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

July 14, 2015

Eshcar Hillel (eshcar@yahoo-inc.com)

Anastasia Braginsky (anastas@yahoo-inc.com)

The goal of this evaluation experiments is to simulate a high-churn scenario, where keys in a column family are repeatedly updated and read. We would like to have predictable latency SLA. In the existing implementation the store is frequently flushed to disk in high churn scenarios even when the set of keys is small and can fit into memory. We further aim to simulate the case where the block-cache is unable to accommodate all store files blocks as it is contended by other stores and regions in the region server. In this case a big portion of read operations is not served entirely from memory; they incur disk access thus degrading the performance.
On the other hand, the memory component in HBASE-13408 is continuously compacted, deferring the flush of data to disk. When the data set is small read operations are served from memory and the performance is predictable as it remains flat..

Experimental Settings

We set up a small cluster of 3 hdfs nodes, with a single region server (and a master). The heap at each node is allocated 1GB, while the global memstore size is limited to 300MB and the block cache size to 100MB (see the hbase-site.xml below). To avoid memory fragmentation we use 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 to run workloads of 5M operations, set target throughput to 1Kops, 50% updates, 50% reads, and to measure their performance over time. We experimented with different distribution of keys including uniform zipfian and hotspot, and also latest and sliding window which are not presented here but show similar trends.

hbase-site.xml

```
<...cluster configuration settings...>
<property>
 <name>hbase.hregion.memstore.block.multiplier</name>
 <value>4</value>
</property>
<property>
 <name>hbase.regionserver.global.memstore.lowerLimit</name>
 <value>0.3f</value>
</property>
<property>
 <name>hbase.regionserver.global.memstore.upperLimit</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>
```

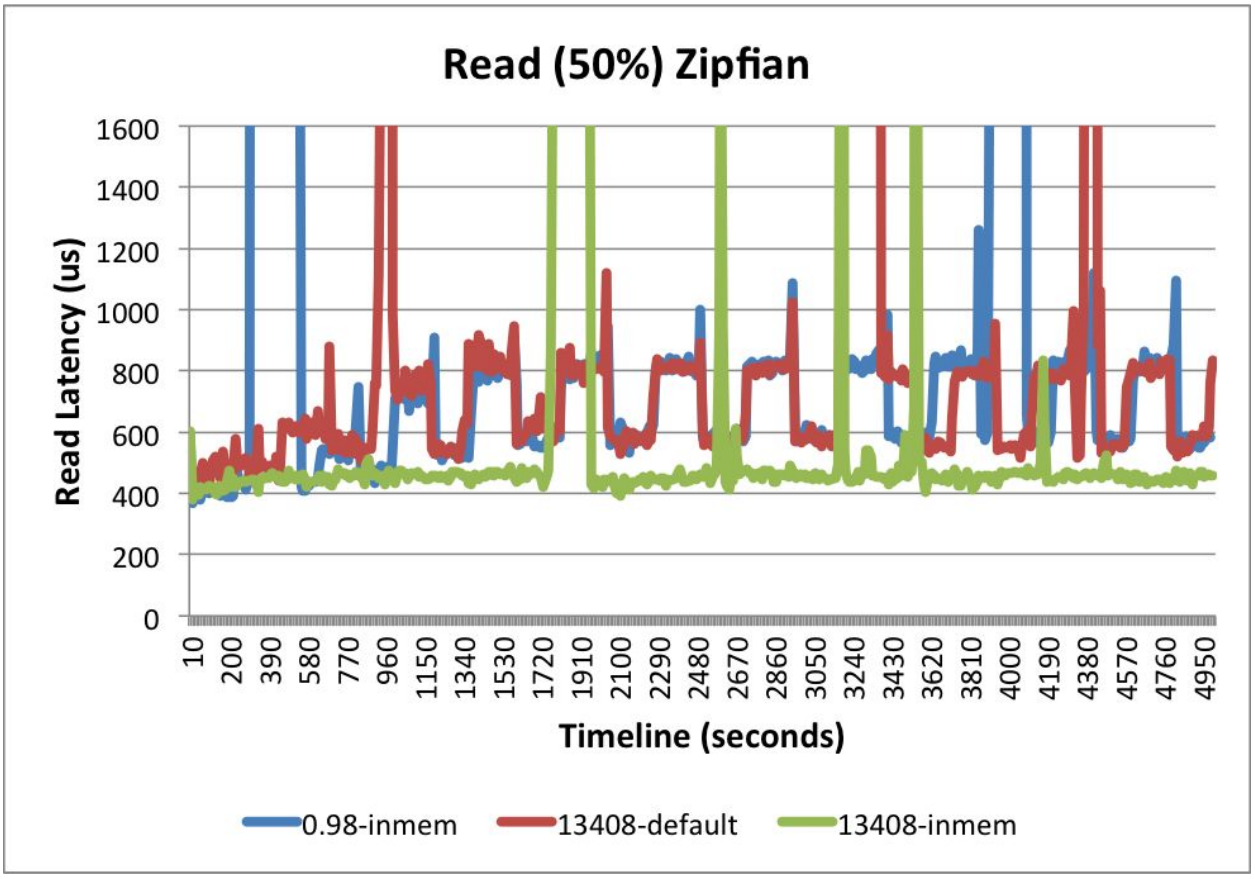
Experimental Results

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 branch 0.98 in-memory column with HBASE-13408 in-memory column.
We also present the performance result for 13408-default implementation, which is almost identical to 0.98 in-memory.

In all experiments and implementations (both 0.98 and 13408), we see some unexplained latency peaks (both in reads and writes). The timing of this peaks and the source of this behaviour is yet unclear to us, however they occur independently from our implementation, and can be segregated from our results.

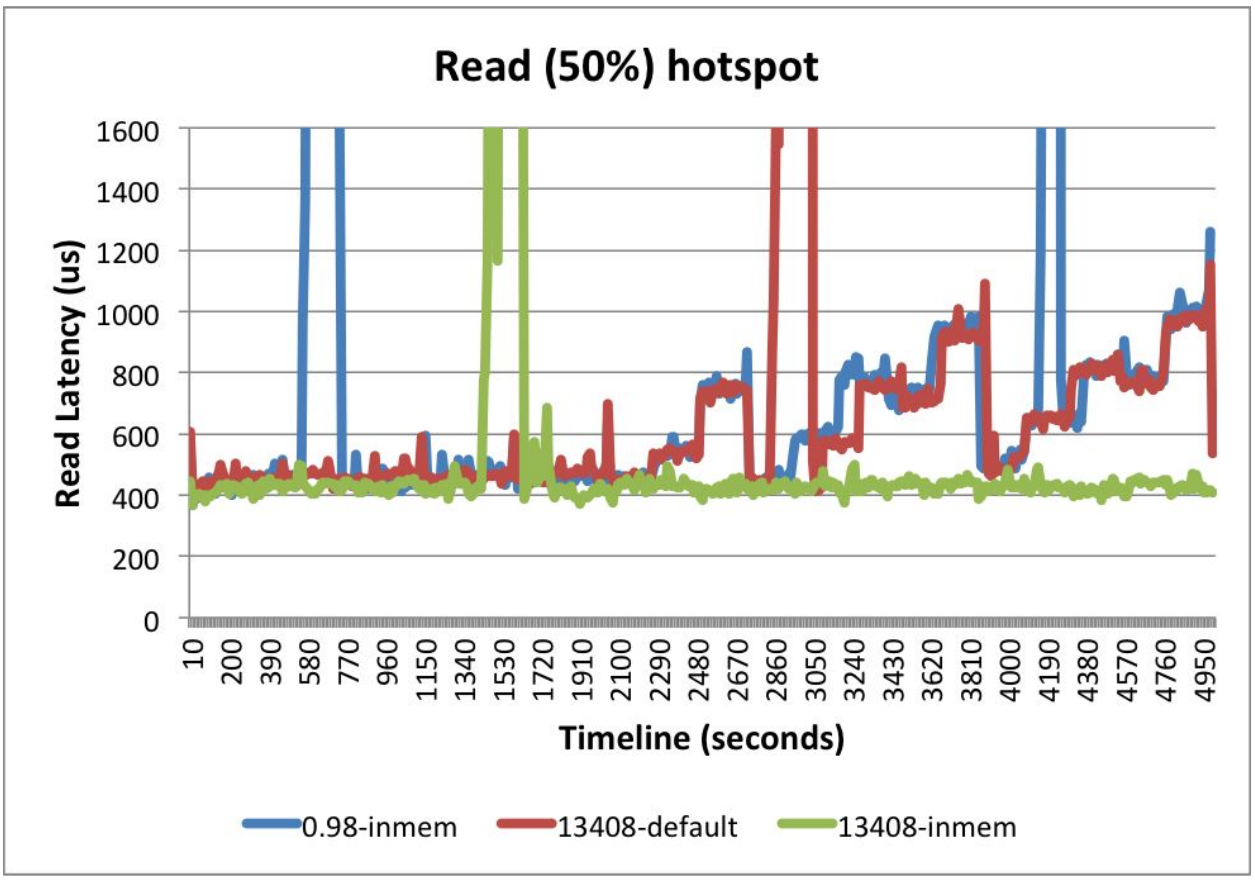
Zipfian distribution

All implementations start with 400+ us latency when the read operations are served from memory. 13408 continuously compacts the data inside the memstore component and therefore is able to maintain flat latency line (ignoring the unexplained peaks).
0.98 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 550 us to 600 us with the third flush that triggers disk compaction. From this point there is one file on disk, the 4th flush results in 800 us latency and the 5th triggers a compaction which reduces the latency back to 600 us. And so on.



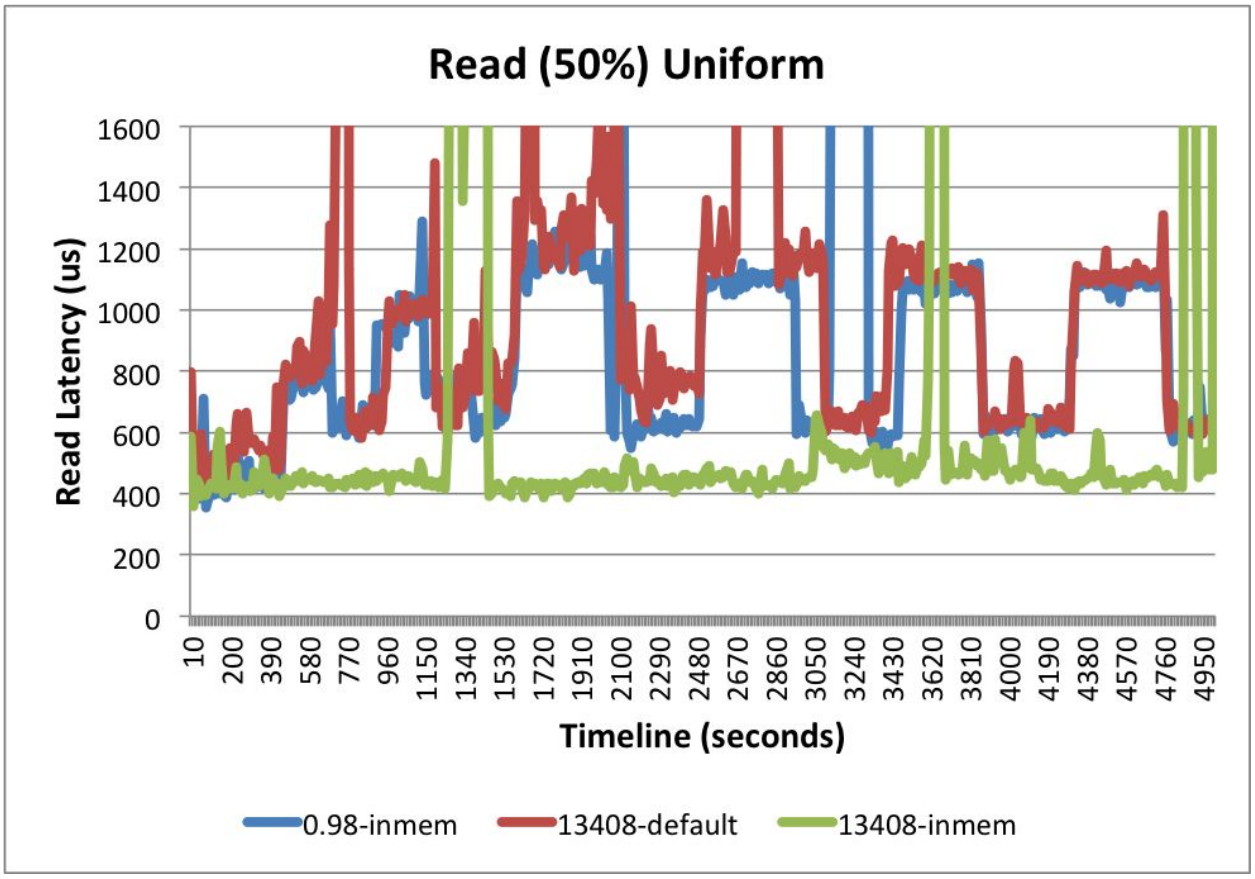
Hotspot distribution

W set 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 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

With uniform distribution we see a pattern similar to the zipfian distribution. However, the latency increases up to 1100-1200us before files are compacted. This may be explained by the file size that is larger (~76MB) with uniform distribution (as there is less data to compact). The 5th flush even triggered a region split, which may also account for the higher latency.



Write operations results

Write latency results are much noisier. The results of 13408 are comparable to 0.98.