HBASE-16417: benchmark results for in-memory flush and compaction

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.

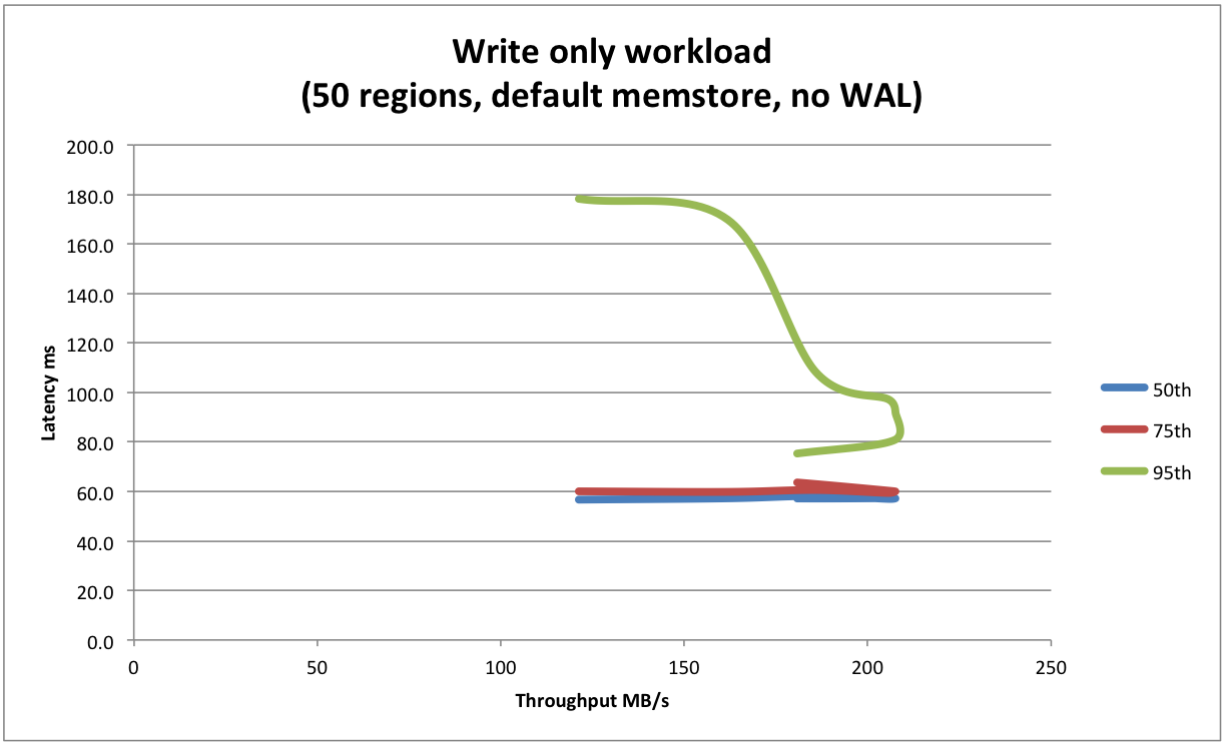

Figure 1. Saturation point at 10 threads (95th percentile)

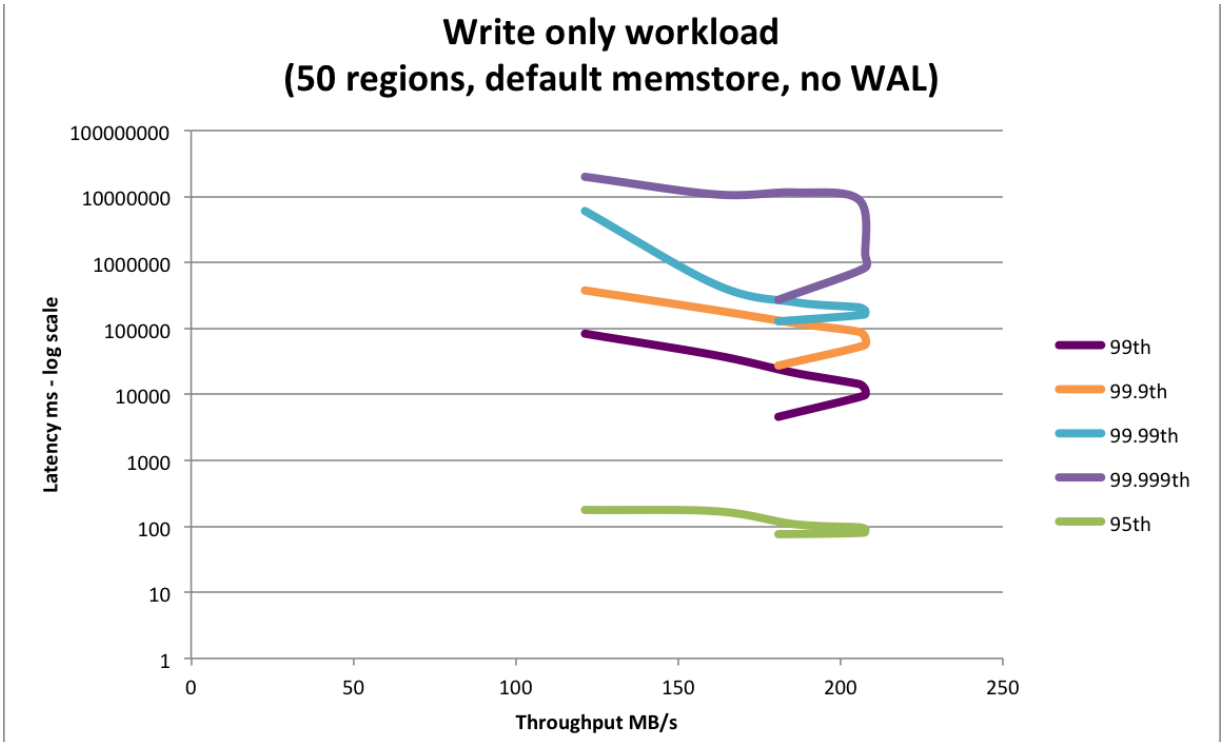

Figure 2. Saturation point at 10 threads (95/99/99.9/99.99/99.999th percentiles)

Benchmark results: single machine

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). The results were similar for all 3; only ic2 is presented here. For data compaction we **set off the MSLAB flag and removed the chunk pool settings** to avoid the inherent space and computation overhead of copying data during compaction. For a fair comparison we also ran no compaction with no mslabs and no chunk pool.
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 exhibits significant overhead. This might be due to memory reclamation of temporary indices.

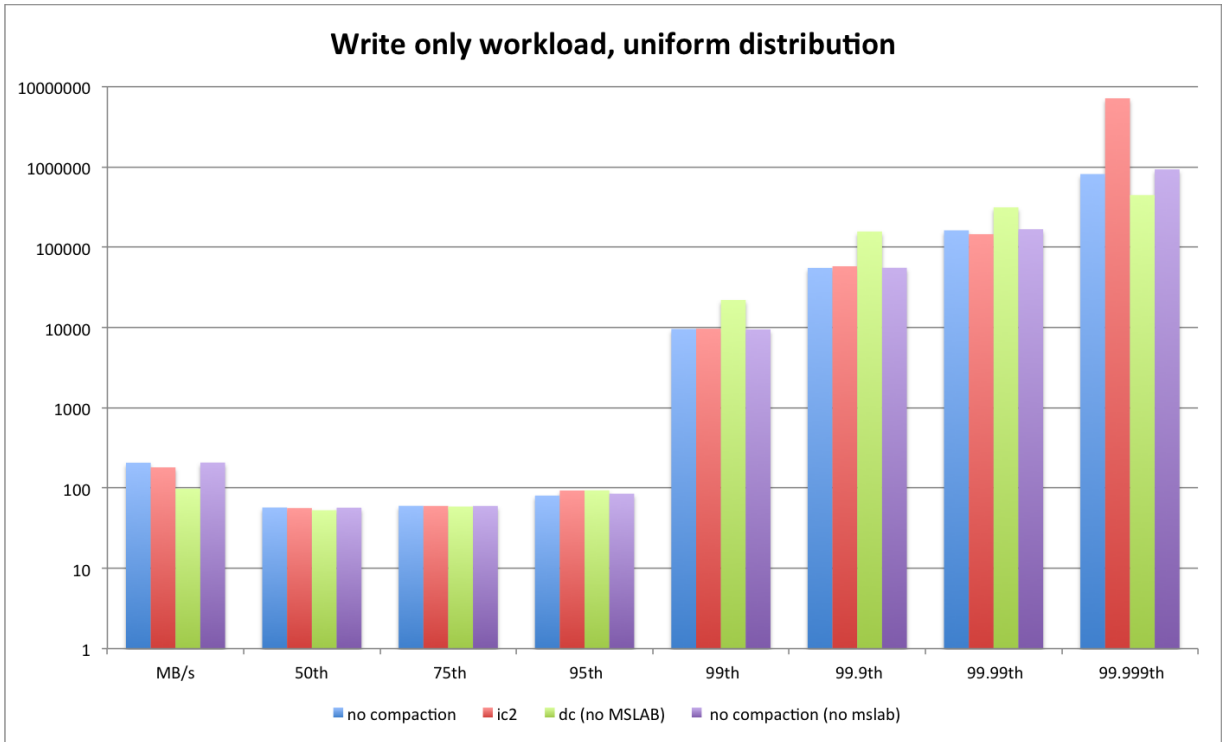

Figure 3. Throughput and latency comparison

Write-only workload, uniform distribution (YCSB)

We write 150GB data using YCSB. Value size is set to 10KB per row (divided to 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** and the chunk pool 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 options are comparable; data compaction throughput and latency is a bit worse than the other options.

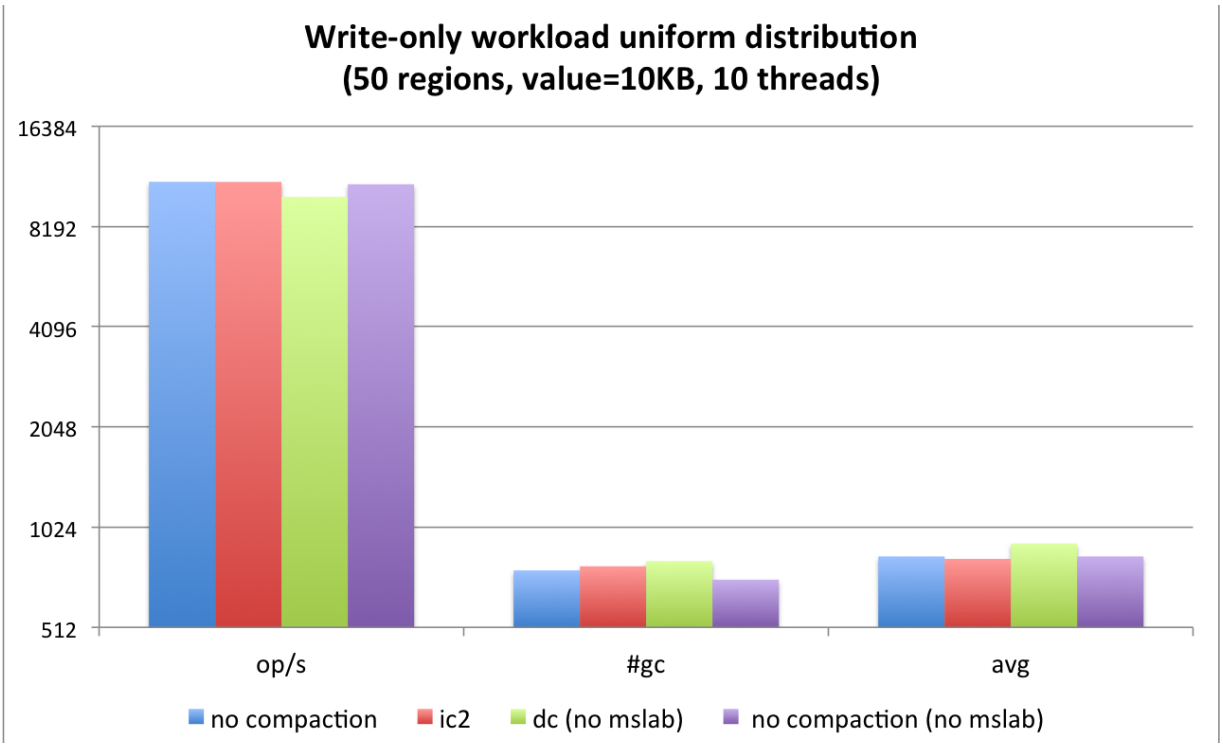

Figure 4. Throughput, avg latency, and gc count comparison

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.
Again all options are comparable but running with no mslabs (and no chunk pool) seem to have an advantage over using mslabs with no compactions.

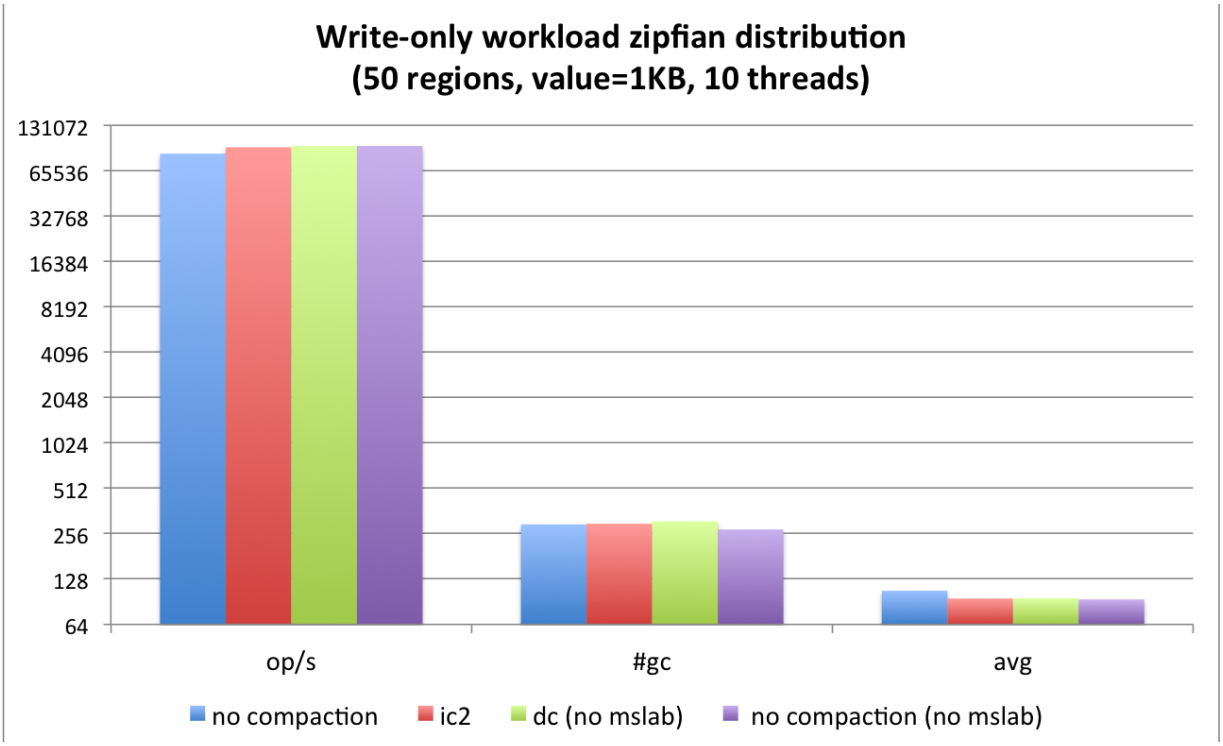

Figure 5. Throughput, avg latency, and gc count comparison

Mixed workload (50%-50%), zipfian distribution

We run mixed workload with YCSB. Initially we pre-load data by writing 5Mx1KB cells chosen uniformly at random. Then we run 50M operations, 50% reads and 50% writes, with keys chosen from a zipfian distribution.
Figure 6 presents the total throughput results, gc count, and read latency percentiles comparison. Results show that in a mixed workload running with no mslabs and no chunk pool has a significant advantage over running with chunk pool and mslabs. This is the case when running with no compaction or with data compaction. The latency gap is due to percentiles higher than the 99 percentile.

Index compaction runs with mslabs and with no chunk pool. The performance is better than when running with mslabs and a chunk pool.
Note that these gaps are significant in mixed workload and not in write-only workload. This might be since now the block cache is used in full capacity, leaving the gc to struggle with less free space. The chunk pool uses more space than the application can afford.

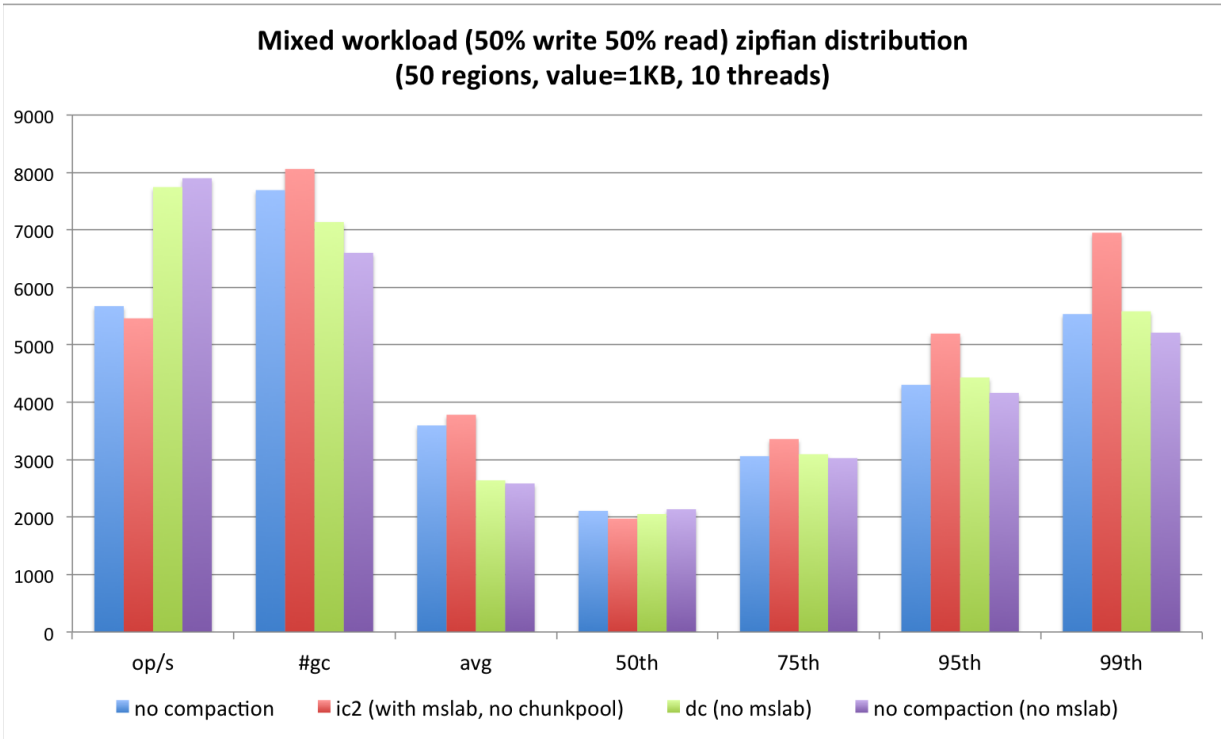

Figure 6. Total throughput and read latency comparison

Benchmark results: cluster (3 machines)

Experiments show that running with mslabs and chunk pool causes degradation in performance. Specifically with mixed workload avg read latency of no compaction with mslabs is 2.5x than the other options running with no mslabs. Additional run to verify these numbers even failed to complete.

Therefore from this point on we focus on running experiments with no chunks pool and no mslabs. We compare the three policies discussed in HBASE-16851: **none** no compaction, **basic** flattening index, merge only upon flush to disk (HBASE-17081), and **eager** data compaction.

In addition rest of the experiments run with WAL, and batching writes at the client side in buffer of size 10KB (vs no WAL and a buffer of size 12MB in previous experiments).

Cluster settings

3 ssd machines, all machines with the same hardware as in the single machine experiments. HDFS is deployed on all machines (3-way replication); hbase: two region servers, master on third machine with the following setting:
16GB heap, from which 40% allocated to memstore and 40% to block cache (default values)

Additional properties:

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

Write-only workload, zipfian distribution, small values

Having 2 RSs we pre-split the table into 100 regions (50 region per RS); data set of 200M keys. We run 20 threads to write 100GB. Keys are chosen from a zipfian distribution, and the value size is only 100B.

Table 1 shows statistics for the write amplification. Where write amplification is defined as the size of data written to disk divided by the size of data written by the client.
Write amplification = (total flush size+total compaction size) / 100,000 MB.
Eager improves the amplification by 21%, while basic improves it by 15%.

	Write amplification	
None	2.201	100%
Basic	1.865	85%
Eager	1.743	79%

Table 1

Figure 7 shows the overall throughput (op/s), average latency, GC, flush, and compaction counters, and size of data written to disk.
Basic and eager outperform no compaction by 15% and 25%, resp. This can be attributed in part to running less GC and in part to executing less IO.

Figure 7. Total throughput, write latency, gc count and write amplification comparison

Mixed workload (95%-5%), zipfian distribution, small values

We pre-split the table into 100 regions (50 region per RS); data set of 200M keys.
Initially we pre-load data by writing 200Mx100B cells chosen uniformly at random (total 20GB).
Then we measure the performance of 200M operations, 95% writes and 5% reads, with keys chosen from a zipfian distribution.
Table 2 shows the reduction in number of cache accesses and cache misses by basic and eager w.r.t no compaction. Cache misses are translated into disk accesses.

	#cache accesses	hit ratio (%)	misses
None	37,000,000	90	3,700,000
Basic	35,500,000	90	3,550,000
Eager	29,500,000	89	3,245,000

Table 2

Figure 8 presents read latency percentiles comparison; it shows modest improvement in average read latency of basic and eager over none.With slower disks (HDD) reduction in cache misses will have more positive affect on avg and read tail latencies.

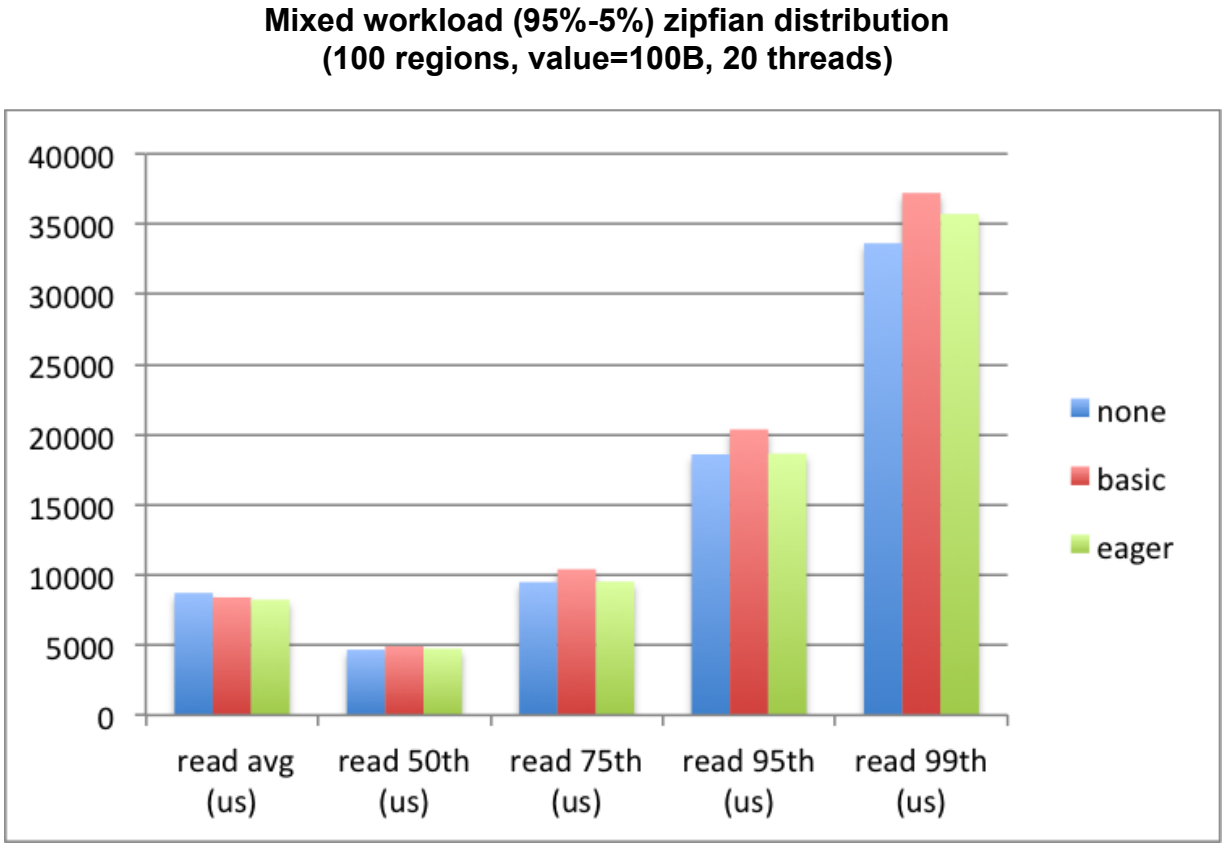

Figure 8. Read latency comparison

Scan memory-only first optimization

The current implementation of a get operation to retrieve values for a specific key scans through all relevant stores of the region; for each store both memory components (memstores segments) and disk components (hfiles) are scanned in parallel.
We suggest to apply an optimization that speculatively scans memory-only components first and only if the result is incomplete scans both memory and disk. We ran the mixed workload experiment for none, basic and eager with and without the optimization.

Figure 9 presents read latency comparison. The optimization improves the avg latency of none and basic by 10% and the avg latency of eager by 7%.
Table 3 summarizes the ratio between number of memory-only scans vs. number of full scans (only 31%-35% of the operations require full scan) when running with the optimization.
Table 4 summarizes reduction in cache accesses (70% less accesses), and cache misses (40%-45% less misses) which lead to this improvement.

	#memory-only scans	#full scans	Reduction in full scans
None+opt	9,897,523	3,460,000	65%
Basic+opt	9,952,320	3,460,000	65%
Eager+opt	9,965,795	3,120,000	69%

Table 3: reduction in number of full scans when running with memory-only first optimization (considering that without the optimization every scan is a full scan).

	#cache accesses	Reduction in cache accesses	Hit ratio (%)	#cache misses	Reduction in cache misses
None	37,000,000	-	90	3,700,000	-
None+opt	10,900,000	71%	80	2,180,000	41%
Basic	35,500,000	-	90	3,550,000	-
Basic+opt	10,700,000	70%	80	2,140,000	40%
Eager	29,500,000	-	89	3,245,000	-
Eager+opt	8,800,000	70%	80	2,140,000	46%

Table 4: reduction in cache accesses and cache misses with memory-only first optimization

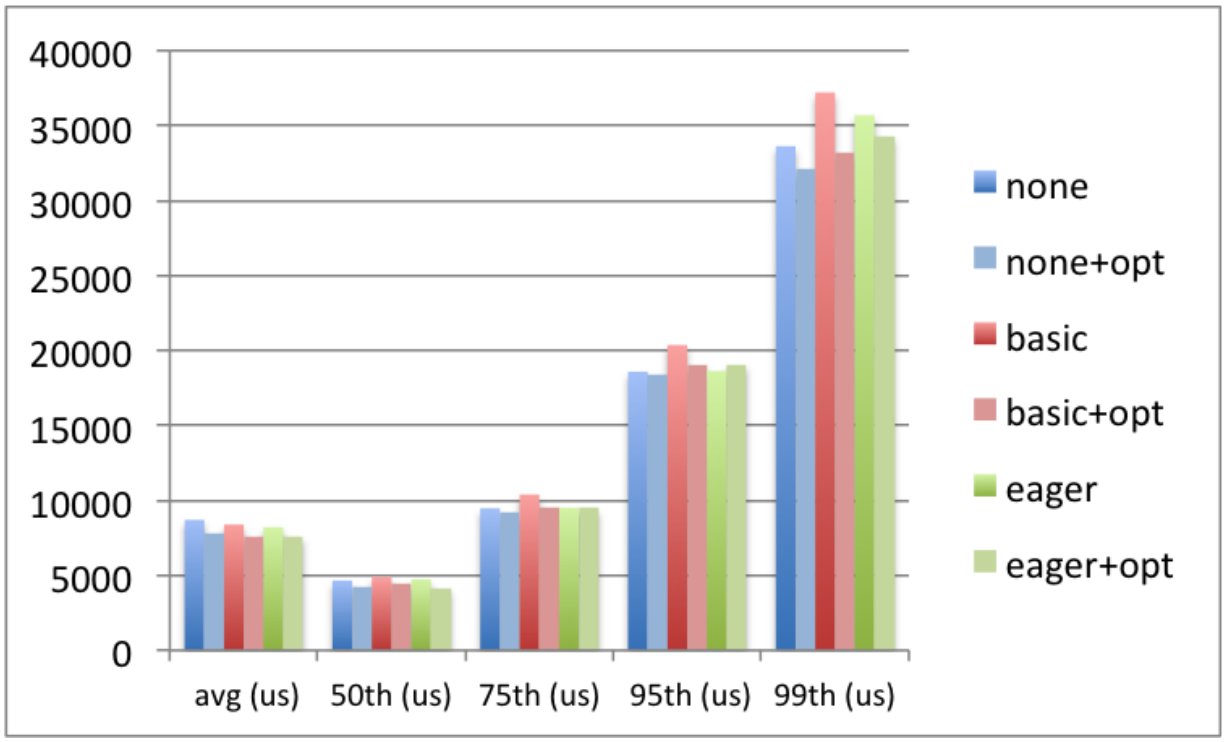

Figure 9. Read latency comparison w/o memory first optimization

Scans workload (95%-5%), zipfian distribution, small values

We pre-split the table into 100 regions (50 region per RS); data set of 200M keys.
Initially we pre-load data by writing 200Mx100B cells chosen uniformly at random (total 20GB).
Then we measure the performance of 200M operations, 95% writes and 5% scans; first key of each scan is chosen from a zipfian distribution, size of scan is chosen uniformly from 1-5000.

Figure 10 shows scan performance of all three are comparable with a modest improvement of basic and eager over none (5%-7%).

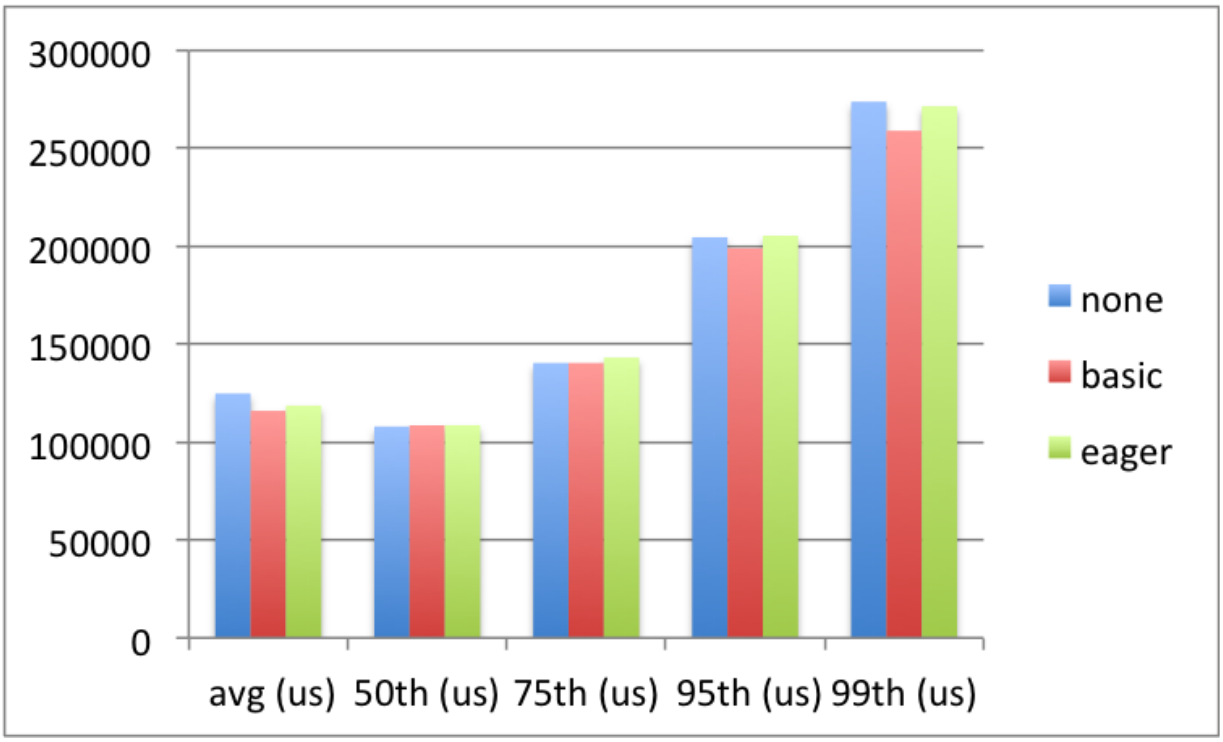

Figure 10. Scan latency comparison