Optimal Re-encryption Strategy for Joins in Encrypted Databases

Florian Kerschbaum, Martin Härterich, Patrick Grofig, Mathias Kohler, Andreas Schaad, Axel Schröpfer, and Walter Tighzert

SAP Applied Research
Vincenz-Priessnitz-Str. 1 Karlsruhe, Germany
firstname.lastname@sap.com

Abstract. In order to perform a join in a deterministically, adjustably encrypted database one has to re-encrypt at least one column. The problem is to select that column that will result in the minimum number of re-encryptions even under an unknown schedule of joins. Naive strategies may perform too many or even infinitely many re-encryptions. We provide two strategies that allow for a much better performance. In particular the asymptotic behavior is $O(n^{3/2})$ resp. $O(n \log n)$ re-encryptions for n columns. We show that there can be no algorithm better than $O(n \log n)$. We further extend our result to element-wise re-encryptions and show experimentally that our algorithm results in the optimal cost in 41% of the cases.

Keywords: Encrypted Database, Proxy Re-encryption, Adjustable Join.

1 Introduction

Recently, encrypted databases [2,7,9] that provide the client with additional protection in the cloud or database-as-a-service setting have emerged. In these databases all data are encrypted at the client – where also the keys are exclusively stored – and queries are performed over encrypted data. In order to perform a selection, e.g. `SELECT x FROM y WHERE z = 1`, the columns for selection (`z` in the example) needs to be encrypted using deterministic encryption, i.e., a plaintext always enciphers to the same ciphertext. In order to perform a join, e.g. `SELECT a.b, c.d FROM a, c WHERE a.e = c.f`, the columns for the join (`e` and `f` in the example) need to be encrypted using the same key. This is achieved using an operation called proxy re-encryption (PRE) [3]. In PRE a proxy translates a ciphertext under one key K_A to a ciphertext under another key K_B without knowing either of the two keys.

The encrypted database performs this PRE when required, i.e. when it has received a query performing a join over two previously unjoined columns. The client then issues a PRE key to the databases which re-encrypts at least one column, such that both columns are encrypted under the same key. The (equi-) join operator can then operate using the same algorithm as on an unencrypted database.

L. Wang and B. Shafiq (Eds.): DBSec 2013, LNCS 7964, pp. 195–210, 2013.

The reason for dynamically adjusting the database encryption to the queries is that this PRE reveals additional information to an attacker observing the database. He now obtains additional ciphertexts he can use in cryptanalysis of the keys. Deterministic encryption is only provably secure in high-entropy domains [1], such that these additional ciphertexts may be of significant help.

When the database issues the PRE key, it has to choose a column which to re-encrypt; in the example either `e` or `f`. Furthermore, it has to make this choice under an unknown schedule of future joins. Naive approaches may even lead to an infinite number of re-encryptions.

In this paper we present a re-encryption column selection algorithm that results in at most $O(n^{3/2})$ re-encryptions for n columns under any schedule of join operations. Furthermore we give a second algorithm that occasionally leads to re-encryption of both columns to be joined, but which results in a better bound of at most $O(n \log n)$ re-encryptions (where, of course, the PRE of two columns counts as two PREs). We show that this is the best possible bound we can achieve under the assumption of an a priori unknown schedule.

The remainder of the paper is structured as follows. We review related work and background on encrypted databases in Section 2. Section 3 gives an introduction to the problem using the naive approaches. Our algorithms including proofs of bounds on the number of re-encryptions are shown in Section 4 . We show our experimental evaluation in Section 5. Section 6 summarizes our findings.

2 Related Work

The design goal of encrypted databases in the database-as-a-service setting is to move the encryption layer above the query processing layer. All query process operators are supposed to work on encrypted data. This ensures that (almost) any query can be processed on the encrypted data.

The first such database was introduced in [7]. It provided special operators for many queries and it was necessary to post-process and filter many queries. This was improved by [2] where the database operators remain unchanged. This enables using existing, commercial database systems for encryption in the cloud. Nevertheless, it requires the use of special encryption schemes such as order-preserving encryption [4].

In [2] the encryption was replaced by using the identifiers in the data dictionary and leaving the dictionary at the client. This requires even less modification to the database and is as secure as order-preserving encryption. Nevertheless, it does not allow aggregation.

Order-preserving [4] or even deterministic encryption is commonly not considered very secure. It is therefore advisable to encrypt only the columns necessary for performing the queries using these encryption schemes. Yet, these columns may not be known in advance and the database must adjust its encryption state to the queries performed.

In order to adjust the encryption to the queries on the fly, [9] proposed the use of onion encryption. While it is possible to choose an optimally secure encryption, if all queries are known upfront, it is difficult to do so, if any queries are processed on the fly. Therefore each data item is encrypted using *onion encryption* and decrypted to the corresponding onion layer on the fly. Our encryption onion 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. In deterministic encryption one plaintext always enciphers to the same ciphertext.
- *L1 – Order-Preserving Encryption*: Allows processing of greater-than comparisons using the encryption scheme of [4].
- *L0 – Data*: The data to be encrypted.

The layers of the onion represent a strict order, i.e. the lower the layer the less secure, but also the more operations it supports. It is important that each lower layer supports all operations that the upper layer supports, such that a decryption never needs to be undone.

The client analyzes each query before executing it. It determines the necessary encryption layer in the onion encryption in the database. Then, before sending the query, the client performs the decryption of the column to that onion layer. No encryption from a lower to a higher encryption layer is ever performed. As such, the level (layer) of encryption in the database is dynamically adjusted to the queries processed.

In order to perform an equi-join operation data is decrypted to the deterministic layer L2, but different columns may still be encrypted using different keys. In this case proxy re-encryption (PRE) [3] can be performed. In PRE a proxy translates a ciphertext encrypted under one key into a ciphertext under another key without decrypting it first, i.e. the proxy does not learn the plaintext or any of the two keys. The proxy does, however, learn a relation between the two keys, such that the security against cryptanalysis is reduced to the secrecy of one key.

We use a proxy re-encryptable, deterministic encryption scheme. An example is the symmetric Pohlig-Hellman encryption [8].

Let p be the prime order of a group $\mathbb{Z}_p$. Let m be an element of $\mathbb{Z}_p$ representing a message to encipher. Let $ord(p) = p-1$ be the order of the multiplicative group $\mathbb{Z}_p^*$ of invertible elements in $\mathbb{Z}_p$. We uniformly choose an element e of $\mathbb{Z}_{ord(p)}$, such that $\gcd(e, ord(p)) = 1$. We encrypt m to the ciphertext c as

$$c = m^e \bmod p$$

We decrypt the ciphertext c as

$$m = c^{e^{-1} \bmod ord(p)} \bmod p$$

The element e is the secret key. Let two database columns A and B have two different keys e_A and e_B, respectively, but both encrypted at the deterministic

layer L2. Furthermore, assume we have chosen to re-encrypt column A to the key of columns B. We then compute the PRE key k as

$$k = e_A^{-1} e_B \bmod ord(p)$$

The database can now perform the proxy re-encryption operation. Each ciphertext c of column A is re-encrypted to a ciphertext c' using the PRE key k as

$$c' = c^k = m^{e_A k} = m^{e_A e_A^{-1} e_B \bmod ord(p)} = m^{e_B} \bmod p$$

In [9] the authors suggest to use this encryption scheme, but on elliptic curves. Unfortunately, their encryption scheme is not decryptable, since they use g^m instead of m. This may require additional storage on the database. Old-fashioned Pohlig-Hellman encryption over multi-precision integers is decryptable. The authors also provide a cryptanalysis of their scheme in [10] under an adjustable join attack. This extends to Pohlig-Hellman encryption.

3 Naive Approaches

Let there be a database with n columns A, B, C and so forth. Initially each column is deterministically encrypted under its own key. We then perform a number of queries on the database, possibly involving join operations. We write

JOIN(A, B)
JOIN(B, C)

for first joining columns and A and B and then columns B and C. Joins with $k \geq 2$ columns can be simulated by joining $k - 1$ pairs. However, the order in which the pairs are chosen is not arbitrary. We will give more details on how to do this efficiently in section 4.8. In order to perform a join operation, at least one column needs to be re-encrypted. We write

JOIN(A, B): $A \leftarrow B$

if column A gets re-encrypted to the key of column B.

The order of the two columns in the join operation is determined by the query string. Therefore the database connector has to choose the right column to re-encrypt.

We consider the effect of a few simple, straight-forward strategies. This should highlight that such simple strategies – while plenty – do not result in the best performance. The first strategy is to always use the first column in the query string. Assume the following schedule

JOIN(A, B): $A \leftarrow B$
JOIN(A, C): $A \leftarrow C$
JOIN(A, B): $A \leftarrow B$
JOIN(A, C): $A \leftarrow C$
$\vdots$

Clearly, this may lead to infinitely many re-encryptions and is therefore inadvisable. There is a maximum number of re-encryptions for any schedule and ideally this should be achieved.

Next, consider a total order of columns, e.g. lexicographically. We always re-encrypt the lower to the upper. Now, assume the following schedule

$$
\begin{aligned}
&\textsc{Join}(A, B)\colon A \leftarrow B\\
&\textsc{Join}(B, C)\colon B \leftarrow C\\
&\textsc{Join}(A, B)\colon A \leftarrow C\\
&\textsc{Join}(C, D)\colon C \leftarrow D\\
&\textsc{Join}(B, C)\colon B \leftarrow D\\
&\textsc{Join}(A, B)\colon A \leftarrow D\\
&\quad\vdots
\end{aligned}
$$

This leads to $\frac{n(n-1)}{2}$, i.e. $O(n^2)$ re-encryptions. Clearly, this is sub-optimal, since the same schedule can be completed with $n-1$ PREs in the following way

$$
\begin{aligned}
&\textsc{Join}(A, B)\colon A \leftarrow B\\
&\textsc{Join}(B, C)\colon C \leftarrow B\\
&\textsc{Join}(A, B)\\
&\textsc{Join}(C, D)\colon D \leftarrow B\\
&\textsc{Join}(B, C)\\
&\textsc{Join}(A, B)\\
&\quad\vdots
\end{aligned}
$$

We follow this idea in the formal definition of our algorithms in the next section.

4 Algorithms

4.1 Data Structures

In our algorithms we store two types of objects: columns and keys. For the sake of exposition, we store these objects as database table rows, but it could as well be Java objects or C/C++ structures. Storing them in database tables enables to be shared between multiple clients of the encrypted database and ensures persistence between different runs of the application of one client.
In the table *Keys* we store

- *KeyId*: An unique identifier for the key. It is the primary database key of the table.
- *Rank*: A rank of the key.

In the table *Columns* we store

- *ColumnId*: An unique identifier for the column. It may be or be generated from the name of the column `TABLE.COLUMN` which enables searching using the name. It is the primary key of the table.

- *Cost*: The cost of re-encrypting this column. For now we assume uniform cost of 1 for each column. We discuss non-uniform costs in 4.9.
- *KeyId*: The identifier of the associated key. This is a foreign key of this table and the primary key of the $Keys$ table.

4.2 Initialization

Algorithm 1. Initialization

```
function INIT
    for all column do
        cost ← 1
        INSERT keyId, cost INTO Keys
        INSERT columnId, cost, keyId INTO Columns
    end for
end function
```

We initialize each column with its own key and cost of 1. Each key is initialized with the cost of the associated column. This is performed as in Algorithm 1. When uploading the encrypted data into the database, the data of each column will be encrypted under its associated key. Subsequently we can perform queries with optional joins.

4.3 Key Retrieval

Algorithm 2. Key Retrieval

```
function GETKEY(column)
    return SELECT keyId FROM Columns WHERE columnId = column
end function
```

When we perform a query we must encrypt parameters and decrypt return values. We therefore need to retrieve the corresponding key identifier for the accessed columns. Algorithm 2 shows that this can be performed using a simple query.

4.4 Column Selection

When performing a join between two columns A and B we need to select one for re-encryption. The function in Algorithm 3 returns the identifier of the column to be re-encrypted. It has already updated the data structure to reflect its new key – that of the other column. We call the column that does not get re-encrpyted the steady column.

Algorithm 3. Column Selection

```
    function JOIN(columnA, columnB)
        keyA ← GETKEY(columnA)
        keyB ← GETKEY(columnB)
        if keyA = keyB then
 5:         return null
        end if
        rankA ← SELECT rank FROM Keys WHERE keyId = keyA
        rankB ← SELECT rank FROM Keys WHERE keyId = keyB
        if rankA > rankB then
10:         lower ← columnB
            (lowerKey, lowerRk) ← (keyB, rankB)
            (upperKey, upperRk) ← (keyA, rankA)
        else
            lower ← columnA
15:         (lowerKey, lowerRk) ← (keyA, rankA)
            (upperKey, upperRk) ← (keyB, rankB)
        end if
        lowerCost ← SELECT cost FROM Columns WHERE columnId = lower
        UPDATE Keys SET rank = lowerRk − lowerCost WHERE keyId = lowerKey
20:     UPDATE Keys SET rank = upperRk + lowerCost WHERE keyId = upperKey
        UPDATE Columns SET keyId = upperKey WHERE columnId = lower
        if lowerRank − lowerCost = 0 then
            DELETE FROM Keys WHERE keyId = lowerKey
        end if
25:     return lower
    end function
```

We make the choice simply by the rank of the key. The column with the key with the lower rank gets re-encrypted. Afterwards, we add the cost of the re-encrypted column to the rank of the steady column and subtract the same cost from the rank of the key of the re-encrypted column.

If the rank of the key of the re-encrypted column reaches 0, then we can delete the key entry, since it no longer encrypts any column.

Note that for any (infinite) schedule of joins the algorithm leads to a finite number of proxy re-encryptions only (i.e. it returns a value different from *null* only a finite number of times). This can be seen easily if we consider a variant of the algorithm where we omit the deletion of keys of rank 0 (lines 22 through 24). Then the sum of the absolute values of differences of the ranks over all pairs of keys is a non-negative integer which is bounded (by $\binom{n}{2}$ times the maximal possible rank) and which increases by at least 2 in each re-encryption step.

The algorithm is reminiscent of the Union-Find algorithm [6], but we do not join the entire group, just the selected column. This reduces the cost for one join operation, since we need to re-encrypt at most one column and not an entire group, but does not increase worst-case cost – due to the re-encryption of columns in shrinking groups – as we show in our analysis next.

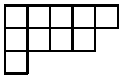
4.5 Analysis

We analyse the worst-case performance of our re-encryption selection algorithms. Here (and also in the analysis of the enhanced algorithm in section 4.7) we obtain the maximal security simply by taking the maximal possible number of different keys given the required functionality, i.e. whenever two columns are not joined in any previous step of the schedule they remain encrypted under different keys.

Let there be n columns and let $t(n)$ be the maximum number of re-encryptions that Algorithm 3 performs where the maximum is taken over all possible schedules of join operations. We now provide a proof that $t(n) = O(n^{3/2})$.

Theorem 1. *Algorithm 3 needs at most $2n^{3/2}$ re-encryptions for any schedule. This bound is optimal in the sense that the asymptotic behavior of $t(n)$ is $O(n^{3/2})$.*

Before we start with the proof of this theorem we recall some notation: A *partition* of n is a sequence $\lambda = (\lambda_1, \ldots, \lambda_k)$ where $\lambda_1 \geq \ldots \geq \lambda_k \geq 1$ are integers such that $\lambda_1 + \lambda_2 + \ldots + \lambda_k = n$. The partition λ can graphically be represented by a *Young diagram* which is composed of k rows containing $\lambda_1, \ldots, \lambda_k$ boxes (where the *top* row has length λ_1). By abuse of notation we use λ to denote both the partition and the associated Young diagram as in

$\lambda = (5, 4, 1) =$ 

The *dominance order* $\trianglerighteq$ on partitions is defined by

$$\lambda \trianglerighteq \mu : \Longleftrightarrow \ \forall i \geq 1 : \lambda_1 + \ldots + \lambda_i \geq \mu_1 + \ldots + \mu_i \ .$$

(Here $\lambda_i = 0$ resp. $\mu_i = 0$ for i greater than the number of rows of λ resp. μ.) It is well known (see [5]) that $\lambda \trianglerighteq \mu$ iff λ can be obtained from μ by succesively moving single boxes from lower to higher rows. The set of all partitions of n together with the dominance order $\trianglerighteq$ is a poset L_n and $(1, \ldots, 1)$ resp. (n) is the unique minimal resp. maximal element of L_n.

Partitions as well as the dominance order occur naturally in our algorithm: Let $\lambda_1 \geq \ldots \geq \lambda_k \geq 1$ be the sizes of the groups at any time during the execution of the algorithm (ordered non-increasingly). Then clearly $\lambda = (\lambda_1, \lambda_2, \ldots, \lambda_k)$ is a partition of n.

This way for every given schedule of join operations the Algorithm 3 produces a series of partitions $\lambda^{(1)}, \lambda^{(2)}, \ldots$ of n. The first partition in the sequence is always $(1, 1, \ldots, 1)$ (n ones) and the schedule can be further extended as long as there is more than one group left i.e. until it reaches the maximal partition (n).

We observe how λ changes when a single step of the algorithm is applied. We remove one element (database column) from one group and add it to another group of at least the same size. In terms of the associated Young diagrams this amounts simply to moving one box up into a higher row. In other words, the series of partitions derived from a join schedule is an increasing (w.r.t. $\trianglerighteq$) chain in the poset L_n.

On the other hand for any series of partitions $\lambda^{(1)}, \lambda^{(2)}, \ldots$ of n such that $\lambda^{(i+1)}$ is obtained from $\lambda^{(i)}$ by moving-up one box we can easily derive a join schedule that produces exactly this series of tableau in the way described above. To this end we just annotate each box with a database column label at the begin. Thereafter, we interpret moving-up such a box (keeping its label) by executing a join of the corresponding database column with any other column in the target row.[1]

Therefore, finding the worst case number of steps of our algorithm is equivalent to finding the longest totally ordered chain in the lattice L_n of partitions. Hence the theorem follows directly from

Proposition 1. *(a) The longest increasing sequence in the poset $(L_n, \trianglerighteq)$ consists of at most $2n^{3/2}$ elements.*
(b) The longest increasing sequence in the poset consists of at least $\frac{2}{3}n^{3/2}$ elements.

Proof. (a) Let $k = [\sqrt{n}]$ and consider an increasing (w.r.t. $\trianglerighteq$) series of Young diagrams of maximal length. We obtain this maximal length by counting the total number of move-up operations of boxes (and adding 1 for the initial diagram). To do so we have a look at the path of each of the n boxes individually. The key observation is that in each step the box not only moves up, but also moves to the right at least one place. Therefore after at most k of its moves it has reached (the end of) a row in the Young diagram with length $\geq k+1$. Since there are only $n < (k+1)^2$ boxes altogether this must be a row with index less than $k+1$. Hence, the box we considered can from now on only have $k-1$ further moves which is a total of $2k-1$.

This holds for every box and hence we obtain $n(2k-1)+1 \leq 2n^{3/2}$ as an upper bound on the length of the given sequence.

(b) We write $n = \frac{1}{2}k(k+1)+r$ for suitable non-negative integers k and $r \leq k$. Then we can construct a sequence of "triangular shaped" Young diagrams:

$$(1,\ldots,1) \to (2,1,\ldots,1) \to (3,2,1,\ldots,1) \to (4,3,2,1,\ldots,1) \to \cdots$$
$$\cdots \to (k,k-1,\ldots,2,1,\ldots,1)$$

(In the last diagram there are $1+r$ ones at the end.) Each step in this sequence further decomposes into a certain number of move-up operations for single boxes. Now let's count these single move-up operations . To go from $(j,j-1,\ldots,2,1,\ldots,1)$ to $(j+1,j,\ldots,2,1,\ldots,1)$ we need to move one box j times, the next box $j-1$ times and so on. This sums up to $\frac{1}{2}j(j+1)$. Altogether we have $\sum_{j=1}^{k} \frac{j(j+1)}{2} = \frac{k(k+1)(2k+7)}{12}$ single moves.

Next we move the "remaining" r ones to the first row. This amounts to rk single moves.

We continue from $(k+r,k-1,\ldots,2,1)$ to (n) in a symmetric (but reverse) way. Just interchange the role of rows and columns in the diagrams. This adds

[1] Caution: Be aware that although this reverse construction may suggest so, in the procedure of going from join schedules to Young diagrams a box is *not* associated with a fixed column and a row is *not* associated with a fixed key!

another $\frac{k(k+1)(2k+7)}{12}$ and hence we get the lower bound $\frac{k(k+1)(2k+7)}{6} + kr + 1$ for the length of the maximal chain. From this we get the claim by a direct calculation. □

Algorithm 3 is very well suited for practical purposes. We give some experimental data of its behavior in section 5.

4.6 Enhanced Version of the Algorithms

We show how to improve the worst case behavior of the algorithm. Note that choosing a key that is neither of the two columns but from a third column is *in general* not a viable option. While this may decrease the overall cost, it may also decrease security. Consider an example where columns A and B are joined under C's key.

$\textsc{Join}(E, C)$: $E \leftarrow C$
$\textsc{Join}(D, C)$: $D \leftarrow C$
$\textsc{Join}(A, B)$: $A \leftarrow C$, $B \leftarrow C$

Clearly, if this schedule continues with

$\textsc{Join}(B, C)$
$\textsc{Join}(A, C)$

then the overall cost is optimal, but the operation is speculative in terms of security. If the schedule does not continue, the adversary is given more information. All columns are encrypted under the same key. He now can use all of them for cryptanalysis.

In the alternative where we replace the third join operation with

$\textsc{Join}(A, B)$: $A \leftarrow B$

there are two remaining, disjunct keys: one for C, D, E and one for A, B. Clearly, this complicates cryptanalysis. Choosing one of the two keys of the joined columns always yields the minimal amount of ciphertexts for cryptanalysis, since at least one re-encryption is necessary in order to perform the join.

As a consequence we only consider re-encryption selection algorithms as *admissible* that guarantee that two columns have different keys unless there is a chain of (previous) joins which links these two columns.

Now for our enhanced algorithm we group columns not by the fact that they share a common key but by the fact that there is a chain of previous join operations that links one column to another. To distinguish this from the *groups* (cf. section 4.4) we considered before we will call a *cluster* of columns (at any given time) the set of columns that is connected w.r.t. previous joins. Note that clusters are unions of groups. Let's call a *cluster key* the (common) key of the largest group in a cluster.

We modify our data structures and algorithms to be able to account for cluster keys by introducing the additional column

– *ClusterKeyId*: The identifier of key associated to the cluster this column belongs to.

in the table *Columns*. During initialization the cluster key of a column gets the same value as the key: Hence the insert statement of Algorithm 1 now reads

INSERT *columnId*, *cost*, *keyId*, *keyId* INTO *Columns*

Yet another algorithm (similar to Algorithm 2) defines a function GETCLUSTERKEY to extract the *ClusterKeyId* for a column.

Algorithm 4. Column Selection (enhanced)

```
    function JOIN2(columnA, columnB)
        if GETKEY(columnA) = GETKEY(columnB) then
            return null
        end if
 5:     keyA ← GETCLUSTERKEY(columnA)
        keyB ← GETCLUSTERKEY(columnB)
        rankA ← SELECT rank FROM Keys WHERE keyId = keyA
        rankB ← SELECT rank FROM Keys WHERE keyId = keyB
        if rankA > rankB then
10:         lower ← columnB
            (lowerKey, lowerRk) ← (keyB, rankB)
            upper ← columnA
            (upperKey, upperRk) ← (keyA, rankA)
        else
15:         lower ← columnA
            (lowerKey, lowerRk) ← (keyA, rankA)
            upper ← columnB
            (upperKey, upperRk) ← (keyB, rankB)
        end if
20:     lowerCost ← SELECT SUM(cost) FROM Columns WHERE clusterKeyId =
    lowerKey
        UPDATE Keys SET rank = lowerRk − lowerCost WHERE keyId = lowerKey
        UPDATE Keys SET rank = upperRk + lowerCost WHERE keyId = upperKey
        UPDATE Columns SET keyId = upperKey WHERE columnId = lower
        UPDATE Columns SET clusterKeyId = upperKey WHERE clusterKeyId =
    lowerKey
25:     if lowerRk − lowerCost = 0 then
            DELETE FROM Keys WHERE keyId = lowerKey
        end if
        if GETKEY(upper) = GETCLUSTERKEY(upper) then
            return lower
30:     end if
        UPDATE Columns SET keyId = upperKey WHERE columnId = upper
        return (lower, upper)
    end function
```

The main change in the column selection algorithm (cf. Algorithm 4) is that now it may return two columns which shall be re-encrypted. By keeping track of the cluster a column belongs to we can *without* degrading the security re-encrypt both columns of a join to the cluster key of higher rank (which they will eventually have anyway). This means there need not be a steady column any more.

4.7 Analysis of the Enhanced Algorithm

Consider the function T defined by $T(1) = 0$ and

$$T(n) = T\left(\lfloor \tfrac{n}{2} \rfloor\right) + T\left(\lceil \tfrac{n}{2} \rceil\right) + \lfloor \tfrac{n}{2} \rfloor \qquad \text{for } n = 2, 3, \ldots \tag{1}$$

or, more explicitly,

$$T(2i) = 2T(i) + i \tag{2}$$

$$T(2i+1) = T(i) + T(i+1) + i \quad . \tag{3}$$

Remark. Using these recursions for T one can also easily prove that $T(n) = \sum_{i<n} Q(i)$ where $Q(i)$ is the sum of the digits of i in its binary expansion. We omit the details here because we will not need this representation in the sequel.

Lemma 1. *For all $n \geq 1$ we have $T(n) \leq \frac{n}{2} \log_2 n$ where equality holds iff n is a power of 2.*

Proof. From the equations (2) and (3) we directly derive the claim by induction because $2(\frac{1}{2} i \log_2 i) + i = i\big((\log_2 i) + 1\big) = \frac{1}{2}(2i)\log_2(2i)$ and (using Jensen's inequality applied to the concave function $x \mapsto x \log_2 x$)

$$\tfrac{(i \log_2 i + (i+1)\log_2(i+1))}{2} \leq \tfrac{i+(i+1)}{2} \log_2 \tfrac{i+(i+1)}{2} = \tfrac{2i+1}{2}(\log_2(2i+1) - 1)$$

hence

$$\tfrac{1}{2} i \log_2 i + \tfrac{1}{2}(i+1)\log_2(i+1) + i < \tfrac{1}{2}(2i+1)\log_2(2i+1) \quad .$$

Moreover, it is clear that equality in $T(n) \leq \frac{n}{2} \log_2 n$ holds iff we never have to the recursion (3), i.e. iff n is a power of 2. □

The significance of the function T arises from

Theorem 2. *For any proxy re-encryption algorithm which is admissible in the sense of section 4.6 there is a schedule on n columns that needs at least $T(n)$ re-encryptions, i.e. its asymptotic behavior is $O(n \log n)$.*

Proof. We divide the columns into two sets of sizes $\lfloor \frac{n}{2} \rfloor$ and $\lceil \frac{n}{2} \rceil$. For either of them we have a worst case schedule with at least $T(\lfloor \frac{n}{2} \rfloor)$ resp. $T(\lceil \frac{n}{2} \rceil)$ re-encryptions. After concatenating these two schedules we get a schedule which still ends in two clusters with *different* keys[2] and which can be extendend by another $\lfloor \frac{n}{2} \rfloor$ joins. Hence we end up with a schedule that by equation (1) requires $T(n)$ re-encryptions. □

[2] This is where we use admissibility!

Theorem 3. *Algorithm 4 applied to n columns needs at most $T(n)$ re-encryptions. Hence it has the optimal worst-case behavior.*

The proof of this theorem uses another lemma:

Lemma 2. *For all $n \geq 2$ we have*

$$T(n) = \max_{1 \leq i \leq n/2} \big(T(i) + T(n-i) + i\big) \quad . \tag{4}$$

Proof (Theorem). Let $\widetilde{T}(n)$ be the number of re-encryption for the worst schedule on n columns. To show that in fact $\widetilde{T} = T$ we look at the last step where there are two clusters of sizes i and $n-i$ respectively (for some $i \leq \frac{n}{2}$). Then

$$\widetilde{T}(n) = \max_{1 \leq i \leq n/2} \big(\widetilde{T}(i) + \widetilde{T}(n-i) + i\big) \qquad (\text{for } n \geq 2).$$

because the columns in the cluster of size $n-i$ contribute with $\widetilde{T}(n-i)$ re-encryptions no matter if these are executed before or after the two clusters have been joined. Columns of the cluster of size i contribute with at most $\widetilde{T}(i)$ re-encryptions before the two clusters are joined and with i re-encryptions after the two clusters are joined. Comparing with Lemma 2 shows that $\widetilde{T}$ satisfies the same recursion as T. □

Proof (Lemma). The proof is again by induction: For $1 \leq i < \lfloor \frac{n}{2} \rfloor$ we calculate

$$\begin{aligned}
&T(i) + T(n-i) + i \\
&= T\left(\left\lfloor \tfrac{i}{2} \right\rfloor\right) + T\left(\left\lceil \tfrac{i}{2} \right\rceil\right) + \left\lfloor \tfrac{i}{2} \right\rfloor + T\left(\left\lfloor \tfrac{n-i}{2} \right\rfloor\right) + T\left(\left\lceil \tfrac{n-i}{2} \right\rceil\right) + \left\lfloor \tfrac{n-i}{2} \right\rfloor + i \\
&= T\left(\left\lfloor \tfrac{i}{2} \right\rfloor\right) + T\left(\left\lceil \tfrac{n-i}{2} \right\rceil\right) + \left\lfloor \tfrac{i}{2} \right\rfloor + T\left(\left\lceil \tfrac{i}{2} \right\rceil\right) + T\left(\left\lfloor \tfrac{n-i}{2} \right\rfloor\right) + \left\lfloor \tfrac{n+i}{2} \right\rfloor \\
&= T\left(\left\lfloor \tfrac{i}{2} \right\rfloor\right) + T\left(\left\lceil \tfrac{n-i}{2} \right\rceil\right) + \left\lfloor \tfrac{i}{2} \right\rfloor + T\left(\left\lfloor \tfrac{i+1}{2} \right\rfloor\right) + T\left(\left\lceil \tfrac{n-i-1}{2} \right\rceil\right) + \left\lfloor \tfrac{n+i}{2} \right\rfloor \\
&= T\left(\left\lfloor \tfrac{i}{2} \right\rfloor\right) + T\left(\left\lceil \tfrac{n-i}{2} \right\rceil\right) + \left\lfloor \tfrac{i}{2} \right\rfloor + T\left(\left\lfloor \tfrac{i+1}{2} \right\rfloor\right) + T\left(\left\lceil \tfrac{n-i-1}{2} \right\rceil\right) + \left\lfloor \tfrac{i+1}{2} \right\rfloor \\
&\quad - \left\lfloor \tfrac{i+1}{2} \right\rfloor + \left\lfloor \tfrac{n+i}{2} \right\rfloor \\
&\leq T\left(\left\lfloor \tfrac{i}{2} \right\rfloor + \left\lceil \tfrac{n-i}{2} \right\rceil\right) + T\left(\left\lfloor \tfrac{i+1}{2} \right\rfloor + \left\lceil \tfrac{n-i-1}{2} \right\rceil\right) - \left\lfloor \tfrac{i+1}{2} \right\rfloor + \left\lfloor \tfrac{n+i}{2} \right\rfloor
\end{aligned}$$

Note that in the last step we use the induction hypothesis (and furthermore use that $\lfloor \frac{i+1}{2} \rfloor < \lceil \frac{n-i-1}{2} \rceil$ because $i < \lfloor \frac{n}{2} \rfloor$).

Now $\lfloor \frac{i}{2} \rfloor + \lceil \frac{n-i}{2} \rceil = \lfloor \frac{n}{2} \rfloor$ unless n is odd and i is even (in which case the value is $\lceil \frac{n}{2} \rceil$), and likewise $\lfloor \frac{i+1}{2} \rfloor + \lceil \frac{n-i-1}{2} \rceil = \lfloor \frac{n}{2} \rfloor$ unless n is odd and $i+1$ is even (where again the value is $\lceil \frac{n}{2} \rceil$).

In any case one of the expressions $\lfloor \frac{i}{2} \rfloor + \lceil \frac{n-i}{2} \rceil$ and $\lfloor \frac{i+1}{2} \rfloor + \lceil \frac{n-i-1}{2} \rceil$ equals $\lfloor \frac{n}{2} \rfloor$ and the other equals $\lceil \frac{n}{2} \rceil$. Hence finally

$$T(i) + T(n-i) + i \leq T\left(\left\lfloor \tfrac{n}{2} \right\rfloor\right) + T\left(\left\lceil \tfrac{n}{2} \right\rceil\right) + \left\lfloor \tfrac{n}{2} \right\rfloor$$

because $-\lfloor \frac{i+1}{2} \rfloor + \lfloor \frac{n+i}{2} \rfloor \leq \lfloor \frac{n}{2} \rfloor$. □

4.8 Multiple Simultaneous Joins

In queries like `SELECT a.b, c.d, e.f FROM a, c, e WHERE a.x = c.z AND b.y = c.z` we need to re-encrypt multiple columns. As already stated we simulate this by joining pairs of columns one after the other. Given a set of k columns that need to be compared for the join, we chose a column corresponding to a key with the highest occuring rank. Then we form pairs consisting of this column and all the other columns successively. Since both our algorithms encrypt "towards the higher rank" this ensures that all columns have the same key as the chosen column after $k-1$ executions of the algorithm. The number of proxy re-encryptions thereby is bounded by $k-1$ for the Algorithm 3 and by k for the Algorithm 4 (since each of the columns is re-encrypted at most once).

4.9 Non-uniform Costs

So far we have assumed uniform costs of 1 for each column, but some columns may be easier to re-encrypt than others. Particularly, the re-encryption cost is linear in the number of elements per column. This means, that it is easier to re-encrypt two columns of size 1 and 2, respectively, than one column of size 4.

We now consider element-wise re-encryption costs by incorporating *non-uniform column costs,* e.g. the size of the column, in our algorithms. Simply, initialize the columns with their respective costs in Algorithm 1. This may significantly reduce the overall costs. Following the example above, consider columns A, B and C of respective sizes 2, 1 and 4 and the join schedule

$\textsc{Join}(A, B)$
$\textsc{Join}(C, A)$

Uniform costs may suggest the following re-encryptions: $A \leftarrow B$, $C \leftarrow A$. This results in 6 element re-encryptions. The worst possible performance for any set of re-encryptions. Instead non-uniform costs using column sizes dictate these re-encryptions: $B \leftarrow A$, $A \leftarrow C$. The result are 3 element re-encryptions. Furthermore, the maximum number of element re-encryptions using either of our algorithms in this example is 4. This is also the minimum worst-case cost under any schedule of join operations.

It is therefore important to note that the analysis of minimum worst-case cost of re-encryption of Section 4 in the general case remains intact. We always achieve the best worst-case cost assuming any future schedule of join operations. To see this, view a column with non-uniform cost c as a group of c columns with uniform cost 1 that always operate successively. Let N be the sum of the costs of all columns, then our algorithm incurs costs of at most $O(N \log N)$.

Nevertheless, in some cases of non-uniform costs we may perform too many re-encryptions for a specific schedule resulting in sub-optimal costs, since the future schedule is unknown. Consider columns A, B, C and D of sizes 1, 5, 2 and 3 and the following join schedule

JOIN(A, B): $A \leftarrow B$
JOIN(C, D): $C \leftarrow D$
JOIN(A, C)

In the third join our algorithm dictates $C \leftarrow A$ leading to 5 element re-encryptions. This clearly leads to the minimal costs of also 5 for a future

JOIN(B, C)

but in case there is no future join, costs are not optimal. It would be more efficient to re-encrypt as $A \leftarrow C$ resulting in a cost of 4 element re-encryptions. Yet, choosing to re-encrypt as $A \leftarrow C$ will increase the worst-case cost under many future join schedules. We therefore choose to optimize the worst-case cost where our bound is tight.

The number of elements in a column may vary, because rows may be inserted or deleted. This further complicates the analysis and possible algorithms also have to account for these future operations. We leave this as future work and currently assume fixed non-uniform costs.

5 Experiments

We performed a number of experiments in order to measure the difference between the best re-encryption cost and our Algorithm 3. We chose $n = 8$ columns and a join schedule of length $m = 16$. Note that we need to find the optimum schedule in $2^m = 65536$ options.

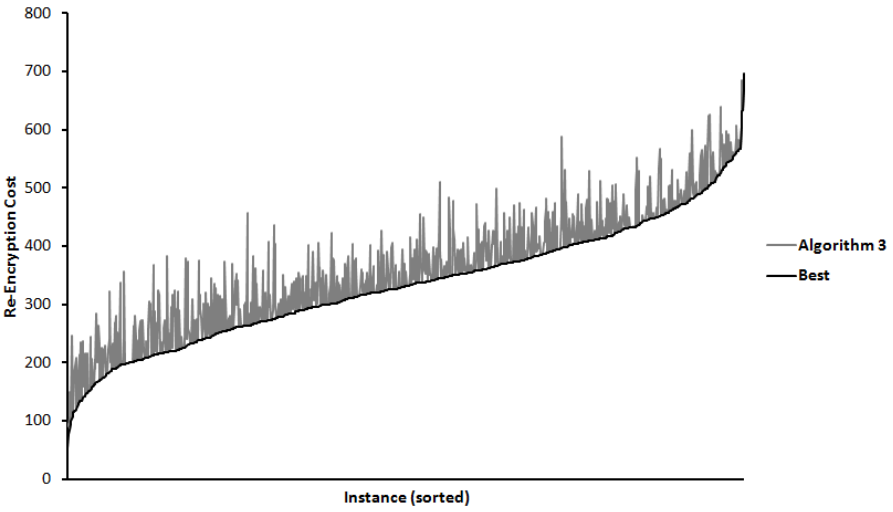


Fig. 1. Cost difference between optimum and Algorithm 3

We chose the schedule uniformly random among all possible pairs of different columns. Pairs could occur repeatedly. We chose the cost for each column uniformly random between 1 and 100.

We performed 1000 experiments. In each we recorded the optimum cost and the cost of our Algorithm 3. In 41% of the experiments our algorithm delivered the optimum schedule. The mean difference to the optimum was 26, i.e. roughly half an average column cost. The maximum difference was 193 and the median difference was 9, such that few large sub-optimal cases account for the majority of the difference.

In Figure 1 we depict our results. We have sorted all experimental results in increasing optimal cost (black line). The gray line depicts the corresponding cost of our algorithm.

6 Conclusion

In this paper we have considered the problem of selecting a column for re-encryption in a deterministically, adjustably encrypted database. To the best of our knowledge this is the first paper considering this problem. We have provided an algorithm that achieves the best possible worst-case bound and a simpler algorithm that performs very well in experimental settings.

References

1. Bellare, M., Boldyreva, A., O'Neill, A.: Deterministic and efficiently searchable encryption. In: Menezes, A. (ed.) CRYPTO 2007. LNCS, vol. 4622, pp. 535–552. Springer, Heidelberg (2007)
2. Binnig, C., Hildenbrand, S., Färber, F.: Dictionary-based order-preserving string compression for main memory column stores. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD) (2009)
3. Blaze, M., Bleumer, G., Strauss, M.J.: Divertible protocols and atomic proxy cryptography. In: Nyberg, K. (ed.) EUROCRYPT 1998. LNCS, vol. 1403, pp. 127–144. Springer, Heidelberg (1998)
4. Boldyreva, A., Chenette, N., Lee, Y., O'Neill, A.: Order-preserving symmetric encryption. In: Joux, A. (ed.) EUROCRYPT 2009. LNCS, vol. 5479, pp. 224–241. Springer, Heidelberg (2009)
5. James, G.: The representation theory of the symmetric groups. LNM 682. Springer (1978)
6. Galler, B., Fischer, M.: An improved equivalence algorithm. Communications of the ACM 7(5) (1964)
7. Hacigümüs, H., Iyer, B., Li, C., Mehrotra, S.: Executing SQL over encrypted data in the database-service-provider model. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD) (2002)
8. Pohlig, S., Hellman, M.: An improved algorithm for computing logarithms over GF(p) and its cryptographic significance. IEEE Transactions on Information Theory 24 (1978)
9. Popa, R., Redfield, C., Zeldovich, N., Balakrishnan, H.: CryptDB: Protecting confidentiality with encrypted query processing. In: Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP) (2011)
10. Popa, R., Zeldovich, N.: Cryptographic treatment of CryptDB's adjustable join. Technical Report MIT-CSAIL-TR-2012-006 (2012)