Bucket Cache:A solution about CMS, Heap Fragment and Big Cache on HBASE

zjusch@163.com

Before introducing BucketCache，first do a simple introduction and analysis for **cache mechanism** in hbase:

a) Block is cached when reading to increase performance;

b) Block type: DataBlock(64K as default), BloomBlock (128K as default), IndexBlock(128K as default);

c) For a random read, we read BloomBlock first, then IndexBlock, and DataBlock at last. If there are two store files in one region, we should read BloomBlock twice + IndexBlock once + DataBlock once at least;

d) BloomBlock and IndexBlock could be classified as MetaBlock, and hit ratio of MetaBlock is nearly always 100% in our online clusters;

e) Read performance is mostly depend on hit ratio of block cache, so DataBlockEncoding could increase read performance by encoding the KeyValue in the memory;

f) We use interface BlockCache to manage the cached blocks, and its implementation includes following three classes：

1. LruBlockCache. the default block cache, using a hash map to manage the mapping from block key to block data, using LRU algorithm to evict block, specifying the capacity when initialization, starting eviction when the use ratio reaches 85% as default, and be freed to 75%.

Advantage: Using hash map supported by JVM, The management of cached blocks is easy and reliable. JVM would help you reclaim the memory of evicted blocks.

Disadvantage:

- For each cached block object, it will be moved from Eden Space to Tenured Space by Young Generation, and reclaim it by Old Generation at last if evicted. We often use CMS in Old Generation, but it will cause **heap fragment** problem after a long time run
- Because of heap fragment, we could see "Promotion failed" when doing CMS, causing a Full GC. In our clusters, Frequency of this type Full GC could be one day, or one week, or one month. In order to decrease the effect of online service, we triggered he Full GC manually as the experience in a off-peak time
- If caching block is faster than evicting block, server may be abort because of OOM

2. SlabCache, another block cache, using off-heap, intending to reduce heap fragment, it is experimental. We studied and tested it at first, the result was bad and considered it is not available to be used in production system. Description of its principle: SlabCache is composed of multiple SingleSizeCaches (SingleSizeCache is a slab allocated cache that caches blocks up to a single size, it contains a list of ByteBuffer, these byte buffers have the same capacity, we copy the data to the exactly one byte buffer when caching block, the byte buffer could be reused after evicting the

block) , When caching block, it first finds what size SingleSizeCache it should fit in, if the block doesn't fit in any, it will return without doing anything. For the default block size 64K, it will create two SingleSizeCaches with the size 64K*1.1 and 64K*2.1 by default. Because of the single size limit, we should use it with LruBlockCache, called DoubleBlockCache. For the DoubleBlockCache, we cache the block both in LruBlockCache and SlabCache, and we get the block from LruBlockCache first, then SlabCache. However, if only hit in SlabCache, block will be cached again in LruBlockCache.

Advantage: The idea of allocating fixed byte buffers when starting server, read/overwrite these byte buffers for getting/caching block

Disadvantage:

- Memory copy when getting/caching all the blocks
- Because of the limit of single size design, memory usage ratio is low, especially for various block size after using Data-Block-Encoding
- The design of DoubleBlockCache is not reasonable, it causes all the blocks will go to the LruBlockCache, the result is CMS or Heap fragment wouldn't get any better

3. **BucketCache,** it could be seen as a improvement with the idea of Slab Cache and a function expansion, its target is a solution for disadvantage of LruBlockCache and supporting big cache for high read performance.

3.1 What's big cache?

The storage of cached blocks is no longer just on the memory, we could store them on the high speed disk, like Fusion-IO,SSD, we could call it as secondary cache

3.2 What's the bucket?

We divided the cache space into lots of buckets, and each bucket has a size tag and a unique sequence number, block will be put in the most suitable bucket according to the size (It is similar to the SingleSizeCache)

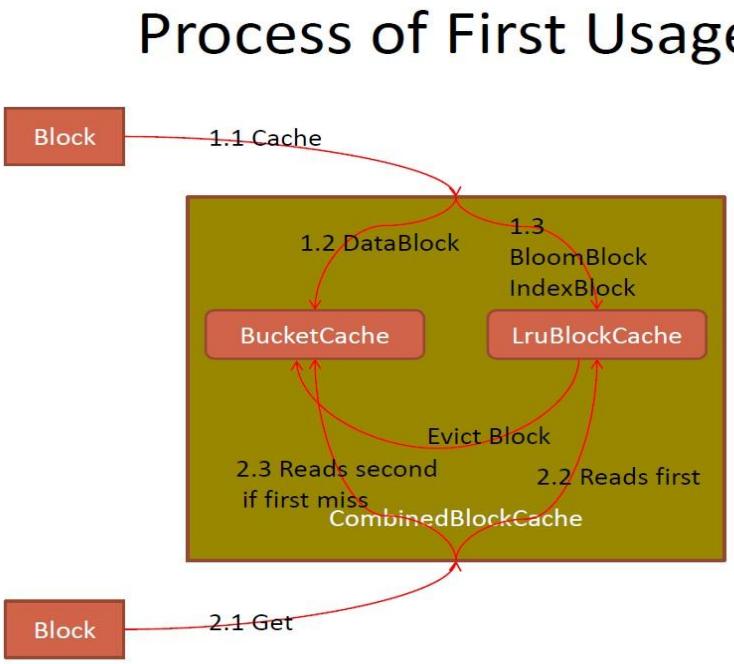
3.3 How to greatly decrease CMS and heap fragment?

Block is stored in a bucket, and the bucket's physical storage is fixed. It means we create the buffers as the configured capacity when server starting, so these buffers will always in the Tenured Space. Getting/Caching block is just read/overwrite these buffers, and it won't cause CMS and heap fragment again.

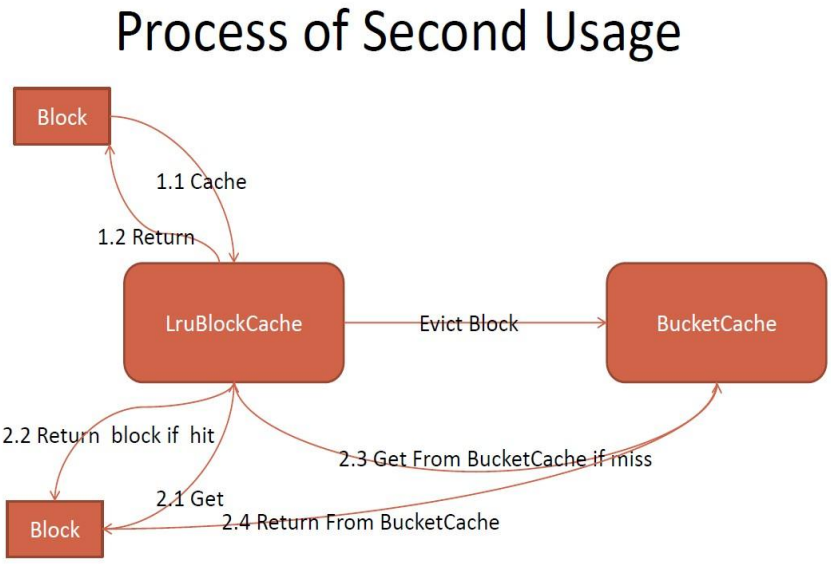
3.4 How does it work?

As the above description, it could be used in two ways:

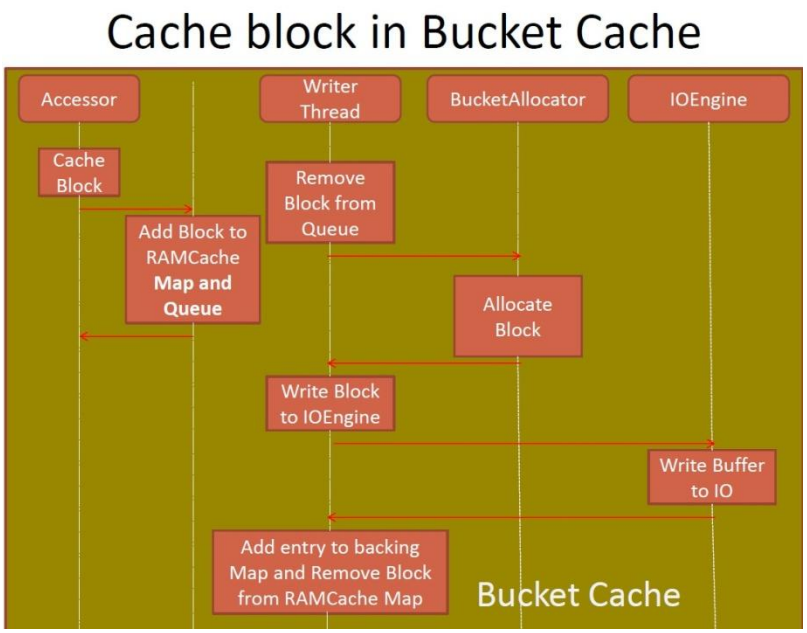
3.4.1 Use it as the mainly block cache in memory, combined with LruBlockCache. Why use LruBlockCache again? As the above (**b), c) d)**) mentioned , Frequency of accessing MetaBlock is quite high and its hit ratio is nearly always 100%, so we put them in LruBlockCache to get good performance.

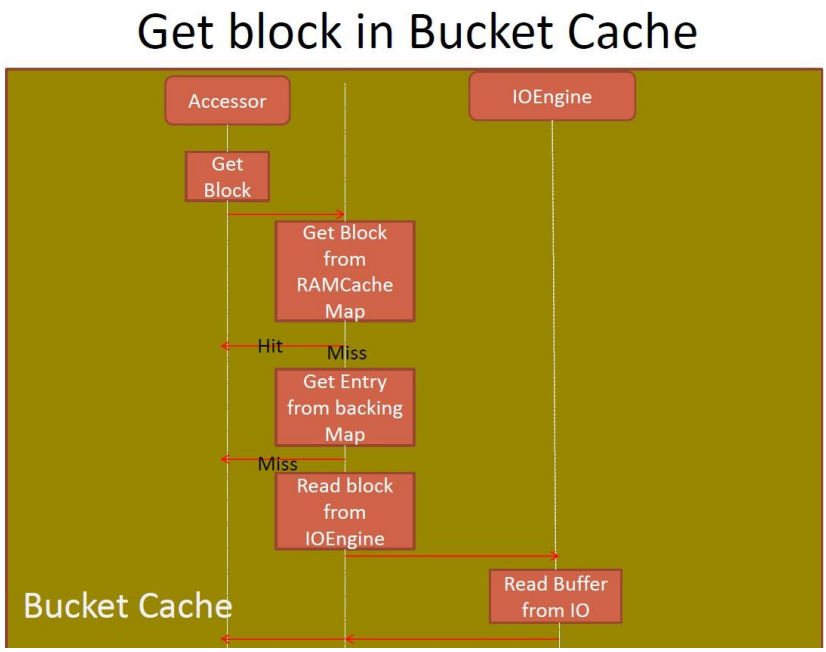


3.4.2 Use it as a secondary cache, storing the block data on the high speed disk like Fusion-IO



3.5 The process of Caching/Getting Block in Bucket Cache?



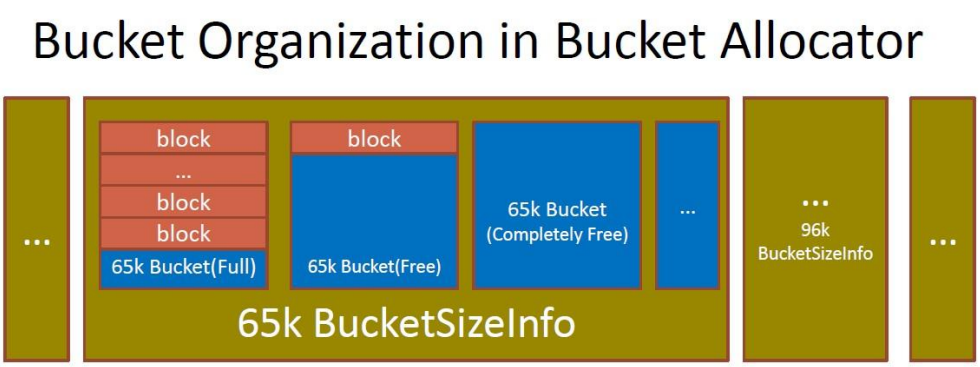


A simple description:

When caching block, we put the block in the RAMMap and the queue. Then Writer Thread will asynchronously remove the block from the queue and write it to the IOEngine (Memory or High speed disk), and record the metadata of this block (e.g. block key, length, offset) in the backingMap;

When getting block, we first read from RAMMap, if miss, then read metadata from backingMap and read data from IOEngine.

3.6 How does it allocate block in the IOEnigne(physical storage)?



1.Each bucket has a fixed capacity, 2MB as default;
2.Each bucket is specified a size and caches blocks up to this size
3.For completely free bucket, its size could be re-specified
4. Bucket allocator just allocate/free blocks logically, physical block data is stored on the IO engine

We divide the physical space into lots of buckets with same capacity, each bucket has a unique sequence number and a size tag, so we could get the block's offset in physical space through the sequence number and internal offset in bucket, also from sequence number and internal offset in bucket to offset in physical space. Based on the above, we could use the Bucket Allocator to allocate the offset in physical space through the calculation on the current buckets state

The above picture is a description of bucket organization in Bucket Allocator:

3.6.1 Each bucket has a size tag, we initialize the kinds of size tag when start and won't add new tag or delete exist tag in current. By default, there are (8+1)K, (16+1)K, (32+1)K, (40+1)K, (56+1)K, (64+1)K, (96+1)K,…, (512+1)K

3.6.2 The buckets with same size tag are managed by one BucketSizeInfo

3.6.3 The size tag of bucket could be changed to another tag dynamically if and only if the bucket is completely free. For example, there are lots of blocks with size about 64K, after the 64K size buckets are all full, other size buckets which are completely free could be changed to 64K size buckets. However, we should ensure that there is one bucket at least for each size tag

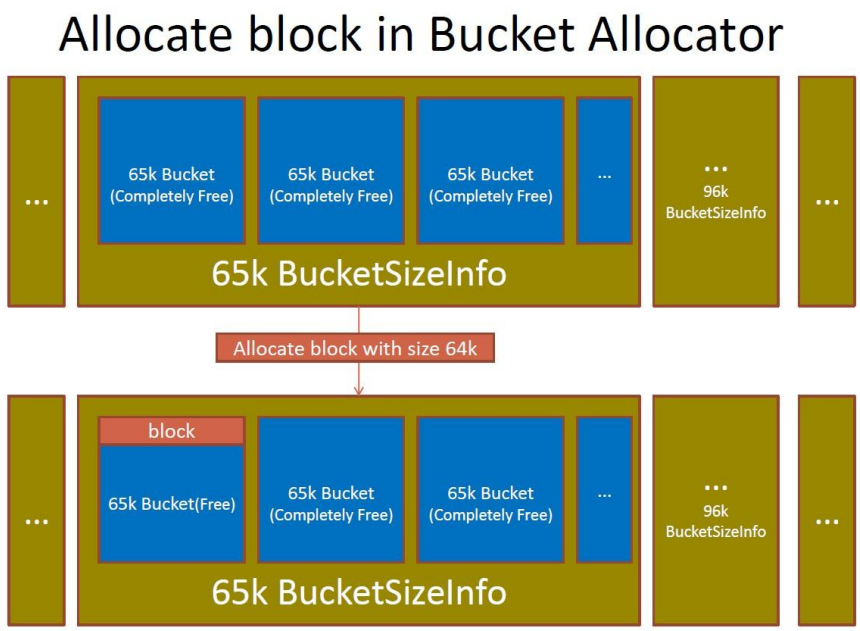
3.6.4 If one block's size is bigger than the biggest bucket's size (513K as default), this block couldn't store in the Bucket Cache

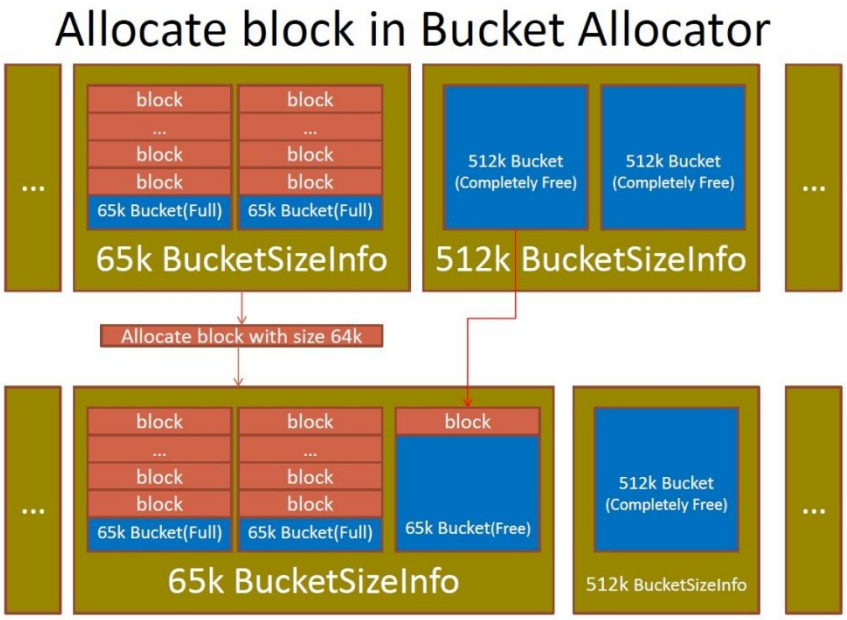
3.6.5 We will free the space if the usage ratio reaches 95% or one size block couldn't be allocated. When freeing the space, we use the LRU algorithm and ensure there must be some blocks are evicted

3.6.6 If all blocks are one size at start, after some time, all blocks are another size, What about? Whether we will call freeing the space frequently?

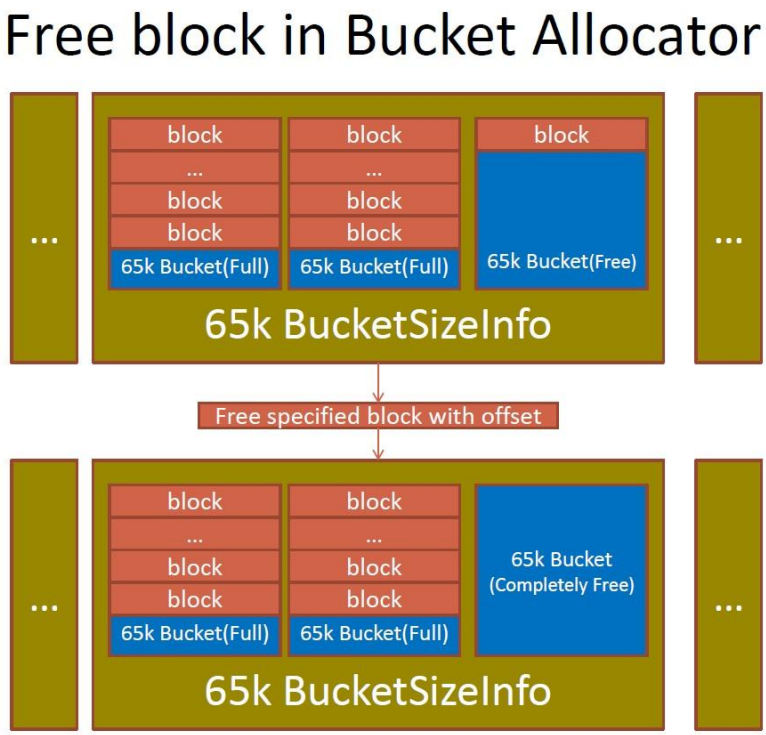
Yes, it may so. But freeing the space is asynchronous, after some time, the buckets' size will be shifted to these blocks' size and be stable at last

3.6.7 Process of allocating block in BucketAllocator





3.6.8 Process of freeing block in BucketAllocator



3.7 The test results of first usage (Use "heap" IOEngine)

Test Results of First Usage

- Test Environment
  - HBase-0.94,5 client, 5 regionserver, 50 threads per client, DataBlock hit ratio about 75%
- Case 1(Read Only):

	QPS	RT	Load	YGC	YGCT	CMS	CMST
Before	11328	4.41ms	9.92	4870	237.9s	268	7.9s
After	13688	3.65ms	11.59	7131	147.6s	0	0

- Test Case 2(Read + Write)

	QPS	WPS	Load	YGC	YGCT	CMS	CMST
Before	11333	3133	10.28	6609	355.4s	437	15.8s
After	12185	3813	10.58	7162	257.5s	10	0.7

3.8 The test results of second usage (Use "file" IOEngine)

Test Results of Second Usage

- Test Environment
  - Using fusionIO to store cached data

- Case (Read Only):

	QPS	RT	Data Block Hit Ratio	Bucket Cache Hit Ratio	IOPS of Read Bucket Cache (Fusion-IO)	RT of Read Bucket Cache (Fusion-IO)	RT of Read Datanode
Before	1934	42.21ms	11.12%	0	0	0	44.98ms
After	14420	2.2ms	22.5%	87.3%	11598	1.5ms	2.58ms

3.9 More details

https://issues.apache.org/jira/browse/HBASE-7404