HBASE-16417: Adaptive Compaction

Adaptive is a heuristic that chooses whether to apply data compaction or not based on the level of redundancy in the data. Adaptive triggers redundancy elimination only for those stores where positive impact is expected.

Adaptive uses two parameters to determine whether to perform redundancy elimination.
The first parameter, **u**, estimates the ratio of unique keys in the memory store based on the fraction of unique keys encountered during the previous merge of segment indices.
The second is the perceived probability **t** that the store can benefit from redundancy elimination. Initially, t=0.5; it then grows exponentially by 2% whenever a compaction is successful and decreased by 2% whenever a compaction did not meet the expectation. It is reset back to the default value (namely 0.5) upon disk flush.

Adaptive triggers redundancy elimination with probability t if the fraction of redundant keys 1-u exceeds a parameter threshold R.

Experiments

We compare the performance of different in-memory compaction policies and different policies.
We experiment with two types of production machines with directly attached SSD and HDD storage. We experiment with two distributions: heavy-tailed (Zipf) and uniform.
All puts write a full row (4 cells of 25 bytes each), and all gets retrieve a single cell.

Write only workload

Our first benchmark set exercises only put operations. We run 500 milion update operations writing total of 50GB.
Figure 1 depicts the performance speedup for Basic and Adaptive vs None (no compaction). Every data point is the median among five runs.
For the Zipf benchmark, the maximal speedup is 48% on SSD and 25.4% on HDD. For the uniform benchmark, which has neither redundancy nor locality, they are 25% and 9%, respectively.
Adaptive is studied under multiple redundancy thresholds: R=0.2, R=0.35 and R=0.5.

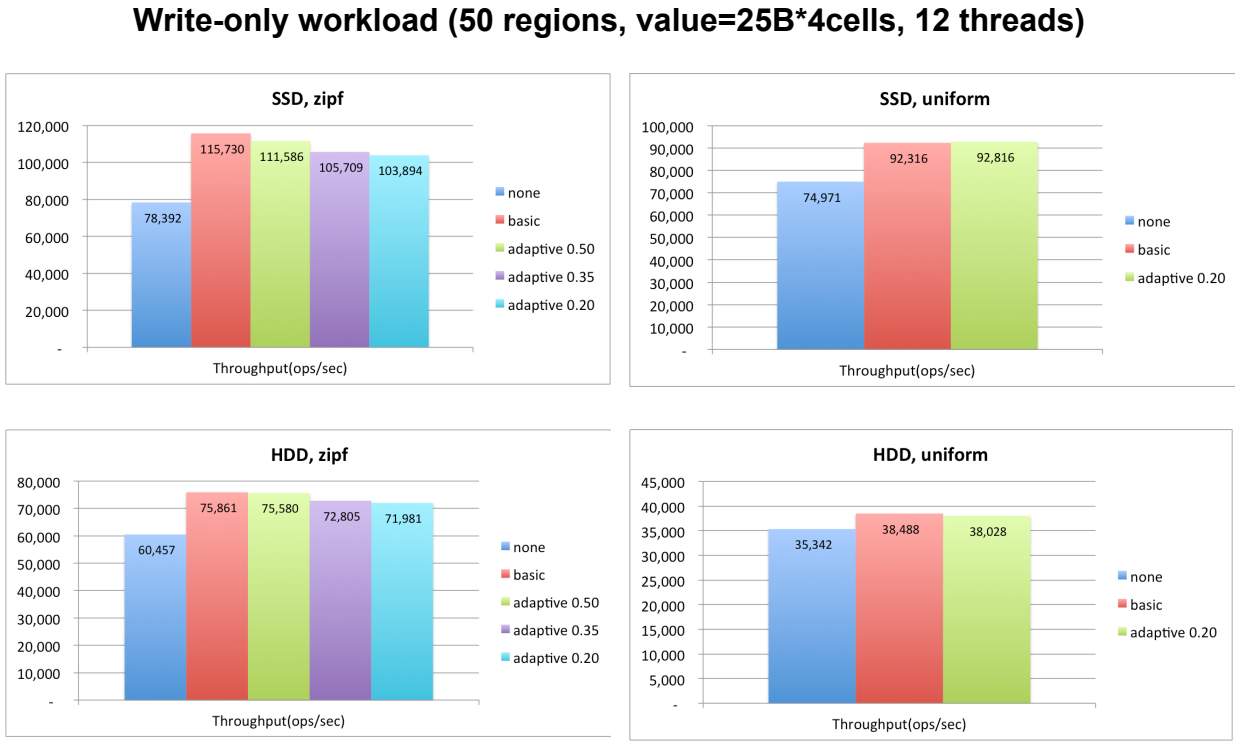


Figure 1. Write throughput speedup vs none achieved with Basic and Adaptive policies. Measured on the Zipf and uniform key distributions, on SSD and HDD hardware.

We also compare the number of bytes written to disk during disk flushes and compaction. Figure 2 depicts the write volume of the different policies. As expected, the lower the redundancy estimate threshold R, the more aggressive the algorithm, and consequently, the higher the savings. All-in-all, there is a tradeoff between the write throughput and storage utilization

The amount of data written to WAL, 185GB (approximately 45% of the write volume) remains constant across all experiments.

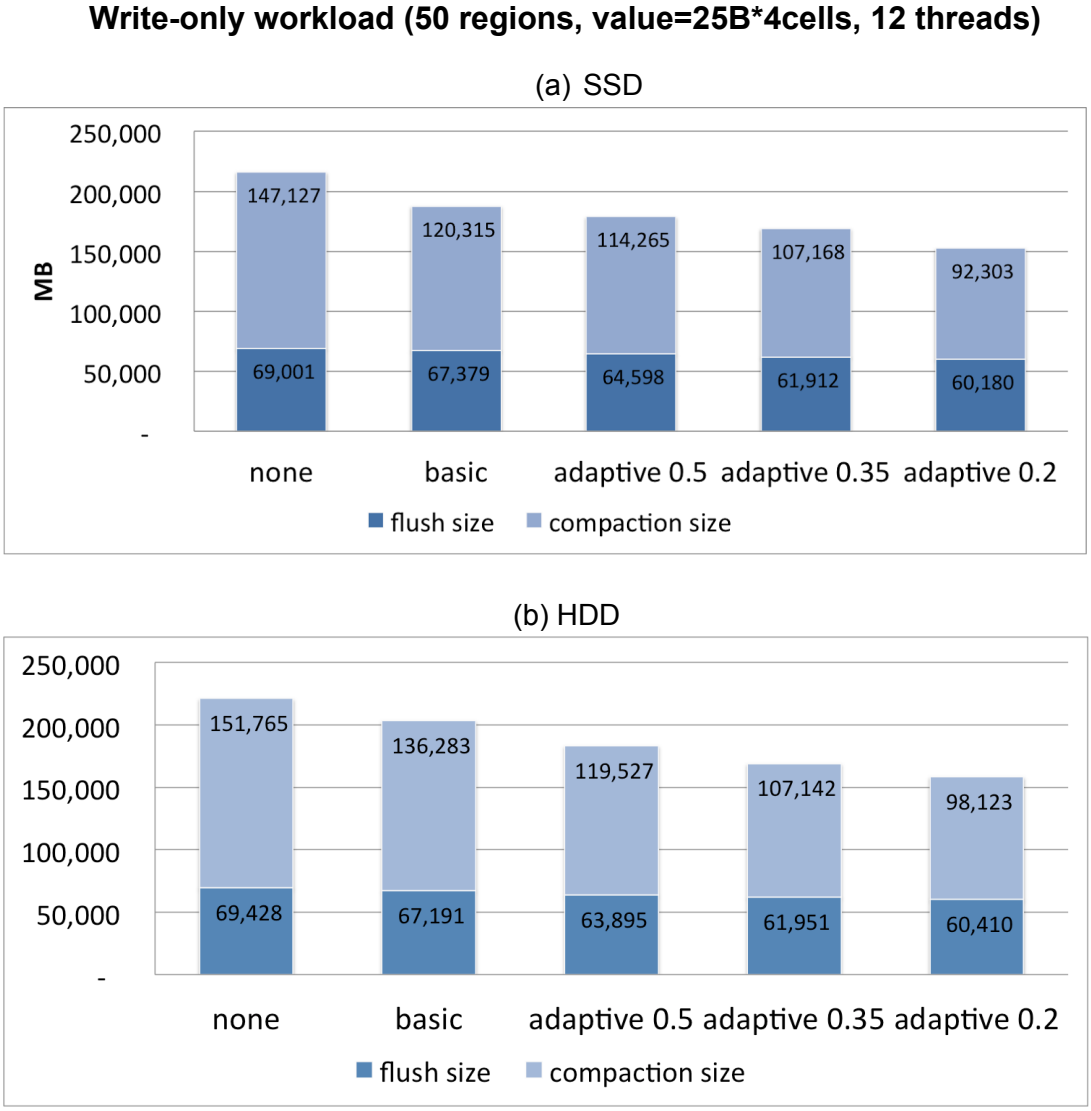


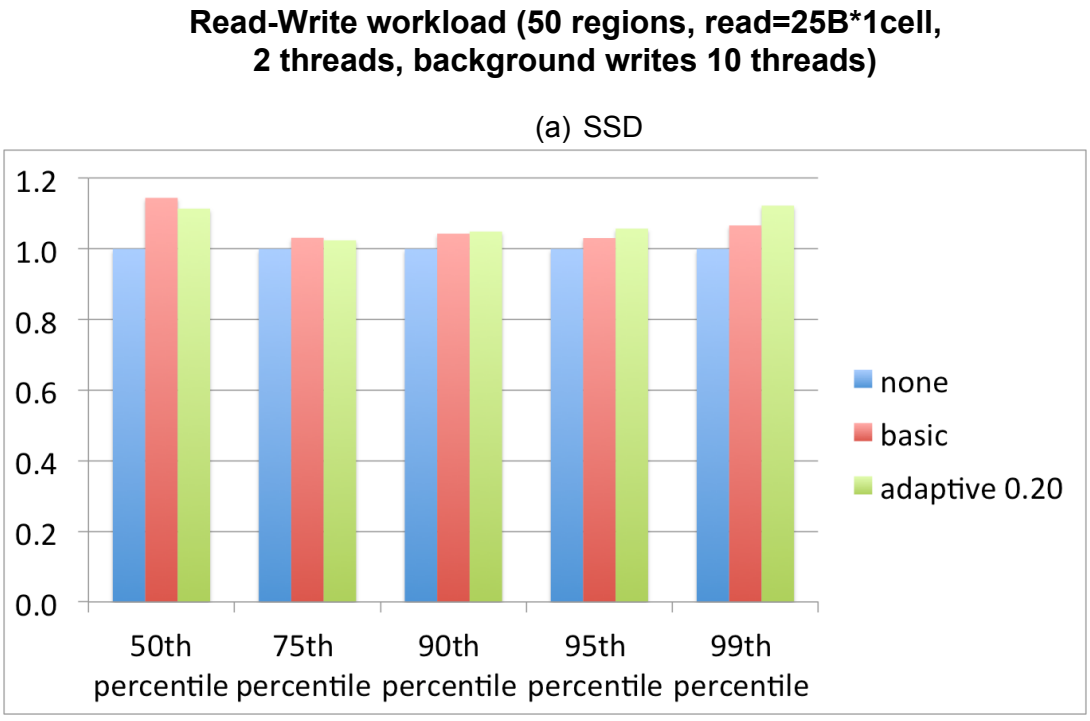
Figure 2. Bytes written by flushes and compactions, measured for different policies, for the Zipf key distribution.

Read-Write Workload

Our second benchmark studies read latencies under heavy write traffic.
We run batched puts from 10 client threads and single-key gets from two other threads.
The keys of all operations are distributed identically (Zipf). We measure the 50th,
75th, 90th, 95th and 99th get latency percentiles. Figure 3 depicts the relative latency slowdown of Basic and Adaptive (R=0.2) versus None.
%(values greater than 1 mean slowdown).

The HDD systems enjoy a dramatic reduction in tail latencies
The tail latencies are cache misses that are served from disk.
The latter are most painful with HDDs. Thanks to in-memory compaction, Adaptive and Basic prolongs the lifetime of data in MemStore. Therefore, more results are served from MemStore, i.e., there are less attempts to read from cache, and consequently from disk, in case of a cache miss.

In SSD systems, the get latencies are marginally slower in all percentiles. This happens because most reads are dominated by in-memory search speed, which depends on the number of segments in the MemStore pipeline (in our experiment, 5).



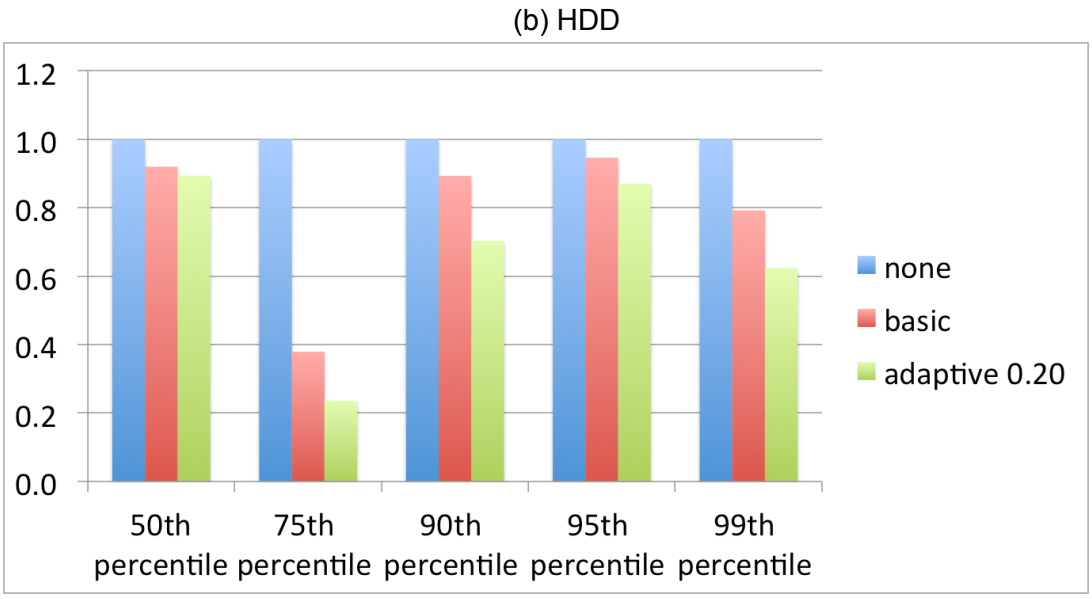


Figure 3. Read latency speedup (respectively, slowdown) of Basic and Adaptive versus None, under high write contention.