CloudDBGuard: Enabling Sorting and Searching on Encrypted Data in NoSQL Cloud Databases

Tim Waage and Lena Wiese

Institute of Computer Science, University of Göttingen
Goldschmidtstraße 7, 37077 Göttingen
{tim.waage|wiese}@cs.uni-goettingen.de

Abstract. Efficient and secure data management in the cloud is a topic of high relevance. Cloud databases play an important role – in particular, because they can offer various options for processing large amounts of data. In this article we discuss several practical aspects of applying property-preserving encryption (PPE) to data stored in wide column stores – an encryption type that allows for secure sorting and searching on encrypted data. We present the CloudDBGuard framework that comprises implementations of several PPE schemes and takes care of the necessary metadata management. CloudDBGuard also makes progress in terms of query interface standardization by providing a unified API that supports both Apache Cassandra and Apache HBase.

1 Introduction

Wide column stores (WCSs) comprise a certain group of non-relational databases ("NoSQL" databases; see [22] for a comprehensive survey). Because of their flexible and efficient behavior, WCSs are widely used by cloud storage providers. This raises concerns how confidential data can be protected from a curious cloud storage provider or other attacks by unknown third parties. So far, WCSs do not offer any means to ensure the protection of confidential data from unauthorized access. Our main goal was hence to leverage secure and user-adaptable property-preserving encrypted storage of data in this type of databases. Our prior work [20] focused on developing a framework to enable the seamless integration of property-preserving encryption with the WCSs HBase and Cassandra – see [18] for the implementation details. Here we report on various extensions of our framework that improve runtime properties and functionality:

- For some encryption schemes we were able to improve their runtime further.
- Moreover, we were able to hide the complexity of the used encryption strategies behind an easy-to-learn API (application programming interface), which simplifies the practical usage of the framework.
- Furthermore, the user has to keep just one master password in mind instead of having to deal with several cryptographic passwords.
- We report on a combined benchmarking that applies several of these encryption schemes in one query.

The article is organized as follows. Section 2 provides the necessary background on wide column stores, property-preserving encryption and onion layer encryption. Section 3 gives an in-depth description of the components of our framework. Section 4 reports on the runtime experiments with 10 benchmark queries that combine several property-preserving encryption types. Section 5 concludes this article with a discussion and suggestions for future work.

2 Background and Related Work

2.1 Motivating Example

An application scenario for the CloudDBGuard framework is a mail server that stores confidential emails of several users. While the users are relieved from storing all emails locally, they still require management features from the mail server. PPE enables the mail server to manage the emails efficiently while preserving their confidentiality. For example, a user wants to search for emails containing a certain search word – is enabled by searchable encryption. Furthermore, the user requires the server to sort the emails by date in order to retrieve only the most recent ones or the emails from a specified time interval – this is enabled by property-preserving encryption. In order to benchmark this scenario (in Section 4), the widely-used Enron corpus [12] served as our data set. The Enron dataset comprises e-mails of 150 employees of a company. Our test queries combine equality tests (to find a sender exactly), range queries (to find emails in a certain timespan) and word search (on the email body).

2.2 The Data Model of Wide Column Stores

Wide column stores (WCSs) are inspired by Googles BigTable architecture [6]. Several open source databases rely on a very similar data model; yet, they differ in implementation details and access methods (like query languages or APIs). Our framework supports Apache Cassandra [13] and Apache HBase [5].

WCSs use tables, rows and columns like traditional relational (SQL-based) databases. However, the fundamental difference is that columns are created for each row independently instead of being predefined by the table structure (that is, database schema). Every row has at least one mandatory column containing its **identifier** (commonly referred to as "row key"). The identifier of a row has to be unique for the whole table and cannot be used by another row. Rows are maintained in lexicographic order by their identifier. WCSs support data **partitioning** in distributed systems. Ranges of the row identifiers serve as units of distribution such that data are partitioned row-wise (that is, horizontally). Due to this range-based partitioning, similar row identifiers are always kept physically close together in the same partition.

To avoid a fixed database schema, WCSs use an internal data format consisting of key-value-pairs. The key part has several components: the so-called keyspace, a table name, a column name and the row identifier. One of these

components is a timestamp, enabling the database to maintain an automatic version control, which can be operated in two ways: either by setting a maximum number of versions to keep, or by specifying a "time-to-live" (TTL) after which data items are to be deleted. More formally WCSs can be considered sparse, distributed, multidimensional maps (see [6]) of the form

(keyspace, table, column, row identifier, timestamp)→value.

The WCS data model is hence different from traditional column stores. The internal workings of WCSs are described in detail in Chapter 8 of [22].

2.3 Property-preserving encryption

Property-preserving encryption (PPE) retains certain properties of the plaintext (like order of numerical values) on the ciphertext – or it relies on additional index structures on encrypted values (to support efficient search on encrypted data). The types of PPE relevant for this work are deterministic encryption (DET), order-preserving encryption (OPE) and searchable encryption (SE):

- **DET.** The purpose of DET is enabling the database server to check for equality by mapping identical plaintexts to identical ciphertexts.
- **OPE.** The purpose of OPE is enabling a server to learn the relative order of data elements without revealing their exact values. OPE encrypts two elements p_1, p_2 of a domain D in a such way that $p_1 \leq p_2 \Rightarrow Enc(p_1) \leq Enc(p_2)$ for all $p \in D$. Thus, its use cases are sorting and range queries over encrypted data. A lot of OPE schemes have been proposed with different strategies to map a plaintext to a ciphertext domain (see [2, 23, 11, 7]).
- **SE.** The purpose of SE is enabling a server to search over encrypted data without revealing plaintext data. Most SE schemes use indexes (see [8, 9, 16]), which are encrypted in such a way, that only a token (a so-called trapdoor) sent by the querying user allows for comparing the searchword with the ciphertext. There are also schemes, that avoid having an index by embedding the trapdoor in a special format into the ciphertext itself (see [16]).

2.4 Onion Layer Model

Our framework adapts the basic idea of CryptDB's [14] onion layer model (OLM): each encrypted data item is surrounded by an outer layer consisting of a strong randomized (RND) encryption. The inner layers consist of the property-preserving encryptions of the data item. The outer RND layer is only removed when the inner layer is needed for query answering. While CryptDB supports relational (SQL) databases, our usage of WCSs requires some changes to the original setting. In particular, row identifier columns must be treated differently from all other columns regarding the onion layer design. They must leak the order of values to allow for row sorting (see the discussion in [20]).

3 The CloudDBGuard framework

In prior work we identified and implemented database-compatible and practically feasible encryption schemes and quantified their performance with various benchmarks. The following subsections introduce a couple of beneficial extensions of the previously developed concepts.

3.1 API

CloudDBGuard aims at executing queries over encrypted data in wide column stores. Yet, it does not follow the approach of a proxy server between the application and database for re-writing queries, decrypting query results, etc (like various other approaches in this field [15, 14]). Instead it introduces an application programming interface (API) taking care of these tasks, that is used by the client application. This API has various advantages over the proxy model:

- A third entity besides client and server can be avoided, which results in less computation and network overhead.
- Most proxy approaches make use of the fact that the majority of SQL queries use a well defined (and rather small) subset of SQL commands. In contrast, some NoSQL databases do not even have query languages. Thus, there would be no uniform way for a proxy to manage incoming requests. The API of CloudDBGuard hides the complexity of the databases' native APIs.
- The WCS data model is realized differently by the databases. In our API the differences in these realizations appear unified for the user.
- The user is able to configure the parameters of the used property preserving encryption schemes much more fine-grained and individually.

The client application using the API of CloudDBGuard runs in a trusted environment. For the API to be able to manage its tasks, it has to maintain auxiliary data, namely keys, metadata and (if necessary) indexes on client side. Since this data has to be stored persistently, it is kept outside the application in the client system's file system. The API manages the database connections, data transfer, encrypting and decrypting. Furthermore it keeps track of metadata and key management. Currently, CloudDBGuard utilizes advanced (index-based) encryption schemes, which allow the system to scale better when datasets become large. Partly, they even provide new functionality (like the ability to search for single words without the need of secondary indexes). The database server never sees any decryption keys, hence it is never able to decrypt private data. Thus, any adversaries (even administrators of the cloud services) are not able to gain sensitive information only from read access. There is no need to change database implementations in order to work with CloudDBGuard.

As FamilyGuard uses encryption schemes that potentially use a high number of cryptographic keys, the manual management of these keys is impractical for the user, but since the database server is not allowed to possess them either, they have to be managed and stored on the client side. This is why FamilyGuard uses

a Java Cryptography Extension KeyStore (JCEKS) provided by the Java Cryptography Extension (JCE) for that task. A JCEKS allows storing an arbitrary number of keys, each of which can be accessed using a custom label. The user has to provide only one single password for the client to gain access to all keys.

3.2 Selective Encryption

Selective encryption only encrypts pre-determined sensitive columns (consider a table of employees, where only the salary has to be kept secret, but not the name, department, etc). Selective encryption helps to save computation time when reading from as well as writing to the database, because it reduces the number of required encryption and decryption operations. It also reduces storage space, because it avoids unnecessary indexes. In the CloudDBGuard API, individual columns can be marked as not to be encrypted.

3.3 Separation of Duties

Separation of Duties describes the concept of more than one person being responsible to complete a task. The same principle can be used to take care of privacy issues that property-preserving encryption alone is not able to cover. Consider again the employee example. If the salary is encrypted with OPE, the exact salary of the employees still does not leak, but it can easily be inferred, who earns the most or who earns more money than others. That might be unwanted. If more than one database instance is available and the available databases are managed by individual and independent authorities, CloudDBGuard can further split up the table in order to avoid such conclusions. The salary column can be stored separated from the rest to avoid the leakage of any connection between salary and employee names. The API of CloudDBGuard supports separation of duties during the process of creating a table. Individual columns of a table can be spread across multiple database instances using the following three strategies:

- Random distribution: In this approach a logical table's columns are distributed randomly across the database instances available for the keyspace.
- Round Robin distribution: This approach distributes columns of a logical table in round robin fashion across the available database instances; the data get distributed as evenly as possible at the point of creating a table.
- Custom Distribution: The user actively specifies, which columns have to be stored separately from which other columns. Considering the example above the user could select the sensitive salary column to be stored separately from all other columns. A more in-depth analysis of customized separation of duties as an optimization problem is given in [4, 3].

3.4 Table Profiles

The encryption schemes for order-preserving and searchable encryption have their individual strengths and weaknesses. That is why the API of CloudDBGuard provides the option of specifying so-called *table profiles* when creating

tables. A table profile determines which combination of PPE encryption schemes is actually used during data insertion. The API supports three table profiles:

- **Optimized reading:** This profile prioritizes schemes that have advantages for queries that involve mainly reading from the database. Thus it is the best choice for "write-once" databases. The OPE schemes best suited for fast reading are [23] and [11]. They have the same type of index, which results in equal reading performance. However, [23] is the preferred choice because it also shows good performance in case of a pre-sorted input (see [19]). For the SE onion layer the scheme of [9] is used. It is the fastest scheme for queries that we have implemented – in particular for repeated queries.
- **Optimized writing**: This profile prioritizes schemes that have advantages for queries that involve mainly writing to the database. Thus, it should be used for scenarios, in which writing occurs more often than reading. The OPE scheme best suited for fast writing is [11], as long as presorted inputs are avoided. For the SE onion layer the scheme of [16] is used; it does not maintain indexes and can insert data faster than [9].
- **Storage efficient:** This profile prioritizes storage needs over computation time and selects the schemes that require the least amount of storage for data and indexes, on client side as well as on server side. Thus, OPE onion layers use [2], since it does not require an index at all. For the same reason SE onion layers use [16].

Other custom table profiles can be added easily. Independent of the used profile, every column gets its own instances of the property-preserving encryption schemes they use. That means, encryption scheme indexes are maintained per column, not per table. This allows the separation of duties as described above; querying answering involves only the index data that is actually required.

3.5 Unification of Data Models

The supported databases Cassandra and HBase follow the WCS data model (see Section 2.2). Yet, they differ in the way of achieving that. Using different databases with separation of duties therefore requires analyzing their differences. Here we discuss how the CloudDBGuard API compensates them.

- The WCS data model dictates the existence of (at least) one column that stores unique row identifiers per table. Cassandra and HBase have different ways of addressing this column. Cassandra requires assigning a concrete name and data type for it while creating a table. In contrast, HBase does not need any of that information, because its row identifier columns are unnamed and are always of type byte blob. Cassandra hence requires more precise definitions for the row identifier column; defining a name and data type is thus mandatory when creating tables with CloudDBGuard.
- Some WCSs allow the row identifier to consist of multiple parts. For example, it is possible in Cassandra to combine multiple fields to create a row identifier, which is then called a composite key. A composite key always consists

of two parts. The first part is the partition key (for data distribution across servers). The second part is the clustering key (for storing data within a partition). Both parts can again consist of multiple fields. In contrast, HBase does not know the concept compound row identifiers. If the combination of multiple fields is desired, that has to be created manually (by string concatenation) and stored as a single row identifier. HBase's native Java API provides prefix filters, that can be used to simulate compound keys. The API of CloudDBGuard hides the complexity of using these from the user.
- Apache Cassandra supports collection types: a field in a row cannot only contain a single value, but also a list, set or map. Elements of these subsets can be addressed by traversal (set), specifying an index (list) or a key (map). In contrast, HBase does not support collection types. Instead, an additional *column qualifier* can be used to realize collection types. With the CloudDBGuard API, the user does not need to care about the difference.
- Cassandra offers a variety of data types, whereas HBase stores everything as byte array. CloudDBGuard follows the HBase approach and stores only byte arrays in encrypted columns in Cassandra as well. On the one hand, except for OPE schemes, outcomes of all used encryption schemes are byte arrays anyway. Storing them as such avoids conversions back to their original data type and saves runtime. Only OPE ciphertexts have to be converted to byte arrays, which can be done fast. On the other hand, seeing only byte blobs in the database makes it much harder for an attacker to infer information.

4 Benchmark

Runtime tests of PPE often assess only one encryption scheme in isolation. In contrast, we benchmark the performance of different types of property-preserving encryption schemes (deterministic, order-preserving and searchable encryption) in their combination. All experiments in this section were run on an Intel Core i7-4600U CPU@2.10GHz, 8GB RAM, a Samsung PM851 256GB SSD using Ubuntu 16.04. The PPE schemes were implemented in Java 8, using cryptographic primitives of the Java Cryptography Extension. The source code is available at `https://github.com/dbsec/FamilyGuard/`. From the Enron corpus [12], we parsed a random subset of 10 000 mails (with $1.03 \cdot 10^7$ words) to simulate an average sized mailbox and created one table row per mail. Our queries select the primary key (the mail identifier) of those rows that meet the specified conditions. We select the primary key, as a unique and a relative small field, in order to let the database fully evaluate these conditions. The small size of the column entries enables to compare database performance without side effects. We wrote queries for each type of scheme separately (Q1-Q3), the six possible combinations of two types (Q4-Q9) and the combination of all the three types (Q10). Details about the columns queried can be found in Table 1.

We tested all queries with each of the three table profiles introduced in Section 3.4. Figures 1 shows the results. As can be seen, the query times remain within acceptable time spans of less than two seconds for Cassandra (left) and

Query	Types of schemes	Queried Columns	Query	Types of schemes	Queried Columns
Q1	DET	receiver	Q6	DET + SE	sender, body
Q2	OPE	timestamp	Q7	OPE + OPE	sender, timestamp
Q3	SE	body	Q8	OPE + SE	body, timestamp
Q4	DET + DET	sender, receiver	Q9	SE + SE	body, subject
Q5	DET + OPE	sender, timestamp	Q10	DET + OPE + SE	sender, timestamp, subject

Table 1. Types of generated queries

even less than one second for HBase (right). This reflects the fact that Cassandra is optimized for writing while HBase is optimized for reading. Furthermore, searchable encryption is by far the most expensive type of property preserving encryption, as all queries involving it take the most time. It has a strong impact, especially when the data fields are large (like in Q9, which involves performing SE on mail bodies). In contrast, deterministic and order-preserving encryption have less impact on runtime.

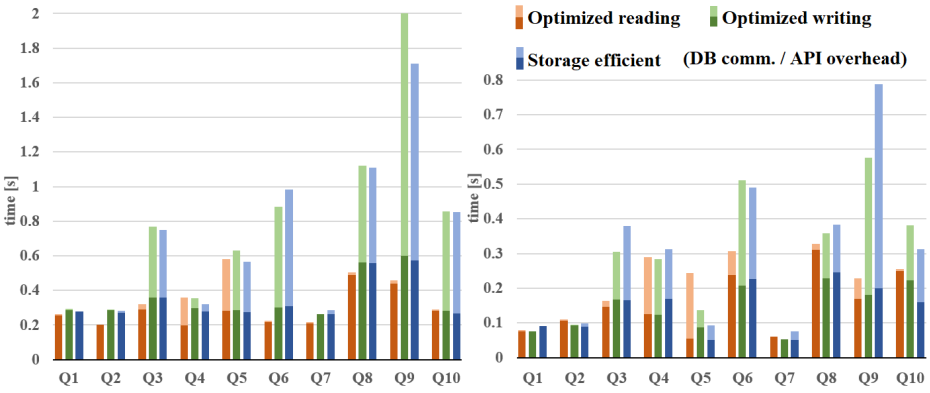

Fig. 1. Query Performance with Cassandra (left) and HBase (right). Measurements for each query: optimized reading (left), optimized writing (middle), storage efficient (right); DB communication (lower part of each bar) + API overhead (upper part)

5 Conclusion and Future Work

We presented the CloudDBGuard framework that extends our prior work in two aspects. Firstly, its functionality was wrapped into an easy-to-use API that hides the complexity of property-preserving encryption and even the native interface of the underlying database from the user. Secondly, the concept of using property-preserving encryption was combined with other ideas to increase security (e.g., separation of duties) and improve runtime performance (e.g., selective

encryption). The framework provideds built-in support for Apache Cassandra and HBase. While the concepts of CloudDBGuard are designed for wide column stores, they are not limited to them. Further databases and encryption schemes can be added by implementing simple interfaces. The data model of wide column stores can be mapped easily to key-value stores (in which the key part might be composed as table:column:qualifier:timestamp) or document stores (where rows can be mapped to documents and columns can be mapped to fields of a document). The general idea of doing has been suggested by other authors, too (e.g., [1, 24]). The major aspect is that the database systems (being open source systems developed by a community) can remain unchanged. Moreover, the API can be integrated into a proxy client between application and database server. In this way, no modifications to the application would be necessary, but an additional architectural component is introduced (as done in the approach of CryptDB [14]). A step further would be the integration of the onion layer model and PPE schemes into the database drivers/native APIs, combining the architectural simplicity of the approach of CloudDBGuard with the opportunity to leave the client application as well as the database server unchanged. This solution however would be very database-specific and the option to transparently use different databases would not be available.

Acknowledgements. This work was funded by the DFG (grant Wi 4086/2-2).

References

1. Atzeni, P., Bugiotti, F., Rossi, L.: SOS (Save Our Systems): A uniform programming interface for non-relational systems. In: Proceedings of the 15th International Conference on Extending Database Technology. pp. 582–585. ACM (2012)
2. Boldyreva, A., Chenette, N., O'Neill, A.: Order-preserving encryption revisited: Improved security analysis and alternative solutions. In: Advances in Cryptology–CRYPTO 2011, pp. 578–595. Springer (2011)
3. Bollwein, F., Wiese, L.: Closeness constraints for separation of duties in cloud databases as an optimization problem. In: British International Conference on Databases. pp. 133–145. Springer (2017)
4. Bollwein, F., Wiese, L.: Separation of duties for multiple relations in cloud databases as an optimization problem. In: Proceedings of the 21st International Database Engineering & Applications Symposium. pp. 98–107. ACM (2017)
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. pp. 1071–1080. ACM (2011)
6. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: A distributed storage system for structured data. ACM Transactions on Computer Systems 26(2), 4 (2008)
7. Chenette, N., Lewi, K., Weis, S.A., Wu, D.J.: Practical order-revealing encryption with limited leakage. In: International Conference on Fast Software Encryption. pp. 474–493. Springer (2016)

8. Curtmola, R., Garay, J., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: improved definitions and efficient constructions. In: Proceedings of the 13th ACM Conference on Computer and Communications Security. pp. 79–88. ACM (2006)
9. Hahn, F., Kerschbaum, F.: Searchable encryption with secure and efficient updates. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. pp. 310–320. ACM (2014)
10. Kadhem, H., Amagasa, T., Kitagawa, H.: MV-OPES: multivalued-order preserving encryption scheme: A novel scheme for encrypting integer value to many different values. IEICE TRANSACTIONS on Information and Systems 93(9), 2520–2533 (2010)
11. Kerschbaum, F., Schröpfer, A.: Optimal average-complexity ideal-security order-preserving encryption. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. pp. 275–286. ACM (2014)
12. Klimt, B., Yang, Y.: The enron corpus: A new dataset for email classification research. In: Machine learning: ECML 2004, pp. 217–226. Springer (2004)
13. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review 44(2), 35–40 (2010)
14. Popa, R.A., Redfield, C.M.S., Zeldovich, N., Balakrishnan, H.: Cryptdb: processing queries on an encrypted database. Commun. ACM 55(9), 103–111 (2012)
15. Popa, R.A., Zeldovich, N., Balakrishnan, H.: Guidelines for using the cryptdb system securely. IACR Cryptology ePrint Archive (2015), `http://eprint.iacr.org/2015/979`, report 2015/979
16. Song, D.X., Wagner, D., Perrig, A.: Practical techniques for searches on encrypted data. In: IEEE Symposium on Security and Privacy. pp. 44–55. IEEE (2000)
17. Sun, W., Wang, B., Cao, N., Li, M., Lou, W., Hou, Y.T., Li, H.: Privacy-preserving multi-keyword text search in the cloud supporting similarity-based ranking. In: Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security. pp. 71–82. ACM (2013)
18. Waage, T.: Familyguard repository, https://github.com/dbsec/FamilyGuard/
19. Waage, T., Homann, D., Wiese, L.: Practical Application of Order-Preserving Encryption in Wide Column Stores. In: Proceedings of the 13th International Joint Conference on e-Business and Telecommunications - SECRYPT. pp. 352–359 (2016)
20. Waage, T., Wiese, L.: Property preserving encryption in nosql wide column stores. In: OTM Confederated International Conferences. pp. 3–21. Springer (2017)
21. Wang, B., Yu, S., Lou, W., Hou, Y.T.: Privacy-preserving multi-keyword fuzzy search over encrypted data in the cloud. In: INFOCOM. pp. 2112–2120. IEEE (2014)
22. Wiese, L.: Advanced Data Management for SQL, NoSQL, Cloud and Distributed Databases. DeGruyter (2015)
23. Wozniak, S., Rossberg, M., Grau, S., Alshawish, A., Schaefer, G.: Beyond the ideal object: towards disclosure-resilient order-preserving encryption schemes. In: Proceedings of the 2013 ACM workshop on Cloud computing security workshop. pp. 89–100. ACM (2013)
24. Yuan, X., Wang, X., Wang, C., Qian, C., Lin, J.: Building an encrypted, distributed, and searchable key-value store. In: Proceedings of the 11th ACM Asia Conference on Computer and Communications Security. pp. 547–558. ACM (2016)