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 show a very small performance overhead when using less secure encryption schemes and dictionary compression.

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 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 database querying problem. These different encryption schemes have different algebraic properties. Let \( c = E_T(x) \) denote the encryption of plaintext \( x \) in encryption type \( T \in \{ \text{OPE}, \text{DET}, \text{HOM} \} \). We denote \( D_T(c) \) the corresponding decryption. Order-preserving encryption has the property that it preserves the order of plaintexts, i.e.

\[
x \leq y \iff E_{\text{OPE}}(x) \leq E_{\text{OPE}}(y)
\]

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

\[
x = y \iff E_{\text{DET}}(x) = E_{\text{DET}}(y)
\]

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

\[
D_{\text{HOM}}(E_{\text{HOM}}(x) \cdot E_{\text{HOM}}(y)) = x + y
\]

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.
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 retrieval 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 \( D_{RND}(\cdot) \) 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.

<table>
<thead>
<tr>
<th>Column</th>
<th>Column Cardinality</th>
<th>Item Size</th>
<th>Plain Size</th>
<th>Compressed Size (Dictionary + Column)</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>First name</td>
<td>5 million, 23 bit</td>
<td>1 Byte</td>
<td>7.45 GB</td>
<td>2 Byte + 0.93 GB = 954 MB</td>
<td>12.5%</td>
</tr>
<tr>
<td>Last name</td>
<td>8 million, 23 bit</td>
<td>1 Byte</td>
<td>7.45 GB</td>
<td>2 Byte + 0.93 GB = 954 MB</td>
<td>12.5%</td>
</tr>
<tr>
<td>Gender</td>
<td>2, 1 bit</td>
<td>1 Byte</td>
<td>7.45 GB</td>
<td>2 Byte + 0.93 GB = 954 MB</td>
<td>12.5%</td>
</tr>
<tr>
<td>City</td>
<td>1 million, 20 bit</td>
<td>1 Byte</td>
<td>7.45 GB</td>
<td>2 Byte + 0.93 GB = 954 MB</td>
<td>12.5%</td>
</tr>
<tr>
<td>Birthday</td>
<td>40, 6 bit</td>
<td>1 Byte</td>
<td>7.45 GB</td>
<td>2 Byte + 0.93 GB = 954 MB</td>
<td>12.5%</td>
</tr>
</tbody>
</table>

Table 1: Example Table for Dictionary Compression
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

<table>
<thead>
<tr>
<th>Enc.</th>
<th>Compressed Size (Dictionary + Column)</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>RND</td>
<td>596.05 GB + 245.87 GB = 862.12 GB</td>
<td>3789%</td>
</tr>
<tr>
<td>DET</td>
<td>305 MB + 21.42 GB = 22.239 GB</td>
<td>0%</td>
</tr>
<tr>
<td>OPE</td>
<td>153 MB + 21.42 GB = 22.086 GB</td>
<td>0%</td>
</tr>
</tbody>
</table>

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

\[
\text{Plain Size} = \text{Item Size} \times \text{World Population}
\]

and the compressed size is calculated as

\[
\text{Compressed Size} = \text{Dictionary + Column} = \text{Item Size} \times \text{Column Cardinality} + \text{Cardinality Size} \times \text{World Population}
\]

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

\[
\text{Column Cardinality}_{\text{RND}} = \text{World Population}
\]

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

\[
\text{Column Cardinality}_{\text{DET}} = \text{Column Cardinality}_{\text{OPE}} = \text{Column Cardinality}
\]

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


