Benchmarking Encrypted Data Storage in HBase and Cassandra with YCSB

Tim Waage, Lena Wiese

Institute of Computer Science
University of Göttingen
Goldschmidtstrasse 7
37077 Göttingen
Germany
{tim.waage,lena.wiese}@uni-goettingen.de

Abstract. Using cloud storage servers to manage large amounts of data has gained increased interest due to their advantages (like availability and scalability). A major disadvantage of cloud storage providers, however, is their lack of security features. In this article we analyze a cloud storage setting where confidentiality of outsourced data is maintained by letting the client encrypt all data records before sending them to the cloud storage. Our main focus is on benchmarking and quantifying the performance loss that such a cloud storage system incurs due to encrypted storage. We present results based on a modification of the Yahoo! Cloud Serving Benchmark using the AES implementation of the Bouncy Castle Java Cryptography Provider for the encryption and decryption steps. The results show that for single read and write operations the performance loss is acceptable (even for stronger encryption with 256 bit keylength) while for range scans the impact can be quite severe.

Keywords: YCSB, Encrypted Data Storage, Benchmarking, Cassandra, HBase

1 Introduction

Cloud computing has been devised as an alternative to intra-company data processing by outsourcing administration tasks to cloud service providers. Individual users take advantage of external processing resources, too, to store and share data. Cloud service providers might in particular offer storage space which can be booked flexibly on demand. The basic application setting we consider in this paper is using cloud storage servers for larger data records to relieve data owners from the storage burden.

One of the main obstacles of a wider adoption of cloud storage is its lack of security. Usually no functionality is provided to ensure confidentiality of the data stored in the cloud. Confidentiality can be endangered by administrators of the cloud storage provider, by malicious intruders or even by other users of the data store. Our main security target is to ensure confidentiality of the externally

stored data record against any illegitimate read accesses. That is, we assume a semihonest behavior of attackers that are curious about the content of the stored records. To achieve protection against these attackers, we provide the data owner with a symmetric cryptographic key that is used to encrypt any outgoing data and decryption only takes place at the data owner site. This symmetric key must be secured at the data owner site.

We assume that attackers do not further interfere with the stored data. In particular, the data store should return the correct records to a client request. Malicious attacks like manipulation of the encrypted data is hence in general still possible – but provided that the employed encryption scheme is secure, manipulation will result in arbitrary distortion of data such that they no longer can be decrypted.

In this paper, we focus on benchmarking this form of encrypted data storage with one symmetric encryption key. The performance results will be obtained with the Yahoo! Cloud Serving Benchmark (YCSB; see [1]). Encryption will be executed by the Java Cryptography Provider offered by the Legion of the Bouncy Castle [2]. The benchmarking results presented in this paper are part of a larger environment where index-based keyword search will be supported as follows: Having a short synopsis of a longer data record (a summary, or simply a list of tags or keywords for the content of the document), and the large record itself, we produce a small clear text index (which can easily be stored at the owner's computer); however the actual record will be stored remotely in a data store hosted by some cloud storage provider as shown in Figure 1.

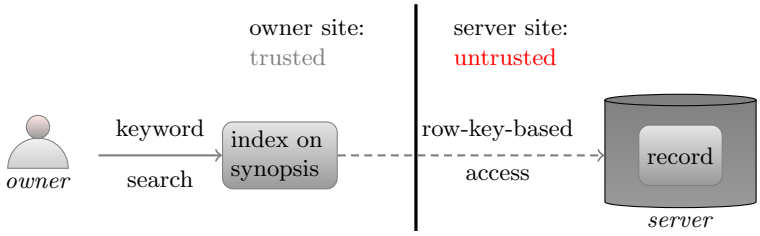


Fig. 1. Data outsourcing with keyword search

In order to comply with the confidentiality requirement, all data leaving the owner site (for being stored at the cloud server) are encrypted. The client is responsible for keeping both the encryption key and the keyword index secret at his own trusted site as shown in Figure 2. After retrieving data from the cloud server, the client has to decrypt the data before processing them.

In this paper we make the following contributions:

– We compare the performance of two data stores (HBase and Cassandra) in both the encrypted and the non-encrypted case in a single-server environment.

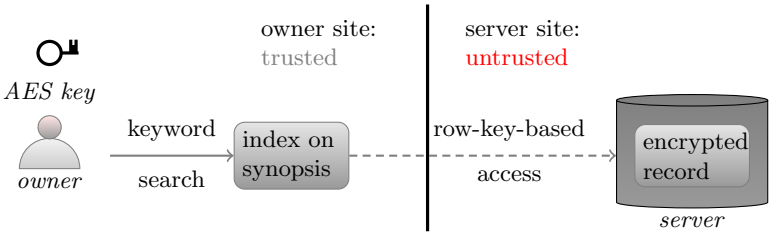


Fig. 2. Data outsourcing with encryption and confidential index

- We present three tests of the data stores using a modification of the Yahoo! Cloud Serving Benchmark (YCSB) to compare the throughput and latency of the system with and without the encryption system.
- The tests executed analyze the two systems when (1) increasing the number of parallel worker threads, (2) increasing the target throughput and (3) increasing the total number of operations.

1.1 Organization of the Article

This article is organized as follows: Section 2 describes the main features of the tested data stores. Section 3 introduces the Bouncy Castle Java Cryptography Provider as well as the encryption primitives used in our settings. In Section 4 the general architecture of the Yahoo! Cloud Serving Benchmark and its different workloads are presented. Our modifications to the original YCSB are described in Section 5. Section 6 describes our three test cases and discusses the outcomes. Related work is surveyed in Section 7. Section 8 concludes the paper with a summary of our results and briefly discusses options for future work.

2 The Data Stores

The Yahoo! Cloud Serving benchmark can be considered as a test environment for key-value stores because accesses are only done by row key (that is, the key identifying an individual record in the data store). Key-value stores have been conjectured to be good candidates when storing big data. Most of them were created as part of Web 2.0 services. From an abstract point of view, they expose a very simple interface consisting of the three operations: put, get and delete.

Because the retrieval and storage operations of a key-value store are so simple, most key-value stores do not have to bother with transactions, isolation of concurrent accesses or complex combinations (like joins) of values. However, in a multi-master setting, several clients can concurrently write to different servers and in this case the key-value store has to provide a strategy for conflict resolution (for example, using vector clocks).

Where key-value stores excel is their distribution features. Basing on the background of research done for distributed hash tables, data in key-value stores

can be flexibly partitioned and automatically replicated among several independent (shared-nothing) servers. Hence, key-value stores are popular when it comes to fast and parallel processing of large amounts of non-interlinked data records.

Typically for the so-called NoSQL data stores most key-value stores also share a lack of certain security features, for example they do not offer user, role or access rights management. It is assumed that a frontend (for example, the cloud storage interface) offers the appropriate authentication and authorization features before letting users access the storage backend. Hence, there is no extra level of protection and a break-in in the frontend will leave the entire data store more vulnerable to attacks than a database product offering more sophisticated user management. A certain level of security could still be achieved by storing the data in an encrypted form, but both systems do not have any native mechanisms for providing encryption. The scope of this article is the examination of the performance impact, if such an additional encryption/decryption step is added.

In our work we focus on Cassandra [3, 4] and HBase [5, 6], which share a couple of commonalities. Both can be considered as key-value stores as well as column family stores (sometimes also called table stores, extensible record stores or wide column stores). They offer high availability by distributing and replicating data. Both are implemented in the Java programming language. A common feature of these systems is their storage organization in *log-structured merge trees* which implement an *append-only* storage: in-memory tables store the most recent writes to the system; once these tables are flushed to disk, no data in these tables will ever be modified again (they are *immutable*). The only modification that is taking place is merging several flushed tables into larger ones – this process is called *compaction*.

Despite the similarities of Cassandra and HBase there are fundamental differences in their design. Cassandra follows a strictly symmetric peer-to-peer concept using the Gossip protocol for coordination purposes, while HBase relies on a master-slave structure. Thus Cassandra runs in a single Java process per node and can make use of the local file system, while HBase needs the database process per node itself, a properly configured Hadoop Distributed File System (HDFS) and a Zookeeper system in order to manage different HBase processes. That means concerning the CAP Theorem [7, 8] Cassandra offers availability and partition-tolerance, while HBase is designed for consistency and availability.

3 Java Cryptography API and the Bouncy Castle Provider

In the Java programming language, the Java Cryptography API (JCA) provides the basic architecture when using cryptographic algorithms; it specifies the interfaces with which cryptographic algorithms can be used in a Java program. The actual implementations are then provided by so-called cryptography providers. The Legion of the Bouncy Castle package for Java implements of most of the interfaces defined in the JCA – in particular, it implements the Java Cryptography Provider class and hence smoothly integrates with the JCA. The Bouncy

Castle provider offers several encryption and decryption routines including the Advanced Encryption Standard (AES). AES is a symmetric encryption algorithm. It uses the Rijndael cipher with a block size of 128 bit. Possible key sizes are 128, 192 and 256 bit. As it turned out in various tests that we conducted the Legion of the Bouncy Castle package is the fastest choice of all cryptography providers that offer at least the same flexibility as the standard Java Cryptography Extension (SunJCE). In particular it also provides AES encryption in cipher feedback mode (CFB), which is important for our future work (e.g. when implementing searchable encryption with flexible word lengths [9]).

In our experiments we employ AES in cipher block chaining (CBC) mode using PKCS7 padding – as provided by the Bouncy Castle provider; other modes can be configured on demand. CBC mode produces a chain of ciphertext blocks by interlinking each block with its predecessor in an XOR operation. More precisely, the initial plain text block is XORed with an initialization vector; the output is encrypted with the AES key. Subsequent plain text blocks are XORed with the encrypted text from the previous block; the output is encrypted with the key. Block cipher algorithms require their input to be a multiple of the block size. Since this is not the case for the lengths of keys, field names and values in YCSB's output (see section 5 for details), we use PKCS7 padding [10], which allows any block size from 2 to 255 bytes.

For decryption, the initial cipher text block is decrypted with the key, and the output is XORed with the initialization vector. Subsequent cipher text blocks are first decrypted with the key; the output is then XORed with the cipher text from the previous block to result in a plaintext block.

4 YCSB Benchmark

The Yahoo! Cloud Serving Benchmark (YCSB) is a framework for evaluating the performance of different key-value stores. It comes with the *Core Package*, consisting of six workloads. Each workload is a specific mix of read and write operations, using different data sizes and request/popularity distributions in order to simulate certain scenarios. Besides those provided workloads there are two ways of creating own custom workloads. On the one hand one can modify the existing Core Package by changing the values in the workload's parameter file. On the other hand it is possible to extend the com.yahoo.ycsb.Workload class and write new code for interacting with different databases. In this work we modify the provided parameter files and introduce some new parameters in order to deal with the aspects of encryption. Thus, in contrast to the original work we added parameters for example for turning the encryption on/off and for specifying encryption key lengths.

A workload in YCSB runs in two phases. Phase one is the *loading phase* that fills the database with records. Every record has a name, also referred to as the record's primary key, which is a string like "userxxx", where xxx is a random number. Such a record contains a set of ⟨field name, value⟩ pairs. Thereby field name is a string like "fieldxxx", where xxx again is a random number and value

is a random String. For our tests we leave the defaults unchanged, which leads to 100 records, each containing ten ⟨field name, value⟩-pairs, where every value has a length of 100 bytes. While YCSB has been used for a performance evaluation of Cassandra [11], there are no workloads on real-world datasets readily available (in both Cassandra and HBase dialects) that would allow for a direct performance comparison. The second phase is the *transaction phase* in which the workload's specified insert/update/read/scan operations are actually executed.

In this work we use the six workloads of the Core Package, because they already give a good impression of the performance aspects we want to examine. Their characteristics are as follows:

- Workload A is a 50/50 mix of read and write operations.
- Workload B focuses heavily on reading and has only a small amount of writing.
- Workload C even only utilizes read operations.
- Workload D inserts a specified number of records and focuses on reading the most recently added ones preferably.
- Workload E is the only core workload not dealing with individual records, but with short ranges of records.
- Workload F combines read and write operations in the way that records are read, modified and then written back into the database.

[12] gives examples for real world applications.

As it turned out, in the majority of examined cases the results for workloads A, B, C, D and F are always very similar in terms of overall throughput, as well as read and write latency, when the recommended order of workloads is obeyed. Only workload E appears to have different characteristics. Thus for the sake of simplicity and scope of this article, we narrow the analysis of the results down to workload A (as representative of the workloads A, B, C, D and F) for making statements on single read/write operations, as well as workload E for making statements on range queries.

5 Running YCSB on Encrypted Data

The necessary changes to run YCSB on encrypted data are as follows.

We added a new class that encapsulates all methods for encrypting and decrypting byte arrays and Strings. As mentioned earlier that class utilizes the Bouncy Castle Provider. One symmetric AES key is generated that is used throughout each run of the workload for encryption and decryption of all stored and retrieved records. A Cipher object is created (and initialized) that executes the block-wise encryption or decryption using CBC mode and PKCS7-padding by calling `Cipher.getInstance("AES/CBC/PKCS7Padding", "BC");`

For the load phase we proceed as follows. After the generation of random field values, we add an encryption step: each generated record key ("userxxx"), field name ("fieldxxx") and field value gets encrypted and collected in a HashMap. A record then consists of a set of such ⟨encrypted field name, encrypted value⟩

pairs. In the load phase a set of records is finally stored (batch loaded) into the database. In a similar manner, before an individual record is stored in the database (for an insertion of a new value or an update of an existing value during the transaction phase), an encryption step is added. Thus for every point in time the database contains only encrypted data, except for the corresponding timestamps. Since in CBC mode the length of the cipher-text has to be a multiple of the key length, the database gets slightly bigger compared to its size storing the same data unencrypted.

For retrieving records from the database the process is straight forward: the requested record's primary key gets encrypted and is then used in order to find the corresponding ⟨encrypted field name, encrypted value⟩ pairs for decryption.

6 Results

We performed three tests in order to examine the impact of different parameters on the performance of Cassandra and HBase. In every scenario measurements are taken with no encryption, as well as with AES encryption using key lengths of 128, 192 and 256 bits. For every test an analysis of the overall throughput, average read and write latencies is given for workload A (single read/write operations) and workload E (range queries). The latency overhead in the read case consists of the time required to encrypt the record's primary key for lookup purposes, retrieve the ciphertexts of increased length from the database and the additional decryption step of field name and value. The latency overhead in the write case consists of the encryption step as well as the additional time required to store the encrypted data of increased length (compared to non-encrypted data). All tests measure individual read/scan/insert/update calls to the database. We conducted our measurements in a single-server environment, which means the database's different replication strategies can not have any distorting effect. The database server is equipped with an Intel Core i7-4600U CPU (Haswell) @ 2.10GHz (4 Cores), 8GB DDR3 RAM and a Samsung PM851 256GB SSD.

6.1 Test 1: Increasing Number of Parallel Worker Threads

Per default the YCSB client uses a single worker thread for communicating with the database. However it is possible to specify an arbitrary number of threads in order to put more load against the database and to simulate how several clients interact with the database simultaneously. Figure 3 (Cassandra) and 4 (HBase) show the performance impact, if the number of threads is increased to two, four and finally eight threads. However the total number of operations remains fixed at 30000, since all threads share the same operation counter.

As can be seen the number of worker threads has no major impact, neither on the overall throughput, nor on read and write latencies, whether encryption is enabled or not. The results are basically constant. An exception appears to be

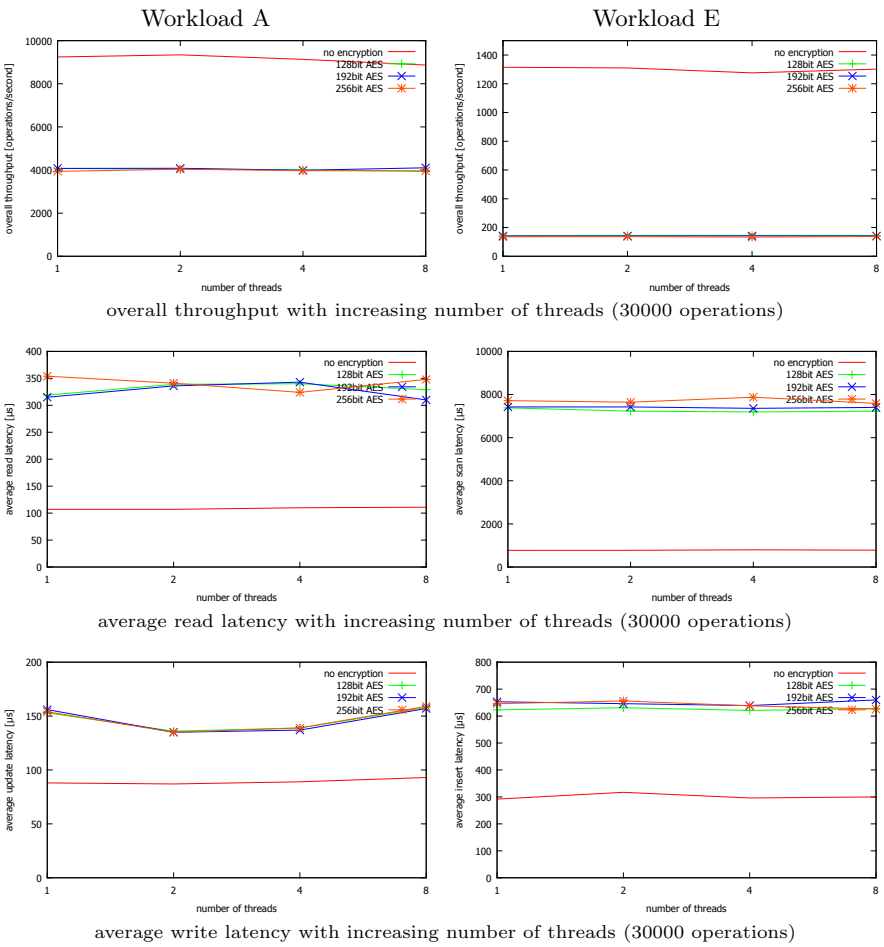
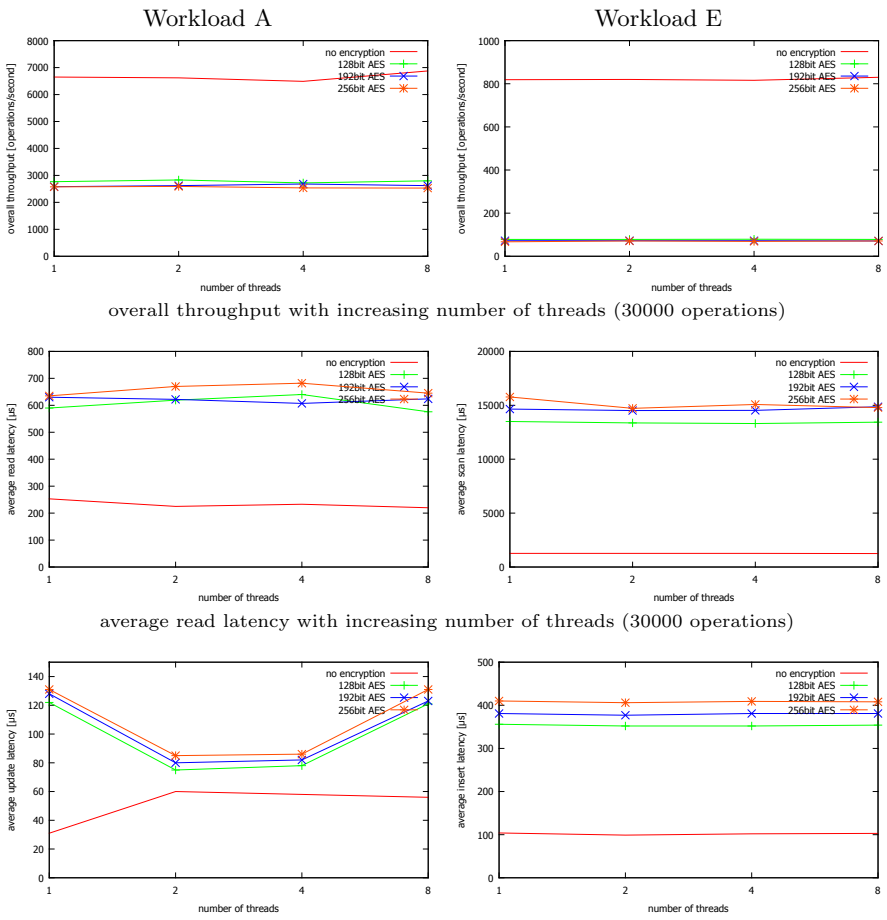


Fig. 3. Test 1: Cassandra - Increasing the number of parallel worker threads

the read latency for single read/write operations, being slightly lower when performing on two or four threads. The AES encryption key length has no significant impact in all cases.

6.2 Test 2: Increasing the Target Number of Operations per Second

Usually the YCSB client tries to do as many operations as possible, but the number of operations per second can be throttled in order to perform latency tests. If a target parameter is specified the YCSB client executes 1-millisecond sleep operations to reduce the throughput accurately until the target throughput is reached. Different target throughputs may result in better or worse latency results.



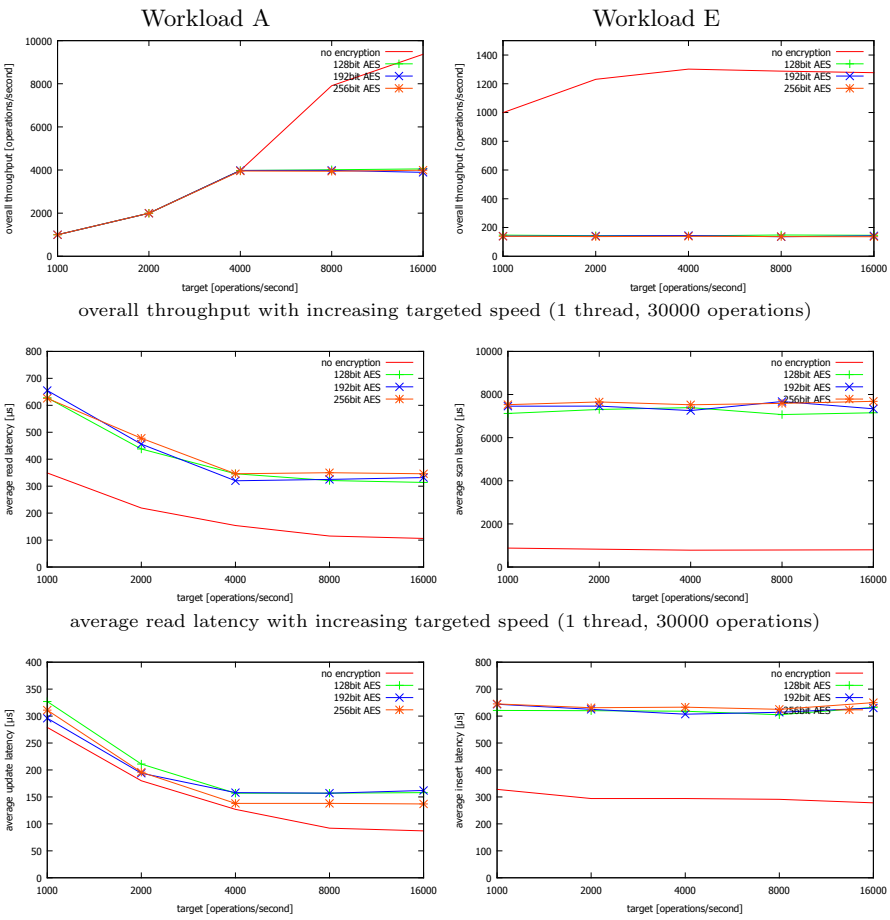
overall throughput with increasing number of threads (30000 operations)

average read latency with increasing number of threads (30000 operations)

average write latency with increasing number of threads (30000 operations)

Fig. 4. Test 1: HBase - Increasing the number of parallel worker threads

Figure 5 and 6 show that enabling encryption leads much earlier to a point, where the databases are saturated and the overall throughput stops increasing. When performing range queries this point is reached very fast compared to working with single read/write operations. For both, read- and write-latencies, can be stated that increasing the targeted number of operations results in lower latencies up to a certain level, when single read/write operations are performed, but no impact can be observed on range queries. Here only HBase shows a sudden increase of write latency, when encryption is enabled and the targeted number of operations exceeds a certain level. As already observed in the first test the length of the AES key does not make any significant difference.



overall throughput with increasing targeted speed (1 thread, 30000 operations)

average read latency with increasing targeted speed (1 thread, 30000 operations)

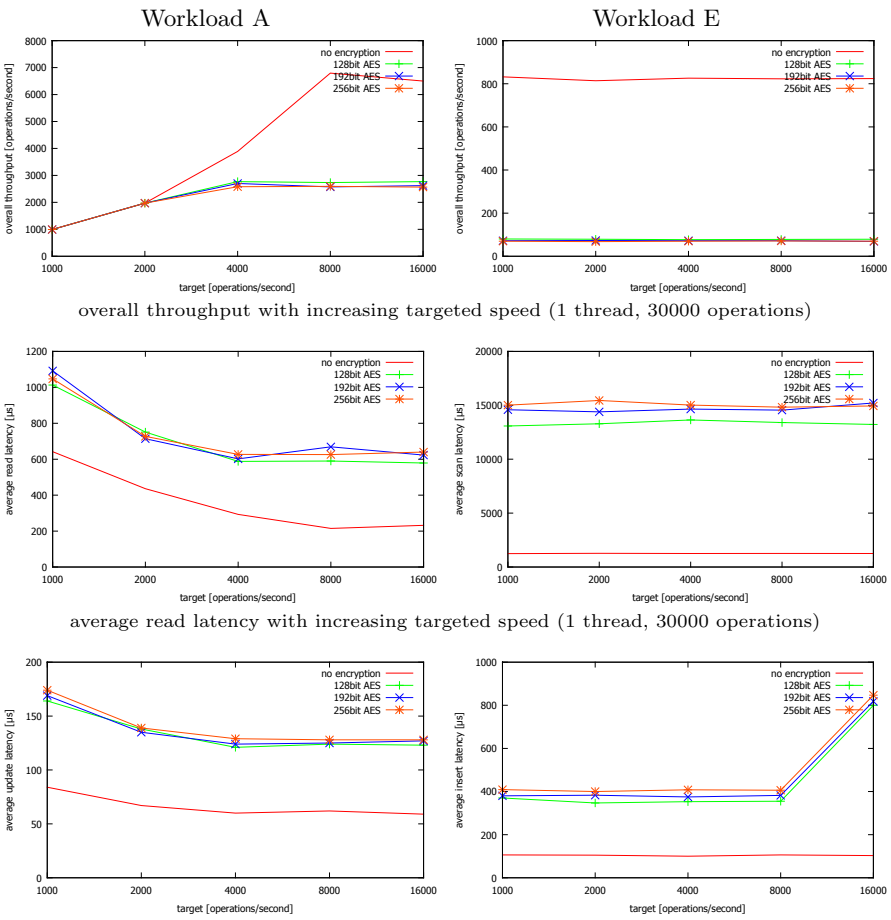
average write latency with increasing targeted speed (1 thread, 30000 operations)

Fig. 5. Test 2: Cassandra - Throttle the target number of operations per second

6.3 Test 3: Increasing the Number of Operations

The total number of operations per transaction phase can be increased in order to examine the databases performance when it comes to larger amounts of operations.

The results are presented in Figure 7 and 8. In both databases increasing the number of operations leads to a higher overall throughput up to a certain level for unencrypted and encrypted operation, except for range queries, when encryption is enabled. For single read/write operations a higher number of operations results in less read/write latency. This effect is even stronger, when encryption is enabled. The same is true for write latencies, when it comes to range queries, while read latencies remain on a constant level.



overall throughput with increasing targeted speed (1 thread, 30000 operations)

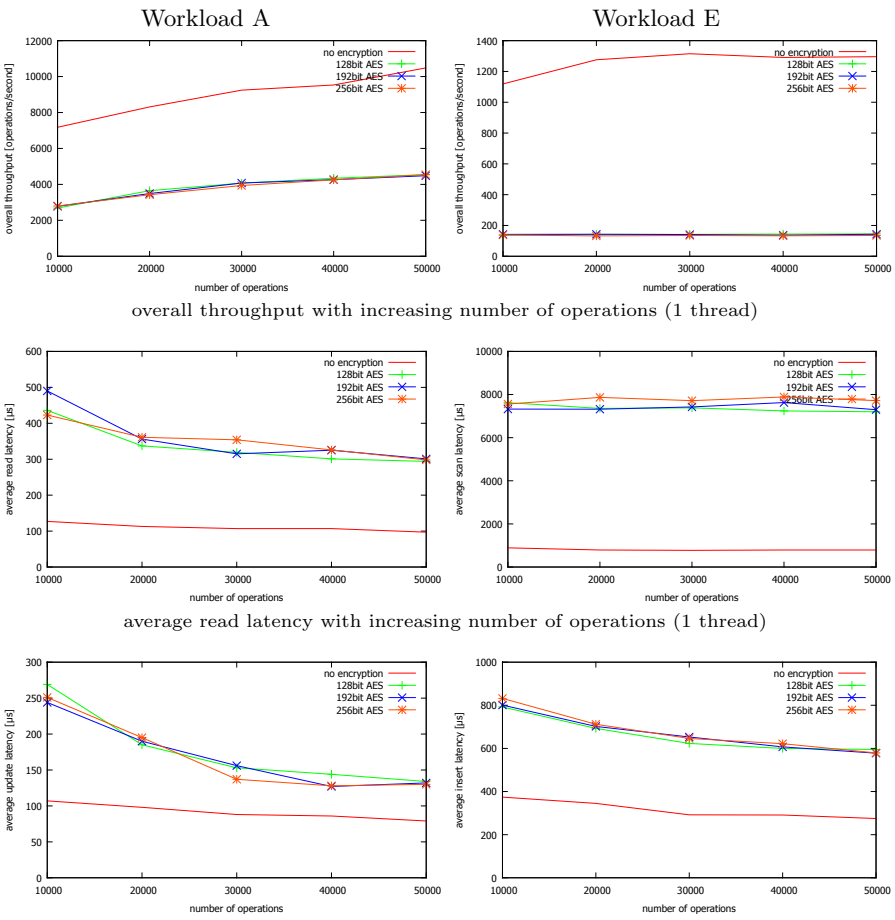
average read latency with increasing targeted speed (1 thread, 30000 operations)

average write latency with increasing targeted speed (1 thread, 30000 operations)

Fig. 6. Test 2: HBase - Throttle the target number of operations per second

6.4 General Observations

It can be stated that turning on encryption reduces the overall throughput down to approximately 40% on average for single read/write operations and only 10% when it comes to range scans. The chosen AES key length has almost no impact, the only noticeable difference occurs in read/write latencies in range queries. With encryption enabled, read latencies are 2-3x higher on average for single read/write operations, even up to 10-12x higher for range queries. However this effect is not so strong for write latencies, which are only 2x higher in every scenario, except for range queries with a higher number of operations, where they can be up to 4x higher. Thus it can be stated that enabling encryption increases latency for read operations more than for write operations.



overall throughput with increasing number of operations (1 thread)

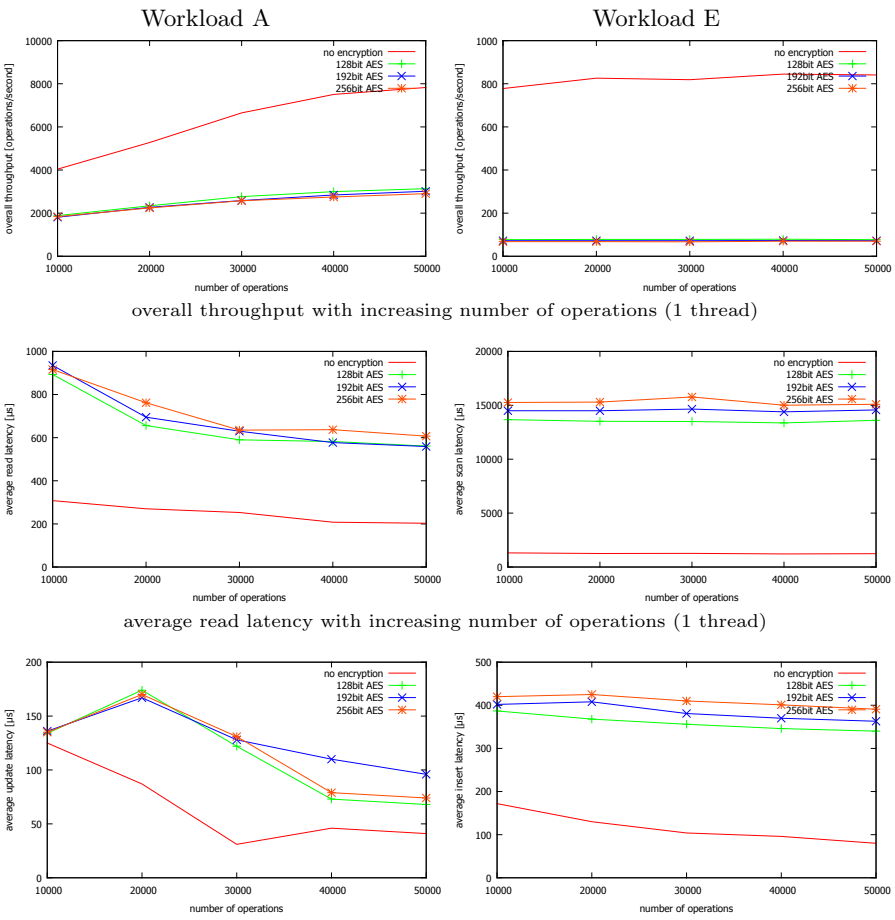
average read latency with increasing number of operations (1 thread)

average write latency with increasing number of operations (1 thread)

Fig. 7. Test 3: Cassandra - Increasing the total number of operations

However the number of parallel worker threads seems to have no consequences for the databases performances. Targeting a higher number of operations per second slightly decreases read/write latencies to a certain point in all cases, except for write latencies in range queries with encryption enabled. Comparing the two databases Cassandra and HBase themselves, it can be stated that Casandra's overall throughput is generally higher in every scenario, in most cases by approximately 50% on average. As mentioned earlier enabling encryption slows down both databases nearly equally. Concerning read/write latencies, it can be observed that on the one hand Cassandra's latencies are lower for read operations. On the other hand latencies of HBase are lower when it comes to write operations. This is true for every tested scenario.

The effect that latency is reduced (when increasing the target throughput or the operation count) has also been observed in [13]. Their analysis revealed



overall throughput with increasing number of operations (1 thread)

average read latency with increasing number of operations (1 thread)

average write latency with increasing number of operations (1 thread)

Fig. 8. Test 3: HBase - Increasing the total number of operations

that this effect is owed to increased amount of compactions (see Section 2) for smaller operation count.

7 Related Work

The original YCSB description can be found in [1]. There, the different workloads are described in detail. The article presents benchmark results for HBase and Cassandra (among others) – with the overall result that Cassandra had the best throughput. Elasticity of the systems is tested by adding new servers at runtime.

The so-called YCSB++ benchmark is presented in [13] and used to analyze HBase and Accumulo. The benchmark extends the original YCSB benchmark for example to support real multi-client accesses instead of only providing the

option to run multiple worker threads. Its focus lies on measuring consistency; from a security perspective, it analyzes the performance of Accumulos cell-level access control lists for read accesses. No encryption is considered there.

[14] pinpoint vulnerabilities in the Cassandra gossip protocol that may reduce availability by introducing zombie or ghost nodes. [15] analyze other security weaknesses in Cassandra. Hence appropriate security measures should be taken already at the client side in order to not expose sensitive data to the database server that might fall victim to attacks due to weaknesses in the database implementation.

8 Conclusion and Future Work

We examined the performance impact of adding an encryption/decryption step to the two popular key-value stores Cassandra and HBase using YCSB by presenting the impact of different runtime parameters. We showed that enabling encryption slows the entire system down to 10-40% of its original throughput capacity. The consequences for the system's read/write latency reach from almost no impact (e.g. Cassandra's write latency for single read/write operations) to a major increase of latency up to 1000-1500% (e.g. read latencies in general for range queries).

Conclusively, it can be said that there is indeed a price to pay in terms of increased latency when applying encryption and decryption to data in the YCSB workloads. However with our settings (with Bouncy Castle as the Java Cryptography Provider) using a stronger AES encryption with longer key length does not incur more overhead than using weaker encryption with shorter key length.

Future work can extend these results as follows. We conducted our experiments in a single-server environment in order to avoid any impact of the different replication strategies used by Cassandra and HBase. However since both are designed to be used in multi-server environments as well, further tests are necessary. Moreover YCSB offers a lot of additional options that can be used for further analysis. Interesting is a modification of the value field length as well as of the different read/scan/update/insert proportions. Furthermore there are many more cloud database architectures besides Cassandra and HBase with different design decisions available for testing. Another important aspect is the impact of different cryptographic systems, both symmetric (for example Blowfish) or asymmetric (for example RSA). As another field of research, fine-grained cryptographic access control can be implemented by more advanced key management (for example, using key hierarchies for different users [16]).

With regard to support for keyword search, an appropriate indexing program (for example, SOLR or ElasticSearch) shall be employed to obtain the confidential keyword index on the client site. Further benchmarking can then quantify the performance loss for this kind of advanced search-based interaction (than merely row-key-based access).

Acknowledgements

This work was partially funded by the DFG under grant number WI 4086/2-1.

References

1. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: Proceedings of the 1st ACM symposium on Cloud computing, ACM (2010) 143–154
2. The Legion of the Bouncy Castle: `http://bouncycastle.org/`.
3. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review **44**(2) (2010) 35–40
4. The Apache Software Foundation: `http://cassandra.apache.org`.
5. Borthakur, D., Gray, J., Sarma, J.S., Muthukkaruppan, K., Spiegelberg, N., Kuang, H., Ranganathan, K., Molkov, D., Menon, A., Rash, S., et al.: Apache hadoop goes realtime at facebook. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of data, ACM (2011) 1071–1080
6. The Apache Software Foundation: `http://hbase.apache.org`.
7. Brewer, E.A.: Towards robust distributed systems. In: PODC. (2000) 7
8. Brewer, E.: A certain freedom: thoughts on the cap theorem. In: Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing, ACM (2010) 335–335
9. Song, D.X., Wagner, D., Perrig, A.: Practical techniques for searches on encrypted data. In: Security and Privacy, 2000. S&P 2000. Proceedings. 2000 IEEE Symposium on, IEEE (2000) 44–55
10. Kaliski, B.: Rfc 2315: Pkcs# 7: Cryptographic message syntax. Request for Comments (RFC) **2315** (1998)
11. Dede, E., Sendir, B., Kuzlu, P., Hartog, J., Govindaraju, M.: An evaluation of cassandra for hadoop. In: Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on, IEEE (2013) 494–501
12. Cooper, B.F.: `https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads/`.
13. Patil, S., Polte, M., Ren, K., Tantisiriroj, W., Xiao, L., López, J., Gibson, G., Fuchs, A., Rinaldi, B.: Ycsb++: benchmarking and performance debugging advanced features in scalable table stores. In: Proceedings of the 2nd ACM Symposium on Cloud Computing, ACM (2011) 9:1–9:14
14. Aniello, L., Bonomi, S., Breno, M., Baldoni, R.: Assessing data availability of cassandra in the presence of non-accurate membership. In: Proceedings of the 2nd International Workshop on Dependability Issues in Cloud Computing, ACM (2013) 2:1–2:6
15. Okman, L., Gal-Oz, N., Gonen, Y., Gudes, E., Abramov, J.: Security issues in nosql databases. In: 10th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), IEEE (2011) 541–547
16. Zhu, Y., Ahn, G.J., Hu, H., Ma, D., Wang, S.: Role-based cryptosystem: A new cryptographic RBAC system based on role-key hierarchy. IEEE Transactions on Information Forensics and Security **8**(12) (Dec 2013) 2138–2153