Fast Searchable Encryption With Tunable Locality

Ioannis Demertzis
University of Maryland
yannis@umd.edu

Charalampos Papamanthou
University of Maryland
cpap@umd.edu

ABSTRACT

Searchable encryption (SE) allows a client to outsource a dataset to an untrusted server while enabling the server to answer keyword queries in a private manner. SE can be used as a building block to support more expressive private queries such as range/point and boolean queries, while providing formal security guarantees. To scale SE to big data using external memory, new schemes with small *locality* have been proposed, where locality is defined as the number of non-continuous reads that the server makes for each query. Previous space-efficient SE schemes achieve optimal locality by increasing the *read efficiency*—the number of additional memory locations (false positives) that the server reads per result item. This can hurt practical performance.

In this work, we design, formally prove secure, and evaluate the first SE scheme with tunable locality and linear space. Our first scheme has optimal locality and outperforms existing approaches (that have a slightly different leakage profile) by up to 2.5 orders of magnitude in terms of read efficiency, for all practical database sizes. Another version of our construction with the same leakage as previous works can be tuned to have bounded locality, optimal read efficiency and up to 60× more efficient end-to-end search time. We demonstrate that our schemes work fast in in-memory as well, leading to search time savings of up to 1 order of magnitude when compared to the most practical in-memory SE schemes. Finally, our construction can be tuned to achieve trade-offs between space, read efficiency, locality, parallelism and communication overhead.

1. INTRODUCTION

Searchable Encryption (SE) enables a data owner to outsource a document collection to a server in a private manner, so that the latter can still answer keyword search queries. In a typical SE scheme, the data owner prepares an encrypted index which is sent to the server. To perform a keyword search given a keyword w, a token $t(w)$ is sent by the data owner to the server that allows him to retrieve pointers to those encrypted documents containing the keyword w, while leaking some information, e.g., the *access patterns* (encrypted documents that satisfy the query) and the search patterns (whether two encrypted queries are the same). An alternative is to use very expensive approaches such as oblivious RAM [17, 25, 29] and fully-homomorphic encryption [14, 15]. However, SE schemes have proven to be very practical at the expense of well-defined leakage. SE was primarily used for private keyword search but can also be used for database search, e.g., point queries. Recent works [7, 12, 13] used SE for range search and more expressive queries which are primarily used in databases. In particular, Demertzis et al. [12] reduced the problem of range search to multi-keyword search using any SE scheme as a black box illustrating the importance of SE in private databases; any advances in SE directly impacts those works.

In contrast to the above cryptographic solutions with rigorous security guarantees, several schemes and entire encrypted database systems have been proposed (mainly in database venues) achieving the desired performance at the cost of more leakage. CryptDB [26] and Monomi [31] utilized *deterministic* and *order preserving encryption*[1] in order to support point/range queries and joins. The aforementioned works achieve very practical performance but it was recently shown that they are susceptible to various attacks [24], for example the leaked statistical and order information allowed recovering the actual patient records in plaintext.

Prior SE Schemes. Since the first work on SE was proposed in 2000 [27], most follow-up works considered scenarios where the encrypted index could fit in memory. However, for very large indexes and databases that must be stored on disk (e.g., see the CW-MC-OXT-4 dataset from the recent work of Cash et al. [6] whose encrypted index had size around 904 GB), these in-memory schemes cannot scale since random access is expensive. In these scenarios, the practical performance of SE schemes depends on the *locality*, namely the number of non-continuous locations that the server accesses for each query. Most SE schemes have poor locality (see the first five rows of Table 1), accessing one random location per result item—and this random allocation of results in memory is necessary for achieving the desired security.

The work of Cash et al. [6] experimentally showed that in-memory SE cannot scale to large datasets, and therefore proposed new SE schemes with good locality guarantees. While trying to reduce[2] locality, it was observed that a number of additional entries per query must be read (usually referred

SIGMOD'17, May 14–19, 2017, Chicago, IL, USA.

DOI: http://dx.doi.org/10.1145/3035918.3064057

[1] *Deterministic encryption* leaks the distribution of the input data. *Order preserving encryption* leaks the distribution of the input data and their order.

[2] Cash and Tessaro [8] and Asharov et al. [4] define locality as the number of non-continuous reads that the server makes per result item. Larger locality implies more non-continuous reads. Throughout this paper we follow this notation and by reducing locality we mean improving locality, therefore $O(1)$ locality means optimal locality.

Scheme	Locality	Read Efficiency	Server Storage
Kamara et al. [19]	$\Theta(w)$	$O(1)$	$\Theta(N+m)$
Curtmola et al. [11], Liesdonk et al. [32]	$\Theta(w)$	$O(1)$	$\Theta(N \cdot m)$
Kamara et al. [18]	$O(w \log N)$	$O(\log N)$	$\Theta(N \cdot m)$
Cash et al. [6] and [7]	$\Theta(w)$	$O(1)$	$\Theta(N)$
Stefanov et al. [28]	$O(w \log^3 N)$	$O(\log^3 N)$	$\Theta(N)$
Chase et al. [9]	$O(1)$	$O(1)$	$\Theta(N \cdot m)$
Cash et al. [8]	$O(\log N)$	$O(1)$	$\Theta(N \cdot \log N)$
Asharov et al. [4] (Scheme 1)	$O(1)$	$O(1)$	$\Theta(N \cdot \log N)$
Asharov et al. [4] (Scheme 2)	$O(1)$	$\Theta(\log N \log\log N)$	$\Theta(N)$
Asharov et al. [4] (Scheme 3)	$O(1)$	$\Theta(\log\log N \log\log\log N)^*$	$\Theta(N)$
Our scheme with optimal locality	$O(1)$	$O(N^{1/(s+1)})$	$\Theta(N \cdot s)$
Our scheme with $O(L)$ locality	$O(L)$	$O(N^{1/s}/L)$	$\Theta(N \cdot s)$
Lower bound [8]	$O(1)$	$O(1)$	$\omega(N)$

Table 1: Comparison of the most representative SE schemes. We denote with N the number of keyword-document pairs, with m the number of unique keywords and with w the size of the result of a keyword search query. Our schemes can be parameterized in terms of locality L. Our most practical scheme is achieved by setting $L = 1$, yielding read efficiency $O(N^{1/(s+1)})$ and space equal to $\Theta(N \cdot s)$. Note that for $L = N^{1/s}$ (which gives constant read efficiency), we can prove our scheme is secure using as leakage only the size of the access pattern (as used in previous works). *Assuming no keyword list has size more than $N^{1-1/\log\log N}$.

to as false positives). The ratio of the total number of entries read over the size of the initial query result was defined as *read efficiency*. Soon after, Cash and Tessaro [8] presented, along with a lower bound, a scheme that requires $\Theta(N \log N)$ space, $O(\log N)$ locality and optimal $O(1)$ read efficiency, where N is the number of document-keyword pairs in the document collection. Finally, Asharov et al. [4] recently presented three schemes: One with $\Theta(N \log N)$ space and optimal read efficiency and optimal locality (Scheme 1 in Table 1) and two other schemes with linear space, optimal locality and very small (asymptotically) read efficiency (Scheme 2 and Scheme 3 in Table 1).[3]

Among the above schemes, Scheme 1 is the fastest in practice (no false positives and no random access) and would be the scheme of choice if one could afford to store $\Theta(N \log N)$ space. However, this can be quite an overkill. For example, for the the CW-MC-OXT-4 dataset [6] whose encrypted index has size around 904 GB and 2,732,311,945 entries, this would mean storing around 28.3 TB. In this paper, we focus on SE schemes with good locality guarantees that occupy *linear space*. Such schemes are, for example, Schemes 2 and 3. While these works achieve close to optimal read efficiency from a theoretical point of view, they introduce a significant practical overhead, as we detail in the following.

Our Contributions. In this paper we propose a novel SE scheme for private keyword and database search with tunable locality. We formally prove the security of our scheme and we also present a thorough description of the implementation and evaluation of our scheme in external and internal memory settings. For a parameter s that controls the space (the space is $s \cdot N$), our most efficient scheme has $O(1)$ locality and $O(N^{1/(s+1)})$ read efficiency—see Line 11 of Table 1. In particular:

1. Our scheme achieves up to 577× less false positives for all practical database sizes when compared to Scheme 2 in a large database (approximately 1 TB) (see Figure 10(b)). This translates into big improvements in practical performance in an external memory setting. We stress here that our scheme's leakage profile is slightly different (not necessarily worse) than Scheme 2 (see discussion on leakage in Section 3.4) and its worst-case asymptotic read efficiency is worse than Scheme 2. There are two reasons we are better in practice despite worse asymptotics: First, due to hidden constants in the polylogarithmic complexity of Scheme 2. Second, and most importantly, our asymptotic complexity is *worst-case* (meaning that there are some queries that we answer optimally or close to optimally) while Scheme's 2 complexities are tight (meaning that there are no queries that are answered optimally)[4].

2. Our scheme is designed to work fast in memory as well, where the bottleneck is not random memory access but computation. In particular it achieves up to 12× speed-up compared to the state-of-the-art in-memory SE scheme by Cash et al. [6][5] (that has optimal read efficiency but bad locality) and up to 347× speed-up compared to Scheme 2 in an in-memory setting (see Figure 7(a)). The speed-ups in the in-memory setting are due to the fact that our scheme substantially reduces the number of PRF (Pseudo Random Function) evaluations required to retrieve the result. In particular, PRFs are the main building block for an SE; they are used for the generation of the encrypted index. Previous SE schemes answer a keyword search query

[3]From now on, we will be referring to the schemes by Asharov et al. [4] simply as Scheme 1, Scheme 2 and Scheme 3.

[4]We do not directly compare to Scheme 3 since it is based on an unrealistic assumption, i.e., that no keyword list has size greater than $N^{1-1/\log\log N}$ (if we also have this assumption, our scheme still compares favorably, see Appendix for a complete analysis).

[5]We only compare with the most basic scheme of Cash et al. [6], since the other ones that have good locality have been proposed for external memory being sub-optimal compared with the schemes of Asharov et al. [4] and introduce more leakage.

by evaluating at least as many PRFs as the result size, in order to find the memory locations that the server has to read. Instead, our SE scheme requires only a constant number of PRF evaluations, irrespective of the size of the result and the read efficiency.

3. Our scheme can be also tuned to achieve locality L and improved read efficiency $O(N^{1/s}/L)$—see Line 12 of Table 1. This is quite important in a parallel setting with L processing units. Using this tuning, each processor can have optimal locality and $O(N^{1/s}/L)$ read efficiency. We highlight this trade-off is non-trivial. The trivial bound would be $O((N/L)^{1/s})$, achieved when one partitions the data set into L pieces of N/L entries and subsequently applies the original optimal locality scheme. In the experimental evaluation, we show that this allows to have always less false positives than Scheme 2 even for $N = 2^{47} - 1$ (an unencrypted database with size 1.12 petabytes — see Figure 11(b)). Interestingly enough, for the case where $L = N^{1/s}$ (that gives optimal read efficiency), our scheme has exactly the same leakage profile as previous work.

2. BACKGROUND

2.1 Prior Work on Other SE Schemes

We discussed prior work on SE schemes with optimal locality guarantees in the introduction section. Here we refer to other related SE work. Song et al. [27] presented the first SE scheme, secure under Chosen Plaintext Attacks (CPA). Later, Goh [16] explained why CPA security is not adequate for the case of SE schemes. Curtmola et al. [11] introduced the state-of-the-art security definitions for SE both in non-adaptive settings, i.e., maintaining security only if all the queries are submitted at once in one batch) and in adaptive settings, i.e., maintaining security even if the queries are progressively submitted, and provided constructions that satisfy their definitions (we use the latter definition in this paper). Following the work of Curtmola et al. [11] several efficient schemes were proposed [19, 28, 18, 7, 23, 5], some of which support updates [19, 28, 18, 23, 5], are parallelizable [18] and support more complex queries [7]. The above works constitute the first generation of SE schemes that provide linear space solutions with optimal read efficiency, but achieve poor locality and can only be used for data that fit in memory. Other such SE schemes include [9, 32, 21] which have significant space overhead, rendering them impractical for both in-memory and external memory models.

2.2 Preliminaries

Negligible Function. A function ν: $\mathbb{N} \rightarrow \mathbb{N}$ is negligible in λ, $negl(\lambda)$, if for every positive polynomial $p(\cdot)$ and all sufficiently large λ, $\nu(\lambda) < 1/p(\lambda)$.

Randomized Encryption (RND). We refer to a randomized symmetric encryption scheme with three polynomial-time algorithms as RND, i.e., *RND = (Gen, Enc, Dec)*, such that *Gen* takes as input a security parameter λ and returns a secret key k, *Enc* takes as an input a secret key and a message and outputs a ciphertext and *Dec* takes as an input the secret key k and a ciphertext and outputs the message that was encrypted. An RND scheme is secure against chosen-plaintext attacks (CPA) if the ciphertexts do not reveal any information about the plaintext even if the adversary can observe the encryption of the messages of his choice. For a formal definition see [20].

Pseudo Random Functions (PRFs). A PRF function F: $\{0,1\}^* \times \{0,1\}^* \rightarrow \{0,1\}^*$ is a two-input function, where the first input is called the *key* and the second is the input x. F can be distinguished from a truly random function only with negligible probability in λ, denoted as $negl(\lambda)$. For a formal definition see [20].

Collision-Resistant Hash Function. H is a collision-resistant hash function if two inputs a and b have the same $H(a) = H(b)$ with negligible probability. For a formal definition see [20].

SE Definition. Let $\mathcal{D}$ be a collection of *documents*. Each document $D \in \mathcal{D}$ is assigned with a unique document identifier and contains a set of keywords from a dictionary Δ. We use the notation $\mathcal{D}(w)$ to denote the document identifiers that contain a keyword w. SE schemes focus on building an *encrypted index* $\mathcal{I}$ on the document identifiers. For simplicity, we only consider the document identifiers instead of the actual documents since these are encrypted independently and stored in the server separately from the encrypted index $\mathcal{I}$; whenever the client retrieves a specific identifier during a search, he can send it to the server in an extra round and the server can send the corresponding document back. Finally, N is the data collection size, i.e., $N = \sum_{\forall w \in \Delta} |\mathcal{D}(w)|$. An SE *protocol* considers two parties, a *client* and a *server* and consists of the following algorithms [11]:

- $k \leftarrow \mathsf{KeyGen}(1^\lambda)$: is a probabilistic algorithm run by the client. It receives as input a security parameter λ and outputs a secret key k.
- $\mathcal{I} \leftarrow \mathsf{Setup}(k, \mathcal{D})$: is a probabilistic algorithm run by the client prior to sending any data to the server. It receives as input a secret key k and the data collection $\mathcal{D}$, and outputs an encrypted index $\mathcal{I}$. Index $\mathcal{I}$ is sent to the server.
- $t \leftarrow \mathsf{Token}(k, w)$: is a deterministic algorithm executed by the client when issuing a query. It receives as input a secret key k and a keyword w, and outputs a token t.
- $\mathcal{X} \leftarrow \mathsf{Search}(t, \mathcal{I})$: is a deterministic algorithm run by the server. It receives as input a token t corresponding to the query keyword and the encrypted index $\mathcal{I}$, and outputs a set $\mathcal{X}$ of document identifiers. In the case of database search the set $\mathcal{X}$ contains the encrypted result.

Security Definition and Leakage Functions. Figure 1 presents the ideal and real games for (semi-honest) adaptive adversaries, as introduced in [10]. These games are used to formally prove the security of an SE scheme. They are partitioned into two worlds, the real and the ideal one. The real world depicts a real SE scheme, where the adversary has access to the Setup and Token algorithms. More specifically, the real scheme creates a secret key to which the adversary does not have access. The adversary selects a document collection which is given as an input to the Setup algorithm.

$\underline{\mathbf{Real}(\lambda)}$	$\underline{\mathbf{Ideal}_{\mathcal{L}_1,\mathcal{L}_2}(\lambda)}$
$k \leftarrow \mathsf{KeyGen}(1^\lambda)$	
$(\mathcal{D}, st_\mathcal{A}) \leftarrow \mathcal{A}(1^\lambda)$	$(\mathcal{D}, st_\mathcal{A}) \leftarrow \mathcal{A}(1^\lambda)$
$\mathcal{I} \leftarrow \mathsf{Setup}(k, \mathcal{D})$	$(\mathcal{I}, st_\mathcal{S}) \leftarrow \mathsf{SimSetup}(\mathcal{L}_1(\mathcal{D}))$
for $1 \le i \le q$	for $1 \le i \le q$
$(w_i, st_\mathcal{A}) \leftarrow \mathcal{A}(st_\mathcal{A}, \mathcal{I}, t_1, \ldots, t_{i-1})$	$(w_i, st_\mathcal{A}) \leftarrow \mathcal{A}(st_\mathcal{A}, \mathcal{I}, t_1, \ldots, t_{i-1})$
$t_i \leftarrow \mathsf{Token}(k, w_i)$	$(t_i, st_\mathcal{S}) \leftarrow \mathsf{SimToken}(st_\mathcal{S}, \mathcal{L}_2(\mathcal{D}, w_i))$
let $\mathbf{t} = (t_1, \ldots, t_q)$	let $\mathbf{t} = (t_1, \ldots, t_q)$
output $v = (\mathcal{I}, \mathbf{t})$ and $st_\mathcal{A}$	output $v = (\mathcal{I}, \mathbf{t})$ and $st_\mathcal{A}$

Figure 1: SE ideal-real security experiments.

Furthermore, $st_\mathcal{A}$ denotes a state maintained by the adversary. The adversary observes the output of the Setup algorithm which is the encrypted index. Then, the adversary is able to select a polynomial number of queries, and for each of these queries he observes the corresponding token. Having this token allows him to retrieve the encrypted result. In the ideal world, the adversary interacts with the simulator. The simulator $\mathcal{S}$ does not have access to the real document collection or the real queries. Instead, the simulator only has access to predetermined leakage functions and by using these leakage functions and his state he attempts to "fake" the algorithms Setup and Token. We consider only the strongest types of adversaries, i.e., adaptive adversaries that can select their own new queries based on previous ones. The adversary tries to distinguish in which world he has access to. We prove that an adversary can distinguish the output of the real world from that of the ideal world only with negligible probability. This means that an adversary does not learn anything else, but the predefined leakage. We refer the reader to [10, 12] for a more detailed explanation of the security game.

As is common in SE definitions, we use two leakage functions, $\mathcal{L}_1$ and $\mathcal{L}_2$. $\mathcal{L}_1$ is associated with what is leaked from the index alone, which means what is leaked prior to the query execution), whereas $\mathcal{L}_2$ represents the leakage produced by the queries (during the query execution). In particular

$$\mathcal{L}_1(\mathcal{D}) = N$$

is the *size pattern*, where $N = \sum_{\forall w \in \Delta} |\mathcal{D}(w)|$. Namely $\mathcal{L}_1$ leaks just the size of the index. Also

$$\mathcal{L}_2(\mathcal{D}, w) = (id(w), \mathcal{D}(w))$$

is the *access pattern* leaking the identifiers of documents matching the query for keyword w, as well as a deterministic function of the keyword w, $id(w)$. The history of $\mathcal{L}_2$ leakage also defines the *search pattern* leakage, which leaks whether two queries are the same.

As mentioned before, we can use an SE scheme for database search by corresponding the notion of document identifiers to tuple identifiers or encrypted tuples (encrypted using RND), and the notion of keywords to searchable attributes. In the case of database search, the $\mathcal{L}_1$ leakage is identical and corresponds to the number of tuples. The $\mathcal{L}_2$ leakage in the database search differs from the previous case because it only contains the size of the encrypted results or similarly the size of the access pattern as shown below. We refer to this leakage as $\mathcal{L}_2^{DB}$

$$\mathcal{L}_2^{DB}(\mathcal{D}, w) = (id(w), |\mathcal{D}(w)|)$$

We consider the two cases separately because when performing database search the leakage is considered less significant than the keyword search problem. In particular, SE in database search achieves leakage that is very close to the optimal one achieved by ORAMs. Their difference is that SE additionally leaks the search pattern. Finally, database search leaks less information compared to keyword search due to structural difference. In keyword search, one document identifier can be included in multiple keywords, while in database search one tuple-id or an encrypted tuple can have exactly one searchable value per attribute. For example, a patient cannot have more than one date of birth.

Definition 1. *Let* (KeyGen, Setup, Token, Search) *be an SE scheme as defined before, let* $\lambda \in \mathbb{N}$ *be the security parameter and consider experiments* $\mathbf{Real}(\lambda)$ *and* $\mathbf{Ideal}_{\mathcal{L}_1,\mathcal{L}_2}(\lambda)$ *presented in Figure 1, where* $\mathcal{L}_1$, $\mathcal{L}_2$ *are leakage functions defined above. We say that the SE scheme is* $(\mathcal{L}_1, \mathcal{L}_2)$*-secure if for all polynomial-size adversaries* $\mathcal{A}$ *there exists polynomial-time simulators* SimSetup *and* SimToken*, such that for all polynomial-time algorithms* Dist*:*

$$\begin{aligned} &|\Pr[\mathsf{Dist}(v, st_\mathcal{A}) = 1 : (v, st_\mathcal{A}) \leftarrow \mathbf{Real}(\lambda)] - \\ &\Pr[\mathsf{Dist}(v, st_\mathcal{A}) = 1 : (v, st_\mathcal{A}) \leftarrow \mathbf{Ideal}_{\mathcal{L}_1,\mathcal{L}_2}(\lambda)]| \\ &\le negl(\lambda)\,, \end{aligned}$$

where the probabilities are taken over the coins of KeyGen *and* Setup*.*

The above security definition applies only to static SE schemes. The extension of static SE schemes to a dynamic setting requires guaranteeing a property called forward privacy [33]; the server does not get to learn that a newly inserted keyword,id pair satisfies a query issued in the past.

Locality, Read Efficiency and the Lower Bound. Cash and Tessaro [8] present formal definitions of *locality* and *read efficiency* in searchable encryption. Intuitively, locality is the number of non-continuous accesses made by the server to retrieve the query result. Moreover, read efficiency is the ratio of the information read when retrieving the query result over the result itself. Finally, Cash et al. proved in [6] that in any secure SE scheme both optimal locality and optimal read efficiency while using $O(N)$ storage cannot be achieved. Briefly, the intuition behind the lower bound is that if a scheme has optimal locality and linear space and the server has observed the location of some queries, then he can look at the non-accessed locations to infer statistical information about the input dataset.

3. OUR NEW SE CONSTRUCTIONS

We now describe the algorithms of our new scheme. Our main scheme in Figures 4 and 5 and has optimal locality, $O(N^{1/s})$ read efficiency and space $O(s \cdot N)$ by setting $L = 1$. In the optimizations that we describe later in Section 3.3 we will show how to further reduce the above to $O(N^{1/(s+1)})$. We also show in Section 3.2 how to adjust our scheme to have locality $O(L)$ and read efficiency $O(N^{1/s}/L)$. From section 3.1 to section 3.2, we focus only on the keyword search scenario; database search was described in section 2.2 and will be presented again in the optimization 3.3 and security analysis 3.4 sections.

3.1 Scheme with Optimal Locality

Our core scheme is inspired by the scheme of Asharov et al. [4], but is different is many ways. Asharov et al. proposed a scheme with optimal locality, optimal read efficiency and $O(N \log N)$ space. This scheme roughly works as follows: It uses $\ell = \log N + 1$ arrays $A_0, A_1, \ldots, A_\ell$[6] of size N. Array A_i consists of $N/2^i$ chunks and stores all keyword lists of size 2^i at randomly-chosen chunks (note it is assumed here that all keyword list sizes are powers of two—we do not have this assumption). This ensures that the size of data read from each array A_i is always the same, which is important for security. Therefore, to retrieve the results for a certain keyword w, one needs to read the right bucket at level i that contains the list. This bucket number is stored in an encrypted form in a separate dictionary and can be retrieved using the token for the keyword w. It is easy to show that such an approach reaches the aforementioned bounds.

Our main idea is to reduce the space of the above scheme by storing only s evenly distributed levels, where s is a small constant in practice (e.g., in our experiments we set $s = 2$ or 4). In particular we pick $p = \lceil \ell/s \rceil$ and we store only the levels $\mathcal{L} = \{\ell, \ell - p, \ell - 2p, \ldots, \ell - (s-1) \cdot p\}$ .This however creates many issues. For example, if level i is not stored, then the queries of size 2^i can no longer be answered. To avoid this problem we choose to store at level $i \in \mathcal{L}$ keyword lists $\mathcal{D}(w)$ such that

$$2^j < |\mathcal{D}(w)| \leq 2^i \,,$$

where $j \in \mathcal{L}$ is the smaller level following i in $\mathcal{L}$. (We stress that if i is the smallest level we ignore the lower bound in the above relation.) To store a keyword list whose size falls in the above range, we pick a random bucket at level i that has enough space (we never split a keyword list across two buckets). While this looks like an easy fix, it creates further problems as we detail in the following paragraph.

In particular, we can no longer guarantee that *all* keyword lists with sizes $(2^j, 2^i]$ can fit in a single bucket at level i, which is important for maintaining our optimal locality. This is because, depending on the order that we store the keyword lists, there might be one that will have to reside in different buckets, ruining our optimal locality. For example, assume two consecutive levels that we store are levels 1 and 3 and the total number of elements we have is $N = 16$. There are three keywords in our data set w_1, w_2 and w_3, with $|\mathcal{D}(w_1)| = |\mathcal{D}(w_2)| = 4$, and $|\mathcal{D}(w_3)| = 8$. All these lists will be stored at level 3, which has two buckets of size 8. If we choose to store w_1 and w_2 in different buckets, then w_3 will have to be divided across the two buckets, increasing its locality from 1 to 2. This also affects the security of the scheme since there exist inputs that could trigger the aforementioned overflow and others that could not.

We address this problem by slightly increasing the space of each level $i \in \mathcal{L}$ from N to $2N + 2^{i+1}$—see Lemma 1. In particular, we are doubling the size of each bucket in each level and adding one more bucket per level. This allows us to guarantee that regardless of the order of the input keyword lists there will always be enough space to store an entire keyword list in one bucket.

LEMMA 1. *Assume level i can store $2N + 2^{i+1}$ entries and let $\mathcal{W}$ be the set of keywords with list sizes $\leq 2^i$. Regardless of the order in which we store keywords $w \in \mathcal{W}$ at level i, there is always going to be enough space within a single bucket (of size 2^{i+1}) for all keywords $w \in \mathcal{W}$.*

PROOF. Let level i be split in at most $\Lambda + 1$ buckets, so

$$2N + 2^{i+1} = \Lambda \cdot 2^{i+1} + y \,, \tag{1}$$

where $0 \leq y < 2^{i+1}$ is the size of the last bucket. We prove our claim by contradiction. Suppose there exists a keyword $w \in \mathcal{W}$ whose list has size $\kappa \leq 2^i$ and for which there is not enough space in any bucket of level i. This means that all Λ buckets in level i have been filled with at least $2^{i+1} - \kappa + 1$ items and the last bucket has been filled with at least $y - \kappa + 1$ items. In that case, if we count the number of items that have been considered so far we have

$$\begin{aligned}
\#\text{ items considered} &\geq \Lambda \cdot (2^{i+1} - \kappa + 1) + y - \kappa + 1 \\
&= \frac{2N + 2^{i+1} - y}{2^{i+1}} \cdot (2^{i+1} - \kappa + 1) + y - \kappa + 1 \\
&\geq \frac{2N + 2^{i+1} - y}{2^{i+1}} \cdot (2^i + 1) + y - 2^i + 1 \text{ (since } \kappa \leq 2^i) \\
&= N + \frac{N}{2^i} + 2 + y - \frac{y}{2} - \frac{y}{2^{i+1}} \\
&\geq N \text{ (since } i \geq 0) \,.
\end{aligned}$$

Therefore we show that the total number of items considered so far is at least N, which is a contradiction. □

The arrangement of the mappings in our scheme is shown in Figure 2. In particular, we present two cases, where in both $N = 64$: (1) We keep all the levels by setting s=7 and store all the keyword-lists of size $|\mathcal{D}(w)|$ in level $\lceil \log |\mathcal{D}(w)| \rceil$. (2) We set $s = 2$ and follow our algorithm which keeps only levels 3 and 6. In the latter case, all keyword-lists of size less than or equal to 8 are mapped to level 3 and the remaining ones to level 6.

Complexities. Clearly the above approach answers queries with optimal locality. Also, the read efficiency is $O(2^{\log N/s}) = O(N^{1/s})$. To see that, note that the maximum penalty in terms of false positives is paid by keywords lists $\mathcal{D}(w)$ with size $2^j + 1$ which are answered by the buckets of size 2^i (this is in case $i > \ell - (s-1) \cdot p$, i.e., i is not the last level). Therefore, by the definition of read efficiency

$$R \leq \frac{2^i}{2^j + 1} < 2^{i-j} \leq 2^{\lceil \lceil \log N \rceil / s \rceil} = O(2^{\log N/s}) = O(N^{1/s}) \,.$$

For $i = \ell - (s-1) \cdot p$ we have $R \leq 2^i$ but since $\ell - (s-1) \cdot p \leq p$ we have the same bound. The space of this approach is $O(s \cdot N)$.

[6]The actual scheme uses hash tables instead of arrays but we use arrays here for clarity.

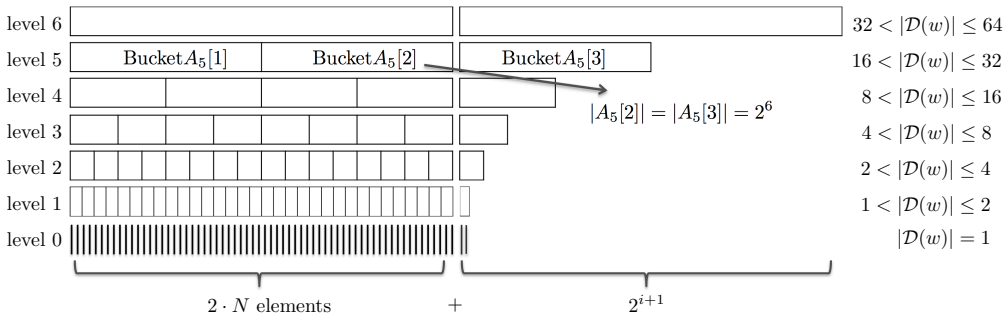


Figure 2: Example for $N = 64$ and $s = 7$. When $s = 2$, our scheme with optimal locality stores only levels 6 and 3, mapping all the queries of levels $0, 1, 2$ to level 3 and all the queries of levels $4, 5$ to level 6. The worst case read efficiency is $N^{1/2} = 8$ occurring when we map a query of size 1 to level 3.

3.2 Tuning the Locality of Our Scheme

Our scheme above achieves optimal locality. However, there are scenarios that we might want to increase slightly the locality to gain in read efficiency—this could be a parallel processing setting. One naive way to increase locality from 1 to L so that to gain in read efficiency is to partition our original data set into N/L data sets and apply our optimal locality scheme separately on each one of the smaller datasets. By using our previously described scheme, this would yield $O((N/L)^{1/s})$ read efficiency (actually, this approach of data partitioning can work for every locality-optimal scheme). We propose here a new scheme with read efficiency $O(N^{1/s}/L)$ and locality L. This is much better in practice, and asymptotically better for any $L = \omega(1)$. The idea is as follows:

In our previous scheme, we chose to store at level i lists that have sizes in $(2^j, 2^i]$, where i and j are adjacent levels in $\mathcal{L}$ with $i > j$. To achieve locality L, we can choose to store at level i keyword lists $\mathcal{D}(w)$ such that

$$L \cdot 2^j < \mathcal{D}(w) \leq L \cdot 2^i \,.$$

(Again, if i is the smallest level we ignore the lower bound in the above relation.) Among those lists, the ones with size $\leq 2^i$ are stored as in the previous scheme; The ones with size $> 2^i$ are split into multiple chunks of size 2^i and one chunk of size less than 2^i. Then these chunks are stored as before. Note that because we again end up storing chunks of size $\leq 2^i$ at level i, Lemma 1 can be recast and still holds, guaranteeing that even with this new, modified algorithm, there will always be a bucket with enough space to store the relevant chunks.

Complexities. The locality for keyword lists of size greater than 2^i is at most L, while the read efficiency for those lists is optimal. For keyword lists of size less or equal to 2^i, the maximum penalty is achieved for the list of size $L \cdot 2^j + 1$, in which case the read efficiency is

$$R \leq \frac{2^i}{L \cdot 2^j + 1} < \frac{2^{i-j}}{L} = O\left(\frac{N^{1/s}}{L}\right) .$$

Again, the above holds for $i > \ell - (s-1) \cdot p$. As opposed to before, for $i = \ell - (s-1) \cdot p$ the above, improved bound does not hold (in particular it is $O(N^{1/s})$), since keyword lists with size 1 must be answered by level i. To avoid that, we also keep level 0, which answers keyword lists with size $\leq L$. See Line 1 of Algorithm Setup. Only the size of the array A_0 can be N, instead of $2N + 1$.

The space remains $O(s \cdot N)$. Note our initial scheme with optimal locality is a special case of the above scheme for $L = 1$. The detailed algorithms of our schemes are shown in Figures 4 and 5. Figure 3 illustrates an example for $s = 2$ and $L = 2$. Note that algorithm Setup takes as input the parameters L (locality) and s (number of levels kept). We observe that our scheme keeps levels 0,3,6. The red arrows illustrate that these queries will be answered by the level above (including false positives), while the blue arrows introduce our new policy. In our new policy, given a stored level i, the $\log L$ levels above it will be stored and answered by the level i. For example, a keyword list of size 16 will be divided into two chunks of size 2^3 and each of these chunks will be stored in level 3.

Technical Details of Our Construction. In Figures 4 and 5 we illustrate our construction in more detail. In particular, the KeyGen algorithm takes as input the security parameter and computes the secret keys that are used in the randomized encryption scheme and the pseudorandom functions. The Setup algorithm takes as input the document collection, the secret keys, and the parameters s and L and in lines 1-6 it initializes the encrypted dictionary (it uses a hash table for the implementation of the encrypted dictionary) and the arrays A_i (each array A_i contains buckets of size 2^{i+1} — a collection of consecutive cells where each cell stores an encrypted (w, id) pair). In lines 7-15 the Setup algorithm places the keyword lists in the arrays A_i, while storing in the dictionary the bucket in which a keyword list or a chunk of the keyword list is stored. Finally, in lines 17-20 the entries of a bucket are randomly permuted and the arrays A_i are encrypted; each entry is encrypted using RND encryption and the produced key is a function of the keyword. This approach is only used in the keyword search scenario where we expect from the server to directly output the document identifiers. However, in the database search scenario, we encrypt each entry using RND encryption without choosing a key as a function of the keyword, since the server does not perform any decryption but outputs only the encrypted result to the client. In Figure 5, the Token algorithm produces the tokens which are given as inputs to the Search algorithm in order to locate the entry of the dictionary which corresponds to the queried keyword. The same algorithm partially decrypts L entries of the dictionary to

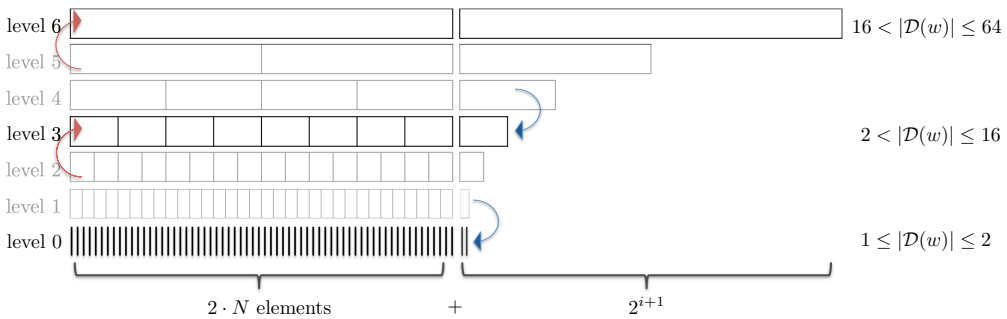


Figure 3: Example for $N = 64$, $s = 2$ and $L = 2$. Our scheme stores levels $0, 3, 6$. The red arrows depict the queries whose answers contain false positives but with optimal locality, while the blue arrows show queries with optimal read efficiency and constant locality.

further detect the correct buckets and the correct array A_i and filters out the false positives; the server attempts to decrypt all the entries inside a bucket but only the entries containing the queried keyword will have the last λ-bits to be 0. The same procedure could be used in the database search scenario to obtain the tuple identifiers of the answer. However, in the next section we provide a more efficient optimization to address this task since the tuples can be stored directly in the encrypted arrays A_i.

Scheme with Read Efficiency $O(R)$. It is easy to see that the above scheme can be tuned to achieve read efficiency $O(R)$ and worst-case locality $O(N^{1/s}/R)$ by setting $L = N^{1/s}/R$. We also note that for $L = N^{1/s}$ our scheme has optimal read efficiency $O(1)$ and has exactly the same leakage profile with prior SE schemes (such as Scheme 2). In the latter case, if we change the arrays A_i into hash tables, then the encrypted dictionary is not required.

3.3 Optimizations of Our Scheme

We now describe various optimizations that can be applied to our scheme. The first two are used in our implementation.

Optimization 1 - Decryption of the Result at the Client. In the current scheme, the decryption of the result (namely the identifiers of the documents containing the searched keyword) take place at the server side (see Line 6 of Algorithm Search). In particular the server performs 2^{i+1} decryption attempts, where 2^i is the size of the bucket from which we retrieve the answer. We can reduce the decryption cost to be proportional to the size of the result, by assigning the decryption to the client. This optimization can be used in keyword search by increasing the number of interactions; one round is required for the server to send the encrypted document-ids to the client and the client to decrypt them in order to request from the server in the second round to return the actual documents. In database search we only need one round of interaction, since using RND allows us to directly encrypt and store the tuples (instead of tuple-ids) in the encrypted index. It is highlighted that in this case the server does not perform any partial decryption of the tuples, i.e., the server identifies a super set of the result and returns this to the client (this super set of the result may contain false positives, which the client is responsible to filter out). This optimization requires the following changes:

- First, do not permute the entries in the bucket, as is done in Line 19 of Algorithm Setup. This will not violate our security since we will not allow any decryptions to take place at the server side.
- Second, instead of encrypting each entry in the bucket with a secret key derived from the respective keyword (see Line 20 of Algorithm Setup), encrypt all entries across all buckets with a single secret key that is stored at the client and is never revealed to the server.
- Finally, during the search, the server just sends the whole bucket encrypted to the client. Since the entries within the bucket are not randomly permuted, the entries corresponding to each keyword are consecutive. If the client knows the *start* and *end* point of each keyword, then he can decrypt only the part of the bucket that contains the actual result while skipping the remaining cells.

We note here that in order for the client to get the *start* and *end* values, our scheme can use an extra encrypted dictionary (like the one used to store offset and level i). To avoid increasing the space though, one could store only one dictionary with the *start* and *end* information, retrieve those and then derive (from *start* and *end*) the bucket offset and the level and request those from the server. The above observations can significantly accelerate the search procedure. Note that the above improvements are not applicable in Scheme 2 since in their case each bucket contains pieces of the result in arbitrary positions.

Optimization 2 - Storing One Level Less. Recall that in our scheme with constant locality we answer queries with size bigger than $2^{\ell-p+1}$ by retrieving one bucket from level ℓ (which both have size $2^{\ell+1}$). Since $2^{\ell+1} \geq N$, the aforementioned queries require reading information that is at least equal to the size of the dataset. We propose not storing level ℓ, but only the levels $\ell - p, \ell - 2p, \ldots, \ell - (s-1)p$, reducing the number of stored levels from s to $s-1$. To accommodate the queries with size $> 2^{\ell-p+1}$, we just have the server return the whole level $\ell - p + 1$. This does not asymptotically increase the read efficiency for level ℓ. In summary, with this new approach, we can store space $O(s \cdot N)$ for a read efficiency of $O(N^{\frac{1}{s+1}})$, and constant read locality. This optimization applies to the locality L scheme as well.

$\mathbf{k} \leftarrow \mathsf{KeyGen}(1^\lambda)$

$(k_1, k_2, k_3) \leftarrow^{\$} \{0,1\}^\lambda$.
$\mathsf{RND} = (Enc, Dec)$ is CPA-secure encryption scheme.
$F : \{0,1\}^* \times \{0,1\}^\lambda \rightarrow \{0,1\}^*$ is a PRF and $\mathsf{H} : \{0,1\}^* \rightarrow \{0,1\}^*$ is a collision resistant hash function.
return (k_1, k_2, k_3).

$\mathcal{I} \leftarrow \mathsf{Setup}(\mathbf{k}, \mathcal{D})$

Parse $\mathbf{k}$ as (k_1, k_2, k_3). Let s and L be publicly-known parameters.
Let $N = |\{\mathcal{D}(w)\}_{w \in \mathbf{W}}|$ and $\ell = \lceil \log N \rceil$. Set $p = \lceil \ell / s \rceil$. Let $\mathcal{L} = \{\ell, \ell - p, \ldots, \ell - (s-1) \cdot p\}$. If $L > 1$ set $\mathcal{L} = \mathcal{L} \cup \{0\}$.
Initialize a hash table HT that can store up to N elements.
for each evenly distributed level $i \in \mathcal{L}$ **do**
    Initialize an array A_i of size $2N + 2^{i+1}$.
    Divide A_i in Λ_i buckets of size 2^{i+1} and one bucket of size $y_i < 2^{i+1}$.
    Let $A_i[1], A_i[2], \ldots, A_i[\Lambda_i + 1]\}$ be the set of those buckets.
for each keyword $w \in \mathbf{W}$ in a random order **do**
    Find adjacent j and i in $\mathcal{L}$ such that $L \cdot 2^j < |\mathcal{D}(w)| \leq L \cdot 2^i$ (if i is the smallest level, we ignore the lower bound).
    Split $\mathcal{D}(w)$ into a set $\mathbf{C}_w$ of chunks containing q_w chunks of size 2^i and one chunk of size $r_w < 2^i$.
    $count = 0$.
    for each chunk $c \in \mathbf{C}_w$ **do**
        $count = count + 1$.
        Let $\mathbf{A}$ be the set of buckets in A_i that have enough space for chunk c (say those are $A_i[j_1], A_i[j_2], \ldots, A_i[j_f]$).
        Pick one bucket $a \in \mathbf{A}$ (say $A_i[x]$) uniformly at random and store c in a at the first available position.
        $HT.\mathbf{add}(\mathsf{H}(F_{k_1}(w)||count), [i||x] \oplus \mathsf{H}(F_{k_2}(w)||count))$.
Add random $(key, value)$ pairs to HT so that the total number of elements it stores is N.
for each stored array A_i (i.e., $i \in \mathcal{L}$) **do**
    for each bucket $b \in A_i$ **do**
        Randomly permute all entries (w, id) within b.
        Replace each entry (w, id) of b with $\mathsf{RND}.Enc_{key}(id||0^\lambda)$ where $key = F_{k_3}(w)$.
Let $\mathcal{A} = \{A_i : i \in \mathcal{L}\}$.
return $(HT, \mathcal{A})$.

Figure 4: KeyGen and Setup algorithms for our scheme with locality L, read efficiency $O(N^{1/s}/L)$ and space $\Theta(s \cdot N)$.

Optimization 3 - More Efficient Level Selection by Increasing Leakage. The basic algorithm for constant locality stores s out of $\ell = \lceil \log N \rceil$ levels which are evenly distributed. This means that selecting a level is a function of N. For instance, let us assume that $N = 2^{32}$ and $s = 4$. In this case our basic algorithm preserves levels 32, 24, 16, and 8. Yet in the case where the maximum keyword list has size 4, then we observe that all queries are answered in level 8 and the remaining levels have no use. Thus, given statistical information about the stored dataset we can construct a better level selection algorithm. Note that an SE scheme using this optimization should increase $\mathcal{L}_1$ by this additional statistical information. An example of such information that can be used is the minimum and maximum word lists. For real datasets this optimization can radically improve the performance of our proposed schemes, but in the experimental evaluation we do not consider it to present fair comparisons to related work.

Optimization 4 - Fault Tolerance. Existing fault-tolerance file system architectures are using the notion of replication to address failures. A typical replication factor for those systems is at least 3, meaning that the initial dataset with size N will be expanded three times. For instance, the default replication factor of Apache Hadoop File System (HDFS) is set to be 3 [30]. We can change our scheme to replicate all keyword/id pairs in all s levels (Lemma 1 will still hold!). In this way, for $s = 3$, our scheme can get fault-tolerance for free (since we are storing $s \cdot 2N$ space), while other schemes would have to explicitly triple the space they are using.

3.4 Security Analysis and Leakage Profile

Our main construction for general values of L leaks the following information (when searching fo keyword w):

1. $\mathcal{L}_1(\mathcal{D})$, as defined in Section 2.2. This is leaked by all previous schemes.

2. A deterministic function of the queried keyword $id(w)$ (search pattern). This is leaked by all previous schemes.

3. The bucket identifier $bucket(w)$ where w is read from. In particular, $bucket(w)$ contains information about the portion of the memory read to retrieve the result of a specific keyword (the level i and the offset of the bucket), which depends on the order in which the keywords were considered in the Setup algorithm. We believe this leakage is not meaningful since the order is decided at random. This information is not leaked by previous schemes.

We emphasize here also that our scheme, due to the first optimization in Section 3.3 and as opposed to all previous schemes, *does not explicitly leak* the exact size of the access pattern (it just leaks an upper bound on the size of the access

$t \leftarrow \mathsf{Token}(\mathbf{k}, w)$

1: Parse $\mathbf{k}$ as (k_1, k_2, k_3).
2: tag $\leftarrow F(k_1, w)$, vtag$\leftarrow F(k_2, w)$, etag$\leftarrow F(k_3, w)$.
3: **return** (tag,vtag,etag).

$\mathcal{X} \leftarrow \mathsf{Search}(t, \mathcal{I})$

1: Parse t as (tag,vtag,etag) and $\mathcal{I}$ as $(HT, \mathcal{A})$.
2: **for** $count = 1$ to L **do**
3: evalue $\leftarrow HT.\mathbf{get}(\mathsf{H}(\text{tag}||count))$.
4: **if** evalue is not NULL **then**
5: $[i, \mathsf{offset}] \leftarrow$ evalue $\oplus\ \mathsf{H}(\text{vtag}||count)$.
6: **for** all entries e in bucket $A_i[\mathsf{offset}]$ **do**
7: **if** $\mathsf{RND}.Dec_{\text{etag}}(e)$ outputs $id||0^{\lambda}$ **then**
8: Add id to the result set $\mathcal{X}$.
9: **return** result $\mathcal{X}$.

Figure 5: Token and Search algorithms for our scheme with locality L, read efficiency $O(N^{1/s}/L)$ and space $\Theta(s \cdot N)$.

pattern through the size of the bucket read). To summarize, we can now formally write our leakage functions as

$$\mathcal{L}_1(\mathcal{D}) = N \text{ and } \mathcal{L}_2^{new}(\mathcal{D}, w) = (id(w), buckets(w)).$$

Leakage Functions for the Case $L = N^{1/s}$. Unlike the general case, our scheme with $L = N^{1/s}$ has exactly the same leakage profile as previous schemes (e.g., Scheme 2): it just leaks $\mathcal{L}_1(\mathcal{D})$ and $\mathcal{L}_2(\mathcal{D}, w)$ (or $\mathcal{L}_2^{DB}(\mathcal{D}, w)$ in the case of the database scenario), as mentioned in Section 2.2.

THEOREM 1. *Given F is a pseudorandom function, H is a collision-resistant hash function and RND is a CPA-secure encryption scheme, the SE scheme of Figures 4 and 5 is $(\mathcal{L}_1, \mathcal{L}_2^{new})$-secure according to Definition 2.2 and in the random oracle model.*

Additionally, for the case when the locality $L = N^{1/s}$, the scheme is $(\mathcal{L}_1, \mathcal{L}_2)$-secure according to Definition 2.2 and in the random oracle model.

In Appendix B, we provide the proof of Theorem 1.

4. DYNAMIC SE (DSE)

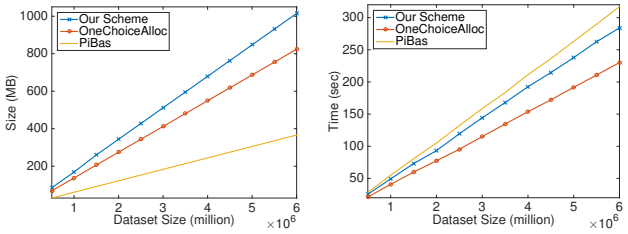
In this section, we present an extension to our current work that allows updates. Allowing updates is a twofold challenge because it requires the following: a) enable efficient inserts and deletes; b) avoid leaking extra information while achieving forward privacy, i.e. it does not reveal that a new update satisfies a previous query. The recent works of [28, 23, 5] do not seem to meet our efficiency requirements since Miers et al. [23] proposed a DSE scheme partially based on ORAMs, Bost [5] requires $O(m \log N)$ available client storage, where m denotes the size of the domain, and Stefanov et al. [28] use ideas mainly inspired by ORAMs and expensive cryptographic tools, such as oblivious sorting.

Proposed Solution. Our DSE scheme is based on a solution proposed by Demertzis et al. [12] for Range SE schemes. It is also used by commercial databases, such as Vertica [22] (organizes the updates in log-merge trees). This commonly used technique is preferable for the following reasons: i) it can use our very efficient static SE scheme as a "black box", ii) it enables easy leakage formulations, iii) it captures forward privacy. The leakage of this approach is essentially the entire history of the $\mathcal{L}_1, \mathcal{L}_2$ leakages of every index that was once "active" at the server. The main idea is that we organize n sequential updates to a collection of at most $\log n$ independent encrypted indexes. In particular, for each new tuple, the data owner initializes a new SE scheme by creating a new SE index that contains only the specific tuple. The single-tuple index is subsequently uploaded to the untrusted server. Whenever two indexes of the same size t are detected there are downloaded by the data owner, decrypted and merged to form a new SE index of size $2t$, again with a fresh secret key. The new index is then used to replace the two indexes of size t. Clearly, a merge may have a cascading effect, i.e. subsequent merges. In this case, all merges are executed at the same time to avoid redundant work, that is constructing and uploading intermediate indexes. Deletions are simulated by inserting cancellation tuples. For further details, we refer the reader to [12]. The amortized update cost is $O(\log n)$ but it can also be de-amortized by using ideas from [3]. The space is linear in the size of the input and the number of updates. The locality of this approach is $O(L \cdot \log n)$ and the read efficiency is $O(N^{1/s}/L)$.

5. EXPERIMENTS

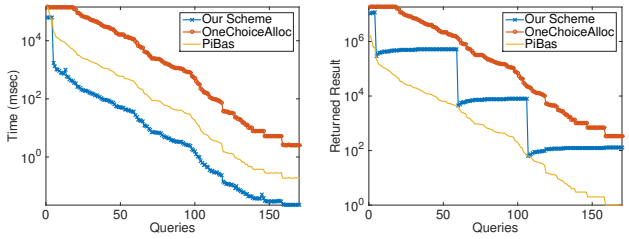
In this section we experimentally evaluate the performance of our proposed scheme. We compare our scheme only with linear-space approaches. If one can afford $N \log N$ space, the best scheme (both asymptotically and in practice) is Scheme 1 of Asharov et al. [4]. As such, we compare our work with the static construction of Cash et al. [6] and Scheme 2 of Asharov et al. [4]—see Table 1. We refer to the former as *PiBas* and to the latter as *OneChoiceAlloc*. We compare our scheme only with the basic construction of Cash et al. [6] (i.e. *PiBas*) because the more optimized proposed schemes (with good locality) are sub-optimal compared with the schemes of Asharov and introduce new leakages (in $\mathcal{L}_1$ leakage). Nevertheless, our scheme can be tuned with the use of the third optimization presented in Section 3.3 to achieve the same performance as the most optimized scheme of Cash et al. [6]. Moreover, we do not compare our scheme with Scheme 3 of Asharov et al. [4] because it assumes that no keyword list has size more than $N^{1-1/\log\log N}$ as shown is Table 1. Instead, we explain in the Appendix why this assumption is not realistic and we experimentally show the superiority of our scheme compared to Scheme 3 of Asharov et al. [4], by adopting the same assumption.

We organize the experimental section as follows. Section 5.1 presents the experimental setting and the technical details of our implementation. Section 5.2 focuses on the comparison of our work with *PiBas* and *OneChoiceAlloc* in an in-memory setting, while Section 5.3 compares our scheme with *PiBas* and *OneChoiceAlloc* in an external memory setting where optimal read efficiency is guaranteed in our scheme and *PiBas* and optimal locality in *OneChoiceAlloc*. Then, we compare our scheme with *OneChoiceAlloc*, where optimal locality is guaranteed in both schemes, and we focus on the comparison of the number of false positives in both approaches. Finally, in section 5.3 we provide experiments in parallel scenarios where we can tune locality to further reduce the number of false positives while assuring optimal locality per parallel processing unit (in particular, for overall locality L, we can have L parallel processing units with constant locality per unit).



(a) Index size (b) Construction time

Figure 6: Index costs



(a) Search Time (b) False Positives

Figure 7: Search costs

5.1 Setup

We carried out the implementation of our scheme, as well as the implementation of *PiBas* and *OneChoiceAlloc* in Java and conducted our experiments on a 64-bit machine with an Intel Xeon E5-2676v3 and 64 GB RAM. We utilized the JavaX.crypto library and the bouncy castle library [1] for the cryptographic operations. In particular, for the PRF and randomized symmetric encryption implementations we used HMAC-SHA-256 and AES128-CBC respectively for encryption. For our in-memory experiments we used single-threaded implementations, since a parallel implementation would favor our scheme compared to the schemes of our competitors. The experiments were conducted on a real dataset [2] consisting of $6,123,276$ records of reported incidents of crime. We consider the query attribute to be the *location description* which contains 173 distinct keywords (this is the x-axis in Figures 7(a),8). Among these keywords the one with minimum frequency contains 1 record, while the one with maximum frequency comprises $1,631,721$ records. The specific dataset is used for the in-memory setting comparison. Our external memory experiments use the above dataset for the comparison with optimal read efficiency and locality $O(N^{1/s})$. For the external memory comparison with optimal locality we created a synthetic dataset. Note that in both, our locality-optimal scheme and *OneChoiceAlloc* the only factor that affects the number of false positives is the number of records. Thus, we create two synthetic datasets where the first contains $N = 2^{37} - 1$ records and one keyword list for each possible size which is a power of 2, such that $|\mathcal{D}(w_i| = 2^i$. The second synthetic dataset has the same data structure, only now it comprises $N = 2^{47} - 1$ records.

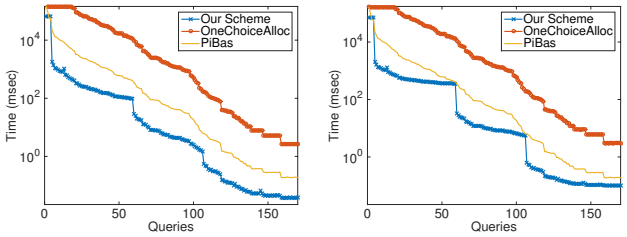
Implementation Details. We implement our locality-optimal algorithm using the first two optimizations described in Section 3.3. In particular, the client sends a token to the server and the server uses this token to locate and return the chunk that contains the answer of the query. This means that it is the responsibility of the client to decrypt and filter out the resulting false positives. We use optimization 1 to reduce the client's workload. In addition, we use optimization 2 to further reduce the required server space. We implement our read efficiency optimal algorithm without encrypted dictionary using for A_i hash tables.

The implementation of *PiBas* is straight-forward and was carried out as proposed in the work of Cash et al. [6]. We implemented the work of Asharov using a dictionary as was proposed for the general case. In particular, the dictionary contains the size of the result for each keyword. Thus, the client sends a token to the server and the server using the dictionary locates the first bucket in which the first result is placed and also the number of total buckets that it has to return to the client. The client is responsible for decrypting the buckets in order to filter out the false positives.

Both our scheme and *OneChoiceAlloc* can be implemented to return the exact result without any false positives as shown in Figures 4 and 5, where the server herself can filter out the false positives. However, doing this decreases the performance of both schemes. Only for the experiment in Figure 9(b), we implement *OneChoiceAlloc* to filter out the false positives at the server, since our read efficiency optimal scheme and *PiBas* do not contain false positives (we also provide an experiment for *OneChoiceAlloc* where the client performs the filtering — see Figure 9(a)). Note that carrying out the filtering on the server can be an efficient solution when the application has strict bandwidth limitations, because then the false positives are removed before transferring the data over the network.

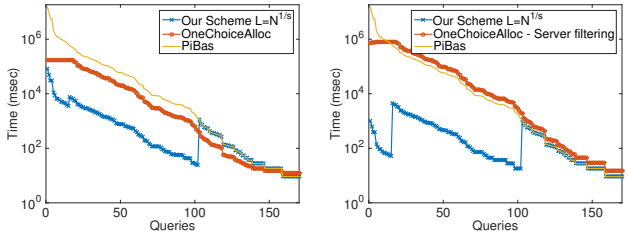
5.2 In-memory Comparison

Index Costs. In the first set of experiments we evaluate the required index size and construction time of our scheme for different dataset sizes N. The results are shown in Figures 6(a) and 6(b) respectively. The construction time includes the I/O cost of transferring the dataset from the disk to the main memory, while the index size represents only the size of the encrypted index, since the size of the encrypted documents (or tuples) is the same for all schemes. Moreover, we partition the initial dataset into 12 sets of 500K tuples each, chosen uniformly at random from the entire dataset. Then, we begin with the first partition and consider the other partitions in each step in order to represent the construction time and index size as the size of the input gradually increases. For the initialization of our scheme we selected $s = 2$. According to our analysis and using the first optimization implies storing 2 levels and having read efficiency $O(N^{1/3})$. In this way we have space requirements comparable to *OneChoiceAlloc* whose space is approximately $3N$, while our case requires approximately $4N$ space; in both cases an encrypted dictionary of size N is required. Recall that due to Lemma 1, our scheme requires to store in each preserved level i an array of size $2N$ and one extra chunk of size 2^{i+1}. These space requirements are included in our figures. As expected, our schemes require slightly more storage and time for constructing the index compared to *OneChoiceAlloc*. We observe that *PiBas* requires less storage than both our scheme and *OneChoiceAlloc*, but the construction time performance is worse because of the need for more PRF evaluation per keyword/identifier



(a) 1 Gbps transfer rate (b) 200 Mbps transfer rate

Figure 8: Search Time



(a) End-to-end search time (b) Server search time

Figure 9: External memory comparison for the real dataset

pair. In particular, our schemes require index size from 85 to 1015 MB and construction time from 25 to 283 sec, while *PiBas* requires index size from 31 to 366 MB and construction time from 28 to 317 sec and *OneChoiceAlloc* requires index size from 69 to 824 MB and construction time from 21 to 230 sec. In addition to these outcomes though, it is worth mentioning that in our scheme the Setup algorithm is highly parallelizable, and therefore the cost of the specific algorithm can be distributed to different machines. In the case of *PiBas* and *OneChoiceAlloc* it is not straight-forward how we could parallelize the construction of the index.

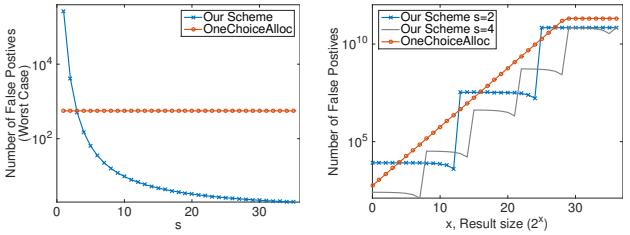
Search Cost. Figure 7(a) illustrates the total time required by the server and client to perform every possible query excluding the communication cost. All schemes require transferring the result size through the network. However, our scheme and *OneChoiceAlloc* need to transfer more data than *PiBas* for each query. Our approach transfers on average 35× more information compared with *PiBas* and in the worst case this number becomes equal to 126×, while *OneChoiceAlloc* always transfers 324× times more data compared with *PiBas*. More specifically, Figure 7(b) shows the number of records returned by our approach, *OneChoiceAlloc* approach and *PiBas* approach, which transfers the exact result without false positives. For visualization purposes, we sort the queries based on their result size and we query each of them. Our schemes reached up to 12× speed-up compared to *PiBas* and achieved 347× speed-up in comparison with *OneChoiceAlloc*. This experiment confirms our non-trivial claim that optimal locality can be successfully used to achieve more efficient SE schemes even for in-memory architectures or fast external storage devices, i.e. solid state drives. Note that our scheme yields the above speed-up by reducing the workload of the server and client in the two following manners: i) the server is only responsible for returning the required chunks without evaluating a PRF for each result item; ii) the chunks contain the results of each query together, thus allowing her to decrypt only the requested result. These two features offered by our construction can, neither be integrated with *PiBas*, nor with *OneChoiceAlloc*.

The purpose of this experiment is to illustrate the amount of less work performed by the client and the server in our approach, and for this reason we exclude the communication cost. Nevertheless, we also conducted the experiments while taking into account the communication cost and observed that having on average 1 Gbps transfer rate in Figure 8(a) yielded results similar with the ones presented in Figure 7(a) when we compared our scheme with *PiBas* and *OneChoiceAlloc*. Additionally, for 200 Mbps transfer rate or more our scheme becomes the most efficient one for all possible queries as shown in Figure 8(b), while when the transfer rate is less than 200 Mbps some queries may become slower than *PiBas*. In comparison with the *OneChoiceAlloc* scheme our scheme is always more efficient regardless of the transfer rate, because our scheme transfers less false positives. For applications with limited communication bandwidth we suggest using the original protocol, where the server filters out the false positives herself , or the scheme with locality $L = N^{1/s}$. The latter scheme described in section 3.2 can be tuned to always be more efficient than *PiBas*. By tuning read efficiency to be optimal, less PRF evaluations are performed in our scheme compared with *PiBas*. Additionally, in the next section, we will show that this scheme is always more efficient than *OneChoiceAlloc* especially for queries with results size $> 2^7$ tuples.

5.3 External Memory Comparison

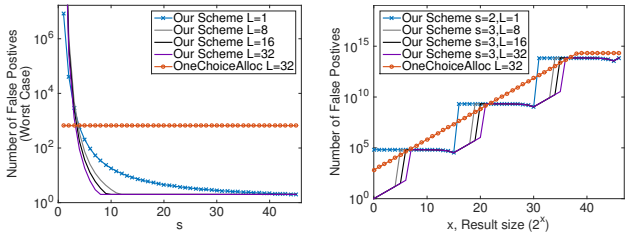
We have already mentioned that any scheme lacking locality cannot be used for big data applications (due to the cost of accessing data on the disk at random locations). In this section, we first compare our optimal read efficiency scheme (with no-optimal locality $L = N^{1/s}$), with *PiBas* which has worst-case locality and *OneChoiceAlloc* which has optimal locality in external memory scenarios, where random accesses become the dominant factor. This comparison provides a very interesting outcome. Despite the fact that *PiBas* has worst-case locality and *OneChoiceAlloc* has optimal locality, both have similar performance, while our scheme achieves up to 60x faster search time compared to *OneChoiceAlloc*. Our scheme requires $5N$ space (N for level 0 and $4N$ for 2 additional levels — an encrypted dictionary is not required), while *OneChoiceAlloc* requires $4N$ space (including the encrypted dictionary). Then, we compare our optimal locality scheme with *OneChoiceAlloc*. In Section 5.2, we experimentally showed the superiority of our scheme over *OneChoiceAlloc*. We now measure only the number of false positives produced by each approach, since it is the only factor that differs between the two schemes and impacts their performance in practice.

Figure 9(a) depicts the end-to-end search time for the real dataset. *PiBas* and our scheme return to the client the exact answer, while in this experiment *OneChoiceAlloc* returns the answer with false positives. We observe that *OneChoiceAlloc* is faster than *PiBas*, while our scheme is up to 60x faster than *OneChoiceAlloc* and for the queries with size $\leq 2^7$ tuples it has the same performance as *PiBas* and at most 2.8x speed-down compared to *OneChoiceAlloc*. This is due to



(a) FP for variable s (b) FP for different sizes

Figure 10: External memory comparison ($N = 2^{37} - 1$)



(a) FP for variable s (b) FP for different sizes

Figure 11: External memory comparison using parallelism ($N = 2^{47} - 1$)

the block size whose size is 4K and can contain at most 2^7 encrypted keyword,id pairs. If the result size is smaller than 2^7 we still have to retrieve a whole block from the disk. Furthermore, if level 7 is the next stored level after 0 , then a query with size 2^6 requires from the server to read 2^6 blocks and for each of these blocks to perform a random access. In this case, for all query results with less than 2^7 tuples our scheme works identically to the scheme of *PiBas*.

In this experiment, we implemented *OneChoiceAlloc* to conduct the filtering of the false positives on the server side (as was originally proposed in the paper [4]). Now, *PiBas*,*OneChoiceAlloc* and our scheme return to the client the result without false positives, so the communication cost and client work become the same for all schemes. Figure 9(b) compares the server search time for the threes schemes. We surprisingly observe that when the server filters the false positives, then for the majority of the queries *PiBas* is slightly better than *OneChoiceAlloc*. *OneChoiceAlloc* is designed to have optimal locality, while *PiBas* has worst-case locality. The reason, why in Figure 9(a) and in Figure 9(b) *OneChoiceAlloc* and *PiBas* have similar performance is because for each piece of the result *PiBas* pays a PRF evaluation (approximately 43 μsec) and a random access (approximately 10 msec), while *OneChoiceAlloc* requires $3 \log N \log \log N$ PRF evaluations (approximately 13msec per result item) and no random accesses. *OneChoiceAlloc* has optimal locality, but requires significantly more cryptographic operations and the benefit gained from locality is nullified by the cost of the additional cryptographic operations. In Figure 9(b), we observe that our scheme requires up to 4 orders of magnitude less search time on the server than *OneChoiceAlloc*.

Below, our scheme has again optimal locality. Figure 10(a) depicts the worst-case read efficiency compared with *OneChoiceAlloc* for the first synthetic dataset of size $N = 2^{37} - 1$ and for different numbers of levels s in our scheme. For $s \geq 4$ our scheme always outperforms *OneChoiceAlloc*, and therefore we consider the interesting cases to be $s = 2$ and $s = 4$.

In Figure 10(b) we compare the number of false positives of our approach and *OneChoiceAlloc*, for all possible queries for $s = 2, 4$. For $s = 2$ (which requires the same space with *OneChoiceAlloc*), our approach outperforms *OneChoiceAlloc* for almost all possible queries, reaching maximum speed-up approximately 577×. This is because our scheme does not penalize queries with the worst-case bound.

We conduct the same experiments on a dataset of size close to 1 petabyte and the resulting outcomes are illustrated in Figures 11(a)and 11(b) (see Our Scheme for $L = 1$ and the parallel *OneChoiceAlloc* for $L = 32$). Figure 11(b) shows that only for a small portion of all possible queries *OneChoiceAlloc* has less false positives than our scheme.In the worst case, our scheme reaches 86× speed-down compared to *OneChoiceAlloc*, but for the biggest portion we achieve significantly higher speed-ups up to 760×.

Since it is impractical to handle a dataset of size close to 1 petabyte without exploiting parallelism, we tune the locality L of our scheme to be equal to the number of parallel processing units (the locality per processor remains $O(1)$). In this case we always achieve a smaller number of false positives for all queries. For comparison reasons we created a parallel implementation of the *OneChoiceAlloc* scheme based on the (naive) idea proposed in Section 3.2. We could not further improve the parallel approach of *OneChoiceAlloc*. Figures 11(a) and 11(b) report the results of the same experiments, but now using 1, 8, 16, 32 parallel processing units for our scheme, and always 32 parallel processing units for the *OneChoiceAlloc* scheme. Recall that when $L > 1$ we also have level 0, but only for level 0 we store an array of size N instead of $2N + 2$. The vast improvement is achieved because for $s = 3$ our complexity is $O(N^{1/3}/L)$ (as explained in Section 3.2) while the complexity of *OneChoiceAlloc* is $\Theta(\log(N/L) \log \log(N/L))$. Thus, the impact of L in our scheme is much larger.

6. CONCLUSIONS

In this paper we revisit the problem of Searchable Encryption (SE) with small locality, and we propose the first SE scheme with tunable locality. Our scheme allows tuning the space, read efficiency, locality, parallelism and communication overhead in order to achieve optimal performance for any arbitrary database architecture. The experimental evaluations show that our schemes outperform the state-of-the-art in-memory and external memory SE. Our work points out that another aspect for consideration in this problem is the number of cryptographic operations required to obtain the result. We show that the scheme of Asharov et al. [4] with optimal locality scales similarly to a scheme with worst-case locality (for external memory evaluation) since it requires significantly more cryptographic operations.

Acknowledgements

This work was supported in part by NSF awards #1514261 and #1526950, the National Institute of Standards and Technology, a Google faculty research award, a Yahoo! faculty research engagement program and a NetApp faculty fellowship. We thank the SIGMOD reviewers for their helpful comments.

7. REFERENCES

[1] Bouncy castle. http://www.bouncycastle.org.

[2] Crimes 2001 to present (city of chicago). https://data.cityofchicago.org/public-safety/crimes-2001-to-present/ijzp-q8t2.

[3] *The Design of Dynamic Data Structures*. Springer Science & Business Media, 1983.

[4] G. Asharov, M. Naor, G. Segev, and I. Shahaf. Searchable Symmetric Encryption: Optimal Locality in Linear Space via Two-Dimensional Balanced Allocations. In *STOC, 2016*.

[5] R. Bost. Sofos: Forward Secure Searchable Encryption. In *CCS*, 2016.

[6] D. Cash, J. Jaeger, S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation. In *NDSS*, 2014.

[7] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Roşu, and M. Steiner. Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries. In *CRYPTO*, 2013.

[8] D. Cash and S. Tessaro. The Locality of Searchable Symmetric Encryption. In *EUROCRYPT, 2014*.

[9] M. Chase and S. Kamara. Structured Encryption and Controlled Disclosure. In *ASIACRYPT*, 2010.

[10] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions. *Journal of Computer Security, 2011*.

[11] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions. In *CCS*, 2006.

[12] I. Demertzis, S. Papadopoulos, O. Papapetrou, A. Deligiannakis, and M. Garofalakis. Practical Private Range Search Revisited. In *SIGMOD*, 2016.

[13] S. Faber, S. Jarecki, H. Krawczyk, Q. Nguyen, M. Rosu, and M. Steiner. Rich Queries on Encrypted Data: Beyond Exact Matches. In *ESORICS*, 2015.

[14] C. Gentry. *A Fully Homomorphic Encryption Scheme*. PhD thesis, Stanford University, 2009.

[15] C. Gentry. Computing Arbitrary Functions of Encrypted Data. *Commun. of the ACM*, 2010.

[16] E.-J. Goh et al. Secure Indexes. *IACR Cryptology ePrint Archive*, 2003.

[17] O. Goldreich and R. Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *J. ACM, 1996*.

[18] S. Kamara and C. Papamanthou. Parallel and Dynamic Searchable Symmetric Encryption. In *FC*, 2013.

[19] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic Searchable Symmetric Encryption. In *CCS*, 2012.

[20] J. Katz and Y. Lindell. *Introduction to modern cryptography*. CRC press, 2014.

[21] K. Kurosawa and Y. Ohtaki. UC-Secure Searchable Symmetric Encryption. In *FC, 2012*.

[22] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. The Vertica Analytic Database: C-store 7 Years Later. *PVLDB*, 2012.

[23] I. Miers and P. Mohassel. IO-DSSE: Scaling Dynamic Searchable Encryption to Millions of Indexes By Improving Locality. In *NDSS*, 2017.

[24] M. Naveed, S. Kamara, and C. V. Wright. Inference Attacks on Property-Preserving Encrypted Databases. In *CCS*, 2015.

[25] B. Pinkas and T. Reinman. Oblivious RAM Revisited. In *CRYPTO*. 2010.

[26] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *SOSP*, 2011.

[27] D. X. Song, D. Wagner, and A. Perrig. Practical Techniques for Searches on Encrypted Data. In *SP*, 2000.

[28] E. Stefanov, C. Papamanthou, and E. Shi. Practical Dynamic Searchable Encryption with Small Leakage. In *NDSS*, 2014.

[29] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path Oram: An Extremely Simple Oblivious Ram Protocol. In *CCS*, 2013.

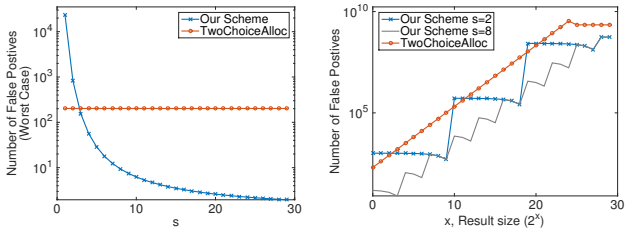
[30] A. Team and Others. Apache HBase Reference Guide.

[31] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. Processing Analytical Queries over Encrypted Data. In *PVLDB*, 2013.

[32] P. Van Liesdonk, S. Sedghi, J. Doumen, P. Hartel, and W. Jonker. Computationally Efficient Searchable Symmetric Encryption. In *SDM*. 2010.

[33] Y. Zhang, J. Katz, and C. Papamanthou. All Your Queries Are Belong to Us: The Power of File-Injection Attacks on Searchable Encryption. In *USENIX 2016*.

Appendix A



(a) FP for variable s (b) FP for different sizes

Figure 12: External memory comparison ($N = 2^{47} - 1$)

In the experimental evaluation section 5 we purposely did not include the comparison of our work with the Scheme 3 proposed by Asharov et al. in [4] (*TwoChoiceAlloc*), due to their assumption of not considering word lists of size more than $N^{1-1/\log\log N}$. This assumption cannot be taken into consideration for real world datasets. For instance, let us examine the real dataset of crime records that was used in our experiments that contains 21 attributes. Then, for 12 attributes out of the total of 21, their assumption is violated. Therefore, it becomes infeasible to use the *TwoChoiceAlloc* scheme on these attributes. Moreover, even though the assumption holds for the 9 remaining attributes note that these contain only unique values or have very small cardinality, i.e. the following attributes: id, case number, date and time, X coordinate, Y coordinate, longitude and latitude. For these attributes *TwoChoiceAlloc* can be used, but even then our scheme yield a smaller number of false positives. More specifically, we applied to our scheme the assumption

of $N^{1-1/\log\log N}$ by computing the level that has chunks bigger than the answer to a query of size $N^{1-1/\log\log N}$. Since it is impossible to consider such sizes (or bigger), we evenly select the appropriate levels out of the remaining excluding those that have chunks with size bigger than the answer of the query. This can be illustrated in an example where $N = 2^{47} - 1$. Then, the assumption that we do not have a query result of size bigger than $N^{1-1/\log\log N}$ means that the possible query sizes range from 1 to 2^{39}. Therefore, our algorithm will select evenly distributed levels starting from level 1 up to level 36 excluding all levels higher than 36. Figures 12(a) and 12(b) compare the two schemes showing that for the same amount of space which is equivalent to setting our redundancy factor $s = 2$ since $TwoChoiceAlloc$ requires $4 \cdot N$ space, our scheme is always better than $TwoChoiceAlloc$. In addition, $TwoChoiceAlloc$ assumes that the size of each word list is a power of 2 and N is also assumed to be a power of 2. Hence, in the worst case the $TwoChoiceAlloc$ scheme requires padding the dataset, thus yielding a final size that reaches $8 * N$. In this case and in order for both schemes to use the same space we tune our redundancy factor to be equal to 8 so we also show the case of our scheme having redundancy factor $s = 8$.

Appendix B: Proof of Theorem 1

For simplicity, we provide the proof of Theorem 1 for constant L using the first optimization described in section 3.3 and for $L = N^{1/s}$ using $\mathcal{L}_2^{DB}$ leakage (database scenario).

The simulator algorithms:

- $\mathsf{SimSetup}(\mathcal{L}_1(\mathcal{D}))$
- $\mathsf{SimToken}(\mathcal{L}_2^{new}(\mathcal{D}, w))$
- $\mathsf{SimToken}(\mathcal{L}_2^{DB}(\mathcal{D}, w))$

are shown in Figure 13 and Figure 14, respectively.

For the first part of the proof (which is the same for both cases of Theorem 1) we must show that no PPT algorithm Dist can distinguish, with more than negligible probability, between the index $\mathcal{I}_{real} = (HT_{real}, \mathcal{A}_{real})$ output by $\mathsf{Setup}(\mathbf{k}, \mathcal{D})$ and the index $\mathcal{I}_{ideal} = (HT_{ideal}, \mathcal{A}_{ideal})$ output by $\mathsf{SimSetup}(\mathcal{L}_1(\mathcal{D}))$. This is because

- Both $\mathcal{A}_{real}$ and $\mathcal{A}_{ideal}$ are sets of s arrays

$$A_\ell, A_{\ell-p}, A_{\ell-2p} \ldots A_{\ell-(s-1)\cdot p}\,,$$

  where in both cases A_i has $2N + 2^i$ entries (A_0 is included for $L > 1$). $\mathcal{A}_{real}$ contains the encryption of values using a CPA-secure scheme, while $\mathcal{A}_{ideal}$ contains random values of the same format.
- Similarly, Dist cannot distinguish HT_{real} from HT_{ideal}, since both have N entries, HT_{real} encrypts entries by "xoring" with the output of a pseudorandom function and HT_{ideal} contains random values.

For the second part of the proof and according to Definition 1, we need to prove that there does not exist a PPT algorithm Dist that can distinguish between the outputs of $\mathsf{Token}(\mathbf{k}, w)$ and the outputs of $\mathsf{SimToken}(st_{\mathcal{S}}, \mathcal{L}_2^{new}(\mathcal{D}, w))$ and $\mathsf{SimToken}(st_{\mathcal{S}}, \mathcal{L}_2^{DB}(\mathcal{D}, w))$. This is because:

- Both the Token and $\mathsf{SimToken}$ produce the same tokens for the same repeated keywords. $\mathsf{SimToken}$ uses the search pattern leakage Prev included in $st_{\mathcal{S}}$.

Case for locality $L = 1$.

- For a given keyword, $\mathsf{SimToken}$ and Token output the same $[i, \mathsf{offset}]$ pair, included in the $\mathcal{L}_2^{new}(\mathcal{D}, w)$ leakage and therefore the distributions are trivially indistinguishable. Note that for $\mathsf{SimToken}$ to output a specific $[i, \mathsf{offset}]$ pair we program the random oracle H accordingly—see Line 9 in Figure 13.

Case for locality $L = N^{1/s}$.

- For a given keyword, $\mathsf{SimToken}$ and Token do not necessarily output the same $[i, \mathsf{offset}]$ pairs. However, the $[i, \mathsf{offset}]$ pairs output by both algorithms are distributed *identically*. In particular the real $[i, \mathsf{offset}]$ pairs output by Token are distibuted uniformly at random (due to such placement by Setup), while $\mathsf{SimToken}$ chooses uniformly at random $[i, \mathsf{offset}]$ pairs among the pairs that she did not choose before—see Line 11 in Figure 14. Again, for $\mathsf{SimToken}$ to output a specific $[i, \mathsf{offset}]$ pair dictated by the simulator we program the random oracle H accordingly—see Line 12 in Figure 14.

$(\mathcal{I}, st_{\mathcal{S}}) \leftarrow \mathsf{SimSetup}(\mathcal{L}_1(\mathcal{D}))$

Let $N \leftarrow \mathcal{L}_1(\mathcal{D})$,$\ell = \log N$, $p = \lceil \ell/s \rceil$. Let $\mathcal{L} = \{\ell, \ell - p, \ldots, \ell - (s-1) \cdot p\}$. If $L > 1$ set $\mathcal{L} = \mathcal{L} \cup \{0\}$.
$\mathbf{k} \leftarrow \mathsf{KeyGen}(1^{\lambda})$.
Initialize a hash table HT of size N random entries and mark them as "unrevealed".
for each evenly distributed level $i \in \mathcal{L}$ **do**
    Initialize an array A_i of size $2N + 2^{i+i}$ with random elements.
    Divide A_i in Λ_i buckets of size 2^{i+1} and one bucket of size $y_i < 2^{i+1}$.
    Let $A_i[1], A_i[2], \ldots, A_i[\Lambda_i + 1]\}$ be the set of those buckets and mark all the buckets as "unrevealed".
Let Prev an empty hash table.
Set $\mathcal{I}$ to be the set $\mathcal{A} = \{A_i : i \in \mathcal{L}\}$ and encrypted dictionary HT.
Set $st_{\mathcal{S}}$ to include $\mathcal{I}$, Prev and $\mathbf{k}$.
return $(\mathcal{I}, st_{\mathcal{S}})$.

$(t_w, st_{\mathcal{S}}) \leftarrow \mathsf{SimToken}(st_{\mathcal{S}}, \mathcal{L}_2^{new}(\mathcal{D}, w))$

Parse $st_{\mathcal{S}}$ as $\mathcal{A} = \{A_i : i \in \mathcal{L}\}$, HT, Prev and $\mathbf{k} = k_1, k_2$.
Let $(id(w), buckets(w)) \leftarrow \mathcal{L}_2(\mathcal{D}, w)$.
if $\mathsf{Prev}.\mathbf{get}(id(w)) \neq \mathsf{null}$ **then**
    return $(\mathsf{Prev}.\mathbf{get}(id(w)), st_{\mathcal{S}})$.
else
    Set tag $= F(k_1, id(w))$, vtag $= F(k_2, id(w))$ and $count = 1$.
    for each $(i, \mathsf{offset}) \in buckets(w)$ **do**
        Pick uniformly at random an "unrevealed" entry $e = (key, value)$ from hash table HT.
        Program the random oracle H such that $\mathsf{H}(\text{tag}||count) = key$ and $\mathsf{H}(\text{vtag}||count) = value \oplus [i, \mathsf{offset}]$.
        Mark e as "revealed" and set $count = count + 1$.
    Set $t_w \leftarrow (\text{tag}, \text{vtag})$.
    $\mathsf{Prev}.\mathbf{add}(id(w), t_w)$.
    Update $st_{\mathcal{S}}$ with new values of Prev, the choices that the random oracle made.
return $(t_w, st_{\mathcal{S}})$.

Figure 13: Simulator algorithms $\mathsf{SimSetup}$ and $\mathsf{SimToken}$ for scheme with $O(L)$ locality and $O(N^{1/s}/L)$ read efficiency.

$(t_w, st_{\mathcal{S}}) \leftarrow \mathsf{SimToken}(st_{\mathcal{S}}, \mathcal{L}_2(\mathcal{D}, w))$

Parse $st_{\mathcal{S}}$ as $\mathcal{A} = \{A_i : i \in \mathcal{L}\}$, HT, Prev and $\mathbf{k} = k_1, k_2$.
Let $(id(w), |\mathcal{D}(w)|) \leftarrow \mathcal{L}_2(\mathcal{D}, w)$.
if $\mathsf{Prev}.\mathbf{get}(id(w)) \neq \mathsf{null}$ **then**
    return $(\mathsf{Prev}.\mathbf{get}(id(w)), st_{\mathcal{S}})$.
else
    Find table A_i for maximum i such that $2^i < |\mathcal{D}(w)|$ (if $|\mathcal{D}(w)|$=1 set $i = 0$).
    $q = \lceil |\mathcal{D}(w)|/2^i \rceil$.
    Set tag $= F(k_1, id(w))$, vtag $= F(k_2, id(w))$.
    for count $= 1$ to q **do**
        Pick uniformly at random an "unrevealed" entry $e = (key, value)$ from hash table HT.
        Pick uniformly at random an offset of an "unrevealed" bucket b at level i.
        Program the random oracle H such that $\mathsf{H}(\text{tag}||count) = key$ and $\mathsf{H}(\text{vtag}||count) = value \oplus [i, \mathsf{offset}]$.
        Mark e and b as "revealed".
    Set $t_w \leftarrow (\text{tag}, \text{vtag})$.
    $\mathsf{Prev}.\mathbf{add}(id(w), t_w)$.
    Update $st_{\mathcal{S}}$ with new values of Prev and the choices that the random oracle made.
return $(t_w, st_{\mathcal{S}})$.

Figure 14: Simulator $\mathsf{SimToken}$ for scheme with $O(N^{1/s})$ locality and $O(1)$ read efficiency.