DEMO: Adjustably Encrypted In-Memory Column-Store

Florian Kerschbaum
Martin Härterich
Axel Schröpfer

Patrick Grofig
Mathias Kohler
Walter Tighzert

Isabelle Hang
Andreas Schaad

SAP
Karlsruhe, Germany
firstname.lastname@sap.com

ABSTRACT

Recent databases are implemented as in-memory column-stores. Adjustable encryption offers a solution to encrypted database processing in the cloud.

We show that the two technologies play well together by providing an analysis and prototype results that demonstrate the impact of mechanisms at the database side (dictionaries and their compression) and cryptographic mechanisms at the adjustable encryption side (order-preserving, homomorphic, deterministic and probabilistic encryption).

Categories and Subject Descriptors

H.2.0 [**Database Management**]: General—*Security, Integrity, and Protection*; D.4.6 [**Operating Systems**]: Security and Protection—*Cryptographic controls*

General Terms

Algorithms, Security

Keywords

Database Outsourcing, Encryption, In-Memory, Column Store

1. INTRODUCTION

In-memory column-store databases [3, 9, 10] use dictionary compression [1] to speed up processing. The smaller the data, the faster the transfer to the CPU.

In-memory column-store databases in the cloud are faced with severe security concerns. Adjustable encryption [8] offers the possibility to process data while encrypted. The clients can then upload data, retain the key and receive only the encrypted result for most SQL queries.

Adjustable encryption so far has only been tested on disk-based row-store databases. We demonstrate that the processing of in-memory column-store databases is very well amenable to adjustable encryption. We show a very small performance overhead when using less secure encryption schemes and dictionary compression.

CCS'13, November 4–8, 2013, Berlin, Germany.
ACM 978-1-4503-2477-9/13/11.
http://dx.doi.org/10.1145/2508859.2512491.

2. BACKGROUND

2.1 Adjustable Encryption

Security is a major concern for outsourced databases. In the database-as-a-service model an independent service provider offers its database to clients. The clients need to entrust their data to the cloud service provider without having control over unwanted disclosures, e.g., to insiders or hackers.

One solution to this outsourced security problem is to encrypt the data before sending it to the cloud. Of course, the decryption key needs to remain only at the client. This is easy to implement for simple storage, but the clients must be remain able to query the database. Therefore the service provider has to solve the complicated task of querying on the encrypted data.

Order-preserving encryption (OPE) [2], deterministic encryption (DET) [6] and (additive) homomorphic encryption (HOM) [5] offer a (partial) solution to the encrypted data querying problem. These different encryption schemes have different algebraic properties. Let $c = E_T(x)$ denote the encryption of plaintext x in encryption type $T \in \{OPE, DET, HOM\}$. We denote $D_T(c)$ the corresponding decryption. Order-preserving encryption has the property that it preserves the order of plaintexts, i.e.

$$x \leq y \Longleftrightarrow E_{OPE}(x) \leq E_{OPE}(y)$$

Deterministic encryption preserves the equality of plaintexts, i.e.

$$x = y \Longleftrightarrow E_{DET}(x) = E_{DET}(y)$$

In (additively) homomorphic encryption multiplication of ciphertexts (modulo a key-dependent constant) maps to addition of the plaintexts, i.e.

$$D_{HOM}(E_{HOM}(x) \cdot E_{HOM}(y)) = x + y$$

In terms of security we have: homomorphic (or standard) encryption is at least as secure as deterministic encryption which is at least as secure as order-preserving encryption. This observation implies that the client in data outsourcing should carefully choose its encryption types. It should only use order-preserving or deterministic encryption if necessary to enable its queries in order to achieve the highest security level. Yet, the set of executed queries may be unknown at design time making this choice undecidable.

Popa et al. offer an intriguing solution to the encryption type selection problem [8]. First, they introduce a further encryption type RND for standard, randomized encryption. This encryption type only allows retrieval, but no queries.

Column	Column Cardinality	Item Size	Plain Size	Compressed Size (Dictionary + Column)	Ratio
First name	5 million, 23bit	49 Byte	365.08 GB	234 MB + 21.42 GB = 22,168 GB	5.9%
Last name	8 million, 23 bit	50 Byte	372.5 GB	381 MB + 21.42 GB = 22,316 GB	5.9%
Gender	2, 1 bit	1 Byte	7.45 GB	2 Byte + 0.93 GB = 954 MB	12.5%
City	1 million, 20 bit	49 Byte	365.08 GB	47 MB + 18.62 GB = 19,120 GB	5.1%
Country	200, 8 bit	49 Byte	365.08 GB	6 KB + 7.45 GB = 7,629 GB	2.0%
Birthday	40, 6 bit	4 Byte	29.80 GB	76 KB + 5.59 GB = 5,722 GB	18.7%

Table 1: Example Table for Dictionary Compression

Note that order-preserving encryption enables a proper superset of queries to deterministic encryption. They therefore compose a layered ciphertext called onion. This encryption onion $E_{RND}(E_{DET}(E_{OPE}(x)))$ is composed of the following layers:

- *L3 – Randomized Encryption*: IND-CPA secure encryption allowing only retrieveal using AES encryption in CBC mode.
- *L2 – Deterministic Encryption*: Allows processing of equality comparisons using the encryption scheme of [6]. Furthermore, this encryption can be (proxy) re-encrypted, such that it is possible to join tables using columns, encrypted under different keys.
- *L1 – Order-Preserving Encryption*: Allows processing of greater-than comparisons using the encryption scheme of [2].
- *L0 – Data*: The data to be encrypted.

This onion at first only allows retrieval – due to the randomized encryption. Should the client encounter a query that requires deterministic encryption, e.g., a selection using equality, then it updates the database. It sends the key $D_{RND}()$ for decrypting the randomized encryption to the database. The database uses a user-defined function to perform the update, such that now the database stores $E_{DET}(E_{OPE}(x))$. This enables the new query to be executed. The same procedure occurs in case of a query that requires order-preserving encryption to execute.

Homomorphic encryption is handled slightly differently and stored in a separate column. The separate column also enables aggregation operations, but does not harm security, since homomorphic encryption is semantically secure. A layering is not possible, since homomorphic encryption needs to encrypt the plaintext x for the correct result in aggregations.

This algorithm represents an adjustment mechanism of the database to the series of executed queries. It enables to dynamically adapt the encryption types, i.e., without knowing all queries in advance. Furthermore, the adjustment is unidirectional. Once decrypted to deterministic or order-preserving encryption it is never necessary to return to a higher encryption level to enable a subsequent query. Yet, security against the service provider has already been weakened, because the less secure ciphertext has been revealed at least once. We call such a database *adjustably encrypted*.

2.2 Column-Store In-Memory Databases

We investigate our encryption algorithms as part of an encrypted, in-memory, column-store database. This has a couple of design implications we highlight in this section.

Column-store databases, such as [3, 9, 10] show excellent performance for analytical workloads. For this they store the data column-wise instead of row-wise. All data for a certain can such be accessed and processed very quickly. The speed of processing can be enhanced further if the data is stored in main memory.

In-memory, column-store databases process the entire column of data for operations, such as a selection. Hence, the speed of transferring data from the main memory to the CPU becomes the bottleneck. A common optimization is to compress the data [1]. This further significantly improves the processing performance.

A common technique is order-preserving dictionary compression [1]. In dictionary compression data values are replaced by data identifiers and in a dictionary their relation is stored. A dictionary is order-preserving, if the order relation of the data identifiers is the same as the order relation of the data values. Figure 1 shows an example dictionary for currency values.

Data Identifier	Data Value
1	\$ 0.99
2	\$ 1.54
3	\$ 9.23

Figure 1: Example Dictionary

Order-preserving dictionaries have the advantage that select operations – even for range queries – can be performed without accessing the dictionary. The database operator is fed with the data identifier (or data identifiers for range queries) to select and can then process the column. Any select operation that needs to lookup the dictionary can be very costly.

Also update or insert operations can be very costly. They often need to recompute the entire column of data. This may also involve some further compression operations.

Note that the order-preserving dictionary is an ideal-secure order-preserving encryption. The database performs this operation automatically, although not as encryption operation. It therefore becomes a crucial design decision for an encrypted database how to integrate with this dictionary.

One approach is to strip the dictionary of the data values and keep those at the client. This has been proposed by Hildenbrand et al. in [4]. On the one hand this achieves ideal-security for the order-preserving encryption, since the database only learns the data identifiers. On the other hand this prevents all operations that require access to the data values, such as aggregation which is a very common operation in analytical work loads.

Another approach is to encrypt the data values in the dictionary. This has been proposed by Popa et al. in [7]. It also achieves ideal-security on the database, but requires $O(n \log n)$ cost for inserting n, since each element needs to be sorted into the dictionary. When using homomorphic encryption [5] this can also achieve aggregation.

A disadvantage of both approaches is that the database always needs to be encrypted in order-preserving encryption. We therefore resort to Boldyreva et al.'s scheme [2] and adjustable encryption as introduced in Section 2.1. Encryption is layered from order-preserving on the innermost layer over deterministic encryption to randomized encryption on the outermost layer. Depending on the operation performed one or more layers of encryption are removed before executing the operator. This results in significantly better security, since only a subset of columns needs to be encrypted order-preserving.

3. CONCLUSION

Enc.	Compressed Size (Dictionary + Column)	Overhead
RND	596,05 GB + 245,87 GB = 862.122 GB	3789%
DET	305 MB + 21,42 GB = 22.239 GB	0%
OPE	153 MB + 21,42 GB = 22.086 GB	0%

Table 2: Dictionary Compression under Encryption

We will demonstrate the impact of adjustable encryption on an in-memory column-store database. For most data dictionary compression works fine. Consider the following example for the world population as in Table 1

Plain size is calculated as

$$Plain\ Size = Item\ Size \times World\ Population$$

and the compressed size is calculated as

$$\begin{aligned} Compressed\ Size &= Dictionary + Column \\ &= Item\ Size \times Column\ Cardinality + \\ &\quad Cardinality\ Size \times WorldPopulation \end{aligned}$$

We denote as $Column\ Cardinality_X$ the column cardinality for encryption scheme $X \in \{RND, DET, OPE\}$. Now observe the following. Under L3 randomized encryption it holds that

$$Column\ Cardinality_{RND} = World\ Population$$

Therefore the dictionary has already the same size as the plain data (plain size) and the column only adds more data. Hence, it is beneficial to not do dictionary compression.

Whereas under L2/L1 deterministic and order-preserving encryption it holds that

$$\begin{gathered} Column\ Cardinality_{DET} = \\ Column\ Cardinality_{OPE} = \\ Column\ Cardinality \end{gathered}$$

The item size might increase due to blocking of the encryption algorithm, but that is the only additional space requirement. Hence, it is beneficial to do dictionary compression. See the examples for the column of first names in Table 2.

We therefore conclude that depending on the security level dictionary compression of in-memory column-store databases can be beneficial, particularly when processing the encrypted column. Furthermore, the impact of compression in case of deterministic or order-preserving encryption is very small. We underpin these results in our demonstration of our prototype.

4. REFERENCES

[1] C. Binnig, S. Hildenbrand, and F. Färber, "Dictionary-based order-preserving string compression for main memory column stores," in *Proceedings of the ACM International Conference on Management of Data*, ser. SIGMOD, 2009.

[2] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill, "Order-preserving symmetric encryption," in *Advances in Cryptology*, ser. EUROCRYPT, 2009.

[3] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees, "The SAP HANA database – an architecture overview," *IEEE Data Engineering Bulletin*, vol. 35, no. 1, pp. 28–33, 2012.

[4] S. Hildenbrand, D. Kossmann, T. Sanamrad, C. Binnig, F. Färber, and J. Wöhler, "Query processing on encrypted data in the cloud," Department of Computer Science, ETH Zurich, Tech. Rep. 735, 2011.

[5] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Proceedings of the 18th International Conference on Advances in Cryptology*. ser. EUROCRYPT, 1999.

[6] S. Pohlig, and M. Hellman, "An improved algorithm for computing logarithms over GF(p) and its cryptographic significance," *IEEE Transactions on Information Theory*, vol. 24, no. 1, pp. 106–110, 1978.

[7] R. A. Popa, F. H. Li, and N. Zeldovich, "An ideal-security protocol for order-preserving encoding," in *34th IEEE Symposium on Security and Privacy*, ser. S&P, 2013.

[8] R. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. "CryptDB: Protecting confidentiality with encrypted query processing." in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, ser. SOSP, 2011.

[9] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik, "C-store: a column-oriented dbms," in *Proceedings of the 31st International Conference on Very Large Data Bases*, ser. VLDB, 2005.

[10] M. Zukowski, P. A. Boncz, N. Nes, and S. Héman, "Monetdb/x100 - a dbms in the cpu cache," *IEEE Data Engineering Bulletin*, vol. 28, no. 2, pp. 17–22, 2005.