CellBlocksSegment in the context of MemStore

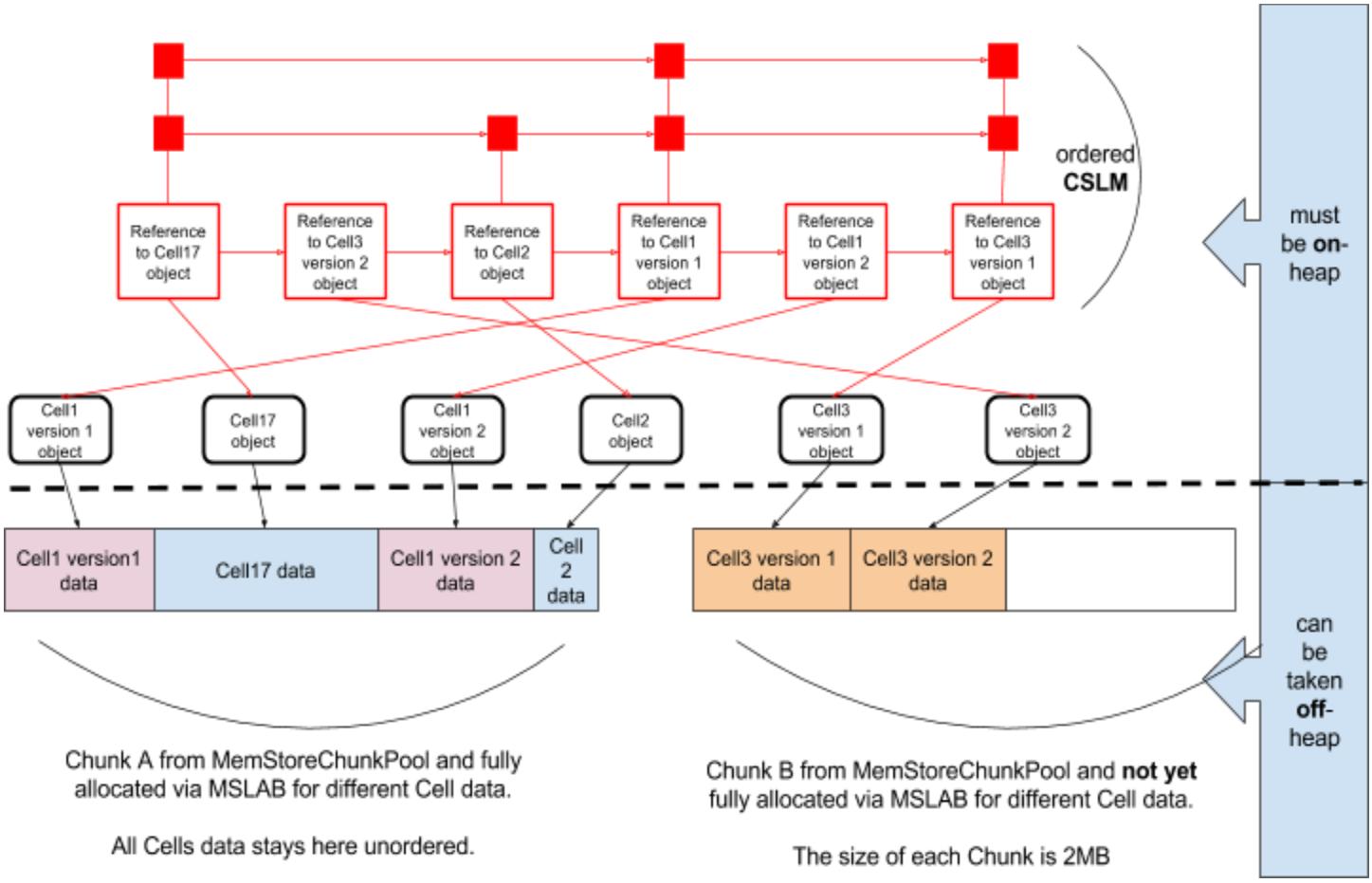
1. The flow of Cell from input to disk
2. What is CellBlocksSegment?
3. Why copies are needed in compacting process?
4. How CellBlocksSegment is transfered to HFile?
5. Why a sequence of bytes with index (to accelerate search), is not the same as CellBlocksSegment?

The flow of the Cell from input to disk

Getting Cells from Input

1. MemStoreChunkPool pre-allocates Chunks of memory in the size of 2MB (configurable)
2. When requested MSLAB allocates a space for a Cell as a range of bytes on the current chunk (taken from MemStoreChunkPool). If there is no MemStoreChunkPool then memory is allocated just directly from JVM Heap.
3. The input Cell is copied on the ByteRange (given by MSLAB). The *Cell internal data* is a variable-size Key and Value (variable-size = the size is different for each Cell) and different other attributes, everything is written on the given ByteRange. This may take big chunk of memory for example 1K.
4. *Cell object* includes offsets to Key, Value and other attributes of the Cell, which are de facto written on the memory allocated from MSLAB. The Cell object is not as big as the size of memory allocated for the actual data.
5. The *reference to the Cell* object is inserted into the ConcurrentSkipListMap. The reference is just a machine word and thus occupies less memory than the Cell Object.

Below please take a look on pictorial representation of what is explained above. This is MutableSegment.

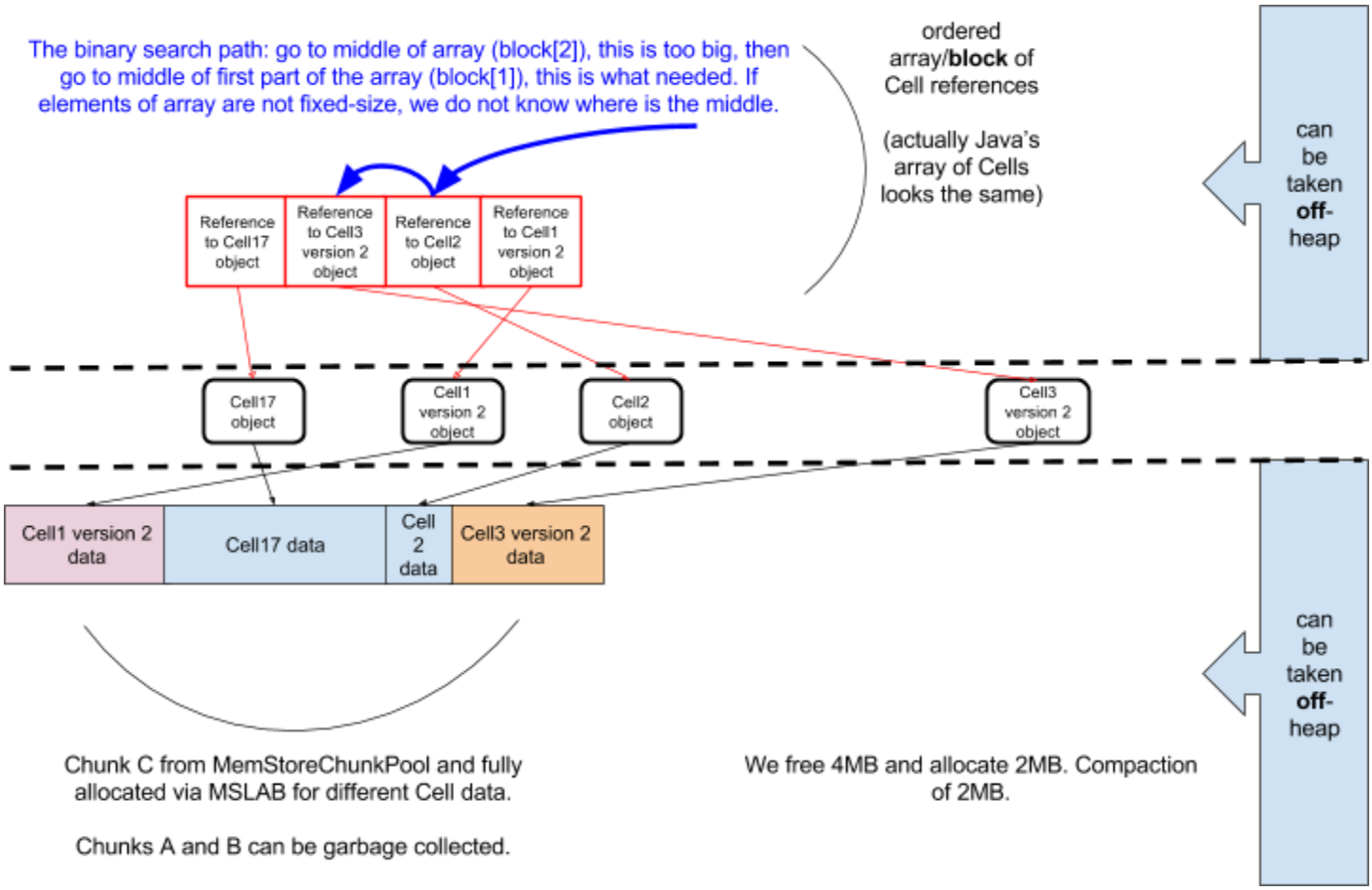


Compacting Cells in Memory

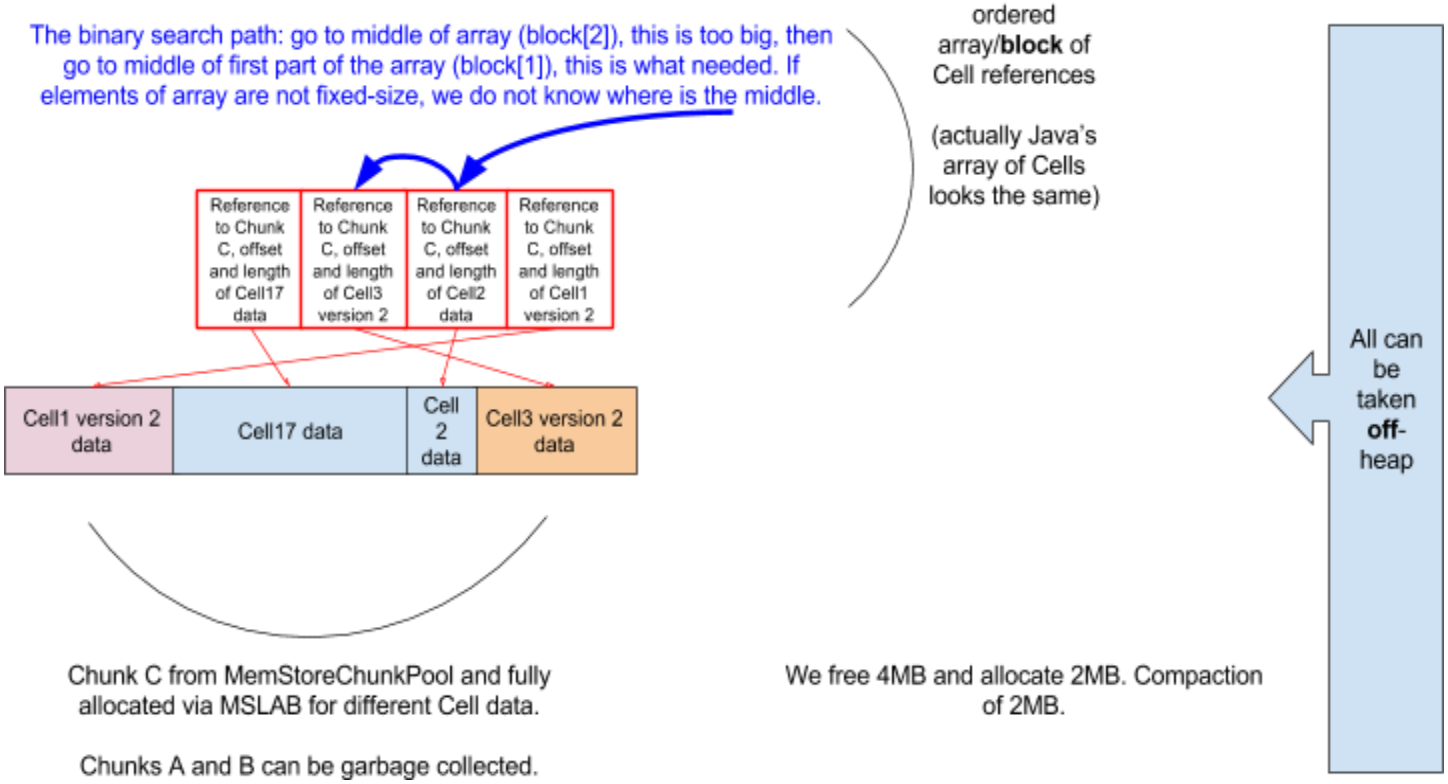
The picture remains the same until in-memory-flush. In the point of time of in-memory-flash nothing is copied the MutableSegment is adapted to be ImmutableSegmentAdapter and moves to be part of CompactionPipeline. When the compaction is actually performed one ImmutableSegmentAdapter (or more) is copied to CellBlocksSegment. The difference is in CellBlocks instead of CLSM. The process of compaction goes as following:

6. The SegmentScanner goes over all ImmutableSegmentAdapters and returns only relevant Cells (no expired versions).
7. The relevant Cell data is copied on the new ByteRange allocated from MSLAB. Pay attention that this copy is essential in order to compact, because we copy the relevant data out of the space where it is intermixed with obsolete data. After that obsolete data can be garbage-collected.
8. The references to the new Cells are now collected in arrays/blocks.
9. Each Cell reference takes the size of machine word. When we know the number of the Cell references in the CellBlock we can randomly jump back and forth among the Cells in O(1) time. This allows the binary search.

Below please take a look on pictorial representation of what is explained above. This is CellBlocksSegment made after compaction of the InmmutableSegmentAdapter above.



Another possibility of total flattening of the index to the Cell data located in the MSLAB Chunk



Flushing Cells to Disk

The compaction process can repeat until we must flush to disk. When CellBlocksSegment moves to be a snapshot the following happens (the process of turning a snapshot of memstore into a HFile according to StoreFlusher):

10. The SnapshotScanner goes over CellBlocksSegment and returns only relevant Cells (no expired versions). Lets say Cell c.
11. Writing data from scanner into sink: sink.append(c) that invokes StoreFile.append(c) that invokes HfileWriterImpl.append(c) that invokes HFileBlock.write(c)...

HFile Structure

Hereby the **simplified** understanding of the HFile structure. When single-level indexing is used, the HFileBlock is plain byte[] (data blocks). The HFileBlockIndex hold references to the byte[] of the HFileBlock. Those references includes the Keys (written as byte array) which have variable size.
When multiple-level indexing is used, we have many levels of references in HFileBlockIndex(es) and we get O(logN) search into the byte[] of the HFileBlock. But we use more memory and we need many read accesses till we get to the actual key on HFileBlock. Please take a look on the picture below.