Secure Bi-attribute Index: Batch Membership Tests over the Non-sensitive Attribute

ABSTRACT

This paper introduces a secure and efficient bi-attribute indexing structure, leveraging a matrix Bloom filter, to support the coexistence of attributes and efficient batch operations over plaintext data. Traditional secure index techniques enable keyword search on encrypted univariate data, but fall short in handling bi-attribute data prevalent in AI and data mining applications. Conventional approaches suffer from inefficiencies when executing prefix queries over one attribute, due to the necessity of duplicate trapdoor generations. However, when one attribute is non-sensitive, it can be processed in plaintext, enhancing query performance while potentially compromising privacy due to inter-attribute correlation. This correlation poses a risk of inference attacks by adversaries possessing auxiliary knowledge, threatening the privacy of even encrypted sensitive attributes.

We address these issues by introducing a case study illustrating searchable encryption on time series data, along with two variants of matrix Bloom filter designed to handle different workload patterns based on prior knowledge of the dataset. To mitigate privacy concerns, we formally define and achieve bounded privacy loss by incorporating noise using the randomized response technique. Consequently, the initialized index complies with locally differential privacy, offering a cryptographically strong guarantee for any set of non-sensitive attribute items.

ACM Reference Format:

. 2018. Secure Bi-attribute Index: Batch Membership Tests over the Non-sensitive Attribute. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 13 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

1.1 Backgrounds

The advent of advanced cryptographic techniques [13, 16] has revolutionized how sensitive data is stored, allowing secure outsourcing to untrusted third parties such as cloud servers. However, this level of security presents challenges when searching encrypted data, as it is inefficient to decrypt all ciphertexts before conducting a search. Searchable encryption, which facilitates keyword search on encrypted data, is designed to solve this issue. With the ever-growing scale of data, it is increasingly impractical to store and process raw data without incurring high costs. Thus, probabilistic data structures, which provide an approximate method for processing searchable encryptions, are gaining traction. The secure index [18] is a notable example, enabling users with a "trapdoor" (generated from a secret key for a keyword) to test for its existence in a Bloom filter (BF) [4]. However, this conventional approach is designed for processing univariate data, whereas many AI and data mining applications require bi-attribute data, e.g., key-value pairs or spatiotemporal data.

Consider the example of log files in operating systems and web servers: each log entry typically records a timestamp along with the log event, essentially forming a bi-attribute record (IP, timestamp). Repeated instances of the same IP address in a log file lead to prefix queries over the timestamp, such as, "Did 158.132.50.85 access the server at 9:00, 9:15, 9:20, and 10:00?" If both attributes are encrypted and processed as a whole, each query must be processed independently with a unique trapdoor generation, which is highly inefficient.

Fortunately, it is often the case that not all attributes of data are sensitive in real-world scenarios [25]. When treated in plaintext, the non-sensitive attribute can facilitate batch operations, such as prefix queries, within a single trapdoor generation. As an example, consider the web server log file illustrated in the left section of Table 1. Here, the IP address is the sensitive attribute, while the timestamp is not. Notice that the trapdoor 'e6c5d3422583e8e5' repeats twice and is likely to surface elsewhere in the log file. For a range query over the timestamp–for instance, an administrator may want to verify if a specific IP accessed the server between 19:00-20:00–a single trapdoor generation suffices. While this approach alleviates the cost of trapdoor generation, it is vulnerable to inference attacks

Table 1: A web server log file.

Ground truth

Time	IP trapdoors
19:22:00	3f3c17b5cfb73e22
19:22:15	ced40518c3e32660
	...
19:23:19	e6c5d3422583e8e5
	...
19:25:55	e6c5d3422583e8e5

With false positives

Time	IP trapdoors
19:22:00	3f3c17b5cfb73e22
19:22:15	ced40518c3e32660
	...
19:23:19	e6c5d3422583e8e5
19:23:24	ced40518c3e32660
	...
19:25:55	e6c5d3422583e8e5
19:25:58	ced40518c3e32660

Table 2: Auxiliary knowledge from compressed session data.

Time slot	IP
19:21:55-19:22:05	158.132.150.83, 141.211.29.122
19:22:15-19:22:25	158.132.50.12, 70.108.10.124
19:22:40-19:22:50	124.197.24.255
19:23:15-19:23:25	158.132.150.83, 167.136.142.43
19:25:50-19:26:00	59.160.0.12, 167.136.142.43

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

where an adversary could exploit the plaintext outcome to deduce the sensitive attribute using **auxiliary knowledge**.

We highlight this vulnerability through an example where the adversary possesses auxiliary knowledge in the form of session data, as shown in Table 2. This table lists all the IP addresses that accessed the server during five distinct time slots[1]. Consequently, by comparing the plaintext of trapdoor 'e6c5d3422583e8e5' with the encrypted log file, it is highly plausible that the IP address is '167.136.142.43', a scenario akin to a known plaintext attack. This illustration underscores that revealing the outcomes of the non-sensitive attribute can lead to privacy loss [24] for the sensitive attribute when the two are **correlated**.

The Bloom Filter (BF) structure, owing to its probabilistic nature, provides a semblance of privacy protection, a characteristic deemed as a "better than nothing" privacy guarantee [3]. This is attributable to the propensity of false positives, which can return a "true" value even for non-existent items. This phenomenon is demonstrated using our previous log file example, where the false positive (highlighted in red) in the right section of Table 1 contradicts the auxiliary knowledge. Although BFs offer this level of privacy protection, it doesn't equate to a cryptographic guarantee for every item.

In this paper, we employ the randomized response technique [34] to introduce noise into the index, thereby ensuring that outcomes of the non-sensitive attribute comply with local differential privacy [12]. Our objective is to concurrently cater to batch processing over the non-sensitive attribute and inter-attribute privacy, thereby facilitating a secure and efficient bi-attribute index that can be deployed on public cloud platforms.

1.2 Contribution

The contributions of this paper span two key areas:

Efficient Query Processing. We introduce a matrix BF-based index as a 2-dimensional extension of the traditional secure index. This structure, which processes two attributes in a partitioned manner while maintaining their coexistence, supports batch operations on one attribute and tailored query strategies on two attributes using split hashing. This is exemplified in a case study that facilitates searchable encryptions over the sensitive attribute and range queries over the non-sensitive attribute within a single trapdoor generation. The locality-sensitive hashes we use circumvent the need for linear, brute-force searches across the entire history. Additionally, we propose two versions within the matrix BF framework, i.e., minimum storage matrix and maximum adaptive matrix. These variations strike a balance between insertion efficiency and membership test efficacy for different workload patterns, capitalizing on common statistical distributions of datasets.

Bounded Privacy Loss. In the context of the data correlation between the two attributes, achieving full utility of the non-sensitive attribute while guaranteeing zero privacy loss of the sensitive attribute simultaneously is unfeasible. To address this issue, we formally define privacy loss as bound by a specific security parameter. This is ensured through the addition of noise via the Randomized Response (RR) technique [34]. Building on this, we propose an initialization approach for the index that guarantees a reasonable level of privacy loss. We then theoretically derive the false positive rate of the matrix BF index. This refers to the probability that an item that does not exist is reported as "true". Based on this derivation, we can determine the maximum capacity of the matrix BF index.

The rest of the paper is organized as follows. Section 2 introduces the necessary preliminaries. Section 3 formalizes the problem, explores baseline solutions, and outlines the threat model. Section 4 delves into the proposed approach in detail and provides theoretical analysis. Section 5 introduces the two variations of the matrix BF. Experimental settings and results are discussed in Section 6. Section 7 reviews related work, and Section 8 concludes the paper.

2 PRELIMINARIES

2.1 Secure Indexes

The idea of a secure index [18] is to generate a trapdoor with pseudo-random functions, which are defined later, and insert the trapdoors into a per-document Bloom filter. When a client wants to know which document contains a value he wants, he generates a trapdoor with the value, sends it to the third party where the document is stored and asks it to perform the searching on the trapdoor. Such an index scheme consists of the following four algorithms:

- $Keygen(\lambda)$: Given a security parameter λ, outputs the private key K_{priv}.
- $Trapdoor(K_{priv}, v)$: Given the master key K_{priv} and value v, outputs the trapdoor T_v for v.
- $BuildIndex(D, K_{priv})$: Given a document D and the private key K_{priv}, outputs the index ID.
- $SearchIndex(T_v, ID)$: Given the trapdoor T_v for value v and the index ID for document D, outputs 1 if $v \in D$, otherwise 0.

Pseudo-random functions. A pseudo-random function $f_k(\cdot) : \{0,1\}^n \times \{0,1\}^s \rightarrow \{0,1\}^m$ is said to be efficiently computable, if for any input $x \in \{0,1\}^n$, there exists a key $k \in \{0,1\}^s$, such that:

- With knowledge of k, $f_k(\cdot)$ has a succinct representation and shall be computationally light.
- Without knowledge of k, $f_k(\cdot)$ shall be computationally hard.

2.2 Locality Sensitive Hashes

The location-sensitive hashing (LSH), proposed by Indyk and Motwani, is a technology mapping similar items into the same hash buckets with high probability [22] and is usually used for the similarity search. Given a similarity search query request, the query item q will be mapped to multiple buckets in the hash table, and we consider items that appear in these buckets are similar points to q. All these similar items will be considered as a group. Generally, the hash function of items close to each other is mapped to the same hash bucket with a higher probability. Let S denote the domain of input items and $||*||$ the distance metric between two items. For any items p and q, LSH function family, i.e., $H = \{h : S \rightarrow U\}$ is called (R, cR, P_1, P_2)-sensitive for distance function $||*||$ if

$$||p,q|| \leq R \; then \; Pr_H[h(p) = h(q)] \geq P_1$$
$$||p,q|| > cR \; then \; Pr_H[h(p) = h(q)] \leq P_2.$$

[1]Such data could originate from the historical records of a compromised proxy server or a firewall, which is compressed in this form to conserve storage

We set $c > 1$ and $P_1 > P_2$ to better do similarity search. The gap between P_1 and P_2 is usually enlarged by using multiple hash functions. Given an d-dimension input item v, different $||*||$ denotes different LSH families which allow the hash function $h_{a,b}$ $R^d \rightarrow Z$ to map v into a hash bucket number. The hash function $h_{a,b}$ in H can be written as:

$$h_{a,b}(v) = \lfloor \frac{av + b}{w} \rfloor,$$

where a is a d-dimension vector randomly chosen from a s-stable distribution, b is a random number chosen from domain $[0, w)$ and w is a large constant.

2.3 Local Differential Privacy

Local differential privacy (LDP) [12] is proposed to provide a stringent privacy guarantee for users in a non-trusted data-collecting system. We say a perturbation mechanism $\mathcal{M}$, a non-deterministic algorithm mapping an input to some output with a certain probability, satisfies with ϵ-LDP ($\epsilon \geq 0$), if and only if for any two data records S_1, S_2 and any possible output $T \in \text{Range}(\mathcal{M})$, the following condition holds:

$$\frac{P[\mathcal{M}(S_1) = T]}{P[\mathcal{M}(S_2) = T]} \leq e^{\epsilon} \tag{1}$$

This is a formal definition of LDP, and the set of all possible outputs is called the value range. Since $P[\mathcal{M}(S_1) = T]$ is very close to $P[\mathcal{M}(S_2) = T]$, it is hard to tell any individual's true answer from observing the outcome. The most straightforward perturbation algorithm to guarantee LDP is the Randomized Response (RR) [34], which has been widely used in the "Yes or No" sensitive problem. Users will flip a biased coin and send true answers with probability p and $1 - p$ for false answers. A widely used RR for binary cases is given as follows:

$$b' = \begin{cases} b, & w.p.\ p \\ 1 - b & w.p.\ 1 - p \end{cases}$$

3 SYSTEM OVERVIEW

3.1 Problem Definition

Generally, a bi-attribute index serves as an efficient strategy for executing queries over two attributes of a single data piece in a partitioned manner. This is refined into a membership test on the **co-existence** of both attributes.

Definition 3.1. **Bi-Attribute Membership Test.** Given a set of bi-attribute data, where (x, y) symbolizes a piece of data from that universe, a bi-attribute membership test over (x, y) is a bivariate function $Q(x, y) = 1$ or 0. This returns true or false for the co-existence of (x, y) in the given set.

This type of membership test must support prefix queries over one attribute. Consequently, when the suffix attribute is readily visible in plaintext, an efficient batch operation becomes feasible, leading to the concept of a bi-attribute batch membership test.

Definition 3.2. **Bi-Attribute Batch Membership Test.** Given a constant value x_1 of variable x (or y_1 of y), a bi-attribute membership test $Q(x, y)$ is deemed batchable if there exists a univariate membership test $Q_y(x_1, y) = 1$ or 0 (or $Q_x(x, y_1)$) for y, returning true or false for the co-existence of (x, y).

3.2 Baseline Solutions

In this section, we will consider some existing solutions and discuss their ability to support bi-attribute batch membership tests.

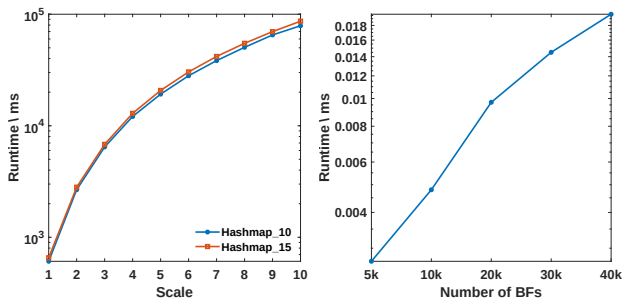
Naive Approach Using a Single BF. A straightforward approach for conducting bi-attribute membership tests is to process the bi-attribute data using a conventional secure index as a whole. However, this method fails to perform bi-attribute batch membership tests since it necessitates dedicated trapdoor generations for each query.

Utilizing Hashmaps. Hashmaps are a widely used category of probabilistic data structures for constructing an index for key-value pairs. They store data in its entirety, which can be space-consuming. Additionally, addressing collision cases requires values to be stored by chaining, and as the scale increases, the search can become costly (even with a b-tree). We validate this with a simple simulation in Fig. 1 (a), where the length of the hashmap is initialized to 10 and 15 times the number of elements at scale=1 (Hashmap_10 and Hashmap_15, respectively) and eventually reduces to 1 and 1.5 times at scale=10. Despite the time complexity for key search being O(1), it does not apply to values.

Employing Multiple BFs. Another intuitive solution involves the use of multiple BFs, a collection of co-working standard BFs. A common example is the multi-dimensional BF (MDBF) proposed by Guo et al., for the insertion and query of an l-dimensional element $(x_1, x_2, ..., x_l)$[20]. MDBF allocates l different standard BFs of the same length, inserting each dimension into its corresponding BF. When querying an element, MDBF checks each dimension in its corresponding BF. Another notable solution is the combinational BF[21]. It handles the membership query of S, which has multiple groups, and the outcome should reveal to which group the input belongs, without prior knowledge of the item counts in each group. However, its performance is not ideal due to the requirement of many hash functions and memory accesses. Furthermore, the probability of misclassification in a combinational BF is high, as false positives on each bit result in incorrect group codes.

In general, existing solutions with multiple BFs function as a set of independent BFs, failing to establish a member relationship within the set of BFs. Thus, using multiple BFs essentially necessitates knowing the BF where one of the attributes is stored, leading to a linear search on a single attribute, then searching the members of that attribute from the corresponding BF. However, the bi-attribute batch membership test fundamentally requires this to be an O(1) complexity process, irrespective of the number of BFs employed. We validate this with a simple simulation in Fig. 1 (b), showing that the runtime of BF search increases with the scale of BF sets, even though the time complexity is O(1) for each lookup in the corresponding BF.

Summary. We have summarized the introduced baselines in the following table, and none of them perfectly satisfy the three conditions. Therefore, in the upcoming section, we will introduce a new data structure, the matrix BF, to address the efficiency issue.



(a) Hashmap. (b) Multiple BFs.

Figure 1: The runtime of hashmap and multiple BFs with scale increasing.

Table 3: A summary of naive baseline solutions

	Batchable	Scalability	Co-existence
Single BF	No	Bi-attribute	Yes
Multiple BFs	Yes	Single attribute	Misclassification
Hashmap	Yes	Single attribute	Yes

3.3 Threat Model

In our setup, the sensitive attribute is encrypted, and the non-sensitive attribute is visible in plaintext. We assume an honest-but-curious adversary with extensive auxiliary knowledge (e.g., one in Table 2) of the inter-attribute correlation, who will attempt to infer the sensitive attribute by eavesdropping on the outcome of the non-sensitive one. An ideal but unattainable method would offer full utility of the non-sensitive attribute with zero privacy loss of the sensitive attribute simultaneously [24]. However, to make it practical, we only require that any published set of outcomes of the non-sensitive attribute results in a bounded privacy loss of the sensitive one:

Definition 3.3. **Security Definition.** A bi-attribute batch membership test $Qy(x_1, y)$ (over the non-sensitive attribute y) has a **bounded privacy loss** if and only if there exists a parameter λ that can converge to 0, such that the following formula holds for any trapdoor T_j and any pair of plaintexts $Pi1$, P_{i2} of attribute x:

$$Pr(T_j \leftarrow Trap(P_{i1})|Ak, \mathbb{Y}, y_1) - Pr(T_j \leftarrow Trap(P_{i2})|Ak, \mathbb{Y}) \leq \lambda \tag{2}$$

where Ak represents the auxiliary knowledge of inter-attribute correlations, $\mathbb{Y}$ is any set of known outputs of y that the adversary is aware of, y_1 is any potential output of $Q_y(x_1, y)$ over y, and $Pr(T_j \leftarrow Trap(P_i))$ is the probability that the adversary could ascertain that a trapdoor T_j is generated from plaintext P_i.

The parameter λ is only a logical concept employed to regulate the privacy loss within the security model. In the following section, we will elucidate how the LDP technique endeavors to assure a bounded privacy loss, managed by the parameter ϵ. Notably, ϵ parallels the role of λ as it characterizes the capability to protect privacy.

4 THE MATRIX BF INDEX

The BF is a well-known space-efficient probabilistic data structure, proposed by Burton Howard Bloom in 1970 [4]. The standard Bloom filter is an array of m bits, all set to 0 initially, along with k independent and uniformly distributed hash functions. To insert an item, we hash it with the k hash functions, and then set the corresponding positions in the bit array to 1. To query whether an item is in the set, we check if all the positions indicated by the hash functions are 1. If they are, we say that the item might be in the set. If any of them is 0, we can definitively say that the item is not in the set. A false positive can occur in a Bloom filter, where it might indicate that an item is in the set when it is not. This happens when the bits for an item that was not inserted happen to have been set to 1 by the insertion of other items.

A matrix BF is a two-dimensional extension of the standard BF, where each row is treated as a separate Bloom filter with parameters (m_1, n_1, k_1), and each column is also treated as a Bloom filter with parameters (m_2, n_2, k_2) [33]. This allows for more complex and flexible querying capabilities.

In this paper, we will use the matrix BF as a foundation to construct our index for bi-attribute membership tests. Table 4 lists the notations that are frequently used in this paper.

Table 4: Notations used in this paper

Symbol	Description
$\mathbb{S}$	the set of bi-attribute items
$\mathbb{S}_1$	the set of attribute 1
$\mathbb{S}_2$	the set of attribute 2
n	number of items
n_1	number of members on attribute 1
n_2	number of members on attribute 2
k	number of hashes
k_1	number of hashes on attribute 1
k_2	number of hashes on attribute 2
m_0	overall bits of matrix BF
m_1	row bits of matrix BF
m_2	column bits of matrix BF
f	false positive rate

4.1 The Basic Structure

To insert a piece of bi-attribute data (x, y) into the matrix BF, each attribute is hashed separately. The first attribute, x, is hashed using k_1 different hash functions, each generating a row index. Similarly, the second attribute, y, is hashed using k_2 different hash functions, each yielding a column index. The intersection of these hashed rows and columns identifies the positions in the matrix that are set to 1.

When querying whether a piece of bi-attribute data (x, y) exists in the set, we follow a similar process. We hash x and y to obtain k_1 and k_2 indices, respectively. We then check the corresponding $k_1 \times k_2$ positions in the matrix. If all of these positions are set to 1, we infer that the queried data (x, y) is likely in the set and return "true". If any of these positions is not set to 1, we conclude that the queried data (x, y) is definitely not in the set and return "false".

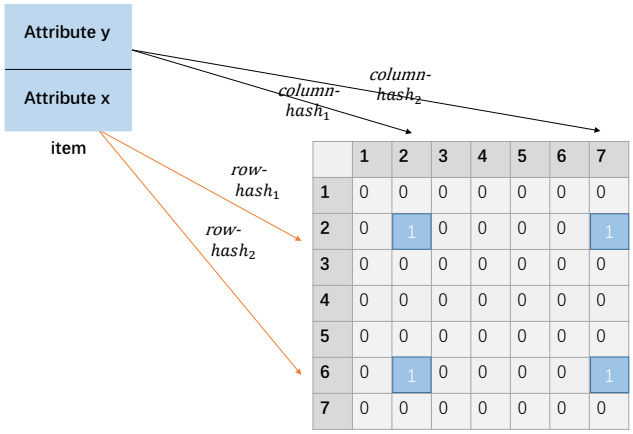


Figure 2: The basic structure of a matrix BF index.

Algorithm 1 Bi-attribute membership test in a matrix BF

Require: Item (x, y), matrix BF M;
1: *ItemInsertion*
2: **for** $i = 1; i < k_1; i++$ **do**
3: **for** $j = 1; j < k_2; j++$ **do**
4: $row\_array[i] \leftarrow rowhash_i(x)$;
5: $column\_array[j] \leftarrow columnhash_j(y)$;
6: $M(row\_array[i], column\_array[j]) \leftarrow 1$;
7: **end for**
8: **end for**
9: *ItemLookup*
10: **for** $i = 1; i < k_1; i++$ **do**
11: **for** $j = 1; j < k_2; j++$ **do**
12: $row\_array[i] \leftarrow rowhash_i(x)$
13: $column\_array[j] \leftarrow columnhash_j(y)$
14: **if** $M(row\_array[i], column\_array[j]) == 0$ **then return** false
15: **end if**
16: **end for**
17: **end for**
18: **return** true

The bi-attribute membership test in a matrix BF is described in Algorithm 1. For item insertion, it starts with two nested loops, where the outer loop iterates from 1 to k_1 and the inner loop iterates from 1 to k_2 (lines 2-3). For each pair of indices (i, j), calculates the ith row hash of x and assigns it to the ith element of *row_array*. It does a similar operation for the column: calculates the jth column hash of y and assigns it to the jth element of *column_array* (lines 4-5). For each pair of indices (i, j), sets the value at the ith row and jth column of the matrix M to 1 (line 6).

For item lookup, it also starts with two nested loops, where the outer loop iterates from 1 to k_1 and the inner loop iterates from 1 to k_2 (lines 10-11). For each pair of indices (i, j), calculates the ith row hash of x and the jth column hash of y, assigning them to *row_array* and *column_array* respectively, just like in the *ItemInsertion* process (lines 12-13). If the value at the ith row and jth column of the matrix M is 0, immediately returns false, stopping the *ItemLookup* process (line 14). If the algorithm has not returned false in the *ItemLookup* process, it returns true, signifying that the item (x, y) is present in the matrix M (line 18).

In this way, the matrix BF performs membership tests on two attributes in a partitioned manner. Different sets of hash functions are applied to each attribute, allowing the structure to maintain the co-existence of the two attributes within each data pair.

4.2 False Positive Rate

A false positive occurs when the matrix BF indicates that a queried item is part of the set when it is, in fact, not. We can calculate the probability of this happening, denoted as f:

Theorem 4.1. *The probability of a false positive in a matrix BF is given by* $(1 - e^{\frac{-nk_1k_2}{m_1m_2}})^{k_1k_2}$.

Proof. Suppose the hashes choose the positions equiprobably. Given an item that is **not** in the membership set,[2] for any bit the probability that it is not set to 1 in a single hash insertion turns out to be:

$$p_1 = 1 - \frac{1}{m_1 m_2} \tag{3}$$

After all the k_1k_2 executions the probability that the bit is not set to 1 is $p_2 = (1 - \frac{1}{m_1m_2})^{k_1k_2}$. When n items are inserted, the probability that the bit is still 0 is $p_3 = (1 - \frac{1}{m_1m_2})^{nk_1k_2}$. Hence, the probability that the bit is set to 1 is $p_4 = 1 - (1 - \frac{1}{m_1m_2})^{nk_1k_2}$. The false positive result occurs when all the queried k_1k_2 bits are set to 1:

$$f = (1 - (1 - \frac{1}{m_1 m_2})^{nk_1k_2})^{k_1k_2} \approx (1 - e^{\frac{-nk_1k_2}{m_1m_2}})^{k_1k_2}$$

□

Next, let us decide the optimal number of hash functions and the lowest false positive rate of a matrix BF:

Theorem 4.2. *The optimal condition of a matrix BF can be gained from substituting* $k = k_1k_2$ *and* $m = m_1m_2$ *into the formula* $k = \frac{m}{n} ln2$, *which is the theoretical optimal condition of a standard BF.*

Proof. Recall that

$$f = e^{k_1k_2 ln(1-e^{\frac{-nk_1k_2}{m_1m_2}})}$$

Let $p = e^{\frac{-nk_1k_2}{m_1m_2}}$, $g = k_1k_2 ln(1 - e^{\frac{-nk_1k_2}{m_1m_2}}) = \frac{-m_1m_2}{n} lnpln(1-p)$. Clearly, f arrives at its minimum when g reaches its minimum. Due to the symmetry of lnp and $ln(1-p)$, the following restriction holds:

$$p = e^{\frac{-nk_1k_2}{m_1m_2}} = \frac{1}{2} \tag{4}$$

[2]If we randomly choose an element from the entire space, and when the domain is large enough, the probability that it falls into a predefined membership set tends to be zero.

It implies f arrives at its minimum when overall 50% bits are occupied with 1. Hence, we have the boundary condition:

$$k_1 k_2 = \frac{m_1 m_2}{n} ln2$$

As well as the minimal value of f:

$$p_{min} = (\frac{1}{2})^{k_1 k_2} \tag{5}$$

Compare with the corresponding results of the standard BF:

$$p = \frac{1}{2}, k = \frac{m}{n} ln2, p_{min} = (\frac{1}{2})^k \tag{6}$$

Replace k and m in equation 6 by $k_1 k_2$ and $m_1 m_2$, then complete the proof. □

4.3 Partitioned Hashing Strategies

The structure of the matrix BF allows us to apply different hashing functions over different attributes. Specifically, we can use general hash functions for the trapdoor insertion/query for normal requests, while employing customized hash functions on the non-sensitive attribute to meet specific requirements. We illustrate this by using our index approach to partition the two attributes and then applying existing efficient hashing technologies (e.g., LSH) to improve performance. We demonstrate this with a real-world application by answering the timestamped membership query (tmt-query) problem posed by Peng et al.[30]: "Given an element in the form of (event, timestamp), can we determine whether any event occurred within a time range (t_1, t_2)?" In our setup, the index supports searchable encryption over the sensitive "event" attribute and range searching over the non-sensitive "timestamp" attribute. Thus, trapdoors are generated only for events, while timestamps remain in plaintext.

Even though we now need only one trapdoor generation for multiple timestamps, traversing the entire history would still require a brute-force linear search. This is still expensive as the scale increases. However, we can further improve this by using LSH, as introduced in the preliminaries section, for searching over the timestamps. Figure 3 illustrates the framework of the case study. Our basic idea is to use a matrix BF index as a preprocessing step to reduce the search range over timestamps by utilizing the locality provided by LSH. Specifically, we set some probing points uniformly on the time domain, corresponding to a series of time blocks that completely cover the entire domain. The size of the time blocks is related to the locality of LSH. Therefore, any item membership test (on each probing point) is essentially a neighbourhood search to identify the time block that the timestamp should belong to. For example, in Figure 3, the query on time block 1 will return false, which means the timestamp does not fall into this time block. We then move to the next probing point, corresponding to time block 2, which returns true and should contain the objective timestamp. Finally, to improve the accuracy, we use a global BF that records the element (event, timestamp) as a whole to pinpoint the objective timestamp within the time block. The entire process is described in Algorithm 2.

Algorithm 2 Range searching with LSH

Require: Timestamp t_s, matrix BF index M, global BF B;

```
1:  FindBlock : (t_s, M, B)
2:  for j = 1; j < k_2; j++ do
3:      column_array[j] ← columnhash_j(y)
4:      for t = 1; t in TimeBlock detect set; t++ do
5:          for i = 1; i < k_1; i++ do
6:              row_array[i] ← rowhash_i(t)
7:              if M(row_array[i], column_array[j]) = 0 then
    Break
8:              end if
9:          end for
10:         Goto GlobalBF : (t, y, B)
11:     end for
12: end for
13: GlobalBF : (t, y, B)
14: for s = 1; s < k_3; s++ do
15:     location_[j] ← columnhash_s(t + y)
16:     if B(location_[j]) == 1 then return false
17:     end if
18: end for
19: return true
```

4.4 Securing the index

4.4.1 Index Initialization. In light of Definition 2, the security requirement is that the release of any set of outcomes related to the non-sensitive attribute should contribute only a bounded privacy loss of the sensitive one. This essentially relates to the distributional similarity of items over the non-sensitive attribute. One extreme case would be if every pair of sensitive items had the same distribution of co-occurring non-sensitive values, meaning that the outcome of any non-sensitive value set contains no information about the sensitive item, even though they may be correlated.

In the context of the matrix BF, for any specified sensitive item, the set of co-existing non-sensitive items has a BF representation, which includes all the matrix BF columns that the sensitive item is hashed to. If the universe of all possible BF representations can be ensured by equation 1, satisfying the definition of LDP, this implies that any possible membership test subset also maintains the same level of privacy. Consequently, we can assert the following:

Corollary 4.3. *If the set of BF representations $\mathbb{B}_1, \mathbb{B}_2, ..., \mathbb{B}_i$ for all non-sensitive items complies with LDP, then any bi-attribute batch membership test over the non-sensitive attribute will have a privacy loss that is bounded (controlled by ϵ).*

As p approaches 0.5, it implies that ϵ approaches 0, leading to an infinite perturbation that makes any inputs indistinguishable. In this scenario, the BF representation returns "yes" with a probability of 50% for any item and hence contains no information about the set of inserted values (and thereby the sensitive attribute).

The initialization of a matrix BF index is described in Algorithm 3. Each item from the membership set I into the matrix M (line 2), and the algorithm starts two nested loops. The outer loop iterates over all the indices i where the sum of the ith row in the array is not zero. The inner loop iterates over all the indices j (line 3-4). For each pair of indices (i, j), the algorithm stores the value of the

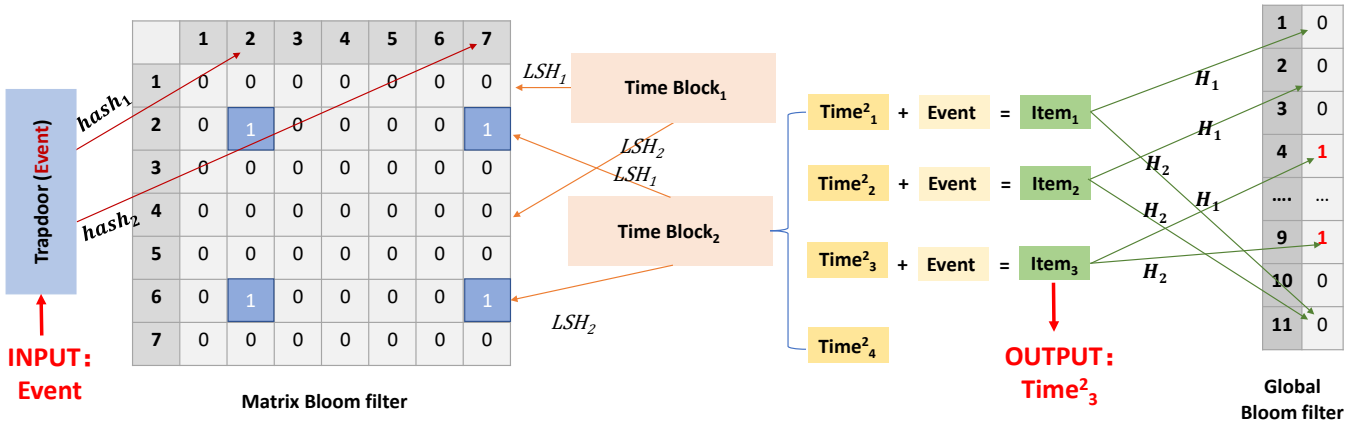


Figure 3: The framework of our study case. Timestamps are queried with LSH in the matrix BF first and are further checked in a global BF.

cell at the ith row and jth column of the matrix M in a temporary variable $Temp$ (line 5). Finally, the algorithm generates a random number between 0 and 1. If this number is less than the perturbation parameter p, the value of the cell at the ith row and jth column of the matrix M remains the same. Otherwise, the value of this cell is switched to its complement, i.e., 1 becomes 0 and vice versa (line 6-9).

The user determines the privacy budget ϵ to choose (which corresponds to the perturbation parameter p), and the initialized matrix BF will have the lowest false positive rate f_1 as the **privacy bound**. The user also determines the highest acceptable false positive rate f_2 as the **utility bound**. Thus, the index starts at a specific state and is filled up as the user inserts new items. As the BF becomes full (or full enough to start producing many false positives), the false positive rate increases and the utility of the BF tends to worsen. Therefore, the BF is expected to operate effectively between f_1 and f_2. The user should reconstruct a new index when the false positive rate exceeds f_2.

Algorithm 3 Index initialization with RR

Require: Membership item set I, Perturbation parameter p, matrix BF M;

1: $Insert(item \in I) \rightarrow M$
2: **for** All i when $\sum row\_array[i]! = 0$ **do**
3: **for** All j **do**
4: $Temp = M(row\_array[i], column\_array[j])$
5: **if** $rand(1) < p$ **then**
6: $M(row\_array[i], column\_array[j]) = Temp$
7: **else**
8: $M(row\_array[i], column\_array[j]) = 1 - Temp$
9: **end if**
10: **end for**
11: **end for**

4.4.2 Index Update. Finally, we turn to the issue of how to handle the deletion or modification of an item that already exists in the index. Traditional BFs do not support deletions, except for variants such as counting Bloom filters [15]. However, even counting Bloom filters do not completely solve this problem, as they can lead to false negatives after deletion [19]. Specifically, when a counter overflows, meaning that the actual number of recorded items exceeds the counter's maximum value, non-zero counters may reduce to zero earlier than expected during the deletion of items. This can result in the invalidation of correct items.

In our framework, we do not need to delete an existing item from the index. The perturbation introduced during index initialization is equivalent to the insertion of fake items. Therefore, we can leave items that are intended to be deleted in the index as if they were inserted during the index initialization as fake items. There is no need, and indeed no way, to distinguish between the two scenarios. The revision of an item, which involves deleting an old item and inserting a new one, is then reduced to a simple item insertion.

Capability. We can now derive the capacity of an initialized index, i.e., the number of items that can be added. Once the values of m and k for the Bloom filter are fixed, the following formula relating n (the number of items in the filter) and f (the false positive rate) holds:

$$n = -\frac{m}{k}\ln(1 - e^{\frac{\ln f}{k}})$$

We can then determine the number of items that can be added upon initialization as follows:

$$N = n_2 - n_1 = -\frac{m}{k}\ln\frac{1 - e^{\frac{\ln f_2}{k}}}{1 - e^{\frac{\ln f_1}{k}}}$$

In this formula, n_2 and n_1 are the numbers of items in the filter when the false positive rates are f_2 and f_1, respectively. Therefore, the expression for N gives the difference in the number of items in the filter as the false positive rate increases from f_1 to f_2, which corresponds to the number of items that can be added to the filter after initialization.

5 HANDLING INTER-ATTRIBUTE CORRELATIONS

So far, we have discussed the matrix BF as if distributions over the two attributes are independent, while this is inconsistent to our problem setting where inter-attribute correlations are involved. In practice, we should manage the size and shape of the matrix to fit the features that we may known beforehead about the dataset.

This section proposes two variants of matrix BF to apply prior knowledge into the matrix BF. The maximum adaptive approach, as the name suggests, is designed to be universally applicable. In contrast, the minimum storage approach leverages on strong prior knowledge and precise information about the repetition of each key in the dataset, which must be known beforehand. Most situations in practice fall somewhere between these two extremes. A classic example is a dictionary (or a subset of it), where we know how many capital letters there are, but we only know the approximate frequency of words starting with each letter. In the minimum storage approach, we deploy a special hash to classify the repetition property, and if the facts don't match the prior knowledge, this classification may result in more elements in some sub-matrices than expected. In practice, we can preserve some redundancy during the initialization phase to accommodate situations where the prior knowledge is insufficient or is inconsistent to the facts.

5.1 Maximum Adaptive Matrix

The number of items on the two attributes and their combination patterns co-determine the shape and size of the matrix BF. We should consider the worst case when having no prior knowledge about the dataset, or the index should be frequently renewed and updated by adding unknown items. That is, for any given n_1, n_2, we pre-allocate large enough spaces for any given $\mathbb{S}$ with parameters n_1, n_2, where the maximum possible combinations is $n_1 n_2$. To be adaptive to the most general case, given m_1 and m_2, treat the rows and columns as dedicated standard BFs, both of which hold the lowest possible false positive rate. Hence, we have

$$k_1 = \frac{m_1}{n_1} ln2, k_2 = \frac{m_2}{n_2} ln2$$

Load factor. The maximum adaptive matrix is relatively empty, see Fig. 4 as a sample. About 50% of rows/columns is set to 0, hence the load factor is approximately 25%. In the worst case where $n = n_1 n_2$, since the queries in row and columns satisfy equation 4, the load factor turns out to be approximately $50\% \times 50\% = 25\%$. In general cases, n is in fact less than $n_1 n_2$, thus the load factor is always less than 25%.

False positive rate. When queries in both rows and columns turn out to be false positive, the overall false positive result occurs. Hence,

$$fpr_{mam} = f_1 \times f_2 = (\frac{1}{2})^{k_1+k_2}$$

Storage overhead. As mentioned, the maximum adaptive matrix is relatively empty. It achieves a better performance of adaptive batch queries at the expense of storage overhead. Let us derive the maximum possible storage cost of the maximum adaptive matrix, and compare it with a standard BF.

Assume a standard BF is allocated to insert the same set of $n_1 n_2$ elements, where the same false positive rate $fpr_{mam} = (\frac{1}{2})^{k_1+k_2}$ is achieved. Let the size of the standard BF be m_0, hence, the corresponding number of hash functions in the standard BF turns out to be $k_0 = \frac{m_0}{n_1 n_2} ln2$. Let $k_0 = k_1 + k_2$, we have:

$$(\frac{m_1}{n_1} + \frac{m_2}{n_2}) ln2 = (\frac{n_2 m_1 + n_1 m_2}{n_1 n_2}) ln2 = (\frac{m_0}{n_1 n_2}) ln2$$

That is,

$$n_1 m_2 + n_2 m_1 = m_0$$

Due to the mean value inequality, we have:

$$2\sqrt{n_1 n_2} \cdot \sqrt{m_1 m_2} \leq m_0$$

Hence,

$$m = m_1 m_2 \leq \frac{(m_0)^2}{4 n_1 n_2} \tag{7}$$

Complexity. For each membership test, there are overall $k_1 k_2$ hashing/comparisons, therefore the complexity is simply $O(k_1 k_2)$. Since

$$k_1 k_2 = (\frac{m_1 m_2}{n_1 n_2}) ln^2 2$$

Substitute by equation 7, we have:

$$k_1 k_2 = (\frac{m_1 m_2}{n_1 n_2}) ln^2 2 \leq \frac{(m_0)^2}{4(n_1 n_2)^2} ln^2 2 = \frac{k_0^2}{4}$$

However, in the case of batch membership tests, the complexity can be further reduced. Consider the best case of key-value pairs, where a series of varying values correspond to the same key. For this special dataset, the key is to be hashed only once, while the values are to be hashed at most $max(k_1, k_2)$ times. Therefore, the best complexity can be lowered to $O(max(k_1, k_2))$.

0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	1	1	0	0	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	1	1	0	0	1	0
1	0	0	1	0	0	0	0	1	1	0	0	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	1	1	0	0	1	0

Figure 4: A sample of the maximum adaptive matrix. The rows/columns in white are entirely set to 0.

5.2 Minimum Storage Matrix

While the maximum adaptive matrix is adaptive to the most general case without prior knowledge, it bears an extensive overhead on the storage cost and hashing numbers. However, in practice, we may be aware of some statistical features of the dataset, with which we can construct a lower storage matrix applying in read-only (or not frequently updated) scenarios, e.g., a dictionary. In this part, we will discuss some typical datasets with strong background knowledge and construct the corresponding matrices according to our theory.

Case A. Let us start at a special case where there exists a bijection between $\mathbb{S}_1$ and $\mathbb{S}_2$, i.e., for any two different elements, there is no repeat on both of the two components. Clearly, in this case

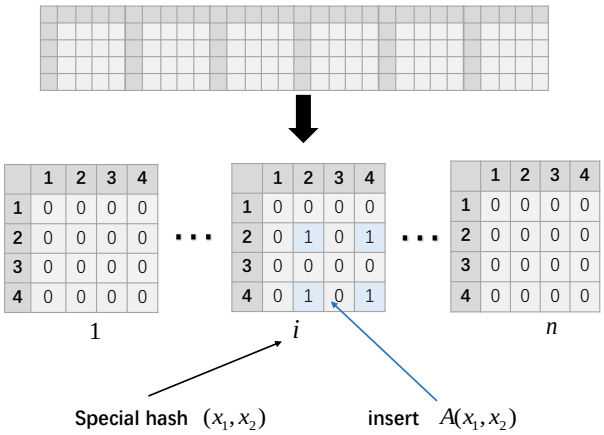


Figure 5: Element insertion/query in a j-matrix.

$n_1 = n_2 = n$, which means the two sets of different components contain the same number of elements. In this special case, the answer is simple. In this case, positions of (k_1, m_1) and (k_2, m_2) are symmetric, since both (k_1, k_2) and (m_1, m_2) are commutable. Naturally, we employ a square matrix for insertion of the n elements, where $k_1 = k_2 = k, m_1 = m_2 = m$.

Case B. In this case, let us consider a more general situation where elements in $\mathbb{S}$ can be represented as a weak combination from $\mathbb{S}_1$ and $\mathbb{S}_2$. For the convenience of discussion, let $n_1 > n_2$. In this case, each element in $\mathbb{S}_2$ combines with elements in $\mathbb{S}_1$ j times. Elements in $\mathbb{S}_1$ are not allowed to repeat. Hence, $n_1 = jn_2$.

Element insertion/query. Let us start at a special case where $n_1 = 2n_2$. We adopt a special hash to classify components in $\mathbb{S}_1$ into 2 types, each belonging to either $\mathbb{S}_{11}$ or $\mathbb{S}_{12}$. In each set, there are $n_2 = \frac{n_1}{2}$ elements. For each set, we employ a square matrix as described in the context of Case A.

For any element to insert, we first employ the special hash to find out which square matrix it belongs to. Then, in the determined square matrix, find out the corresponding rows for components in $\mathbb{S}_1$ as well as columns for components in $\mathbb{S}_2$.

For the two square matrices, components in $\mathbb{S}_2$ are always mapped into the same columns. Hence, it equals merging the two matrices from left to right to get a $2m_2 \times m_2$ matrix. For the general case, when $n_1 = jn_2$, similarly, the special hash should choose which of the j sets an element belongs to. Then for each set, a square matrix is adopted, and the j square matrices are stuck together to build a $jm \times m$ j-matrix. See Fig. 5 as an example. When any elements are queried, similar rules are executed to find out if the k_1k_2 bits are 1.

False positive rate. Let us discuss the special case where $n_1 = 2n_2$ first. Suppose overall $2m^2$ bits are employed to build this matrix. Now let us prove the false positive rate in this scenario equals using a standard BF, where the same number of bits $(2m^2)$ is allocated to insert the n_1 elements.

Obviously, the false positive rate of that standard BF is $(\frac{1}{2})^{\frac{2m^2}{n_1}ln2}$. In the matrix, since the two square matrices share the same false positive rate, only one of them needs to be calculated. The false positive rate turns out to be:

$$fpr_{2-matrix} = (\frac{1}{2})^{\frac{m^2}{n_2}ln2} = (\frac{1}{2})^{\frac{2m^2}{n_1}ln2}$$

Extending to the j-matrix case, we have the following theorem:

Theorem 5.1. *When jm^2 bits are adopted to build a general $jm \times m$ j-matrix case, the false positive rate equals that of a standard BF where the same number of bits (jm^2) is used for insertion of the same number of elements. (Say, $n_1 = jn_2$)*

Proof. The false positive rate of that standard BF is $(\frac{1}{2})^{\frac{jm^2}{n_1}ln2}$. Since the j square matrices share the same false positive rate, just one of them needs to be considered. The false positive rate turns out to be:

$$fpr_{j-matrix} = (\frac{1}{2})^{\frac{m^2}{n_2}ln2} = (\frac{1}{2})^{\frac{jm^2}{n_1}ln2}$$

□

6 EXPERIMENTS

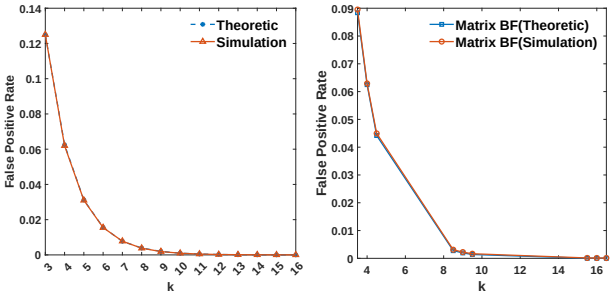
The experiments are conducted using MATLAB R2021a on a PC equipped with an Intel i7-10700K RTX 3090 eight-core processor, 128GB RAM, and Windows 10 OS. To implement the BFs, we need several different hash functions. For this, we opt for universal hash functions [5] to map elements into BFs. For any item X represented as $X =< x_1, x_2, ..., x_b >$ in b-bits, the ith hash function over X $h_i(x)$ is calculated as $h_i(x) = (d_{i1} \times x_1) \oplus (d_i 2 \times x_2) \oplus ... \oplus (d_i 3 \times x_3)$, where $\times$' signifies the bitwise AND operator and $\oplus$' represents the bitwise XOR operator. The values of d_i are predetermined random numbers that share the same range as the hash function.

6.1 False Positive Rate

We use two real-world datasets from the Bag of Words database, which consists of five text collections. We chose **KOS** (with 353160 key-value pairs from 3430 keys and 5851 values) and **NIPS** (with 746316 key-value pairs from 1500 keys and 12375 values). Moreover, we create a synthetic bi-attribute dataset, **SYN_FD**, which is fully-duplicated. This dataset is formed from the Euclidean cross product of two scalar datasets where elements are random numbers, which gives the bi-attribute data a duplication feature on both attributes. Both scalar datasets are fixed with 1000 numbers, resulting in a total of 1000000 bi-attribute data in the dataset.

Verification of Theorem 4.1. We use **SYN_FD** for this part. For the standard BF, we insert a piece of bi-attribute data as a whole. Let k be different integers, and allocate an appropriate value of m using the formula $k = \frac{m}{n}ln2$. For the matrix BF, we allocate the same number of bits and items where $k = k_1k_2$, as a comparison to the standard BF. The insertion rules of a single row/column for the matrix BF are the same as for a standard BF.

The left-hand part of Fig.6 shows the tendency of false positive rates of the standard BF varying with respect to k, and the right-hand part is that of the matrix BF. Notably, the points in the matrix BF are more concentrated because m_1 and m_2 are commutable, which results in a square matrix BF. However, since $k = k_1k_2$, if both k_1 and k_2 are required to be strict integers, there are fewer choices for k. Therefore, we approximate some points near $k = 8$ and $k = 16$ where the number of hash functions is fixed to integers,



(a) Standard BF. (b) Matrix BF.

Figure 6: Comparison of theoretical/experimental value of the optimal false positive rate in a standard/matrix BF.

while the corresponding m_1 and m_2 are calculated from the non-integer values of k. As seen from Fig.6, the experimental results align well with the theoretical values. Thus, Theorem 4.1 is verified, which implies the performance of our matrix BF is equivalent to the standard BF.

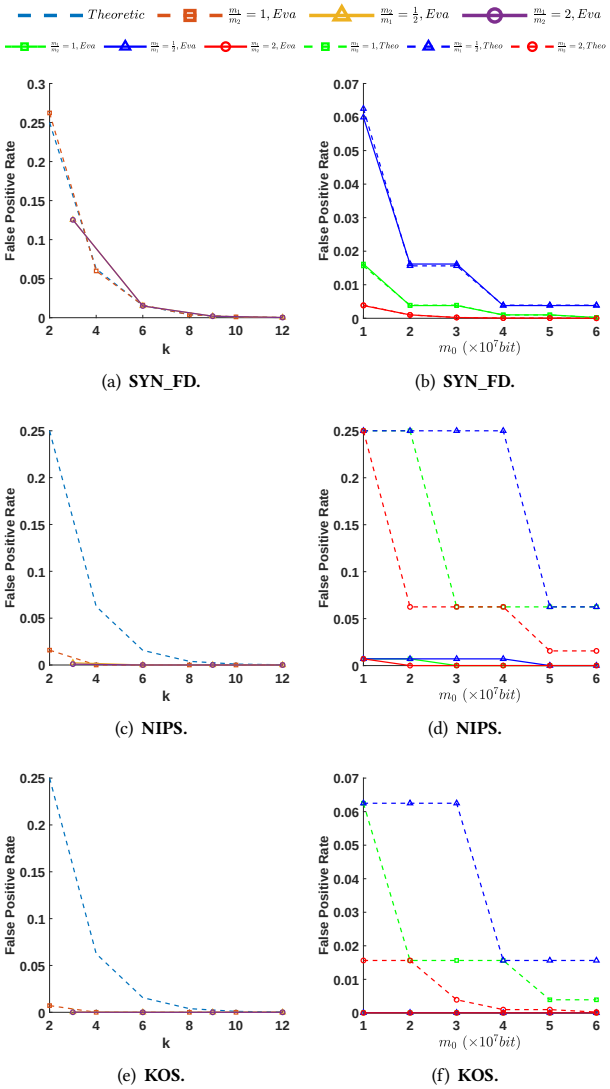
On maximum adaptive matrix. The performance of the maximum adaptive matrix is evaluated under various parameter settings on the aforementioned three datasets. We allow $k = k_1 + k_2$ to take on different integer values. The parameters m_i are adjusted to appropriate values using the formula $k_i = \frac{m_i}{n_i} ln2, i = 1, 2$, respectively. The results are illustrated in Figs. 7 (a) (c) (e), which indicate that the evaluated false positive rate (fpr) aligns with the theoretical value when the matrix is fully populated, as evaluated on **SYN_FD**. However, as the two real-world datasets are not as duplicated as the synthetic one, the matrices are less loaded, leading to a lower false positive rate.

Figs. 7 (b) (d) (f) assess the false positives of different proportions of $\frac{m_1}{m_2}$ as m_0 varies. Similarly, the evaluated values are well-aligned with the theoretical ones when the matrix is fully populated, i.e., as evaluated on **SYN_FD**, and are considerably lower for real-world datasets. This suggests that the maximum adaptive matrix requires a significant amount of storage for typical cases, a trade-off for the generality it offers.

In our final set of experiments, we examine the load factor of our matrix with respect to the proportion of inserted elements. The parameters selected for this are identical to those in previous experiments. Table 5 illustrates that the load factors for different values of $\frac{m_1}{m_2}$ are nearly identical when the same proportion of elements is inserted. In particular, when the matrix is fully loaded, the load factor is very close to the predicted 25%.

On minimum storage matrix. This part evaluates the robustness and storage efficiency of the minimum storage matrix. We set $k = \frac{k_1}{j} = k_2$ and $n_1 = jn_2$ with $n_2 = 144$, and allow k to take different integer values. The value of m_2 is adjusted accordingly, based on the formula $k^2 = \frac{m_2^2}{n_2} ln2$.

We plot the false positive rates versus k^2 for $j = 2, 10, 40, 100$ on the left part of Fig. 8. To make an approximation, we take some points nearby non-integer k^2 values while fixing the number of hash functions to integers. The corresponding number of bits is then



(a) **SYN_FD.** (b) **SYN_FD.**

(c) **NIPS.** (d) **NIPS.**

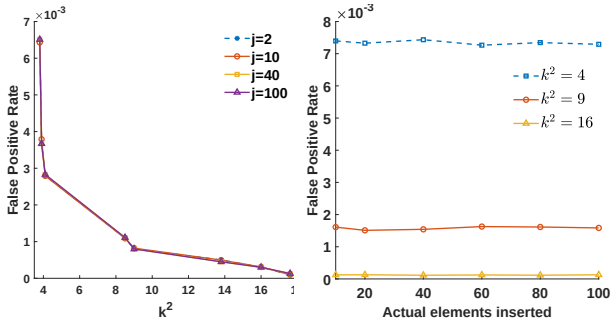
(e) **KOS.** (f) **KOS.**

Figure 7: False positive rate of maximum adaptive matrix with respect to k and m_0.

Table 5: Load factor & proportion of elements of different $\frac{m_1}{m_2}$'s

Load factor / Proportion	$\frac{1}{4}$	$\frac{1}{2}$	1
20%	0.063681451	0.065117817	0.065625635
40%	0.121001445	0.120029788	0.122250358
60%	0.170028163	0.167312289	0.175074846
80%	0.213055668	0.210467359	0.217504935
100%	0.249106352	0.24721041	0.256099335

calculated from the non-integer values of k. Our results indicate that



(a) fpr with respect to k^2. (b) Robustness.

Figure 8: The false positive rate and robustness of minimum storage matrix.

Table 6: Storage Comparison

Method		Key	Value	fpr	k_1	k_2	$m_0(MB)$
Maximum adaptive matrix		10^4	10^6	$\frac{1}{2}^4$	2	2	2.9×10^4
				$\frac{1}{2}^{12}$	6	6	8.9×10^4
Minimum storage matrix	j=50			$\frac{1}{2}^4$	2	2	34.4
	j=50			$\frac{1}{2}^{12}$	4	4	103.19
	j=100			$\frac{1}{2}^4$	2	2	68.79

all four j-matrices perform similarly and remain almost unchanged regardless of the value of j.

Next, we fix k^2 at $4, 9, 16$ and plot the false positive rates with respect to j. As illustrated in the right-hand part of Fig. 8, the false positive rate remains stable as j increases, indicating the high robustness of our proposed structure.

Finally, we compare the storage cost for initializing a minimum storage matrix and a maximum adaptive matrix at the same theoretical false positive rate, as shown in Table 6. The results demonstrate that when the dataset duplication level is low, the storage can be reduced by several orders of magnitude if we have sufficient prior knowledge about the datasets. This highlights the potential for significant storage efficiency gains with the minimum storage matrix, particularly in read-only or infrequently updated scenarios where dataset characteristics can be accurately predicted.

6.2 Batch Performance

In this part, we assess the search time performance of our proposed structure. We utilize the following datasets:

(1) **Solar:** Extracted from the Solar Power Data for Integration Studies, this dataset consists of 1-year worth of solar power data and hourly day-ahead forecasts for around 6000 simulated PV plants in Alabama. We use a subset of this data for our experiment, treating solar power as events and time as values. The dataset consists of 62 events and 287 values (in the form of hour and minute), resulting in 105120 combinations, many of which are duplicated.
(2) **Electricity:** This dataset contains electricity consumption data from 321 clients, recorded every 15 minutes from 2012 to 2014. Here, electricity consumption is treated as the event, and the time is considered the value. The dataset has 32 events and 95 values (in the form of hour and minute), leading to 105120 combinations with numerous duplications.
(3) **SYN_ND:** This is a synthetic, non-duplicated dataset generated from 20 events and 3599 values. Each pair from the two scalar sets is unique, resulting in 71980 non-duplicated pairs. The form of the value is (minute, second).

We evaluate the performance by comparing the total hashing generation times concerning different w values, the locality of the Locality-Sensitive Hashing (LSH) used, and the trapdoor generation times. The results are listed in Table 7. From these results, it is evident that LSH substantially reduces the hashing time for timestamp searching, with a reasonable cost of trapdoor generation.

The LSH comparison time decreases as w increases for two reasons. First, LSH allows nearby data to be checked only once, thus reducing redundant operations. Second, as the locality w increases, each time block contains more data, reducing the overall time required for comparisons.

Our synthetic data yields better results in terms of reducing search times because it is non-duplicated. Real datasets contain some non-existing and duplicated values, which slightly degrade the performance. However, the results are still significantly better than using a brute force approach. In summary, an LSH-based index can effectively decrease the hashing times during brute force searches over timestamps.

6.3 Privacy Guarantee

This part evaluates the privacy guarantees provided by the initialized matrix BF index. We configure the matrix size to 5000×5000 and insert some bi-attribute data, including a piece of target data for which we possess auxiliary knowledge to infer. We then perturb the matrix BF using RR for all non-zero rows. If more than one data record is discovered within the time range valid for the auxiliary knowledge, we deem the perturbation to have successfully disproves the auxiliary knowledge.

Fig. 9 presents the disprove rates. When $p = 1$, no perturbation is applied. N_s in the left-hand part denotes the number of inserted sensitive attributes, with the hash number fixed at 4. The disprove rate for $p = 1$ originates from the inherent false positives of the BF, which is referred to as the "better than nothing" privacy. This rate increases with the number of inserted items. Regardless of the initial state, all rates quickly rise and converge to 1 in relation to $1 - p$, taking advantage of the RR perturbation mechanism.

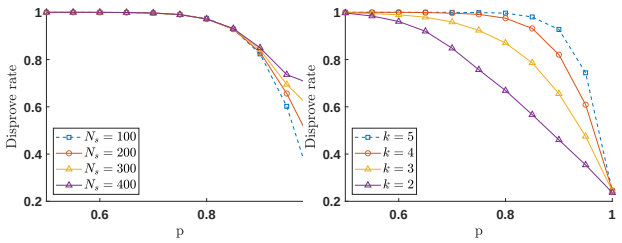
The right-hand part of the figure keeps the number of items constant while varying the hash numbers, which influences the slopes of the curves. A higher hash number yields more privacy for the same p value. This demonstrates the ability of our approach to provide privacy guarantees, validating its effectiveness in preserving the privacy of sensitive attributes.

7 RELATED WORK

The Bloom filter has emerged as a pivotal technology in the realm of searchable encryption, facilitating the search of sensitive data in ciphertext. The inaugural endeavor in this field, known as secure indexes [18], has since been followed by further studies [2, 29].

Table 7: Times of searching and trapdoor generation.

Data-Set	Comparsion methods	Comparsion times									
		w=1	w=2	w=3	w=4	w=5	w=6	w=7	w=8	w=9	w=10
Solar	Brute force	149.67									
	LSH	149.67	127.82	119.71	111.31	106.27	99.59	92.03	84.50	82.20	75.05
Electricity	Brute force	52.50									
	LSH	52.50	45.32	43.07	41.11	38.71	37.99	35.12	32.99	31.31	27.56
SYN	Brute force	1860.5									
	LSH	1860.5	838.17	615.10	474.73	342.2	299.5	237.85	205.25	190.85	181.57
Trapdoor		0	1	1.5	2	2.5	3	3.5	4	4.5	5



(a) Number of sensitive attributes. (b) Number of hashes.

Figure 9: The disprove rate of auxiliary knowledge.

Despite their advances, these methodologies fail to distinguish between sensitive and non-sensitive data. Moreover, they do not facilitate the search of non-sensitive data in plaintext, whilst simultaneously offering secure solutions for sensitive counterparts. There have been several pragmatic attempts to overcome these limitations, with a prominent example being the hybrid cloud approach [27, 28], which solely outsources non-sensitive data. Furthermore, the work of Mehrotra et al. probes into partitioned data security for outsourced data, dividing attributes into sensitive and non-sensitive groups [25]. Although these works introduce partitioned data security and explore deterministic methods, our research concentrates on establishing an efficient probabilistic structure.

The privacy guarantee LDP [8, 11, 23] employed in our paper is initially proposed for protecting data privacy in an untrusted environment, where each user perturbs her values before sending them to the data collector. Some typical scenarios for applications of LDP include frequency estimation [1], mean estimation [11], and high-dimensional data collections [10]. Besides, LDP has been widely applied to real-life applications, such as Chorme [14], iOS [31], Win 10 [9], and Samsung [26].

There have also been previous efforts to extend the Bloom filter into a matrix form. Geravand et al. proposed a bit matrix to detect copy-paste content in a literature library [17]. Wang et al. introduced a Bloom filter in matrix form, representing a new type of Bloom filter [32]. It employs s rows of Bloom filters with k hashes and an additional special hash to decide which row of the Bloom filter to insert into. The multi-set problem also aligns closely with the bi-attribute membership tests we study. Yu et al. proposed vBF [36] within the framework of BF, creating v BFs for v sets. For a query element a, vBF performs a series of queries across all the v BFs. If BF_i reports true, a is considered to be in set i, otherwise it is not. There are other BF-based solutions to the multi-set problem as well, such as the coded BF [6], the Bloomier filter [7], and kBF [35].

8 CONCLUSION

This paper presents a secure bi-attribute index, enabling batch operations over one attribute. Efficiency-wise, the proposed matrix BF structure partitions query processing over two attributes, one of which can be non-sensitive and processed in plaintext. The structure keeps the attribute co-existence and supports any existing efficient hashing technique to further enhance the performance under certain scenarios. Privacy-wise, the proposed initiation approach with RR protects the BF representation of the plaintext outcome and achieves a bound privacy loss due to the property of LDP.

REFERENCES

[1] Raef Bassily and Adam Smith. 2015. Local, private, efficient protocols for succinct histograms. In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*. 127–135.

[2] Steven Michael Bellovin and William R Cheswick. 2007. Privacy-enhanced searches using encrypted bloom filters. (2007).

[3] Giuseppe Bianchi, Lorenzo Bracciale, and Pierpaolo Loreti. 2012. âĂİ Better Than NothingâĂİ Privacy with Bloom Filters: To What Extent?. In *International Conference on Privacy in Statistical Databases*. Springer, 348–363.

[4] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

[5] J Lawrence Carter and Mark N Wegman. 1979. Universal classes of hash functions. *Journal of computer and system sciences* 18, 2 (1979), 143–154.

[6] Francis Chang, Wu-chang Feng, and Kang Li. 2004. Approximate caches for packet classification. In *IEEE INFOCOM 2004*, Vol. 4. IEEE, 2196–2207.

[7] Denis Charles and Kumar Chellapilla. 2008. Bloomier filters: A second look. In *European Symposium on Algorithms*. Springer, 259–270.

[8] Rui Chen, Haoran Li, A Kai Qin, Shiva Prasad Kasiviswanathan, and Hongxia Jin. 2016. Private spatial data aggregation in the local setting. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 289–300.

[9] Bolin Ding, Janardhan Kulkarni, and Sergey Yekhanin. 2017. Collecting telemetry data privately. *arXiv preprint arXiv:1712.01524* (2017).

[10] Rong Du, Qingqing Ye, Yue Fu, and Haibo Hu. 2021. Collecting high-dimensional and correlation-constrained data with local differential privacy. In *2021 18th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*. IEEE, 1–9.

[11] John C Duchi, Michael I Jordan, and Martin J Wainwright. 2013. Local privacy and statistical minimax rates. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*. IEEE, 429–438.

[12] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. 2006. Calibrating noise to sensitivity in private data analysis. In *Theory of cryptography conference*. Springer, 265–284.

[13] Fatih Emekci, Ahmed Methwally, Divyakant Agrawal, and Amr El Abbadi. 2014. Dividing secrets to secure data outsourcing. *Information Sciences* 263 (2014), 198–210.

[14] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. 2014. Rappor: Randomized aggregatable privacy-preserving ordinal response. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*. 1054–1067.

[15] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. 2000. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM transactions on*

networking 8, 3 (2000), 281–293.
[16] Craig Gentry. 2009. *A fully homomorphic encryption scheme*. Stanford university.
[17] Shahabeddin Geravand and Mahmood Ahmadi. 2011. A novel adjustable matrix bloom filter-based copy detection system for digital libraries. In *2011 IEEE 11th International Conference on Computer and Information Technology*. IEEE, 518–525.
[18] Eu-Jin Goh. 2003. Secure indexes. *Cryptology ePrint Archive* (2003).
[19] Deke Guo, Yunhao Liu, Xiangyang Li, and Panlong Yang. 2010. False negative problem of counting bloom filter. *IEEE transactions on knowledge and data engineering* 22, 5 (2010), 651–664.
[20] Deke Guo, Jie Wu, Honghui Chen, and Xueshan Luo. 2006. Theory and network applications of dynamic bloom filters. In *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*. IEEE, 1–12.
[21] Fang Hao, Murali Kodialam, TV Lakshman, and Haoyu Song. 2009. Fast multiset membership testing using combinatorial bloom filters. In *IEEE INFOCOM 2009*. IEEE, 513–521.
[22] Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. 604–613.
[23] Shiva Prasad Kasiviswanathan, Homin K Lee, Kobbi Nissim, Sofya Raskhodnikova, and Adam Smith. 2011. What can we learn privately? *SIAM J. Comput.* 40, 3 (2011), 793–826.
[24] Tiancheng Li and Ninghui Li. 2009. On the tradeoff between privacy and utility in data publishing. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. 517–526.
[25] Sharad Mehrotra, Shantanu Sharma, Jeffrey Ullman, and Anurag Mishra. 2019. Partitioned data security on outsourced sensitive and non-sensitive data. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 650–661.
[26] Thông T Nguyên, Xiaokui Xiao, Yin Yang, Siu Cheung Hui, Hyejin Shin, and Junbum Shin. 2016. Collecting and analyzing data from smart device users with local differential privacy. *arXiv preprint arXiv:1606.05053* (2016).
[27] Kerim Yasin Oktay, Murat Kantarcioglu, and Sharad Mehrotra. 2017. Secure and efficient query processing over hybrid clouds. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 733–744.
[28] Kerim Yasin Oktay, Sharad Mehrotra, Vaibhav Khadilkar, and Murat Kantarcioglu. 2015. SEMROD: secure and efficient MapReduce over hybrid clouds. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 153–166.
[29] Saibal K Pal, Puneet Sardana, and Ankita Sardana. 2014. Efficient search on encrypted data using bloom filter. In *2014 International Conference on Computing for Sustainable Global Development (INDIACom)*. IEEE, 412–416.
[30] Yanqing Peng, Jinwei Guo, Feifei Li, Weining Qian, and Aoying Zhou. 2018. Persistent bloom filter: Membership testing for the entire history. In *Proceedings of the 2018 International Conference on Management of Data*. 1037–1052.
[31] ADP Team et al. 2017. Learning with privacy at scale. *Apple Mach. Learn. J* 1, 8 (2017), 1–25.
[32] Jiacong Wang, Mingzhong Xiao, and Yafei Dai. 2007. MBF: A real matrix Bloom filter representation method on dynamic set. In *2007 IFIP International Conference on Network and Parallel Computing Workshops (NPC 2007)*. IEEE, 733–736.
[33] Zhu Wang, Tiejian Luo, Guandong Xu, and Xiang Wang. 2013. A new indexing technique for supporting by-attribute membership query of multidimensional data. In *International Conference on Web-Age Information Management*. Springer, 266–277.
[34] Stanley L Warner. 1965. Randomized response: A survey technique for eliminating evasive answer bias. *J. Amer. Statist. Assoc.* 60, 309 (1965), 63–69.
[35] Sisi Xiong, Yanjun Yao, Qing Cao, and Tian He. 2014. kBF: a bloom filter for key-value storage with an application on approximate state machines. In *IEEE INFOCOM 2014-IEEE Conference on Computer Communications*. IEEE, 1150–1158.
[36] Minlan Yu, Alex Fabrikant, and Jennifer Rexford. 2009. BUFFALO: Bloom filter forwarding architecture for large organizations. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies*. 313–324.