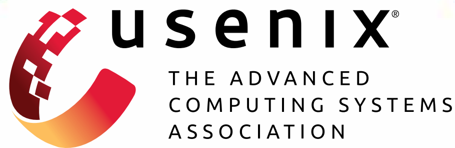

I/O-Efficient Dynamic Searchable Encryption meets Forward & Backward Privacy

Priyanka Mondal, *University of California, Santa Cruz;* Javad Ghareh Chamani, *HKUST;* Ioannis Demertzis, *University of California, Santa Cruz;* Dimitrios Papadopoulos, *HKUST*

https://www.usenix.org/conference/usenixsecurity24/presentation/mondal

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the 33rd USENIX Security Symposium is sponsored by USENIX.

I/O-Efficient Dynamic Searchable Encryption meets Forward & Backward Privacy

Priyanka Mondal
UC Santa Cruz

Javad Ghareh Chamani
HKUST

Ioannis Demertzis
UC Santa Cruz

Dimitrios Papadopoulos
HKUST

Abstract

We focus on the problem of I/O-efficient Dynamic Searchable Encryption (DSE), i.e., schemes that perform well when executed with the dataset on-disk. Towards this direction, for HDDs, schemes have been proposed with good *locality* (i.e., low number of performed non-continuous memory reads) and *read efficiency* (the number of additional memory locations read per result item). Similarly, for SSDs, schemes with good *page efficiency* (reading as few pages as possible) have been proposed. However, the vast majority of these works are limited to the *static* case (i.e. no dataset modifications) and the only dynamic scheme fails to achieve forward and backward privacy, the de-facto leakage standard in the literature. In fact, prior related works (Bost [CCS'16] and Minaud and Reichle [CRYPTO'22]) claim that I/O-efficiency and forward-privacy are two *irreconcilable* notions. Contrary to that, in this work, we "reconcile" for the first time forward and backward privacy with I/O-efficiency for DSE both for HDDs and SSDs. We propose two families of DSE constructions which also improve the state-of-the-art (non I/O-efficient) both asymptotically and experimentally. Indeed, some of our schemes improve the in-memory performance of prior works. At a technical level, we revisit and enhance the *lazy de-amortization* DSE construction by Demertzis et al. [NDSS'20], transforming it into an I/O-preserving one. Importantly, we introduce an oblivious-merge protocol that merges two equal-sized databases without revealing any information, effectively replacing the costly oblivious data structures with more lightweight computations.

1 Introduction

Searchable encryption has been the topic of a huge line of research (e.g., [14, 15, 19–23, 26, 29, 30, 34, 36, 40, 45, 48, 52, 53, 60–63, 66, 67, 69–72]). At the same time, it has been proposed for use in a number of different applications, e.g., for encrypted relational and graph databases [16, 18, 28, 44], annotated image search [2], encrypted email [54], or maintaining a secure gun registry service [43]. In fact, very recently, MongoDB announced support for a mode called *queryable* encryption, using techniques inspired by this line of research. The approach followed by most works is to strive for a high performance at the cost of well-defined *leakages* observed by the server storing the encrypted dataset. I.e., although it is required that the server never accesses decrypted data, *search* query on the data allows the server to learn some information that typically includes which entries are accessed for a query (*access pattern leakage*), which queries are for the same term (*search pattern leakage*), and how many entries are returned (*volume pattern leakage*). Dynamic Searchable Encryption (DSE) schemes have additional leakage considerations related to updates, i.e., do not reveal any connection of newly modified entries to previous ones, or do not reveal information of deleted entries during queries that take place after the deletion. DSE schemes that protect against both these leakages are called *forward-and-backward private* [12].

I/O overhead of DSE schemes. Existing state-of-the-art DSE schemes achieve extremely low computational overhead for search and update queries, typically in the order of a few milliseconds or even microseconds, even for massive datasets [14, 26]. The key reason for this is that they employ extremely lightweight symmetric-key cryptographic techniques, mainly pseudorandom functions (PRF) and symmetric key encryption. At a high level, most existing schemes use some variation of an *encrypted map* index [14], a key-value storage data structure where the values are encrypted tuples of the form (w, id) (where w is a keyword/term, and id is the identifier of a record from the dataset, and the tuple attests to "w appears in id)[1] and keys are computed *pseudo-randomly* with a PRF. This not only simplifies subsequent searches significantly, but also, due to the PRF usage, tuples are placed in random-looking positions in the index, which is crucial for minimizing the information revealed to the server about structure of the underlying dataset.

While this random placement does not significantly affect DSE performance when the encrypted index is stored in RAM

[1] In our constructions the tuple also contain the operation type: *(w,id,op)*.

Scheme	Storage	Search			Update Cost	FP / BP	HDD / SSD /
		Locality	Read Efficiency	Page Efficiency			
LayeredSSE [55]	$O(N)$	$O(n_w/p+\log N)$	$O(\log\log N)$	$O(\log\log N)$	$O(n_w)$	✗ / ✗	SSD
Local[LayeredSSE] [55]*	$O(N)$	$O(1)$	$\tilde{O}(\log\log N)$	$\tilde{O}(\log\log N)$	$O(n_w)$	✗ / ✗	HDD
SD_a[1C [6]]	$O(N)$	$O(\log N)$	$O(\log N)$	$O(\log N)$	$O(\log N)(am.)$	✓ / II	HDD
SD_a[2C [6]]*	$O(N)$	$O(\log N)$	$O(\log N)$	$O(\log N)$	$O(\log^2 N)(am.)$	✓ / II	HDD
SD_a[NlogN [6]]	$O(N\log N)$	$O(\log N)$	$O(\log N)$	$O(\log N)$	$O(\log^2 N)(am.)$	✓ / II	HDD/SSD
SD_a[sN [31]]	$O(sN)$	$O(N^{\frac{1}{s}}+\log N)$	$O(\log N)$	$O(N^{\frac{1}{s}}/p+\log N)$	$O(s\log N)(am.)$	✓ / II	HDD/SSD
SD_a[Tethys/Pluto [10]]	$O(N)$	$O(n_w/p+\log N)$	$O(p)$	$O(\log N)$	$O(N\log N)(am.)$	✓ / II	SSD
L-SD_d[1C [6]]	$O(N)$	$O(\log N)$	$\tilde{O}(\log N)$	$\tilde{O}(\log N)$	$O(\log^2 N)$	✓ / II	HDD
L-SD_d[2C [6]]*	$O(N)$	$O(\log N)$	$\tilde{O}(\log N)$	$O(\log N)$	$O(\log^2 N)$	✓ / II	HDD
L-SD_d[NlogN [6]]	$O(N\log N)$	$O(\log N)$	$O(\log N)$	$O(\log N)$	$O(\log^3 N)$	✓ / II	HDD/SSD
L-SD_d[sN [31]]	$O(sN)$	$O(N^{\frac{1}{s}}+\log N)$	$O(\log N)$	$O(N^{\frac{1}{s}}/p+\log N)$	$O(s\log^2 N)$	✓ / II	HDD/SSD

Figure 1: Comparison of different *locality-aware* and *page-efficient* DSE schemes. For any function $f(N)$ the nonation $\tilde{O}(f(N))$ denote $O(f(N)(\log f(N))^x)$ for some constant x; n_w denotes the number of updates for a keyword w, and p denotes memory page size and *am.* stands for amortized complexity. **FP** and **BP** stands for forward privacy and backward privacy respectively; all the schemes except the schemes from [55] satisfy **FP** and **BP-II**. The sources of the schemes that $\mathsf{SD}_a[\cdot]$ and L-$\mathsf{SD}_d[\cdot]$ are instantiated with are cited inside the brackets. The schemes marked with * restrict keyword-lists' size to be at most $N^{1-1/\log\log N}$.

it can have a major negative impact, when it is stored on hard-disk. To see why, we need to consider that the I/O-related overhead of accessing disk data consists also of moving the disk head (a "mechanical" component). When using DSE to retrieve a result of R tuples, due to their random placement, causes (close to) R such head "jumps" (i.e. disk seeks in HDDs) if the index is sufficiently large. In this case, this turns out to be the main DSE bottleneck! Reading data from consecutive location costs orders of magnitude less than reading from random positions. In the relevant literature, the number of such jumps required for one DSE search query is referred to as *locality*. Cash and Tessaro [17] identified different interesting trade-offs between *locality* and *read efficiency*, i.e., the number of extra data that needs to be retrieved beyond the result itself (e.g., if we just want to minimize locality we can just retrieve and locally decrypt the entire dataset; clearly not a favorable trade-off in this case). Subsequent works [6,27,31] proposed improved schemes both from a theoretical perspective [6] and with good practical performance [6, 31]. For the case of SSDs, recent works [10, 55] indicate that the search performance depends on another metric which is called *page efficiency*, which is the ratio of the number of pages retrieved for the encrypted results to the number of pages that would have been retrieved for the plaintext result.

Unfortunately, all of these schemes with bounded *locality* and *page efficiency* are restricted in the *static* dataset case, without support for updates. The work closest to ours is the recent work of Minaud and Reichle [55] from CRYPTO'22, which suffers from two important drawbacks. First, its updates are very inefficient–the overhead of adding a single (w,id) tuple is proportional to the number of entries that already contain w. This is as costly as running a search for w, whereas prior DSE achieve constant or, at worst, logarithmic cost w.r.t the dataset size. Second, and arguably most important, the DSE of [55] *is not forward-private*. In practice, it stores all tuples for the same keyword directly in consecutive locations in the encrypted index (which very naturally gives it good locality), so just looking at the location a new entry is stored during an update, the server infers information about whether it is related to prior entries. Forward privacy has been shown to be very important in practice, not only because it allows the dataset to be incrementally built but also because it impairs existing privacy attacks against DSE [73].

Forward-privacy AND "good" locality? At first glance, these two properties seem inherently contradicting. Forward-privacy seeks to "eliminate" any information about new entries that can be inferred from where they are placed in the encrypted index (e.g., using random placement). On the other hand, good locality requires that entries for the same keyword are placed close to each other, ideally in contiguous disk locations! Indeed, the authors of [55] claim that the two properties, *locality* and *forward privacy*, *"seem to be fundamentally at odds."* Prior to this, Bost [11] claimed that the two are *"irreconcilable notions."*

Our results: I/O Efficient DSE with forward and backward privacy. Based on the above observation, it may seem that we cannot hope to achieve both of these properties. In this work, we disprove this by proposing the first DSE schemes that reconcile the two properties achieving bounded (logarithmic or polylogarithmic) *locality* and *page efficiency*, good overall practical performance, and forward-and-backward privacy! Our results can be summarized as follows. First, we observe that the generic "static-to-dynamic" transformation of Demertzis et al. [26], called SD_a, also works as a locality-preserving compiler that can produce forward and backward private DSE from existing *locality-aware*[2] static schemes

[2]Schemes with good locality are referred as *locality-aware* schemes

with just an extra logarithmic overhead for locality. This approach yields the first such DSE in the literature, albeit with "amortized" updates, as the client needs to periodically merge indexes into larger ones. In practice, while most updates are fast, some of them are significantly more resource-consuming.

Second, aiming for schemes with de-amortized updates, we begin with the de-amortized version of SD_a presented as the SD_d scheme in [26]. SD_d relies on oblivious dictionaries [68], [57] for the de-amortization, which makes updates computationally intensive and requires multiple rounds of interaction. In this work, we revisit this de-amortization strategy and *refine* it to make it work for a variety of schemes based on our computationally lighter approach of *oblivious-merge*. Our *oblivious-merge* merges encrypted indexes into bigger ones through multiple oblivious passes and oblivious-sort [5] and compaction operations in order to compute important metadata, decide about the placement of the input-elements, adjust properly the required dummy records and do the final placement. These operation require to operate on elements in chunks/batches, as opposed to the individual, one-by-one manner of SD_d. This necessitates multiple oblivious passes on these chunks and de-amortized oblivious-sort and compaction implementations in a manner that preserves forward and backward privacy. We named this new *locality-aware-deamortized* SD_a to be $\mathsf{L}\text{-}\mathsf{SD}_d$. Using *oblivious sorting* based *oblivious merge*, instead of oblivious dictionaries as [26], asymptotically improves our update overhead by a logarithmic factor and also reduces the amount of interaction.

Third, we instantiate our two transformations $\mathsf{SD}_a[\cdot]$ and $\mathsf{L}\text{-}\mathsf{SD}_d[\cdot]$ with several static schemes with (i) good *locality* (i.e. *locality-aware schemes*), well suited for HDD storage, i.e., the One Choice Allocation (1C), Two Choice Allocation (2C), NlogN from [6] and [31]; (ii) and with good *page efficiency*, well suited for SSD storage, i.e., the NlogN, [31] and Tethys/Pluto from [10]. While $\mathsf{SD}_a[\cdot]$ can be instantiated with any static SE scheme, $\mathsf{L}\text{-}\mathsf{SD}_d[\cdot]$ can be instantiated only with One-Choice-Allocation, Two-Choice-Allocation and NlogN schemes. Figure 1 provides an overview of the asymptotic performance of the multiple schemes we achieve in this way from the literature [26,55]. Interestingly, not only our schemes are the *first* to combine I/O efficiency with forward and backward privacy, but they also asymptotically *outperform* prior (non forward-private) works.

Finally, we experimentally evaluate all of the proposed new DSE in HDD, SSD, and RAM for variable dataset and result sizes, considering both synthetic and real datasets [1]. We plan to make our code publicly available after publication. Our experimental results are very encouraging; it shows DSE with good I/O performance and strong privacy is not only possible but yields schemes with good practical performance. Below, we discuss our evaluation in more detail.

Experimental evaluation. We implemented all of our schemes and compare their search and update computation time with the state-of-the-art forward/backward private DSE schemes in HDD and SSD settings (Section 4). In particular, we implement 1C, 2C, NlogN schemes with $\mathsf{SD}_a[\cdot]$ and $\mathsf{L}\text{-}\mathsf{SD}_d[\cdot]$[3], and compared them with the original PiBAS based SD_a and SD_d of [26]. In terms of search, SD_a[NlogN] has the best performance among the schemes at the cost of more storage. However, all of our amortized (SD_a[1C], SD_a[2C], and SD_a[NlogN]) and de-amortized ($\mathsf{L}\text{-}\mathsf{SD}_d$[1C] and $\mathsf{L}\text{-}\mathsf{SD}_d$[NlogN]) schemes with "good-locality" outperform PiBAS based SD_a and SD_d. I.e., they are up to **two** and **three** orders of magnitude faster in SSD and HDD settings respectively. Regarding the update computation time, PiBAS based SD_a has the worst update time. Turning to the de-amortized schemes, $\mathsf{L}\text{-}\mathsf{SD}_d$[1C] has the best performance in memory and disk and is up to **4**$\times$ and **179**$\times$ faster than SD_d[PiBAS][4] respectively. Although we do asymptotic comparisons of our schemes with LayeredSSE and Local[LayeredSSE] [55] in Figure 1, we did not implement them, as these schemes are rather a theoretical work.

Comparison with additional prior works. Minaud and Reichle [56] independently and concurrently presented two theoretical page-efficient DSE schemes that are forward private. The selection of the construction is based on a relationship between the number of entries, page size, and number of keywords, which leaks more information than our approaches do (setup leakage). In addition, their constructions target only SSDs (and cannot be used for HDDs) require up to $O(N)$ client storage and its updates are very inefficient (similar to [55]). The concept of using oblivious sorting periodically and then de-amortizing it to equalize average and worst-case scenarios has been adopted in previous works, such as in the rebuilding of square-root ORAM [36] and hierarchical ORAMs [60] when inputs exceed trusted client memory. We are not the first to employ and de-amortize oblivious sort. Oblivious-sort was utilized also by [64]. However, their construction does not achieve backward privacy. Their solution was tailored to ensure quasi-optimal search time (i.e., search time independent of the deleted records), which is not our primary objective. Oblivious sort is used to avoid contiguous memory locations of deleted entries through binary search and to obliviously-sort/permute their indexes with sublinear available client memory (mirroring challenges faced by hierarchical and square-root ORAMs in prior works).

Limitations. A parallel line of research concentrates on leakage-abuse attacks for SE [9, 13, 25, 41, 49–51], as well as on defenses and mitigation techniques [28, 42, 65]. Our work does not offer new insights into leakage-abuse attacks or mitigation techniques. Instead, it adheres to the standard forward and backward privacy leakage profile for DSE.

[3] Referred as SD_a[1C],SD_a[2C], $\mathsf{L}\text{-}\mathsf{SD}_d$[NlogN] etc.

[4] SD_d is the original version of [26] based on PiBAS, and SD_d[PiBAS] is our new framework instantiated with PiBAS, which are equivalent.

2 Preliminaries

Notation. Let $(x';y') \leftrightarrow P(x;y)$ denote a protocol execution between a client and a server, which may consist of multiple rounds of communication, and $(x';y') \leftarrow A(x;y)$ denote an algorithm execution, with no communication between client and server—x and x' are the input and output for the client, and y and y' are the input and output for the server. We denote by $\lambda \in \mathbb{N}$ a security parameter and by $v(\lambda)$ a negligible function in λ. PPT stands for Probabilistic Polynomial-Time. $\mathcal{D}$ is a collection of n documents with identifiers $id_1,\dots,id_n$. A document contains a set of keywords from a dictionary Δ. Let DB consist of N tuples of the form (w,id,op)–file id contains keyword w, and op is either add or del, which denotes whether the tuple is for an insertion or deletion. $DB(w)$ is the set of identifiers of documents that contain keyword w. We use n_w to denote $|DB(w)|$, i.e. the result size of keyword w. The aforementioned tuples can be expanded to $(w,id,op,rank,n_w)$, where $0 \leq rank < n_w$. EDB denotes the encrypted database/index stored in the server.

Pseudo Random Functions (PRFs) [46]. A PRF function $F:\{0,1\}^\lambda \times \{0,1\}^* \rightarrow \{0,1\}^*$ is a two input function where the first input is the *key* and the second is the input x. F can be distinguished from a truly random function by a PPT adversary only with negligible probability $v(\lambda)$.

Dynamic Searchable Encryption (DSE). A DSE scheme Σ consists of a `Setup` algorithm, and (possibly interactive) protocols `Search` and `Update`–$\Sigma = ($`Setup`, `Search`, `Update`$)$.

- $(K,\sigma;EDB) \leftarrow$ `Setup`(λ,N) takes as input the security parameter λ and N. Returns EDB to the server, and the secret key K and local state σ to the client.
- $(res,K,\sigma;EDB) \leftrightarrow$ `Search`$(K,w,\sigma;EDB)$ is a protocol for searching keyword w. The output of this protocol is the query result res (i.e., $DB(w)$). The protocol may or may not modify K, σ and EDB.
- $(K,\sigma;EDB) \leftrightarrow$ `Update`$(K,(w,id,op),\sigma;EDB)$ is a protocol that inserts/removes (w,id) to/from the DB–$op = add/del$. The protocol may modify K, σ and EDB. This protocol modifies EDB and may modify K and σ.

The `Search` and `Update` algorithm/protocols for the schemes presented in section 3.2 also include an additional parameter Γ, which is an instance of a static SE scheme[5]

Following [11,12,26,35], we start from an empty database. Given an input DB of size N, the client populates EDB by calling the `Update` protocol N times. Other works [33,47] equivalently modeled updates in document granularity, i.e., inserting or deleting an entire document. We focus on retrieving only the document identifiers upon `Search`; the client may retrieve the documents separately if needed. This leads to a straightforward leakage formulation and is more natural for database queries (e.g., [28,32,44]).

DSE–Leakages & Forward/Backward privacy. The standard security of a DSE scheme is parametrized by a leakage function $\mathcal{L} = (\mathcal{L}^{Stp},\mathcal{L}^{Srch},\mathcal{L}^{Updt})$. $\mathcal{L}^{Stp}$ corresponds to the leakage during the setup phase–in our case it reveals size of the database N; $\mathcal{L}^{Srch}$ corresponds to the leakage during search queries; $\mathcal{L}^{Updt}$ corresponds to the leakage during update queries. *Search pattern* leakage reveals which searches are related to the same w, and *access pattern* leakage reveals $DB(w)$ during a search for w. *Access pattern* leakage is unavoidable if the client retrieves the actual files with an additional round of communication with the server; schemes that avoid this leakage (e.g. by storing files in oblivious data structures) are referred as *result hiding* schemes.

A secure DSE scheme with leakage $\mathcal{L}$ should reveal nothing to an adaptive PPT[6] adversary about the database DB other than the leakage $\mathcal{L}$. This is formally captured by a standard real/ideal experiment presented in the extended version—for more details see [12,35,64].

Forward and backward privacy [12,64] have become the de-facto security guarantees for modern DSE.

Forward privacy (**FP**): limits the information revealed due to updates. In particular, an $\mathcal{L}$-adaptively secure DSE is forward private *iff* the update leakage function $\mathcal{L}^{Updt}$ can be written as: $\mathcal{L}^{Updt}(op,w,id) = \mathcal{L}'^{Updt}(op,id)$ where $\mathcal{L}'$ is a stateless function, $op = add/del$, and id is a file identifier. Particularly, it should be impossible to tell whether an insertion is for a new keyword or a previously inserted/searched one.

Backward privacy (**BP**): ensures that during searches the server does not learn the identifiers of deleted documents that contained the searched keyword w. Bost et al. [12] proposed various formulations for this property. Here, we target backward private schemes that reveal the identifiers of (non-deleted) documents currently containing w (known as $TimeDB(w)$ leakage) and the timestamps and type (i.e. insertion and deletion) of all prior updates for w (known as $Updates(w)$ leakage). This corresponds to the **BP-II** definition from [12] (see the extended version for more details). $TimeDB(w)$ accounts for the leakage from retrieving the actual files; but we focus only on retrieving the document identifiers during `Search`, and we never use it in our proofs as none of our constructions explicitly leaks $TimeDB(w)$.

Definition 1 ([12]). *A DSE scheme Σ is $\mathcal{L}$-adaptively-secure with forward and backward privacy, iff $\mathcal{L}^{Updt}(op,w,id) = \mathcal{L}'(op)$ and $\mathcal{L}^{Srch}(w) = \mathcal{L}''(TimeDB(w),Updates(w))$* ***and*** *iff for any adaptive* PPT *adversary* Adv *issuing polynomially many queries q, there exists a stateful* PPT *simulator* $\mathsf{Sim} = (SimSetup,\ SimSearch,\ SimUpdate)$ *such that* $|\Pr[\mathsf{Real}^{\mathsf{DSE}}_{\mathsf{Adv}}(\lambda,q) = 1] - \Pr[\mathsf{Ideal}^{\mathsf{DSE}}_{\mathsf{Adv},\mathsf{Sim},\mathcal{L}}(\lambda,q) = 1]| < v(\lambda)$,

[5] Γ is a static searchable encryption scheme such that $\Gamma = ($`KeyGen`, `Setup`, `Search`$)$..

[6] An adaptive PPT decides its next step based on its previously observed leakages/search results.

where $\mathcal{L}'$ and $\mathcal{L}''$ are stateless functions; op is insertion or deletion, and id is a file identifier.

DSE for HDDs: Locality/Read-Efficiency/Update Locality/Update Cost/Space. To scale SE to big data using external memory (i.e., HDD drives), SE schemes with "small" locality and read efficiency have been proposed before. *Locality* is defined as the number of *non-contiguous* memory accesses made during the search by the server. *Read-efficiency* is the ratio of the total amount of data read/retrieved during the search over the actual query result size (for querying a keyword w)—see [17] for formal definitions. Cash and Tessaro [17] proved that any secure SE scheme with both optimal locality $O(1)$ and optimal read efficiency $O(1)$ requires $\omega(N)$ space. For locality-aware DSE schemes, in addition to the "search" and "static" efficiency, i.e., (i) locality, (ii) read-efficiency and (iii) space, we focus on two update metrics: (iv) update-locality, i.e., the number of *non-contiguous* memory accesses during an update and (v) update asymptotic cost, i.e., the asymptotic cost of one update (i.e., insert/delete one (w, id, op) tuple). See the extended version for the exact definitions.

DSE for SSDs: Page-Efficiency/Space. Bossuat et al. [10] proposed a new I/O-efficiency dimension for SSDs, which is called *page efficiency*—SSD performance mainly depends on page efficiency, aiming at reading as few memory pages as possible. More formally, *page efficiency* is the ratio of the total number of pages that the server accesses (using SE) over the optimal number of accessed pages in a plaintext case. For page-efficient DSE schemes, in addition to the "search" and "static" efficiency dimensions, i.e., (i) page-efficiency and (ii) space, we focus on (iii) update efficiency, i.e., the number of pages accessed during one update (i.e., insert/delete one (w, id, op) tuple) and (iv) update asymptotic cost (i.e., asymptotic cost of insert/delete of one (w, id, op) tuple). See the extended version for the exact definitions.

Oblivious sort. An oblivious sorting algorithm sorts an array of N elements without leaking information about the relative ordering of the input elements [4, 8, 38, 59]. We use the oblivious bucket sort [5], which has $O(N \log N)$ time complexity, and uses a temporary client storage of two buckets, where one bucket contains 512 elements. We can decompose bucket sort into $N \log N/\gamma$ rounds of $O(\gamma)$ work per round (e.g., for $\gamma = O(\log N)$ the oblivious bucket sort can be completed in N rounds). We highlight that fetching $O(\gamma)$ elements from disk requires always $O(1)$ locality, since accesses are performed on a bucket granularity, and on consecutive buckets— see the extended version for more details.

Oblivious compaction. Given an array of N elements, some of which are tagged with bit 1, and the rest with bit 0, an oblivious compaction [37, 39, 58] generates an output in which all the elements tagged with bit 1 appear before all the elements tagged with bit 0, without leaking the information of which elements were tagged with bit 1 or 0. Our constructions need the compaction to be order preserving, i.e. the relative ordering of the elements tagged with bit 1/0 do not change after the compaction. We use the oblivious bucket sort [5] mentioned above for performing order preserving compaction.

Oblivious Maps(OMAP) [24, 68]. An oblivious map is a data structure that supports oblivious read/get and write/put functionality for encrypted $(key, value)$ pairs (i.e., oblivious hash map). That is, all same-length operation sequences appear indistinguishable. OMAP has the following algorithms/protocols: (i) OMAP.`Setup` initializes an empty data structure with maximum capacity N blocks, (ii) OMAP.`put` adds/overwrites a $(key, value)$ pair, and (iii) OMAP.`get` returns the *value* of a given *key*. We have used the AVL tree based implementation by Wang et al. [68] on top of PathORAM [65]. For a map with capacity N, each oblivious access requires $O(\log^2 N)$ operations/accessed-blocks, $O(\log N)$ roundtrips, and $O(\log^2 N)$ locality—see [68] for more details.

One-Choice Allocation. Asharov et al. [6] presented the One-Choice Allocation scheme (we will refer to it as 1C) that offers optimal locality, optimal space overhead and $O(\log N \log\log N)$ read efficiency. Given a database of size N, this scheme allocates an array of $m = N/\log N \log\log N$ bins, each of size $3\log N \log\log N$[7]. For each keyword w it computes a hash value $h(w)$, and stores the ith document identifier in the bin $(h(w)+i) mod\ m$. Bins are filled up with *dummy* entries, if not full. The formal description of 1C is provided the extended version.

Two-Choice Allocation. Asharov et al. also presented the Two-Choice Allocation (2C) in [6]. For a database of size N, this scheme offers optimal locality and space overhead and $O((\log\log N)^A (\log\log\log N)^2)$ read efficiency assuming keyword list sizes $< N^{1-1/(\log\log N)^A}$, for a constant $A \geq 1$. This scheme allocates an array of $m = N/(\log\log N)^A(\log\log\log N)^2$ bins, each bin with size $z \cdot (\log\log N)^A\ (\log\log\log N)^2$, where $2 \leq z \leq 4$ (we use $A = 1$). The keyword lists are padded to be nearest power of 2 and stored in decreasing length order. For list for w, first it divides bins into groups of $sb = \frac{m}{n_w}$ *superbins*; i.e. each *superbin* consists of n_w bins. Then two superbins are chosen with two hash functions: $h_1(w)\%sb$ and $h_2(w)\%sb$. Finally, the superbin that has minimal load stores the list for w. The formal description of 2C is provided the extended version.

NlogN Scheme. The third scheme presented by Asharov et al. in [6] offers optimal locality and read efficiency at the cost of $O(\log N)$ storage overhead, i.e. a total of $O(N \log N)$ storage. We highlight this scheme provides optimal page efficiency. This scheme consists of $(\log N + 1)$ hash tables, each of size N. The keyword-lists are padded so that their length becomes nearest power of 2. The kth hash-table stores lists of size 2^k. The pseudocode and more details of the NlogN scheme are

[7] The bins overflow with negligible probability when the bin size is set to $3\log N \log\log N$ for 1C.

provided in the extended version. Demertzis et al. [31] proposed a variation of the NlogN scheme in which only s of the above mentioned hash tables are stored; s is evenly distributed (i.e. the server stores the levels $\{0, x, 2x, \dots (s-1) \cdot x\}$, where x is set to be $\lceil \frac{\log N+1}{s} \rceil$). In the worst case, a keyword-list is divided into smaller $O(N^{\frac{1}{s}})$ chunks. This scheme achieves optimal read efficiency, $O(N^{\frac{1}{s}})$ locality and has a $O(s)$ space overhead, and $O(\min\{N^{\frac{1}{s}}/p, p\})$ page efficiency (p denotes the memory page size). We refer to this scheme as sN.

Encrypted dictionary. In our DSE constructions (in section 3) along with the encrypted index we have an encrypted dictionary. The encrypted dictionary maintains a keyword to keyword-counter mapping, i.e. keyword w maps to $|DB(w)|$. To perform searches in 1C, 2C, NlogN, and sN schemes the keyword-counter is required (to know how many bins to fetch, or which levels to search). We will refer to the encrypted dictionary as $EDB.\textsc{Dict}$ and the encrypted index as $EDB.\textsc{Ind}$. For an index with N entries the size of $EDB.\textsc{Dict}$ is at most N, as there can be at most N distinct keywords.

3 DSE—I/O efficiency meets Forward/Backward Privacy

This section introduces our I/O efficient DSE schemes, which are the first to simultaneously achieve good I/O performance and forward/backward privacy. In Section 3.1, we apply the SD_a transformation, as proposed by Demertzis et al. [26], in conjunction with I/O efficient static SE schemes. In Section 3.2, we address the more challenging task of offering de-amortized I/O efficient constructions that ensure forward/backward privacy. Here, we introduce a novel *oblivious merge* framework to meet these objectives and present three implementations of this framework: the dynamic de-amortized 1C, 2C, and NlogN schemes.

3.1 Amortized Constructions using SD_a [26]

Overview of $\mathsf{SD}_a[\cdot]$. Demertzis et al. [26] introduced a compiler, SD_a, that converts any result hiding static SE scheme into a forward/backward private DSE scheme. The essence of SD_a is to store the results of N updates across $\log N$ indexes (ranging from $0th$ to $(\log N - 1)th$). The ith index (EDB_i) has a size of 2^i. With each update, the client initializes the static SE scheme (using SE's `Setup`) to produce and upload an encrypted index with size 1 (EDB_0). When two same-sized indexes emerge (in server), the client downloads, decrypts and combines them into a doubled size index, amortizing the cost of updates. Searches are performed in each index independently. Figure 2 shows an update example using SD_a. The optimized pseudocode for SD_a, as depicted in Figure 2, can be found in the extended version.

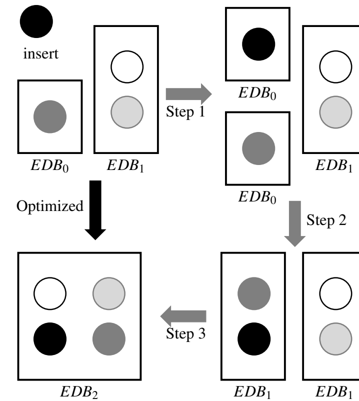

Figure 2: Update in SD_a. Before the insertion of the new (black) entry three previous consecutive insertions have created EDB_0. and EDB_1. After the fourth insertion, two EDB_0 indexes exist (Step 1), which are downloaded and merged to a single EDB_1 of size 2 (Step 2). Now, two EDB_1 indexes exist, which are downloaded and merged to a single EDB_2 of size 4 (Step 3). The black arrow shows an optimized version of SD_a where the intermediate steps are skipped.

In [26], the authors use PiBAS [14] as the underlying static SE scheme. PiBAS maps every (w, id, op) tuple to a pseudorandom location (using a counter cnt_w that is maintained locally for each keyword). During search for w, all locations for counter values $1, \dots, cnt_w$ are accessed to retrieve the results. The results are decrypted and the deleted entries are filtered locally at the client. While this gives us a very simple and lightweight scheme, when it comes to I/O overhead, due to the pseudorandom placement, the number of disk head movements (HDD case) and the number of fetched pages (SDD case) is equal to the result size. In other words, $\mathsf{SD}_a[\cdot]$ when applied to PiBAS gives a DSE with the worst possible *search locality* and *page efficiency*.

I/O-efficient $\mathsf{SD}_a[\cdot]$. Our main observation here is that when applied to an *I/O-efficient* static scheme, the SD_a transformation gives dynamic schemes with "good" I/O performance. In fact, in the extended version we prove two theorems that state that the resulting $\mathsf{SD}_a[\Gamma]$ transformation will retain same *space-overhead* as that for Γ, where as for *locality*, *read-efficiency* (for HDDs) and *page-efficiency* (for SSDs) an additional factor of $O(\log N)$ is introduced as each index is queried independently

Locality-aware $\mathsf{SD}_a[\cdot]$ (for HDDs). There exist various candidates for static locality-aware schemes, the 1C and 2C schemes from [6] achieve optimal *locality*, storage, and polylogarithmic *read-efficiency*. The NlogN scheme from [6] provides optimal *locality* and *read-efficiency* but increases storage. Other recent schemes are also introduced in [7, 27, 31].

Page-Efficient $\mathsf{SD}_a[\cdot]$ (for SSDs). Tethys [10] and NlogN [6] are candidates for static page-efficient schemes. Table 1

$\underline{(K,\sigma;EDB) \leftarrow \texttt{Setup}(\lambda,N)}$

1: $\ell \leftarrow \lceil \log N \rceil$; $upd_{cnt} \leftarrow 0$;
2: Set matrix P of size $(\ell+1)\cdot 4$
3: $k_{rnd} \leftarrow \text{RND}.\texttt{KeyGen}(1^\lambda)$
4: $\sigma \leftarrow \{upd_{cnt}\}$ and $K \leftarrow (k_{rnd}, P)$
5: Initialize an empty EDB
6: **return** EDB to Server and (K,σ) to the Client

$\underline{DB(w) \leftrightarrow \texttt{Search}(K,\Gamma,q,\sigma;EDB)}$

Client $\leftrightarrow$ Server:
1: $\mathcal{X} \leftarrow \emptyset$
2: **for** $i = \ell \cdots 0$ **do**
3: **if** $\mathsf{OLDEST}_i.\textsc{Ind} \neq \emptyset$ **then**
4: $\mathcal{X} \leftarrow \mathcal{X} \cup \Gamma.\texttt{Search}(K[i][0], q, \sigma; \mathsf{OLDEST}_i)$
5: **if** $\mathsf{OLDER}_i.\textsc{Ind} \neq \emptyset$ **then**
6: $\mathcal{X} \leftarrow \mathcal{X} \cup \Gamma.\texttt{Search}(K[i][1], q, \sigma; \mathsf{OLDER}_i)$
7: **if** $\mathsf{OLD}_i.\textsc{Ind} \neq \emptyset$ **then**
8: $\mathcal{X} \leftarrow \mathcal{X} \cup \Gamma.\texttt{Search}(K[i][2], q, \sigma; \mathsf{OLD}_i)$

Client:
9: $DB(w) \leftarrow \{id \mid (w,id,add) \in \mathcal{X} \wedge (w,id,del) \notin \mathcal{X}\}$

Figure 3: SD_d/L-SD_d: from Γ=(KeyGen, Setup, Search) to DSE (de-amortized version).

shows various SD_a instances from *locality-aware* and *page-efficient* static schemes. In this work, we emphasize the most practical I/O-efficient static SE schemes: 1C and 2C (suitable for HDDs) and NlogN(compatible with both HDDs and SSDs). The security and forward/backward privacy of the DSE schemes resulting from the $\mathsf{SD}_a[\cdot]$ transformation are directly derived from [26, Theorem 1]. In the extended version we provide a more detailed discussion for $\mathsf{SD}_a[\cdot]$.

3.2 De-amortized constructions: L-$\mathsf{SD}_d[\cdot]$

Although $\mathsf{SD}_a[\cdot]$ provides a clean way to transform a static SE scheme to a dynamic one, its drawback is that it has amortized update cost. In the worst case an update may require rebuilding an index of size N at once. Figure 16 illustrates the above weakness of schemes with amortized update cost. In this section, we construct I/O efficient DSE schemes with de-amortized updates and with forward and backward privacy. Our starting point is the de-amortization strategy that Demertzis et al. [26] proposed for SD_a, called SD_d (which is tailored on PiBAS [14]; pseudocode of PiBAS can be found in the extended version. In this work, we introduce L-SD_d, which is simpler than SD_d and separates the static-to-dynamic transformation from the underlying static SE scheme. We highlight that L-SD_d is not a compiler; it is specifically tailored for certain schemes, namely 1C, 2C, and NlogN schemes. We present L-SD_d primarily as a framework for the sake of clarity and concise presentation. Below, we explain the new L-SD_d:

Overview of L-$\mathsf{SD}_d[\cdot]$. The main idea of L-$\mathsf{SD}_d[\cdot]$ framework is to split construction steps of index level i over the previous 2^i insertions. We use a static SE scheme Γ=(KeyGen, Setup, Search). For dataset size N, L-$\mathsf{SD}_d[\Gamma]$ keeps $\log N$ index levels. Every ith index level has 4 indexes (a.k.a. NEW_i, OLD_i, OLDER_i and OLDEST_i); every index stores 2^i real entires. The L-$\mathsf{SD}_d[\Gamma]$.Search simply queries the OLD_i, OLDER_i and OLDEST_i indexes by individually calling Γ.Search for all $\log N$ levels as a black box (see Search in Figure 3). Index NEW_i is used as a temporary buffer for moving elements between levels $i-1$ and i during L-$\mathsf{SD}_d[\Gamma]$.Update. Below, we explain L-$\mathsf{SD}_d[\Gamma]$.Update in three rules:

<u>(Rule 1):</u> An update for (w,id,op) (insert or delete) is performed by creating NEW_0 (i.e. an encrypted index $\mathsf{NEW}_0.\textsc{Ind}$ and an encrypted dictionary $\mathsf{NEW}_0.\textsc{Dict}$) by calling Γ.KeyGen and Γ.Setup in a black-box manner (see lines 11-12 in Figure 4)—this instantly triggers the next rule.

<u>(Rule 2):</u> If $\mathsf{NEW}_i.\textsc{Ind}$ is full, then NEW_i is moved[8] to the oldest *empty* index among OLDEST_i,OLDER_i, and OLD_i (see lines 13-15 for level 0 and lines 5-10 for level i in Figure 4). At the end of 2^i updates (equivalently moving $2\cdot 2^{i-1}$ elements from level $(i-1)$) $\mathsf{NEW}_i.\textsc{Ind}$ becomes full, which triggers two actions in level $(i-1)$: OLD_{i-1} is moved to OLDEST_{i-1}, and OLDER_{i-1} is set to be empty (line 6).

<u>(Rule 3):</u> If both $\mathsf{OLDER}_{i-1}.\textsc{Ind}$ and $\mathsf{OLDEST}_{i-1}.\textsc{Ind}$ are full, all the elements of $\mathsf{OLDER}_{i-1}.\textsc{Ind} \cup \mathsf{OLDEST}_{i-1}.\textsc{Ind}$ are moved to $\mathsf{NEW}_i.\textsc{Ind}$ over the next 2^i updates. Thus, at the end of these 2^i updates $\mathsf{NEW}_i.\textsc{Ind}$ is full, and hence NEW_i is moved to the oldest *empty* index in level i using Rule 2.

We demonstrate the above L-$\mathsf{SD}_d[\Gamma]$ rules graphically in Figure 5. So far, all the actions needed for enforcing the Rules 1 and 2 during L-$\mathsf{SD}_d[\Gamma]$.Update and L-$\mathsf{SD}_d[\cdot]$.Search are either agnostic to the static SE scheme Γ, or are using Γ in a black-box manner. We mention that the actions needed for L-$\mathsf{SD}_d[\Gamma]$.KeyGen are also agnostic to Γ (see Figure 3).

We would like to emphasize that up to this point, L-SD_d is identical to SD_d [26]. Below, we detail how L-SD_d differs from SD_d. Rule 3 gradually builds NEW_i by moving elements from $\mathsf{OLDER}_{i-1}.\textsc{Ind}$ and $\mathsf{OLDEST}_{i-1}.\textsc{Ind}$, i.e. Γ.Setup for NEW_i needs to be executed in an incremental fashion. In other words, Γ.Setup cannot be used as a black-box here. To overcome this obstacle, we define a new *oblivious merge* protocol as follows:

$$\mathsf{NEW}_i \leftarrow \texttt{oblMerge}_i(K,\sigma,\mathsf{OLDEST}_{i-1},\mathsf{OLDER}_{i-1})$$

For each index level i, an instance of *oblivious merge* (denoted as $\texttt{oblMerge}_i$) needs to be executed. The $\texttt{oblMerge}_i$ protocol takes as input a structure of necessary keys as K (created and maintained from L-$\mathsf{SD}_d[\cdot]$), client's local state σ, OLDEST_{i-1} and OLDER_{i-1}, and creates NEW_i. We claim that if a static Γ.Setup can be transformed to an $\texttt{oblMerge}_i$ with the following three properties then it can be used in conjunction with the L-SD_d's update functionality to de-amortize

[8] $\mathsf{OLDEST}_i \leftarrow \mathsf{NEW}_i$ is syntactic sugar for $(\mathsf{OLDEST}_i.\textsc{Ind}, \mathsf{OLDEST}.\textsc{Dict}) \leftarrow (\mathsf{NEW}_i.\textsc{Ind}, \mathsf{NEW}_i.\textsc{Dict})$

$(K,\sigma;EDB) \leftrightarrow$ `Update`$(K,(w,id,op),\Gamma,\sigma;EDB)$

Client $\leftrightarrow$ Server:

1: Parse K as (k_{rnd},P) ▷ k_{rnd} is encryption key, P is an array of PRF keys
2: **for** $i=\ell,\ldots,1$ **do**
3: **if** $\mathsf{OLDEST}_{i-1}.\textsc{Ind} \neq \emptyset \wedge \mathsf{OLDER}_{i-1}.\textsc{Ind} \neq \emptyset$ **then** ▷ Γ is $\{\mathsf{PiBAS},\mathsf{1C},\mathsf{2C},\mathsf{NlogN}\}$
4: Execute the next γ steps of $\Gamma.\mathtt{oblMerge}(K,\sigma,\mathsf{OLDEST}_{i-1},\mathsf{OLDER}_{i-1})$ and update NEW_i
5: **if** $\mathsf{NEW}_i.\textsc{Ind}$ is full **then** ▷ Client can deduce this from upd_{cnt}
6: Server sets $\mathsf{OLDEST}_{i-1} \leftarrow \mathsf{OLD}_{i-1}$ and $\mathsf{OLDER}_{i-1} \leftarrow \emptyset$
7: **if** $\mathsf{OLDEST}_i.\textsc{Ind} = \emptyset$ **then** server sets $\mathsf{OLDEST}_i \leftarrow \mathsf{NEW}_i$ and client sets $P[i][0] \leftarrow P[i][3]$
8: **else if** $\mathsf{OLDER}_i.\textsc{Ind} = \emptyset$ **then** server sets $\mathsf{OLDER}_i \leftarrow \mathsf{NEW}_i$ and client sets $P[i][1] \leftarrow P[i][3]$
9: **else** server sets $\mathsf{OLD}_i \leftarrow \mathsf{NEW}_i$ and client sets $P[i][2] \leftarrow P[i][3]$
10: Client sets $P[i][3] \leftarrow \Gamma.\mathtt{KeyGen}(1^\lambda)$
11: Client sets $P[0][3] \leftarrow \Gamma.\mathtt{KeyGen}(1^\lambda)$
12: Client runs $\Gamma.\mathtt{Setup}(P[0][3],\ (w,id,op))$ and sends the output to server who stores it as NEW_0
13: **if** $\mathsf{OLDEST}_0.\textsc{Ind} = \emptyset$ **then** server sets $\mathsf{OLDEST}_0 \leftarrow \mathsf{NEW}_0$ and client sets $P[0][0] \leftarrow P[0][3]$
14: **else if** $\mathsf{OLDER}_0.\textsc{Ind} = \emptyset$ **then** server sets $\mathsf{OLDER}_0 \leftarrow \mathsf{NEW}_0$ and client sets $P[0][1] \leftarrow P[0][3]$
15: **else** Server sets $\mathsf{OLD}_0 \leftarrow \mathsf{NEW}_0$ and client sets $P[0][2] \leftarrow P[0][3]$
16: Client sets $upd_{cnt} \leftarrow upd_{cnt}+1$
17: **return** EDB to Server and (K,σ) to Client

Figure 4: SD_d/$\mathsf{L\text{-}SD}_d$: from Γ=(`KeyGen`, `Setup`, `Search`) to DSE (de-amortized version).

the corresponding $\mathsf{SD}_a[\Gamma]$ construction (maintaining forward and backward privacy). These properties are:

- (P1): *Obliviousness*—$\mathtt{oblMerge}_i$ has to be oblivious—an algorithm/protocol is oblivious iff for any two same-sized sequence of memory accesses their resulting access patterns are indistinguishable for anyone but the client. In other words, we assume the existence of a simulator, $\mathtt{Sim\text{-}oblMerge}_i$, which takes as an input only the size of the ith index (i.e. 2^i) and is indistinguishable from $\Gamma.\mathtt{oblMerge}_i$ by any PPT adversary (later, we prove that our $\Gamma.\mathtt{oblMerge}_i$ is oblivious for $\Gamma \in \{\mathsf{1C},\mathsf{2C},\mathsf{NlogN}\}$).

- (P2): *Input-Output Indistinguishability*—A PPT adversary can recover a "mapping" between the encrypted input entries (contained in $\mathsf{OLDEST}_{i-1}.\textsc{Ind}$ and $\mathsf{OLDER}_{i-1}.\textsc{Ind}$) and the corresponding output ones (in $\mathsf{NEW}_i.\textsc{Ind}$) after the execution of $\mathtt{oblMerge}_i$ only with a negligible advantage.

- (P3): *Decomposability*—Given $\mathtt{oblMerge}_i$ that requires $s(2^i)$ steps to produce NEW_i, can be decomposed into $s(2^i)/\gamma$ rounds/steps of γ work at a time (i.e., per update).

We need (P1) in order to achieve forward privacy. If $\mathtt{oblMerge}_i$ is oblivious, then nothing is leaked about the elements that we are moving from level $(i-1)$ to level i, i.e., during an individual update/movement the leakage function $\mathcal{L}^{Updt}$ can be written as: $\mathcal{L}^{Updt}(op,w,id) = \mathcal{L}'^{Updt}(op,id)$ where $\mathcal{L}'$ is a stateless function. (P2) is not required for the security of $\mathsf{L\text{-}SD}_d[\Gamma]$, e.g., the original PiBAS-based $\mathsf{L\text{-}SD}_d$ in [26] does not achieve (P2) (explained later). However, we will see that (P2) is byproduct of (P1) when Γ is I/O efficient, as well as it simplifies the security analysis of our $\mathsf{L\text{-}SD}_d[\Gamma]$ (the simulator can use the search simulator of Γ in black box).

Regarding (P3), every algorithm/protocol naturally can be decomposed into instructions (e.g., CPU instructions), or sets of instructions. By maintaining a state locally (e.g. a counter for each ith level), and running a set of instructions at a time, we guarantee the correct execution of the algorithm/protocol. (P3) allowed us to describe $\mathsf{L\text{-}SD}_d[\Gamma]$ independent to the used static scheme Γ and simplify the presentation of $\Gamma.\mathtt{oblMerge}_i$.

The original `Update` protocol of SD_d in [26], which was PiBAS-specific SD_d, achieves properties (P1) and (P3). We can also transform PiBAS (and its setup) to an `oblMerge` protocol in order to fit to the the new $\mathsf{L\text{-}SD}_d[\cdot]$ transformation. For completeness, we provide PiBAS.`oblMerge` in the extended version. Both versions, SD_d presented in [26] and SD_d[PiBAS] fail to achieve (P2) (i.e. *input-output indistiguishability*). For moving each input entry e (from $\mathsf{OLDEST}_{i-1}.\textsc{Ind} \cup \mathsf{OLDER}_{i-1}.\textsc{Ind}$)—one by one, it simply does two oblivious accesses to an oblivious dictionary, and it performs a "write" in a random position in NEW_i (using a PRF), i.e. the server can see the corresponding chosen positions in NEW_i. However, the security does not fail since the positions in NEW_i are chosen randomly. We can also construct $\mathsf{L\text{-}SD}_d$[PiBas] that achieves all the properties, including (P2). This construction improves the update cost of the original SD_d by a logarithmic factor (the pseudocode provided in the extended version). The proof has been omitted because it can be derived from the $\mathsf{L\text{-}SD}_d$ proof, as it is based on a subset

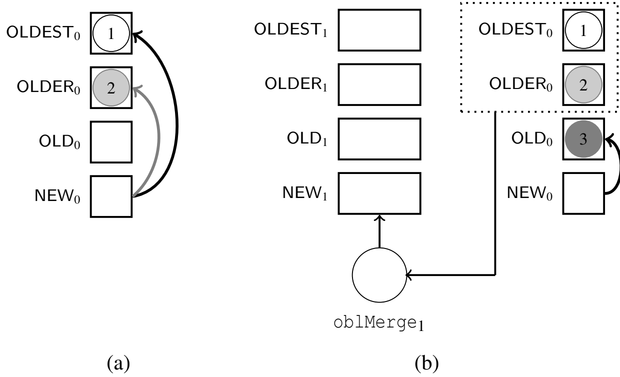

Figure 5: SD_d/$\mathsf{L\text{-}SD}_d[\Gamma]$ framework. (a) The first element inserted to NEW_0 (Rule 1). NEW_0 becomes full and it is moved to OLDEST_0 (Rule 2). Similarly, the second inserted element to NEW_0 (Rule 1) is moved to OLDER_0 (Rule 2). (b) During the third insertion, both OLDEST_0 and OLDER_0 are full and Γ.`oblMerge`$_1$ is called (Rule 3) to move elements from $\mathsf{OLDEST}_0 \cup \mathsf{OLDER}_0$ to NEW_1. The third insertion is moved to OLD_0 (Rules 1 and 2).

of the computations required by $\mathsf{L\text{-}SD}_d$. In our experimental evaluation, we refer to this construction as $\mathsf{L\text{-}SD}_d$[PiBas].

Phases of `oblMerge` framework. Now we will present our `oblMerge` framework that ensures properties (P1), (P2), and (P3) for the static SE scheme $\Gamma \in \{$1C, 2C, NlogN$\}$, in detail. Similar to SD_a, we have $\mathsf{L\text{-}SD}_d$[1C] and $\mathsf{L\text{-}SD}_d$[2C] (suitable for HDDs), as well as $\mathsf{L\text{-}SD}_d$[NlogN] (compatible with both HDDs and SSDs). The corresponding is presented in Figure 6. Some of the steps are not necessary for all the schemes in $\{$1C, 2C, NlogN$\}$, but we needed them in order to create the general `oblMerge` framework. Towards the end of this section we discuss some optimization specific to each scheme. Here, we also would like to point out, the temporary client storage cost is upper bounded by the update cost (e.g. $O(\log^2 N)$ for $\mathsf{L\text{-}SD}_d$[1C]), while the permanent client storage is constant.

The high level idea is, instead of moving elements directly from $\mathsf{OLDER}_{i-1}.\textsc{Ind} \cup \mathsf{OLDEST}_{i-1}.\textsc{Ind}$ to $\mathsf{NEW}_i.\textsc{Ind}$ (using oblivious dictionaries, as done in [26]), move them to a buffer $\mathbf{BUF}_1$ first. This step will ensure property (P2). Then with oblivious operations, re-order the entries of $\mathbf{BUF}_1$ to satisfy the I/O efficiency requirements of scheme Γ, and then move $\mathbf{BUF}_1$ to $\mathsf{NEW}_i.\textsc{Ind}$. This step and the previous step together ensure property (P1). The buffer, $\mathbf{BUF}_2$, is used to create the keyword dictionary, $\mathsf{NEW}_i.\textsc{Dict}$, in an oblivious fashion as well. The references to OLDER_{i-1} and OLDEST_{i-1} are passed as parameters to the `oblMerge`$_i$ protocol. Each `oblMerge`$_i$ instance runs in four phases. Below we discuss the four phases in detail. Refer to Figure 6 for the pseudocode. If a step in the pseudocode does not mention who is performing it (i.e. Client or Server), it means multiple rounds of communication happens between the client and the server to perform that step. Finally, at the end of 2^i calls to `oblMerge`$_i$ it returns NEW_i (i.e. $\mathsf{NEW}_i.\textsc{Ind}$ and $\mathsf{NEW}_i.\textsc{Dict}$).

Phase 1. Entries of $\mathsf{OLDER}_{i-1}.\textsc{Ind} \cup \mathsf{OLDEST}_{i-1}.\textsc{Ind}$ are first copied (not moved) to $\mathbf{BUF}_1$. Next, $\mathbf{BUF}_1$ is obliviously sorted based on the lexicographic order of the keywords (line 3). The sort places entries for the same keyword adjacent to each other in $\mathbf{BUF}_1$, and the dummy entries are placed at the end. Each entry of $\mathbf{BUF}_1$ is then assigned a *rank* and an n_w value via two linear scans (line 4). Next, each keyword-list in $\mathbf{BUF}_1$ is padded to make their lengths to be nearest power of 2 (line 5). Another oblivious sort is performed on $\mathbf{BUF}_1$, based on the lexicographic order of the keywords, as well as descending order of the n_w values (line 6). This sort places same keyword entries adjacent to each other and entries with higher n_w values are placed before entries with lower n_w values in $\mathbf{BUF}_1$. We provide a more detailed pseudocode for lines 4 and 5 in the extended version.

Phase 2. This phase prepares the elements in $\mathbf{BUF}_1$ for $\mathsf{NEW}_i.\textsc{Ind}$ by adding proper position tags to them (lines 12-13), and prepares $\mathbf{BUF}_2$ for creating $\mathsf{NEW}_i.\textsc{Dict}$ (lines 9-11). For every entry in $\mathbf{BUF}_1$ $(w, id, op, rank, n_w)$ an entry is appended to $\mathbf{BUF}_2$. When $rank = 0$, a real entry (w, n_w, p) is appended to $\mathbf{BUF}_2$. In all other cases a dummy entry $(\perp, \perp, \perp)$ is added (line 11) (because we can neither reveal the length of each keyword-list, nor the number of unique keywords). Here, p is a uniformly generated random key of λ bit. The `Map` function of Γ is called for each entry of $\mathbf{BUF}_1$, which returns a pair $(level, pos)$. For every $\Gamma \in \{$1C, 2C, NlogN$\}$ the pseudocode for the corresponding `Map` function is provided in the extended version. For 1C and 2C the value of *level* indicates the bin number assigned to this entry, and *pos* is 0 by default. Whereas, for NlogN scheme $level (\in \{0, \ldots, i\})$ indicates the array level, and position indicates the position $pos(\in \{0, \ldots, (2^{i-level}\text{-}1)\})$ in the array level. The encrypted entry $(w, id, op, level, pos)$ replaces the existing $(w, id, op, rank, n_w)$ entry in $\mathbf{BUF}_1$.

Phase 3. This phase adds dummy elements (lines 14-15) to $\mathbf{BUF}_1$, beacuse every bin needs to be filled up to their maximum capacity. For 1C and 2C if bin size is b_i then the dummy element $(\perp, \perp, \perp, bin, 0)$ is appended to $\mathbf{BUF}_1$ b_i times. This step is repeated for each bin number *bin*. For NlogN scheme, array level k contains 2^{i-k} lists of size 2^k. Hence, the dummy entry $(\perp, \perp, \perp, k, pos)$ is appended 2^k times for each $pos \in \{0, \ldots, (2^{i-k} - 1)\}$. This step is repeated for levels $k \in \{0, \ldots, i\}$. We add a total of $3 \cdot 2^i$ dummy elements for 1C and 2C and $2^i \cdot \log 2^i$ dummy elements for NlogN. The goal is to add dummy entries for every vacant positions of the bins/levels. But we want to hide the number of real elements in each bin/level, and so the number of dummy elements we add is equal to the index size. Extra dummy elements are discarded in Phase 4.

Phase 4. This phase consists of few steps. The oblivious sort on line 16 sorts $\mathbf{BUF}_1$ based on $(level, pos)$ values. Dummy elements come last, as usual. But now we have more elements assigned to each bin/level than their capacity. Hence, with a linear scan (line 17) unnecessary elements are tagged with

$\Gamma \in \{\mathsf{1C}, \mathsf{2C}, \mathsf{NlogN}\}$
If $\Gamma = \mathsf{1C}$, $N' = 3 \cdot 2^i, m_i = \lceil \frac{2^i}{\log 2^i \log\log 2^i} \rceil$, $\forall level \in \{0 \ldots m_i - 1\}$ $b_{level} = 3 \cdot \log 2^i \log\log 2^i$ and $c_{level} = 0$
If $\Gamma = \mathsf{NlogN}$, $N' = 2^i \cdot \log(2^i + 1)$, $m_i = (i+1)$, $\forall level \in \{0 \ldots m_i - 1\}$ $b_{level} = 2^{level}$ and $c_{level} = 2^{i-level}$
If $\Gamma = \mathsf{2C}$, $N' = z \cdot 2^i, m_i = \lceil \frac{2^i}{(\log\log 2^i)(\log\log\log 2^i)^2} \rceil$, $\forall level \in \{0 \ldots m_i - 1\}$ $b_{level} = z \cdot (\log\log 2^i)(\log\log\log 2^i)^2$ and $c_{level} = 0$, $2 \le z \le 4$

$\mathsf{NEW}_i \leftrightarrow \Gamma.\mathtt{oblMerge}_i(K, \sigma; \mathsf{OLDEST}_{i-1}, \mathsf{OLDER}_{i-1})$
Client $\leftrightarrow$ Server:

//** Phase 1 - Preparing sorted input array **//
1: Client parses K as (k_{rnd}, P) ▷ all encryptions and decryptions are done with key k_{rnd}, P is an array of PRF keys
2: Server initializes arrays $\mathbf{BUF}_1$ of size $3 \cdot N'$, and $\mathbf{BUF}_2$ of size N' to be empty at Server
3: Copy $\mathsf{OLDEST}_{i-1}.\textsc{Ind} \cup \mathsf{OLDER}_{i-1}.\textsc{Ind}$ to $\mathbf{BUF}_1$; obliviously sort $\mathbf{BUF}_1$ w.r.t. lexicographic order of keywords
4: Perform two linear scans (one in reverse and one in correct order) on $\mathbf{BUF}_1$, to add the *rank* and n_w values to each entry of keyword w. The entries now look like $(w, id, op, rank, n_w)$, where $0 \le rank < n_w$.
5: Linearly scan $\mathbf{BUF}_1$ and pad each keyword-list with dummy elements so that its size is nearest power of 2, hide list lengths by padding total size to $2N'$
6: Perform oblivious sort on $\mathbf{BUF}_1$ w.r.t. lexicographic order of the keywords and descending order of the n_w values. Keep first N' elements.

//** Phase 2—Prepare index elements and prepare keyword counters **//
7: **for** each $j = 1 \ldots |\mathbf{BUF}_1|$ **do**
8: Client decrypts $(w, id, op, rank, n_w) \leftarrow \mathrm{RND}.\mathtt{Dec}(k_{rnd}, \mathbf{BUF}_1[j])$
9: Client generates $p \xleftarrow{\$} \{0,1\}^\lambda$
10: **if** $rank = 0$ **then** Client appends $(\mathrm{RND}.\mathtt{Enc}(k_{rnd}, (w, n_w, p)))$ to $\mathbf{BUF}_2$
11: **else** Client appends $(\mathrm{RND}.\mathtt{Enc}(k_{rnd}, (\perp, \perp, \perp)))$ to $\mathbf{BUF}_2$
12: Client generates $((level, pos), \sigma) \leftarrow \mathtt{Map}(P[i][3], w, rank, n_w, \sigma)$ ▷ $pos = 0$ for 1C and 2C
13: Client writes $(\mathrm{RND}.\mathtt{Enc}(k_{rnd}, (w, id, op, level, pos)))$ to $\mathbf{BUF}_1[j]$

//** Phase 3—Add dummies **//
14: **for** $level = 0 \ldots m_i - 1$ **do**
15: for every $pos \in \{0 \ldots c_{level}\}$ call Client appends $(\mathrm{RND}.\mathtt{Enc}(k_{rnd}, (\perp, \perp, \perp, level, pos)))$ to $\mathbf{BUF}_1$ b_{level} times

//** Phase 4—Final placement **//
16: Perform oblivious sort on $\mathbf{BUF}_1$ w.r.t. the integer values of $(level, pos)$ in ascending order
17: Linearly scan $\mathbf{BUF}_1$, and tag **first** b_{level} entries for each $level \in \{0, \ldots, \Sigma.m_i - 1\}$ and $pos \in \{0, \ldots, c_{level}\}$ with 1, and with 0 the rest of them (the existing $(level, pos)$ tags are replaced with 0/1 tags, the entries now look like (w, id, op, x), where $x \in \{0,1\}$)
18: Perform order preserving oblivious compaction on $\mathbf{BUF}_1$; keep first N' entries; discard the 0/1 tags
19: Perform oblivious sort on $\mathbf{BUF}_2$ w.r.t. the random keys in ascending order; keep first 2^i entries; discard random keys
20: Server moves $\mathbf{BUF}_1$ to $\mathsf{NEW}_i.\textsc{Ind}$ ▷ elements in each bin/pos is randomly shuffled
21: **for** each $s \in \mathbf{BUF}_2$ **do**
22: Client decrypts $(w, cnt_w) \leftarrow \mathrm{RND}.\mathtt{Dec}(k_{rnd}, s)$
23: Client generates $(key, value) \leftarrow \mathsf{PiBAS}.\mathtt{Map}((P[i][3], k_{rnd}), w, cnt_w, 1)$ ▷ parameter 1 is used by the PRF F
24: Client writes $\mathsf{NEW}_i.\textsc{Dict}[key] \leftarrow value$
25: **return** NEW_i

Figure 6: $\mathtt{oblMerge}$ framework of $\Gamma \in \{\mathsf{1C}, \mathsf{2C}, \mathsf{NlogN}\}$

a 0; and the entries to be kept are tagged with a 1. This tag replaces the existing $(level, pos)$ tags. Next, an order preserving oblivious compaction (line 18) is performed so that all elements tagged with 0 come at the tail of the output and can be discarded altogether. We use another round of oblivious bucket sort (which is order preserving) for the compaction. NEW_i.IND is created with the remaining entries of $\mathbf{BUF}_1$ (line 20). $\mathbf{BUF}_2$ is also obliviously sorted (line 19) based on the random keys in ascending order. The dummy entries with $p = \perp$ will be placed at the end of the sorted buffer. There can be at most 2^i entries such that $p \neq \perp$, as there can be at most 2^i distinct keywords at index level i. Hence, the first 2^i elements are used to create NEW_i.DICT (lines 21-24).

Locality-aware $\mathsf{L\text{-}SD}_d[\cdot]$. The search-*locality* and search-*read efficiency* of $\mathsf{L\text{-}SD}_d[\Gamma]$ depends on the same of the static SE scheme Γ. For both *locality* and *read efficiency*, a $\log N$ factor is added due to querying $3 \cdot \log N$ dictionaries and calling Γ.`Search` on $3 \cdot \log N$ indexes (for OLDEST, OLDER, and OLD). For example, 1C offers $O(1)$ *locality* and $O(\log N \log\log N)$ *read efficiency*, whereas $\mathsf{L\text{-}SD}_d[\mathsf{1C}]$ offers $O(\log N)$ *locality* and $O(\log N \log\log N + \log N)$ *read efficiency*. Similarly, $\mathsf{L\text{-}SD}_d[\mathsf{NlogN}]$ offers $O(\log N)$ *locality* and $O(\log N)$ *read efficiency*, as opposed to optimal *locality* and *read-efficiency* that is offered by NlogN.

Page-efficient $\mathsf{L\text{-}SD}_d[\cdot]$. We can instantiate $\mathsf{L\text{-}SD}_d[\cdot]$ on SSDs with static page efficient schemes like NlogN [6] and its variations i.e. sN [31]. The *page efficiency* of $\mathsf{L\text{-}SD}_d[\mathsf{NlogN}]$ is $O(\log N)$ vs. the optimal *page efficiency* offered by NlogN, whereas for $\mathsf{L\text{-}SD}_d[\mathsf{sN}]$ *page efficiency* is $O(N^{\frac{1}{s}}/p + \log N)$.

Security and efficiency. We formally state and prove the security of $\mathsf{L\text{-}SD}_d[\Gamma]$ (for $\Gamma \in \{\mathsf{1C}, \mathsf{2C}, \mathsf{NlogN}\}$) in the extended version. The security of $\mathsf{SD}_d[\mathsf{PiBAS}]$ is already proven in [26]. We prove also that when we instantiate a locality-aware static scheme Γ with $\mathsf{L\text{-}SD}_d[\cdot]$, it retains its space overhead, but for *locality* and *read-efficiency* an additional $3\log N$ factor is introduced as $3\log N$ indexes are queried (similar proof for the *page-efficient* $\mathsf{L\text{-}SD}_d[\cdot]$ transformation can be proven)—see the extended version.

1C **Optimization.** In 1C scheme we do not need to make the keyword-lists' sizes to be of power of 2. We also do not need to store the larger keyword-lists into the bins before the smaller lists. Hence the lines 5 and 6 in Figure 6 are not required for 1C. The two linear scans that add *rank* and n_w values to each entry can be also removed (line 4). This is because, after the first oblivious sort the entries for the same keyword are all placed together in adjacent locations. In Phase 2, with the help of a single counter one can count how many occurrences of a particular keyword has been seen. Based on this counter value $\mathbf{BUF}_2$ can be updated accordingly. The same counter value can be used in 1C.`Map` function to compute the bin numbers.The counter needs to be reset to 0 every time a new keyword is observed. We provide the optimized pseudocode for 1C.`oblMerge` in the extended version.

Scheme	Size(GB) for $\lvert DB\rvert = 2^{23}$	Size(GB) for $\lvert DB\rvert = 2^{26}$
$\mathsf{L\text{-}SD}_d[\mathsf{NlogN}]$	49	436
$\mathsf{L\text{-}SD}_d[\mathsf{1C}]$	7.5	60
$\mathsf{SD}_d[\mathsf{PiBAS}]$	5	40

Figure 7: Needed storage for a dataset and encrypted index.

NlogN **Optimization.** Here the order of storing lists does not matter. Thus, the second oblivious sort (line 6) can be omitted in Phase 1 for the NlogN scheme. This can be further optimized by storing only s evenly distributed arrays, instead of $\ell = \log N + 1$ arrays, which is essentially the Ns scheme.

2C **Optimization.** For 2C scheme, the client needs to maintain a map to remember which bin has how many entries. This is because, placement of a keyword-list into a superbin depends on how full the superbin is. There are can be maximum of $m = \lceil N/\log\log N(\log\log\log N)^2 \rceil$ bins. But in $\mathsf{L\text{-}SD}_d$ there are $\log N$ index levels. Hence, to maintain this information the required client storage is $O(m \cdot \log N)$. To achieve a constant client storage, this map can be stored at the server in oblivious maps (see section 2).

Update cost. The `oblMerge` framework shown in Figure 6 mainly consists of three types of building blocks: bucket oblivious sorts, basic **for** loops and linear scans. Linear scans are realized with basic **for** loops. The costliest operation among these is the bucket oblivious sort [5], which can be decomposed into N steps of $O(\log N)$ work per step, assuming an index contains at most N entries (recall the ith index contains 2^i elements). In the extended version we explain how to deamortize the bucket oblivious sort in detail. Even the **for** loops can be executed for $O(\log N)$ iterations during a particular call of $\texttt{oblMerge}_i$ (for the ith index). Overall, an $\texttt{oblMerge}_i$ protocol can be decomposed into N steps of at most $O(\log N)$ work per step. There are $\log N$ levels in $\mathsf{L\text{-}SD}_d[\cdot]$. Hence, the *worst case* update cost is $O(\log^2 N)$ for $\mathsf{L\text{-}SD}_d[\mathsf{1C}]$, and $\mathsf{L\text{-}SD}_d[\mathsf{2C}]$. For $\mathsf{L\text{-}SD}_d[\mathsf{NlogN}]$ the update cost is $O(\log^3 N)$, because it has at most $\log N$ levels per index.

4 Experimental Evaluation

We report the performance of our schemes and compare them with previous state-of-the-art works.

We implemented $\mathsf{SD}_a[\mathsf{1C}]$, $\mathsf{SD}_a[\mathsf{2C}]$, $\mathsf{SD}_a[\mathsf{NlogN}]$, $\mathsf{SD}_a[\mathsf{sN}]$, $\mathsf{L\text{-}SD}_d[\mathsf{1C}]$ and $\mathsf{L\text{-}SD}_d[\mathsf{sN}]$ with approximately $31K$ lines in C++. We used OpenSSL-AES [3] PRF evaluation and semantically secure encryption. We also used Oblivious MAP of [26] for the $\mathsf{SD}_d[\mathsf{PiBAS}]$ implementation, and merge-sort as the last step in the implementation of bucket oblivious sort [5]. We ran our experiments on a machine with Intel Xeon E-2174G 3.8GHz processor, 128GB RAM, 1TB SSD, and 5TB HDD running Ubuntu 20.04 LTS (limited to one

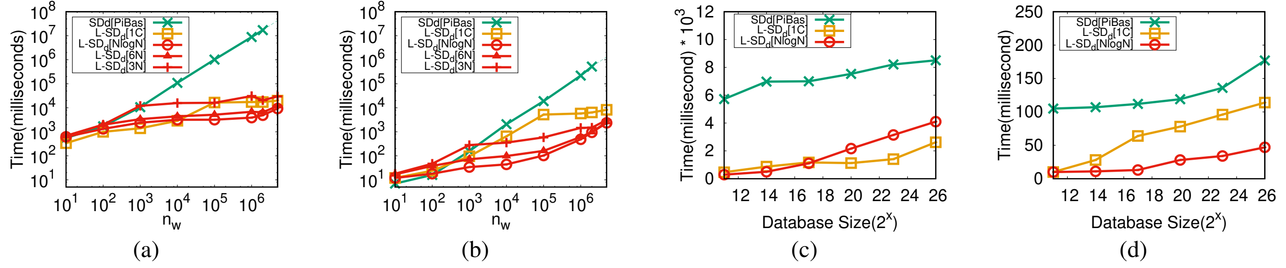

Figure 8: Search computation time for $|DB| = 2^{23}$ and variable result size for (a) $|block| = 32$B in (a) HDD, (b) SSD. Search computation time for variable database size and $n_w = 1K$ in (c) HDD, (d) SSD.

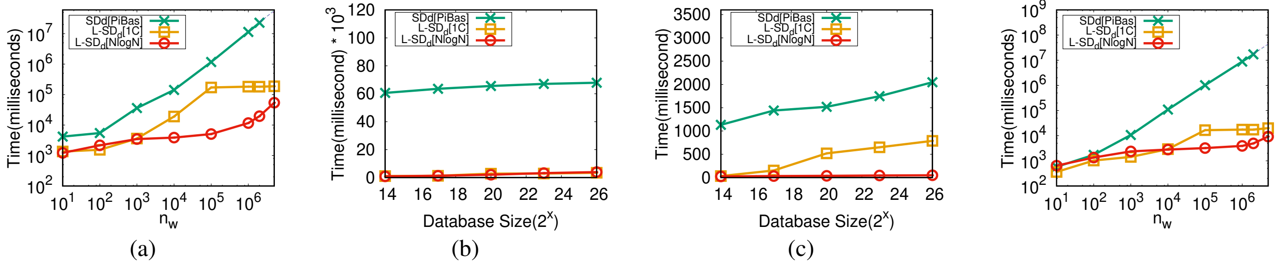

Figure 9: Search computation time for $|DB| = 2^{23}$ and variable result size for (a) $|block| = 512$B in HDD. Search computation time for variable database size and $n_w = 10K$ in (b) HDD, (c) SSD. (d) Search computation time for $|DB| = 2^{23}$, $|block| = 32$B in HDD and variable result size for WAN machines with 24.7ms network delay and 2.5Gbps bandwidth.

CPU core for our experiments). Our code is available online.[9]

We focus on the computation time for Search and Update queries and measure these parameters for variable-size synthetic datasets with $|DB| = 2^{11}$-2^{26} records randomly shuffled before insertion, each time setting the total number of distinct keywords $|W|$ to $|DB|/100$. Likewise, we report results for varying result size n_w between 10 and $5M$ documents. We also consider two block sizes: 32B and 512B. To highlight the significance of I/O in a memory-constrained environment, we conduct our experiments in a cold cache setting, but we also provide two sets of experiments to show the effect of cache on the performance of our schemes. In the first one, we keep the different ratios of datasets in the memory (as cache) and respond to the queries for those encrypted data from memory. Note that we normalize the cache for each scheme. In other words, we fix the cache size and load as much as data we can in that memory. In the second, we assume there are many users (200 for HDD and 75 for SSD) each of which has her own dataset with size 2^{20} and they execute their own search queries randomly when the cache in the system is enabled.

Experiments were repeated five times, and the average result is reported. We also provide a comparison between the needed storage on the server for each de-amortized scheme in Figure 7 for a dataset with $|block| = 32B$ where $|DB| = 2^{23}$ and $|DB| = 2^{26}$. Note that we focus on small client storage schemes. Therefore, client storage is constant.

We also repeated the search experiments on a real dataset consisting of 22 attributes and 6,123,276 records of reported crime incidents in Chicago [1]. We used two different attributes, containing 34 and 170 distinct keywords, respectively, and keyword frequency ranging from 1 record to 1,631,721 records. Finally, we simulated the search and update time of our schemes when run over WAN with 24.7ms delay and 2.5Gbps bandwidth on AWS (between two machines on Ireland and Frankfurt zones).

4.1 Search Performance

Our first set of experiments focuses on search performance and we demonstrate the impact of variable data block sizes, variable result sizes, variable database sizes, and cache size on all our schemes and compare them with SD$_a$[PiBAS] [26] and SD$_d$[PiBAS] [26]. We do not provide the big-block experiments for SSD due to disk space limits.

Variable Block and Result Sizes. Figure 8 (a,b) and Figure 9 (a) show the search computation time of a dataset with size 2^{23} $|block| = 32$B as the result size n_w changes. Similar results for our amortized schemes are in Figure 10 (a),(b),(e).

The conclusions from these experiments are: (i) As expected, SD$_a$[PiBAS] and SD$_d$[PiBAS] have the worst performance among all other schemes for large result sizes due to their poor locality (in HDD) and page efficiency (in SSD). They need to change the position of the hard-drive head or bring different pages of the results, stored at random locations, which leads to significant slowdown. (ii) The SD$_a$[NlogN]

[9] https://github.com/jgharehchamani/DSE-with-IO-Locality

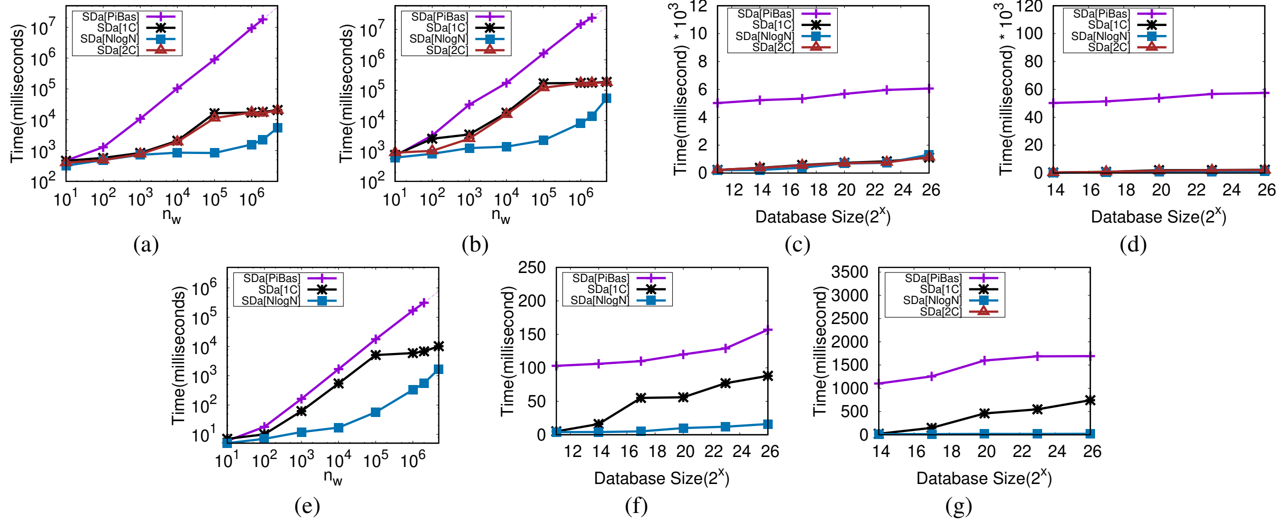

Figure 10: Search computation time for $|DB| = 2^{23}$ and variable result size for (a) $|block| = 32$B in HDD, (b) $|block| = 512$B in HDD. Search computation time for variable database size and (c) $n_w = 1K$ in HDD, (d) $n_w = 10K$ in HDD. Search computation time for $|DB| = 2^{23}$ and variable result size for (e) $|block| = 32$B in SSD. Search computation time for variable database size and (f) $n_w = 1K$ in SSD, (g) $n_w = 10K$ in SSD.

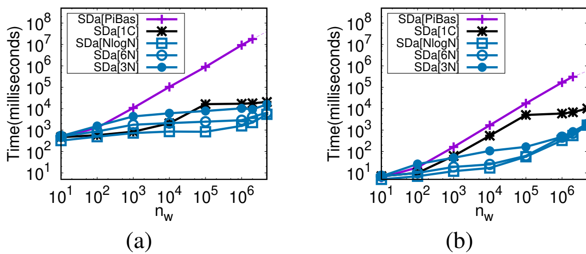

Figure 11: Search computation time for $|DB| = 2^{23}$, variable result size, and $|block| = 32$B for different NlogNsettings in (a) amortized HDD, (b) amortized SSD.

and L-SD$_d$[NlogN] schemes achieve the best performance in the amortized and de-amortized setting, at the cost of extra storage. The number of indexes searched in L-SD$_d[\cdot]$ is three times than that of SD$_a[\cdot]$, hence the search time of L-SD$_d$[NlogN] is slightly more than SD$_a$[NlogN]. (iii) SD$_a$[2C] performs better than SD$_a$[1C] for small result sizes. However, their performance is the same for result sizes more than the bin's threshold, since, after the threshold entries are stored using SD$_a$[1C] (therefore we ignored this scheme in SSD experiments). (iv) The execution time for SD$_a$[1C] for result sizes bigger than 10^5 remains approximately constant, as at this point the database is read in its entirety (i.e. all bins contain results). (v) When the block size is increased from 32B to 512B, the search time of all schemes increases, but the gap between SD$_a$[2C]/SD$_a$[1C] and SD$_a$[PiBAS] decreases. This is due to the increase in the volume of the data read for blocks of size 512B, which becomes the dominant factor in the performance. (vi) Amortized and de-amortized versions of PiBAS, 1C, and NlogN have similar performance. Finally, all our schemes have excellent performance in practice. E.g., SD$_a$[1C] and SD$_a$[NlogN]/L-SD$_d$[NlogN] are up to three orders of magnitude faster than SD$_a$[PiBAS]/SD$_d$[PiBAS] (for $|n_w| = 10^5$ and $|DB| = 2^{23}$ SD$_a$[1C], SD$_a$[NlogN], and SD$_a$[PiBAS] take 16s, 0.8s, and 903s respectively).

Variable Database Size. Figures 8 (c,d) and Figures 9 (b),(c) show the effect of database size on search computation. (Results for the amortized schemes are in Figures 9 (c),(d),(f),(g)). For small blocks it varies search time for variable database sizes between $2^{11} - 2^{26}$ and for result size n_w=1K in HDD and SSD. As the figures show, the search time of all schemes increases as the database size increases, because of the added levels in the data structures. However, all our schemes are significantly faster than SD$_a$[PiBAS] and SD$_d$[PiBAS]. For instance, the amortized schemes SD$_a$[1C], SD$_a$[2C], and SD$_a$[NlogN] are faster than SD$_a$[PiBAS] by $2-136\times$, $5-159\times$, and $4-209\times$ respectively. Also, the de-amortized schemes, L-SD$_d$[1C] and L-SD$_d$[NlogN] are faster than SD$_d$[PiBAS] by $2-58\times$ and $3-70\times$ respectively.

Effect of the value of s on SD$_a$[NlogN] and L-SD$_d$[NlogN]. The previous experiments showed that our NlogN-based schemes have the best performance at a cost of more (i.e. $\log N$ times) storage. However, as discussed in Section 3.2, these schemes can be used more "cleverly" to reduce the storage overhead. In Figure 8 (a,b), we measured the effect of keeping s intermediate levels instead $\log N$ levels for the de-amortized schemes (e.g., L-SD$_d$[3N] refers to keeping 3 levels). Similar experiments for amortized schemes are presented in Figure 11. If the level that a keyword-list should be stored does not exist, we split the list into multiple equal-sized

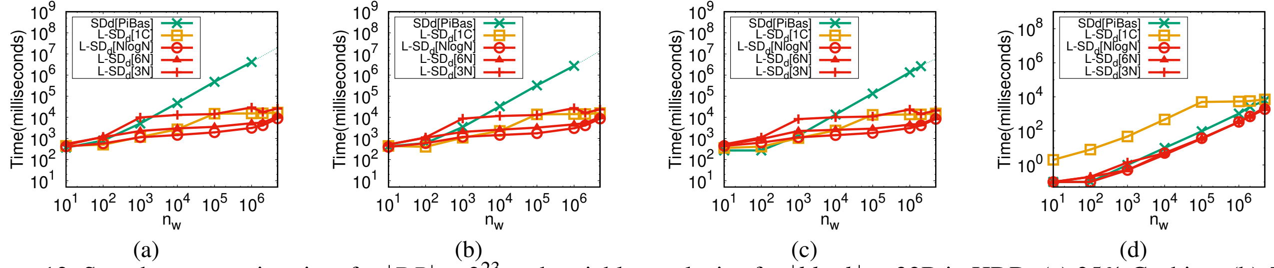

Figure 12: Search computation time for $|DB| = 2^{23}$ and variable result size for $|block| = 32$B in HDD, (a) 25% Caching, (b) 50% Caching, (c) 75% Caching, (d) 100% Caching.

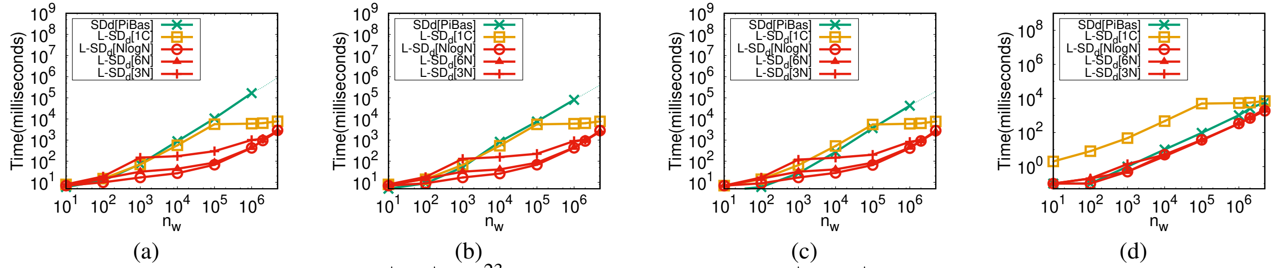

Figure 13: Search computation time for $|DB| = 2^{23}$ and variable result size for $|block| = 32$B in SSD, (a) 25% Caching, (b) 50% Caching, (c) 75% Caching, (d) 100% Caching.

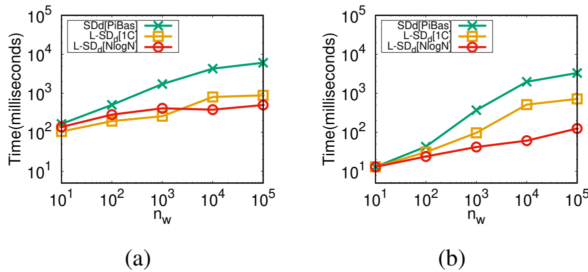

Figure 14: Search computation time for $|DB| = 2^{20}$ for each user and variable result size for $|block| = 32$B with caching enabled in (a) HDD where 200 users exist in the system (b) SSD where 75 users exist in the system

chunks (last chunk is padded, if necessary) and store them in the next closest existing level. The experiment shows that even when keeping fewer levels than $\log N$, SD$_a$[NlogN]/L-SD$_d$[NlogN] outperform PiBAS and 1C based schemes in terms of search time, due to its practical locality and page efficiency, while also reducing storage to $3s \times N$. E.g., when storing every 8th level (SD$_a$[3N]/L-SD$_d$[3N]), the achieved scheme is up to 30× and 1949× faster than SD$_a$[1C] and SD$_a$[PiBAS] and up to 8× and 833× faster than L-SD$_d$[1C] and L-SD$_d$[PiBAS] for big result sizes.

Cache Experiment. Figures 12 and 13 show the effect of cache on the search time. They represent the search computation time over a dataset with size 2^{23} and $|block| = 32B$ for variable result sizes and variable cache sizes on HDD (Figure 12) and SSD (Figure 13). As explained above, for these experiments we fix the cache size in memory according to the SD$_d$[PiBAS] scheme which has the smallest storage size in the de-amortized schemes (e.g., 25% of the encrypted dataset of SD$_d$[PiBAS]), and we use the same cache size as for other schemes for fairness. The figures show that: i) as the cache size increases, all schemes' perform better and their search time reduces. However, SD$_d$[PiBAS] benefits more from the cache than others (i.e., its search time improves up to 8× while L-SD$_d$[1C] and L-SD$_d$[NlogN] only improve up to 2.4× and 2×) due to the smaller needed storage. ii) all our schemes still outperform SD$_d$[PiBAS] for big enough result sizes (>1K). Even when 75% of data is cached in SD$_d$[PiBAS], L-SD$_d$[1C] and L-SD$_d$[NlogN] are up to 191× and 641× faster. iii) When all data is cached (assuming it fits entirely in memory), L-SD$_d$[NlogN] outperforms other schemes (as expected) because it has the best locality and needs minimal cryptographic operations. On the other hand, L-SD$_d$[1C] becomes worse than SD$_d$[PiBAS] in big result sizes because in these queries the dominant overhead comes from crypto and L-SD$_d$[1C] needs to execute Sim3× more decryptions than the other schemes due to padding.

In a second experiment (Fig. 14) we emulate a scenario with multiple users (200 for HDD and 75 for SSD) each with her own independent dataset (of size 2^{20}). With cache enabled, we executed random queries among users. As the figure shows, due to the size of datasets being larger than the memory (encrypted indexes for HDD and SSD were 1.8TB and 675GB), we see a similar trend as Figures 8 (a, b).

Search Over Real Datasets. We evaluated search times on two attributes of the crime dataset [1], with (a) 34, and (b) 170 distinct keywords. We measured the search time for differ-

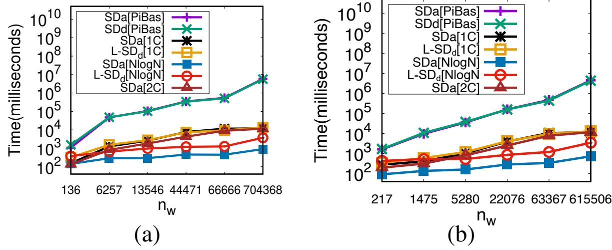

Figure 15: Crime Dataset—Search computation time vs variable result size for an attributed with (a) $|W| = 34$, (b) $|W| = 170$.

ent keywords associated with increasing numbers of results. Our experiments show that our schemes clearly outperform both SD_a[PiBAS] and L-SD_d[PiBAS], and we reach similar conclusions as previously (Figure 15).

Search Over WAN We also simulated the end-to-end search time when client and server are located on two different machines. Our testbed was two AWS machines in Ireland and Frankfurt (with 24.7ms delay and 2.5Gbps bandwidth). Our experiments show a similar trend as the single machine case due to the constant roundtrip number and high network bandwidth, so our schemes outperform SD_a[PiBAS] and SD_d[PiBAS] (see Fig. 10 (d)).

4.2 Update Performance

Next, we report the update performance of our schemes via two sets of experiments: (i) update cost of the amortized scheme, (ii) update cost of the de-amortized schemes.

Update Cost of Amortized schemes. In the first experiment, we measured the update time for 1K consecutive updates. Each time, the client fetches and merges some previously built levels and then uploads them to the server. Clearly, the update cost depends on the number of previously inserted indexes and it increases when more indexes need to be merged. We repeated the same experiment for SD_a[NlogN] and saw the same behaviour (see Figure 16). It is clear that SD_a[PiBAS] has the best update cost for small merges (due to storing fewer entries) and the worst update cost for big merges (as random I/Os increases). The minimum and maximum observed time for SD_a[PiBAS] is 2ms and 12905ms, while for SD_a[NlogN] they are 58ms and 6421ms.

Update Cost of De-Amortized schemes. To measure the update performance of our de-amortized schemes, first we measured the update computation time for variable database sizes in memory. According to our experiment (Figure 17 (a)), L-SD_d[1C] outperforms other schemes for database sizes above $100K$ which is compatible with its better asymptotics (e.g., L-SD_d[1C] is up to $2.5\times$ faster than SD_d[PiBAS] for database size of 5M). Furthermore, L-SD_d[NlogN] has the worst performance in the memory setting due to the large

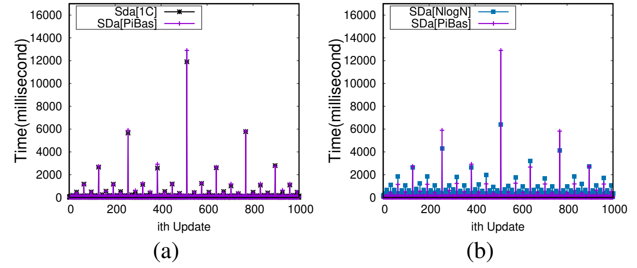

Figure 16: Update computation time of amortized schemes for 1K updates starting from an empty dataset (a) SD_a[1C] vs SD_a[PiBAS] (b) SD_a[NlogN] vs SD_a[PiBAS]

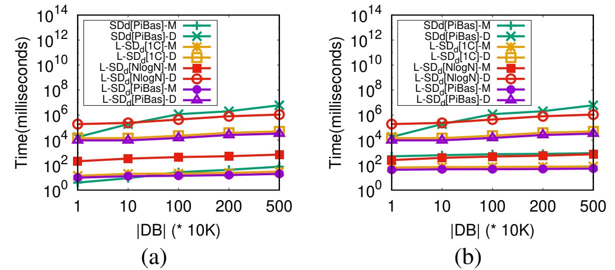

Figure 17: Update computation time for variable database sizes (a) over single machine (b) over WAN machines with 24.7ms network delay and 2.5Gbps bandwidth.

number of layers it needs to create for each level of the de-amortized framework. We also provided L-SD_d[PiBAS] update time to show our framework is applicable to SD_d[PiBAS] and can reduce its cost from $O(\log^3 N)$ to $O(\log^2 N)$. Finally, we re-executed this using HDD storage. We observe that L-SD_d[1C] is still the most efficient scheme in all database sizes. On the other hand, L-SD_d[NlogN] becomes better than SD_d[PiBAS] in bigger database sizes due to its better locality (e.g., L-SD_d[1C] and L-SD_d[NlogN] are $123\times$ and $5\times$ faster than SD_d[PiBAS] for size 5M).

Update Over WAN. We measured end-to-end update times when client and server are located on different AWS machines, as above (Figure 17 (b)). The performance of memory-based SD_d[PiBAS] worsens due to the round trips for OMAP access and the amount of data that must be transferred over the network (L-SD_d[1C] is $8-11.3\times$ and L-SD_d[NlogN] is $1.2-2\times$ faster than SD_d[PiBAS]). That said, the performance of disk-based schemes is similar to the single-machine case as the disk overhead is the dominant cost and the bandwidth is high enough to "cover" for the network overhead.

5 Conclusion

In this work, we proposed the first I/O efficient DSE schemes with forward/backward privacy by re-visiting prior "static-to-dynamic" compilers. First, we came up with a new *oblivious merge* framework, that enabled us to place entries for the same keyword close to each other (preserving locality). Moreover, we optimized performance by replacing oblivious data

structures for more lightweight and easier-to-implement oblivious sorting algorithms and linear scans. We implemented both amortized and de-amortized transformations with I/O-efficient static schemes, such as 1C, 2C, NlogN, and sN (for s = 3 and 6), and compared their search and update times with prior works, overall showcasing our schemes' superior performance in various settings and configurations.

Acknowledgments

We would like to thank the anonymous reviewers for their constructive feedback. This work was partially supported by Hong Kong RGC under grant 16200721.

References

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

[2] Pixek app. https://pixek.io/.

[3] OpenSSL: The open source toolkit for SSL/TLS. https://www.openssl.org/, 2003.

[4] AJTAI, M., KOML´OS, J., AND SZEMER´EDI, E. An o(n log n) sorting network. In *TOC, 1983*.

[5] ASHAROV, G., CHAN, T.-H. H., NAYAK, K., PASS, R., REN, L., AND SHI, E. Bucket oblivious sort: An extremely simple oblivious sort. In *arXiv, 2020*.

[6] ASHAROV, G., NAORY, M., SEGEVZ, G., AND SHAHAFZ, I. Searchable symmetric encryption: Optimal locality in linear space via two-dimensional balanced allocations. In *STOC, 2016*.

[7] ASHAROV, G., SEGEV, G., AND SHAHAF, I. Tight tradeoffs in searchable symmetric encryption. In *CRYPTO,2018*.

[8] BATCHER, K. E. Sorting networks and their applications. In *SJCC, 1968*.

[9] BLACKSTONE, L., KAMARA, S., AND MOATAZ, T. Revisiting leakage abuse attacks. In *NDSS, 2022*.

[10] BOSSUAT, A., BOST, R., FOUQUE, P., MINAUD, B., AND REICHLE, M. Sse and ssd: Page-efficient searchable symmetric encryption. In *CRYPTO 2021*.

[11] BOST, R. Sofos: Forward secure searchable encryption. In *CCS, 2016*.

[12] BOST, R., MINAUD, B., AND OHRIMENKO, O. Forward and backward private searchable encryption from constrained cryptographic primitives. In *CCS, 2017*.

[13] CASH, D., GRUBBS, P., PERRY, J., AND RISTENPART, T. Leakage-abuse attacks against searchable encryption. In *CCS* (2015).

[14] CASH, D., JAEGER, J., JARECKI, S., JUTLA, C., KRAWCZYK, H., ROSU, M., AND STEINER, M. Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation. In *NDSS, 2014*.

[15] CASH, D., JARECKI, S., JUTLA, C., KRAWCZYK, H., ROŞU, M.-C., AND STEINER, M. Highly-scalable searchable symmetric encryption with support for boolean queries. In *CRYPTO, 2013*.

[16] CASH, D., NG, R., AND RIVKIN, A. Improved structured encryption for SQL databases via hybrid indexing. In *International Conference on Applied Cryptography and Network Security 2021*.

[17] CASH, D., AND TESSARO, S. The Locality of Searchable Symmetric Encryption. In *EUROCRYPT, 2014*.

[18] CHAMANI, J. G., DEMERTZIS, I., PAPADOPOULOS, D., PAPAMANTHOU, C., AND JALILI, R. Graphos: Towards oblivious graph processing. *PVLDB* (2023).

[19] CHAMANI, J. G., WANG, Y., PAPADOPOULOS, D., ZHANG, M., AND JALILI, R. Multi-user dynamic searchable symmetric encryption with corrupted participants. *IEEE Transactions on Dependable and Secure Computing* (2021).

[20] CHANG, Y. C., AND MITZENMACHER, M. Privacy preserving keyword searches on remote encrypted data. In *ACNS, 2005*.

[21] CHASE, M., AND KAMARA, S. Structured encryption and controlled disclosure. In *ASIACRYPT, 2010*.

[22] CHEN, TIANYANG, E. A. Bestie: Very practical searchable encryption with forward and backward security. In *Computer Security–ESORICS 2021: 26th European Symposium on Research in Computer Security, Darmstadt, Germany, October 4–8, 2021, Proceedings, Part II 26. Springer International Publishing, 2021*.

[23] CURTMOLA, R., GARAY, J., KAMARA, S., AND OSTROVSKY, R. Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions. In *CCS, 2006*.

[24] DAUTERMAN, E., FANG, V., DEMERTZIS, I., CROOKS, N., AND POPA, R. A. Snoopy: Surpassing the scalability bottleneck of oblivious storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (2021).

[25] DAUTRICH JR, J. L., AND RAVISHANKAR, C. V. Compromising Privacy in Precise Query Protocols. In *Proceedings of the 16th International Conference on Extending Database Technology* (2013), ACM, pp. 155–166.

[26] DEMERTZIS, I., GHAREH CHAMANI, J., PAPADOPOULOS, D., AND PAPAMANTHOU, C. Dynamic searchable encryption with small client storage. In *NDSS 2020*.

[27] DEMERTZIS, I., PAPADOPOULOS, D., AND PAPAMANTHOU, C. Searchable encryption with optimal locality: Achieving sublogarithmic read efficiency. In *CRYPTO, 2018*.

[28] DEMERTZIS, I., PAPADOPOULOS, D., PAPAMANTHOU, C., AND SHINTRE, S. Seal: Attack mitigation for encrypted databases via adjustable leakage. In *Usenix, 2020*.

[29] DEMERTZIS, I., PAPADOPOULOS, S., PAPAPETROU, O., DELIGIANNAKIS, A., AND GAROFALAKIS, M. Practical private range search revisited. In *SIGMOD* (2016).

[30] DEMERTZIS, I., PAPADOPOULOS, S., PAPAPETROU, O., DELIGIANNAKIS, A., GAROFALAKIS, M., AND PAPAMANTHOU, C. Practical private range search in depth. *TODS, 2018*.

[31] DEMERTZIS, I., AND PAPAMANTHOU, C. Fast searchable encryption with tunable locality. In *SIGMOD, 2017*.

[32] DEMERTZIS, I., TALAPATRA, R., AND PAPAMANTHOU, C. Efficient searchable encryption through compression. In *PVLDB, 2018*.

[33] ETEMAD, M., KÜPÇÜ, A., PAPAMANTHOU, C., AND EVANS, D. Efficient dynamic searchable encryption with forward privacy. *PETS, 2018*.

[34] GHAREH CHAMANI, J., PAPADOPOULOS, D., KARBASFORUSHAN, M., AND DEMERTZIS, I. Dynamic searchable encryption with optimal search in the presence of deletions. In *31st USENIX Security Symposium, USENIX Security 2022* (2022), pp. 2425–2442.

[35] GHAREH CHAMANI, J., PAPADOPOULOS, D., PAPAMANTHOU, C., AND JALILI, R. New constructions for forward and backward private symmetric searchable encryption. In *CCS, 2018*.

[36] GOLDREICH, O., AND OSTROVSKY., R. Software protection and simulation on oblivious rams. In *Journal of the ACM, 43(3), 1996*.

[37] GOODRICH, M. T. Data-oblivious external-memory algorithms for the compaction, selection, and sorting of outsourced data. In *SPAA, 2011*.

[38] GOODRICH, M. T. Zig-zag sort: A simple deterministic data-oblivious sorting algorithm running in o(n log n time. In *TOC, 2014*.

[39] GOODRICH, M. T., AND MITZENMACHER., M. Mapreduce parallel cuckoo hashing and oblivious ram simulations. In *CoRR, 2010*.

[40] HE, KUN, E. A. Secure dynamic searchable symmetric encryption with constant client storage cost. In *IEEE Transactions on Information Forensics and Security 16 (2020): 1538-1549*.

[41] ISLAM, M. S., KUZU, M., AND KANTARCIOGLU, M. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS* (2012).

[42] KAMARA, S., AND MOATAZ., T. Encrypted multi-maps with computationally-secure leakage. In *eprint, 2019*.

[43] KAMARA, S., MOATAZ, T., PARK, A., AND QIN, L. A decentralized and encrypted national gun registry. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021* (2021), IEEE, pp. 1520–1537.

[44] KAMARA, S., AND MOATAZ , T. Sql on structurally-encrypted databases. In *ASIACRYPT, 2018*.

[45] KAMARA, S., AND PAPAMANTHOU, C. Parallel and dynamic searchable symmetric encryption. In *FC, 2013*.

[46] KATZ, J., AND LINDELL, Y. *Introduction to modern cryptography*. CRC press, 2020.

[47] KIM, K. S., KIM, M., LEE, D., PARK, J. H., AND KIM, W.-H. Forward secure dynamic searchable symmetric encryption with efficient updates. In *CCS, 2017*.

[48] KORNAROPOULOS, E. M., MOYER, N., PAPAMANTHOU, C., AND PSOMAS, A. Leakage inversion: Towards quantifying privacy in searchable encryption. In *CCS* (2022).

[49] KORNAROPOULOS, E. M., PAPAMANTHOU, C., AND TAMASSIA, R. Data recovery on encrypted databases with k-nearest neighbor query leakage. In *Data Recovery on Encrypted Databases with k-Nearest Neighbor Query Leakage* (2019), SP.

[50] KORNAROPOULOS, E. M., PAPAMANTHOU, C., AND TAMASSIA, R. The state of the uniform: Attacks on encrypted databases beyond the uniform query distribution. In *SP* (2020).

[51] KORNAROPOULOS, E. M., PAPAMANTHOU, C., AND TAMASSIA, R. Response-hiding encrypted ranges: Revisiting security via parametrized leakage-abuse attacks. In *SP* (2021).

[52] KUROSAWA, K., AND OHTAKI, Y. UC-Secure Searchable Symmetric Encryption. In *FC, 2012*.

[53] LAI, R. W., AND CHOW., S. S. Forward-secure searchable encryption on labeled bipartite graphs. In *International Conference on Applied Cryptography and Network Security. Cham: Springer International Publishing, 2017*.

[54] MIDORIKAWA, T., TACHIKAWA, A., AND KANAOKA, A. Helping johnny to search: Encrypted search on webmail system. In *13th Asia Joint Conference on Information Security, AsiaJCIS 2018, Guilin, China, August 8-9, 2018* (2018), IEEE Computer Society, pp. 47–53.

[55] MINAUD, B., AND REICHLE, M. Dynamic local searchable symmetric encryption. In *CRYPTO 2022*.

[56] MINAUD, B., AND REICHLE, M. Hermes: I/o-efficient forward-secure searchable symmetric encryption. Cryptology ePrint Archive, Paper 2023/166, 2023. https://eprint.iacr.org/2023/166.

[57] MISHRA, P., PODDAR, R., CHEN, J., CHIESA, A., AND POPA, R. A. Oblix: An efficient oblivious search index. In *Oblix: An Efficient Oblivious Search Index* (2018), SP.

[58] MITCHELL, J. C., AND ZIMMERMAN, J. Data-oblivious data structures. In *STACS, 2014*.

[59] NGAI, N., DEMERTZIS, I., CHAMANI, J. G., AND PAPADOPOULOS, D. Distributed & scalable oblivious sorting and shuffling. In *2024 IEEE Symposium on Security and Privacy (SP)* (2024).

[60] OSTROVSKY., R. Efficient computation on oblivious rams. In *In STOC, pages 514–523, 1990.*

[61] RIZOMILIOTIS, P., AND GRITZALIS., S. Simple forward and backward private searchable symmetric encryption schemes with constant number of roundtrips. In *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop. 2019.*

[62] SONG, D. X., WAGNER, D., AND PERRIG, A. Practical Techniques for Searches on Encrypted Data. In *SP, 2000*.

[63] SONG, XIANGFU, E. A. Forward private searchable symmetric encryption with optimized i/o efficiency. In *IEEE Transactions on Dependable and Secure Computing 17.5 (2018): 912-927*.

[64] STEFANOV, E., PAPAMANTHOU, C., AND SHI, E. Practical dynamic searchable encryption with small leakage. In *NDSS, 2014*.

[65] STEFANOV, E., VAN DIJK, M., SHI, E., FLETCHER, C., REN, L., YU, X., AND DEVADAS, S. Path oram: an extremely simple oblivious ram protocol. In *CCS 2013*.

[66] SUN, SHI-FENG, E. A. Practical backward-secure searchable encryption from symmetric puncturable encryption. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. 2018.*

[67] WANG, J., AND CHOW., S. S. Forward and backward-secure range-searchable symmetric encryption. In *Cryptology ePrint Archive (2019).*

[68] WANG, X. S., NAYAK, K., LIU, C., CHAN, T., SHI, E., STEFANOV, E., AND HUANG, Y. Oblivious data structures. In *CCS, 2014*.

[69] WANG, Y., AND PAPADOPOULOS, D. Multi-user collusion-resistant searchable encryption with optimal search time. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security* (2021).

[70] WANG, Y., AND PAPADOPOULOS, D. Multi-user collusion-resistant searchable encryption for cloud storage. *IEEE Transactions on Cloud Computing* (2023).

[71] WANSHAN, X., ZHANG, J., AND YUAN., Y. Desse: A dynamic efficient forward searchable encryption scheme. In *IEEE Access 8 (2020): 144480-144488.*

[72] XU, PENG, E. A. Rose: Robust searchable encryption with forward and backward security. In *IEEE transactions on information forensics and security 17 (2022): 1115-1130.*

[73] ZHANG, Y., KATZ, J., AND PAPAMANTHOU, C. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *USENIX 2016*.