Item 89
git_comments:

1. **comment:** * * This package implements a memory manager for off-heap data based on pools and arenas. * It is derived from the netty memory manager with the following changes: * - Classes have been redefined to store data little-endian. (Append "L" to class names) * - A "trim" method has been added to reduce a buffer's maxCapacity and free the extra memory. * - allocate(min, max) * * This manager is based on the following memory abstractions: * - a "chunk" is a piece of memory allocated from the operating system, * - a "page" is is a piece of memory allocated from a chunk, and * - an "element" is a piece of memory allocated from a page. * * This memory manager classifies the memory request according to the size of requested memory. * - "subpage" memory manager which breaks a page into equal sized elements. * It uses a bitmap to indicate which ones are free. (why not linked list?) * - "normal" memory manager which breaks a chunk into runs of pages. * Management is done using the "buddy system", so each run is a power-of-2 pages. * - "huge" memory manager which allocates memory larger than a chunk. * It goes straight to the operating system. * * The PoolArena utilizes these three memory managers, trying to minimize fragmentation * in the big buffers while keeping good performance for the small buffers. * * The three submanagers are: * subpage allocations. A page is divided into equal sized elements with a bitmask indicating * which elements are free. The page is placed into a pool (linked list) * according to the size of the elements. * Normal allocations. A chunk is divided into multiple pages according to the buddy system. * Chunks are kept in a list partially ordered to minimize fragmentation. * Huge allocations. Memory larger than a chunk is allocated directly from the OS. * * To avoid contention, there are multiple "arenas" rather than a single monolithic memory manager. * Threads are assigned to an arena on a round-robin basis, and a buffer is always returned * to the arena of its origin. * * * Note for "normal" allocations: Rather than keeping one fully ordered list of chunks, * it keeps several lists, and always * searches the first list, followed by the next list. Thus it achieves an approximate ordering without * having to do a full sort. * * * This PoolArena allocator is configured according to Java environment variables. * io.netty.allocator.numDirectArenas - the number of arenas to divide among threads. * io.netty.allocator.pageSize - The size of each page, must be power of 2. * io.netty.allocator.maxOrder - Specifies chunkSize as pageSize * 2^maxOrder. * * It may not make sense to use this allocator for small allocations. * Each buffer which is allocated uses a Java class which must also be allocated. * Thus we may not save on Java memory management for small buffers.
   label: code-design
2. * * Licensed to the Apache Software Foundation (ASF) under one * or more contributor license agreements. See the NOTICE file * distributed with this work for additional information * regarding copyright ownership. The ASF licenses this file * to you under the Apache License, Version 2.0 (the * "License"); you may not use this file except in compliance * with the License. You may obtain a copy of the License at * * http://www.apache.org/licenses/LICENSE-2.0 * * Unless required by applicable law or agreed to in writing, software * distributed under the License is distributed on an "AS IS" BASIS, * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. * See the License for the specific language governing permissions and * limitations under the License.
3. Trim the single page to a tiny size.
4. * Trim a large block to a smaller block.
5. * Allocate and free the largest small allocation
6. * Test the allocation and free of a "normal" allocation
7. Release the buffer. Verify the element is returned to pool and page still allocated.
8. Allocate a large block and trim to a single page
9. default
10. Allocate a single page
11. Allocate the buffer and verify we have the expected number of pages
12. **comment:** * Grow a tiny buffer to a normal one
    label: code-design
13. if the line contains pattern, then success.
14. * Trim a page down to a tiny buffer.
15. * Allocate and free the largest tiny allocation
16. * Allocate and free the tiniest tiny allocation
17. * * Licensed to the Apache Software Foundation (ASF) under one * or more contributor license agreements. See the NOTICE file * distributed with this work for additional information * regarding copyright ownership. The ASF licenses this file * to you under the Apache License, Version 2.0 (the *

"License"); you may not use this file except in compliance * with the License. You may obtain a copy of the License at * * http://www.apache.org/licenses/LICENSE-2.0 * * Unless required by applicable law or agreed to in writing, software * distributed under the License is distributed on an "AS IS" BASIS, * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. * See the License for the specific language governing permissions and * limitations under the License.
18. Get our current state as a string
19. Do for each line in the string
20. **comment:** * * Unit test the memory allocator and trim() function. * The results are confirmed by examining the state of the memory allocator. * In this test, we are working with a single chunk and a * small set of subpage allocations. * * Chunk status can be verified by matching with * "Chunk ... <bytes consumed>/". * The subpage allocators are verified by matching with * "<nr allocated>/ ... elemSize: <size>" * * * A cleaner approach would create new methods to query * - how many pages (total) have been allocated, and * - how many elements of a particular size have been allocated.
    label: code-design
21. We didn't find a matching line, so fail the test
22. * Allocate and free the smallest small allocation
23. * Allocate and free a larger run of pages
24. * Test the allocation and free of a "subpage" allocation
25. * Allocate and free a single page
26. Resize the tiny block to two pages
27. * An allocator with some stuff added to aid testing
28. Allocate a tiny buffer
29. * * Verify our current state matches the pattern. * * Note: Uses the existing "toString()" method and extracts information * by matching a pattern to one of the output lines. *
30. * A convenience method to do all the tests.
31. * * Allocate memory to a buffer container. * @param cache TODO: not sure * @param buf - A buffer which will contain the allocated memory * @param minRequested - The smallest amount of memory. * @param maxRequested - The maximum memory to allocate if convenient.
32. OTHERWISE: Normal allocation of pages from a chunk.
33. * Create an arbitrary size chunk of memory which is not part of the pool.
34. * Create a chunkSize chunk of memory which will be part of the pool
35. * HeapArena is an arena which allocates memory from the Java Heap
36. * Copy memory from one allocation to another.
37. If the list was empty, allocate a new page, allocate an item, and add page to the list. This is awkward. "allocateNormal" does subpage allocation internally, and it really shouldn't know anything at all about subpages. Instead, we should allocate a complete page and add it to the desired list ourselves.
38. If the buffer can be allocated from the skip list, then allocate it.
39. CASE: minCapacity is a subpage
40. minCapacity <= pageSize/2
41. * * A PoolArenaL is a single arena for managing memory. * It represents a composite memory manager which consists of other * memory allocators, where each of the allocators * works best for different allocation sizes. * * More specifically, a PoolArenaL includes the following allocators: * tiny: (16-256) - multiple lists of pages of items, where each list has items of a fixed size 16*N. * small: (512-pageSize/2) - multiple lists of pages of items, where each list has items of a fixed size 2^N. * normal: (pageSize-chunkSize) - a skiplist of chunks ordered to minimize fragmentation, where * each chunk is divided into pages using the buddy system. * huge: (>chunkSize) - memory is allocated directly from the operating system. * * Note: In a multi-threaded environment, the "parent" creates multiple "arenas", * distributing threads among them to minimize contention. * * @param <T>
42. Create a buffer header, limiting growth to minimize copying
43. Create the tiny pools, ranging in size from 16 to 256 in steps of 16. Create the small pools, ranging in size from 512 to pagesize/2 as powers of 2. Create a skip list of chunks consisting of separate lists connected together. Chunks migrate between the lists depending on how much free space they have.
44. * * Find which list holds subpage buffers of the given size. * @param elemSize * @return
45. CASE: HUGE allocation.
46. CASE: buffer has grown. Copy data from old buffer to new.
47. * * Bump the requested size up to the size which will actually be allocated * @param reqCapacity - the requested size * @return the large, normalized size
48. * * Allocate a huge (>chunksize) buffer. * @param buf * @param reqCapacity

49. * * Allocate a "normal" (page .. chunk) sized buffer from a chunk in the skiplist. * @param buf - the buffer header which will receive the memory. * @param minRequested - the minimum requested capacity in bytes. * @param maxRequested - the maximum requested capacity.
50. * Create an array (uninitialized) of subpage lists.
51. * * Create a new arena. * @param parent - The global memory manager (where we go to allocate a completely new buffer). * @param pageSize - The minimum size of memory for using the "normal" allocation (buddy system). * @param maxOrder - The size of a "chunk", where chunkSize = pageSize * 2^maxOrder. * @param pageShifts - pageSize expressed as a power of 2. (redundant?) * @param chunkSize - chunkSize in bytes. (redundant?)
52. * * Allocate a buffer from the current arena. * Unlike netty.io buffers, this buffer can grow without bounds, * but it will throw an exception if growth involves copying a page * or more of data. Instead of being an upper bounds sanity check, * the "max" capacity is used to opportunistically allocate extra memory. * Later, the capacity can be reduced very efficiently. * To avoid excessive copying, a buffer cannot grow if it must copy * more than a single page of data. * @param cache TODO: not sure * @param minRequested The smallest capacity buffer we want * @param maxRequested If convenient, allocate up to this capacity * @return A buffer with capacity between min and max capacity
53. * DirectArena is an arena which allocates memory from off-heap (direct allocation).
54. **comment:** CASE: HUGE allocation, don't change it. CASE: normal or small allocation, then round up to 2^n OTHERWISE: tiny allocations. Round up to the next multiple of 16
    label: code-design
55. CASE: buffer has shrunk. Copy data, but also reset the reader/writer positions. move to buffer has shrunk case? If requested, release the old memory.
56. else "small", pick list based on power of 2. Whether tiny or small, allocate an item from the first page in the corresponding list
57. Add the new chunk to the skip list at the "newly initialized" location.
58. if "tiny", pick list based on multiple of 16 minCapacity < 512
59. **comment:** Sanity check to not grow beyond the maxCapacity. This check may not be relevant any more since we have reinterpreted maxCapacity. Do nothing if capacity doesn't actually change. Cache some local values to make them more accessible. Allocate new memory for the buffer
    label: code-design
60. * Create a new buffer, but don't allocate memory yet.
61. * Display the contents of a PoolArena
62. * Release or recycle a chunk of memory
63. Allocate a buffer from the chunk.
64. * * Change the size of a buffer by allocating new memory and copying the old data to it. * @param buf - the buffer containing the memory * @param newCapacity - the desired capacity * @param freeOldMemory - whether to release the old memory or not.
65. Create a new chunk.
66. * Initialize a subpage list
67. **comment:** This code should be reorganized. case: <= maxTiny: allocateTiny{select which tiny list, allocate subpage from list.} case: <= maxSmall: allocateSmall{select which small list, allocate subpage from list.} case: <= maxNormal: allocateNormal otherwise: allocateHuge where maxTiny=256, maxSmall=pageSize/2, maxNormal=chunkSize.
    label: code-design
68. * * Free a piece of memory. * @param chunk * @param handle
69. * * Allocate a page to be used for subpage buffers, knowing the subtree is UNUSED * @param normCapacity * @param curIdx * @param val * @return
70. Starting at current node, follow left hand children, until reaching node of desired size. Note that runLength and memoryMapIdx move in unison.
71. We can't trim if the result will become a subpage
72. Initialize buffer with as large as possible capacity within the requested range (assert: we know the buffer is >= minCapacity)
73. * * Allocates a buffer with size between minRequested and maxRequested * @param minRequested * @param maxRequested * @return
74. * * Allocates a buffer of the given size from current chunk * @param capacity - requested capacity of the buffer * @return - handle to the buffer, -1 if failed
75. **comment:** Search through the subtrees until finding an unused node s.t. runlength >= min
    label: code-design
76. We are about to allocate from one of our children, so we become BRANCH

77. If the buffer was a HUGE allocation, then we leave it alone
78. If the buffer is growing, then we aren't really trimming.
79. Current node is a parent of the desired node. It now becomes a "BRANCH".
80. If buffer was allocated from a subpage, then free it.
81. **comment:** Mark the node as "unused"
    label: code-design
82. **comment:** If at top of tree, done If the buddy is allocated, we can stop since no more merging can occur move to current node's parent, effectively merging with UNUSED buddy
    label: code-design
83. **comment:** OTHERWISE: allocating subpage buffer. Special case: maxCapacity is normCapacity. Note: this case should be moved to PoolArena.
    label: code-design
84. * * Allocate a run of pages where the run size is within minCapacity thru maxCapacity. * @param minRequested - the minimum size of the buffer * @param maxRequested - the maximum size of the buffer to be allocated if convenient * @param node - the subtree of this chunk to search * @return handle to the allocated memory * * More specifically, this routine finds an unused node in the binary tree, * s.t. the node is big enough to contain minCapacity, and is not * bigger than the size to contain maxCapacity. * * A node is the correct size to contain x bytes, if * runlength(node) == roundup-power-of-2(x) * equivalently, runlength(node) >= x and runlength(node.child) < x * equivalently, runlength(node) >= x and runlength(node)/2 < x * * Similarly, a node is the correct size to contain min...max bytes, if * runlength(node) >= roundup-power-of-2(min) && runlength(node) <= roundup-power-of-2(max) * or equivalently, runlength(node) >= min && runlength(node)/2 < max
85. * * Allocate a new page for splitting into subpage items. * Note: this routine doesn't belong here. Instead, we should have a "subpage" allocator * which is invoked instead of us and is responsible for the entire subpage allocation sizes. * @param normCapacity - the actual size of the buffer we will allocate * @param curIdxn - the node where our search stargs * @param val - contents of the current node * @return a handle to the allocated buffer.
86. **comment:** At this point, we have an unused node s.t. runlength >= min. In other words, it is larger than the minimum, but it may also be larger than the maximum.
    label: code-design
87. **comment:** if we failed to find an unused node which is big enough, then failure.
    label: code-design
88. **comment:** We are at an unused node which satisfies both conditions. Allocate it.
    label: code-design
89. Continue descending subtree looking for a node s.t. runlength/2 < max
90. **comment:** * * A Chunk is a large, fixed size piece of memory allocated from the operating system. * This PoolChunk allocator divides a chunk into a run of pages using the buddy system. * The actual allocation will be the size requested, rounded up to the next * power-of-2. * * This allocator does "normal" allocations, where the requested size varies * from page size to chunk size. * * The allocator is based on buddy system. It views a chunk as a binary tree with * 1 run of chunksize, or * 2 runs of chunksize/2, or * 4 runs of chunksize/4, or * ... * 2^maxOrder runs of pagesize * * Each node in the binary tree is labeled: * - unused. The node and all its children are unallocated * - allocated. The node (and all its children) are allocated * as a single request. The children remain marked "unused". * - branch. At least one descendent is allocated. * * The binary tree is represented in the "memoryMap", * which saves the node status in a simple array without using links. * The array indices indicate the position within the tree as follows: * 0 - unused * 1 - root * 2,3 - children of 1 * 4,5 6,7 - children of 2 and 3 * 8,9 10,11 12,13 14,15 - next level of children. * * Thus, i/2 points to the parent of i, * i*2 points to left child, i*2+1 points to right child. * * Note the current code also deals with smaller subpage allocations. * The overall memory manager only comes here when it wants a new page, * not every time it allocates a subpage piece of memory.
    label: code-design
91. * Creates an uninitialized array of subpage lists
92. If not found, search the other subtree (tail recursion)
93. Otherwise initialize buffer as a subpage allocation.
94. return new handle to the reduced size buffer.
95. If this is a normal allocation
96. Verify the memory is allocated
97. Search one random subtree (recursively)
98. * returns the percentage of the chunk which has been allocated

99. * Create a new "chunk" of memory to be used in the given arena. * * @param arena - the arena this chunk belongs to * @param memory * @param pageSize - the size of a page (known to arena, why here?) * @param maxOrder - how many pages to a chunk (2^^maxOrder pages) * @param pageShifts - page size as number of shifts (2^^pageShifts) * @param chunkSize - the size of a chunk (pagesize*2^^maxOrder(known to arena)
100. Pick one of the children and continue descending its subtree.
101. We are now at the desired node. Mark it allocated.
102. * * Initialize a buffer given a handle that was allocated from this chunk. * @param buf The buffer to be initialized. * @param handle The handle representing memory allocated from this chunk. * @param minRequested The minimum requested capacity. * @param maxRequested The maximum requested capacity.
103. If the buffer was a subpage, then we also leave it alone.
104. CASE: allocating runs of pages, make use of maxCapacity since we can trim it later
105. * * Free up memory to reduce the size of a run of pages. * The resulting run starts on the same page, and * the trailing pages are returned to the memory manager. * Trim is intended to be an efficient way to reduce the size of a buffer. * No new memory is allocated, nor is any data copied. * @param handle - which run of pages was allocated from the current chunk * @param smallerSize - the new desired size the new run of pages * @return a new handle to the smaller run of pages, or -1 if can't trim.
106. * * Return a buffer back to the memory pool. * @param handle
107. **comment:** Right hand child is now an unused buddy. Mark it "UNUSED". (done - should already be marked "UNUSED")
    label: code-design
108. Otherwise, it should have been allocated from the chunk Update the nr of bytes free start at current node and work up the tree
109. ... then add the memory to the buffer container
110. * * Create a new, empty pool of chunks. * @param arena - the bigger arena this pool belongs to * @param nextList - the next list to consider (in the same pool) * @param minUsage - contains chunks with the specified usage (min ... max) * @param maxUsage
111. Release buffer back to the original chunk
112. If usage changed, then move to different list
113. If usage has change, then add to the neighboring list instead
114. return new handle for the smaller buffer
115. Trim the buffer, possibly getting a new handle.
116. If usage changed, then move to next list
117. * * Allocate a buffer with the requested size * @param buf - the container to hold the buffer * @param minRequested - the minumum requested capacity * @param maxRequested - the maximum capacity * @return
118. * * Shrink the buffer down to the specified size, freeing up unused memory. * @param chunk - chunk the buffer resides in * @param handle - the handle to the buffer * @param size - the new desired "size" * @return a new handle to the smaller buffer
119. * * Remove a chunk from the current linked list of chunks * @param cur
120. Move the chunk to a different list if usage changed significantly
121. * * Release a buffer back to the original chunk. * @param chunk * @param handle
122. If list is empty, then allocation fails Do for each chunk in the list
123. * * Add a chunk to the current chunklist * @param chunk
124. If we successfully allocated from the chunk ...
125. **comment:** } else if (usage < minUsage) { // TODO: Could this result in a recursive loop? prevList.add(chunk); return; Add chunk to linked list.
    label: requirement
126. * * A list of chunks with similar "usage". If a chunk is added to the wrong list, * it will migrate to the next list which hopefully will be the correct one. * * @param <T>
127. if our page is now empty (numAvail == maxNumElems), ... Do not remove if this page is the only one left in the pool.
128. * * Search a bitmap to find the next available bit. * @return index to a free bit
129. mark the corresponding bit as "free" If we were full, add our page to the pool of subpages If we are not empty, then keep us among the pool of subpages
130. **comment:** * * Override the abstract allocator. Normally, the abstract allocator * defaults the second parameter to MAXINT as a "sanity check", but we * have reinterpreted the second parameter as "max requested".

label: code-design
131. * * Allocate a buffer from the current thread's direct arena.
132. **comment:** * * Exception thrown after resizing a buffer results in excessive copying. * The buffer has been properly resized, so It is possible to ignore the exception and continue.
label: code-design
133. Check for the easy resizing cases, and return if successfully resized.
134. * * Initialize a new buffer for "normal" allocations. * @param chunk - which chunk the buffer came from * @param handle - which pages within the chunk * @param offset - byte offset to the first page * @param length - the requested length * @param maxLength - the max limit for resizing.
135. * * A ByteBuf optimized to work with little endian direct buffers * and the netty pool allocator. * The buffer can be of any size - tiny, small, normal, or huge. * The class contains all information needed to free the memory * (which chunk, which pages, which elements). * * @param <T>
136. Trim down the size of the current buffer, if able to.
137. **comment:** TODO: Is this correct?
label: code-design
138. Reallocate the data.
139. * * Free the memory and recycle the header.
140. * * Change the size of an allocated buffer, reallocating if appropriate. * @param newCapacity * @return
141. **comment:** * * A buffer allocated from the netty pool allocator, optimized for little endian access. *
label: code-design
142. * * Allocate a new or reused buffer within provided range. Note that the buffer may technically be larger than the * requested size for rounding purposes. However, the buffers capacity will be set to the configured size. * * @param minSize The minimum size in bytes. * @param maxSize The maximum size in bytes. * @return A new ByteBuf.

git_commits:

1. **summary:** DRILL-336: Modified the netty direct memory manager to:
message: DRILL-336: Modified the netty direct memory manager to: 1) Efficiently reduce ("trim") the size of a memory buffer, releasing the extra memory back to the memory manager. 2) Opportunistically allocate larger buffers if it can be done efficiently. 3) Raise a warning exception if resizing a buffer copies more than 1 page of data. 4) Added Javadocs and some in-code comments. 5) Created a Junit test to verify basic functionality. The "trim" and allocation changes are primarily in PoolChunkL, which breaks a "chunk" into runs of pages using a buddy system. Take advantage of memory interface changes. Memory fixes
label: code-design

github_issues:

github_issues_comments:

github_pulls:

github_pulls_comments:

github_pulls_reviews:

jira_issues:

1. **summary:** Extend off heap memory manager to support growing vectors
description: Add the following interfaces to the bufferl memory manager: ptr = alloc(size, min, max) - allocate a block with given size, but with a total capacity between min+max. trim(ptr) - free up extra capacity, so current size is the capacity Rationale: It isn't always possible to anticipate the size of a vector. When creating a new vector, one strategy is to over-allocate the vector and then trim the size once the vector is complete. These routines allow us to implement this strategy.
label: code-design
2. **summary:** Extend off heap memory manager to support growing vectors
description: Add the following interfaces to the bufferl memory manager: ptr = alloc(size, min, max) - allocate a block with given size, but with a total capacity between min+max. trim(ptr) - free up extra capacity, so current size is the capacity Rationale: It isn't always possible to anticipate the size of a vector.

When creating a new vector, one strategy is to over-allocate the vector and then trim the size once the vector is complete. These routines allow us to implement this strategy.

3. **summary:** Extend off heap memory manager to support growing vectors
   description: Add the following interfaces to the bufferl memory manager: ptr = alloc(size, min, max) - allocate a block with given size, but with a total capacity between min+max. trim(ptr) - free up extra capacity, so current size is the capacity Rationale: It isn't always possible to anticipate the size of a vector. When creating a new vector, one strategy is to over-allocate the vector and then trim the size once the vector is complete. These routines allow us to implement this strategy.
   label: code-design
4. **summary:** Extend off heap memory manager to support growing vectors
   description: Add the following interfaces to the bufferl memory manager: ptr = alloc(size, min, max) - allocate a block with given size, but with a total capacity between min+max. trim(ptr) - free up extra capacity, so current size is the capacity Rationale: It isn't always possible to anticipate the size of a vector. When creating a new vector, one strategy is to over-allocate the vector and then trim the size once the vector is complete. These routines allow us to implement this strategy.

jira_issues_comments:

1. **body:** Here is a revised API for extending the memory manager. The new API makes use of the existing interfaces, although with a slightly different interpretation. ByteBuffer buf = allocator.newDirectBuffer(min, max); Allocate a direct buffer with capacity between min and max. Formerly, max was an upper bound on resizing the buffer. Now, max is an opportunistic value - give us that much if convenient. buf.capacity(newSize); As before, resize the buffer to the new size and preserve the old data. The new version will - efficiently "trim" a buffer to a smaller size for "normal" allocations. - throw a "TooMuchCopying" exception when resizing requires copying more than a page of data. - Correctly resize the buffer, even if the exception is thrown. The "TooMuchCopying" exception is more a warning than an error. It signals an inefficient use of the capacity(newSize) method. Whether the exception is thrown or not, the buffer will be resized correctly.
   label: code-design
2. merged in e80c32e