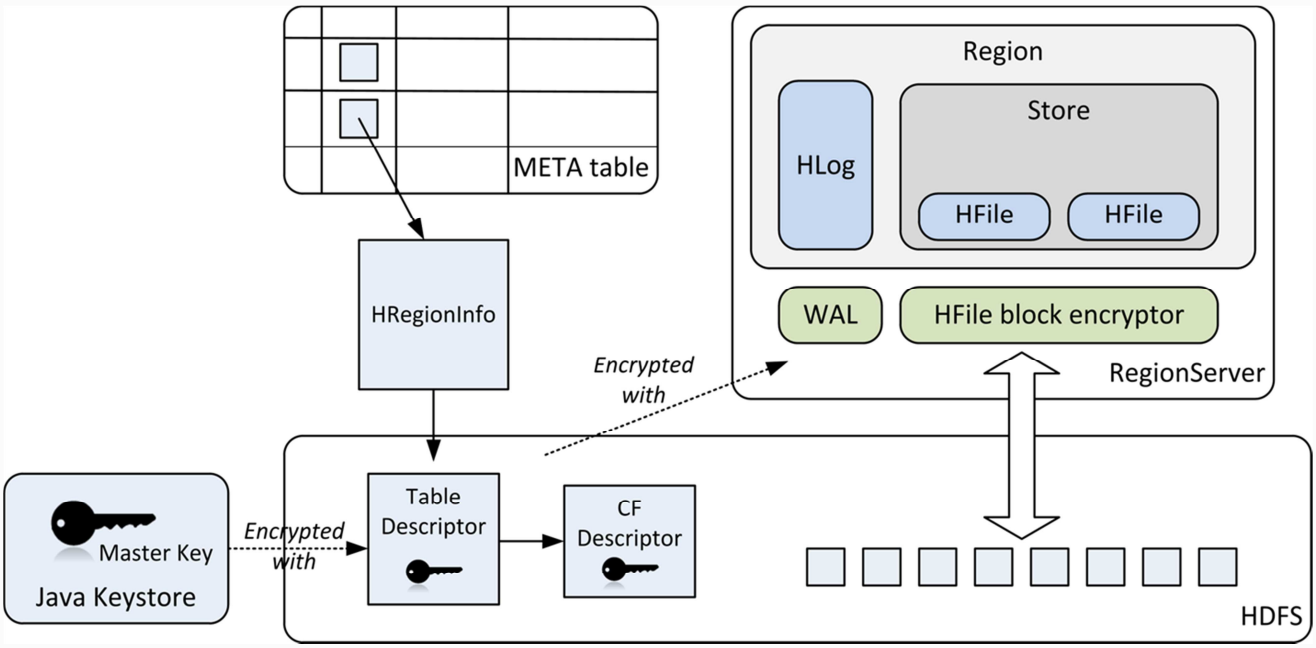
With the attached patch (*7544.patch*) we introduce transparent encryption for HFiles and the WAL for consideration. This work builds on a crypto codec framework to be submitted to the Hadoop Common and MapReduce projects. It will be maintained out of tree as a patch and a feature branch hosted on GitHub until those APIs show up downstream.

Here is an overview of the changes:



We extend the HFile V2 reader and writer, and the block encoding infrastructure, with support for optional encryption of HFile block data (all blocks except the fixed file trailer and a new key material meta block). Encryption plugs in almost exactly like compression codecs, and can be configured along with existing compression options, or alone.

The cluster admin first creates a master key and then deploys it to any key management system that integrates with the Java KeyStore API. How HBase retrieves key material is configurable via the site file. The key may be stored on the cluster servers, protected by a secure KeyStore file, or on an external keyserver, or in a hardware security module. This master key is resolved as needed by HBase processes through a configurable mechanism provided by the crypto framework. Then, encryption keys can be specified in schema on a per CF basis. Currently the cluster admin does it at table creation or on existing tables by modifying the column descriptor via HBaseAdmin#modifyFamily. We envision a key management tool possibly doing this at some future time. Per CF keys facilitates low impact incremental key rotation and reduces the scope of any external leak of key material. The key is stored in the CF schema metadata, encrypted with the cluster master key. Once the CF is configured for encryption, any new HFiles will be encrypted. To insure encryption of all HFiles, trigger a major compaction. The key for decryption, encrypted with the cluster master key, is stored in the HFiles in a new meta block. At file open time the key will be extracted from the HFile, decrypted with the master

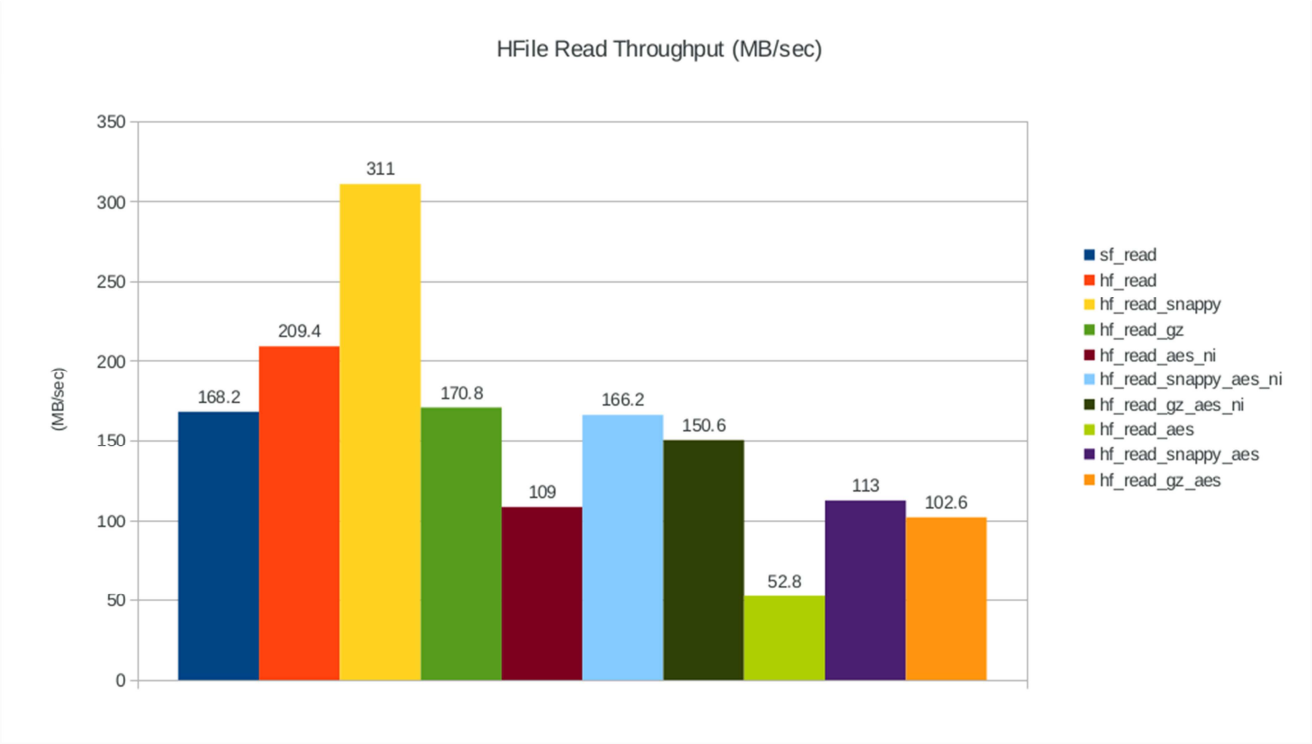
key (if available), and used for decryption of the remainder of the HFile. An HFile will be unreadable if the master key is not available. Should remote users somehow acquire access to the HFile data because of some lapse in HDFS permissions or from inappropriately discarded media, there will be no means to decrypt either the file key or the file data.

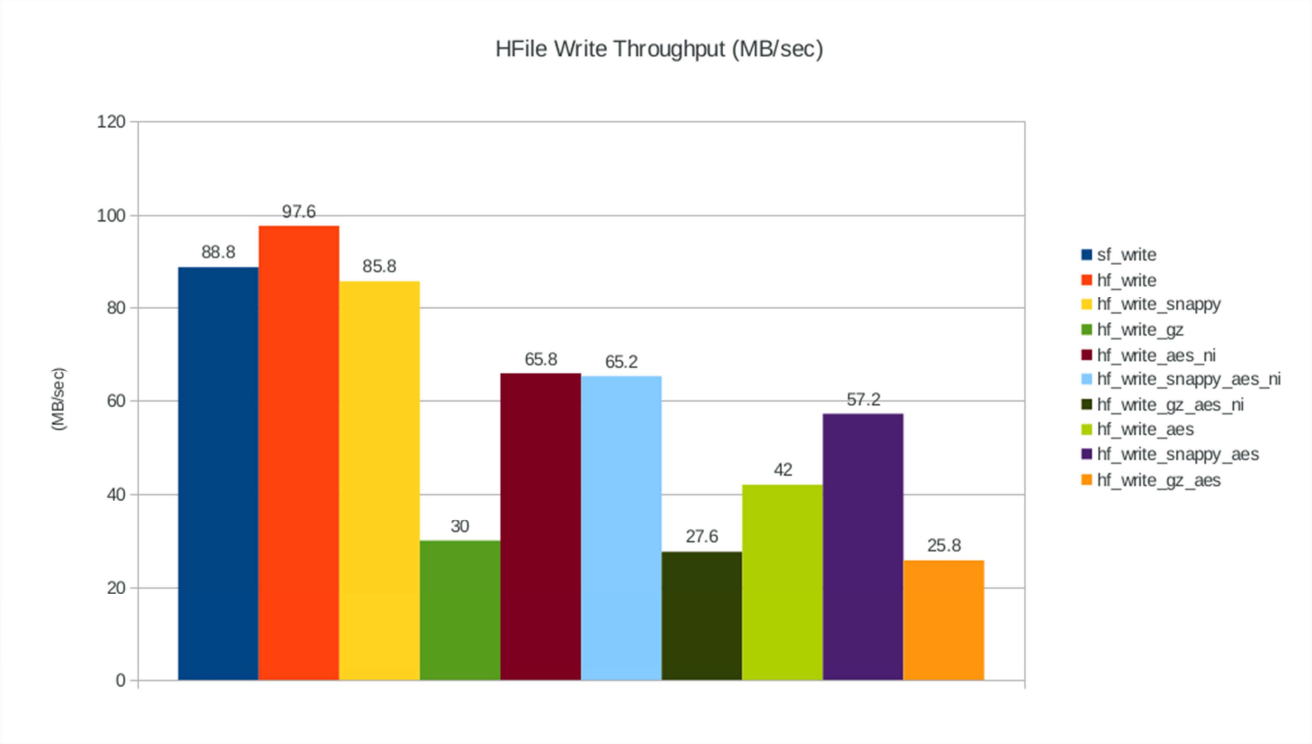
Key rotation is made simple by this design. First, the CF key is changed in the column descriptor. Then, major compaction is triggered either on the table at once or region by region. Once compaction has completed, all files will be (re)encrypted with the new key material. While this process is ongoing, HFiles encrypted with old key material can of course be decrypted as described above. Depending on how pluggable compaction policies evolve, we may need to make slight modifications to insure there is a compaction mode for each variant that truly rewrites at least all of the data for the encrypted CF if the admin, or a key management tool, really wants that to happen.

Meanwhile, we also introduce a new configuration option for encrypting the WAL. Even though WALs are transient, it is necessary to encrypt the WALEdits to avoid circumventing HFile protections for encrypted column families. Because the crypto framework introduces crypto codecs that share an interface with compression codecs, we use existing SequenceFile support for compression so require no special provisions for metadata.

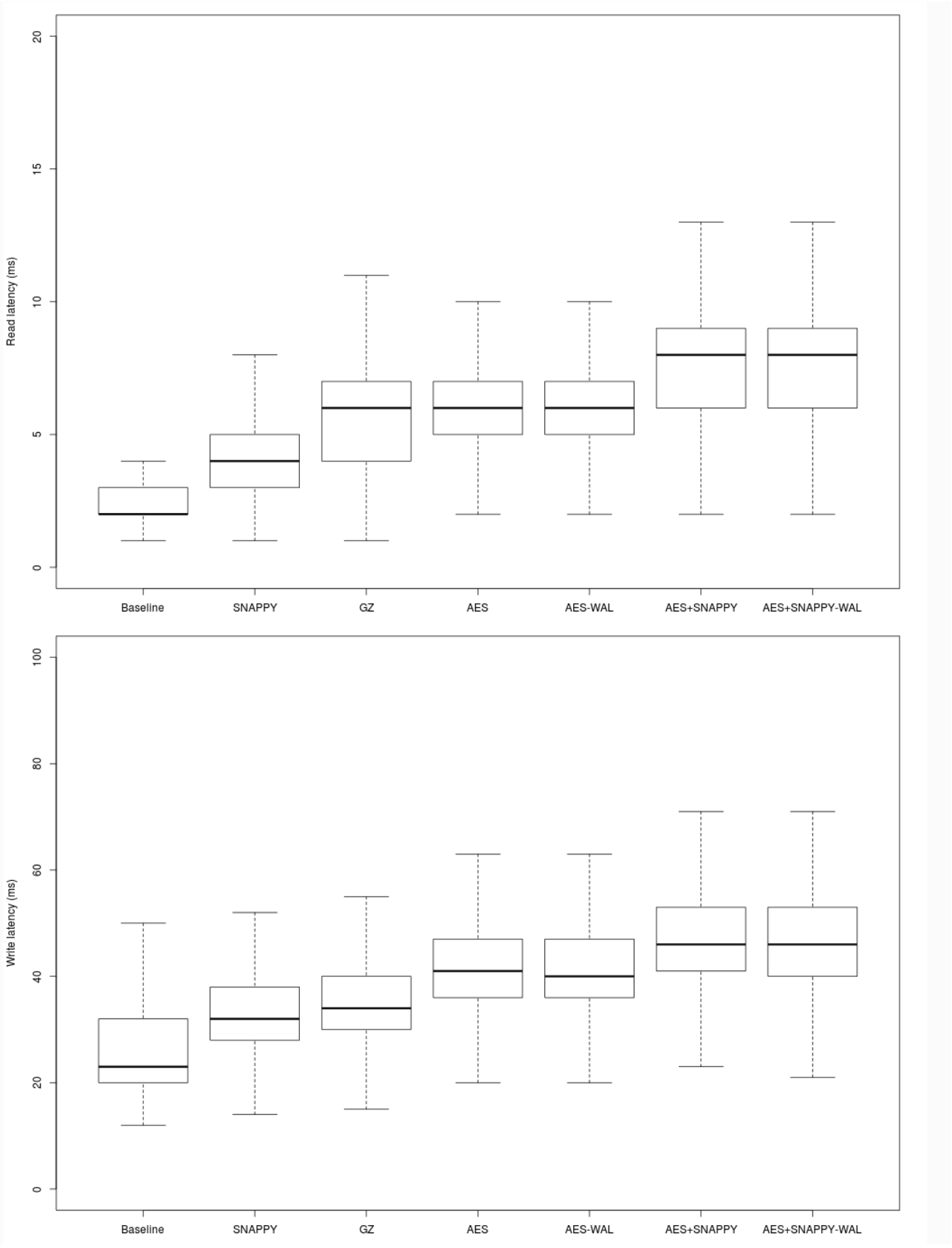
The crypto codec framework includes an AES codec optimized for AES-NI hardware acceleration, available on any modern Intel CPU. We find that, using this AES-NI codec, introduction of symmetric block encryption to the HFile read and write code paths introduces an overhead roughly on par with GZIP compression for reads, and half that as for writes. Further performance tests and code optimizations are planned. We use AES in CTR mode to enable future work on parallel decryption of HFile blocks using multiple hardware threads.

Currently, read throughput can be improved slightly by compressing data with SNAPPY first:



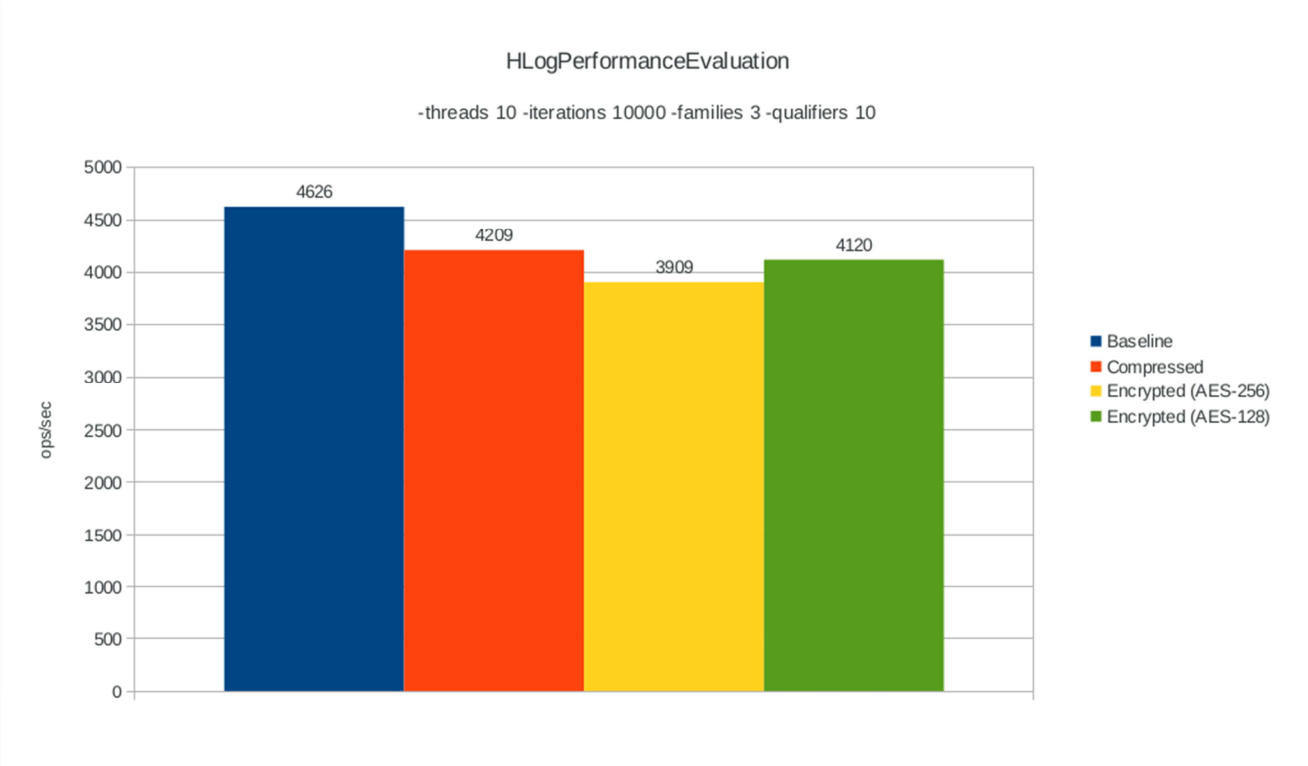


Although, enabling both compression and encryption codecs on an HFile does introduce additional latency, as measured by a new large unit test TestEncryptionLoadTest, based on TestMiniClusterLoad:


Read latency (ms)
0
5
10
15
20
Baseline
SNAPPY
GZ
AES
AES-WAL
AES+SNAPPY
AES+SNAPPY-WAL
Write latency (ms)
0
20
40
60
80
100
Baseline
SNAPPY
GZ
AES
AES-WAL
AES+SNAPPY
AES+SNAPPY-WAL

Interestingly, these results also do not show any significant end to end latency improvement if WAL encryption is disabled.

Future work is planned on optimizing WAL encryption. We currently encrypt the entire WAL. We have experimented with encrypting on a per WALedit basis instead of using SequenceFile record compression but the results demonstrate poorer relative performance. Encrypting the entire WAL with AES-256 (at the full 14 rounds) introduces a ~15% throughput drop as measured by the HLogPerformanceEvaluation microbenchmark. However, encrypting the entire WAL with AES-128 (at the full 10 rounds) is competitive with the ~9% overhead observed when WAL compression is enabled.



We would like to bring the impact of encryption lower still, because the overheads of encryption and compression, if enabled, are cumulative. It might be nice to support both at the same time with reasonable performance. We can consider employing a reduced round AES-128 variant after careful analysis of the performance improvement versus added risk. Attacks on reduced round AES have been studied and detailed in the literature. We are aware of successful medium complexity related key attacks. Such an attack would not be possible here because an attacker (a user with TABLE ADMIN but not GLOBAL ADMIN) can only change the CF keys for their tables, they cannot control the master key material. The CF keys on tables beyond the attacker's control are encrypted with the master key and are therefore not available for manipulation in a chosen key attack. The attacker can only mount a key recovery attack on the master key using their CF keys as plaintext. The best known key recovery attack has a complexity of $2^{124.9}$ against 8 round AES-128, making even this infeasible in practice, and we would of course continue to encrypt CF keys with the master key using full strength 14 round AES-256.