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.

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.

CacheType and MemoryType of HFileBlocks

The blocks that are retrieved from the Block Cache are of two types – one that is retrieved from a L1 cache like block cache and other from L2 cache like Bucketcache. In future there can be only L2 cache and the mechanism of how these block types work inter-operably may change but we will always have blocks coming either from bucket cache or LRU cache. The whole idea of serving blocks from offheap applies to blocks that comes from caches that support offheap (in our case the offheap mode of Bucket cache).

In order to clearly know the type of blocks coming from L1 or L2 we mark the blocks coming from BucketCache as CacheType – L2 and those blocks coming from Lru as L1. The ref counting applies to L2 blocks only.

Remember in Bucket cache there is a RAMQueueEntry that holds the blocks from the HDFS before serializing it to the buckets. So when there is a read and a block is available in the RAMQueueEntry they are just the blocks from HDFS and there is no ref counting needed for those blocks. So the blocks that is coming out of the buckets are only marked as L2 type.

The returnBlock() impl for other cache types like LRU cache is going to be a noop as they denote blocks from L1 cache.

Coming to the MemoryType of the blocks – now in BucketCache we can have different implementations. ByteBufferIOEngine and FileIOEngine types (This could have more impl in future also).

The engine knows clearly if the blocks formed from these engines and to be used in the read path is going to refer to a shared memory area of the offheap. In case of FileIOEngine there is no such thing because the serialized BB from the files are copied to a native buffer and so there is no direct reference. So for the ByteBufferIOEngine case the blocks created out of it is SHARED_MEM whereas in case of FileIOEngine the blocks in NON_SHARED_MEM. In our read path if we see that a block is created out of a SHARED_MEM type we will create a cell of type SharedMemoryCell so that in cases of CP, non –java clients we could do a clone() of the cells the details of which we can see in coming sections.

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).

How Gets and Scans work currently

Scans are stateful. To issues a scan we create a scan and then do HTable.getScanner(); Using this scanner we do subsequent next() calls.

So for the scan() in the server side the call that comes to RsRpcServices first creates the scan using Region.getSCanner() API.

When the client calls next() on this scanner(), the RsRpcSErvices (that has cached the created RegionScanner) issues a next() on the RegionScanner(). This next() call fetches those many rows based on the limit and once all the Results (List<Cell>) are collected, the CellBlock is created for the java cases and for the non-java cases we create the PBed result object.

So here in case of scan() the cellblock is created in the RSRpcServices layer.

Gets are stateless. From the client side, HTable.get() is the only API the client uses, that returns a single result. On the server side the get() request lands on the RsRpcServices layer, which issues a Region.get() call. Inside HRegion, the get() will create a scanner using getScanner() and a RegionScanner.next() call to create a single Result. The result is now created as a cellblock for Java clients and a PBed result is created for a non-java case.

RpcCallBack mechanism

An RpcCallBack interface is created with a method run() in it.

void run();

The RpcCallContext is added with a

void setCallBack(RpcCallBack callback)

So every RpcCallContext can be associated with a callback.

We use this RpcCallBack to ensure that we return the blocks in case of scans/gets.

The RegionScannerImpl is made of type RpcCallback. So any scanner created for the scans/gets will automatically become a type of RpcCallback.

Scans

When a scan is created at the RsRpcServices layer we associate the current call context with two types of RpcCallBack – one the current RegionScanner and other RegionScannerSessionCloseableCallBack.

Why we need two types of callbacks is that – we now try to return the blocks once the cellblocks are created at the RpcServer layer. So the current RpcCallContext needs to know whether the read is partially over aka scan.next() with moreResults or the read is completely over aka scan.next() with no more results or scan.close() call. Hence based on the current state of the scan these two callbacks are set on the current RpcCallContext and once the cellblocks are created the Rpcserver executes this callback.

Based on the condition of the scan ie. In case of scan.next() call the scan is not fully over and hence we set the SessionCloseableCallBack but in case of the scan getting over or a closeScanner() call we set the current RegionScanner itself.

The SessionCloseableCallBack – this will try to close all the blocks used in the current scan.next() call except the last block.

Just after creating the cellblock that is created from the cellScanner at the RpcServer layer, the run() method of the callback is executed. Based on the type of the callback the corresponding run method is executed.

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

HRegion.getScanner(scan, boolean cellBlockSupported).

The Boolean would indicate if the client has cellblock support so that the results can be set in the payload.

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

We will see in upcoming sections why we need this Boolean **cellBlockSupported**.

Gets

The gets basically work on the HRegion. We try to use the callback mechanism for gets also. To do this, we have moved the logic of gets() in HRegion to RsRpcServices – where a scanner is created using getScanner() and using this scanner – we do the scanner.next() and retrieve the result.

To introduce the callback, just after creating the scanner – we set this scanner (a type of RpcCallback) to the RpcCallContext using the setCallBack(RpcCallBack callback).

When the cellblock is created from the cellScanner in the RpcServer layer, the Closeable's close() is invoked from the RpcServer ensuring that the scanner is closed and the blocks are returned back. In this case there is only one case of calling the final close() as gets do not have states.

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.

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.

This is where the above discussed SharedMemoryCell interface comes into use. Since the SharedMemoryCell type of cells are created for blocks coming out of SHARED_MEM type those cells can be cloned to create a copy of the cells so that we can get rid of the shared memory references. So once the copy is done we can return the blocks as and when the read gets completed.

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.

Note that the getScanner() API introduced newly for scans would explicity set the state of the RegionScanner saying cellBlockSupported as true. But in case of non-java client case since we use the old API getScanner() this cellBlockSupported is going to be false and hence internally the results are copied to ensure that we get rid of the references to the SHARED_MEMORY area.

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 for every next() call instead of holding those blocks till the scanner.close is called, we are adding a new interface with the following API.

We introduce an interface BatchSessionClosableScanner with an API batchSessionClose().

The RegionScanner, KeyValueScanner and HFileScanner implements this interface.

This interface provides a mechanism for the scanners to do a close/cleanup in between every next() call.

The RpcCallBack that is created in the RsRpcServices scan() (RegionScannerSessionCloseableCallBack) tries to invoke this batchSessionClose() so that after every next() call we are able to return the blocks in the current next() except the current one.

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.