Order Preserving Encryption for Numeric Data

Rakesh Agrawal Jerry Kiernan Ramakrishnan Srikant Yirong Xu

IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120

ABSTRACT

Encryption is a well established technology for protecting sensitive data. However, once encrypted, data can no longer be easily queried aside from exact matches. We present an order-preserving encryption scheme for numeric data that allows any comparison operation to be directly applied on encrypted data. Query results produced are sound (no false hits) and complete (no false drops). Our scheme handles updates gracefully and new values can be added without requiring changes in the encryption of other values. It allows standard database indexes to be built over encrypted tables and can easily be integrated with existing database systems. The proposed scheme has been designed to be deployed in application environments in which the intruder can get access to the encrypted database, but does not have prior domain information such as the distribution of values and cannot encrypt or decrypt arbitrary values of his choice. The encryption is robust against estimation of the true value in such environments.

1. INTRODUCTION

Database systems typically offer access control as the means to restrict access to sensitive data. This mechanism protects the privacy of sensitive information provided data is accessed using the intended database system interfaces. However, access control, while important and necessary, is often insufficient. Attacks upon computer systems have shown that information can be compromised if an unauthorized user simply gains access to the raw database files, bypassing the database access control mechanism altogether. For instance, a recent article published in the Toronto Star [14] describes an incident where a disk containing the records of several hundred bank customers was being auctioned on eBay. The bank had inadvertently sold the disk to the eBay re-seller as used equipment without deleting its contents. Drawing upon privacy legislations and guidelines worldwide, Hippocratic databases also identify the protection of personal data from unauthorized acquisition as a vital requirement [1].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD 2004 June 13-18, 2004, Paris, France.
Copyright 2004 ACM 1-58113-859-8/04/06 . . . $5.00.

Encryption is a well established technology for protecting sensitive data [7] [22] [24]. Unfortunately, the integration of existing encryption techniques with database systems causes undesirable performance degradation. For example, if a column of a table containing sensitive information is encrypted, and is used in a query predicate with a comparison operator, an entire table scan would be needed to evaluate the query. The reason is that the current encryption techniques do not preserve order and therefore database indices such as B-tree can no longer be used. Thus the query execution over encrypted databases can become unacceptably slow.

We present an encryption technique called OPES (Order Preserving Encryption Scheme) that allows comparison operations to be directly applied on encrypted data, without decrypting the operands. Thus, equality and range queries as well as the MAX, MIN, and COUNT queries can be directly processed over encrypted data. Similarly, GROUP BY and ORDER BY operations can also be applied. Only when applying SUM or AVG to a group do the values need to be decrypted. OPES is also endowed with the following properties:

- The results of query processing over data encrypted using OPES are exact. They neither contain any false positives nor miss any answer tuple. This feature of OPES sharply differentiates it from schemes such as [13] that produce a superset of answer, necessitating filtering of extraneous tuples in a rather expensive and complex post-processing step.
- OPES handles updates gracefully. A value in a column can be modified or a new value can be inserted in a column without requiring changes in the encryption of other values.
- OPES can easily be integrated with existing database systems as it has been designed to work with the existing indexing structures such as B-trees. The fact that the database is encrypted can be made transparent to the applications.

Measurements from an implementation of OPES in DB2 show that the time and space overhead of OPES are reasonable for it to be deployed in real systems.

1.1 Estimation Exposure

The security of an encryption scheme is conventionally assessed by analyzing whether an adversary can find the key used for encryption. See [22] [24] for a categorization of different levels of attacks against a cryptosystem.

When dealing with sensitive numeric data, an adversary does not have to determine the exact data value p corresponding to an

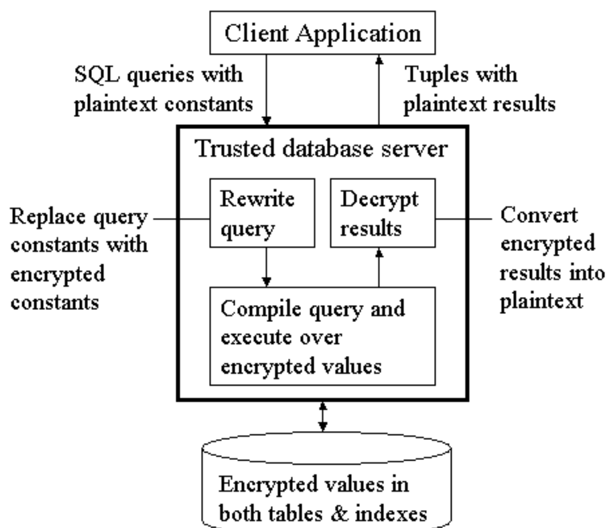


Figure 1: Transparent encryption in the "trusted database software with vulnerable storage" setting.

encrypted value c; a breach may occur if the adversary succeeds in obtaining a tight estimate of p. For a numeric domain P, if an adversary can estimate with $c\%$ confidence that a data value p lies within the interval $[p_1, p_2]$ then the interval width $(p_2 - p_1)$/domain-width(P) defines the amount of estimation exposure at $c\%$ confidence level.

Clearly, any order-preserving encryption scheme is vulnerable to tight estimation exposure if the adversary can choose any number of unencrypted (encrypted) values of his liking and encrypt (decrypt) them into their corresponding encrypted (unencrypted) values. Similarly, any order-preserving encryption is not secure against tight estimation exposure if the adversary can guess the domain and knows the distribution of values in that domain.

We consider an application environment where the goal is safety from an adversary who has access to all (but only) encrypted values (the so called *ciphertext only* attack [22] [24]), and does not have any special information about the domain. We will particularly focus on robustness against estimation exposure.

1.2 Threat Model

We assume (see Figure 1):

- *The storage system used by the database software is vulnerable to compromise.* While current database systems typically perform their own storage management, the storage system remains part of the operating system. Attacks against storage could be performed by accessing database files following a path other than through the database software, or in the extreme, by physical removal of the storage media.
- *The database software is trusted.* We trust the database software to transform query constants into their encrypted values and decrypt the query results. Similarly, we assume that an adversary does not have access to the values in the memory of the database software.
- *All disk-resident data is encrypted.* In addition to the data values, the database software also encrypts schema information such as table and column names, metadata such as column statistics, as well as values written to recovery logs. Otherwise, an adversary may be able to use this information to guess data distributions.

1.3 Pedagogical Assumptions and Notations

The focus of this paper is on developing order-preserving encryption techniques for numeric values and assumes conventional encryption [22] [24] for other data types as well as for encrypting information such as schema names and metadata. We will sometime refer to unencrypted data values as plaintext. Similarly, encrypted values will also be referred to as ciphertext.

We will assume that the database consists of a single table, which in turn consists of a single column. The domain of the column will be initially assumed to be a subset of integer values, $[p_{\min}, p_{\max})$. The extension for real values is given later in the paper.

Assume the database $\tilde{P}$ consists of a total of $|\tilde{P}|$ plaintext values. Out of these, $|P|$ values are unique, which will be represented as $P = p_1, p_2, \ldots, p_{|P|}, p_i < p_{i+1}$ The corresponding encrypted values will be represented as $C = c_1, c_2, \ldots, c_{|P|}, c_i < c_{i+1}$.

Duplicates can sometimes be used to guess the distribution of a domain, particularly if the distribution is highly skewed. A closely related problem is that if the number of distinct values is small (e.g., day of the month), it is easy to guess the domain. We will initially assume that the domain to be encrypted either does not contain many duplicates or contains a distribution that can withstand a duplicate attack, and discuss the handling of duplicates later in the paper.

1.4 Paper Layout

The rest of the paper is organized as follows. We first discuss related work in Section 2. We give an overview of OPES in Section 3. The next three sections give details of the three main phases of OPES. We describe extensions to handle real values and duplicates in Section 7. In Section 8, we study the quality of the encryption produced by OPES and present performance measurements from a DB2 implementation. We conclude with a summary and directions for future work in Section 9.

2. RELATED WORK

Summation of Random Numbers A simple scheme has been proposed in [3] that computes the encrypted value c of integer p as $c = \sum_{j=0}^{p} R_j$, where R_j is the jth value generated by a secure pseudo-random number generator R. Unfortunately, the cost of making p calls to R for encrypting or decrypting c can be prohibitive for large values of p.

A more serious problem is the vulnerability to estimation exposure. Since the expected gap between two encrypted values is proportional to the gap between the corresponding plaintext values, the nature of the plaintext distribution can be inferred from the encrypted values. Figure 2 shows the distributions of encrypted values obtained using this scheme for data values sampled from two different distributions: Uniform and Gaussian. In each case, once both the input and encrypted distributions are scaled to be between 0 and 1, the number of points in each bucket is almost identical for the plaintext and encrypted distributions. Thus the percentile of a point in the encrypted distribution is also identical to its percentile in the plaintext distribution.

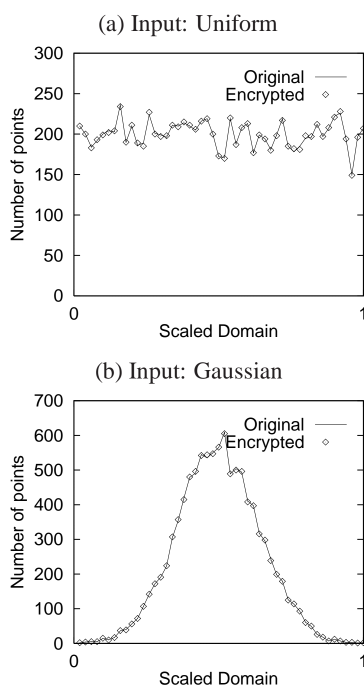


Figure 2: Summation of random numbers: Distribution of encrypted values tracks the input distribution.

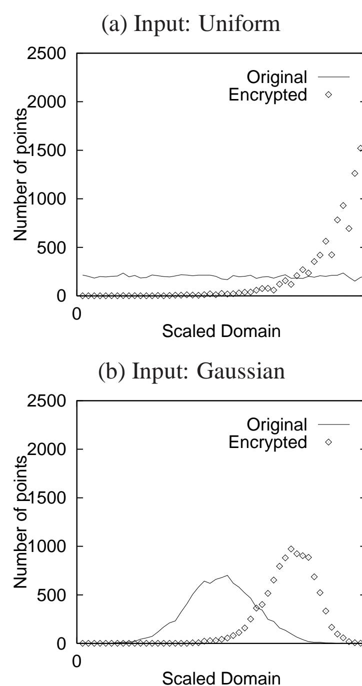


Figure 3: Polynomial functions: Encryption of different input distributions look different.

Polynomial Functions In [12], a sequence of strictly increasing polynomial functions is used for encrypting integer values while preserving their order. These polynomial functions can simply be of the first or second order, with coefficients generated from the encryption key. An integer value is encrypted by applying the functions in such a way that the output of a function becomes the input of the next function. Correspondingly, an encrypted value is decrypted by solving these functions in reverse order. However, this encryption method does not take the input distribution into account. Therefore the shape of the distribution of encrypted values depends on the shape of the input distribution, as shown in Figure 3 for the encryption function given in Example 10 in [12]. This illustration suggests that this scheme may reveal information about the input distribution, which can be exploited.

Bucketing In [13], tuples are encrypted using conventional encryption, but an additional bucket id is created for each attribute value. This bucket id, which represents the partition to which the unencrypted value belongs, can be indexed. The constants appearing in a query are replaced by their corresponding bucket ids. Clearly, the result of a query will contain false hits that must be removed in a post-processing step after decrypting the tuples returned by the query. This filtering can be quite complex since the bucket ids may have been used in joins, subqueries, etc. The number of false hits depends on the width of the partitions involved. It is shown in [13] that the post-processing overhead can become excessive if a coarse partitioning is used for bucketization. On the other hand, a fine partitioning makes the scheme vulnerable to estimation exposure, particularly if an equi-width partitioning is used.

It has been pointed out in [6] that the indexes proposed in [13] can open the door to interference and linking attacks. Instead, they build a B-tree over plaintext values, but then encrypt every tuple and the B-tree at the node level using conventional encryption. The advantage of this approach is that the content of B-tree is not visible to an untrusted database server. The disadvantage is that the B-tree traversal can now be performed by the front-end only by executing a sequence of queries that retrieve tree nodes at progressively deeper level.

Other Relevant Work Rivest et al. [21] suggest that the limit on manipulating encrypted data arises from the choice of encryption functions used, and there exist encryption functions that permit encrypted data to be operated on directly for many sets of interesting operations. They call these functions “privacy homomorphisms”. The focus of [21] and the subsequent follow-up work [2] [8] [9] has been on designing privacy homomorphisms to enable arithmetic on encrypted data, but the comparison operations were not investigated in this line of research.

In [10], a simple but effective scheme has been proposed to encrypt a look-up directory consisting of (key, value) pairs. The goal is to allow the corresponding value to be retrieved if and only if a valid key is provided. The essential idea is to encrypt complete tuples, but associate with every tuple the one-way hash value of its key. Thus, no tuple will be retrieved if an invalid key is presented. Answering range queries was not a goal of this system.

In [23], interesting schemes are proposed to support keyword searches over an encrypted text repository. The driving application for this work is the efficient retrieval of encrypted email messages. Naturally, they do not discuss relational queries and it is not clear how their techniques can be adapted for relational databases.

In [4], a smart card with encryption and query processing capabilities is used to ensure the authorized and secure retrieval of encrypted data stored on untrusted servers. Encryption keys are

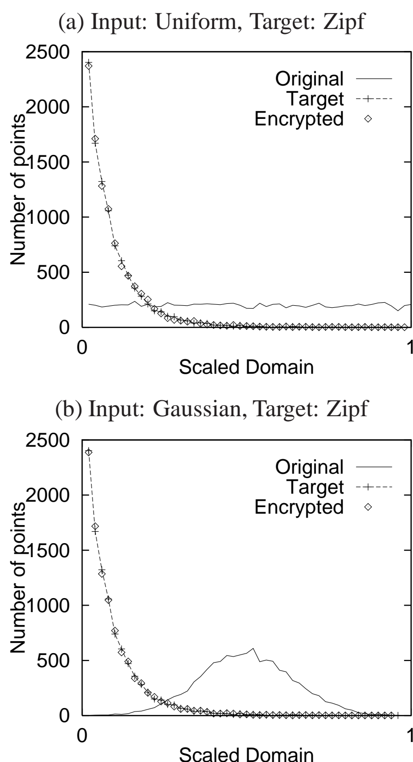


Figure 4: Illustrating OPES.

maintained on the smart card. The smart card can translate exact match queries into equivalent queries over encrypted data. However, range queries require creating a disjunction for every possible value in the range, which is infeasible for real data values. The smart card implementation could benefit from our encryption scheme in that range queries could be translated into equivalent queries over encrypted data.

In [25], the security and tamper resistance of a database stored on a smart card is explored. They consider snooping attacks for secrecy, and spoofing, splicing, and replay attacks for tamper resistance. Retrieval performance is not the focus of their work and it is not clear how much of their techniques apply to general purpose databases not stored in specialized devices.

Amongst commercial database products, Oracle 8i allows values in any of the columns of a table to be encrypted [18]. However, the encrypted column can no longer participate in indexing as the encryption is not order-preserving.

Related work also includes research on order-preserving hashing [5] [11]. However, protecting the hash values from cryptanalysis is not the concern of this body of work. Similarly, the construction of original values from the hash values is not required.

3. PROPOSED ORDER-PRESERVING ENCRYPTION SCHEME

The basic idea of OPES is to take as input a user-provided target distribution and transform the plaintext values in such a way that the transformation preserves the order while the transformed values follow the target distribution. Figure 4 shows the result of running OPES with different input distributions and the same target distribution. Notice that the distribution of encrypted values looks identical in both 4(a) and 4(b), even though the input distributions were very different.

3.1 Intuition

To understand the intuition behind OPES algorithm, consider the following encryption scheme:

Generate $|P|$ unique values from a user-specified target distribution and sort them into a table T. The encrypted value c_i of p_i is then given by $c_i = T[i]$. That is, the ith plaintext value in the sorted list of $|P|$ plaintext values is encrypted into the ith value in the sorted list of $|P|$ values obtained from the target distribution. The decryption of c_i requires a lookup into a reverse map. Here T is the encryption key that must be kept secret.

Clearly, this scheme does not reveal any information about the original values apart from the order, since the encrypted values were generated solely from the user-specified target distribution, without using any information from the original distribution. Even if an adversary has all of the encrypted values, he cannot infer T from those values. By appropriately choosing target distribution, the adversary can be forced to make large estimation errors.

This simple scheme, while instructive, has the following shortcomings for it to be used for encrypting large databases:

- The size of encryption key is twice as large as the number of unique values in the database.
- Updates are problematic. When adding a new value p, where $p_i < p < p_{i+1}$, we will need to re-encrypt all $p_j, j > i$.[1]

OPES has been designed such that the result of encryption is statistically indistinguishable from the one obtained using the above scheme, thereby providing the same level of security, while removing its shortcomings.

3.2 Overview of OPES

When encrypting a given database P, OPES makes use of all the plaintext values currently present P,[2] and also uses a database of sampled values from the target distribution. Only the encrypted database C is stored on disk. At the same time, OPES also creates some auxiliary information $\mathcal{K}$, which the database system uses to decrypt encoded values or encrypt new values. Thus $\mathcal{K}$ serves the function of the encryption key. This auxiliary information is kept encrypted using conventional encryption techniques.

OPES works in three stages:

1. *Model:* The input and target distributions are modeled as piece-wise linear splines.
2. *Flatten:* The plaintext database P is transformed into a "flat" database F such that the values in F are uniformly distributed.

[1] It is possible to avoid immediate re-encryption by choosing an encrypted value for p in the interval (c_i, c_{i+1}), but T would still need updating. Moreover, there might be cases where $c_{i+1} = c_i+1$ and therefore inserting a new value will require re-encryption of existing values.
Note that the encryption scheme $c_i = T[p_i]$ circumvents the update problem. But now the size of the key becomes the size of the domain. It is also vulnerable to percentile exposure, as discussed earlier in Section 2.

[2] If an installation is creating a new database, the database administrator can provide a sample of expected values.

3. *Transform:* The flat database F is transformed into the cipher database C such that the values in C are distributed according to the target distribution.

Note that

$$p_i < p_j \Longrightarrow f_i < f_j \Longrightarrow c_i < c_j.$$

We give details of the three stages in Sections 4, 5 and 6 respectively.

4. MODELING THE DISTRIBUTIONS

Techniques for modeling data distributions have been studied extensively in the database literature in the context of estimating the costs of different query execution plans. As stated in [16], there are two broad categories of techniques: histogram-based that capture statistical information about a distribution by means of counters for a specified number of buckets, and parametric that approximate a distribution by fitting the parameters of a given type of function. We experimented with several histogram-based techniques [15], including equi-depth, equi-width, and wavelet-based methods, but found that the flattened values obtained were not uniformly distributed unless the number of buckets was selected to be unreasonably large. The main source of the problem was the assumption that the distribution is uniform within each bucket. The parametric are suitable for closed-form distributions but lead to poor estimations for irregular distributions [16], which we expect to be the norm in our application.

We, therefore, resorted to a combination of histogram-based and parametric techniques. As in [16], we first partition the data values into buckets and then model the distribution within each bucket as a linear spline. The spline for a bucket $[p_l, p_h)$ is simply the line connecting the densities at the two end-points of the bucket.[3]

We also allow the width of value ranges to vary across buckets. However, unlike [16], we do not have a given fixed number of buckets. Rather, we use the minimum description length (MDL) principle [20] to determine the number of buckets.

4.1 Bucket Boundaries

The bucket boundaries are determined in two phases:[4]

1. *Growth phase.* The space is recursively split into finer partitions. Each partitioning of a bucket reduces the maximum deviation from the density function within the newly formed buckets when compared to their parent bucket.

2. *Prune phase.* Some buckets are pruned (merged into bigger buckets). The idea is to minimize the number of buckets and yet have the values within buckets after mapping be uniformly distributed. We use the MDL principle to obtain this balance.

The details of these two phases are discussed next.

[3]In [16], the splines are not continuous across buckets; they use linear regression over data values present in a bucket for determining the spline. However, such discontinuities may cause undesirable breaks in the uniformity when we flatten plaintext values.

[4]This procedure is reminiscent of the procedure for building decision tree classifiers, and in particular SLIQ [17], but the details are quite different.

4.2 Growth Phase

We are given a bucket $[p_l, p_h)$, with $h - l - 1$ (sorted) points: $\{p_{l+1}, p_{l+2}, \ldots, p_{h-1}\}$. We first find the linear spline for this bucket. Next, for each point p_s in the bucket, we compute its expected value if the points were distributed according to the density distribution modeled by the linear spline (i.e., the expected value of the $(s - l)$th smallest value in a set of $h - l - 1$ random values drawn from the distribution). We then split the bucket at the point that has the largest deviation from its expected value (breaking ties arbitrarily). We stop splitting when the number of points in a bucket is below some threshold, say, 10.

4.3 Prune Phase

The MDL principle [20] states that the best model for encoding data is the one that minimizes the sum of the cost of describing the model and the cost of describing the data in terms of that model. For a given bucket $[p_l, p_h)$, the local benefit LB of splitting this bucket at a point p_s is given by

$$\begin{aligned} \text{LB}(p_l, p_h) &= \text{DataCost}(p_l, p_h) - \text{DataCost}(p_l, p_s) \\ &\quad - \text{DataCost}(p_s, p_h) - \text{IncrModelCost} \end{aligned}$$

where DataCost(p_1, p_2) gives the cost of describing the data in the interval $[p_1, p2)$ and IncrModelCost is the increase in modeling cost due to the partitioning of a bucket into two buckets.

The global benefit GB of splitting this bucket at p_s takes into account the benefit of further recursive splits:

$$\text{GB}(p_l, p_h) = \text{LB}(p_l, p_h) + \text{GB}(p_l, p_s) + \text{GB}(p_s, p_h).$$

If $GB > 0$, the split is retained; otherwise, the split at p_s and all recursive splits within $[p_l, p_h)$ are pruned. Note that we do this computation bottom up, and therefore the cost is linear in the number of splits.[5]

We now provide the functions for the computation of DataCost and IncrModelCost. Assume momentarily the existence of a mapping M that transforms values sampled from a linear density function into a set of uniformly distributed values. We specify M in the next section. As we shall see, M will have two parameters: a quadratic coefficient and a scale factor.

4.3.1 DataCost

We want to flatten a given data distribution into a uniform distribution. So, given a bucket, we first flatten the values present in the bucket using the mapping M, and then compute the cost of encoding the deviations from uniformity for the mapped values.[6]

Let the set of data values $\{p_l, p_{l+1}, \ldots, p_{h-1}\}$ be mapped into $\{f_l, f_{l+1}, \ldots, f_{h-1}\}$ using M. The encoding of a value $p_i \in$

[5]One might wonder why we did not combine pruning with the growth phase and stop splitting a bucket as soon as the local benefit became zero or negative. The reason is that the benefit of partitioning may start showing only at a finer granularity, and it will often be the case that the local benefit is less than zero even though the global benefit is much greater than zero.

[6]Note that our implementation encodes only the statistically significant deviations to avoid overfitting, i.e., rather than a single expected value, we consider the range of values that would occur with a uniform distribution, and only encode values that are outside this range. We omit this detail for brevity.

$[p_l, p_h)$ would cost

$$\text{Cost}(p_i) = \log |f_i - E(i)|,$$

where $E(i)$, the expected value of the ith number assuming uniformity, is given by

$$E(i) = f_l + \frac{i-l}{h-l}(f_h - f_l).$$

The cost of encoding all the values in the interval $[p_l, p_h)$ is then given by

$$\text{DataCost}(p_l, p_h) = \sum_{i=l+1}^{h-1} \text{Cost}(p_i).$$

4.3.2 *IncrModelCost*

If we have m buckets, we need to store $m+1$ boundaries, m quadratic coefficients, and m scale factors. Thus the model cost will be $(3m+1) \times 32$, assuming 32 bits for each of these values. More importantly, the cost of an additional bucket

$$\text{IncrModelCost} = 32 \times 3 = 96.$$

5. FLATTEN

The overall idea of the flatten stage is to map a plaintext bucket B into a bucket B^f in the flattened space in such a way that the length of B^f is proportional to the number of values present in B. Thus, the dense plaintext buckets will be stretched and the sparse buckets will be compressed. The values within a bucket are mapped in such a way that the density will be uniform in the flattened bucket. Since the densities are uniform both inter-bucket and intra-bucket, the values in the flattened database will be uniformly distributed. We specify next a mapping function that accomplishes these goals.

5.1 Mapping Function

OBSERVATION 1. *If a distribution over $[0, p_h)$ has the density function $qp + r$, where $p \in [0, p_h)$, then for any constant $z > 0$, the mapping function*

$$M(p) = z(\frac{q}{2r}p^2 + p)$$

will yield a uniformly distributed set of values. □

This follows from the fact that the slope of the mapping function at any point p is proportional to the density at p:[7]

$$\frac{dM}{dp} = \frac{z}{r}(qp + r) \propto qp + r.$$

We will refer to $s := q/2r$ as the quadratic coefficient. Thus

$$M(p) = z(sp^2 + p)$$

A different scale factor z is used for different buckets, to make the inter-bucket density uniform as well. We describe next how the scale factors are computed.

[7] An equivalent way to think about this is that the space around p, say from $p-1$ to $p+1$ is mapped to a length of

$$M(p+1) - M(p-1) = \frac{2z}{r}(qp + r) \propto qp + r.$$

5.2 Scale Factor

We need to find the scale factor z, one for each bucket B such that:

1. Two distinct values in the plaintext will always map to two distinct values in the flattened space, thereby ensuring incremental updatability.
2. Each bucket is mapped to a space proportional to the number of points n in that bucket, i.e., if w is the width of the bucket and $w^f = M(w)$ is the width after flattening, then $w^f \propto n$.

The first constraint can be written as:

$$\forall p \in [0, w): \; M(p+1) - M(p) \geq 2.$$

The 2 in the RHS (instead of 1) ensures two adjacent plaintext values will be at least 2 apart in the flattened space. As we will explain in Section 5.5, this extra separation makes encryption tolerant to rounding errors in floating point calculations. Expanding M, we get

$$\forall p \in [0, w): \; z \geq 2/(s(2p+1)+1).$$

The largest value of $2/(s(2p+1)+1)$ will be at $p = 0$ if $s \geq 0$, and at $p = w-1$ otherwise. Therefore we get

$$\hat{z} = \begin{cases} 2, & s \geq 0 \\ 2/(1 + s(2w-1)), & s < 0 \end{cases}$$

where $\hat{z}$ denotes the minimum value of z that will satisfy the first constraint.

To satisfy the second constraint, we want

$$w^f = Kn$$

for all the buckets. Define

$$\hat{w}^f = \hat{z}(sw^2 + w)$$

as the minimum width for each bucket, and define

$$K = \max\left[\hat{w}_i^f\right], i = 1, \ldots, m.$$

Then the scale factors

$$z = \frac{Kn}{sw^2 + w}$$

will satisfy both the desired constraints, since $z > \hat{z}$, and $w^f = z(sw^2 + w) = Kn$.

5.3 Encryption Key

Let us briefly review what we have at this stage. The modeling phase has yielded a set of buckets $\{ B_1, \ldots, B_m \}$. For each bucket, we also have a mapping function M, characterized by two parameters: the quadratic coefficient s and the scale factor z. We save the $m+1$ bucket boundaries, the m quadratic coefficients, and the m scale factors in a data structure $\mathcal{K}^f$. The database system uses $\mathcal{K}^f$ to flatten (encrypt) new plaintext values, and also to unflatten (decrypt) a flattened value. Thus $\mathcal{K}^f$ serves the function of the encryption key.

Note that $\mathcal{K}^f$ is computed once at the time of initial encryption of the database. As the database obtains new values, $\mathcal{K}^f$ is used to encrypt them, but it is not updated, which endows OPES with the incremental updatability property.[8]

[8] In order to encrypt stray values outside of the current

5.4 Mapping a Plaintext Value into a Flat Value

Represent the domains of the input database P and the flat database F as $[p_{\min}, p_{\max})$ and $[f_{\min}, f_{\max})$ respectively. Note that

$$f_{\max} = f_{\min} + \sum_{i=1}^{m} w_i^f$$

where $w_i^f = M_i(w_i)$. Recall that w_i is the length of plaintext bucket B_i, and w_i^f the length of the corresponding flat bucket.

To flatten a plaintext value p, we first determine the bucket B_i into which p falls, using the information about the bucket boundaries saved in $\mathcal{K}^f$. Now p is mapped into the flat value f using the equation:

$$f = f_{\min} + \sum_{j=1}^{i-1} w_j^f + M_i(p - p_{\min} - \sum_{j=1}^{i-1} w_j).$$

5.5 Mapping a Flat Value into a Plaintext Value

We can rewrite the previous equation as

$$p = p_{\min} + \sum_{j=1}^{i-1} w_j + M_i^{-1}(f - f_{\min} - \sum_{j=1}^{i-1} w_j^f)$$

where

$$M^{-1}(f) = \frac{-z \pm \sqrt{z^2 + 4zsf}}{2zs}$$

and z and s represent respectively the scale factor and the quadratic coefficient of the mapping function M.[9] So, unflattening requires using the information in $\mathcal{K}^f$ to determine the flat bucket B_i^f in which the given flat value f lies and then applying M^{-1}. Out of the two possible values for M^{-1}, only one will be within the bucket boundary.

Note that $M(p)$, as well as $M^{-1}(f)$, will usually not be integers values, and are rounded to the nearest integer. To remove the possibility of errors due to rounding floating point calculations, we verify whether $M^{-1}(f) = p$ immediately after computing $M(p)$. If it turns out that $M^{-1}(f)$ is actually rounded to $p - 1$, we encrypt p as $f + 1$ instead of f. Since we ensured that two adjacent plaintext values are at least 2 apart in the flattened space when computing the scale factors, $M^{-1}(f + 1)$ will decrypt to p and not to $p + 1$. Similarly, if $M^{-1}(f) = p + 1$, we encrypt p as $f - 1$.

range $[p_{\min}, p_{\max})$, we create two special buckets, $B_0 = [\text{MINVAL}, p_{\min})$ and $B_{m+1} = [p_{\max}, \text{MAXVAL}]$ where [MINVAL, MAXVAL] is the domain of the input distribution. Since these buckets initially do not contain any values, we estimate the s and z parameters for them. The quadratic coefficient s for the buckets is set to 0. To estimate the scale factor for B_0, we extrapolate the scaling used for the two closest points in B_1 into B_0 and define z_0 to be $(f_2 - f_1)/(p_2 - p_1)$. Similarly, the scale factor for B_{m+1} is estimated using the two closest values in buckets B_m To simplify exposition, the rest of the paper ignores the existence of these special buckets.

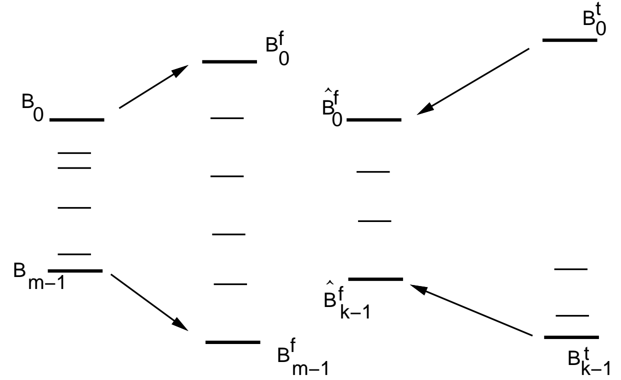
[9] When $|sf|$ is small relative to $|z|$, the computation of $-z + \sqrt{z^2 + 4zsf}$ will result in loss of precision when z is positive (and similarly $-z - \sqrt{z^2 + 4zsf}$ will lose precision when z is negative). Thus we use the alternate formulation [19]

$$M^{-1}(f) = \frac{-2f}{-z \mp \sqrt{z^2 + 4zsf}}$$

in those cases.

1. Compute the buckets used to transform the plaintext distribution into the flattened distribution

2. From the target distribution, compute the buckets for the flattened distribution $\hat{B}^f$.



3. Scale the buckets of the flattened target distribution to equal the range of the flattened distribution computed in Step 1, and scale the target distribution proportionately.

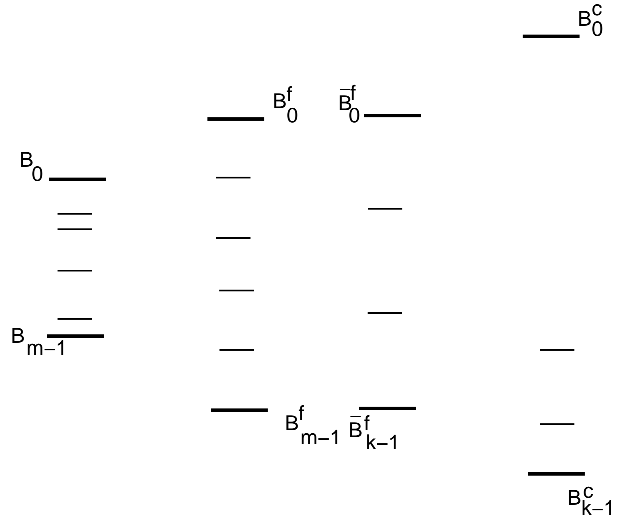


Figure 5: Scaling the target distribution.

6. TRANSFORM

The transform stage is almost a mirror image of the flatten stage. Given a uniformly distributed set of flattened values, we want to map them into the target distribution. An equivalent way of thinking about the problem is that we want to flatten the target distribution into a uniform distribution, while ensuring that the distribution so obtained "lines-up" with the uniform distribution yielded by flattening the plaintext distribution.

Figure 5 portrays this process. We already have on hand the buckets for the plain text distribution (Step 1). We bucketize the target distribution, independent of the bucketization of the plaintext distribution (Step 2). We then scale the target distribution (and the flattened target distribution) in such a way that the width of the uniform distribution generated by flattening the scaled target distribution becomes equal to the width of the uniform distribution generated by flattening the plaintext distribution (Step 3).

We will henceforth refer to the scaled target distribution as the cipher distribution.

6.1 Scaling the Target Distribution

The modeling of target distribution yields a set of buckets $\{ B_1^t, \ldots, B_k^t \}$. For every bucket B^t of length w^t, we also get the mapping function M^t and the associated parameters s^t and z^t. For computing the scale factor z^t for each bucket, we use a procedure similar to the one discussed in Section 5.2, except that the first constraint is flipped. We now need to ensure that two adjacent values in the flat space map to two distinct values in the target space (whereas earlier we had to ensure that two adjacent values in the plaintext space mapped to two distinct values in the flat space).

An analysis similar to Section 5.2 yields

$$z^t = \frac{K^t n^t}{s^t (w^t)^2 + w^t}$$

where

$$K^t = \min \left[\frac{\hat{z}_i^t (s_i^t (w_i^t)^2 + w_i^t)}{n_i^t} \right], i = 1, \ldots, k$$

and

$$\hat{z}^t = \begin{cases} 0.5/(1 + s^t(2w^t - 1)), & s^t > 0 \\ 0.5, & s^t \leq 0. \end{cases}$$

Let $\hat{B}^f$ be the bucket in the flat space corresponding to the bucket B^t, with length $\hat{w}^f$. We also have buckets $\{ B_1^f, \ldots, B_m^f \}$ from flattening the plaintext distribution. As before, let bucket B^f have length w^f. We want the range of the two flat distributions to be equal. So we define the matching factor L to be

$$L = \left(\sum_{i=1}^{m} w_i^f \right) / \left(\sum_{i=1}^{k} \hat{w}_i^f \right).$$

We then scale both the target buckets B^t and the flattened target buckets $\hat{B}^f$ by a factor of L. So the length of the cipher bucket B^c corresponding to the target bucket B^t is given by $w_i^c = L w_i^t$ and the length of the scaled flattened target bucket $\bar{B}^f$ is given by $\bar{w}^f = L \hat{w}^f$.

6.2 Mapping Function

We now specify the function M^c for mapping values from the bucket B^c to the flat bucket $\bar{B}^f$. The quadratic coefficient for M^c is determined as $s^c = s^t/L$, and the scale factor z^c is set to z^t, for reasons explained next.

Recall that $s^t := q^t/2r^t$, where $n^t = q^t x + r^t$ is the linear approximation of the density in the bucket B^t. When we expand the domain by a factor of L, q^t/r^t is reduced by a factor of L. Therefore $s^c = s^t/L$.

Now z^c should ensure that $M^c(w^c) = \bar{w}^c$. Setting $z^c = z^t$ provides this property since

$$\begin{aligned} M^c(w^c) &= z^c(s^c(w^c)^2 + w^c) \\ &= z^t((s^t/L)(Lw^t)^2 + Lw^t) \\ &= L\, M^t(w^t) \\ &= L\, \hat{w}^f \\ &= \bar{w}^f \end{aligned}$$

6.3 Mapping Flat Values to Cipher Values

We save the bucket boundaries in the cipher space in the data structure $\mathcal{K}^c$. For every bucket, we also save the quadratic coefficient s^c and the scale factor z^c.

A flat value f from the bucket $\bar{B}_i^f$ can now be mapped into a cipher value c using the equation

$$c = c_{\min} + \sum_{j=1}^{i-1} w_j^c + (M_i^c)^{-1}(f - f_{\min} - \sum_{j=1}^{i-1} \bar{w}_i^f)$$

where

$$(M^c)^{-1}(f) = \frac{-z \pm \sqrt{z^2 + 4szf}}{2zs}.$$

Only one of the two possible values will lie within the cipher bucket, and we round the value returned by $(M^c)^{-1}$.

A cipher value c from the bucket B_i^c is mapped into a flat value f using the equation

$$f = f_{\min} + \sum_{j=1}^{i-1} \bar{w}_j^f + M_i^c(c - c_{\min} - \sum_{j=1}^{i-1} w_j^c).$$

6.4 Space Overhead

The size of the ciphertext depends on the skew in the plaintext and target distributions. Define $g_{\min}^p$ to be the smallest gap between sorted values in the plaintext, and $g_{\max}^p$ as the largest gap. Similarly, let $g_{\min}^t$ and $g_{\max}^t$ be the smallest and largest gaps in the target distribution. Define $G^p = g_{\max}^p / g_{\min}^p$, and $G^t = g_{\max}^t / g_{\min}^t$. Then the additional number of bits needed by the ciphertext in the worst case can be approximated as $\log G_p + \log G_t$. Equivalently, an upper bound for $c_{\max} - c_{\min}$ is given by $G_p \times G_t \times (p_{\max} - p_{\min})$.

To see why this is the case, consider that when flattening, we need to make all the gaps equal. If almost all the gaps in the plaintext are close to $g_{\min}^p$ while only a few are close to $g_{\max}^p$, we will need to increase each of the former gaps to $g_{\max}^p$, resulting in a size increase of $g_{\max}^p / g_{\min}^p$. Similarly, there can be a size increase of $t_{\max}^p / t_{\min}^p$ when transforming the data if most of the target gaps are close to $t_{\max}^p$.

Note that we can explicitly control G_t since we choose the target distribution. While G_p is outside our control, we expect that $G_p \times G_t$ will be substantially less than 2^{32}, i.e., we will need at most an additional 4 bytes for the ciphertext than for the plaintext.

7. EXTENSIONS

7.1 Real Values

An IEEE 754 single precision floating point number is represented in 32 bits. The interpretation of positive floating point values simply as 32-bit integers preserves order. Thus, OPES can be directly used for encrypting positive floating point values.

Negative floating point values, however, yield an inverse order when interpreted as integers. Nevertheless, their order can be maintained by subtracting negative values from the largest negative (-2^{31}). The query rewriting module (Figure 1) makes this adjustment in the incoming query constants and the adjustment is undone before returning the query results.

A similar scheme is used for encrypting 64-bit double precision floating point values.

7.2 Duplicates

An adversary can use duplicates to guess the distribution of a domain, particularly if the distribution is highly skewed. Similarly, if the number of distinct values in a domain is small (e.g., day of the month), it can be used to guess the domain. The solution for both these problems is to use a *homophonic* scheme [22] in which a given plaintext value is mapped to a range of encrypted values.

The basic idea is to modify the flatten stage as follows. First, when computing the scale factors for each bucket using the constraint that the bucket should map to a space proportional to the number of points in the bucket, we include duplicates in the number of points. Thus, regions where duplicates are prevalent will be spread out proportionately, and adjacent plaintext values in such regions will be mapped to flattened values that are relatively far apart.

Suppose that using our current algorithm, a plaintext value p maps into a value f in the flat space, and $p+1$ maps into f'. When encrypting p, we now randomly choose a value from the interval $[f, f')$. Thus the encrypted values of p will be uniformly spread in the interval $[f, f')$. Combined with the intra-bucket uniformity generated by the linear splines and the inter-bucket uniformity from the scale factors, this will result in the flattened distribution being uniform even if the plaintext distribution had a skewed distribution of duplicates. This is the only change to the algorithm – having hidden the duplicates in the flatten stage, no change is necessary in the transform stage.

Selections on data encrypted using this extension can be performed by transforming predicates, e.g., converting equality against a constant into a range predicate. But some other operations such as equijoin cannot be directly performed. This might be acceptable in applications in which numeric attributes are used only in selections. For example, consider a hospital database used for medical research. Patient data will typically be joined on attributes such as patient-id that can be encrypted with conventional encryption. However, numeric attributes such as age and income may strictly be used in range predicates.

8. EVALUATION

In this section, we study empirically the following questions:

1. *Distribution of Encrypted Values*: How indistinguishable is the output of OPES from the target distribution?
2. *Percentile Exposure*: How susceptible to the percentile exposure are the encrypted values generated by OPES?
3. *Incremental Updatability*: Does OPES gracefully handles updates to the database?
4. *Key Size*: How big an encryption key does OPES need?
5. *Time Overhead*: What is the performance impact of integrating OPES in a database system?

8.1 Experimental Setup

The experiments were conducted by implementing OPES over DB2 Version 7. The algorithms were implemented in Java, except for the high precision arithmetic that was implemented in C++ (using 80-bit long doubles). The experiments were run using version 1.4.1 of the Java VM on a Microsoft Windows 2000 workstation with a 1GHz Intel processor and 512 MB of memory.

8.2 Datasets

We used the following datasets in our experiments:

- *Census:* This dataset from the UCI KDD archive (http://kdd.ics.uci.edu/databases/census-income/census-income.html) contains the PUMS census data (about 30,000 records). We used the income field in our experiments.
- *Gaussian:* The data consists of integers picked randomly from a Gaussian distribution with a mean of 0 and a standard deviation of MAXINT/10.
- *Zipf:* The data consists of integers picked randomly from a Zipf distribution with a maximum value of MAXINT, and skew (θ) of 0.5.
- *Uniform:* The data consists of integers picked randomly from a Uniform distribution between -MAXINT and MAXINT.

Our default dataset size for the synthetic datasets was 1 million values. The plaintext values were 32-bit integers. Both flattened and final ciphertext values were 64-bit long.

8.3 Distribution of Encrypted Values

We tested whether it is possible to statistically distinguish between the output of OPES and the target distribution by applying the Kolmogorov-Smirnov test used for this purpose. The Kolmogorov-Smirnov test answers the following question [19]:

> Can we disprove, to a certain required level of significance, the null hypothesis that two data sets are drawn from the same distribution function?

We conservatively try to disprove the null hypothesis at a significance level of 5%, meaning thereby that the distribution of encrypted values generated by OPES differs from the chosen target distribution.[10] In addition to the Census data, we used four sizes for the three synthetic datasets: 10K, 100K, 1M, and 10M values. For each of these input datasets, we experimented with three target distributions: Gaussian, Zipf, and Uniform.

We could not disprove the null hypothesis in any of our experiments. In other words, the distribution of encrypted values produced by OPES was consistent with the target distribution in every case.

We also checked whether the output of Stage 1 (flatten) can be distinguished from the Uniform distribution. Again, in every case, we could not disprove the hypothesis that the distributions were indistinguishable, implying that flattening successfully masked the characteristics of the plaintext distribution.

We should mention here that we also experimented with modeling input distribution using equi-width and equi-depth histograms (with the same number of buckets as in our MDL model). When we applied the Kolmogorov-Smirnov test to check the indistinguishability of the flattened distributions so obtained from the uniform distribution, the hypothesis was rejected in every case except when the input data was itself distributed uniformly. These results reaffirmed the value of using the proposed piece-wise linear function for modeling a density distribution.

[10] Note that this test is much harsher on OPES than using a stronger significance level of 1%. If the null hypothesis is rejected at a significance level of 5% , it will also be rejected at a significance level of 1%.

Input distribution	Target distribution	Change in Percentile
Census	Gaussian	37
Census	Zipf	7
Census	Uniform	38
Gaussian	Zipf	45
Gaussian	Uniform	17
Zipf	Uniform	44

Figure 6: Average change between the original percentile and the percentile in the encrypted distribution.

8.4 Percentile Exposure

Figure 6 shows the average change between the original percentile and the percentile in the encrypted distribution. For example, suppose the plaintext data had a range between 0 and 100, and the ciphertext had a range between 0 and 1000. Then, a plaintext value of 10 that was mapped to a ciphertext value of 240 would have a change of $|10 - 24|$, or 14 percentile points. Thus, the first line in the figure states that each value moved on average by 37 percentile points when going from Census to Gaussian. The reason there is relatively less percentile change when transforming Census to Zipf is that Census itself is largely Zipfian. Hence by judiciously choosing a target distribution that is substantially different from the input data, we can create large change in the percentile values, which shows the robustness of OPES against percentile exposure.

8.5 Incremental Updatability

For an encryption scheme to be useful in a database system, it should be able to handle updates gracefully. We have seen that with OPES a new value can easily be inserted without requiring changes in the encryption of other values.

Recall that we compute the bucket boundaries and the mapping functions when the database is encrypted for the first time, and then do not update them (unless the database administrator decides to re-encrypt the database afresh). We studied next whether the encrypted values remain consistent with the target distribution after updates. For this experiment, we completely replaced all the data values with new values, drawn from the same plaintext distribution. But we did not update $\mathcal{K}^p$ or $\mathcal{K}^c$. We did this experiment with all the four types of datasets, and for each of them we considered Gaussian, Zipf, and Uniform distributions.

Applying the Kolmogorov-Smirnov test again, we found that even with this 100% replacement, the resulting distributions were still statistically indistinguishable from the target distributions.

8.6 Key Size

The size of the encryption key $\mathcal{K}$ depends on the number of buckets needed for partitioning a distribution, the total size being roughly three times the number of buckets. We found that we did not need more than 200 buckets for any of our datasets (including those with 10 million values); for Uniform, the number of buckets needed was less than 10. Thus, the encryption key can be just a few KB in size.

8.7 Time Overhead

We used a single column table in these experiments. The reason was that we did not want to mask the overhead of encryption; if we were to use wider tuples with columns that were not encrypted/decrypted, our overhead would come out to be lower. The column was indexed.

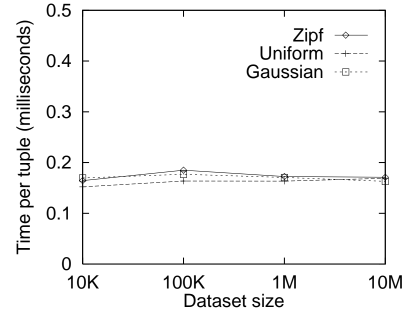


Figure 7: Time per tuple (in milleseconds) required to build the model.

Figure 7 shows the model building cost for the 3 synthetic distributions, for dataset sizes ranging from 10,000 to 10 million records. The time increases linearly with the dataset size, and is similar for all 3 distributions. The total time was less than 4 minutes for 1 million records. It is a one-time cost, which can be reduced by using a sample of the data.

Figure 8 shows the overhead due to encryption of database inserts. The encrypted values followed a Zipf distribution, and the plaintext values followed a Gaussian distribution. The graph shows the percentage overhead of encrypting and inserting 1 to 10,000 values when compared to the cost of inserting an identical number of plaintext values, for databases of different sizes (10K to 10M). Figure 9 shows the absolute times for inserting additional values in a table having 10 million tuples. Figures 8 and 9 clearly show that this overhead is negligible.

Figure 10 shows the impact of decryption on the retrieval of tuples from databases of different sizes. We have also varied the number of tuples retrieved to study the impact of selectivity. The encrypted values followed a Zipf distribution, and the plaintext values followed a Gaussian distribution. The overhead ranges from around 3% slower for equality predicates to around 50% slower when selecting 1 million records.

To understand the reason for higher overhead for less selective queries, note that the decryption time per tuple is constant. Figure 11 shows the time per tuple to retrieve k plaintext tuples from a table with 10 million tuples, as well as the corresponding time per tuple to retrieve and decrypt k encrypted tuples. DB2 has excellent performance on sequential I/O, which reduces per record I/O cost for less selective queries. The percentage overhead due to decryption, therefore, increases. The absolute numbers, however, are very reasonable: around 3 micro-seconds to decrypt one value.

9. SUMMARY

With the dramatic increase in the amount of data being collected and stored in databases, it has become vital to develop effective techniques for protecting sensitive data from misuse. The access control mechanisms conventionally used by database systems become helpless if an intruder can get unauthorized access to the database files. Encryption can be used to provide an extra level of security. Unfortunately, the use of standard encryption techniques

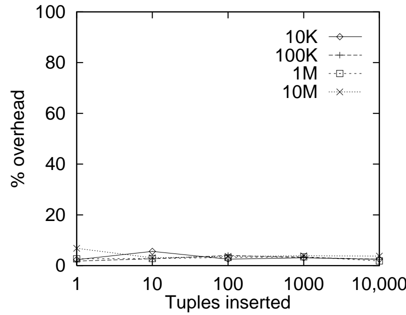


Figure 8: Percentage overhead for tuple insertions due to encryption.

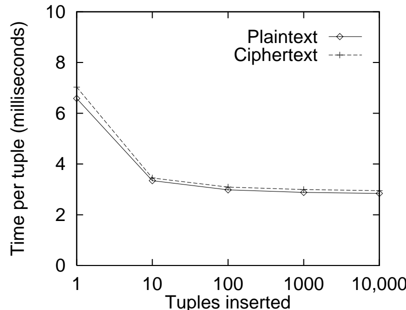


Figure 9: Time per tuple (in milliseconds) required to insert additional tuples.

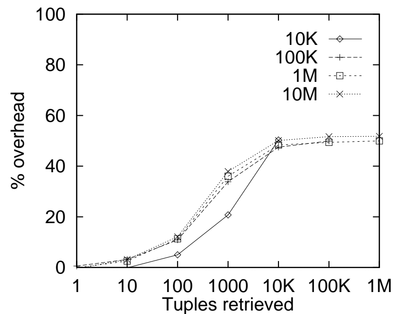


Figure 10: Percentage overhead on retrieval of tuples.

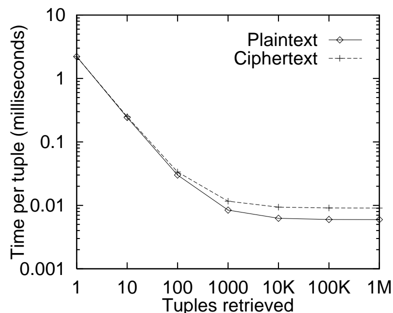


Figure 11: Time per tuple (in milliseconds) required to retrieve tuples.

for this purpose results in degradation in the performance of the database system. The main source of the problem is that the standard techniques do not preserve order and therefore the database indices such as B-tree can no longer be used for answering range queries.

We proposed a new order preserving encryption scheme, OPES, that allows queries with comparison operators to be directly applied to encrypted numeric columns. Query results neither contain any false positive nor miss any answer tuple. New values can be added without triggering changes in the encryption of other values. OPES is designed to operate in environments in which the intruder can get access to the encrypted database, but does not have prior information such as the distribution of values and cannot encrypt or decrypt arbitrary values of his choice. In such environments, OPES is robust against an adversary being able to obtain a tight estimate of an encrypted value. The measurements from an implementation over DB2 shows that the performance overhead of OPES on query processing is small and reasonable for it to be deployed in production environments.

In the future, we plan to study the encryption of non-numeric data such as variable length strings. We also plan to investigate system issues such as key management and the impact of encryption on query plans and query optimization.

Acknowledgments We wish to thank Bala Iyer for helping us understand the subtleties of ideas in [13]. It is also a pleasure to acknowledge Ramesh Agarwal for occasional musings on this topic.

10. REFERENCES

[1] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Hippocratic databases. In *Proc. of the 28th Int'l Conference on Very Large Databases*, Hong Kong, China, August 2002.

[2] N. Ahituv, Y. Lapid, and S. Neumann. Processing encrypted data. *Communications of the ACM*, 30(9):777–780, 1987.

[3] G. Bebek. Anti-tamper database research: Inference control techniques. Technical Report EECS 433 Final Report, Case Western Reserve University, November 2002.

[4] L. Bouganim and P. Pucheral. Chip-secured data access: Confidential data on untrusted servers. In *28th Int'l Conference on Very Large Databases*, pages 131–142, Hong Kong, China, August 2002.

[5] Z. J. Czech, G. Havas, and B. S. Majewski. An optimal algorithm for generating minimal perfect hash functions. *Information Processing Letters*, 43(5):257–264, 1992.

[6] E. Damiani, S. D. C. di Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati. Balancing confidentiality and efficiency in untrusted relational dbmss. In *Proc. of the 10th ACM Conf. on Computer and Communications Security (CCS)*, October 2003.

[7] D. Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.

[8] J. Domingo-Ferrer and J. Herrera-Joancomarti. A privacy homomorphism allowing field operations on encrypted data. *I Jornades de Matematica Discreta i Algorismica, Universitat Politecnica de Catalunya*, March 1998.

[9] J. Domingo i Ferror. A new privacy homomorphism and applications. *Information Processing Letters*, 60(5):277–282,

1996.

[10] J. Feigenbaum, M. Y. Liberman, and R. N. Wright. Cryptographic protection of databases and software. In *Proc. of the DIMACS Workshop on Distributed Computing and Cryptography*, 1990.

[11] E. A. Fox, Q. F. Chen, A. M. Daoud, and L. S. Heath. Order-preserving minimal perfect hash functions and information retrieval. *ACM Transactions on Information Systems (TOIS)*, 9:281–308, 1991.

[12] S. C. Gultekin Ozsoyoglu, David Singer. Anti-tamper databases: Querying encrypted databases. In *Proc. of the 17th Annual IFIP WG 11.3 Working Conference on Database and Applications Security*, Estes Park, Colorado, August 2003.

[13] H. Hacigümüş, B. R. Iyer, C. Li, and S. Mehrotra. Executing SQL over encrypted data in the database-service-provider model. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, Madison, Wisconsin, June 2002.

[14] T. Hamilton. Error sends bank files to eBay. The Toronto Star, September 15, 2003.

[15] Y. E. Ioannidis. The history of histograms (abridged). In *Proc. of 29th Int'l Conf. on Very Large Data Bases (VLDB)*, Berlin, Germany, September 2003.

[16] A. Konig and G. Weikum. Combining histograms and parametric curve fitting for feedback-driven query result-size estimation. In *Proc. of the 25th Int'l Conference on Very Large Databases*, Edinburgh, Scotland, 1999.

[17] M. Mehta, R. Agrawal, and J. Rissanen. SLIQ: A fast scalable classifier for data mining. In *Proc. of the Fifth Int'l Conference on Extending Database Technology (EDBT)*, Avignon, France, March 1996.

[18] Oracle Corporation. *Database Encryption in Oracle 8i*, August 2000.

[19] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, second edition, 1992.

[20] J. Rissanen. *Stochastic Complexity in Statistical Inquiry*. World Scientific Publ. Co., 1989.

[21] R. L. Rivest, L. Adelman, and M. L. Dertouzos. On data banks and privacy homomorphisms. In *Foundations of Secure Computation*, pages 169–178, 1978.

[22] B. Schneier. *Applied Cryptography*. John Wiley, second edition, 1996.

[23] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *IEEE Symp. on Security and Privacy*, Oakland, California, 2000.

[24] D. R. Stinson. *Cryptography: Theory and Practice*. CRC Press, 2nd edition, 2002.

[25] R. Vingralek. Gnatdb: A small-footprint, secure database system. In *28th Int'l Conference on Very Large Databases*, pages 884–893, Hong Kong, China, August 2002.