to happen, we should ensure that our results are not referring to the Shared memory area (the memory area containing the block from block cache).

As mentioned earlier the cellblock creation would ensure that the cells are copied to an output stream and thus we are sure that the reference of these cells to the shared memory area is removed. To achieve this, we are converting the cells in the result to a **Cellblock buffer (in Region level itself)** and pass it to RpcServer via the Payload of the controller as opposed to how it happens currently (currently the cell block is created in RPC layer). We have added an overridden API in HRegion which takes RpcController as parameter. Once the cell block is

created we are free to return the blocks in turn which decrements the ref count on those blocks. Calling close() on the scanner would propagate this call till the HFileScanner which in turn would return the blocks.

HRegion.get(get, RpcController)

The regionserver's get() will make use of this new API, in other words the client issued gets() will make use of this new API.

The existing HRegion.get() API is still used for the CP cases and non-java client cases. We discuss about CP and how we do reference counting in those cases in the upcoming sections.

Scan

Every next() call return N rows or even partial rows. Only at end, the Scanner created will get closed. We could hold all the block being referred and close them when scanner.close() happens but this would mean that we are holding up blocks though a next() call has returned the results back to the client.

To avoid this after every next() call creates a list of Results, **we create a cell block out of it like we did for the get() case** and **we return back all the blocks used in the read except the current block (the latest block that was read)**. The latest block cannot be returned back because it could be used for the scan's next 'next()' call. Note that in case where we know that the next() call indicates there are no more results to be fetched, we return back all the blocks including the latest one.

This is a kind of partial close after every batch of scan. The RsRpcServices layer will do this partial close /return after every next() call and we create cell blocks and piggy back the created cell blocks on the PayloadCarryingRpcController. Once a cell block is created we are sure that we have got rid of all the references to shared memory and it is safe to return the blocks.

We have added the overloaded version of getScanner() in HRegion for any scans originating from the java based client.

HRegion.getScanner(scan, RPCController).

The existing getScanner() API could be used for the non-java client cases and the CP cases.

Reads involving CPs

CP can create a scanner on a Region and retrieve Cells. These cells they might keep in its state too. So there is no clear way to know when these cells would be used and when the references of these cells to the shared memory area would be released. In such a case we have to recreate a Cell copying the data from the shared memory area into a dedicated byte array so that we get rid of the references to the shared memory area. When cells are created from L2_CACHE, we will mark them with a new interface SharedMemoryCell**.** We copy the cells to only when the cells are of type SharedMemoryCell.

In order to do this copy, when the CP wraps a scanner using the pre/post hook, we mark the actual scanner on which the wrapping happens such that it knows that the copy should happen. So when the next() call happens on the wrapped scanner, the results would be copied as and when the results are created.

Non-java based client

For the non-java based clients both in case of gets() and scan() we use the existing getScanner() and get() API and ensure that the results are always copied to a new byte array so that there are no references to the shared memory area. This in principle is same as what we do for the CP cases.

One may wonder, if the goal for HBASE-11425 is to serve directly from the offheap/onheap memory area without doing the copy, we are still doing this copy to do the reference counting and freeing the references, it should be noted that this copy is not a copy of the entire block as it happens now. Only the required results are copied.

New Scanner interfaces

In order to do this return block or decrementing the ref count, we are adding a new interface with the following API.

We introduce an interface FinalizableScanner with an API finalizeScan(boolean finalizeAll). (Any other better name here?)

The RegionScannerImpl, StoreScanner, StoreFileScanner, KeyValueHeap and HFileScanner implements this interface.

Note that we are not adding this new API to the RegionScanner interface, if done it would mean that any CP wrapping the scanner would also need to implement the new interface. In order to avoid that we make the RegionSCannerImpl to implement the new interface.

Compaction cases

The compaction reads also would go through this flow where we keep track of the blocks being referenced and finally return them after they are used. In case of compaction there is a forceful eviction on the blocks of the compacted files that happens because the compacted files are no longer valid. But we cannot do this eviction if the block was being part of any parallel read operations. In such a case we don't evict the block if the ref count > 0, instead we mark those blocks with a Boolean. When the actual read operation completes we decrement the ref count to 0 and then evict that block.

Conclusion

Our preliminary round of testing suggests that this scheme works fine and we are able to serve the reads directly from the offheap/onheap memory of the L2_CACHE.