On Querying Encrypted Databases

Moheeb Alwarsh and Ray Kresman

Department of Computer Science
Bowling Green State University
Bowling Green, OH 43403
{moheeba, kresman}@bgsu.edu

Abstract

This paper presents a new range query mechanism to query encrypted databases that reside at third-party, untrusted, servers. This paper is a continuation of work done by others [1]; our scheme seeks to improve the precision of querying encrypted data sets, increase the utilization of server side processing and reduce the computation and memory utilization on the client side. We compare our algorithm with previous work, and quantify the performance improvements using numerical results.

Key words: Trust, encrypted database, bucketization, binary search.

1 Introduction

Data is a vital asset for business and educational enterprises. In house storage and maintenance of such assets have an impact on the bottom line of enterprises [7]. Database as service (DAS) is marketed as an outsourcing solution that can reduce the total cost of ownership of these assets. DAS allows for the full utilization of databases with professional support and maintenance by these service providers [3]. This brings up a related issue: storage of sensitive data at the providers' site may jeopardize the security of the stored information. One solution, then, is to store the data in encrypted form, and have clients download the relevant encrypted tables from the remote database, decrypt and do query processing [2]. This process expends bandwidth and client resources for decryption and intermediate storage.

Researchers have proposed various solutions to split the work between the client and server sites, while addressing the privacy concerns [4], [8-9]. This paper is a continuation of work done by other researchers. We propose a new algorithm, Binary Query Bucketization (BQB), that seeks to improve the precision of querying encrypted data sets, increase the utilization of server side processing and reduce the computation and memory utilization on the client side. We evaluate the performance of our approach with other researchers.

The paper is organized as follows. Section 2 reviews work done by others. Section 3 introduces our algorithm, BQB. Section 4 addresses performance comparison with other algorithms. Concluding remarks appear in Section 5.

2 Related Work

One approach to querying encrypted data from untrusted sites is Bucketization [10]. It divides a column in a table into several buckets where each bucket has an ID and a range that defines the minimum and maximum values in this bucket. The client holds this indexing information, which identifies the *range* of each of the buckets at DAS, while the actual data, in encrypted form, is stored at DAS (also known as server).

We use an example to illustrate Bucketization. Suppose the database is about student GPA as shown in Table 1, and we use two buckets. Suppose range of Bucket 1 covers GPA <= 1, while bucket 2 covers the rest. For a query on students with GPA 2 or better, the client consults the local index and asks the server to send just bucket 2; it then decrypts the data and responds to the user query. However, a query for GPA < 2 will mean downloading, and decrypting all of the two buckets, buckets 1 and 2. In either case, the client has to filter out *false positives* – unwanted data that is outside of the query range. Thus, the technique may lack precision when the queried

data is not on the buckets boundaries. False positives help measure the precision of queries when dealing with encrypted databases. Other researchers have focused on ways to reduce the number of false positives while maintaining the privacy of data.

Query Optimal Bucketization (QOB) [1], is predicated on the assumption that queries are uniformly distributed. By associating a cost to each bucket (see below), QOB spreads the data over the, given M, buckets in an attempt to reduce the number of false positives while minimizing the total cost of buckets. The algorithm has two parts.

Consider again the GPA information in Table 1. The first part of QOB builds a frequency table to house each distinct GPA value and its frequency; for example, an entry in the frequency table is (1.5, 3) since there are 3 students with GPA 1.5.

QOB finds optimal bucket boundaries in a set of values V, V = $\{v_1,...,v_n\}$, using at most M buckets. Boundaries of QOB are identified by finding the minimum Bucket Cost (BC) for each bucket, with each bucket holding holds values in the range $[v_i,v_j]$. BC is defined as:

$$BC(i,j) = (v_j - v_i + 1) * \sum f_t$$

where f_t is the frequency of each value in the range. As shown in Table 2, the cost for a bucket that stores all GPAs between [0.7 to 1.2] is BC (5 – 1 + 1) * 6 = 30.

The second part builds the optimal bucket partitions. This is achieved by attempting all possible data distributions across the M buckets and computing cost for each distribution, and finally choosing the distribution that yields the least cost. Suppose M = 2. Then, try out all possible distributions between the two buckets: [(0,7-0.8), (0.9-3)], [(0.7-0.9), (1.0-3)], and so on. The cost for the first distribution is easily be shown to be BC (2 – 1 +1) * 2 + BC (7 - 3 + 1) * 12 = 4 + 60 = 64. It can also be readily shown that the least cost distribution is given by Table 3 (see [1] for complete details).

Deviation Bucketization (DB) [5] extends QOB by further reducing false positives, but at the expense of additional buckets: while QOB uses M buckets, DB needs at most M^2 buckets. Intuitively, the number of false positives will decline as it is inversely proportional to the granularity of the bucket values.

DB is divided into three steps. In Step 1, the QOB bucket output is generated and named as first level buckets, and mean value for each bucket is also computed. Step 2 computes a deviation array that captures the deviation of each data value from the mean found in Step 1. In Step 3, QOB is applied again, but to the deviation array of Step 2 to yield new second level buckets. Unlike Step 1, Step 3 does not use the frequency information for each data value. An example of the second level table ranges for the GPA data is shown in Table 4; as compared to QOB, DB needs three additional, second-level, buckets.

The second level buckets repartition each bucket (of DB) into at most M additional buckets based on how far a value in the original bucket deviates from the mean. Said another way, while QOB may hold low and high frequency data in the same bucket, DB tries to split them – recognizing that high frequency ones are more likely to be queried -- to permit a reduction in the number of false positives. Similar to QOB, DB keeps index information at the client. Additional details of DB may be found in [5].

3 Binary Query Bucketization

The advantage of DB is that the number of false positives decline as each bucket becomes more granular. But the down side is more client storage needs, and granular data has more privacy issues than less granular data. In this section, we propose an improved algorithm, Binary Query Bucketization (BQB).

Like QOB and DB, BQB stores DAS data in encrypted form along with the bucket ID. For example, EncryptionOfStudentGPAInformation ("0.7, 2121212, John") yields ("sldfjkl23k4jl234jklkj23l4kj23l4kj23lk4j"), as shown by the DAS data in Table 5.

While QOB and DB need only the first two attributes of this table, BQB [11] maintains an extra plaintext attribute, autoID for each tuple, as shown by the third column of Table 5. The AutoID field is not related to the bucket ID; it can be generated either dynamically by creating a view that has an auto number combined with the encrypted table, or by adding the auto number with the encrypted table when uploading the encrypted data. AutoID is just a monotonically increasing number assigned with

each tuple and has *no* relation to the actual data contained in that tuple. The client side index is the same as that of QOB. We also note that BQB employs M buckets, and adopts the same strategy as QOB in determining the bucket distribution. Now, we are ready to discuss the details of BQB.

Our algorithm, shown in Figure 1, attempts to reduce the number of false positives by doing a binary search on the encrypted data housed at DAS. Suppose the user's query range is < V, i.e. the query retrieves GPA values < V. Using the client index, the algorithm starts by identifying the buckets that are needed to answer the query. It then constructs a query for the server to fetch two quantities: the min and max AutoID among all of these buckets. For example, if the buckets of interest are the first four buckets, the min will be the AutoID of the first tuple of the first bucket while max will be the AutoID of the last tuple of the fourth bucket. While QOB and DB would have retrieved and decrypted all of the tuples in all of these buckets, BQB retrieves much less, as shown below; for now, it retrieves just these two AutoIDs. For convenience, let us call these two AutoIDs as x and y.

This sets in motion a binary search algorithm to focus on the region (x, y). The client then asks the server for the encrypted tuple at the midpoint of x and y, i.e. tuple z, z = (x + y) / 2. The client decrypts tuple z, and extracts the data value (GPA), say z_v, for this tuple. If z_v > V, then, our new region of interest is (x, z) else our region of interest is (z, y). The binary search continues, in a similar fashion, in this new region. Eventually, the binary search terminates when the region is null. At that point, we would have obtained the AutoID, say q_{id}, corresponding to the original GPA query value V.

Armed with q_{id}, we make one final query to the server to retrieve all tuples with AutoIDs between 1 and q_{id}. With this, the query is complete, i.e. the query to retrieve GPA values < V corresponds to the decryption of these tuples whose AutoIDs lie between 1 and q_{id}.

The number of steps needed to complete the binary search is easily shown to be bounded above by Log r, where r = y - x.

4 Performance Results

The dataset we used in this experiment is similar to the one in [1] and [5]; it contains more than 5 x 10^5 tuples (576,097 to be exact, for a total of 35 Mb of non-encrypted data) taken from "Forest CoverType" Archive database [6] with the actual data values ranging from 1 to 360. We ran the three algorithms with 50 different values for M, the number of buckets, varying M from 4 to 54. Each run was repeated for 1000 queries.

We stored the DAS data in encrypted form, but for brevity we discuss the results for non-encrypted DAS storage. At the end of the section, we remark on the performance for encrypted DAS storage.

We measured three quantities to characterize the performance of the three algorithms:

- number of false positives;
- size of superset – total number of tuples returned to the client. It includes data within the query range, and false positives or data outside of the query range; and
- turnaround time – elapsed time, from start to finish.

The number of false hits for the three algorithms is depicted in Figure 2. Note that for M = 4, this number is about sixty-eight thousand for QOB, and around sixteen thousand for DB. With BQB, the number of false hits is no more than 20. The large number of hits for the former two algorithms may be attributed to their inability to retrieve single tuples.

Figure 3 represents the size of the superset, the volume of the number of tuples retrieved for each of the three schemes. As seen in Figure 3, the size of superset for QOB, DB and BQB are 358,403, 306,430, and 289,710 respectively, for M = 4. Note that the superset counts the total number of tuples retrieved from DAS, those that don't match the user's query – false positives - and those that do. Relating Figure 3 to Figure 2 can provide insight into error rate – the percent of records that don't match user's query; for example, error rates for QOB and DB are 20% and 5% (70,000 / 358,000 = 20%; 16738 / 306430 = 5%), while for BQB the error rate is close to 0 (18 / 289710), all for M = 4.

As noted in [1], DB performs better than QOB since it employs more buckets with finer partitioning. Consider the case where data values are whole numbers in the range 1 to 360. The maximum number of buckets that can be

produced in this range is 360/2 or 180 buckets. Thus, when 13 buckets are used by QOB, DB would employ up to 169 (= 13^2) buckets, and DB would reach the maximum number of 180 buckets when QOB's bucket use goes up by one more. However, privacy may be a concern for DB since the data range is much smaller, and there is not much that separates the buckets or the values inside of these buckets. For example the bucket partitioning looks like [1~2], [3~4], [5~6], and so on. Of course, BQB does not exhibit this problem.

Figure 4 compares the mean turnaround time for various bucket sizes. For smaller bucket sizes -- 4 through 12 – we deployed a sequential version of the three algorithms, and a parallel -- or multiple process -- version for larger bucket sizes. As shown in Figure 4, BQB has a slower turnaround than either of DB or BQB; for M = 4, the QOB, DB and BQB turnaround times are 1, 1 and 3 seconds, and for M=20, these values are 43, 76 and 93 seconds**.** Slower turnaround time of BQB can be attributed to the network delay experienced during the binary search process when BQB retrieves single tuples.

Figure 5 shows the size of the returned false hits in mega bits; while this information can also be extrapolated from Figure 2, it nevertheless provides an easy handle on the bandwidth consumption for the three algorithms.

Encrypted DAS

For brevity, we did not show the performance results when DAS data is stored in encrypted form. Decryption is quite time consuming/expensive, and is an added penalty for *all* of the three schemes. Since the other two schemes have to do lot more decryption than BQB (see Figure 2, for the number of false hits), they take many hours to complete the decryption process for large datasets, such as the one noted in the beginning of this Section. So, we use a smaller, but *encrypted* data set to illustrate this point, see Figure 6 for an example. As one would expect, BQB terminates much earlier than the other two algorithms.

Recall that the number of false positives with BQB is bounded above by Log r, where r equals the range of the user's query. Thus, the overall performance of BQB with encrypted storage at DAS should be significantly better than that of the other two algorithms. In fact, when decryption is factored in, the small penalty due to multiple client requests - as experienced by BQB's binary search - is more than offset by the gain in significantly fewer decryptions that are needed for the proposed algorithm.

5 Concluding Remarks

In this paper we proposed a new algorithm for querying encrypted data that is stored at untrusted servers. Our approach is a variant of a scheme used by others. The novelty of our scheme is that it employs a, new, binary search step to precisely determine the range of the actual data that is relevant to the user's query. During the binary search process we decrypt one tuple at a time instead of doing bulk decryption of, often unwanted data – as is common with other approaches – and discarding it later.

Following the binary search, BQB retrieves the relevant data in bulk and decrypts it. Since all of this data correspond to the user's query, no false positives are generated following the binary search process. This results in a significant reduction both in data transmission, and the number of decryptions at the client. However, the number of client-server interactions, while bounded above by Log r, is a bit higher with the proposed scheme as compared to the other two schemes.

GPA	SSN	Name
0.7	2121212	john
0.8	6545545	mike
0.9	5121123	rob
1.0	5482123	tom
1.0	2384865	steve
1.2	5315422	ali
1.5	6689555	ahmed
1.5	5165888	adam
1.5	9954521	cres
3.0	5458862	cris
3.0	7148787	rob
3.0	2342336	mike
3.0	3334445	rich
3.0	2222234	amanda

Table 1: GPA Data

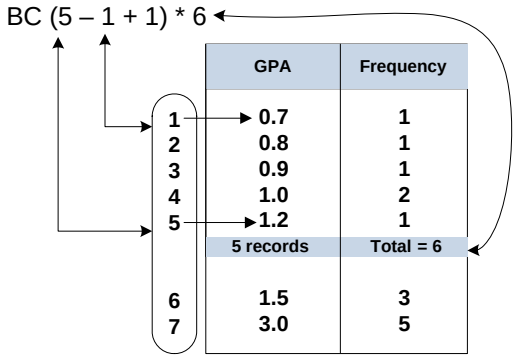

Table 2: Computation of BC [1]

GPA	ID
[0.7 ~ 1.0]	Bucket_1
[1.2 ~ 3.0]	Bucket_2

Table 3: QOB Bucketization (M = 2) – Index Information

Algorithm BQB (V)

Input: Required value to search for

Output: Query result with zero false positive.

Select all buckets from QOB table > or < V

Construct select statement to retrieve (Max, Min) AutoID and their encrypted records

if (returned set != user query)

Mid = Min + ((Max-Min)/2)

Query="Select Etuple From ETable Where AutoID= Mid"

Decrypt = Decrypt (Query) → extract V

While (Max > Min)

If (Decrypt > V)

Max = Mid - 1

else

Min = Mid + 1

Mid = Min + ((Max-Min)/2)

Query="Select Etuple From Etable Where AutoID= Mid"

Decrypt = Decrypt (Query) → extract V

End While

end if

Return: Select eTuple From Etable Where AutoID > or < Mid

Figure 1: Binary Query Bucketization – Binary Search

Partition	BID
[0.7 ~ 0.8]	Bucket_1_1
[0.9 ~ 1.0]	Bucket_1_2
[1.2 ~ 3.0]	Bucket_2_1

Table 4: Second Level Bucketization

Bucket	eTuple	AutoID
Bucket_1	Sdd342kjk23ksdfsdfsdfsd234sdf453453ed	1
Bucket_1	234lj2kdjsfi33345345ergfdfgdgxfgdfykjkklj5	2
Bucket_1	Pqasdososoiwfgksdjkhlkerkljdkfjlieieywhsdf	3
Bucket_1	2#Kskdjkskqwserjk34j5k0s9fksdjf09345jkdf	4
Bucket_2	si4@sdkjweqq345jfdg09345kjdfg0;lskfgfkfg	5
Bucket_2	Sidfjkeierijek334e34jk509345098dfjkfg;pwj	6
……...	…….	…

Table 5: DAS Encrypted Table

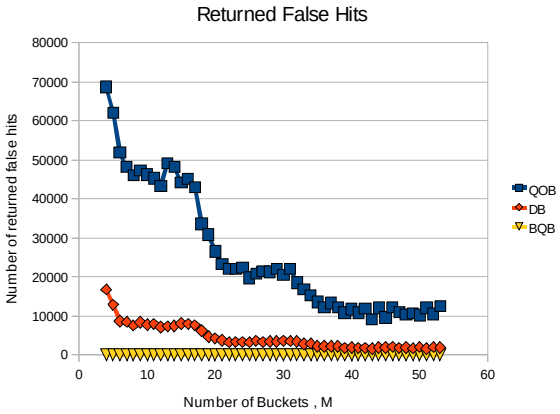

Figure 2: False Positives

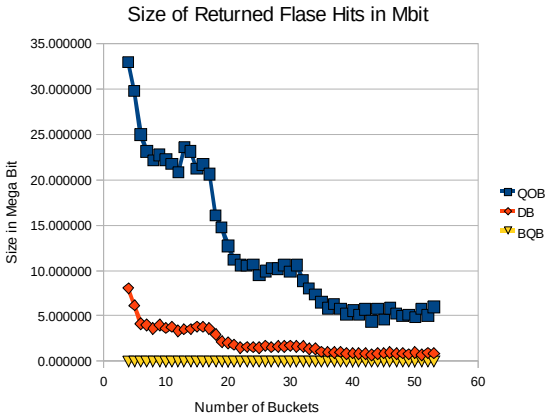

Figure 5: Bandwidth Usage

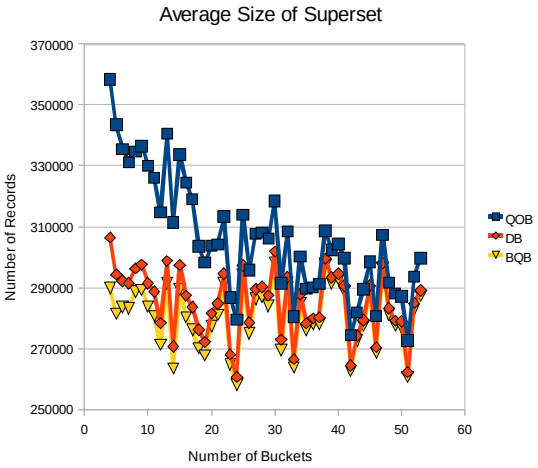

Figure 3: Number of Tuples Processed by the Client

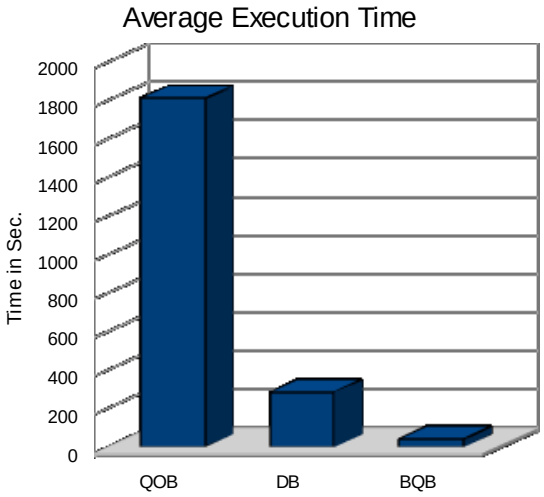

Figure 6: Performance on Encrypted Data - Mean Turnaround Time.

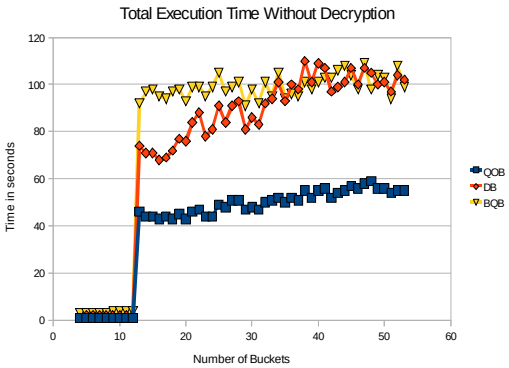

Figure 4: Process Turnaround Times

Reference

1. Hore, B., Mehrotra, S., Tsudik, G.: A Privacy-Preserving Index for Range Queries. In: International Conference on Very Large Data Bases, pp. 720–731, 2004.

2. Agrawal, R., Kiernan, J., Srikant, R., and Xu, Y.: Order Preserving Encryption For Numeric Data. In: Book Order preserving encryption for numeric data, Series Order preserving encryption for numeric data, ed., Editor ed.^eds., pp. 563-574, ACM, 2004.

3. Hacigumus, H., Iyer, B., and Mehrotra, S.: Providing Database as a Service. In: IEEE International Conference on Data Engineering (ICDE), San Jose , California , 2002.

4. Mykletun, E., Tsudik, G.: Incorporating a Secure Coprocessor in the Database-as a-Service Model. In: International Workshop on Innovative Architecture for Future Generation High Performance Processors and Systems, 2005.

5. Yvonne Y., and Huiping G.: An Improved Indexing Scheme for Range Queries. In: International Conference on Security and Management(SAM'08)", Las Vegas, 2008.

6. The UCI KDD Archive. Forest CoverType Database <http://kdd.ics.uci.edu/databases/covertype/covertype.html>

7. Ozcelik, Y., Altinkemer, K. : Impacts of Information Technology (IT) Outsourcing on Organizational Performance: A Firm-Level Empirical Analysis. In: 17th European Conference on Information System, 2009.

8. Haber, S., Horne, W., Sander, T., Yao, D.: Privacy-Preserving Verification of Aggregate Queries on Outsourced Databases. In: Technical Report HPL-2006-128, HP Labs, 2006.

9. Damiani, E., Vimercati, S., Jajodia, S, Paraboschi, S., Samarati, P.: Balancing Confidentiality and Efficiency in Untrusted Relational DBMSs. In: Proc. 10th ACM Conf. On Computer and Communications Security, Washington, DC, pp. 93-102, 2003.

10. Hacigümüs, H., Iyer, B., Li, C., Mehrotra, S.: Executing SQL Over Encrypted Data in the Database-Service-Provider Model. In: SIGMOD Conference, pp. 216-227, 2002.

11. Alwarsh, M.: An Improved Algorithm for Querying Encrypted Database. Bowling Green State University, Master's project, 2010.