Supporting Tags in HBase

Tag format

<1 byte type code><2 byte tag length><tag>

As we see that there is a scope for N number of tags in a KV there is a need for the type. The type helps in understanding the way the tag needs to be interpreted. For example, an ACL tag is different from the Visibility tag. Different coprocessors can be configured to handle different tags. As we are dealing with N number of tags with in a KV we need the type to indicate what the tag represents. Going through this document will help in understanding how we will handle different types of tags using CPs.

The following methods are added in the Cell interface already

int getTagsOffset ()

int getTagsLength ()

int getTagsArray ()

When we want to persist the tags in the HFiles then there is always a need to bump the minor version of the HFile.

Client Usage

Clients can specify the tags in the Puts they want to using the

Put.add (KeyValue kv).

The KeyValue here would be the one that knows about tags.

Sample:

```
Put putKVWithTags = new Put(row);
List<Tag> tags = new ArrayList<Tag>();
tags.add(new Tag(Tag.Type.Visibility, "private&confidential"));
tags.add(new Tag(Tag.Type.Acl, "READ"));

putKVWithTags.add(new KeyValue(row, fam, qual, HConstants.LATEST_TIMESTAMP,   value, tags));
```

The other way could be using the OperationAttributes.

```
Put put = new Put(row);
put.add(fam, qual, HConstants.LATEST_TIMESTAMP, value);
put.setAttribute("visibility", Bytes.toBytes("private"));
table.put(put);
```

When we go with the OperationAttribute then it is up to the CP to read this attribute and take appropriate action based on the type of the tag. So we may have a separate CP for visibility and a separate CP for ACL(we do have one already).

```
public static class TestCoprocessorForTags extends BaseRegionObserver {
    @Override
    public void prePut(final ObserverContext<RegionCoprocessorEnvironment> e, final Put put,
        final WALEdit edit, final Durability durability) throws IOException {
```

```
      byte[] attribute = put.getAttribute("visibility");

      byte[] cf = null;
      List<? extends Cell> updatedCells = new ArrayList<Cell>();
      if (attribute != null) {
        for (List<? extends Cell> edits : put.getFamilyMap().values()) {
          for (Cell cell : edits) {
            KeyValue kv = KeyValueUtil.ensureKeyValue(cell);
            if (cf == null) {
              cf = kv.getFamily();
            }
            KeyValue.Tag tag = new KeyValue.Tag(KeyValue.Tag.Type.Visibility, attribute);
            List<Tag> tagList = new ArrayList<Tag>();
            tagList.add(tag);

            KeyValue newKV = new KeyValue(kv.getRow(), 0, kv.getRowLength(), kv.getFamily(), 0,
                kv.getFamilyLength(), kv.getQualifier(), 0, kv.getQualifierLength(),
                kv.getTimestamp(), KeyValue.Type.codeToType(kv.getType()), kv.getValue(), 0,
                kv.getValueLength(), tagList);
            ((List<KeyValue>) updatedCells).add(newKV);
          }
        }
        // add new set of familymap to the put. Can we update the existing kvs
        // itself
        NavigableMap<byte[], List<? extends Cell>> familyMap = new TreeMap<byte[], List<? extends Cell>>(
            Bytes.BYTES_COMPARATOR);
        put.getFamilyMap().remove(cf);
        // Update the family map
        put.getFamilyMap().put(cf, updatedCells);

      }

}
```

Using the Put.add (KeyValue kv) will allow one to specify the tags per KV.

Using the OperationAttributes.setAttributes () will allow tags to be specified per put.

If a user uses both, like he specifies tags using the Put.add (KeyValue kv) and also uses the OperationAttributes, the OperationAttributes will take the priority and the same will be used. We can check on this behavior in terms of the client usage.

Validations of Tags

Note that with any of the above approaches if there is a need to validate the correctness of the tags then it can be done only the Server side. Again the corresponding CP for the specific tag would be aware as to what type of validations needs to be applied. Client side validations are currently not considered in this approach.

Using KeyValueCodec to persist tags:

The format of the KeyValueCodec is

<length of the KV><byte array comprising the KV>

Currently keyValueCodec is used in case of WalCellCodec when there is no compression is used.

In case of WAL we tend to read every KV from the stream. So the KV persisted in the WAL needs to be read back individually.

So using KeyValueCodec.decoder() would get individual KVs. The newly introduced RPC layer also tries to use this KeyValueCodec.

Making changes to the in memory KeyValue structure would help us to persist the tag byte array also into the HFile data using KeyValueCodec. KVs with tag and without tag can be serialized in the same manner without any change to the write path though the read path needs some modifications to understand and interpret the tag as we deal with HFileBlock bytebuffers. The HFileWriter.append(KeyValue) would look like

```
{
      DataOutputStream out = fsBlockWriter.getUserDataStream();
      Encoder encoder = kvCodec.getEncoder(out);
      encoder.write(kv);
      // How to handle this memstoreTS
      if (this.includeMemstoreTS) {
        WritableUtils.writeVLong(out, kv.getMemstoreTS());
      }
    }
```

In memory representation of tags in the KeyValue format

Tags will be added to the existing KV format at the end of the exisiting byte array that comprises of the keylength, valuelength, keyarray and the valuearray. Internally we know that the keylength has the rowkey, cf, qualifier, type and timestamp.

<keylength><valuelength><keyarray><valuearray><taglength><tagarray>

Here the taglength is the total tag length and the tag array represents the byte array that forms the entire tag array (this includes multiple tags).

The tag length includes the tag type, the individual tag length and byte array for the individual tag.

The <taglength> - will be an int like how we write the key length and value length. Should be possible to make it as a vint so that instead of 4 bytes the length would mostly occupy 1 byte.

Using KeyValueCodec would mean that the above in memory structure of the KV format is applicable only when tags are embedded in the KV. For KVs without tags the existing in memory structure would be applicable.

When we don't go with KeyValueCodec we may have to always persist the taglength as 0 so that we would be able to know that the KV with taglength 0 does not have tags.

Having tags in the KV structure should not impact an application that does not use tags. Hence the choice of persisting KVs without tags and reading them back should have minimal/zero impact on the critical code paths.

Using KeyValueCodec or having an optional way of specifying tags at the HFile/HFileblock level(applicable when we don't use KeyValueCodec) are some techniques that could be considered for having a minimal performance for use cases that does not have tags.

HFile/encoder changes to support KeyValueCodec way of Serialization

->HFileBlock

So when we try to use the KeyValueCodec with the HFile, though we can make the HFileWriter to serialize individual KVs as KeyValueCodec does but when we read from the hfiles it is not per KV but HFileBlocks. Basically we issue a full read or a positional read to the DFS stream and end up fetching one full HFileBlock. Now already we have the byte buffer of one block from the underlying stream.

If we don't go with KeyValueCodec.encoder() type of serialization of KVs in HFiles, in case of NO blocks encoders, we would be issuing reads on the byte buffer to read the key length and value length and then use this info to position the byte buffer to form individual KVs. In case of KVs with tags we would need to issue 2 more additional reads to read the taglength and the tag byte array.

With KeyValuecodecs in place now we would issue two reads to read the total KV size and then read the entire byte array and then form the KVs. The advantage here is that we don't need to have special logic to serialize KVs with tags. Just that we serialize the entire KV so there would be no special way to handle KVs with tags. So the presence of tags is confined only with KeyValue class. The KeyValue class would identify the tag part in the internal byte array representation of per KV.

```
int kvSize = blockBuffer.getInt();
byte[] kv = new byte[kvSize];
blockBuffer.get(kv, 0, kvSize);
currentKV = new KeyValue(kv, 0, kvSize);
```

->MemstoreTS

KeyValueCodec does not have the facility to add MemstoreTS. So this would make us add the memstoreTS to be added after KeyValueCodec.encoder() is done.

->HFileBlockEncoders

All the encoders will need to understand the tags. To make this simpler in the 1st version we would be making the base class tag aware and just keep writing/parsing the tag data. Later we can identify techniques to encode these tags specifically based on the encoder types.

Currently have a simple Dictionary based implementation for Tags like how WAL dictionary is used.

HFileBlockEncoders would work the same way but would need to issue one additional read to get the total KVSize and then parse the individual KV byte array elements to go through the encoding algo.

The conclusion is that, we could use KeyValueCodec.encoder() to serialize the KVs but we would not be able to use the KeyValueCodec.decoder() on the HFileBlocks. But the introducing KeyValueCodec would

unify the way KVs are persisted in the entire system. We need to quantify the performance results of this though.

Going with KeyValueCodec would make the write path simple so that the KV with and without tag is serialized in the same way and we may not need any HFile level or HFileBlockHeader level changes to know if there are tags in the persisted KVs, though the read path needs some changes to understand the presence of tags.

Making Tags Optional like MemStoreTS

If we don't use the KeyValueCodec to serialize the KVs in HFile, we could handle the tags like MemstoreTS in order to minimize the impact on the read performance.

The idea of making these tags optional is similar to that of how MemStoreTS works.

The implementation of making tags optional can be per HFile or per Data block. In case of per HFile

For cases where there are no tags,

All KVs added to memstore will have the taglength written as 0. When we do a flush of the memstore we go through every KV and write them into the HFile and once we close the file we add the Meta data saying tagpresent = true and avg_tag_len = 0.

Any reads on the above flushed HFile happens we would create a reader that would know that though the tag length was written the avg_tag_len is still 0 so instead of reading the taglength from the byte buffer it would do a skip of the 4 bytes thus not impacting the read flow.

When we do a compaction of the above HFiles, we can use the above information and create a writer that does not add the Tag information at all in the newly created compacted HFiles. For this HFile the metadata tagPresent = false and avg_tag_len = 0.

It would be better if we could use the **optional tag feature per block rather than per HFile**.

This would involve changes to the HFileBlockHeader. Add 2 bytes to the end of the HFileBlockHeader. If the first byte is 1 it means tagPresent = true and if the second byte is 1 then it means that tag_len is > 0, meaning there was atleast one kv with tag in that block.

Whenever a writer is created for flush or compaction always write with tags. But per block keep track of the total number of tag bytes being written. In the HFileBlockHeader add the info as said above.

So for blocks with no tags the first byte will be 1 and the second byte will be 0.

For blocks that has even one tag the first byte will be 1 and the second byte will also be 1.

So any reader that opens up on this file will read this hfileblock info and decide whether on that block it has to read the tag length bytes or just reposition the buffer instead of reading it.

So for blocks with tags the taglength will always be read whereas for blocks with no tags the buffer will just be skipped(means the position of the byte buffer is just moved by 4 bytes).

How scan uses the Tags:

One example of how scan can make use of the tags is in the case of Visibility labels. We will see some sample code that helps in doing this

```
Scan s = new Scan(row);
s.setAttribute("visibility", Bytes.toBytes("private&confidential"));
```

In order to filter out this tag with the type Visibility we will implement the preScannerOpen () method of the CP.

We can set the filter that tries to filter out KVs that do not contain this tag “private”. So we can have a new filter called TagFilter or more specifically if the filter is going to deal with Visibility labels then we can call it as VisibilityFilter.

```
scan.setFilter(new TagFilter(scan.getAttribute("visibility")));
```

Inside this filter we can have the logic to find out the tags that matches the one given in the setAttribute of the scan.

The filterKeyValue () method in the filter can be implemented to filter out the KVs that do not have the specified tags.

```
public ReturnCode filterKeyValue(KeyValue kv) throws IOException {
    if (kv.getTagsLength() > 0) {
      byte[] tagArray = kv.getTagsArray();
      Tag decodeTag = KeyValueUtil.decodeTag(tagArray);
      if (decodeTag.getType().equals(Tag.Type.Visibility)) {
        if (this.tag == null) {
          // If my scan attribute from which this.tag is set, is null, then just say SKIP because the required
          // visibility tag is not provided
          return ReturnCode.SKIP;
        }
        byte[] tagInKV = decodeTag.getTag();
        // Evaluate the tag from the KV with the tag passed to the filter.
        // Parse again the kv tag to get the tree structure
        VisibilityParser parser = new VisibilityParser(tagInKV);
        try {
          List<Node> children = parser.parse(tagInKV).getChildren();
          for (Node node : children) {
            switch (node.getNodeType()) {
              case AND: {
                break;
              }
              case OR: {
                break;
              }
              case TERM: {
                break;
              }
            }
          }
        } catch (Exception e) {
        }

        boolean equals = Bytes.equals(this.tag, tagInKV);
        if (equals) {
          return ReturnCode.INCLUDE;
        } else {
          return ReturnCode.SKIP;
        }
      } else {
        return ReturnCode.SKIP;
```

```
      }
    }
    return ReturnCode.SKIP;
  }
```

Backward Compatability

As the way the KVs are serialized is getting modified we may need to handle backward compatability issues for HFiles without tags in them. Bumping the minor version would help us in assigning a newer version to HFiles with tags and any older HFile should be handled by the existing way of write/read path.