Vol.2011-MPS-85 No.31
2011/9/16

Semi-ShuffledBF :ブルームフィルタを用いた安全かつより高速なプライバシ保護検索手法の提案

金子静花[†]　天笠俊之[††]　渡辺知恵美[†]

DaaS（Database as a Service）において，ユーザはインターネット上の第三者が管理するデータベースのサービスをネットワーク経由で利用することができる．このような環境では，ユーザがデータ管理者から機密情報を守ることが困難となる．この問題に対し，我々は先行研究において「ブルームフィルタを用いたスキーマ情報を隠蔽するプライバシ保護検索手法」を提案した．この手法では，各タプルに対して問合せ用のブルームフィルタを生成し，タプル毎にキーを用いてブルームフィルタのビット列をシャッフルする（ShuffledBF）．これにより，ビットパターンの漏えいを防ぐことが可能となる．その反面，問合せの際，各タプル毎にハッシュ関数を適用してシャッフルしたビット列を復元する必要があり，タプル数に比例した処理時間がかかってしまうという問題があった．　一方，ブルームフィルタのシャッフルを行わない（Non-ShuffledBF）場合，ビットパターンの推測が可能となりセキュリティ上問題がある．　そこで我々は，問合せの第 1 段階の絞り込みに Non-ShuffledBF を用い，第 2 段階の絞り込みに ShuffledBF を用いるハイブリッドな手法 Semi-ShuffledBF を提案する．

Semi-ShuffledBF :Performance Improvement of a Privacy-Preserving Query Method for a DaaS Model Using a Bloom filter

SHIZUKA KANEKO[†] TOSHIYUKI AMAGASA[††] CHIEMI WATANABE[†]

In database-as-a-service, users can utilize a database service that is maintained by third parties via the Internet. In such an environment, it becomes difficult for the user to hide confidential information from the data administrator. To solve this problem, we previously proposed “Privacy-Preserving Query Method Hiding Schema Information Using a Bloom filter.” With this method (ShuffledBF), we generate a Bloom filter for the queries of each tuple and shuffle bit sequence by using the key in each tuple. In this way, it is possible to prevent the leakage of bit patterns. On the other hand, it is problematic that the time required is proportional to the number of tuples processed, because we must restore the shuffled bit sequence by applying a hash function to each tuple when we run the query. While, in contrast, the case of a Non-Shuffled Bloom filter(Non-ShuffledBF) has a security problem. Therefore, we propose a hybrid technique called Semi-ShuffledBF that consists of two steps: 1) Non-ShuffledBF and 2) ShuffledBF.

1. Introduction

Recently, database-as-a-service (DaaS) has attracted considerable attention. DaaS provides a data management service in the cloud computing environment. Many DaaS services have already been provided by Amazon, Google, Microsoft, et al., such as S3, EC2, SimpleDB, Azure, Google Apps Engine, and so on. DaaS services are used by individuals and small companies who find it difficult to administer the DBMS on a constant basis.

In such an environment, we should note that DaaS administrators are third parties from the viewpoint of DaaS users. Therefore, it is natural that users need to hide sensitive data from DaaS administrators. To achieve such a user requirement, techniques for privacy-preserving query processing have been investigated by many researchers [1][2][3][5][7][9].By using the investigated techniques, users can store data that is encrypted at the client end, and can issue queries to the† encrypted database to receive the appropriate results, without leaking the original value of the stored data in the DBMS.

Figure 1 shows the general flow of the Privacy Protection Method.

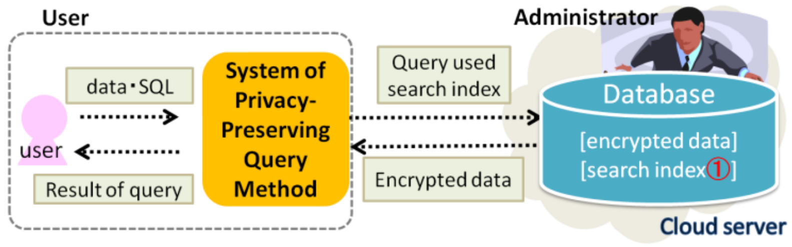


Figure 1: System of Privacy Preserving Query Method.

As the first step, the system encrypts each tuple on the client side, and sends the encrypted tuples to the database server. During this time, the system also sends the search index (① in Figure 1) for the corresponding tuple. The search index is used by the query processor on the database server to process queries without leaking the original value of the encrypted tuples. Previous studies [1][2][3][5][7][9] prepared the search index for each attribute in each tuple, and a scheme for making an index has been proposed according to the data types and operation types issued in the query.

†お茶の水女子大学
Ochanomizu University
††筑波大学
Tsukuba University

In our previous studies [12], we have proposed ShuffledBF, which is a technique for privacy-preserving processing using a Bloom filter. ShuffledBF combines the search indices for multiple attribute values in a tuple, and then, it conceals the schema information and distribution information of the original table. ShuffledBF generates hash values that are used for generating a Bloom filter by using the attribute and the identifiable values of the tuple. Therefore, even if two tuples have the same attribute value, the hash values that are generated from the attribute values are different from each other. By using this mechanism, ShuffledBF guarantees a high level of security. However, ShuffledBF has the problem of low processing speed.

In this study, we focus on Non-ShuffledBF. Non-ShuffledBF generates hash values based only on the attribute value, and if two tuples have the same attribute value, then the hash values are the same. The performance of query processing is obviously better by using Non-ShuffledBF than by using ShuffledBF; however, it cannot guarantee privacy preserving against adversaries. In this paper, we propose Semi-ShuffledBF, which is combining ShuffledBF to utilize the advantages of both the mechanisms. We investigate the query processing time in a single-server environment, and find that Semi-ShuffledBF can improve the query processing time in circumstances when the selection ratio is low.

The paper is organized as follows. The previous study on ShuffledBF is presented in Section 2, and the proposed structure of Semi-ShuffledBF is described in Section 3. We show the results of our performance measurements in Section 4 and the related research in Section 5. Finally, we discuss our conclusions and future work in Section 6.

2. ShuffledBF

ShuffledBF is a Privacy-Preserving Query Method using a Bloom filter. ShuffledBF can carry out a selection protection operation to a single relation; when selecting, it can carry out the operations of exact string, matching part, and number attribute. Therefore, ShuffledBF has high security, and its schema information and value cannot be guessed from the original data of the query and the stored data on the server.

The generation of the search index is described in Section 2.1, and the query is described in Section 2.2. The means of transrating a numeric attribute is presented in Section 2.3.

2.1 Generating the Search Index

In this section, we describe the means to convert tables, encrypt tuples, and convert queries. This method uses a Bloom filter for the search index. A Bloom filter is an index that can quickly determine whether a collection contains an element. It is characterized as space efficient and can perform faster searches and OR calculations, and detect false positives.

Figure 2 shows an example of a conversion table.

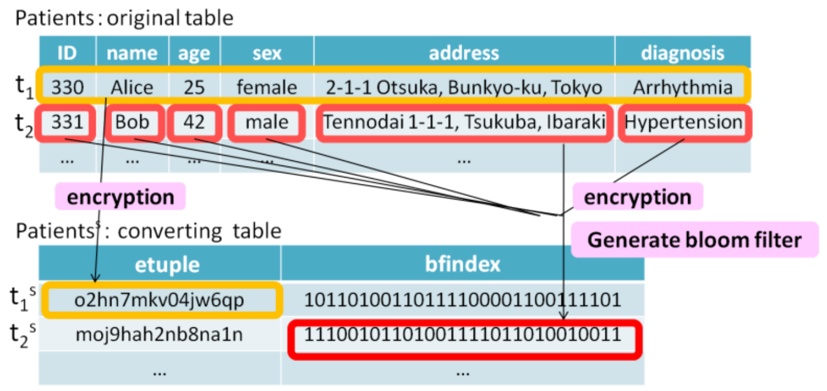


Figure 2: Example of a conversion table.

The table Patients is composed of the attributes ID, name, sex, address, and diagnosis. We prepare Patientss in the server that correspond to Patients. This table has only two attributes: etuple and bfindex. The attribute etuple stores the values of the encrypted tuples. The attribute bfindex is the search index of the tuple. Because there is only one bfindex created per tuple regardless of the source schema, it is difficult to determine the attributes of Patients that are derived from Patientss. This makes the distribution of the values confusing and prevents attacks on the data.

Figure 3 shows a flow diagram of generating a Bloom filter index (ShuffledBF) from the tuple t1.

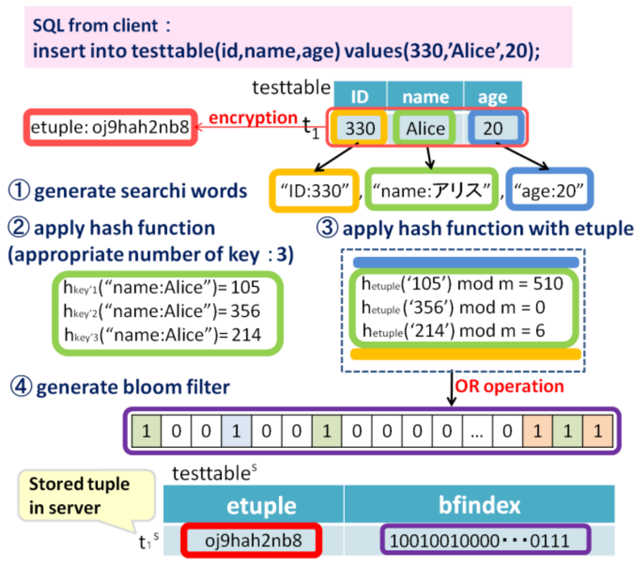


Figure 3: Generating a Bloom filter index.

Bfindex is an index that uses a Bloom filter. Its structure is based on the attribute names and values of the tuple. For example, the corresponding word of the value "Alice" of the attribute "attribute" in Figure 3's tuple t1 is "name:Alice" (① in Figure 3). For each word that is made in this way, multiple hash functions are applied.② in Figure 3 is an example of applying three hash functions for "word:Alice." We used the HMAC hash function and some required keys. ② in Figure 3 uses three keys: key1, key2, and key3. Next, we apply these hash values via HMAC using etuple as a key. If the tuples have the same value of standing bits in different locations, applying a second hash function prevents the gathering of features of the original data from the bit pattern. A Bloom filter index that does not apply the first hash function is called Non-ShuffledBF (NSBF).

2.2 Query

In this section, we describe the means of translating a query. Figure 4 shows an example of translating a query.

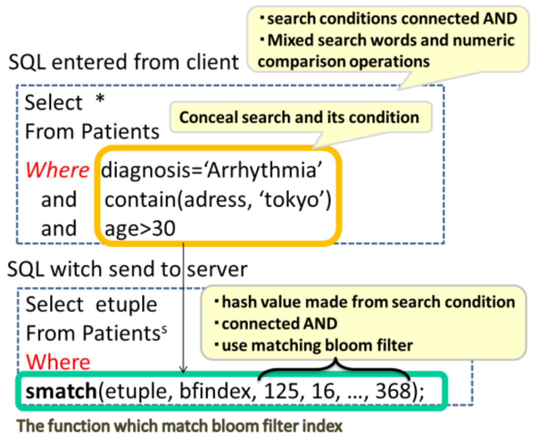

・search conditions connected AND
・Mixed search words and numeric comparison operations
SQL entered from client
Select *
From Patients
Conceal search and its condition
Where diagnosis='Arrhythmia'
and contain(adress, 'tokyo')
and age>30
SQL witch send to server
Select etuple
From Patients^s
Where
・hash value made from search condition
・connected AND
・use matching bloom filter
smatch(etuple, bfindex, 125, 16, ..., 368);
The function which match bloom filter index

Figure 4: Example of translation of query.

The upper SQL in Figure 4 is entered by the client, and the lower SQL is sent to the server. The lower SQL replaces the condition of the attributes and the text search of each tuple to the conditions of bfindex. Therefore, database administrators cannot read what we have specified as the condition of the attribute.

Next, Figure 5 shows how query processing proceeds on the server.

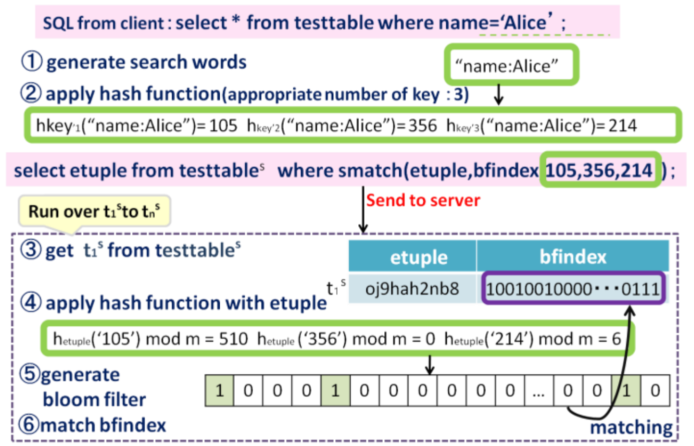
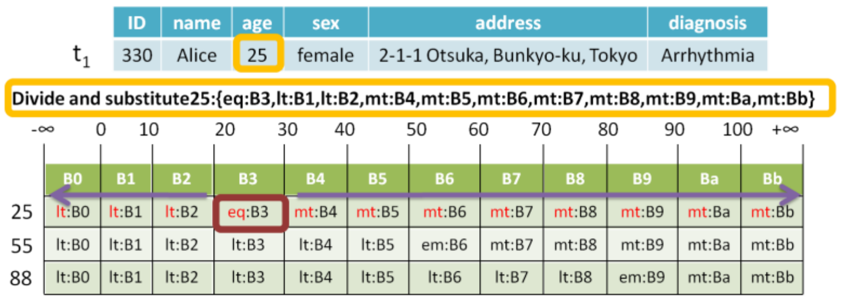

SQL from client : select * from testtable where name='Alice' ;
①generate search words
"name:Alice"
②apply hash function(appropriate number of key：3)
hkey1("name:Alice")= 105 hkey2("name:Alice")= 356 hkey3("name:Alice")= 214
select etuple from testtable^s where smatch(etuple,bfindex 105,356,214);
Send to server
Run over t1^s to tn^s
③get t1^s from testtable^s
etuple
bfindex
t1^s
oj9hah2nb8
10010010000･･･0111
④apply hash function with etuple
hetuple('105') mod m = 510 hetuple('356') mod m = 0 hetuple('214') mod m = 6
⑤generate bloom filter
1 0 0 0 1 0 0 0 0 0 0 ... 0 0 1 0
⑥match bfindex
matching

Figure 5: Example of processing query.

First, generate the words from the search condition (① in Figure 5), apply the first hash function, and then, send the SQL to the server (② in Figure 5). At the server side, match the query conditions by processing smatch function written in where section in SQL by each tuple. In the smatch function, we apply the hash function used by the key as the hash values generated at the client side (105,356,214 in④ in Figure 5). We match the values to the Bloom filter (⑤ in Figure 5). In this way, the user can hide the type of the number and operation to the schema information as well as the search condition.

2.3 Translating a Numeric Attribute

Because it is impossible for the Bloom filter to compare numbers, we need to translate the numbers in the data into words in order to apply a Bloom filter to the numbers. Basically, the domain is divided into several buckets of numeric attributes, and the generated words are added to the bucket name and attribute name.



	ID	name	age	sex	address	diagnosis
t_1	330	Alice	25	female	2-1-1 Otsuka, Bunkyo-ku, Tokyo	Arrhythmia

Divide and substitute25:{eq:B3,lt:B1,lt:B2,mt:B4,mt:B5,mt:B6,mt:B7,mt:B8,mt:B9,mt:Ba,mt:Bb}

	B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	Ba	Bb
25	lt:B0	lt:B1	lt:B2	eq:B3	mt:B4	mt:B5	mt:B6	mt:B7	mt:B8	mt:B9	mt:Ba	mt:Bb
55	lt:B0	lt:B1	lt:B2	lt:B3	lt:B4	lt:B5	em:B6	mt:B7	mt:B8	mt:B9	mt:Ba	mt:Bb
88	lt:B0	lt:B1	lt:B2	lt:B3	lt:B4	lt:B5	lt:B6	lt:B7	lt:B8	em:B9	mt:Ba	mt:Bb

Figure 6: Representation of bucket of numeric attributes.

In Figure 6, 25,55,88 represent each set of words.

If the limit of the bucket is less than the value of v against all buckets B = B1, ···, Bb, we add the word 「<attribute name>:lt:<bucket name>」. If the limit of the bucket is more than the value of v, we add the word 「<attribute name>:mt:<bucket name>」. In another case, we add the word 「<attribute name>:eq:<bucket name>」. When comparing the values with the magnitude using this method, for example, if you want to get a value greater than 75, you focus on (B7) (left of (B8)) that contain 75, and search for the tuple that has the word 「<attribute name>:lt:B7」. On the other hand, if you want to get a value less than 35, you focus on the right bucket (B5) and search for the tuple that has the word 「<attribute name>:lt:B5」. Thus, we can treat a numeric comparison operation as a matching string.

3. Semi-ShuffledBF

ShuffledBF is secure, because it is difficult to estimate the original data from the Bloom filter on the server. However, there is a problem that the processing time is very long, because we apply the hash function against all tuples with each query (see Figure 10). Therefore, we propose Semi-ShuffledBF, which can perform privacy-preserving queries without rack safety and is faster than combined Non-ShuffledBF that does not apply a conversion function.
We describe the basic idea of Semi-ShuffledBF in Section 3.1, the way of inserting data in Section 3.2, and the method of the query in Section 3.3.

3.1 Basic Idea

Semi-ShuffledBF is an index that is a combination of the ShuffledBF and Non-ShuffledBF indices, which narrows down the number of tuples to be applied to a hash function. At query time, we expect to reduce the number of tuples by applying it to the hash function by narrowing down using Non-ShuffledBF and ShuffledBF indices. We apply the following hash function on the Non-ShuffledBF index:

$$h'_i(x) = \lceil |g_j(h_i(x) \ mod \ \lceil m/l \rceil)| \rceil \ mod \ m \quad \cdots \quad (1)$$

Here, m is the bit length of the Bloom filter, l is an integer parameter and takes the value of at least 2, and the function g_j is set for each attribute A_j in the source table R (A_1, …, A_n). On the other hand, on ShuffledBF we set up the remainder of m, which is the bit length of the Bloom filter, and on Non-ShuffledBF, we set up the remainder of [m/l]. We increase the false positives from Non-ShuffledBF by increasing the value of l, so it is difficult to estimate the original data. The g_j function is a function for determining the location of the bit standing of each attribute A_j, and for preventing the duplication of the position of a bit in a single Bloom filter. In addition, Semi-ShuffledBF does not separate Non-ShuffledBF from ShuffledBF, so the false positives may be higher by combining them. However, we consider that this can be adjusted by making the bit length m of the Bloom filter slightly longer.

3.2 Inserting data

The insertion of data into Semi-ShuffledBF is divided into the following four stages:

(1) Generate ShuffledBF (Figure 3 in Section 2.1).
(2) Generate Non-ShuffledBF.
(3) Carry out an OR operation in ShuffledBF and Non-ShuffledBF.
(4) Store the result in bfindex.

First, we generate the search words in ShuffledBF. And we generate it in Non-ShuffledBF by applying the Equation(1). We store the result by performing an OR operation in Non-ShuffledBF and ShuffledBF as Semi-ShuffledBF in bfindex.

3.3 Method of the Query

The method of the query of Semi-ShuffledBF is divided into the following two stages.

(1) Search with Non-ShuffledBF.
(2) Search with ShuffledBF.

First, apply Non-ShuffledBF to each tuple. Thus, only the tuples that correspond to Non-ShuffledBF are applied to ShuffledBF.

3.4 Effect of Semi-ShuffledBF

The effect of using Semi-ShuffledBF is a secure, more rapid search. ShuffledBF has the problem that the processing time is very long, because we apply the hash function against all tuples with each query. Semi-ShuffledBF uses Non-ShuffledBF, which can process rapidly and apply only the tuples matched by the Non-ShuffledBF hash function. In this way, Semi-ShuffledBF becomes faster.

Figures 7 and 8 show examples of ShuffledBF and Semi-ShuffledBF queries.

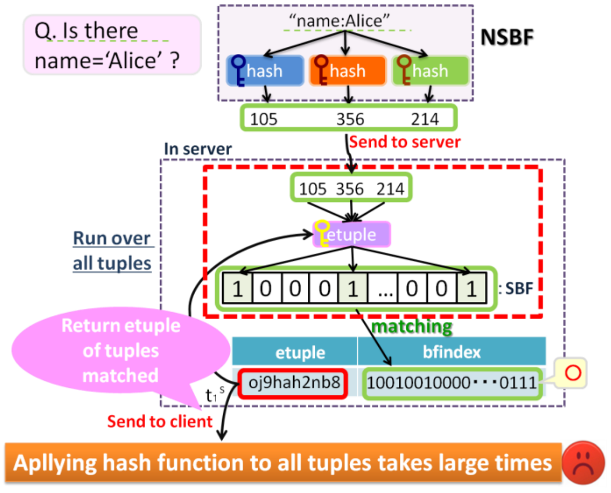


Figure 7: Example of ShuffledBF query.

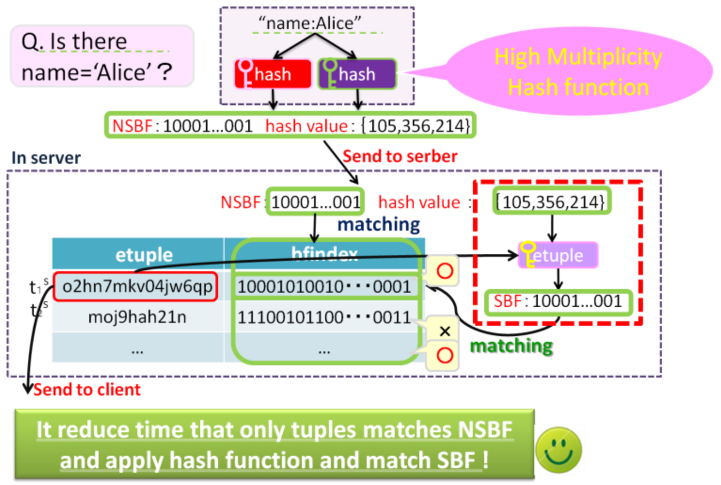

Q. Is there name='Alice' ?
"name:Alice"
hash
hash
High Multiplicity Hash function
NSBF : 10001...001 hash value : {105,356,214}
In server
Send to serber
NSBF 10001...001 hash value : {105,356,214}
matching
etuple
bfindex
etuple
t₁ˢ
t₂ˢ
o2hn7mkv04jw6qp
10001010010・・・0001
O
moj9hah21n
11100101100・・・0011
×
...
...
O
SBF : 10001...001
matching
Send to client
It reduce time that only tuples matches NSBF and apply hash function and match SBF !

Figure 8: Example of Semi-ShuffledBF query.

The process marked within red dots in Figures 7 and 8 is the part of the process that is performed faster. Figure 7 shows the process for all tuples; on the other hand, Figure 8 shows the process for only the tuples matched with Non-ShuffledBF. This improved processing speed is because the process for the tuples was omitted, which is not correct.

4. Performance Evaluation

As a preliminary experiment for the proposed method, we extended the programs of the Privacy-Preserving Query Method that is built on previous research, used DBMS to improve the performance, and evaluated the performance.

4.1 Experimental Environment

We evaluated the performance using a Linux Server (CPU: Intel (R) Xeon (R) 2.00 GHz Memory: 8 GB) and a database server (PostgresSQL) as the experimental environment. We used 100,000 tuples of artificial data and specified the length of the Bloom filter m as 128 bytes, the function gj as a primary function, and the partition number l as 10.

In this study, we measured only the query processing time on the server.

In the Privacy-Preserving Query Method, users can obtain the result by re-querying the data after leeching and decoding the correct tuple that is searched on the server. In fact, the time for leeching to the client and decoding may be very long, but in this study, we do not consider this duration because it is beyond the scope of this proposed method.

4.2 Experimental Results

Following are the performance results of ShuffledBF, Non-ShuffledBF, and Semi-ShuffledBF on the experimental server.

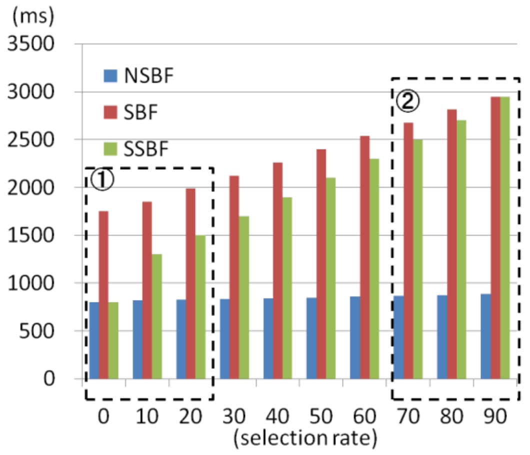


Figure 10: Performance results
(selection rate sets "the number of correct tuples/the number of all tuples").

The case where the selection rate is a row (① in Figure 10), processing efficiency is improved when using Semi-ShuffledBF. However, the case where the selection rate is high (② in Figure 10), the processing efficiency does not improve.

4.3 Considerations

In this experiment, there are two reasons why Semi-ShuffledBF has not yet improved significantly compared to ShuffledBF.

(1) The process used for Non-ShuffledBF in not very fast.

We expected that the process would be fast, because we were basically operating with only bit, but it actually took about 700 ms. This is because you need a table scan to check all the tuples.

(2) The same action (processing times) occurs for answer tuples.

Semi-ShuffledBF reduces processes to not fit query condition. In the case that the selection rate is high, however, the effect is small because the processes are not many. Therefore, the result is the same as in the preceding section.

Following are some possible ways to improve these two issues:

(1) Grant bitmap indices for Semi-ShuffledBF.

It is suggested that bitmap indices be granted to bfindex to make the primary search on Non-ShuffledBF faster.

Thus, because you can access the location of the bits directly, without performing a file scan of all the tuples, it is expected that the I/O costs of the disk can be reduced.

(2) Do not search by ShuffledBF.

In the case when the selection rate is high, use Semi-ShuffledBF to search most of the tuples rather than ShuffledBF.

It is suggested to calculate the selection rate after searching by Non-ShuffledBF. Thus, in the case when the selection rate is high, it is considered that all the tuples are correct and a search by ShuffledBF is not required.These approaches will increase the false positives of the search, but we believe that this problem can be solved by the artifice in client as mentioned in Section 4.1.

5. Related Research

Many studies have been performed on the Privacy-Preserving Query Method for outsourcing. Hacigumus et al. proposed to store the search index to the database on the server, how to query the index on the server generated by a user and the generation of query exection which exect divided instead of the query on client .The search index is provided for each attribute and are produced by different methods in data types and the calculation of used the conditions.

In the method of generating the index, the distributed value may obtain the original value. On the other hand, Hore et al. proposed a method for splitting the bucket, which makes it difficult to estimate the value of the distribution [7]. Agrawal et al. [1] proposed a conversion method of the number attribute that preserves the relationship. This method can prevent the estimation of the original value by converting the distribution of values that are different from the original distribution. It can process both compared and combined operations. Lee et al. [8] and Hasan et al. [6] have used the proposed method. Aggregations and k-neighbor [11] used an encryption method with homomorphism, which has also been proposed by Mykletun et al. [4] and Ge et al. [10]. In these studies, there is a problem of security and performance, as exists in our research. On creating an index for each cell, there is a problem that if there are many data on the server, it is possible that someone may obtain the original value by analyzing the trend of the index values. In case of checking the condition for each tuple, there is the problem that you cannot use the index. In addition, if you create an index, someone may obtain the original value from the index. In our proposed method, the possibility of obtaining the original value is low compared to the method of generating an index for each cell, because the index is in one tuple. It is anticipated that Semi-ShuffledBF displays a performance improvement by applying a bitmap index of the Bloom filter. We conclude that the ability to obtain the original values from the index is low because it contains shuffled bits.

6. Conclusions and Future Work

We proposed Semi-ShuffledBF, which can perform Privacy-Preserving Queries without rack safety and faster than Non-ShuffledBF that does not apply the conversion function.

In the future, we plan to speed up the performance of Semi-ShuffledBF and establish its indicators of performance and security.

We will also evaluate the performance of other DaaS, such as Windows SQL Azure.

References

1) **Agrawal R., Kiernan J., Srikant R., and Xu Y.: Order preserving encryption for numerical data, Proceedings of the 2004 SIGMOD International Conference, pp.563‐574（2004）.**
2) **Bellovin S. and Cheswick W. : Privacy-enhanced searches using encrypted bloom filters”（2004）.**
3) **Boneh D., Crescenzo G.D., Ostrovsky R., and Persiano G.: Public Key Encryption with Keyword Search, Proceedings of EUROCRYPT ’ 04, vol.3027 LNCS, pp.506–522 (2004）.**
4) **E. Mykletun, G. Tsudik：Aggregation queries in the database-as-a-service model. IFIP WG 11.3 on Data and Application Security （2006） .**
5) **H. Hacigumus, B. Iyer, C. Li, and S. Mehrotra.:“Executing SQL over Encrypted Data in the Database-Service-Provider Model,” Proceeding of the ACM SIGMOD International Conference on Management of Data, pp. 216-227,（2002）.**
6) **Hasan Kadhem, Toshiyuki Amagasa, and Hiroyuki Kitagawa：“A Secure and Efficient Order Preserving Encryption Scheme for Relational Databases," Int'l Conf. on Knowledge Management and Information Sharing（KMIS 2010）, Valencia, Spain, October 25-28（2010） .**
7) **Hore B, Mehrotra S., and Tsudik G.：A privacy-preserving index for range queries, Proceedings of the 30th International Conference on Very Large Data Bases, pp.720–731（2004） .**
8) **S. Lee, T. Paek, D. Lee, T. Nam, and S. Kim： Chaotic Order Preserving Encryption for Efficient and Secure Queries on Databases, IEICE Transactions on Information and Systems E92.D（11）, 2207-2217（2009） .**
9) **Ting Yu and Shushil Jajodia：Secure Data Management in Decentralized Systems, Springer-Verlag NewYork Inc, p.462（2006） .**
10) **Tingjian Ge, Stanley B. Zdonik：Answering Aggregation Queries in a Secure System Model. Proceedings of VLDB 2007, pp.519-530（2007） .**
11) **W. K. Wong, D. W. Cheung, B. Kao and N. Mamoulis：Secure kNN computation on encrypted databases, Proceedings of the 35th VLDB Conference, pp. 139-152（2009）**
12) **Watanabe C. and Arai Y.：Privacy-Preserving Queries for a DAS model using Two-Phase Encrypted Bloomfilter, Proc. of International Conference on Database Systems for Advanced Applications (2009）.**

謝辞 本研究の一部は科学研究費補助金若手研究（B）（課題番号: 21700099）によるものである.