World Wide Web (2020) 23:2185–2214
https://doi.org/10.1007/s11280-019-00774-x

An enhanced wildcard-based fuzzy searching scheme in encrypted databases

Jiaxun Hua[1] · Yu Liu[1] · He Chen[1] · Xiuxia Tian[2] · Cheqing Jin[1]

Received: 13 February 2019 / Revised: 17 September 2019 / Accepted: 23 December 2019 /
Published online: 13 March 2020

Abstract
Under the overwhelming trend in Cloud Computing, Cloud Databases possessing high scalability / high availability / high parallel performance have become a prevalent paradigm of data outsourcing. In consideration of security and privacy, both individuals and enterprises prefer to outsource service data in encrypted form. Unfortunately, most encrypted databases cannot support such complicated queries as wildcard-based fuzzy searching, which, to some extent, limits the practicability in real applications. To explore more business logic in encrypted databases, an enhanced wildcard-based fuzzy searching scheme (*enWFS*) is proposed in this paper, which integrates specialized *Adjacent Character Matrix/Tensor* into proxy middleware, appends two types of ancillary columns into data tables, as well as designs an advanced adaptive overwriting method to revise query expressions with wildcards ('%' and '_'). Meanwhile, some security enhancements and *TupleRank* are added to *enWFS* scheme so as to achieve superior fuzzy searching experiences. Extensive experiments based on real datasets demonstrate effectiveness, feasibility of our proposal.

Keywords Encrypted database · Fuzzy searching · Wildcards

1 Introduction

Cloud computing, moving rapidly from a hazy prototype to an established technology, is likely to be another milestone in IT history. For even more individuals and enterprises, gone are the days when struggling with complicated servers and sophisticated local data management systems, which will provide them with great flexibility, reliability and economic

This article belongs to the Topical Collection: *Special Issue on Web Information Management and Applications*
Guest Editors: Yi Cai and Jianliang Xu

✉ Xiuxia Tian
xxtian@fudan.edu.cn

[1] Faculty of Information, East China Normal University, Shanghai, 200062, China

[2] College of Computer Science & Technology, Shanghai University of Electric Power, Shanghai, 200090, China

savings. As we have noticed, several tech giants and startups are taking the potential of this market, giving rise to some typical platform like Microsoft Azure, Amazon Web Services and Google Cloud Platform.

From a remote shared cluster of configurable computing resources, cloud customers can enjoy the on-demand high-quality applications and services based on their stored data in the cloud server [9]. Meanwhile, due to privacy concerns, data owners tend to encrypt sensitive data (e.g. personal health conditions, incomes, addresses, etc.) prior to outsourcing, which in turn obsoletes the data utilization.

To explore more business logic over encrypted data, many researchers and scholars in academia and industry have been devoting much effort to the enhancement of practicability and feasibility in encrypted databases. CryptDB [20] is a distinguished encrypted database which supports normal queries being executed over ciphertext. It designs a *proxy middleware* between clients and servers, where initial SQL statements are transparently overwritten. With extra help of several extended ancillary columns, original semantics are preserved in ciphertext-based queries.

Additionally, some encryption algorithms are developed to enrich basic functions in encrypted databases. Searchable symmetric encryption (SSE) is proposed for keyword searching with encrypted inverted indexes [4, 13, 21, 24]. Then, dynamic searchable symmetric encryption (DSSE) achieves alterations on various centralized indexes to enhance applicability [2, 11, 12, 14]. Furthermore, research work about similarity searching among documents or words via locality sensitive hashing (LSH) algorithm is widely discussed [1, 7, 8, 10, 15, 17, 18, 23, 25–27]. Unfortunately, these existing schemes are not applicable to outsourced databases due to the design of centralized index, but even more damaging was the lack of support for wildcard-based fuzzy searching. Our previous work [5], for the first time, fulfills wildcard-based fuzzy searching (i.e. SQL statements with 'like' clauses including wildcards '%' and '_') directly over encrypted data (on the foundation of CryptDB). Nevertheless, query accuracy can hardly satisfy cloud customers when one or more underscores included in 'like' clauses.

In this paper, we study further in this field, and put forward some exciting improvements in accuracy of fuzzy searching. As is shown in Figure 1, the proxy middleware between clients and outsourced cloud databases in the *CPD* framework achieves transparency and semantic preserving by overwriting SQL statements properly. Thanks to the new statistical language models *CFA/ACM/ACT*, the newly-designed component *character filling* has managed to upgrade the accuracy of searching result. The major contributions of our proposed schemes are summarized as follows:

- An enhanced wildcard-based fuzzy searching scheme (*enWFS*) is proposed, which expands extra functionality and practicability for the client-proxy-database framework like CryptDB.
- Two types of ancillary columns: c-LSH and c-BF, are well-designed. The former type works for similarity searching via locality sensitive hashing algorithm, while the latter works for maximum substring matching via specific BloomFilter vectors.
- With additional boost of new *Adjacent Character Matrix/Tensor* based on the corpus of attribute values, we put forward the idea of *character filling*, and present an advanced adaptive overwriting method (*AAOM*, for short) to handle queries of different wildcard cases with *better accuracy*.
- On account of the accuracy loss resulting from the ciphertext-based queries, *TupleRank* over searching results is proposed to serve cloud customers with better searching experience.

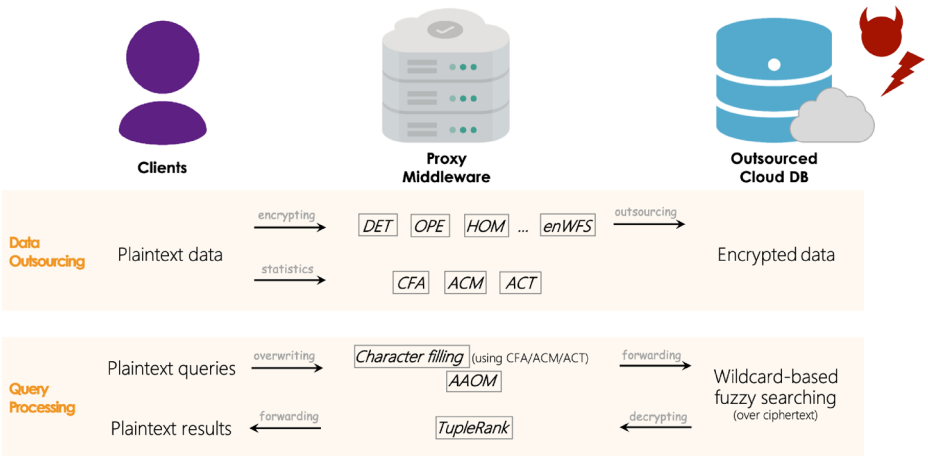


Figure 1 **Client-Proxy-Database(*CPD*) framework** synthesizes various types of encryption together at proxy middleware, where deterministic encryption (*DET*) preserves symmetric character for en/decryption, order-preserving encryption (*OPE*) preserves order among numeric values, homomorphic encryption (*HOM*) achieves aggregation computing, and wildcard-based fuzzy searching (*enWFS*) handles queries with 'like' clauses. During data outsourcing, apart from the encryption of plaintext data, statistics over the plaintext data will also be conducted, resulting in specialized statistical language models (*CFA*/*ACM*/*ACT*). During query processing, the advanced adaptive overwriting method (*AAOM*), together with *character filling* will revise the original query statement, so that the revised statement can be executed directly over encrypted databases. Finally, *TupleRank* over results can improve the searching experience. The proxy middleware is reliable and secure, while cloud DB may attract adversaries' attacks

- With extensive experiments on three real datasets, we evaluate the time consumption, memory cost, storage overhead and matching performance of ancillary columns by adjusting the parameters. Result accuracy and efficiency of *enWFS*, which are examined subsequently, indicate the effectiveness and feasibility of our proposal.

This paper is an extension of our previous conference paper[5]. Compared with the conference version, this journal version proposes new statistical language models (Section 3.2), newly-designed component *character filling* (Section 4.2) and an Advanced Adaptive Overwriting Method (Section 4.2), so that *the accuracy and validity of fuzzy searching results are significantly improved* (especially in the case of queries with underscores '_', see Section 5.2.4). *TupleRank* (Section 4.3) is introduced additionally to improve users' searching experiences. Last but not least, we provide more intelligible illustrations and examples for readers to comprehend our ideas readily.

The rest of this paper is organized as follows. We review the related work in Section 2. Section 3 introduces some basic concepts, statistical language models and the adversary model for our proposal. *enWFS* scheme including details about ancillary columns, *character filling*, advanced adaptive overwriting method (*AAOM*) is demonstrated in Section 4, together with *TupleRank*, extra security-enhancing improvements and the security analysis. Section 5 reports performance evaluation. The paper is summarized in Section 6.

2 Related work

With the popularity of cloud computing, security issues in data outsourcing have drawn much public concern, which stimulates much research effort in this field. We will review several security schemes of outsourced database and searchable encryption schemes as follows.

2.1 Security schemes of outsourced database

The innovative proposal of CryptDB [20] by Popa et al. provides a practical proxy-based middleware to combine various attribute-based encryption algorithms over encrypted database, attracting world-wide attention to security schemes of outsourced database. Then much subsequent work [16, 19, 22] further studied its security definitions, feasible frameworks, extensible functions and performed some optimization.

Tetali et al. introduced Paillier cryptosystem, ElGamal encryption, DET (deterministic), OPE (order-preserving encryption), random function into MyCrypt [22], which is designed to be integrated with parallel computing framework like MapReduce. However, the only support of massive data computation scenario limits its expansibility.

Li et al. proposed an lightweight framework L-EncDB [16] for outsourced database. L-EncDB employs format-preserving encryption, fuzzy query encryption, order-preserving encryption to process different types of query respectively. Nevertheless, this proposal weakens the role of middleware and relies too much on those encryption algorithms, resulting in some defects in functionality, expansibility and security.

2.2 Searchable encryption schemes

Searchable symmetric encryption (SSE) is designed to enhance querying over ciphertext. Much early proposals [4, 13, 21, 24] are based on inverted index. Then, some other work [2, 11, 12, 14] employs dynamic searchable symmetric encryption (DSSE) that optimizes updating strategy in inverted index. Besides, the studies about searching with Boolean expressions are explored to increase accuracy [3].

In recent years, many proposed schemes have been attempting to achieve fuzzy searching with the help of similarity matching [1, 8, 10, 15, 23, 25–27]. Most of those proposals introduce locality sensitive hashing (LSH) to map similar items together, and Bloom Filter to change measuring methods. Tetali et al. [23] by Wang et al. is one of the earliest schemes to propose fuzzy searching, where every word in each file is encoded, resulting in a Bloom-Filter vector. Then, it evaluates similarity by computing the inner products among vectors for top-k results. Kuzu et al. [15] generated feature vectors by embedding keyword strings into the Euclidean space which approximately preserves the relative Levenshtein distance. Fu et al. [8] proposed an efficient multi-keyword fuzzy ranked searching scheme which can tolerate common spelling mistakes. Wang et al. [26] by Wang et al. stores encrypted inverted indexes while mapping similar objects into the same or neighbor keyword buckets by LSH algorithm according to Euclidean distance, which eliminates sparse vectors from bi-gram mapping and promotes the correctness. Consequently, their proposal can support similarity searching over large-scale feature-rich encrypted multimedia data with the help of a high-dimensional feature vector. Unfortunately, wildcard-based fuzzy searching is not supported in those existing schemes. Other drawbacks, such as coarse-grained metric of similarity comparison and immoderate dependence on ancillary programs, restrict the practicability in real applications as well.

3 Preliminaries

In this section, some basic concepts for our schemes are demonstrated first, then the specialized Adjacent Character Matrix / Tensor, function model and the adversary model are presented.

3.1 Basic concepts

3.1.1 N-gram

The n-gram model is a typical probabilistic language model for predicting the next item in a sequence based on the $(n-1)$-order Markov model, which is now extensively applied in communication theory, statistical natural language processing, biological sequence analysis and data compression. With the co-occurrence probability of a contiguous sequence of n items from a given sample of text, it can infer the structure or other information of the sample, where the 'item' (or named 'unit') can be syllables, characters, words or base pairs according to various applications.

In general, bi-gram is the most commonly used method to partition a sequence, where each change of single character will double the influence on bi-gram results. Tri-gram provides better judgment on the next item, but it is a more strict method which only suits some specific scenes like existing judgment. Counting uni-gram preserves repetitions, which brings benefits to letter-confused comparison cases such as letter misspelling, letter missing, and reverse order of two letters in a word. However, it reduces the degree of constraint while increases false positives. Table 1 shows a few character-based n-gram methods for preserving different implicit relations within original strings[1].

In order to compress storage space in encrypted databases while achieving fuzzy searching, we partition attribute values into several fragments with character-based n-gram method, so the connotative relations among characters are preserved. We will adopt some of these n-gram methods and evaluate their performances in our experiments. Afterwards, these n-gram fragments will be further vectorized by different algorithms.

3.1.2 Bloom Filter

Judging whether a specific element exists in prepared sets remains critical in numerous applications, but traditional solutions based on Hash Table may expose its low storage efficiency if sets are of large scale. Consequently, a compact data structure named *Bloom Filter* came into being, which can reflect existence of specific elements in the prepared set, as is shown in the following example (Figure 2).

In our wildcard-based fuzzy searching scheme, given fragments of the original string $\mathcal{S} = \{e_1, e_2, e_3, \cdots\}$, the Bloom Filter maps each element e_i into a sparse array by several independent hash functions (*i.e.* converts the fragments into a sparse vector), so that a newly coming substring e' can be judged whether it is in the original string (*i.e.* Positive answer will be returned *iff* all bits in matched positions are true).

[1]For the sake of simplicity, among illustrations of this paper, the beginning and the end of a string will be uniformly represented as '#' (as you can see in Table 1). But these two cases will be treated respectively during the implementation.

Table 1 Various N-gram Forms in our scheme

N-gram methods	Value	Description
(original string)	apple	/
counting uni-gram [8]	a1, p1, p2, l1, e1	preserve repetitions
bi-gram	ap, pp, pl, le	preserve adjacent letters
tri-gram	app, ppl, ple	preserve triple adjacent letters
prefix and suffix	#a, e#	the beginning/end of a string

3.1.3 Locality sensitive hashing

One of the typical applications of Locality-Sensitive Hashing (LSH) in high-dimensional data is to provide an efficient method for approximate nearest neighbor searching, where $p_1 > p_2$ and $d_1 < d_2$ are needed in Definition 1.

Definition 1 (Locality-Sensitive Hashing) Given a distance metric function $d(\cdot, \cdot)$, a hash function family $\mathcal{H}$ is $(d_1, d_2, p_1, p_2) - sensitive$ if each function $h \in \mathcal{H}$ satisfies:
if $d(x, y) \leq d_1$, $Pr[h_i(x) = h_i(y)] \geq p_1$;
if $d(x, y) \geq d_2$, $Pr[h_i(x) = h_i(y)] \leq p_2$.

The specific manifestation of LSH is different under various measurements, but there is no available method for levenshtein distance among text. Hence, character-based n-gram will convert a string to several fragments first, and then the Bloom Filter and MinHashing algorithm will return a short integer 'signature'. Eventually, LSH can achieve nearest neighbor searching.

3.2 The specialized adjacent character matrix / tensor

In encrypted databases, ciphertext-based queries, combined with wildcards, may lead to accuracy loss in the query results, which often shows up as including many redundant tuples in the answers. To explore a better trade off between data security and accuracy of

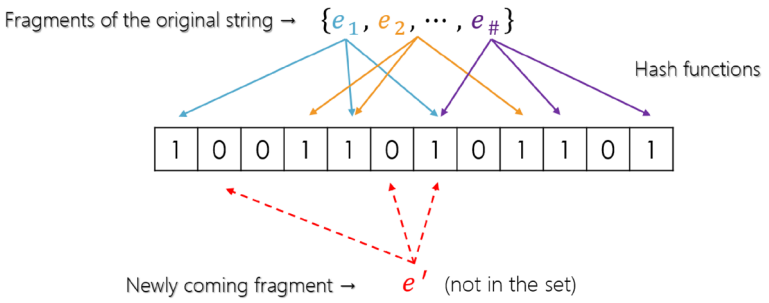


Figure 2 **A simple Bloom Filter Structure**: Each element in the set will be hashed into different positions in an array by several hash functions. When the existence of a coming element e' needs confirming, e' will also be hashed by the same group of functions. Then, e' will be reckoned to be "in the set" *iff* all hashed positions have value '1'

query results, we design specialized statistical language models (named Adjacent Character Matrix & Adjacent Character Tensor, or *ACM* & *ACT* for short) to depict connotative features of attribute values in the database.

Before data owners encrypt and outsource their data to the cloud database, they should conduct some analyses over attribute values of the whole table or some specific columns[2]. To be specific, data owners should follow the steps below to obtain Adjacent Character Matrix and Tensor:

(1) Assuming that all the values in databases only contain printable ASCII characters, we build a matrix $M[100][100]$ and a tensor $T[100][100][100]$;
(2) All the plaintext strings in those specific columns will be partitioned into bi-gram and tri-gram fragments;
(3) We run some statistical analyses over the bi-gram and tri-gram fragments by counting the frequency of each unique fragment;
(4) Update the relevant grids (items) in *ACM* and *ACT*.

Let's see an illustrative example to understand the process better. The left side of Figure 3 gives the statement of the example, where we suppose there are only two words in the corpus of attribute values, and six characters exist in the domain accordingly. After strings are partitioned and fragments are counted, Adjacent Character Matrix (*ACM*) and Adjacent Character Tensor (*ACT*) will be obtained.

Apart from *ACM* and *ACT*, an array recording the character frequency in the same corpus is also necessary. We name it Character Frequency Array, or *CFA* for short.

3.3 Function model

Let $\mathcal{D} = \{D_1, D_2, D_3, \cdots\}$ denotes the set of sensitive columns in a data table, $D_i = \{d_i^1, d_i^2, d_i^3, \cdots\}$ denotes different attribute values d_i^j in column D_i, and $d_i^j = \{w_{i,j}^1, w_{i,j}^2, w_{i,j}^3, \cdots\}$ denotes the set of words (substrings) $w_{i,j}^k$ contained in d_i^j. Then, two types of ancillary columns based on different indexing methods are proposed.

The former type of columns is named c-LSH, in which the ciphertext values result from Locality Sensitive Hashing:

$$v - LSH_i^j = \mathcal{L}_{(m,n)}(\bigcup_{k=1}^{\left|d_i^j\right|} \mathcal{G}_{ss}(w_{i,j}^k)) \tag{1}$$

where $\mathcal{L}_{(m,n)}$ denotes the LSH converting operator with LSH dimension[3] m and matching parameter n, and $\mathcal{G}_{ss}(\cdot)$ denotes n-gram methods for similarity searching.

The latter type of columns is named c-BF whose values result from Bloom Filter:

$$v - BF_i^j = \mathcal{B}_{(h,l)}(\bigcup_{k=1}^{\left|d_i^j\right|} \mathcal{G}_{msm}(w_{i,j}^k)) \tag{2}$$

[2]In many real applications, sensitive information may only contained in some of the columns in data tables, so data owners just need to focus on those specific columns. *i.e.* attribute values in those specific columns are regarded as the corpus.

[3]LSH dimension means the number of features returned from LSH corresponding to a word in attribute values. Details will be introduced in the next section.

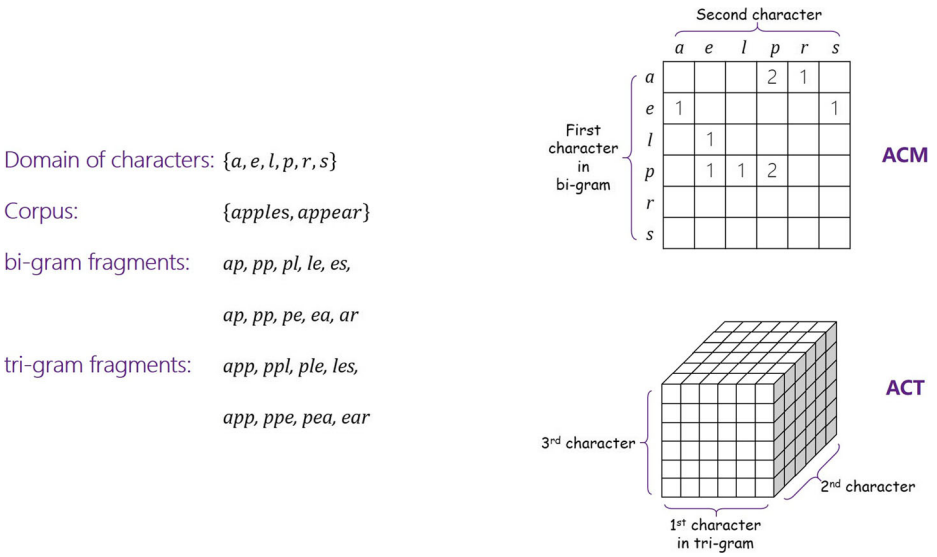


Figure 3 An example to illustrate construction of *ACM* & *ACT*

where $\mathcal{B}_{(h,l)}$ denotes the Bloom Filter converting operator with the length of vector space l and the number of hashing functions h, and $\mathcal{G}_{msm}(\cdot)$ denotes n-gram methods for maximum substring matching.

On the basis of aforementioned concepts and modules, we present the formalized definition of *enWFS* scheme, which mainly illustrates the processes of function generation, feature extraction, ciphertext conversion and expression overwriting.

Definition 2 (Formalized definition of *enWFS*) A proxy-based encrypted database supports wildcard-based fuzzy searching with the following polynomial-time algorithms (steps):

- $(K_{det}, \mathcal{L}_{(m,n)}, \mathcal{B}_{(h,l)}) \leftarrow KeyGen(\lambda, m, n, h, l)$: Given security parameter λ, LSH dimension m, matching parameter n, length of vector in Bloom Filter l, and the number of hashing functions h, $KeyGen(\cdot)$ returns a primary key K_{det} for symmetric encryption, LSH converting operator $\mathcal{L}_{(m,n)}$,and Bloom Filter converting operator $\mathcal{B}_{(h,l)}$. Security parameter λ helps initialize the hash functions and randomization processes.
- $(CFA, ACM, ACT) \leftarrow Stat_{i,j,k}(\mathcal{G}(w_{i,j}^k))$: N-gram method $\mathcal{G}(\cdot)$ first partitions given words (substrings) contained in attribute values, then $Stat(\cdot)$ conducts some statistical analyses (frequency counting)[4] over the characters and n-gram fragments, so CFA, ACM and ACT are acquired.
- $(v-DET_i^j, v-LSH_i^j, v-BF_i^j) \leftarrow \text{Index}(d_i^j, \mathcal{L}_{(m,n)}, \mathcal{B}_{(h,l)})$: Given LSH converting operator $\mathcal{L}_{(m,n)}$ and Bloom Filter converting operator $\mathcal{B}_{(h,l)}$, plaintext attribute values $d_i^j = \bigcup_{k=1}^{|d_i^j|}\{w_{i,j}^k\}$ are encrypted into deterministic (DET) ciphertext $v-DET_i^j$, LSH

[4]This is an *offline* process conducted by data owners regularly. After updates of CFA, ACM and ACT are completed locally, both will be outsourced to the cloud.

ciphertext $v - LSH_i^j$, and BloomFilter ciphertext $v - BF_i^j$ (which will be stored in ancillary columns).

- $(v - DET_i^j || v - LSH_i^j || v - BF_i^j) \leftarrow Query(OW(queries\ with\ 'like'\ clauses))$: Given query expressions with 'like' clauses, the advanced adaptive overwriting method $OW(\cdot)$ revises the expressions with regard to different wildcard cases. After the results (DET ciphertext) are returned from the cloud database, they will be decrypted with K_{det}.

There also exist other functions such as updating and deleting that achieve dynamics of our schemes, and our proposed overwriting method can be extended to scenarios where statements include 'create' and 'insert'.

3.4 Adversary model

In consideration of data confidentiality, most cloud users will encrypt sensitive columns before outsourcing the data to cloud DB, whereas they aren't willing to compromise the practicality due to the encryption. So these users may use fuzzy searching schemes like our previously-proposed *FSE* and new *enWFS* to expand functionality. On the other hand, adversaries may attempt to invade data privacy via *additional components* brought about by fuzzy searching schemes.

As for our *enWFS* scheme, the security threats can be summarized into four parts: indexes, statistical language models, trapdoor queries, and confidentiality of results. Indexes refer to the functional ciphertext (LSH & BloomFilter ciphertext in ancillary c-LSH and c-BF columns) that helps to find tuples in fuzzy searching over encrypted columns. Three statistical language models include Character Frequency Array, Adjacent Character Matrix, and Adjacent Character Tensor. The trapdoor queries mean the revised SQL statements (by our overwriting method) that can be executed directly in encrypted databases.

Focusing on those threats, adversaries may hack into cloud DB servers (as shown in Figure 1), crack the functional ciphertext, and monitor cloud users' queries, trying to extract private data in sensitive (encrypted) columns.

4 Our proposed *enWFS* scheme

We first introduce how two types of ancillary columns come into being in this section. Subsequently, the advance adaptive overwriting method, *TupleRank* and some security-enhancing improvements are described.

4.1 Two types of ancillary columns

On the basis of *CPD* framework, the design of attribution splitting in cloud databases synthesizes diverse types of encryption that can handle different query semantics. An instance in Table 2 shows two types of ancillary columns (c-LSH and c-BF) appended into a data table in the cloud database, along with a column of symmetric deterministic (DET) ciphertext (c-DET) that is encrypted with reversible encryption[5].

[5]After several tuples are matched via ancillary columns, corresponding DET ciphertext will be returned for decryption. *i.e.* Ancillary columns are used for fuzzy searching, and all matched results are achieved by decrypting DET ciphertext.

Table 2 Ancillary columns stored in encrypted databases

$c-DET$	$c-LSH$(m=4,wid=2)	$c-BF_1$	...	$c-BF_{\lceil l/32 \rceil}$
En("I love apple")	19030024,01000409,00020412	1077036627	...	1957741388
En("lave banana")	01000409,00020303	1079642851	...	625017556
En("I love coconut")	19030024,01000409,06000700	1626500087	...	1687169793

4.1.1 c-LSH

The c-LSH column stores ciphertext of attribute values generated by Locality Sensitive Hashing, which will be used for similarity searching. LSH maps similar items together with certain probability that equals to the jaccard distance between their Inverted Position Arrays (IPA for short).

To be specific, our scheme follows the steps below to generate LSH ciphertext (see the example in Figure 4):

1. Split the attribute value d_i^j into words $w_{i,j}^k$ (*e.g.* $w_{i,j}^k$ = 'lave' $\in d_i^j$ in the instance);
2. Partition each word $w_{i,j}^k$ into several fragments using bi-gram or tri-gram methods (bi-gram method is used in the instance);
3. Map each fragment into a sparse vector by Bloom Filter. Then record the positions in the vector where bit '1' stands, resulting in an Inverted Position Array;

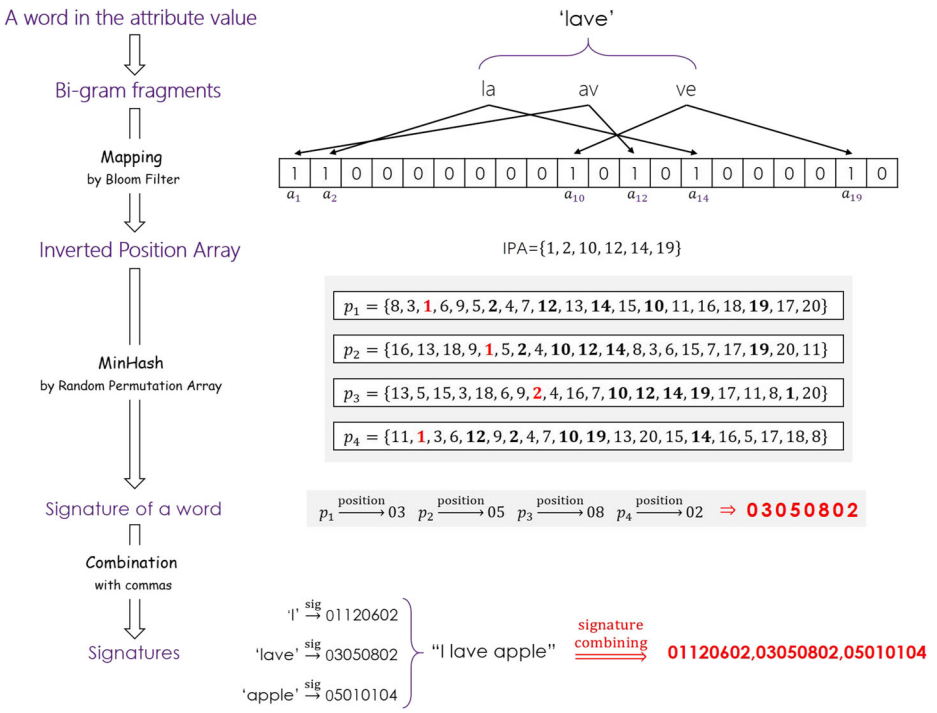


Figure 4 A sample with bi-gram method to illustrate how LSH ciphertext of an attribute value is generated

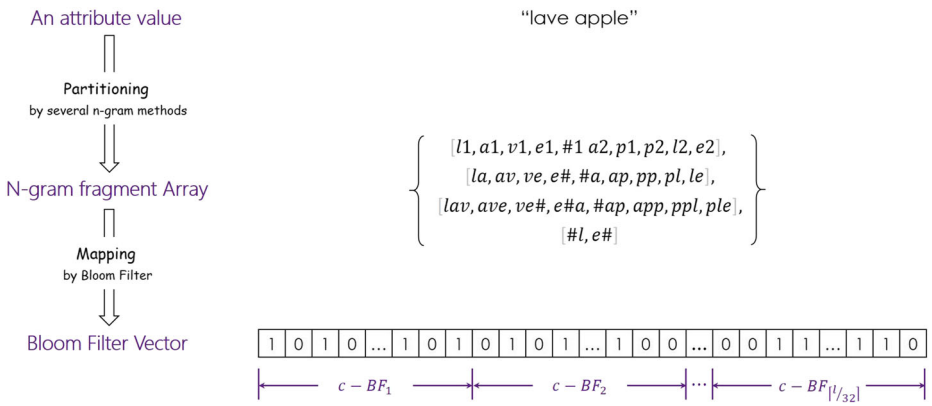


Figure 5 An instance to illustrate the process of Bloom Filter conversion

4. Execute dimension reduction with LSH (MinHash), we can obtain an m-dimensional feature (signature)[6] (LSH dimension $m = 4$ in the instance);
5. If an attribute value consists of several words, combine signatures of each word with commas. So far, we have got LSH ciphertext of $d_i^j : v - LSH_i^j$.

Such a converting process shifts measurement from levenshtein distance on text to jaccard similarity on IPAs, so that the particular MinHash algorithm could reduce the dimension of numeric features for each item. Eventually, each attribute value is converted into a linked sequence as LSH ciphertext that will be stored in the c-LSH column.

4.1.2 c-BF

As a supplement to the c-LSH column above, the other type of ancillary columns, c-BF, consists of multiple columns that store ciphertext of attribute values generated by Bloom Filter. Since the design of c-BF is aimed at maximum substring matching, connotative relations among characters should be preserved as much as possible. Consequently, we adopt several n-gram methods[7] in Bloom Filter converting. The following steps, together with the example in Figure 5, help to illustrate this process better.

(1) Suppose that an attribute value d_i^j is to be converted into BloomFilter ciphertext, the entire string will be input into several n-gram methods;
(2) N-gram fragments from disparate methods (*e.g.* counting uni-gram, bi-gram, tri-gram, prefix and suffix are used in the instance) are grouped together into an array, then Bloom Filter maps the fragment array into a fixed-length sparse vector;
(3) Due to the limitation of field length in databases, Bloom Filter vector will be split to a series of segments. These segements are stored in columns $c - BF_1$, $c - BF_2$, $\cdots$, $c - BF_{\lceil l/32 \rceil}$ respectively (supposing length limitation being 32 bits).

[6] A word's signature on each dimension is assured of an equal width by padding zero.
[7] We use popular n-gram methods described in Section 3.1.1 .

After BloomFilter ciphertext is generated, maximum substring matching will be executed via bit matching between the bit vector of the keyword in the 'like' clause and that stored in separate columns ($c-BF_1 \sim c-BF_{\lceil l/32 \rceil}$), which can be implemented through native logical operator *AND* ('&') over bits in databases. Accordingly, original SQL statements should be overwritten like this: *select c-DET from T where* $c-BF_0$ *& 1=1 and* $c-BF_1$ *& 3=3*.

To achieve better effectiveness of Bloom Filter, we will test the accuracy of maximum substring matching under different lengths of BloomFilter vectors in Section 5.2.2.

Algorithm 1 The Advanced Adaptive Overwriting Method (AAOM).

Input: the original query expression (with 'like' clauses) Q, CFA, ACM, ACT, flag array of 'like' clauses F
Output: the revised expression that can match ciphertext in CryptDB $\mathcal{Q}$

$\mathcal{Q}$ = Preprocess(Q); //split a clause s into some if '%' contained within s (not on both sides) *e.g.* col LIKE 'ab%c%' ⇒ col LIKE 'ab%' and col LIKE '%c%'
$F = F$.ProcessAccordingTo(Q, $\mathcal{Q}$);
$S = \{s_1, s_2, \cdots\}$ is the set of 'like' clauses extracted from $\mathcal{Q}$; //$s_i = \{c_j \mid c_j$ denotes characters in s_i (in original order)$\}$
Initialize $\mathcal{V} = \varnothing$;
foreach $s_i \in S$ **do**
    Initialize $V_i = \varnothing$ corresponding to s_i;
    if no wildcards '%' or '_' contained in s_i **then**
        $v_i^1 = s_i$.ConvertToDET(); $V_i = V_i \cup \{v_i^1\}$; // V_i here is DET ciphertext
        $\mathcal{V} = \mathcal{V} \cup \{V_i\}$; //$\mathcal{V}$ stores various ciphertext of clauses
    else
        $s_i' = s_i$.strip('%') ;
        V_i = Sub-AAOM(s_i', f_i, CFA, ACM, ACT); //run Algorithm 2
        Append '%' to each $v_i^j \in V_i$ if V_i being LSH ciphertext and '%' exists on either side of original s_i;
        $\mathcal{V} = \mathcal{V} \cup \{V_i\}$;
foreach $V_i \in \mathcal{V}$ **do**
    if several elements $(v_i^1, v_i^2, \cdots)$ exist in V_i **then**
        Split corresponding clause s_i of $\mathcal{Q}$ into several clauses $s_i^1, s_i^2, \cdots$;
        Append logical operators 'and' among BloomFilter ciphertext segments, while append 'or' among different cases of character filling in certain clauses with '_' (see Alg. 2) ;
        Replace corresponding plaintext $s_i^1, s_i^2, \cdots$ with ciphertext $v_i^1, v_i^2, \cdots$;
    else
        Replace corresponding plaintext s_i with ciphertext v_i^1;
    Change original column names into names of corresponding ancillary columns;
return $\mathcal{Q}$;

4.2 The Advanced Adaptive Overwriting Method (*AAOM*)

Algorithm 2 Sub-algorithm of algorithm 1 (Sub-AAOM)

Input: a preprocessed 'like' clause s_i', the flag of the clause f_i, CFA, ACM, ACT
Output: the set of revised clause V_i

mcu = the number of maximum consecutive underscores ('_') in s_i';
Initialize $V_i = \varnothing$ corresponding to s_i';
switch mcu **do**
  case *0*
    if f_i = 'w' **then**
      v_i^1 = LSHMapping(Bi-gram(s_i')); $V_i = V_i \cup \{v_i^1\}$;
      return LSH ciphertext V_i;
    else
      $a = s_i'$.nGram(uni, bi, tri).ToArray(); //see Fig. 5
      V_i = BFMapping(a).BFSplit(LEN);
      return BloomFilter ciphertext V_i;
  case *1*
    Form the formula $\mathcal{P}(\lambda_1, \lambda_2, \cdots) = \prod_{j=2}^{|s_i'|} P(c_j|c_{j-1})$, where $P(c_j|c_{j-1}) = \frac{ACM[c_{j-1}][c_j]}{CFA[c_{j-1}]}$; // variable λ_k refers to respective '_' in s_i'; *e.g.* $P(p|a) = \frac{ACM[a][p]}{CFA[a]} = \frac{1087}{3215}$, $P(\_|a) = \frac{ACM[a][\lambda_1]}{CFA[a]} = \frac{ACM[a][\lambda_1]}{3215}$.
    break;
  case *2*
    Form the formula $\mathcal{P}(\lambda_1, \lambda_2, \cdots) = \prod_{j=3}^{|s_i'|} P(c_j|c_{j-2}, c_{j-1})$, where $P(c_j|c_{j-2}, c_{j-1}) = \frac{ACT[c_{j-2}][c_{j-1}][c_j]}{ACM[c_{j-2}][c_{j-1}]}$; break;
  otherwise
    $a_i = s_i'$.nGram(uni, bi, tri).ToArray();
    V_i = BFMapping(a_i).BFSplit(LEN);
    return BloomFilter ciphertext V_i;
Fill variable combination $\Lambda = (\lambda_1, \lambda_2, \cdots)$ in $\mathcal{P}$ with characters, so the candidate set $\Delta = \{\Lambda_1, \Lambda_2, \cdots\}$ is achieved;
foreach $\Lambda_k \in \Delta$ **do**
  Traverse in CFA, ACM or ACT to compute $\mathcal{P}_{\Lambda_k}$;
  if $\mathcal{P}_{\Lambda_k} == 0$ **then**
    Eliminate Λ_k from Δ;
if f_i = 'w' **then**
  $j = 1$;
  foreach possible variable combination $\Lambda_k \in \Delta$ **do**
    v_i^j = LSHMapping(Bi-gram(s_i'.fillUnderscore(Λ_k)));
    Insert v_i^{j++} into V_i in descending order according to $\mathcal{P}_{\Lambda_k}$;
  return LSH ciphertext V_i;
else
  foreach $\Lambda_k \in \Delta$ **do**
    $a = s_i'$.FillUnderscore(Λ_k).nGram(uni, bi, tri).ToArray();
    $V_i = V_i$.AppendBFGroup_DESC(BFMapping(a).BFSplit(LEN), $\mathcal{P}_{\Lambda_k}$);
  return BloomFilter ciphertext V_i;

As is known to us, wildcard characters are used in 'like' clauses within SQL statements. The underscore symbol '_' represents an arbitrary single character in the string, while the percentage symbol '%', a coarse-grained wildcard, matches zero or more characters.

In our previously proposed overwriting method[5], the c-LSH column is suitable for processing 'like' clauses where *unbroken words* without underscore '_' are included[8], and achieves ideal performance in accuracy. Unfortunately, the performance will decline gradually when one or more underscores are contained in the clause (The multiple columns c-BF are used in this case). Thus, we make every effort to explore some reformation over the previous method.

Based on aforementioned two types of ancillary columns, together with Character Frequency Array and the specialized Adjacent Character Matrix & Adjacent Character Tensor, we introduce the newly-designed component ***character filling*** into the correspondingly-improved advanced adaptive overwriting method (*AAOM*) in this section. Algorithm 1-2 describe our proposal. We will also provide the following instance to better illustrate our idea.

Suppose that a cloud user is going to execute a wildcard-based fuzzy searching Q, he should also submit a flag array of 'like' clauses $F = \{f_1 =$ 'w', $f_2 =$ 'f'$\}$ to declare whether an *unbroken word*('w') or a *fragment of words*('f') is contained in each 'like' clause[9].

$$Q = \text{select } pname\_cDET, paddress\_cDET \text{ from } T \\ \text{where } pname \text{ like } 'a\_\_le\%' \text{ and } paddress \text{ like } 'h\%l\_nd';$$

Our proposed method takes the following steps to overwrite the SQL statement that can be executed in encrypted databases:

(1) (Line 1, Alg. 1) Preprocess the query Q: split a clause s into some if '%' contained within s (not on both sides). Then we can obtain
$\mathcal{Q} =$ select $pname\_cDET, paddress\_cDET$ from T
where $pname$ like $'a\_\_le\%'$
and $paddress$ like $'h\%'$ and $paddress$ like $'\%l\_nd'$;

(2) (Line 2, Alg. 1) According to the clause splitting in the previous step, F is revised as $F = \{f_1 =$ 'w', $f_2 =$ 'f', $f_3 =$ 'f'$\}$;

(3) (Line 3, Alg. 1) The clause set $S = \{s_1 =' a\_\_le\%', s_2 =' h\%', s_3 =' \%l\_nd'\}$ is obtained;

(4) For each s_i in S, *AAOM* will convert it to different types of ciphertext (with regard to different cases as follows);

(5) (Lines 7-9, Alg. 1) A certain clause without any wildcard will be directly converted to DET ciphertext for equality judgment;

(6) (Lines 11-12, Alg. 1) If some wildcards are included in the clause, percentage symbols '%' at the beginning/end of s_i will be temporarily stripped, then s_i' will be handed over to *Sub-AAOM* (run Alg. 2)
(In our instance, $s_1' =' a\_\_le', s_2' =' h', s_3' =' l\_nd'$);

[8]**Remark:** In the clause 'a_le%', 'a_le' may refer to 'apple', 'ankle', etc, so 'a_le' can be treated as an *unbroken word* in the clause 'a_le%'; On the contrary, 'app%' may refer to 'applaud', 'applicant', etc, so 'app' in the clause 'app%' is reckoned to be a *fragment of words*.

[9]This flag array provides a reference for our algorithm, and it will be revised in the process.

(7) (Line 1, Alg. 2) *Sub-AAOM* first examine s_i', and acquire maximum consecutive underscores mcu in it[10], so $mcu = 2, 0, 1$ with respect to s_1', s_2', s_3';

(8) (`For` s_2': Lines 4-11, Alg. 2) Since f_2 = 'f', 'h' will be partitioned by counting-unigram, bi-gram and tri-gram. Then, the array a with fragments from disparate n-gram methods is mapped by Bloom Filter. Finally, the obtained BloomFilter Vector is split to a series of segments (with the same length *LEN*) that are stores in $v_2^1, v_2^2, \cdots \in V_2$ (In our instance, $V_2 = \{v_2^1 = 112, v_2^2 = 630, v_2^3 = 577, v_2^4 = 400\}$ is the BloomFilter ciphertext of s_2') (details about BloomFilter ciphertext are in Section 4.1.2);

(9) (`For` s_3': Lines 12-14, Alg. 2) CFA and ACM help the algorithm to carry out *character filling*: Following the Markov Assumption and bi-gram method, we form the formula $\mathcal{P}(\lambda_1) = P(\lambda_1|l) \cdot P(n|\lambda_1) \cdot P(d|n)$;

(10) (`For` s_3': Lines 21-25, Alg. 2) Assume that character 'a' and 'u' are possible to be filled within s_3' (according to the corpus of encrypted columns), $\Lambda_1 = (\lambda_1 =' a')$, $\Lambda_2 = (\lambda_1 =' u')$ are obtained (item frequency of 'land': $\mathcal{P}_{\Lambda_1} = 0.0052$; item frequency of 'lund': $\mathcal{P}_{\Lambda_2} = 0.000021$);

(11) (`For` s_3': Lines 32-36, Alg. 2) s_3' will firstly be filled with Λ_1 and Λ_2 respectively. Since f_3 = 'f', after these two *fragments of words* ('lund' and 'land') partitioned by several n-gram methods, BloomFilter ciphertext of them will be appended to V_3 in descending order of probabilities
(*e.g.* BloomFilter ciphertext of 'lund' is {101,165,717,536}, BloomFilter ciphertext of 'land' is {202,686,873,411}, so $V_3 = \{\{v_3^1 = 202, v_3^2 = 686, v_3^3 = 873, v_3^4 = 411\}, \{v_3^5 = 101, v_3^6 = 165, v_3^7 = 717, v_3^8 = 536\}\}$ according to $\mathcal{P}_{\Lambda_1}$ and $\mathcal{P}_{\Lambda_2}$);

(12) (`For` s_1': Lines 15-16, Alg. 2) ACM and ACT help to conduct *character filling*: Following the Markov Assumption and tri-gram model, we form the formula $\mathcal{P}(\lambda_1, \lambda_2) = P(\lambda_2|a, \lambda_1) \cdot P(l|\lambda_1, \lambda_2) \cdot P(e|\lambda_2, l)$;

(13) (`For` s_1': Lines 21-25, Alg. 2) Assume that characters 'nk' and 'pp' are possible to be filled within s_1' (according to the corpus of encrypted columns), $\Lambda_1 = (\lambda_1 =' n', \lambda_2 =' k')$, $\Lambda_2 = (\lambda_1 =' p', \lambda_2 =' p')$ are obtained (item frequency of 'ankle': $\mathcal{P}_{\Lambda_1} = 0.000172$; item frequency of 'apple': $\mathcal{P}_{\Lambda_2} = 0.00371$);

(14) (`For` s_1': Lines 26-31, Alg. 2) s_1' will firstly be filled with Λ_1 and Λ_2 respectively. Now that f_1 = 'w', we partition 'ankle' and 'apple' by bi-gram method respectively. Then, LSH ciphertext of these two *unbroken words* is inserted into V_1 in descending order of probabilities (*e.g.* LSH ciphertext of 'ankle' is {05020304}, LSH ciphertext of 'apple' is {05010104}, so $V_1 = \{\{v_1^1 = 05010104\}, \{v_1^2 = 05020304\}\}$ according to $\mathcal{P}_{\Lambda_1}$ and $\mathcal{P}_{\Lambda_2}$);

(15) (Lines 13-14, Alg. 1) We temporarily strip '%' at the beginning/end of s_1, s_2, s_3 in Step 6. So here, '%' will be restored if V_i is LSH ciphertext and '%' exists on either side of original s_i (In our instance, elements in $\mathcal{V}$ are revised to $V_1 = \{\{05010104\%\}, \{05020304\%\}\}$, $V_2 = \{112, 630, 577, 400\}$, $V_3 = \{\{202, 686, 873, 411\}, \{101, 165, 717, 536\}\}$);

(16) (Lines 15-22, Alg. 1) Since several elements exist in V_1, V_2 and V_3, clauses s_1, s_2 and s_3 are split into several clauses. Then, clauses between s_1^1 and s_1^2 will be joined by 'or' (because v_1^1 and v_1^2 are related to different cases of *character filling*); clauses among $s_2^1 - s_2^4$ will be joined by 'and' (because $v_2^1 - v_2^4$ are BloomFilter ciphertext segments);

[10]We don't consider the number of total underscores. *e.g.* $mcu = 2$ for 'a_ _l_'.

clauses among $s_3^1 - s_3^4$ will be joined by 'and' (BloomFilter ciphertext segments); clauses among $s_3^5 - s_3^8$ will be joined by 'and' (BloomFilter ciphertext segments); two groups of clauses $s_3^1 - s_3^4$ and $s_3^5 - s_3^8$ will be joined by 'or' (because $v_3^1 - v_3^4$ and $v_3^5 - v_3^8$ are BloomFilter ciphertext that related to different cases of *character filling*);

(17) After original column names changed to names of corresponding ancillary columns, we can obtain final $\mathcal{Q}$:

$$\begin{aligned}
&\mathcal{Q} = \text{select } pname_cDET, paddress_cDET \text{ from } T \quad \text{where} \\
&(pname_cLSH \text{ like } '05010104\%' \text{ or } pname_cLSH \text{ like } '05020304\%') \\
&\text{and} \\
&(paddress_cBF_1 \,\&\, 112 = 112 \text{ and } paddress_cBF_2 \,\&\, 630 = 630 \\
&\text{and } paddress_cBF_3 \,\&\, 577 = 577 \text{ and } paddress_cBF_4 \,\&\, 400 = 400) \\
&\text{and} \\
&((paddress_cBF_1 \,\&\, 202 = 202 \text{ and } paddress_cBF_2 \,\&\, 686 = 686 \\
&\text{and } paddress_cBF_3 \,\&\, 873 = 873 \text{ and } paddress_cBF_4 \,\&\, 411 = 411) \\
&\text{or} \\
&(paddress_cBF_1 \,\&\, 101 = 101 \text{ and } paddress_cBF_2 \,\&\, 165 = 165 \\
&\text{and } paddress_cBF_3 \,\&\, 717 = 717 \text{ and } paddress_cBF_4 \,\&\, 536 = 536))
\end{aligned}$$

By far, *AAOM* algorithm has converted an ordinary SQL statement into that which can fulfill wildcard-based fuzzy searching in encrypted databases. Additionally, one more thing should be noted. The specialized ACM and ACT described in this paper can carry out *character filling* in cases where the number of maximum consecutive underscores is no more than 2 in a clause. If more than 2 consecutive underscores are included in the 'like' clause, the user may deliver much higher uncertainty about the string with wildcards. So we use Bloom Filter to handle this case.

Besides, ACT can be generalized to higher-dimensional structures: a x-dimensional adjacent character tensor can fulfill cases where the number of maximum consecutive underscores is no more than $x - 1$. However, in truth, clauses with maximum consecutive underscores no more than 2 are believed to cover most cases (the number of total underscores in a clause won't influence *character filling*).

4.3 *TupleRank* over searching results

TupleRank is designed to improve searching experience in wildcard-based fuzzy searching over encrypted columns in encrypted databases (not all the columns are actually encrypted), where redundant tuples are often involved in results (*i.e.* false positive). Just as a common practice indicated by contemporary web searching engines (*e.g.* Google and Baidu), we arrange the result tuples in the order of *item frequency* in the corpus of encrypted columns (*i.e.* Tuples with high item frequency in encrypted columns will be ranked forward). Intuitively, the item frequency is easy to be computed with the assistance of *CFA*, *ACM* and *ACT* (according to the *Markov Assumption*).

The *TupleRank* mechanism will take effect in the following two ways with regard to different wildcard cases in 'like' clauses.

- (`No wildcards / no underscores / not less than three consecutive underscores in 'like' clauses`) In this case, *CFA*, *ACM* and *ACT* won't be involved in the overwriting process, and no item frequency is computed. So we *postprocess* the query results via computing item frequency of each distinct value (using *CFA*, *ACM* or *ACT*), and rearranging tuples in the result.

For example, if a user launched a query Q:

$$Q = \text{select } pid, pname\_cDET \text{ from } T \text{ where } pname \text{ like } '\_a\_\_le\%';$$

And the decrypted tuples are:

$$(129,'aisle\ carpet'), (002,'apple\ juice'),$$
$$(007,'apple\ juice'), (108,'ankle\ boot')$$

Suppose that the item frequency of 'aisle carpet' is 0.00218, that of 'apple juice' is 0.0103, that of 'ankle boot' is 0.0023. Tuples after rearragnement will be:

$$(002,'apple\ juice'), (007,'apple\ juice'),$$
$$(108,'ankle\ boot'), (129,'aisle\ carpet')$$

- (`No more than two maximum consecutive underscores`) In the other case, *CFA*, *ACM* or *ACT* will take part in the overwriting process. Following the operations in Lines 12-16 & 30, Alg. 2 (corresponding to step 14 in the instance), we can break the original query into several sub-queries and schedule them according to item frequency.

  Following the same query Q in the previous example, we first obtain the overwritten query statement from *AAOM*:

$$\mathcal{Q} = \text{select } pid, pname\_cDET \text{ from } T$$
$$\text{where } pname\_cLSH \text{ like } '05010104\%' \text{ or } pname\_cLSH \text{ like } '05020304\%';$$

  Suppose that LSH ciphertext of 'ankle' is '05020304' (item frequency of 'ankle' is 0.000172), LSH ciphertext of 'apple' is '05010104' (item frequency of 'apple' is 0.00371). Sub-queries of $\mathcal{Q}$ are scheduled in the order:

$$\mathcal{Q}_1 = \text{select } pid, pname\_cDET \text{ from } T$$
$$\text{where } pname\_cLSH \text{ like } '05010104\%';$$
$$\mathcal{Q}_2 = \text{select } pid, pname\_cDET \text{ from } T$$
$$\text{where } pname\_cLSH \text{ like } '05020304\%';$$

  After these two sub-queries are successively executed, the ranked result is also achieved after easy combination.

4.4 LSH-based security improvements

In our scheme, ciphertext on ancillary columns determines the security issues. However, values in c-LSH column might leak some extra information, such as sequences of plaintext, to malicious adversaries. To maximize the preservation of data privacy, some improvements are further proposed in this section. We name the *enWFS* scheme equipped with these security-enhancing improvements *enWFS-SE*.

4.4.1 Modifying sequences of LSH ciphertext

The c-LSH column in each tuple stores a set of LSH ciphertext for the whole string, and the ciphertext of each word in the string is arranged in the original order (see Figure 4). Consequently, the sequences of ciphertext can reflect the relevancy among words. To overcome this potential loophole, we revise the sequences of ciphertext randomly by hashing permutation functions $\mathcal{H} : y = ax + b \mod c$, where a is relatively-prime to c. Since the matching process only relies on existence of LSH ciphertext rather than order, sequence

revision, which blurs the correlation between plaintext words in the string and specific LSH ciphertext, may have limited impact on the results.

4.4.2 Appending confusing LSH ciphertext

Another feasible way to enhance the security of c-LSH column is to append redundant ciphertext, but what kind of ciphertext content should be added is the target of our discussion. The first solution is appending repeated ciphertext within the set of LSH ciphertext for the whole string (*e.g.* If original LSH ciphertext of 'lave apple' is '03050802,05010104', revised ciphertext will be '05010104,03050802,05010104'). Although it's simple and effective, this solution only improves limited security. Another better solution is to append some random ciphertext that doesn't result from actual plaintext attribute values (*e.g.* The revised ciphertext of 'lave apple' is '06190214,03050802,05010104', where '06190214' is not ciphertext resulting from an actual attribute value). Because of the same fact that the matching process only relies on existence of LSH ciphertext, extra ciphertext may not influence the correct tuples that should be in the matching results. Nevertheless, this improvement will lead to the decline in result accuracy (increasing the proportion of false positive tuples).

4.5 Security analysis

In this paper, we expand extra functionality (enhanced wildcard-based fuzzy searching) for CPD-framework-based CryptDB, so the security of basic architecture (CPD framework) can be guaranteed with solutions in CryptDB [20]. In addition, we also follow the widely-accepted security frameworks in the field of searchable symmetric encryption (SSE) [6, 12, 15, 21], so the discussions about security issues in those papers are beyond the scope of this paper. In this section, we will focus on analyzing whether our scheme is resistant to security threats defined in Section 3.4.

- **Security of indexes (ancillary columns)** Suppose there are 4 columns $\{I, A_1, A_2, C\}$ in a data table T, where C is a sensitive column containing privacy. Apart from the deterministic ciphertext column (c-DET) for C, our scheme will append ancillary c-LSH column (c-LSH) and c-BF columns (c-BF_1, c-BF_2, $\cdots$, c-BF_L), resulting in encrypted data table: $\{I, A_1, A_2, c-DET, c-LSH, c-BF_1, c-BF_2, \cdots c-BF_L\}$. As is introduced in Section 4.1, ancillary c-LSH and c-BF columns contain ciphertext resulting from minHash and BloomFilter mapping process on n-gram fragments of attribute values in column C. Due to the irreversibility of the Hashing algorithms, adversaries are incapable of extracting sensitive data from ancillary columns, even though they hack into database systems. Moreover, security improvements in Section 4.4 may further promote privacy level of LSH ciphertext.
- **Security of statistical language models** Three statistical language models, CFA / ACM / ACT, which describe connotative features of attribute values, are vital components that contribute to the significant promotion in result accuracy. In our scheme, these language models are generated by counting the frequency of each unique n-gram fragment of private attribute values (see Section 3.2), and this process is conducted by data owners in local environment before outsourcing to the proxy middleware.

  For adversaries, when they are monitoring users' queries at cloud DB servers, they cannot see any trails of statistical language models in the revised SQL statements (although these language models do get involved in the overwriting process). Even if adversaries managed to intercept and achieve the models from clients, values in $M[i][j]$

or $T[i][j][k]$ only stand for the frequency of characters and n-gram fragments, which causes very limited contributions to their attacks.

- **Security of trapdoor queries** The revised SQL statements serve as trapdoor queries in our scheme. During the overwriting process, *AAOM* will check each keyword in 'like' clauses that is related to encrypted columns first. If possible, *character filling* will utilize *statistical language models* to fill in the missing characters in *unbroken words*. According to our former security analysis, this step won't raise security issues.

  Then, *AAOM* will convert those keywords into LSH ciphertext or BloomFilter ciphertext according to different cases. No matter which kind of ciphertext is involved in the revised SQL statements, the trapdoor queries won't leak data privacy, in view of the security analysis of indexes (ancillary columns).
- **Security of result confidentiality** After the fuzzy searching being executed in encrypted databases, query results including plaintext of non-sensitive columns and symmetric deterministic ciphertext of sensitive columns will be returned. Except for statistical information leakage about the user's searching pattern[11], deterministic encryption still guarantees confidentiality of results.

Additionally, another new component, *TupleRank*, which rearranges the order of tuples, doesn't violate data confidentiality as well. That's because *TupleRank* depends on statistical language model-based item frequency to postprocess results at proxy middleware or schedule several sub-queries at cloud DB. Therefore, *enWFS* scheme is resistant to security threats defined in Section 3.4.

5 Performance evaluation

In this section, we will introduce our experiment setup first, and evaluate the performance of all additional components in *enWFS* scheme.

Sections 5.2.1 and 5.2.2 evaluate the time/memory/storage overheads and matching accuracy under different parameters in c-LSH/c-BF columns. Section 5.2.3 demonstrates time/storage overheads during the construction of new statistic language models *CFA/ACM/ACT*. Section 5.2.4 assesses the accuracy of fuzzy searching results with our new component *character filling* and the improved algorithm *AAOM*. Section 5.2.5 shows the performance of another new component *TupleRank*. Section 5.2.6 depicts the time efficiency of different schemes.

5.1 Experiment setup

Three datasets will be used in our experiments, and the details of each set can be viewed in Table 3.

All the experiments are run on a computer with Intel® Core i5 CPU and 16GB RAM. The database platform is MySQL 8.0, and Java JDK version is 11. We will compare our proposed *enWFS* scheme (coded with Java) with *enWFS-SE* (*enWFS* equipped with security-enhancing improvements), *FSE* (our previously proposed method[5]), and JDBC (plaintext queries/operations over unencrypted database) from different aspects.

[11] This is a limitation of Searchable Symmetric Encryption constructions [6, 12, 15, 20, 21], which is out of the scope of our scheme.

Table 3 Datasets used in our experiments

Dataset name	Description	Size
TOEFL	Original words in Toefl vocabulary and some variants (e.g. variants of "word" are "words"/"wosrd"/"sword")	5000 rows
CSDN	A dataset contains leaked CSDN (www.csdn.net) accounts, including usernames, passwords, and emails	50000 rows
Reuter news	Several pieces of news from Reuters (www.reutersagency.com)	17805 rows

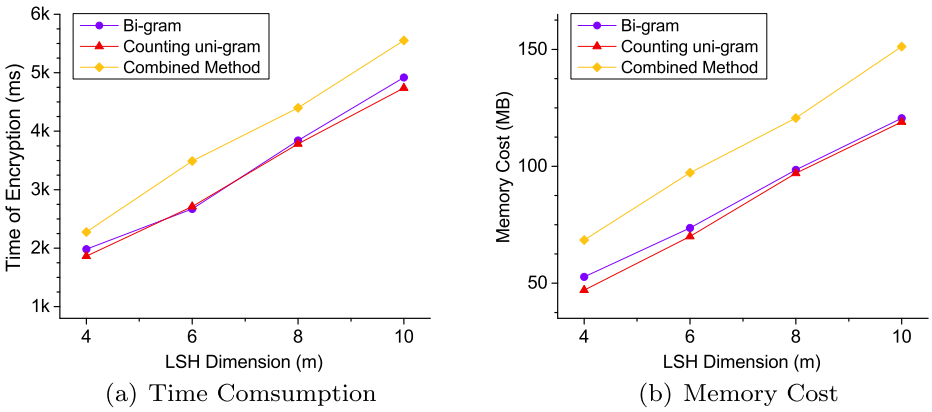


Figure 6 Time and Memory Overheads of LSH Encryption

5.2 Result evaluation

5.2.1 Performance of c-LSH column

In this part, we use TOEFL dataset to evaluate the time/memory/storage overheads in construction process and matching accuracy of the c-LSH column under different parameter values of LSH dimension m. As we have discussed above, LSH ciphertext results from n-gram fragments, so we introduce the counting uni-gram method and a combined method (with both counting uni-gram and bi-gram methods) as comparisons apart from the popular bi-gram method.

Figures 6a and b show the time consumption and memory cost during the construction of LSH ciphertext. Figure 7a depicts the storage cost of the ancillary c-LSH column. We can see that the higher LSH dimension is, the more time, memory, and storage space will be occupied. This is because more random permutation arrays are needed to generate more features of attribute values. Besides, the combined method also consume more resources, as more n-gram fragments are to be processed.

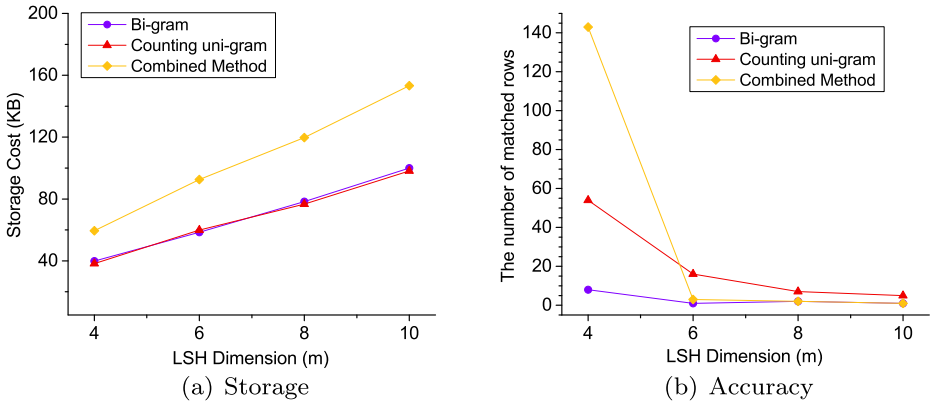


Figure 7 Storage Overhead and Performance of c-LSH

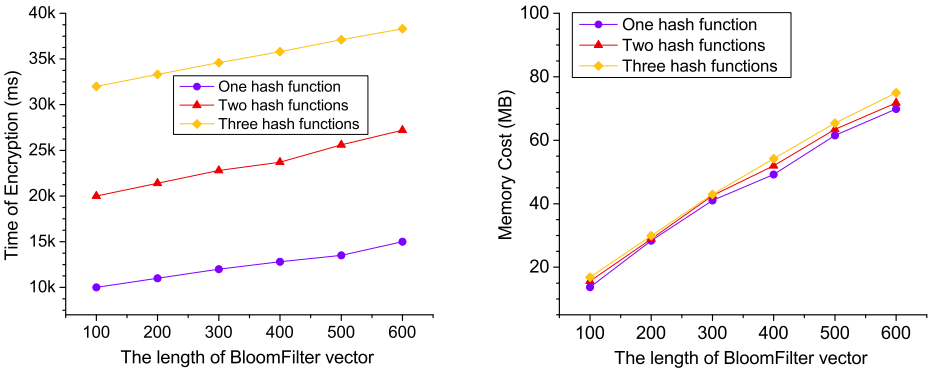


Figure 8 Time and Memory Overhead of BloomFilter Encryption

Figure 7b illustrates the matching accuracy of c-LSH column. From the tendency of matched rows, we can conclude that *the proportion of false positive reduces gradually* while the correct items remain unchanged, and a better matching effect will be achieved when LSH dimension $m \geq 6$. To sum up, $m = 6$ is an ideal parameter value which achieves the balance between time/memory/storage overheads and accuracy.

5.2.2 Performance of c-BF columns

The multiple ancillary columns c-BF are designed for maximum substring matching. In the second experiment, we use CSDN dataset to test the c-BF columns under different numbers of hash functions and BloomFilter vector length. We also attempt to find out an appropriate setting (# of hash functions and BloomFilter vector length) in c-BF columns according to the time/memory/storage overheads and matching accuracy.

Figure 8a and b depict the time consumption and memory cost during the construction of BloomFilter ciphertext. Figure 9a shows the storage cost of the ancillary c-BF columns under different length of BloomFilter vector. Intuitively, encryption will take more time when more hash functions are used, but the memory cost varies slightly among different numbers of hash functions. Storage cost of c-BF columns grows linearly with the length of BloomFilter vector. Moreover, compared with c-LSH column, the construction of c-BF columns took less average time[12] because MinHash algorithm won't be involved in BloomFilter ciphertext converting.

Figure 9b demonstrates the matching accuracy of c-BF columns. With regard to the amount of matched rows, we can conclude that false positive rate drops rapidly at first, and then gets stable when sparsity is close to one-sixth (*i.e.* About 9 ancillary c-BF columns (nearly 300 bits) are needed to store BloomFilter vectors for 50 plaintext characters). Therefore, BloomFilter vector length = 300 and one hash function reach the balance between matching accuracy and time/memory/storage overheads.

[12] About 5000 rows of data are processed in the c-LSH experiment while nearly 50000 rows of data are processed here.

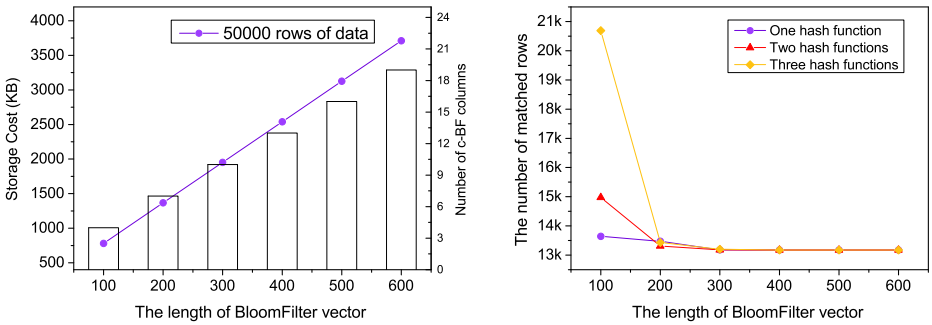


Figure 9 Storage Overhead and Performance of c-BF

Table 4 Construction of *CFA*

	Time (ms)	Storage (KB)
TOEFL (5000 rows)	15672	0.39
CSDN (50000 rows)	121743	0.39

Table 5 Construction of *ACM*

	Time	Storage
TOEFL	13432	39.06
CSDN	107576	39.06

Table 6 Construction of *ACT*

	Time	Storage
TOEFL	10383	3906.25
CSDN	82593	3906.25

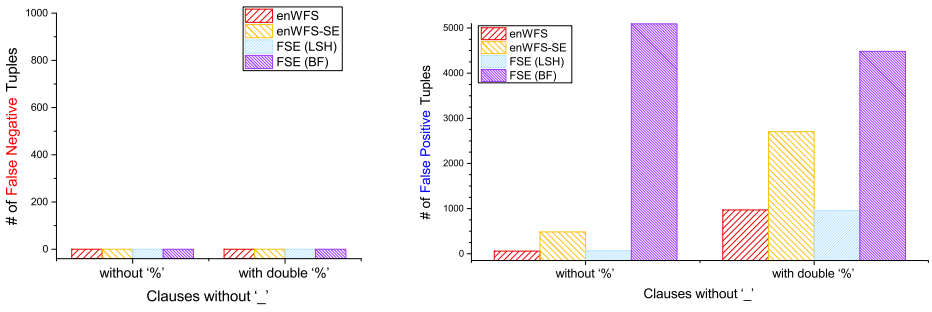
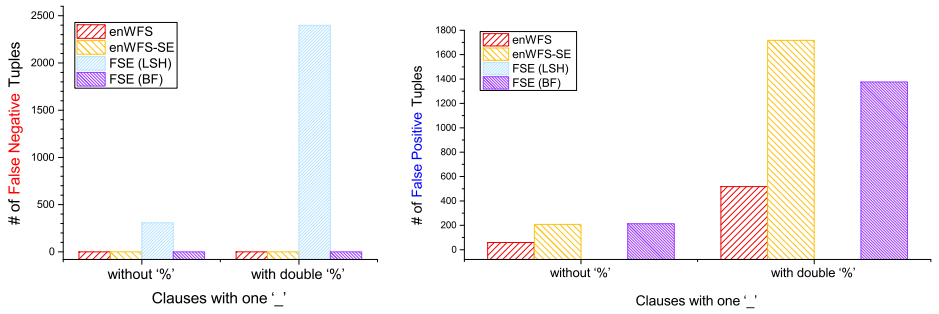


Figure 10 Accuracy of Results (Without '_' in clauses)



(a) False Negative (b) False Positive

Figure 11 Accuracy of Results (With one '_' in clauses)

Remark By far, reaching a balance between time/memory/storage overheads and matching accuracy, appropriate parameters for c-LSH and c-BF columns have been determined: LSH dimension $m = 6$, BloomFilter vector length = 300, the amount of hash functions = 1. These settings will be applied in the following experiments.

5.2.3 Construction of CFA/ACM/ACT

CFA, ACM and ACT are vital components in our proposed *enWFS* scheme. We evaluate the efficiency of respective construction processes[13] and storage cost of each structure on TOEFL and CSDN datasets in this (see Tables 4, 5 and 6) part.

In general, we can see that CFA needs the most time to be constructed, as more items should be traversed when counting frequency.

As to the storage overhead, there is no doubt that ACT occupies the most space.

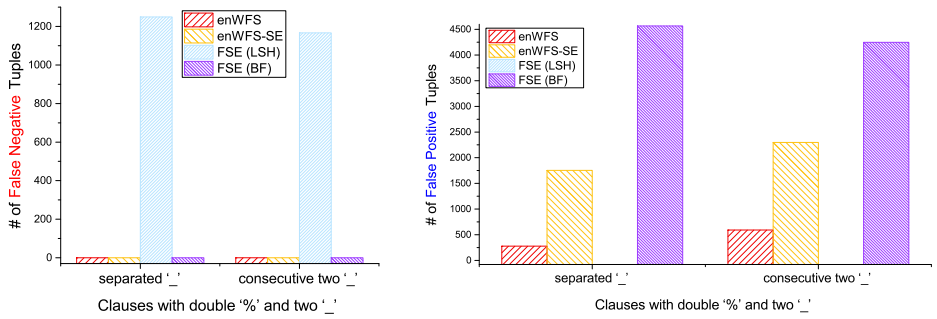
5.2.4 Performance of Advanced Adaptive Overwriting Method (AAOM)

Effectiveness of the overwriting method has direct impact on the query results. We spare no effort in this paper to improve the overwriting method, so that better accuracy of query results may be acquired. In this set of experiments, we use Reuter news dataset to measure the fuzzy searching accuracy among plaintext queries, *enWFS*, *enWFS-SE*, and *FSE*. It should be noted that which type of ancillary columns (c-LSH or c-BF) to use in the overwriting method is determined manually in *FSE*[14], while this decision is made by the algorithm itself in *enWFS* according to the flag array of 'like' clauses (see Section 4.2).

Here we will redefine two performance metrics for our experiment. On the premise that correct tuples remain in the results (*i.e.* no false negative), we can reckon a method to be better if matched rows resulting from this method are closer to matched rows from the plaintext queries (*i.e.* less false positive).

[13] We construct these three structures separately in the experiment. In fact, the construction can be conducted simultaneously in real applications.

[14] So we evaluate the *FSE* (LSH) and *FSE* (BF) respectively with regard to c-LSH and c-BF.



(a) False Negative (b) False Positive

Figure 12 Accuracy of Results (With two '_' in clauses)

- **False Negative:** On the basis of the result set K from "Query over plaintext", the number of *false negative* in another result set X means the number of missing tuples that should be in X. (*i.e.* $|\{t|t \in K, t \notin X\}|$)
- **False Positive:** On the basis of the result set K from "Query over plaintext", the number of *false positive* in another result set X means the number of redundant tuples that shouldn't be in X. (*i.e.* $|\{t|t \notin K, t \in X\}|$)

We evaluate these methods in different cases of clauses (with regard to the number of underscores) using different keywords (*e.g.* 'america', 'estate', 'manage', etc).

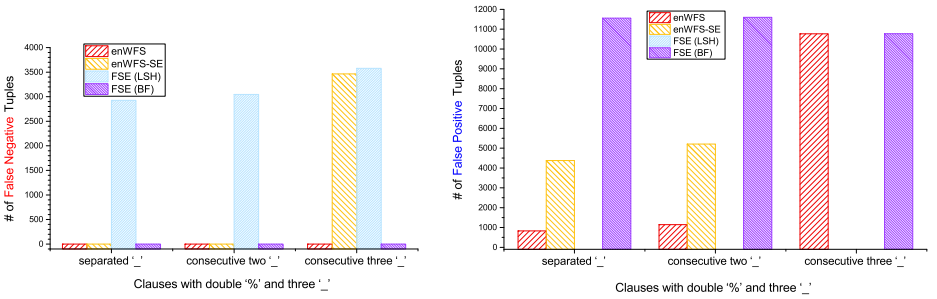
When no underscores included in the clause, all the schemes don't miss any correct tuples (see Figure 10a, *i.e.* no false negative), but *FSE* (BF) may include many redundant tuples in the results (see Figure 10b, *i.e.* false positive). Both *enWFS* and *FSE* (LSH) achieve the best accuracy.

However, when it comes to the case where one underscore is included in the clause, *enWFS* outperforms *FSE* (BF) (see Figure 11b, **62%-70% better**). Since c-LSH column is suitable for similarity searching over *unbroken words* without underscores, *FSE* (LSH) can hardly match any tuples (*i.e.* false negative) when underscores are involved in 'like' clauses[15]. This phenomenon can also be observed in the following experiments. Besides, *enWFS-SE* sacrifices much accuracy for better data privacy.

The performance gap can be owed to the *character filling* via statistical language model: CFA, ACM and ACT. *Character filling* fills '_' in *unbroken words*, so that c-LSH column may help to achieve accurate results. More obvious advantage (**86%-93% better**) can be noticed in the two-underscore case (see Figure 12).

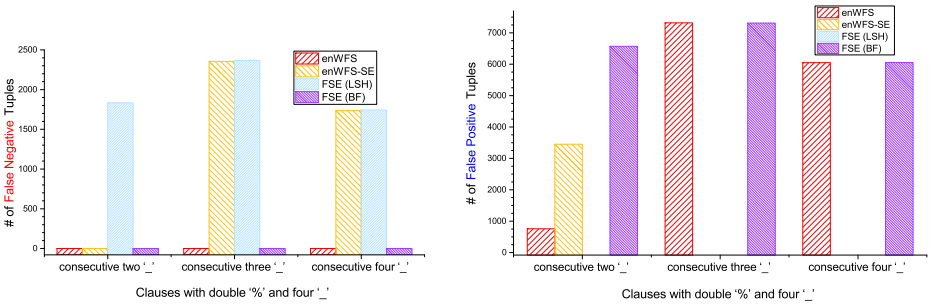
Figure 13 depicts the performance in the three-underscore case. If three separated underscores or consecutive two underscores are included in 'like' clauses (in the first two groups of bar charts), *enWFS* achieves similar outstanding accuracy (since CFA, ACM and ACT can be used to fill separated three underscores and consecutive two underscores). However, no schemes perform well (introducing massive false positive) when consecutive three underscores exist in the clause. As what has been discussed at the end of Section 4.2, only higher-dimensional ACT may bring about a complete turnabout to the performance of our method.

[15]Following the standard of good accuracy stated above the accuracy metrics, if the result set from a scheme has false negative phenomenon, we won't evaluate its # of false positive tuples anymore.



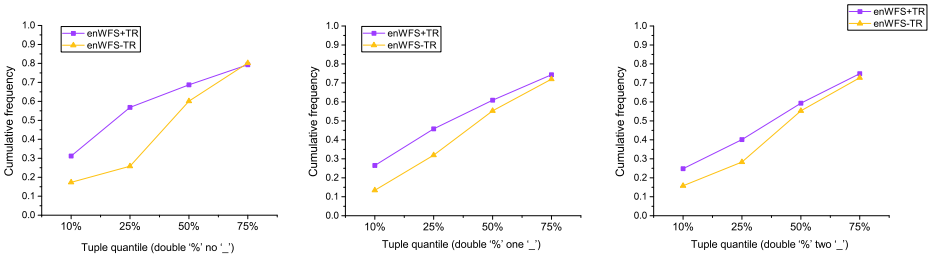
(a) False Negative (b) False Positive

Figure 13 Accuracy of Results (With three '_' in clauses)



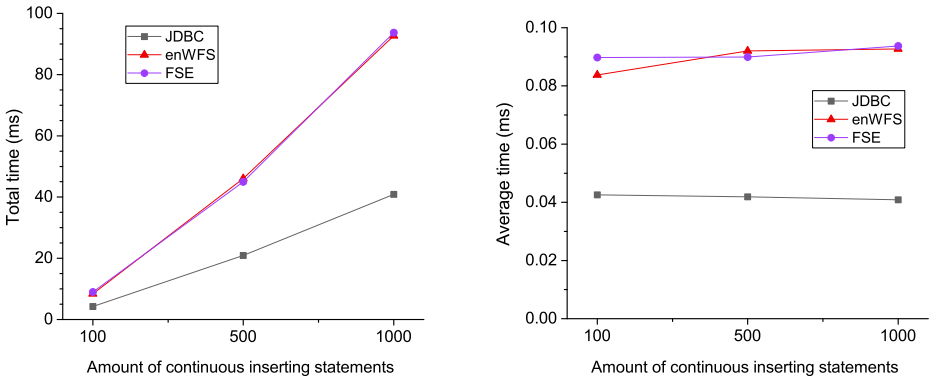
(a) False Negative (b) False Positive

Figure 14 Accuracy of Results (With four '_' in clauses)



(a) Without '_' (b) With one '_' (c) With two '_'

Figure 15 Cumulative Frequency of Tuples at different quantiles



(a) Total Time (b) Average Time

Figure 16 Efficiency of Inserting Statements

Analogously, the aforementioned conclusion can also account for the similar phenomenon (in the consecutive-three-underscore case and consecutive-four-underscore case) in Figure 14.

Furthermore, the second group of bar charts in Figure. 13b and the first group in Figure 14b confirm that with the same maximum consecutive underscores, the number of total underscores has limited influence on *character filling* and result accuracy of *enWFS* scheme.

5.2.5 Performance of TupleRank

In this part, we evaluate the effectiveness of *TupleRank*, which is put forward to improve searching experience in wildcard-based fuzzy searching. For attribute values whose corresponding column name appears in the 'like' clause, we calculate the cumulative normalized frequency at different tuple quantiles in results[16].

For example, eight tuples in results are arranged as follows:

$$(129,'aisle\ carpet'),(132,'aisle\ carpet'),(002,'apple\ juice'),(007,'apple\ juice'),$$
$$(011,'apple\ juice'),(015,'apple\ juice'),(108,'ankle\ boot'),(111,'ankle\ boot')$$

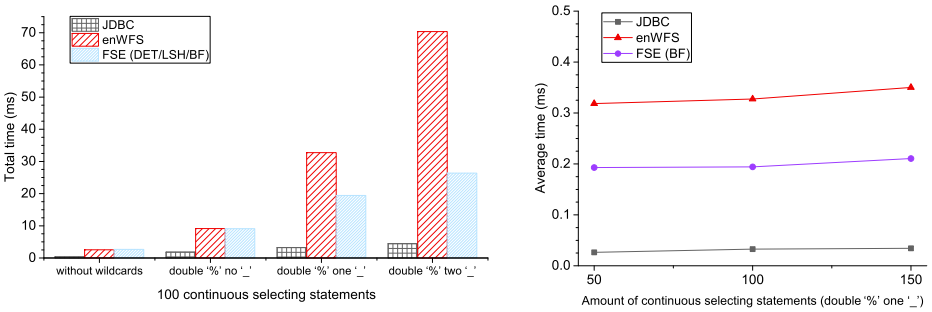
Item frequency of the second element in each tuple: 'aisle carpet': 0.00218; 'apple juice': 0.0103; 'ankle boot': 0.0023. So cumulative normalized frequency $\xi = 0.04346$ at 10% quantile, $\xi = 0.08692$ at 25% quantile, $\xi = 0.4976$ at 50% quantile, and $\xi = 0.90828$ at 75% quantile.

Figure 15 shows the performance of *TupleRank* in different wildcard cases. With *TupleRank*, tuples with high item frequency are likely to be gathered at the front part of results, which is just like how searching engines and top-k ranking work.

5.2.6 Efficiency

We evaluate the efficiency of *JDBC*, *enWFS* and *FSE* in inserting and selecting operations in the last experiment.

[16]Results from experiments in Section 5.2.4 are used here.



(a) Total Time (b) Average Time

Figure 17 Efficiency of Selecting Statements

As is shown in Figure 16a, since *enWFS* follows the same idea of encryption (LSH and BloomFilter ciphertext) as *FSE*, inserting performance of these two schemes is alike. Total encrypting time of both schemes increases linearly with the amount of inserting statements while the average time of a single inserting operation keeps steady. *JDBC* has the best efficiency because no ciphertext needs to be computed and inserted.

Efficiency in selecting operations is presented in Figure 17. When no wildcards or no underscores involved in a 'like' clause (the first & second group of bar charts in Figure 17a), *enWFS* spends almost the same time as *FSE* on 100 selecting statements because *character filling* isn't invoked.

On the other hand, if one or more underscores are involved in the 'like' clause (the third & forth group in Figure 17a), *enWFS* needs more time to accomplish the queries because more cases of clauses after *character filling*[17] should be dealt with. However, in the light of great improvement in accuracy, a bit more time is worth consuming as an acceptable compromise.

6 Conclusion

In this paper, we have a further research on the problem of wildcard-based fuzzy searching in proxy-based encrypted databases. With the boost of *Adjacent Character Matrix / Tensor*, *character filling*, advanced adaptive overwriting method (*AAOM*) effectively reduces the uncertainty within 'like' clauses and achieves satisfying accuracy of query results. Besides, *TupleRank* rearranges the order of tuples according to the item frequency, which makes the searching experience more user-friendly. In the future, more state-of-the-art techniques, such as serialization and compression of ancillary columns, will be studied to reduce storage overhead in encrypted databases.

Acknowledgments Supported by the National Key Research and Development Program of China (No. 2016YFB1000905), NSFC (Nos. 61772327, 61532021, U1501252, U1401256 and 61402180).

[17] *e.g.* If the original clause is 'a__le%', attribute values satisfying 'apple%', 'ankle%', etc, should all be retrieved.

References

1. Boldyreva, A., Chenette, N.: Efficient Fuzzy Search on Encrypted Data. In: FSE 2014 Workshop, pp. 613–633 (2014)
2. Cash, D., Jaeger, J., Jarecki, S., Jutla, C.S., Krawczyk, H., Rosu, M., Steiner, M.: Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation. In: NDSS (2014)
3. Cash, D., Jarecki, S., Jutla, C.S., Krawczyk, H., Rosu, M., Steiner, M.: Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries. In: CRYPTO 2013, Proceedings, Part I, pp. 353–373 (2013)
4. Chase, M., Kamara, S.: Structured Encryption and Controlled Disclosure. In: International Conference on the Theory and Application of Cryptology and Information Security, pp. 577–594 (2010)
5. Chen, H., Tian, X., Jin, C.: Fuzzy Searching Encryption with Complex Wild-Cards Queries on Encrypted Database. In: APWeb-WAIM 2018, pp. 3–18 (2018)
6. Curtmola, R., Garay, J., Kamara, S., Ostrovsky, R.: Searchable Symmetric Encryption:Improved Definitions and Efficient Constructions. In: ACM Conference on Computer and Communications Security, pp. 79–88 (2006)
7. Fan, K., Yin, J., Wang, J., Li, H., Yang, Y.: Multi-Keyword Fuzzy and Sortable Ciphertext Retrieval Scheme for Big Data. In: GLOBECOM 2017, pp. 1–6 (2017)
8. Fu, Z., Wu, X., Guan, C., Sun, X., Ren, K.: Toward efficient multi-keyword fuzzy search over encrypted outsourced data with accuracy improvement. IEEE Trans. Inf. Forensic. Secur. **11**(12), 2706–2716 (2017)
9. Gonzalez, L.M.V., Rodero-merino, L., Caceres, J., Lindner, M.A.: A break in the clouds: towards a cloud definition. Comput. Commun. Rev. **39**(1), 50–55 (2009)
10. Hahn, F., Kerschbaum, F.: Searchable encryption with secure and efficient updates. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, pp. 310–320 (2014)
11. Kamara, S., Papamanthou, C.: Parallel and Dynamic Searchable Symmetric Encryption. In: International Conference on Financial Cryptography and Data Security, pp. 258–274 (2013)
12. Kamara, S., Papamanthou, C., Roeder, T.: Dynamic Searchable Symmetric Encryption. In: Acm Conference on Computer and Communications Security, pp. 965–976 (2012)
13. Kurosawa, K., Ohtaki, Y.: Uc-Secure Searchable Symmetric Encryption. In: International Conference on Financial Cryptography and Data Security, pp. 285–298 (2012)
14. Kurosawa, K., Sasaki, K., Ohta, K., Yoneyama, K.: Uc-Secure Dynamic Searchable Symmetric Encryption Scheme. In: IWSEC, pp. 73–90 (2016)
15. Kuzu, M., Islam, M.S., Kantarcioglu, M.: Efficient Similarity Search over Encrypted Data. In: IEEE International Conference on Data Engineering, pp. 1156–1167 (2012)
16. Li, J., Liu, Z., Chen, X., Xhafa, F., Tan, X., Wong, D.S.: L-encdb: a lightweight framework for privacy-preserving data queries in cloud computing. Knowl.-Based Syst. **79**, 18–26 (2015)
17. Liu, Z., Li, J., Li, J., Jia, C., Yang, J., Yuan, K.: Sql-based fuzzy query mechanism over encrypted database. Int. J. Data Warehous. Min. (IJDWM) **10**(4), 71–87 (2014)
18. Liu, Z., Ma, H., Li, J., Jia, C., Li, J., Yuan, K.: Secure Storage and Fuzzy Query over Encrypted Databases. In: International Conference on Network and System Security, pp. 439–450 (2013)
19. Popa, R.A., Li, F.H., Zeldovich, N.: An ideal-security protocol for order-preserving encoding. IEEE Symposium on Security and Privacy, pp. 463–477 (2013)
20. Popa, R.A., Redfield, C.M.S., Zeldovich, N., Balakrishnan, H.: Cryptdb: Protecting Confidentiality with Encrypted Query Processing. In: ACM SOSP, pp. 85–100 (2011)
21. Song, D.X., Wagner, D.A., Perrig, A.: Practical Techniques for Searches on Encrypted Data. In: IEEE Symposium on Security and Privacy, pp. 44–55 (2000)
22. Tetali, S.D., Lesani, M., Majumdar, R., Millstein, T.D.: Mrcrypt: Static Analysis for Secure Cloud Computations. In: ACM SIGPLAN OOPSLA, pp. 271–286 (2013)
23. Wang, B., Yu, S., Lou, W., Hou, Y.T.: Privacy-Preserving Multi-Keyword Fuzzy Search over Encrypted Data in the Cloud. In: IEEE INFOCOM, pp. 2112–2120 (2014)
24. Wang, C., Cao, N., Li, J., Ren, K.: Secure Ranked Keyword Search over Encrypted Cloud Data. In: IEEE International Conference on Distributed Computing Systems (ICDCS), pp. 253–262 (2010)

25. Wang, J., Ma, H., Tang, Q., Li, J., Zhu, H., Ma, S., Chen, X.: Efficient verifiable fuzzy keyword search over encrypted data in cloud computing. Comput. Sci. Inf. Syst. **10**(2), 667–684 (2013)
26. Wang, Q., He, M., Du, M., Chow, S.S.M., Lai, R.W.F., Zou, Q.: Searchable encryption over feature-rich data. IEEE Trans. Dependable Sec. Comput. **15**(3), 496–510 (2018)
27. Wei, X., Zhang, H.: Verifiable Multi-Keyword Fuzzy Search over Encrypted Data in the Cloud. In: International Conference on Advanced Materials and Information Technology Processing, pp. 271–277 (2016)