

A Partially Order-Preserving Index Scheme

Jonathan Dautrich, jjldj@jjldj.com

UC Riverside, Professor Ravishankar

July 3, 2009

1 Introduction

It has become advantageous in certain scenarios for a database owner to outsource hosting and management of his database to a third party. This approach is known as *database as a service* or *DAS*. However, in the case of highly sensitive data, it may be necessary to protect it even from the third party database administrator. As a result, several methods have been proposed that allow for encryption of the relational data in such a way as to permit some limited set of queries to be executed directly on the encrypted data.

Most encryption methods, like hashing techniques, tend to diffuse the data, such that two nearby plaintext values may have completely dissimilar ciphertext values. As a result, performing range queries on the encrypted data is particularly challenging. We here propose a new partially order-preserving indexing scheme that permits efficient range queries on encrypted data. This work is largely derived from prior work done by Agrawal et al. on fully order preserving encryption [1] and work by Hore et al. on a bucketization approach for range queries [2].

Our method resembles the bucketization technique used in [2], but enforces an ordering of the ciphertext buckets that correlates with the ordering of the plaintext buckets. A plaintext value may still end up in one of several buckets (diffusion), but instead of selecting the target buckets randomly, they will be selected based on their proximity to the ciphertext bucket corresponding to the plaintext value. (See section 2) This imposes a partial ordering on the ciphertext values, allowing large range queries to yield fewer false positives than they do in [2], while at the same time preserving some of the privacy lost to the total ordering seen in [1].

2 Details of Proposed Method

In the DAS model, there are commonly three actors: a data owner, a data user/client, and the server/adversary. We here make no distinction between the client and the database owner, though it would be fairly straightforward to separate the two by introducing sufficient communication between them. Here, we treat the client as trusted and responsible for encrypting and partitioning (bucketizing) the original data before sending it to the server.

2.1 Encryption and Bucketization

2.1.1 Defining Buckets

Our scheme is similar to [2] in that we store on the server the entire plaintext record encrypted as one etuple. For each attribute to be indexed, we store a separately encrypted version of the attribute value alongside the etuple. For simplicity, we consider attributes with a discrete, finite domain, such as a fixed-length integer or text field. The resulting ciphertext will fall into its own discrete, finite domain of size at least that of the corresponding plaintext domain.

The plaintext domain will be divided into m buckets or *partitions*, each enclosing the same number of elements (equi-depth, n/m , where n is the total number of records). The ciphertext domain will also be divided into m buckets, but such that each encloses the same range of potential ciphertext values (equi-width). For a plaintext bucket P_i , the corresponding ciphertext bucket C_i is referred to as its *target* bucket.

The plaintext values contained in each bucket P_i will be mapped into a certain range over the ciphertext domain. This range will always encompass whole ciphertext buckets, so this can be thought of as diffusing the plaintext bucket P_i into some set of ciphertext buckets. We now define an additional security parameter k which controls the number of buckets into which a given plaintext bucket will be diffused. If $k = 0$, all values in P_i will be mapped into the corresponding target bucket C_i . For non-zero k , the k buckets extending in each direction from the target bucket will also be included (C_{i-k} through C_{i+k}). As an example, if $k = 2$, then bucket P_5 will be diffused into buckets C_{5-2} through C_{5+2} , or C_3 through C_7 , for a total of $2k + 1 = 5$ buckets.

2.1.2 Diffusion

Diffusion should occur such that plaintext values are pseudorandomly and uniformly mapped to ciphertext values over the entire ciphertext bucket range (C_{i-k}, C_{i+k}). It is advantageous to make this mapping deterministic, as we shall see later, but it is not necessary that it be reversible. As such, we will use a keyed one-way hash function to map the plaintext values into ciphertext values. Input to the hash function will be the plaintext value v , the range of ciphertext values encompassed by (C_{i-k}, C_{i+k}), and a secret key s . Output will be a ciphertext value falling somewhere in the bucket range (C_{i-k}, C_{i+k}). These hashed/encrypted values serve as the encrypted index for the attribute. (Note that we do not here consider a mechanism for handling duplicates without giving away extra statistical information.)

It can be seen that these encrypted attribute values impose a total ordering on the etuples that is not wholly consistent with the plaintext attribute's ordering. It is however *partially* consistent, in that each value can be no more than a fixed number of buckets away from its target bucket. This ultimately allows us maintain a performance guarantee over large range queries. (See 2.2 and 4.1)

The client will need to store the boundaries of the buckets along the plaintext domain and the secret hash key s , as well as the main encryption key used for encrypting the tuples. Since the divisions along the ciphertext domain are of fixed width, he will only need to remember the number of buckets to reproduce this information. The bucket boundaries and hash key will need to be stored separately for each indexed attribute, and will be used to transform plaintext queries into ciphertext queries usable by the server.

It should be noted that the concept of buckets will be transparent to the server. The server will store the ciphertext values in the encrypted attribute's field, and will return results to modified

range queries, as we will see below. The server is not explicitly aware of the ciphertext bucket boundaries, though he could easily determine them, as can be seen from the way queries are executed below.

2.1.3 Overflow

The above method works fine for the plaintext buckets toward the center of the list, but falls apart for those toward the beginning or end. That is, for the first plaintext bucket P_1 , there is no C_{1-k} bucket for it to diffuse into. We propose simply using a wrap-around technique, treating the ciphertext bucket list as a circle instead of a line. This is feasible since the ciphertext domain is of a fixed size. This will add another quick step to processing on the client side, and will result in the server occasionally needing to issue two separate queries, one covering the ciphertext values at the beginning of the list and one for those at the end. Given a reasonable number of buckets, this overhead should be minimal and occur infrequently.

2.2 Queries

2.2.1 Transforming the Range Query

The system above allows for the effective execution of range queries with a limited number of false positives. When a query Q for a plaintext range $[A, B]$ on a given attribute is issued, the client first determines the plaintext buckets P_i, P_j containing A, B respectively. It then identifies the corresponding target ciphertext buckets C_i, C_j . Due to the way data are diffused, all values encompassed by the plaintext value must fall within the range of the ciphertext buckets C_{i-k}, C_{j+k} . The upper and lower boundaries of this range A', B' are then determined by the client and used as the modified range query values sent to the server. Note that these are ciphertext boundary values and did not result from applying the hashing function directly to A or B . As such, they only allow the server to determine which plaintext bucket each of the query boundaries is in. They permit the server to know neither the exact value of the plaintext endpoints, nor which of the ciphertext values correspond to them.

2.2.2 Range Query Results

The modified range query is processed by the server using an index built on the partially order-preserving attribute, and the results are returned. There will be some false positives included in the result set that must be filtered out by the client after decryption. It can be seen, however, that the number of false positives is nearly independent of the query size. We show in 4.1 that the number of false positives is limited to $(n/m)(2k + 2)$, even for large queries.

This independence of the query can be seen intuitively when we consider that a given ciphertext bucket consists entirely of elements from the $2k + 1$ plaintext buckets *closest* to it. Once all these buckets are encompassed by the plaintext query, the ciphertext bucket contains no false positives. Expanding the size of the plaintext query simply includes additional ciphertext buckets with no false positives.

Seen from another view, for a very small plaintext range query, most of the results will be false positives. However, every time the plaintext query is expanded to include another full bucket, only one additional ciphertext bucket needs to be included as well, allowing the increase in results to be matched by the increase in true positives, and thus keeping the false positive count constant.

2.2.3 Other Queries

The deterministic nature of the encryption / hash function we use allows other types of queries, such as simple equality queries, to be conducted on the attribute as well. This may still result in false positives in the event of hash collisions, though this is not likely to occur frequently. When it does occur, it can be handled by the client in post-query filtering as is done for range queries. Such a query would require a hashing operation by the client, but should have, on the server-side, the same efficiency as an equality query on a standard database. Join queries would be more challenging, and would require that the separate tables use the same bucket boundaries and secret hash key for encrypting the attribute. This may cause additional privacy problems and is not further discussed here.

2.3 Performance and Privacy Claims

This method makes two primary claims, one regarding privacy, and the other regarding performance, both of which are discussed more rigorously in section 4. The performance claim states that no more than $(n/m)(2k + 2)$ false positive results are returned with any range query, regardless of size. This claim is discussed briefly above. The privacy claim states that no refinement of ciphertext results from a given plaintext query can be performed using query intersections. Therefore, every range query spanning b plaintext buckets can be associated with no fewer than $2k + b$ ciphertext buckets. This is shown in 4.2.

3 Comparative Evaluation

We here compare different aspects of three different schemes useful for performing range queries on encrypted data. They are the Order Preserving Encryption Scheme (OPES) [1], the Privacy Preserving Index / bucketization approach discussed in [2] (PPI), and our proposed partially order-preserving scheme (POP).

3.1 False Positives & Communication Load

The OPE scheme boasts no false positives, as it perfectly preserves the original ordering, and thus encrypted range query endpoints can specify the exact values desired. PPI on the other hand incurs a false positive count that is proportional to the number of true positives in the worst case, though not quite linear in the average case. POP sports a constant number of false positives, independent of the query size. This results in a very high percentage of false positives for very small queries, but a percentage much lower than that of PPI for queries encompassing more than a few plaintext buckets.

Communication overhead is determined by the server-to-client overhead (number of false positives) and the client-to-server overhead (query size). Both OPE and POP are able to convert the plaintext queries to a pair of encrypted values that can be used by the server to perform a simple range query, so overhead is minimal. PPI on the other hand requires the client to convert the query to a disjunction of bucket IDs that he is requesting from the server, which can make for a rather large message, especially if the bucket sizes are small.

3.2 Computation, Storage, & Client Burden

OPE exhibits a high up-front cost for preparing the data, sending each attribute through three separate stages of manipulation involving scaling, encryption, etc. Multiple data values must be saved for each partition/bucket used by the scheme. Incoming plaintext queries are translated (and values encrypted) using a retained or reconstructed key mechanism based off the stored data.

PPI involves a fairly small computational overhead after buckets are selected, needing only to randomly select buckets and locations for diffusion. It too must store bucket boundaries but must also store the list of buckets into which each bucket is diffused. PPI has the additional burden of translating the plaintext range query into an itemized bucket disjunction to be passed on to the server. In PPI, false positive results are returned to the client, so a post-query filtering step is needed.

POP requires a moderate amount of computation, passing each value through a keyed hash function. Storage requirements include the hash function key and the bucket boundaries, giving POP the least significant storage needs of the three methods. Application of the hash function to query values is only needed for equality queries, as range queries use known/easily computable ciphertext bucket boundaries, making client processing load for query preparation fairly small. POP, like PPI, returns false positives to the client, so must also implement the potentially expensive filtering step.

3.3 Privacy, Estimation Exposure, & Ordering Information

Out of the three schemes, OPE reveals the most information. Notably, OPE preserves the order of all elements without giving away the original distribution of the values. However, as the authors admitted, *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*. PPI on the other hand assumes that the adversary knows the original distribution of plaintext values, but relies on the controlled diffusion into multiple buckets, which are returned all-or-nothing, to confound attempts to associate individual plaintext values with individual ciphertext values, foiling tight estimation attempts. POP also prevents tight estimation attacks by making use of the same diffusion principle. (See 4.2)

While POP does not reveal the full ordering of the values as does OPE, POP does reveal a partial ordering in that it explicitly orders the buckets. Even with ordered buckets, POP preserves privacy by diffusing plaintext values within the target ciphertext bucket and $2k$ surrounding buckets. With a large enough range query, POP will result in ciphertext values that the adversary can know for certain fall within the requested range. Nevertheless, as we see in 4.2, he cannot utilize this information to perform refining query intersections producing plaintext ranges with ciphertext ranges tighter than the ones obtained by executing a normal query on the plaintext range.

4 Performance and Privacy Proofs

4.1 Performance Guarantee

We here show that for any query, the upper bound on the number of false positives is constant. That is, it is independent of the query size, upper-bounded at $(n/m)(2k + 2)$.

Proof:

Let m be the number of buckets, n be the total number of elements, and d be the number of elements per bucket $d = n/m$. We denote the total number of values returned as R . The false positive count will be R_F , while the true positive count is R_T , such that $R = R_F + R_T$. We recall that k is the number of buckets to the left and right of the target bucket into which values will be diffused.

We first consider the case in which the plaintext query spans one bucket. Here, we have:

$$\begin{aligned} R &= d(2k + 1) \\ 1 &\leq R_T \leq d \end{aligned}$$

(Zero-element queries are not considered.) This gives us:

$$\begin{aligned} -1 &\geq -R_T \geq -d \\ -R_T &\leq -1 \\ R - R_T &\leq R - 1 \\ R_F &\leq R - 1 \\ R_F &\leq d(2k + 1) - 1 \\ R_F &\leq d(2k + 2) \end{aligned}$$

which satisfies our requirement.

Next we consider the case in which the query spans $q \geq 2$ plaintext buckets:

$$\begin{aligned} R &= d(2k + q) \\ d(q - 2) + 2 &\leq R_T \leq dq \\ -dq &\leq -R_T \leq -d(q - 2) - 2 \\ R - R_T &\leq R - d(q - 2) - 2 \\ R_F &\leq R - d(q - 2) - 2 \\ R_F &\leq d(2k + q) - d(q - 2) - 2 = 2dk + 2d - 2 \\ R_F &\leq d(2k + 2) \end{aligned}$$

This also satisfies our requirements, proving that the number of false positives for any query is upper-bounded by $d(2k+2)$.

4.2 Privacy Guarantee

We claim that for any two queries Q_1, Q_2 with corresponding ciphertext queries Q'_1, Q'_2 , their intersecting query/result $Q_3 = Q_1 \cap Q_2, \widehat{Q}_3 = Q'_1 \cap Q'_2$, will always satisfy $Q'_3 \subseteq \widehat{Q}_3$, where Q'_3 is the ciphertext query resulting from executing plaintext query Q_3 . That is, \widehat{Q}_3 will always completely enclose Q'_3 . This means that no intersection can be performed to give any more information about the ciphertext values associated with a plaintext range than would running a query using that range. As a result, no refinement through successive intersection is possible. As a corollary, any range

query encompassing only a single plaintext value will be guaranteed to return at least $d(2k + 1)$ ciphertext values.

Proof:

We prove the main claim as follows. Let Q_{iA} be the index of the left-most element encompassed by query Q_i , and Q_{iB} the index of the right-most. We assume, WLOG, that Q_1 is left of Q_2 . Then we have that:

$$\begin{aligned} Q_3 &= Q_1 \cap Q_2 = (Q_{2A}, Q_{1B}) \\ \widehat{Q}_3 &= Q'_1 \cap Q'_2 = (Q'_{2A}, Q'_{1B}) \\ Q'_3 &= (Q'_{3A}, Q'_{3B}) \end{aligned}$$

We note that since $Q_{3A} = Q_{2A}$, we can conclude that $Q'_{3A} = Q'_{2A}$, since we're using the same bucket boundary rules we would have used to construct Q'_2 . Similarly, $Q'_{3B} = Q'_{1B}$. Thus, we have:

$$\begin{aligned} Q'_{3A} &= Q'_{2A} = \widehat{Q}_{3A} \\ Q'_{3B} &= Q'_{1B} = \widehat{Q}_{3B} \end{aligned}$$

This confirms that Q'_3 is completely enclosed within \widehat{Q}_3 (identical in fact), confirming our claim.

We note that while it is not possible to obtain result refinement through intersection, it is possible to obtain results for queries not yet issued, using a simple intersection as described above. This is undesirable, but inherent to our approach and does not seem a major concern for our work.

5 Conclusion

We have proposed a new partially order-preserving index scheme for performing range queries on encrypted data. The work is a mix of new ideas and concepts discussed in prior work by Hore [2] and Agrawal [1], and attempts to improve upon these existing works by combining some of their strengths. The scheme exhibits a limited number of false positives, independent of the query size, as well as a privacy guarantee limiting an adversary's ability to match up individual or small plaintext blocks with their corresponding ciphertext blocks. The method provides an effective means for executing queries over encrypted data, while serving as a starting place for further development of a few of the ideas presented herein.

References

- [1] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Order preserving encryption for numeric data. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 563–574, New York, NY, USA, 2004. ACM.
- [2] Bijit Hore, Sharad Mehrotra, and Gene Tsudik. A privacy-preserving index for range queries. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 720–731. VLDB Endowment, 2004.