HBASE-16417: Policy for in-memory flush&compaction - benchmark results

We run benchmarks on a single machine cluster to compare performance of no compaction (default memstore), index-compaction, and data-compaction under various workloads, with the goal of finding the optimal policy for in-memory flush and compaction.

Benchmark Settings

Hardware

SSD machine, 48GB ram, 12 cores, 2.9 TB disk

HBase configuration (fixed)

16GB heap
50 regions (presplit)
50 columns

Additional global parameters:

```
<property>
      <name>hbase.regionserver.global.memstore.size</name>
      <value>0.42</value>
</property>
<property>
      <name>hfile.block.cache.size</name>
      <value>0.38</value>
```

```
</property>
<property>
      <name>hbase.hstore.flusher.count</name>
      <value>10</value>
</property>
<property>
      <name>hbase.hstore.blockingStoreFiles</name>
      <value>25</value>
</property>
```

In most cases we run with MSLAB enabled

```
<property>
      <name>hbase.hregion.memstore.mslab.enabled</name>
      <value>true</value>
</property>
<property>
      <name>hbase.hregion.memstore.chunkpool.maxsize</name>
      <value>1</value>
</property>
<property>
      <name>hbase.hregion.memstore.chunkpool.initialsize</name>
      <value>0.5</value>
</property>
```

Saturation point: #threads=10

We would like to test the system at a point where it is loaded to the maximum it can stand before pushing back (namely, block updates). This is called the *saturation point*. It is the point of maximum throughput with minimal latency.
To identify the saturation point we
(1) run PE write-only workload with different numbers of threads (5,10,12,15, 20, 30, 50) and,
(2) plot throughput-latency graphs of these executions for each of the latency percentiles reported by PE.

Figure 1 shows that for the 50th and 75th percentiles there is no difference in latency w.r.t. The number of threads. The difference can be seen first in the 95th percentile. IT shows that the saturation point is achieved at 10 threads where the throughput is maximal (207 MB/s) and latency is minimal (80.7 ms).
The same trend is depicted in figure 2 for higher percentiles.

We ran the same tests with WAL and the saturation point was the same with 10 threads.

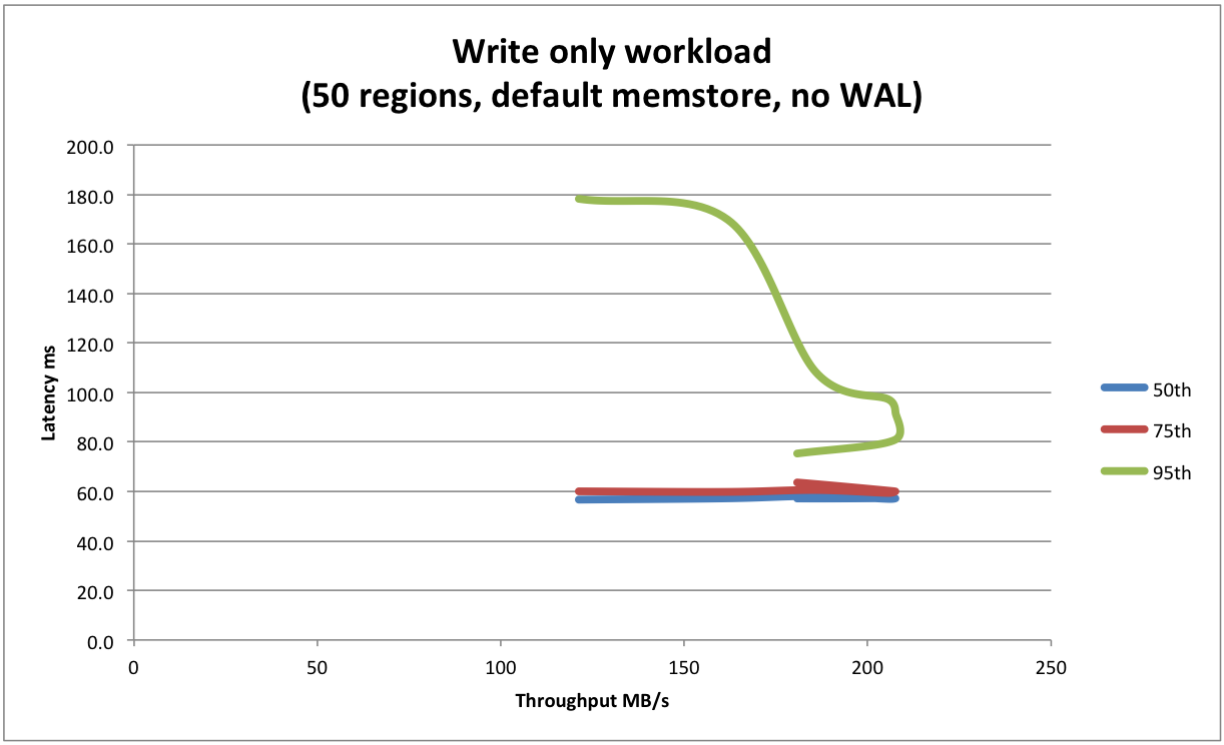

Figure 1. Saturation point at 10 threads (95th percentile)

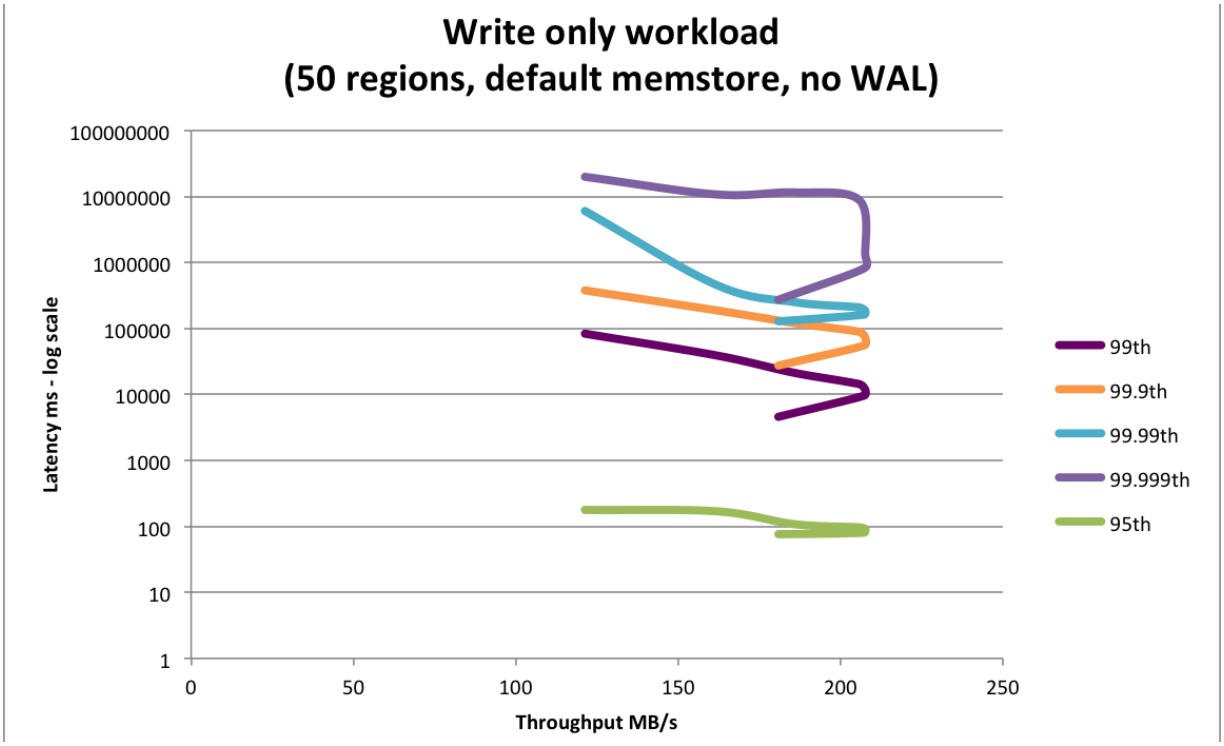

Figure 2. Saturation point at 10 threads (95/99/99.9/99.99/99.999th percentiles)

Next we compare the performance of different options (no-, index-, data-compaction) under different workloads.

Benchmark results

Write-only workload, uniform distribution (PE)

We write 50GB data using PE with the above settings. Value size is set to 200B; with 50 columns this results in 10KB per row.
We run index compaction with varying number of segments in the pipeline before merging the index: greater than 1 (ic1), greater than 2 (ic2), greater than 3 (ic3). For data compaction we **set off the MSLAB flag** to avoid the inherent space and computation overhead of copying data during compaction.
Figure 3 shows that up until the 95th percentile all options are comparable. At the 99th percentile data compaction starts to lag behind -- indeed in a uniform workload there is not much point in doing data compaction. The overhead might stem from running SQM to determine which versions to retain. One way to close this gap is to not run data compaction when there is no gain in it. A good policy should be able to identify this with no extra cost.
At the 99.999th percentile index compaction also exhibits significant overhead. This might be due to memory reclamation of temporary indices.

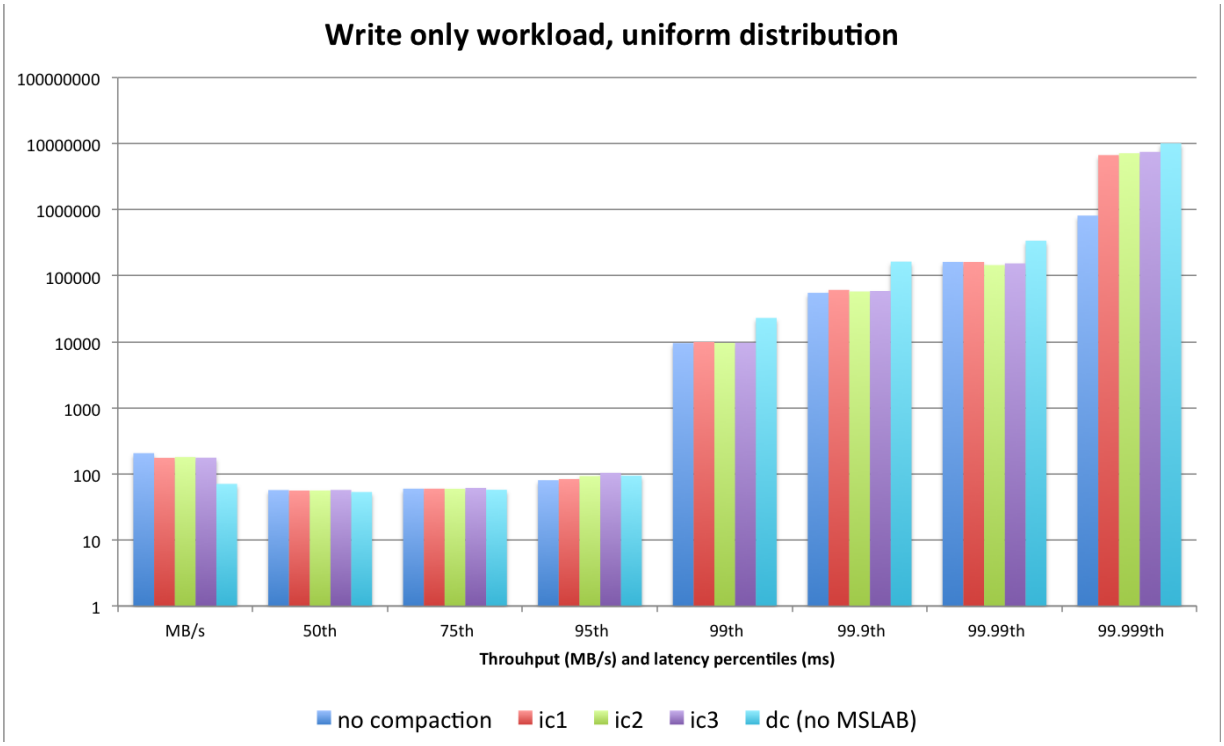

Figure 3. Throughput and latency comparison

Write-only workload, uniform distribution (YCSB)

We write 150GB data using YCSB. Value size is set to 10KB per row (divided between 50 columns).
We run index compaction which merges the index when the number of segments in the pipeline is greater than 2 (ic2). For data compaction we **set off the MSLAB flag** to avoid the inherent space and computation overhead of copying data during compaction.

A YCSB run involves network overhead; therefore it needs to work harder to stress the system. To overcome this YCSB allows using client buffers which significantly reduce the network overhead and increase the load on the region server. The downside is that for more than 99% of the operations measuring latency only considers the time it takes to write the data to the local buffer; there's no point in presenting these measurements.
Instead, Figure 4 shows the overall throughput (op/s), the gc count and the average latency. Usually it is not a good idea to look at the average latency as its variance is high, but since YCSB does not report percentiles beyond the 99th, both #gc and avg can serve as an approximation of the 99.9.. percentiles.
The trends are similar to those depicted in Figure 3 (PE results for uniform workload): all three options are comparable, where data compaction throughput and latency are inferior with respect to the performance of the other two options.

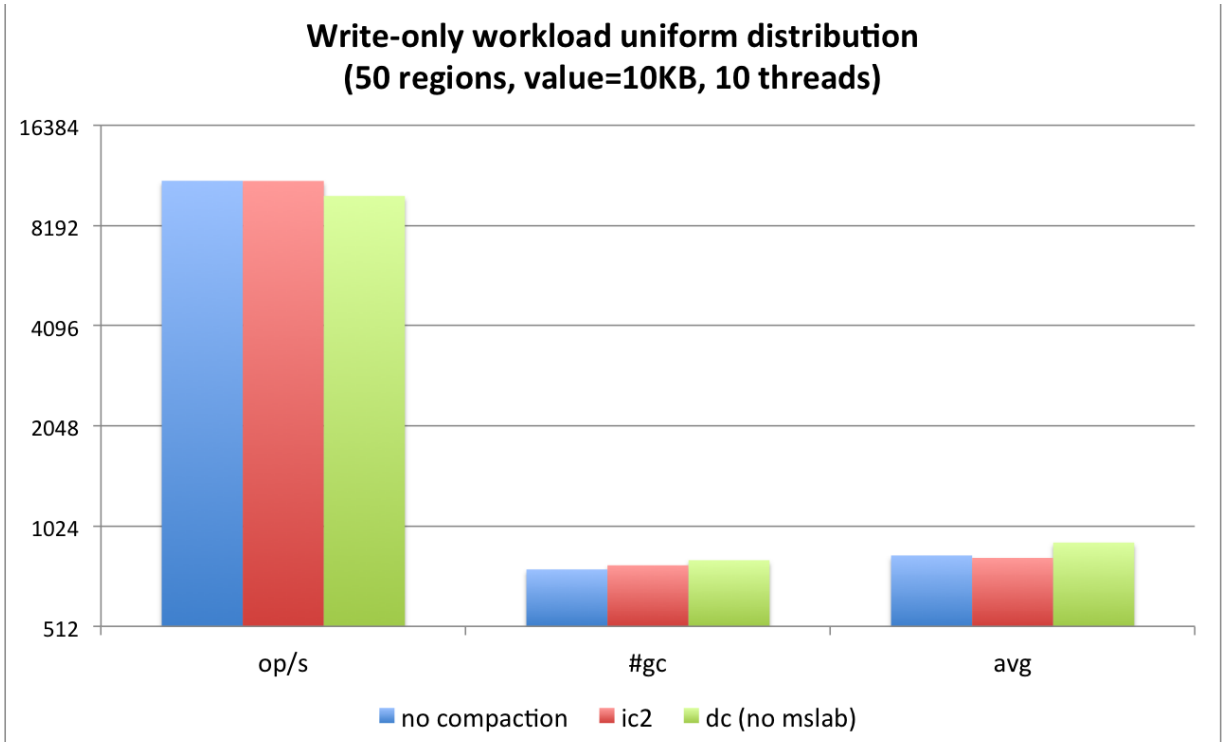

Figure 4. Throughput, avg latency, and gc count comparison

Write-only workload, zipfian distribution

We write 50GB data using YCSB. This time keys are chosen from a zipfian distribution, and the value size is only 1KB.
Figure 5 shows the overall throughput (op/s), the gc count and the average latency.
Since the values are smaller flattening the index has more impact and index compaction performs better than no compaction.

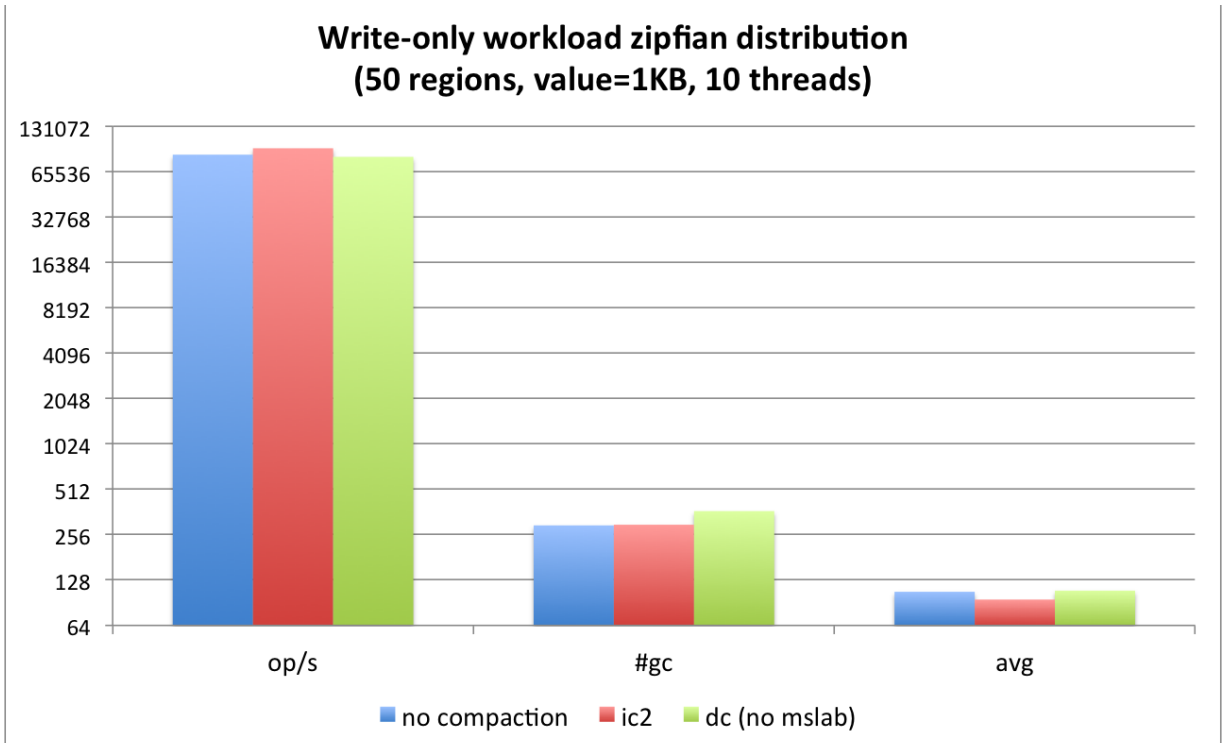

Figure 5. Throughput, avg latency, and gc count comparison