HBASE-11425 tries to make sure data can be in off heap throughout the read path. This makes a need for Cells to be backed by byte buffers (Direct Byte Buffer - DBB) rather than byte arrays. The conversion of KeyValue to Cell in the entire read path is server side, was the 1st step. KeyValue wants the backing to be done by a single byte[].

Still, the Cell contains only APIs which deals with byte[]s. We have getRowArray() /getFamilyArray() etc. When the cell data is actually in a DBB, these APIs will need a data copy from DBB to temp byte[]s which makes it inefficient. Instead if there were means of getting the components as ByteBuffers itself, those could have been used. We can add these buffer based APIs to Cell but it will make the Public exposed Cell interface bit ugly. There is no buffer backing in client side at least. So for a user, these new APIs won’t be making much sense.

To overcome this we can go with a new interface approach. We will make an extension interface for Cell and keep the new APIs in this. This new interface can be kept InterfaceAudience Private only. We can call it as ByteBufferedCell.

Methods in new Interface

ByteBuffer getRowByteBuffer()

ByteBuffer getFamilyByteBuffer()

ByteBuffer getQualifierByteBuffer()

ByteBuffer getValueByteBuffer()

ByteBuffer getTagsByteBuffer()

Along with all getXXXArray() methods we have getXXXOffset() and getXXXLength() APIs so that one can refer to each component bytes range. For the new APIs ideally these APIs of offset and lengths are not required. On Java ByteBuffer one can set the position and limit accordingly such that the BB object itself wraps the offset and length info. But we are not going to take this approach because of performance concerns. In order to properly change the position and limit on the BB object, one has to duplicate the actual Cell backing BB object. This will cause a lot of short living Java objects creation (every time the getXXXByteBuffer() API is been called). Instead we plan to continue to with the offset and length APIs usage along with getXXXByteBuffer() API.

We can safely use the getXXXLength API along with getXXXArray() as well as getXXXByteBuffer(). The length is not going to change. But the offset is a tricky one. Offset of a cell component (row/cf/qual etc) in a BB is its start position in the Buffer. When the Cell is backed by DBB and one calls the getXXXArray() method, we will copy the corresponding component bytes to a temp byte[] (of length= component length) and here the offset will be always ‘0’. Where the offset in the ByteBuffer (returned by getXXXByteBuffer() API will be normally a non ‘0’ position). To avoid this ambiguity, we plan to add new set of getter APIs for the offset in BBs. These are

int getRowPositionInByteBuffer()

int getFamilyPositionInByteBuffer()

int getQualifierPositionInByteBuffer()

int getValuePositionInByteBuffer()

int getTagsPositionInByteBuffer()

When any code uses getXXXByteBufffer(), it has to use getXXXPositionInByteBuffer() and getXXXLength to determine the range of component bytes in the BB.

Changes in read path

The ByteBufferedCell changes in code will be limited to very few areas. When the actual HFileBlock is backed by DBB, during read, we will create Objects of a class which implements the new Interface (ie. ByteBufferedCell). In the read path the Object will get passed a Cell reference. We will make a CellComparator version which is aware of the ByteBufferedCell. When it gets Cells for compare, it will check whether the Cell is a ByteBufferedCell and if so, instead of getXXXArray() API it will use the getXXXByteBuffer() API for the comparison. We will do compare on BBs rather than byte[]s.

Also we will add Util APIs in Cellutil and those also will be ByteBufferedCell aware. For eg: take API matchingRows() which has to check rowkeys in 2 cells and say whether they match of not. When the Cell is a ByteBufferedCell, it will use getXXXByteBuffer() API for equality check.

Unsafe BB compare

Our current comparator (Bytes) having a PureJavaComparer as well as UnsafeComparer. The former one will be much slower as it compares bytes one after other serially. The UnsafeComparer do compare 8 bytes at a time (Until we have the 8 bytes blocks in the byte[]). (Pls refer to Code Bytes# LexicographicalComparerHolder#UnsafeComparer). Similar way we can have Unsafe based comparer for BB also. This helps in two ways. (wrt performance gain). We know that get() on Java NIO BB is slower compared to getting bytes from a plain byte[]. There are boundary checks in the get() code path. Using the Unsafe way of accessing the BB bytes can avoid this overhead. Also we can make sure we can do 8 bytes compare at once. (Until we have Long block and then 4 bytes as it is read as in Int and then Short and Byte). Also we can make sure the code in the UnsafeBBComparer is inline able.