[HBASE-13408] HBase In-Memory Memstore Compaction: Master Evaluation Results

October 22, 2015

Eshcar Hillel (eshcar@yahoo-inc.com)

Anastasia Braginsky (anastas@yahoo-inc.com)

We ran again the same benchmarks, measuring latency, this time with the master code. As a reminder, we aim to simulate a high-churn scenario, where a set of keys in a specific column family (store) are repeatedly updated and read. We further aim to simulate the case where the memory used by the store is limited, and the block-cache is unable to accommodate all store files blocks as the memory is contended by other stores and regions in the region server.

We see that when the block-cache hit ratio drops to 80% the performance gain of compacted memstore over the default memstore ranges between 30% to 65% depending on the workload and the status of the files on disk (before or after compaction). In case the hit ratio drops lower, e.g., due to contention by other regions, we expect the gain to be even higher.

Experimental Settings

We set up a small cluster of 3 hdfs nodes, with a single region server (and a master). The region server is allocated 1GB heap space, and is configured to have global memstore space of 300MB, flush size set to 128MB, and a block-cache of 100MB (see the hbase-site.xml below). To avoid memory fragmentation we enable mslab chunks of size 2MB.

We use a data set of 128K records, record size is 1KB.
We do not load the table with any initial data. Instead we use YCSB client with 10 threads running workload of 5M read and write operations, exercising different distributions for choosing the keys.
We start by setting target throughput to 1Kops with 50% reads and 50% updates. The figures below present the latency results for read operations with zipfian, hotspot, uniform, and latest distributions. to complete the evaluation the figures at the end of the document present the latency results for update operations.

hbase-env.sh

```
export HBASE_HEAPSIZE=1000
```

hbase-site.xml

```
<...cluster configuration settings...>
<property>
 <name>hbase.hregion.memstore.block.multiplier</name>
 <value>4</value>
</property>
<property>
 <name>hbase.regionserver.global.memstore.size.lower.limit</name>
 <value>1.0f</value>
</property>
<property>
 <name>hbase.regionserver.global.memstore.size</name>
 <value>0.3f</value>
</property>
<property>
 <name>hfile.block.cache.size</name>
 <value>0.1f</value>
</property>
<property>
 <name>hbase.hregion.memstore.mslab.enabled</name>
 <value>true</value>
</property>
<property>
 <name>hbase.hregion.memstore.mslab.chunksize</name>
 <value>2097152</value>
</property>
</configuration>
```

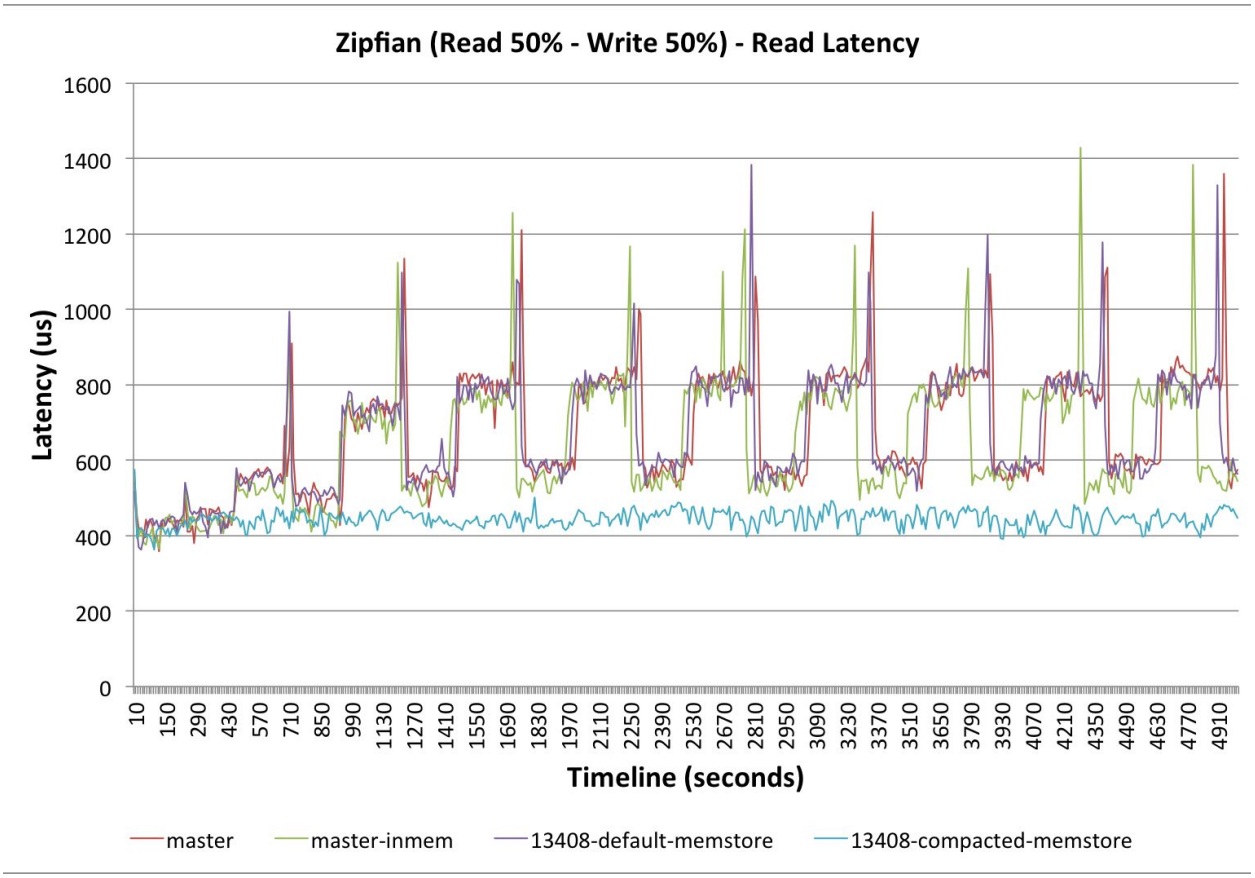
Experimental Results

The figures below depicts the average latencies of read and write operations over time. Latencies are accumulated over intervals of 10 seconds. We compare the performance of the memstore in master branch with and without in-memory column (green and red lines, respectively), the default memstore in HBASE-13408 where the column is not of type in-memory (purple line) and the compacted memstore in HBASE-13408 with in-memory column (light blue line).
All experiments exhibit similar performance patterns: all memstore but the compacted memstore frequently flush the content of the memory to disk; at some point during the experiment store files blocks fill up the block-cache space, thus imposing disk access on a big portion of read operations, degrading their performance. On the other hand, the compacted memstore implementation continuously compacts the memory content, deferring its flush to disk. As the data set is small enough

to fit into memory, read operations are served from memory and their latency LSA is predictable, remaining almost flat.

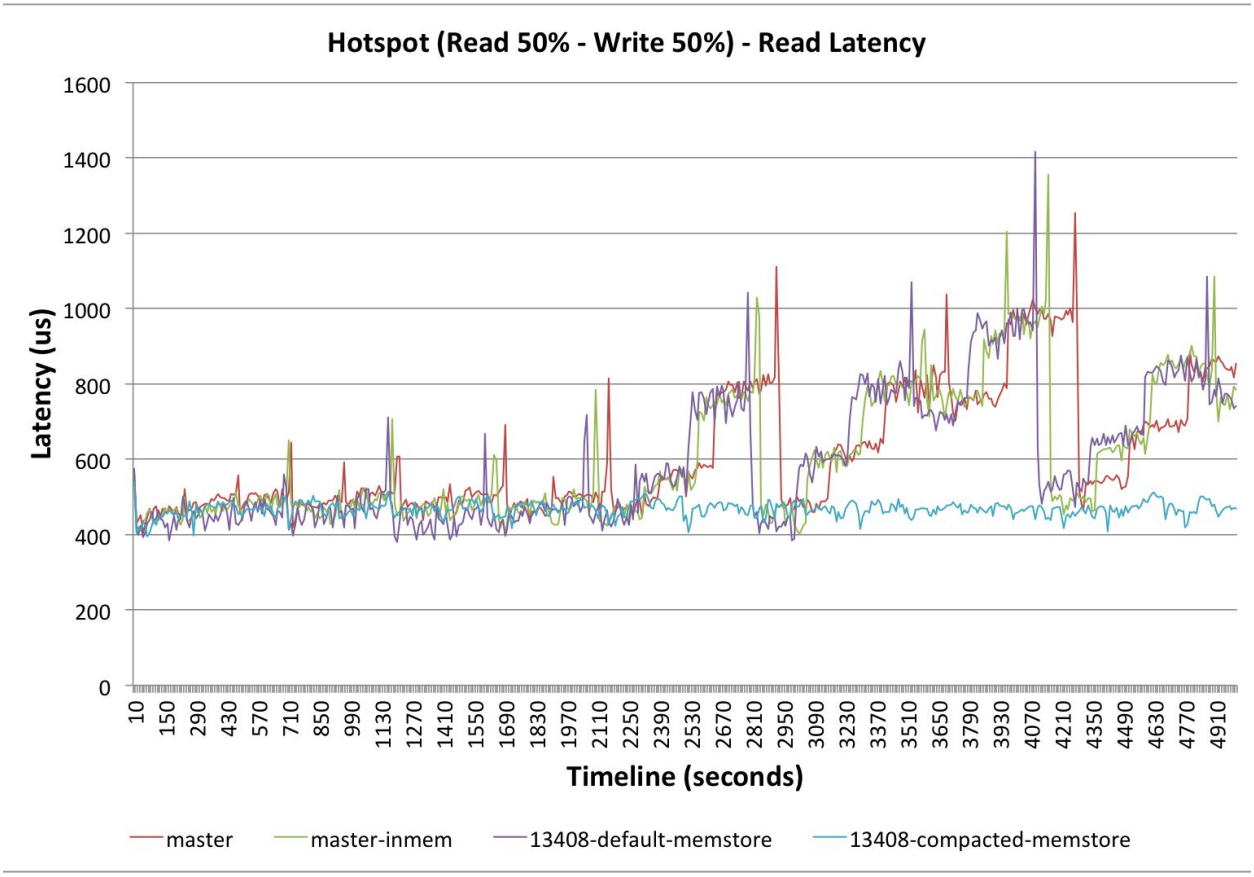
Zipfian distribution

The first figure depicts the latency of read operations when we employ a skewed zipfian distribution. All implementations start at 400us as all read operations are served from memory. The compacted memstore continuously compacts the memory content and therefore is able to maintain a flat latency line. In all other cases the default memstore implementation regularly flushes data when the memstore reaches the 128MB limit. When the data is flushed to disk it is also being compacted creating files of size ~60MB. After the second flush the block cache is full, hit ratio falls to less than 80%, and the latency increases to 550us to 600us with the third flush that triggers disk compaction. From this point there is one file on disk, the 4th flush results in 800us latency and the 5th triggers a compaction which reduces the latency back to 600us. And so on.



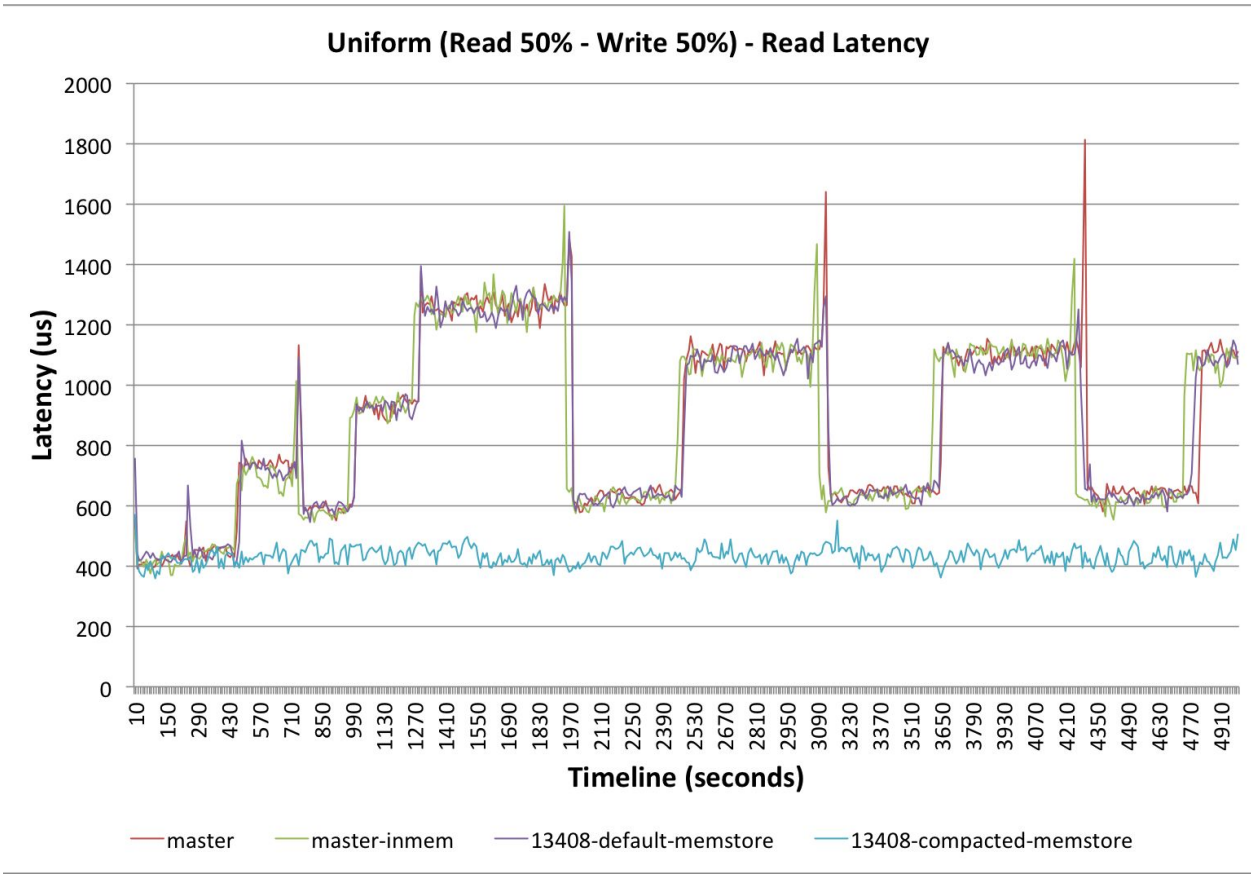
Hotspot distribution

In the hotspot distribution 90% of the operations to access 10% of the keys. The remaining 10% access 90% of the keys uniformly at random. With this distribution each flush of the default memstore creates a file of size 24MB, and it takes much longer for the block cache to fill. However once it does, the performance degrade over time as it is harder to ensure the blocks containing the hot keys reside in block cache.



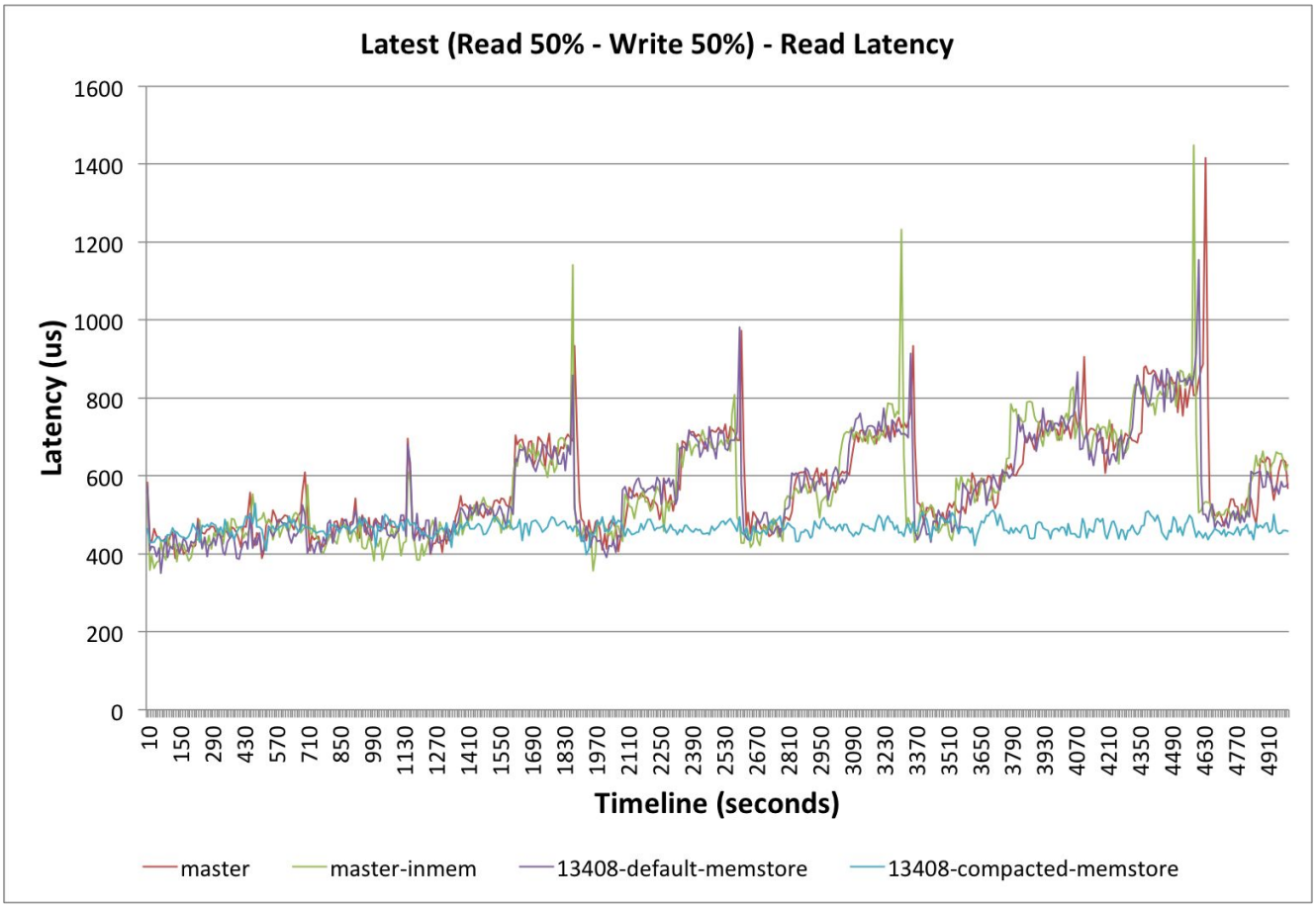
Uniform distribution

When the keys are drawn uniformly at random from the entire range the pattern is similar; latency increases up to 1100-1200us before files are compacted. This may be explained by the file size, 76MB, that is bigger with uniform distribution (as there is less data to compact). The second disk compaction (after the 5th flush) creates a file big enough to trigger a region split, which may also account for the higher latency.



Latest distribution

The latest distribution is similar to the zipfian distribution except that the most recently inserted records are in the head of the distribution.



Write operations results

Write latencies of HBASE-13408 are comparable to master in all distributions.