Searchable Encryption with Optimal Locality: Achieving Sublogarithmic Read Efficiency

Ioannis Demertzis[1(✉)], Dimitrios Papadopoulos[2], and Charalampos Papamanthou[1]

[1] University of Maryland, College Park, USA
{yannis,cpap}@umd.edu
[2] Hong Kong University of Science and Technology, Kowloon, Hong Kong
dipapado@cse.ust.hk

Abstract. We propose the first linear-space searchable encryption scheme with constant locality and *sublogarithmic* read efficiency, strictly improving the previously best known read efficiency bound (Asharov et al., STOC 2016) from $\Theta(\log N \log\log N)$ to $O(\log^{\gamma} N)$ where $\gamma = \frac{2}{3} + \delta$ for any fixed $\delta > 0$ and where N is the number of keyword-document pairs. Our scheme employs four different allocation algorithms for storing the keyword lists, depending on the size of the list considered each time. For our construction we develop (i) new probability bounds for the offline two-choice allocation problem; (ii) and a new I/O-efficient oblivious RAM with $\tilde{O}(n^{1/3})$ bandwidth overhead and zero failure probability, both of which can be of independent interest.

1 Introduction

Searchable Encryption (SE), first proposed by Song et al. [30] and then formalized by Curtmola et al. [12], enables a data owner to outsource a private dataset $\mathcal{D}$ to a server, so that the latter can answer keyword queries without learning too much information about the underlying dataset and the posed queries. An alternative to expensive primitives such as oblivious RAM and fully homomorphic encryption, SE schemes are practical at the expense of formally-specified leakage. In typical SE schemes, the data owner prepares a private index which is sent to the server. To perform a query on keyword w, the data owner engages in a protocol with the server such that by the end of the protocol the data owner retrieves the list of document identifiers $\mathcal{D}(w)$ of documents containing w. During this process, the server should learn nothing except for the (number of) retrieved document identifiers—referred to as (size of) *access pattern*—and whether the keyword search query w was repeated in the past or not—referred to as *search pattern*.

To retrieve the document identifiers $\mathcal{D}(w)$ (also referred to as *keyword list* in the rest of the paper), most existing SE schemes require the server access approximately $|\mathcal{D}(w)|$ randomly-assigned memory locations [10,12,14,23,24,30,31]. While this random allocation is essential for security, it creates a big

H. Shacham and A. Boldyreva (Eds.): CRYPTO 2018, LNCS 10991, pp. 371–406, 2018.
https://doi.org/10.1007/978-3-319-96884-1_13

bottleneck when accessing large indexes stored on disk.[1] Therefore the aforementioned schemes cannot scale for big data that do not fit in memory due to poor *locality*—the number of non-contiguous memory locations that must be read to retrieve the result.

Locality and Read Efficiency Trade-offs. One trivial way to design an SE scheme that has optimal locality $L = 1$ is to have the client download the whole encrypted index for every query w. Unfortunately, such an approach requires $O(N)$ bandwidth, where N is the total number of keyword-document pairs. Cash et al. [10] were the first to observe this trade-off: To improve the locality of SE, one should expect to read additional entries per query. The ratio of the total number of entries read over the size of the query result was defined as *read efficiency*. This trade-off was subsequently formalized by Cash and Tessaro [11] who showed it is impossible[2] to construct an SE scheme with linear space, optimal locality and optimal read efficiency.

In response to this impossibility result, several positive results with various trade-offs have appeared. Cash and Tessaro [11] presented a scheme with $\Theta(N \log N)$ space, $O(1)$ read efficiency and $O(\log N)$ locality, which was later improved to $O(1)$ by Asharov et al. [6]. Demertzis and Papamanthou [16] presented a scheme with bounded locality $O(N^{\epsilon})$, $O(1)$ read efficiency and linear space (for constant $\epsilon < 1$). More recently, Asharov et al. [5] studied the locality in Oblivious RAMs, proposing a construction that, for an arbitrary sequence of accesses (and therefore for SE as well), achieves $O(N)$ space, $O(\log^2 N \log^2 \log N)$ read efficiency, and $O(\log N \log^2 \log N)$ locality. While asymptotically worse than [6], this work has better security as it leaks no access pattern. Finally, significant speedups due to locality in SE implementations have been observed by Miers and Mohassel [25] and Demertzis et al. [15,16].

Constant Locality with Linear Space. Practical reasons described above have motivated the study of even more asymptotically-efficient SE schemes, and in particular those with *constant locality* and *linear space*. Asharov et al. [6] presented two such SE schemes based on a two-dimensional generalization of the "balls and bins" problem. In particular, the first scheme (A1) uses two-choice allocation, has very low read efficiency $\Theta(\log \log N \log^2 \log \log N)$ but is based on the assumption that all lists $\mathcal{D}(w)$ have size $\leq N^{1-1/\log \log N}$.[3] Recently, Asharov et al. [7] provided a version of A1 with improved read efficiency and a

[1] Demertzis and Papamanthou [16] recently showed that low-locality SE may improve practical performance for in-memory data too, due to reduced number of server crypto operations.

[2] The result holds for a setting where lists $\mathcal{D}(w)$ are stored at non-overlapping positions.

[3] We tested this assumption for 4 real datasets: One containing crime records in Chicago since 2001 [1], the Enron email dataset [2], the USPS dataset [4] and the TPC-H dataset [3]. The Enron email dataset does not violate the assumption, which is not the case for the other datasets where almost half of the contained attributes violate it. For the crimes dataset, for example, the assumption was violated in 12 out of 21 attributes for 31% of the keywords on average.

better bound $N/\log^3 N$ for the maximum $\mathcal{D}(w)$ size. The second construction of [6] (A2) has $\tilde{\Theta}(\log N)$ read efficiency, uses one-choice allocation and makes no assumptions about the dataset. To our knowledge, A2 is the best SE scheme with $O(1)$ locality and $O(N)$ space known to-date for *general datasets*.

Our Contribution. Motivated by the above positive results and the impossibility result of [11], we ask whether it is possible to build an SE scheme for general datasets with: (i) *linear space*, (ii) *constant locality*, and (iii) *sublogarithmic read efficiency*. We answer this question in the affirmative by designing the first such SE scheme, strictly improving upon the best known scheme A2 [6]. For the rest of the paper we set $\gamma = 2/3 + \delta$ for $\delta > 0$ arbitrarily small. We show that the read efficiency of our scheme is $O(\log^\gamma N)$ as opposed to that of A2 which is $\Theta(\log N \log\log N)$. Parameter δ above affects the constants in the asymptotic notation which grow with $O(1/\delta)$.

1.1 Summary of Our Techniques

Our techniques (like previous works on low-locality SE) use the notion of an *allocation algorithm*, whose goal is to store the dataset's keyword lists in memory such that each keyword list $\mathcal{D}(w)$ can be efficiently retrieved by accessing memory locations *independent* of the distribution the SE dataset—this is needed for security reasons. Common techniques to achieve this, store keyword lists using a balls-and-bins procedure [6].

Starting Point. We first observe that keyword lists of size less than $N^{1-1/\log^{1-\gamma} N}$, for some $\gamma < 1$, can be allocated using (as a black box) the parameterized version of scheme A1 of Asharov et al. [6]. In particular, we show in Theorem 6 that for $\gamma = 2/3 + \delta$ scheme A1 yields $\Theta(\log^\gamma N)$ read efficiency, as desired. Therefore we only need to focus on allocating the dataset's keyword lists that have size $> N^{1-1/\log^{1-\gamma} N}$.

Our Main Technique: Different Allocation Algorithms for Different Size Ranges. Let $\gamma = 2/3 + \delta$ as defined above. We develop three allocation algorithms for the remaining ranges: Lists with size in $(N^{1-1/\log^{1-\gamma} N}, N/\log^2 N]$, also called *medium*, are allocated using an *offline two-choice allocation* procedure [28] and multiple stashes to handle overflows. Lists with size in $(N/\log^2 N, N/\log^\gamma N]$, also called *large*, are first split into further subranges based on their size and then each subrange is allocated into a separate array using the same algorithm. Finally, for lists with size in $(N/\log^\gamma N, N]$, also called *huge*, there is no special allocation algorithm: We just read the whole dataset. We now provide a summary of our allocation algorithms for medium and large lists.

1.2 Medium Keyword Lists Allocation

Our allocation algorithm for medium keyword lists is using an offline two-choice allocation (OTA),[4] where there are m balls and n bins and for each ball two possible bins are chosen independently and uniformly at random. After *all choices* have been made, one can run a maximum flow algorithm to find the final assignment of balls to bins such that the maximum load is minimized. This strategy yields an almost perfectly balanced allocation (where max-load $\leq \lceil m/n \rceil + 1$) with probability at least $1 - O(1/n)$ [28].

Central Idea: One OTA Per Size and Then Merge. We use one OTA separately for every size s that falls in the range $(N^{1-1/\log^{1-\gamma} N}, N/\log^2 N]$ as follows: Let $\mathbf{A}_s$ be an array of M buckets $\mathbf{A}_s[1], \mathbf{A}_s[2], \ldots, \mathbf{A}_s[M]$, for some appropriately chosen M. One can visualize a bucket $\mathbf{A}_s[i]$ as a vertical structure of unbounded capacity. Let k_s be the number of keyword lists of size s and let $b_s = M/s$ be the number of superbuckets in $\mathbf{A}_s$, where a superbucket is a collection of s consecutive buckets in $\mathbf{A}_s$. We perform an OTA of k_s keyword lists to the b_s superbuckets. From [28], there will be at most $\lceil k_s/b_s \rceil + 1$ lists of size s in each superbucket with probability at least $1 - O(1/b_s)$, meaning the load of each bucket due to lists of size s will be at most $\lceil k_s/b_s \rceil + 1$ with the same probability, given there are s buckets in a superbucket.

Our final allocation merges arrays $\mathbf{A}_s$ for all sizes s corresponding to medium keyword lists into a new array $\mathbf{A}$ of M buckets—see Fig. 5. To bound the final load of each bucket $\mathbf{A}[i]$ in the merged array $\mathbf{A}$ one can compute $\sum_s (\lceil k_s/b_s \rceil + 1)$ which is $O(N/M + \log^\gamma N)$—see Lemma 4. If we set $M = N/\log^\gamma N$, our allocation occupies linear space and each bucket $\mathbf{A}[i]$ has load $O(\log^\gamma N)$—thus to read one list, one reads the two superbuckets initially picked by the OTA yielding read efficiency $O(\log^\gamma N)$.

Handling Bucket Overflows with Additional Stashes. Our analysis above assumes the maximum load of each bucket is at most $\lceil k_s/b_s \rceil + 1$. However, there is a noticeable probability $O(1/b_s)$ of overflowing beyond this bound—this will cause our allocation to fail, leaking information about the dataset. To deal with this problem, for each size s, we place the lists of size s that overflow in a stash $\mathbf{B}_s$ (at the server) that can store up to $O(\log^2 N)$ such overflowing lists. In particular, we prove that when the OTA described previously is performed for medium lists, at most $O(\log^2 N)$ lists of size s overflow with non-negligible probability and thus our stashes $\mathbf{B}_s$ suffice, see Lemma 5. We also stress that we need the condition $s \leq N/\log^2 N$ to keep the space of the stashes linear—see Theorem 7, justifying the pick of $N/\log^2 N$ as endpoint of the range where we apply OTA. Finally, the existence of stashes $\mathbf{B}_s$ differentiates our allocation from those of [6], allowing us to avoid their impossibility result (see discussion in Sect. 7).

[4] Deriving the results of this paper using the, more lightweight, online version of the problem is an interesting open problem. Section 7 elaborates on the difficulties that arise in that case.

New Probability Bounds for OTA. Our proof for the $O(\log^2 N)$ stash size extends the analysis of [28] non-trivially—we prove two new results in Sect. 3: First, in Theorem 1 we show that in an OTA, the probability $> \tau$ bins overflow decreases with $(1/\tau)^\tau$. For this proof we show the 0/1 random variables indicating bin overflow are *negatively associated* [17]. Second, in Theorem 2 we show the probability an OTA of m balls to n bins yields a maximum load of $> \lceil m/n \rceil + \tau$ is $\leq O(1/n)^\tau + exp(-n)$.

Accessing Stashes Obliviously. Because keyword lists of size s *might* now live in the stash $\mathbf{B}_s$, retrieving a keyword list $\mathcal{D}(w)$ is a two-step process: First, access the superbuckets that were initially assigned by the OTA and then access a position x in the stash. In case $\mathcal{D}(w)$ is not in the stash (because it was not an overflowing list), x should be still assigned a stash position chosen from the unoccupied ones, if such a position exists. If not, there will be a collision, in which case the adversary can deduce information about the dataset distribution, e.g., that the dataset contains at least $\log^2 N$ lists of size $|\mathcal{D}(w)|$. To avoid such leakage, the stash must be accessed obliviously.

New ORAM with $o(\sqrt{n})$ Bandwidth, $O(1)$ Locality & Zero Failure Probability. Since the stash has only $\log^2 N$ entries of size $|\mathcal{D}(w)|$ each, one can access it obliviously by reading it all. But this increases read efficiency to $\log^2 N$, which is no longer sublogarithmic. Thus, we need an ORAM with (i) $O(1)$ locality, (ii) $o(\sqrt{n})$ bandwidth and (iii) zero failure probability since it will be applied on only $\log^2 N$ indices. In Sect. 4, we devise a new ORAM satisfying the above (with $O(n^{1/3} \log^2 n)$ bandwidth) based on one recursive application of Goldreich's and Ostrovsky's square-root ORAM [18]. This protocol can be of independent interest. To finally ensure our new ORAM has $O(1)$ locality, we use I/O-efficient oblivious sorting by Goodrich and Mitzenmacher [20].

1.3 Large Keyword Lists Allocation

We develop an Algorithm $\mathsf{AllocateLarge}(\mathsf{min}, \mathsf{max})$ that can allocate lists with sizes in a general range $(\mathsf{min}, \mathsf{max}]$. We will be applying this algorithm for lists in the range $(N/\log^2 N, N/\log^\gamma N]$. The algorithm works as follows. Let $\mathbf{A}$ be an array that has $2N$ entries, organized in N/max buckets of capacity $2\mathsf{max}$ each. To store a list of size $s \in (\mathsf{min}, \mathsf{max}]$, a bucket with available size at least s is chosen. To retrieve a list, the entire bucket where the list is stored is accessed using our ORAM construction from Sect. 4—note that ORAM is relatively cheap for this range, since N/max is small.

In this way we always pay the cost of accessing lists of size max, even for smaller list sizes $s > \mathsf{min}$. The read efficiency of this approach is clearly at least $\mathsf{max}/\mathsf{min}$, which for the specified range above is $\log^2 N/\log^\gamma N = \omega(\log N)$ for $\gamma < 1$. Still, this is not enough for our target, which is sublogarithmic read efficiency. Therefore, we need to further split this range into multiple subranges and apply the algorithm for each subrange independently. The number of subranges depends on the target read efficiency, i.e., it depends on γ (but not on N). For example, for $\gamma < 1$ it suffices to have 3 subranges, whereas setting $\gamma = 0.75$ would

require splitting $(N/\log^2 N, N/\log^\gamma N]$ into a fixed number of 11 subranges. In general, as $\delta > 0$ decreases and $\gamma = 2/3 + \delta$ gets closer to 2/3 the number of subranges will increase. We note that using an ORAM of better worst-case bandwidth (e.g., $O(\log^{1/5} \log^2 N)$ instead of $O(\log^{1/3} \log^2 N)$) would reduce the necessary number of subranges (see discussion in Sect. 7).

2 Notation and Definitions

We use the notation $\langle C', S' \rangle \leftrightarrow \Pi \langle C, S \rangle$ to indicate that protocol Π is executed between a client with input C and a server with input S. After the execution of the protocol the client receives C' and the server receives S'. Server operations are in light gray background. All other operations are performed by the client. The client typically interacts with the server via an `Encrypt-And-Write` *data* operation, with which the client encrypts *data* locally with a CPA-secure encryption scheme and writes the encrypted data *data* remotely to server and via a `Read-And-Decrypt` *data* operation, with which the client reads encrypted data *data* from server and decrypts them locally.

In the following, $\mathcal{D}$ will denote the searchable encryption dataset (SE dataset) which is a set of keywords lists $\mathcal{D}(w_i)$. Each keyword list $\mathcal{D}(w_i)$ is a set of keyword-document pairs (w_i, id), called *elements*, where id is the document identifier containing keyword w_i. We denote with N the size of our dataset, i.e., $N = \sum_{w \in \mathbf{W}} |\mathcal{D}(w)|$, where $\mathbf{W}$ is the set of unique keywords of our dataset $\mathcal{D}$. Without loss of generality, we will assume that all keyword lists $\mathcal{D}(w_i)$ have size $|\mathcal{D}(w_i)|$ that is a power of two. This can always be enforced by padding with dummy elements, and will only increase the space at most by a factor of 2. Finally, a function $f(\kappa)$ is *negligible*, denoted $\mathsf{neg}(\kappa)$, if for sufficiently large κ it is less than $1/p(\kappa)$, for all polynomials $p(\kappa)$.

2.1 Searchable Encryption

Our new SE scheme uses a modification of the square-root ORAM protocol as a black box, which is a two-round protocol. Therefore to model our SE scheme we use the protocol-based definition (Setup, Search) as proposed by Stefanov et al. [31].

- $\langle st, \mathcal{I} \rangle \leftrightarrow \textsc{Setup}\langle (1^\kappa, \mathcal{D}), 1^\kappa \rangle$: Setup takes as input security parameter κ and SE dataset $\mathcal{D}$ and outputs secret state st (for client), and encrypted index $\mathcal{I}$ (for server).
- $\langle (\mathcal{D}(w), st'), \mathcal{I}' \rangle \leftrightarrow \textsc{Search}\langle (st, w), \mathcal{I} \rangle$: Search is a protocol between client and server, where the client's input is secret state st and keyword w. Server's input is encrypted index $\mathcal{I}$. Client's output is set of document identifiers $\mathcal{D}(w)$ matching w and updated secret state st' and server's output is updated encrypted index $\mathcal{I}'$.

Just like in previous works [6], the goal of our SE protocols is for the client to retrieve the document identifiers (i.e., the list $\mathcal{D}(w)$) for a specific keyword w. The document themselves can be downloaded from the server in a second

$bit \leftarrow \mathbf{Real}^{\mathsf{SE}}(\kappa)$:

1: $D_0 \leftarrow \mathsf{Adv}(1^\kappa)$; $\langle st_0, \mathcal{I}_0\rangle \leftrightarrow \textsc{Setup}\langle(1^\kappa, \mathcal{D}_0), 1^\kappa\rangle$;
2: **for** $k = 1$ to q **do**
3: $\quad w_k \leftarrow \mathsf{Adv}(1^k, \mathcal{I}_0, M_1, \ldots, M_{k-1})$;
4: $\quad \langle(\mathcal{D}(w_k), st_k), \mathcal{I}_k\rangle \leftrightarrow \textsc{Search}\langle(st_{k-1}, w_k), \mathcal{I}_{k-1}\rangle$;
5: $\quad$ Let M_k be the messages from client to server in the SEARCH protocol above;
6: $bit \leftarrow \mathsf{Adv}(1^k, \mathcal{I}_0, M_1, M_2, \ldots, M_q)$;
7: **return** bit;

$bit \leftarrow \mathbf{Ideal}^{\mathsf{SE}}_{\mathcal{L}_1,\mathcal{L}_2}(\kappa)$:

1: $D_0 \leftarrow \mathsf{Adv}(1^\kappa)$; $(st_{\mathcal{S}}, \mathcal{I}_0) \leftarrow \textsc{SimSetup}(1^\kappa, \mathcal{L}_1(\mathcal{D}_0))$;
2: **for** $k = 1$ to q **do**
3: $\quad w_k \leftarrow \mathsf{Adv}(1^k, \mathcal{I}_0, M_1, \ldots, M_{k-1})$;
4: $\quad (st_{\mathcal{S}}, M_k, \mathcal{I}_k) \leftarrow \textsc{SimSearch}(st_{\mathcal{S}}, \mathcal{L}_2(w_k), \mathcal{I}_{k-1})$;
5: $bit \leftarrow \mathsf{Adv}(1^k, \mathcal{I}_0, M_1, M_2, \ldots, M_q)$;
6: **return** bit;

Fig. 1. Real and ideal experiments for the SE scheme.

round, by just providing $\mathcal{D}(w)$. This is orthogonal to our protocols and we do not consider/model it here explicitly. We also note that we focus only on static SE. However, by using generic techniques, e.g., [14], we can extend our schemes to the dynamic setting. The correctness definition of SE is given in the extended version [13]. We now provide the security definition.

Definition 1 (Security of SE). *An SE scheme* (SETUP, SEARCH) *is secure in the semi-honest model if for any PPT adversary* Adv*, there exists a stateful PPT simulator* (SIMSETUP, SIMSEARCH) *such that*

$$|\Pr[\mathbf{Real}^{\mathsf{SE}}(\kappa) = 1] - \Pr[\mathbf{Ideal}^{\mathsf{SE}}_{\mathcal{L}_1,\mathcal{L}_2}(\kappa) = 1]| \leq \mathsf{neg}(\kappa)\,,$$

where experiments $\mathbf{Real}^{\mathsf{SE}}(\kappa)$ *and* $\mathbf{Ideal}^{\mathsf{SE}}_{\mathcal{L}_1,\mathcal{L}_2}(\kappa)$ *are defined in Fig. 1 and where the randomness is taken over the random bits used by the algorithms of the SE scheme, the algorithms of the simulator and* Adv*.*

Leakage Functions $\mathcal{L}_1$ and $\mathcal{L}_2$. As in prior work [6], $\mathcal{L}_1$ and $\mathcal{L}_2$ are leakage functions such that $\mathcal{L}_1(\mathcal{D}_0) = |\mathcal{D}_0| = N$ and $\mathcal{L}_2(w_i)$ leaks the access pattern *size* $|\mathcal{D}(w_i)|$ and the *search pattern* of w_i. Formally for a keyword w_i searched at time i, $\mathcal{L}_2(w_i)$ is

$$\mathcal{L}_2(w_i) = \begin{cases} (|\mathcal{D}(w_i)|, j) & \text{if } w_i \text{ was searched at time } j < i \\ (|\mathcal{D}(w_i)|, \perp) & \text{if } w_i \text{ was never searched before} \end{cases}. \tag{1}$$

2.2 Oblivious RAM

Oblivious RAM (ORAM), introduced by Goldreich and Ostrovsky [18] is a compiler that encodes the memory such that accesses on the compiled memory do

(chosen, alternative) ← OfflineTwoChoiceAllocation(m, n)

1: Let $\{1, \ldots, m\}$ be a set of balls and $\{1, \ldots, n\}$ be a set of bins;
2: Initialize A and B to be empty arrays of m entries;
3: **for** $i = 1, \ldots, m$ **do**
4: Pick two bins a_i and b_i from $\{1, \ldots, n\}$ independently and uniformly at random;
5: $\mathsf{A}[i] = a_i$; $\mathsf{B}[i] = b_i$;
6: (chosen, alternative) ← MaxFlowSchedule$(m, n, \mathsf{A}, \mathsf{B})$;
7: **return** (chosen, alternative);

Fig. 2. Offline two-choice allocation of m balls to n bins.

not reveal access patterns on the original memory. Formal correctness and security definitions of ORAM are given in the Appendix. We give the definition for a read-only ORAM as this is needed in our scheme—the definition naturally extends for writes as well:

- $\langle \sigma, \mathsf{EM} \rangle \leftrightarrow$ OramInitialize$\langle (1^\kappa, \mathsf{M}), 1^\kappa \rangle$: OramInitialize takes as input security parameter κ and memory array M of n values $(1, v_1), \ldots, (n, v_n)$ of λ bits each and outputs secret state σ (for client), and encrypted memory EM (for server).
- $\langle (v_i, \sigma'), \mathsf{EM}' \rangle \leftrightarrow$ OramAccess$\langle (\sigma, i), \mathsf{EM} \rangle$: OramAccess is a protocol between client and server, where the client's input is secret state σ and an index i. Server's input is encrypted memory EM. Client's output is value v_i assigned to i and updated secret state σ'. Server's output is updated encrypted memory EM′.

3 New Bounds for Offline Two-Choice Allocation

As mentioned in the introduction, our medium-list allocation uses a variation of the classic balls-in-bins problem, known as *offline two-choice allocation*—see Fig. 2. Assume m balls and n bins. In the selection phase, for the i-th ball, two bins a_i and b_i are chosen independently and uniformly at random. After selection, in a post-processing phase, the i-th ball is mapped to either bin a_i or b_i such that the maximum load is minimized. This assignment is achieved by a maximum flow algorithm [28] (for completeness, we provide this algorithm in Fig. 13 in the Appendix). The bin that ball i is finally mapped to is stored in an array chosen$[i]$ whereas the other bin that was chosen for ball i is stored in an array alternative$[i]$. Let L^*_{max} denote the maximum load across all bins after this allocation process completes. Sanders et al. [28] proved the following.

Lemma 1 (Sanders et al. [28]). *Algorithm* OfflineTwoChoiceAllocation *in Fig. 2 outputs an allocation* chosen *of m balls to n bins such that* $L^*_{\mathsf{max}} > \lceil \frac{m}{n} \rceil + 1$

with probability at most $O(1/n)$.[5] *Moreover, the allocation can be performed in time* $O(n^3)$.

For our purposes, the bounds derived by Sanders et al. [28] do not suffice. In the following we derive new bounds. In particular:

1. In Sect. 3.1, we derive probability bounds on the *number of overflowing bins*, i.e., the bins that contain more than $\lceil \frac{m}{n} \rceil + 1$ balls after the allocation completes.
2. In Sect. 3.2, we derive probability bounds on the *overflow size*, i.e., the number of balls beyond $\lceil \frac{m}{n} \rceil + 1$ that a bin contains.
3. In Sect. 3.3, we combine these to bound the total number of overflowing balls.

3.1 Bounding the Number of Overflowing Bins

For every bin $\ell \in [n]$, let us define a random 0-1 variable Z_ℓ such that Z_ℓ is 1 if bin ℓ contains more than $\lceil \frac{m}{n} \rceil + 1$ balls after $\mathsf{OfflineTwoChoiceAllocation}$ returns and 0 otherwise. What we want is to bound is the random variable $Z = \sum_{\ell=1}^{n} Z_i$, representing the total number of overflowing bins. Unfortunately we cannot use a Chernoff bound directly, since (i) the variables Z_i are not independent; (ii) we do not know the exact expectation $\mathbb{E}[Z]$. However, we observe that if we show that the variables Z_i are *negatively associated* (at a high level negative association indicates that for a set of variables, whenever some of them increase the rest tend to decrease—see the Appendix for a precise definition) and if we get an *upper bound* on $\mathbb{E}[Z]$ we can then derive a Chernoff-like bound for the number of overflowing bins. We begin by proving the following.

Lemma 2. *The set of random variables* $Z_1, Z_2, \ldots, Z_n$ *is negatively associated.*

Proof. For all $i \in [n]$, $j \in [n]$ and $k \in [m]$ let X_{ijk} be the random variable such that

$$X_{ijk} = \begin{cases} 1 & \text{if } \mathsf{OfflineTwoChoiceAllocation} \text{ chose the two bins } i \text{ and } j \text{ for ball } k \\ 0 & \text{otherwise} \end{cases}.$$

For each k it holds that $\sum_{i,j} X_{ijk} = 1$, since only one pair of bins is chosen for ball k. Therefore, by [17, Proposition 11], it follows that each set $\mathbf{X}_k = \{X_{ijk}\}_{i\in[n],j\in[n]}$ is negatively associated. Moreover, since the sets $\mathbf{X}_k, \mathbf{X}_{k'}$ for $k \neq k'$ consist of mutually independent variables (as the selection of bins is made independently for each ball), it follows from [17, Proposition 7.1] that the set $\mathbf{X} = \{X_{ijk}\}_{i\in[n],j\in[n],k\in[m]}$ is negatively associated. Now consider the disjoint sets U_ℓ for $\ell \in [n]$ defined as $U_\ell = \{X_{ijk} \mid \mathsf{chosen}[k] = \ell \ \wedge \ (\ell = i \vee \ell = j)\}$, where chosen is the array output by $\mathsf{OfflineTwoChoiceAllocation}$. Let us now define $h_\ell(X_{ijk}, X_{ijk} \in U_\ell) = \sum_{X_{ijk}\in U_\ell} X_{ijk}$ for $\ell \in [n]$. Clearly each h_ℓ is a non-decreasing function and therefore by [17, Proposition 7.2] the set of random

[5] Sanders et al. [28] gave a better bound $O(1/n)^{\lceil \frac{m}{n} \rceil + 1}$ which is $O(1/n)$ since $\lceil m/n \rceil \geq 0$. Our analysis is simplified when we take this looser bound $O(1/n)$.

variables $\mathbf{Y} = \{Y_\ell\}_{\ell \in [n]}$ where $Y_\ell = h_\ell$ is also negatively associated. We can finally define Z_ℓ for $\ell = 1, \ldots, n$ as

$$Z_\ell = f(Y_\ell) = \begin{cases} 0 & \text{if } Y_\ell \leq \lceil m/n \rceil + 1 \\ 1 & \text{otherwise} \end{cases}.$$

Since f is also a non-decreasing function (as whenever Y_ℓ grows, $Z_\ell = f(Y_\ell)$ may only increase) therefore, again by [17, Proposition 7.2], it follows that the set of random variables $Z_1, Z_2, \ldots, Z_n$ is also negatively associated. □

Lemma 3. *The expected number of overflowing bins* $\mathbb{E}[Z]$ *is* $O(1)$.

Proof. For all bins $\ell \in [n]$, it is $\mathbb{E}[Z_\ell] = \Pr[Y_\ell > \lceil m/n \rceil + 1] \leq \Pr[L^*_{\mathsf{max}} > \lceil m/n \rceil + 1] = O(1/n)$, by Lemma 1 (where L^*_{max} is the maximum load across all bins after allocation). By linearity of expectation and since $Z = \sum Z_i$, it is $\mathbb{E}[Z] = O(1)$. □

Theorem 1. *Assume* OfflineTwoChoiceAllocation *from Fig. 2 is used to allocate* m *balls into* n *bins. Let* Z *be the number of bins that receive more than* $\lceil m/n \rceil + 1$ *balls. Then there exists a fixed positive constant* c *such that for sufficiently large* n[6] *and for any* $\tau > 1$ *we have* $\Pr[Z \geq c \cdot \tau] \leq \left(\frac{e}{\tau}\right)^{c \cdot \tau}$.

Proof. By Lemma 3 we have that there exists a fixed constant c such that $\mathbb{E}[Z] \leq c$ for sufficiently large n. Therefore, by Lemmas 2 and 8 in the Appendix (where we set $\mu_H = c$ since $\mathbb{E}[Z] \leq c$) we have that for any $\delta > 0$

$$\Pr[Z \geq (1+\delta) \cdot c] \leq \left(\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right)^c \leq \left(\frac{e^{1+\delta}}{(1+\delta)^{(1+\delta)}}\right)^c.$$

Setting $\delta = \tau - 1$ which is > 0 for $\tau > 1$, we get the desired result. □

3.2 Bounding the Overflow Size

Next, we turn our attention to the number of balls Y_ℓ that can be assigned to bin ℓ. In particular, we want to derive a probability bound $\Pr[Y_\ell > \lceil m/n \rceil + \tau]$ defined in general for parameter $\tau \geq 2$—Sanders et al. [28] studied only the case where $\tau = 1$. To do that, we will bound the probability that after OfflineTwoChoiceAllocation returns the maximum load L^*_{max} is larger than $\lceil m/n \rceil + \tau$ for $\tau \geq 2$. We now prove the following.

Theorem 2. *Assume* OfflineTwoChoiceAllocation *from Fig. 2 is used to allocate* m *balls into* n *bins. Let* L^*_{max} *be the maximum load across all bins. Then for any* $\tau \geq 2$

$$\Pr\left[L^*_{\mathsf{max}} \geq \left\lceil \frac{m}{n} \right\rceil + \tau\right] \leq O(1/n)^\tau + O(\sqrt{n} \cdot 0.9^n).$$

[6] This means that there exists a fixed constant n_0 such that for $n \geq n_0$ the statement holds—we provide an estimate of the constants c and n_0 in the extended version [13].

Proof. Our analysis here closely follows the one of [28]. Without loss of generality, we assume the number of balls m to be a multiple of the number of bins n[7] and we will set $b = m/n$. Let now (a_i, b_i) be the two random choices that $\mathsf{OfflineTwoChoiceAllocation}$ makes for ball i where $i = 1, \ldots, m$. For a subset $U \subseteq \{1, \ldots, n\}$ of bins we define the random variables $X_1^U, \ldots, X_m^U$ such that $X_i^U = 1$, if $a_i, b_i \in U$, and 0 otherwise, i.e., X_i^U is 1 only if both selections for the i-th ball are from subset U, which unavoidably leads to this ball being assigned to a bin within subset U. The random variable $L_U = \sum_{i=1}^{m} X_i^U$ is called the *unavoidable load* of U. Also, for a set U and a parameter τ, let $P_U = \Pr[L_U \geq (b+\tau)|U|+1]$. Finally, let L^*_{max} be the *optimal load*, namely the minimum maximum load that can be derived by considering all possible allocations *given* the random choices $(a_1, b_1), \ldots, (a_m, b_m)$. Since $\mathsf{MaxFlowSchedule}$ computes an allocation with the optimal load, we must compute the probability $\Pr[L^*_{\mathsf{max}} > b + \tau]$, where $\tau \geq 2$. From [29, Lemma 5] we have $L^*_{\mathsf{max}} = \max_{\emptyset \neq U \subseteq \{1,\ldots,n\}} \{L_U/|U|\}$. Thus,

$$\begin{aligned}\Pr[L^*_{\mathsf{max}} > b + \tau] &= \Pr[\exists U \subseteq [n] : L_U/|U| > b + \tau] \\ &\leq \textstyle\sum_{\emptyset \neq U \subseteq [n]} \Pr[L_U \geq (b + \tau)|U| + 1] = \sum_{|U|=1}^{n} \binom{n}{|U|} P_U \,,\end{aligned}$$

where the inequality follows from a simple union bound and for the last step we used the fact that P_U is the same for all sets U of the same cardinality. This is because for all sets U_1 and U_2 with $|U_1| = |U_2|$ we have that $\Pr[L_{U_1} \geq (b + \tau)|U_1| + 1] = \Pr[L_{U_2} \geq (b + \tau)|U_2| + 1]$ since U_1 and U_2 are identically distributed. Next, we need to bound the sum $\sum_{|U|=1}^{n} \binom{n}{|U|} P_U$. For this we will split the sum into three separate summands

$$T_1 = \sum_{1 \leq |U| \leq \frac{n}{8}} \binom{n}{|U|} P_U, \; T_2 = \sum_{\frac{n}{8} < |U| < \frac{nb}{b+\tau}} \binom{n}{|U|} P_U \text{ and } T_3 = \sum_{\frac{nb}{b+\tau} \leq |U| \leq n} \binom{n}{|U|} P_U.$$

We begin with the simple observation that $T_3 = 0$. To see why, note that for $|U| \geq nb/(b + \tau)$ it holds that $P_U = \Pr[L_U \geq (b + \tau)|U| + 1] = \Pr[L_U \geq (b+\tau)nb/(b+\tau)+1] = \Pr[L_U \geq m+1] = 0$ as m is a natural upper bound for L_U (i.e., if both selections fall within U for all balls). Regarding T_2, from [28, Lemma 9] we have $\sum_{\frac{n}{8} < |U| < \frac{nb}{b+1}} \binom{n}{|U|} P^*_U = O(\sqrt{n} \cdot 0.9^n)$, where $P^*_U = \Pr[L_U \geq (b+1)|U|+1]$. Clearly, for all U, $P_U \leq P^*_U$. Moreover, $\sum_{\frac{n}{8} < |U| < \frac{nb}{b+\tau}} P^*_U \leq \sum_{\frac{n}{8} < |U| < \frac{nb}{b+1}} P^*_U$ for all $\tau \geq 2$. Putting it all together, we have

$$T_2 \leq \sum_{\frac{n}{8} < |U| < \frac{nb}{b+1}} \binom{n}{|U|} P^*_U = O(\sqrt{n} \cdot 0.9^n).$$

[7] If not, we pad to $m = n\lceil m'/n \rceil$ balls, where m' is the original number of balls. Then, to get an allocation for the m' balls, we get an allocation for the m balls and we remove the unnecessary balls. Clearly, if L^* is the optimal maximum load for the m' balls, then $L^* \leq L^*_{\mathsf{max}}$ (if $L^* > L^*_{\mathsf{max}}$ you can get a better allocation for the m' balls by allocating m balls, a contradiction) and therefore whatever probability bounds we derive for L^*_{max} holds for L^*.

By Lemma 9 in the Appendix, $T_1 = O(1/n)^{b+\tau} = O(1/n)^{b+\tau}$ for all $\tau \geq 2$ hence for all $\tau \geq 2$ it is $\sum_{|U|=1}^{n} \binom{n}{|U|} P_U = O(1/n)^{\tau}$, as $b \geq 0$, which completes the proof. □

3.3 Bounding the Total Number of Overflowing Balls

Let $T > 0$ be the number of overflowing balls, i.e., $T = \sum_{i=1}^{\ell} Z_i(Y_i - \lceil m/n \rceil - 1)$. Using Theorems 1 and 2, and by a simple application of the law of total probability, we can now prove the following result.

Theorem 3. *Assume* OfflineTwoChoiceAllocation *from Fig. 2 is used to allocate m balls into n bins. Let T be the number of overflowing balls as defined above. Then there exist positive constants c, c_1, c_2 such that for large n and for any $\tau \geq 2$ it is*

$$\Pr[T > c \cdot \tau^2] \leq \left(\frac{e}{\tau}\right)^{c \cdot \tau} + \left(\frac{c_1}{n}\right)^{\tau} + c_2\sqrt{n} \cdot 0.9^n \,.$$

Proof. Define the events $E : T > c \cdot \tau^2$, $E_1 : Z > \tau$ and $E_2 : L^*_{\mathsf{max}} > \lceil m/n \rceil + \tau$, for some $\tau \geq 2$. There is no way there can be more than τ^2 overflowing balls if both the number of overflowing bins and the maximum overflow per bin is at most τ. This implies that $E \subseteq E_1 \cup E_2$. By a standard union bound and applying Theorems 1 and 2, we have $\Pr[E] \leq \left(\frac{e}{\tau}\right)^{c \cdot \tau} + O(1/n)^{\tau} + O(\sqrt{n} \cdot 0.9^n)$, which completes the proof by taking c_1 and c_2 to be the constants in $O(1/n)$ and $O(\sqrt{n} \cdot 0.9^n)$ respectively. □

4 New ORAM with $O(1)$ Locality and $o(\sqrt{n})$ Bandwidth

Our constant-locality SE construction uses an ORAM scheme as a black box. In particular, the ORAM scheme that is used must have the following properties:

1. It needs to have constant locality, meaning that for each oblivious access it should only read $O(1)$ non-contiguous locations in the encrypted memory. Existing ORAM constructions with polylogarithmic bandwidth have *logarithmic* locality. For example, a path ORAM access [33] traverses $\log n$ binary tree nodes stored in non-contiguous memory locations—therefore we cannot use it here. This property is required as our underlying SE scheme must have $O(1)$ locality;
2. It needs to have bandwidth cost $o(\sqrt{n} \cdot \lambda)$. This property is required because we would be applying the ORAM scheme on an array of $O(\log^2 N)$ entries, yielding overall bandwidth equal to $o(\log N \cdot \lambda)$, which would imply sublogarithmic read efficiency for the underlying SE scheme.

We note here that an existing scheme that almost satisfies both properties above is the ORAM construction from [27, Theorem 7] by Ohrimenko et al. (where we set $c = 3$). This ORAM has $O(1)$ locality and $O(n^{1/3} \log n \cdot \lambda)$ bandwidth. However we cannot apply it here due to its failure probability which is $\mathsf{neg}(n)$, where n is the size of the memory array. Unfortunately, since our array

has $O(\log^2 N)$ entries (N is the size of the SE dataset), this gives a probability of failure $\mathsf{neg}(\log^2 N)$ which is not $\mathsf{neg}(N)$.

Our proposed ORAM construction is a hierarchical application of the square-root ORAM construction of Goldreich and Ostrovsky [18]. Here, we provide a description of the amortized version of our construction (i.e., the read-efficiency and locality bounds we achieve are amortized over n accesses) in Fig. 3. The deamortized version of our ORAM construction is achieved using techniques of Goodrich et al. [21] for deamortizing the square root ORAM, in a straightforward manner (formal description and analysis of the deamortized version can be found in the extended version [13]).

ORAM Setup. Given memory M with n index-value pairs $(1, v_1), \ldots, (n, v_n)$ we allocate three main arrays for storage: A of size $n_a = n + n^{2/3}$, B of size $n_b = n^{2/3} + n^{1/3}$, and C of size $n_c = n^{1/3}$. Initially A stores all elements encrypted with CPA-secure encryption and permuted with a pseudorandom permutation[8] $\pi_a : [n_a] \to [n_a]$ and B and C are empty, containing encryptions of dummy values. We also initialize another pseudorandom permutation $\pi_b : [n_b] \to [n_b]$ used for accessing elements from array B. In particular, if an element $x \in [n]$ is stored in array B, it is located at position $\pi_b[\mathsf{Tab}[x]]$ of B, where Tab is a locally-stored hash table mapping an element $x \in [n]$ to $\mathsf{Tab}[x] \in [n_b]$. Note the hash table is needed to index elements in B as $n_b < n$.

ORAM Access. To access element x, the algorithm always downloads, decrypts and sequentially scans array C. Similarly to the square-root ORAM, we consider two cases:

1. *Element x is in C.* In this case the requested element has been found and the algorithm performs two additional dummy accesses for security reasons: it accesses a random[9] position in array A and a random position in array B.
2. *Element x is not in C.* In this case we distinguish the following subcases.
   - Element x is not in B.[10] In this case x can be retrieved by accessing the random position $\pi_a[x]$ of array A. Like previously, the algorithm also accesses a random position in array B.
   - Element x is in B. In this case x can be retrieved by accessing the random position $\pi_b[\mathsf{Tab}[x]]$ of array B. Like previously, the algorithm also accesses a random position in array A.

After the access above, the retrieved element x is written in the next available position of C, the algorithm computes a fresh encryption of C and writes C back to the server. Just like in square-root ORAM, some oblivious reshuffling must occur: In particular, every $n^{1/3}$ accesses, array C becomes full and both C and the contents of B are obliviously reshuffled into B. Every $n^{2/3}$ accesses, when

[8] In practice π_a is implemented with efficient small-domain PRPs (e.g., [22,26,32]).

[9] This position is not entirely random—it is chosen from those that have not been chosen so far.

[10] This can be decided by checking whether $\mathsf{Tab}[x]$ is null or not.

B becomes full, all elements are obliviously reshuffled into A. We describe this reshuffling process next.

Reshuffling, epochs and superepochs. Our algorithm for obliviously accessing an element x described proceeds in *epochs* and *superepochs*. An epoch is defined as a sequence of $n^{1/3}$ accesses. A superepoch is defined as a sequence of $n^{2/3}$ accesses.

At the end of every epoch C becomes full, and all elements in C along with the ones that have been accessed in the current superepoch (and are now stored in B) are obliviously reshuffled into B using a fresh pseudorandom permutation π_b. In our implementation in Fig. 3, we store all the elements that must be reshuffled in an array SCRATCH. After the reshuffling C can be emptied (denoted with $\perp$ Line 30) so that it can be used again in the future. At the end of every superepoch all the elements of the dataset are obliviously reshuffled into array A using a fresh pseudorandom permutation π_a and arrays B, C and SCRATCH are emptied.

Oblivious Sorting with Good Locality. As in previous works, our reshuffling in the ORAM protocol is performed using an oblivious sorting protocol. Since we are using the ORAM scheme in an SE scheme that must have good locality, we must ensure that the oblivious sorting protocol used has good locality as well, i.e., it does not access too many non-contiguous locations. One way to achieve that is to download the whole encrypted array, decrypt it, sort it and encrypt it back. This has excellent locality $L = 1$ but requires linear client space. A standard oblivious sorting protocol such as Batcher's odd-even mergesort [8] does not work either since its locality can be linear.

Fortunately, Goodrich and Mitzenmacher [20] developed an oblivious sorting protocol for an external memory setting that is a perfect fit for our application—see Fig. 16 in the Appendix. The client interacts with the server only by reading and writing b consecutive blocks of memory. We call each b-block access (either for read or write) an *I/O operation*. The performance of their protocol is characterized in the following theorem.

Theorem 4 (Goodrich and Mitzenmacher [20], Goodrich [19]). *Given an array X containing n comparable blocks, we can sort X with a data-oblivious external-memory protocol that uses $O((n/b)\log^2(n/b)$ I/O operations and local memory of $4b$ blocks, where an I/O operation is defined as the read/write of b consecutive blocks of X.*

In the above oblivious sorting protocol, value b (the number of consecutive blocks downloaded/uploaded in one I/O) can be parameterized, affecting the local space accordingly. In our case, we set b to be equal to $n^{1/3}\log^2 n$—see Lines 24 and 29 in Fig. 3, which is enough for achieving constant locality in our SE scheme.

Our final result is as follows (proof can be found in the Appendix).

Theorem 5. *Let n be the size of the memory array and λ be the size of the block. Our ORAM scheme (i) is correct according to Definition 2; (ii) is secure according to Definition 3, assuming pseudorandom permutations and CPA-secure*

Protocol $\langle \sigma, \mathsf{EM} \rangle \leftrightarrow$ OramInitialize$\langle (1^\kappa, \mathsf{M}), \perp \rangle$:

Parse M as $(1, v_1), (2, v_2), \ldots, (n, v_n)$ where $|i, v_i| = \lambda$ (the values are λ bits long);
Let $n_a \leftarrow n + n^{2/3}$, $n_b \leftarrow n^{2/3} + n^{1/3}$, $n_c \leftarrow n^{1/3}$;
Let A, B and C be arrays of size n_a, n_b and n_c respectively. Initialize them with $\mathbf{0}$ entries;
Let $\mathsf{SCRATCH}$ be an array of size n_b. Initialize it with $\mathbf{0}$ entries;
Let $\pi_a : [n_a] \to [n_a]$ and $\pi_b : [n_b] \to [n_b]$ be pseudorandom permutations;
For $i = 1, \ldots, n$, store (i, v_i) at location $\pi_a[i]$ in A;
Encrypt-And-Write arrays A, B, C and $\mathsf{SCRATCH}$ and add them to EM;
Let $\mathsf{count}_a \leftarrow 0$ and $\mathsf{count}_b \leftarrow 0$;
Let Tab be an empty hash table;
Set $\sigma = (\pi_a, \pi_b, \mathsf{Tab}, \mathsf{count}_a, \mathsf{count}_b)$;
return $\langle \sigma, \mathsf{EM} \rangle$;

Protocol $\langle (v_i, \sigma'), \mathsf{EM}' \rangle \leftrightarrow$ OramAccess$\langle (\sigma, i), \mathsf{EM} \rangle$:

Parse σ as $(\pi_a, \pi_b, \mathsf{Tab}, \mathsf{count}_a, \mathsf{count}_b)$ and EM as $(A, B, C, \mathsf{SCRATCH})$;
Increment count_a and count_b;
Read-And-Decrypt array C;
if $(i, v_i) \in C$ **then** ▷ (i, v_i) was accessed before and is stored in C
    $\mathsf{index}_a \leftarrow \pi_a[n + \mathsf{count}_a]$;
    $\mathsf{index}_b \leftarrow \pi_b[n^{2/3} + \mathsf{count}_b]$;
else
    if $\mathsf{Tab}[i] \neq \mathsf{null}$ **then** ▷ (i, v_i) is stored in $B[\mathsf{index}_b]$
        $\mathsf{index}_a \leftarrow \pi_a[n + \mathsf{count}_a]$;
        $\mathsf{index}_b \leftarrow \pi_b[\mathsf{Tab}[i]]$;
    else ▷ (i, v_i) is stored in $A[\mathsf{index}_a]$
        $\mathsf{index}_a \leftarrow \pi_a[i]$;
        $\mathsf{index}_b \leftarrow \pi_b[n^{2/3} + \mathsf{count}_b]$;
Read-And-Decrypt $A[\mathsf{index}_a]$;
Read-And-Decrypt $B[\mathsf{index}_b]$;
Retrieve (i, v_i) from either $A[\mathsf{index}_a]$ or $B[\mathsf{index}_b]$ or C;
$C[\mathsf{count}_b] \leftarrow (i, v_i)$;
Encrypt-And-Write array C;
$\mathsf{Tab}[i] \leftarrow \mathsf{count}_a$;
Encrypt-And-Write element $(\mathsf{Tab}[i], v_i)$ at position count_a of array $\mathsf{SCRATCH}$;
if $\mathsf{count}_a > n^{2/3}$ **then** ▷ Transition to a new superepoch
    Let π_a and π_b be new pseudorandom permutations;
    $\mathsf{count}_a \leftarrow 0$ and $\mathsf{count}_b \leftarrow 0$;
    $\langle \perp, A \rangle \leftrightarrow$ ObliviousSorting$\langle (\pi_a, n_a, n^{1/3} \log^2 n), A \rangle$; ▷ large rebuild
    Set $B \leftarrow \perp$; $C \leftarrow \perp$; $\mathsf{SCRATCH} \leftarrow \perp$; Set $\mathsf{Tab} \leftarrow \perp$;
if $\mathsf{count}_b > n^{1/3}$ **then** ▷ Transition to a new epoch
    Let π_b be new pseudorandom permutation;
    $\mathsf{count}_b \leftarrow 0$;
    $\langle \perp, B \rangle \leftrightarrow$ ObliviousSorting$\langle (\pi_b, n_b, n^{1/3} \log^2 n), \mathsf{SCRATCH} \rangle$; ▷ small rebuild
    Set $C \leftarrow \perp$;
return $\langle (v_i, (\pi_a, \pi_b, \mathsf{Tab}, \mathsf{count}_a, \mathsf{count}_b)), (A, B, C, \mathsf{SCRATCH}) \rangle$;

Fig. 3. Read-only ORAM construction with $O(n^{1/3} \log^2 n \cdot \lambda)$ amortized bandwidth and $O(1)$ amortized locality.

encryption; (ii) has $O(n^{1/3}\log^2 n \cdot \lambda)$ *amortized bandwidth and* $O(1)$ *amortized locality per access and requires client space* $O(n^{2/3}\log n + n^{1/3}\log^2 n \cdot \lambda)$.

Standard deamortization techniques from [21] can be applied to make the overheads of our ORAM worst-case as opposed to amortized. A formal treatment of this is presented in the extended version of our paper [13], giving the following result.

Corollary 1. *Let* $\lambda = \Omega(n^{1/3})$ *bits be the block size. Then our ORAM scheme has* $O(n^{1/3}\log^2 n \cdot \lambda)$ *worst-case bandwidth per access,* $O(1)$ *worst-case locality per access and* $O(n^{1/3}\log^2 n \cdot \lambda)$ *client space.*

5 Allocation Algorithms

As we mentioned in the introduction, to construct our final SE scheme we are going to use a series of *allocation algorithms*. The goal of an allocation algorithm for an SE dataset $\mathcal{D}$ consisting of q keyword lists $\mathcal{D}(w_1), \mathcal{D}(w_2), \ldots, \mathcal{D}(w_q)$ is to store/allocate the elements of all lists into an array $\mathbf{A}$ (or multiple arrays).

Retrieval Instructions. To be useful, an allocation algorithm should also output a hash table Tab such that $\mathsf{Tab}[w]$ contains "instructions" on how to correctly retrieve a keyword list $\mathcal{D}(w)$ after the list is stored. For example, for a keyword list $\mathcal{D}(w)$ that contains four elements stored at positions $5, 16, 26, 27$ of $\mathbf{A}$ by the allocation algorithm, some valid alternatives for the instructions $\mathsf{Tab}[w]$ are: (i) *"access positions* 5, 16, 26, 27 *of array* $\mathbf{A}$"; (ii) *"access all positions from* 3 *to* 28 *of array* $\mathbf{A}$"; (iii) *"access the whole array* $\mathbf{A}$". Clearly, there are different tradeoffs among the above.

Algorithm $(\mathbf{A}, \mathsf{Tab}) \leftarrow \mathsf{AllocateSmall}(\mathcal{D}, N)$: (taken from [6])

Set $\epsilon \leftarrow 1/\log^{1-\gamma} N$;
Let $\mathsf{max} \leftarrow N^{1-\epsilon}$, $C = c_s \cdot \log^{\gamma} N$ and $B \leftarrow N/C$[a] ;
Let $\mathbf{A}$ be an array of B buckets—each bucket has capacity C;
Initialize an empty hash table Tab;
for sizes $s = \mathsf{max}, \mathsf{max}/2, \mathsf{max}/4, \ldots, 1$ **do**
    for each keyword w such that $|\mathcal{D}(w)| = s$ **do**
        Pick α and β from $\{1, \ldots, \frac{B}{s}\}$ independently and uniformly at random;
        Let $\mathbf{A}\{\alpha, s\}$ and $\mathbf{A}\{\beta, s\}$ be two superbuckets;
        Let $x \in \{\alpha, \beta\}$ correspond to the superbucket with the minimum load;
        Store $\mathcal{D}(w)$ horizontally into superbucket $\mathbf{A}\{x, s\}$;
        $\mathsf{Tab}[w] = (s, \alpha, \beta, \perp)$;
if there is a bucket $\mathbf{A}[i]$ that overflows **then return** FAIL;
else
    Pad every bucket $\mathbf{A}[i]$ to C elements using dummy values;
    return $(\mathbf{A}, \mathsf{Tab})$;

[a] Constant c_s can be appropriately chosen in [6].

Fig. 4. Allocation algorithm for small sizes from Asharov et al. [6].

Independence Property. For security purposes, and in particular for simulating the search procedure of the SE scheme, it is important that the instructions $\mathsf{Tab}[w]$ output by an allocation algorithm for a keyword list $\mathcal{D}(w)$ are *independent* of the distribution of the rest of the dataset—intuitively this implies that accessing $\mathcal{D}(w)$ does not reveal information about the rest of the data. This independence property is easy to achieve with a "read-all" algorithm, where the whole array is read every time a keyword is accessed, but this is very inefficient. Another way to achieve this property is to store the lists using a random permutation π—this is actually the allocation algorithm used by most existing SE schemes, e.g., [12]. This "permute" approach has however very bad locality since it requires $|\mathcal{D}(w)|$ random jumps in the memory to retrieve $\mathcal{D}(w)$. In the following we present the details of our allocation algorithms for small, medium, large and huge lists. We begin with some terminology.

5.1 Buckets and Superbuckets

Following terminology from [6], our allocation algorithms use fixed-capacity *buckets* for storage. A bucket with capacity C can store up to C *elements*—in our case an *element* is a keyword-document pair (w, id). To simplify notation, we represent a set of B buckets $A_1, A_2, \ldots, A_B$ as an array $\mathbf{A}$ of B buckets, referring to bucket A_i as $\mathbf{A}[i]$. Additionally, a *superbucket* $\mathbf{A}\{k, s\}$ is a set of the following s consecutive buckets

$$\mathbf{A}[(k-1)s+1], \mathbf{A}[(k-1)s+2], \ldots, \mathbf{A}[ks]\,.$$

We say that we store a keyword list $\mathcal{D}(w) = \{(w, id_1), (w, id_2), \ldots, (w, id_s)\}$ *horizontally* into superbucket $\mathbf{A}\{k, s\}$ when each element (w, id_i) is stored in a separate bucket of the superbucket.[11] Finally, the *load* of a bucket or a superbucket is the number of elements stored in each bucket or superbucket.

5.2 Allocating Small Lists with Two-Dimensional Allocation

For small keyword lists we use the two-dimensional allocation algorithm of Asharov et al. [6], by carefully setting the parameters from scratch. For completeness we provide the algorithm in Fig. 4, which we call $\mathsf{AllocateSmall}$. Let $C = c_s \cdot \log^{\gamma} N$, for some appropriately chosen constant c_s. The algorithm uses $B = N/C$ buckets of capacity C each. It then considers all small keyword lists starting from the largest to the smallest, and depending on the list's size s, it picks two superbuckets from $\{1, 2 \ldots, B/s\}$ uniformly at random, horizontally placing the keyword list into the superbucket with the minimum load. The algorithm records both superbuckets as instructions in a hash table Tab. If, during

[11] E.g., consider an array $\mathbf{A}$ consisting of 20 buckets $\mathbf{A}[1], \mathbf{A}[2] \ldots, \mathbf{A}[20]$ where each bucket $\mathbf{A}[i]$ has capacity $C = 5$. Superbucket $\mathbf{A}\{3, 4\}$ contains the buckets $\mathbf{A}[9], \ldots, \mathbf{A}[12]$. Horizontally storing $\{a_1, a_2, \ldots, a_4\}$ into $\mathbf{A}\{3, 4\}$ means storing a_1 into $\mathbf{A}[9]$, a_2 into $\mathbf{A}[10]$, and so on.

this allocation process some bucket overflows, then the algorithm fails. We now have the following result.

Theorem 6. *Algorithm* AllocateSmall *in Fig. 4 outputs* FAIL *with probability* $\mathsf{neg}(N)$. *Moreover the output array of buckets* $\mathbf{A}$ *occupies space* $O(N)$.

Proof. For the algorithm to fail, the load of some bucket $\mathbf{A}[i]$ (i.e., maximum load) must exceed $O(\log^{\gamma} N)$. We show this probability is negligible for our choice of $\gamma = 2/3 + \delta$. We recall AllocateSmall allocates all keyword lists using a two-dimensional balanced allocation [6]. For our proof we apply [6, Theorem 3.5] that states: For $\mathsf{max} = N^{1-\epsilon}$, $B \geq N/\log N$ and for non-decreasing function $f(N)$ such that $f(N) = \Omega(\log\log N)$, $f(N) = O(\sqrt{\log N})$ and $f(2N) = O(f(N))$ the maximum load of a two-dimensional balanced allocation is $\frac{4N}{B} + O(\log \epsilon^{-1} \cdot f(n))$ with probability at least $1 - O(\log \epsilon^{-1}) \cdot N^{-\Omega(\epsilon \cdot f(N^{\epsilon}))}$. In our case, it $\epsilon = 1/\log^{1-\gamma} N$ and $B = N/\log^{\gamma} N$ and we also pick $f(N) = \sqrt{\log N}$. Note that all conditions for f and B and ϵ of [6, Theorem 3.5] are satisfied assuming $1/2 < \gamma < 1$. Also, for this choice of parameters we have that the probability the maximum load is more than $O(\log^{\gamma} N)$ is at most $O(\log(\log^{1-\gamma} N)) \cdot N^{-\Omega(\ell(N))}$ where $\ell(N)$ is

$$\frac{1}{\log^{1-\gamma} N} \sqrt{\log\left(N^{1/\log^{1-\gamma} N}\right)} = \frac{\sqrt{\log^{\gamma} N}}{\log^{1-\gamma} N} = \log^{3\gamma/2-1} N\,.$$

Since our construction uses $\gamma = 2/3 + \delta$ for any small $\delta > 0$ it is always $3\gamma/2 - 1 > 0$ and therefore the above probability is negligible. □

Note now that a list of size s can be read by accessing s consecutive buckets (i.e., a superbucket), therefore the read efficiency for these lists is $O(\log^{\gamma} N)$.

5.3 Allocating Medium Lists with OTA

The allocation process for medium lists is shown in Fig. 5 and the algorithm is described in Fig. 6. The algorithm uses an array $\mathbf{A}$ of $B = N/\log^{\gamma} N$ buckets, where each bucket has capacity $C = 3 \cdot \log^{\gamma} N$. Just like AllocateSmall, the allocation algorithm for medium sizes stores a list $\mathcal{D}(w)$ of size s horizontally into one of the superbuckets $\mathbf{A}\{1, s\}, \mathbf{A}\{2, s\}, \ldots, \mathbf{A}\{B/s, s\}$.

However, unlike AllocateSmall, the superbucket that is finally chosen to store $\mathcal{D}(w)$ depends only on keyword lists of the *same* size with $\mathcal{D}(w)$ that have already been allocated and not on all other keyword lists encountered so far. In particular, let k_s be the number of keyword lists that have size s. Let also $b_s = B/s$ be the number of superbuckets with respect to size s. To figure out which superbucket to pick for horizontally storing a particular keyword list of size s, the algorithm views the k_s keyword lists as *balls* and the b_s superbuckets as *bins* and performs an offline two-choice allocation of k_s keyword lists (balls) into b_s superbuckets (bins), as described in Sect. 3. When, during this process some superbucket contains $\lceil k_s/b_s \rceil + 1$ keyword lists of size s, any subsequent keyword list of size s meant for this superbucket is instead placed into a stash

$\mathbf{B}_s$ that contains exactly $c \cdot \log^2 N$ buckets of size s each for some fixed constant c derived in Theorem 1. Our algorithm will fail, if

- Some bucket $\mathbf{A}[i]$ overflows (i.e., the number of elements that are eventually stored in $\mathbf{A}[i]$ exceeds its capacity C), which as we show in Lemma 4 never happens; or
- More than $c \cdot \log^2 N$ keyword lists of size s must be stored at some stash $\mathbf{B}_s$, which as we show in Lemma 5 happens with negligible probability.

All the choices that the algorithm makes, such as the two superbuckets originally chosen for every list during the offline two-choice allocation as well as the position in the stash (in case the list was an overflowing one) are recorded in Tab as retrieval instructions. We now prove the following lemma.

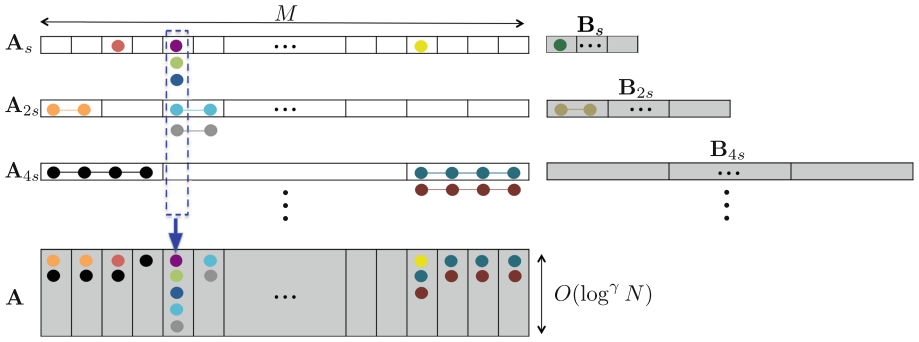

Fig. 5. Allocation of medium lists. Each ball represents a list of size $N^{1-1/\log^{1-\gamma} N}$. Two balls chained together represent a keyword list of double the size and so on. Arrays $\mathbf{A}_i$ show the OTA assignments for all lists of a specific size i. Arrays $\mathbf{A}_i$ are merged into array $\mathbf{A}$ of M buckets of capacity $O(\log^\gamma N)$ each. Overflowing lists of size i are placed in the stash $\mathbf{B}_i$. Only light-gray arrays are stored at the server—white arrays are only used for illustrating the intermediate results.

Lemma 4. *During the execution of algorithm* AllocateMedium *in Fig. 6, no bucket* $\mathbf{A}[i]$ *(for all* $i = 1, \ldots, B$*) will ever overflow.*

Proof. For each size $s = 2\mathsf{min}, 4\mathsf{min}, \ldots, \mathsf{max}$, Line 15 of AllocateMedium allows at most $\lceil k_s/b_s \rceil + 1$ keyword lists of size s to be stored in any superbucket $\mathbf{A}\{i, s\}$. Since every keyword list of size s is stored horizontally in a superbucket $\mathbf{A}\{i, s\}$, it follows that every bucket $\mathbf{A}[i]$ within every superbucket $\mathbf{A}\{i, s\}$ will have load, due to keywords lists of size s, at most $s \cdot (\lceil k_s/b_s \rceil + 1)/s = \lceil k_s/b_s \rceil + 1$. Therefore the total load of a bucket $\mathsf{A}[i]$ due to all sizes $s = 2\mathsf{min}, 4\mathsf{min}, \ldots, \mathsf{max}$ is at most $\sum_s \left(\left\lceil \frac{k_s}{b_s} \right\rceil + 1 \right) \leq \sum_s \frac{k_s}{b_s} + \sum_s 2$. We now bound the above sums separately. Since $b_s = B/s$, $\sum_s k_s \cdot s \leq N$ and $B = N/\log^\gamma N$ it is $\sum_s \frac{k_s}{b_s} = \frac{1}{B} \sum_s k_s \cdot s \leq \frac{N}{B} = \log^\gamma N$. As $\mathsf{min} = 2^{\log N - \log^\gamma N + 1}$, $\mathsf{max} = N/\log^2 N = 2^{\log N - 2 \log\log N}$ and size s takes only powers of 2, there are at most $\log^\gamma N - 2 \log\log N$ terms in the

sum $\sum_s 2$ and therefore $\sum_s \left(\left\lceil \frac{k_s}{b_s} \right\rceil + 1 \right) \leq 3 \cdot \log^\gamma N - 4 \cdot \log\log N \leq 3 \cdot \log^\gamma N$, which equals the bucket capacity C in AllocateMedium. Thus no bucket will ever overflow. □

Lemma 5. *During the execution of algorithm* AllocateMedium *in Fig. 6, no stash* $\mathbf{B}_s$ *(for* $s = 2\mathsf{min}, 4\mathsf{min}, \ldots, \mathsf{max}$*) will ever overflow, except with probability* $\mathsf{neg}(N)$.

Proof. Recall that for each $s = 2\mathsf{min}, 4\mathsf{min}, \ldots, \mathsf{max}$, placing the k_s keyword lists of size s into the b_s superbuckets of size s is performed via an offline two-choice allocation of k_s balls into b_s bins. Also recall that the lists that end up in the stash $\mathbf{B}_s$ (that has capacity $\log^2 N$) are originally placed by the allocation algorithm in superbuckets containing more than $\lceil k_s/b_s \rceil + 1$ keyword lists of size s, thus they are *overflowing*. Let T_s be the number of these lists. By Theorem 3, where we set $T = T_s$ and $n = b_s$ and $\tau = \log N$, we have that for large b_s and for fixed constants c, c_1 and c_2

Algorithm $(\mathbf{A}, \mathcal{B}, \mathsf{Tab}) \leftarrow \mathsf{AllocateMedium}(\mathcal{D}, N)$:

1: Set $\epsilon \leftarrow 1/\log^{1-\gamma} N$;
2: Let $\mathsf{min} \leftarrow N^{1-\epsilon}$, $\mathsf{max} \leftarrow \frac{N}{\log^2 N}$, $C \leftarrow 3 \cdot \log^\gamma N$, $B \leftarrow N/C$ and $\ell = c \cdot \log^2 N$;[a]
3: Let $\mathbf{A}$ be an array of B buckets—each bucket has capacity C;
4: Initialize an empty hash table Tab;
5: **for** sizes $s = 2\mathsf{min}, 4\mathsf{min}, \ldots, \mathsf{max}$ **do**
6: Let $\mathbf{B}_s$ be an array of ℓ buckets—each bucket has capacity s; ▷ This is the stash
7: Let k_s be the number of keywords in $\mathcal{D}$ with $|\mathcal{D}(w)| = s$;
8: Let $b_s \leftarrow B/s$ be the number of superbuckets with respect to size s;
9: Let $\mathsf{inStash}_s \leftarrow 0$; $i \leftarrow 0$;
10: $(\mathsf{chosen}, \mathsf{alternative}) \leftarrow \mathsf{OfflineTwoChoiceAllocation}(k_s, b_s)$;
11: **for** each keyword w such that $|\mathcal{D}(w)| = s$ **do**
12: Increment i;
13: Set $\alpha \leftarrow \mathsf{chosen}[i]$; Set $\beta \leftarrow \mathsf{alternative}[i]$;
14: **if** superbucket $\mathbf{A}\{\alpha, s\}$ contains $\leq \lceil \frac{k_s}{b_s} \rceil$ keyword lists of size s **then**
15: Store $\mathcal{D}(w)$ horizontally into superbucket $\mathbf{A}\{\alpha, s\}$;
16: $\mathsf{Tab}[w] = (s, \alpha, \beta, 1)$;
17: **else** ▷ Move to stash
18: Increment $\mathsf{inStash}_s$;
19: **if** $\mathsf{inStash}_s > \ell$ **then return** FAIL; ▷ Stash overflows
20: Store $\mathcal{D}(w)$ in the bucket $\mathbf{B}_s[\mathsf{inStash}_s]$;
21: $\mathsf{Tab}[w] = (s, \alpha, \beta, \mathsf{inStash}_s)$;
22: **if** there is a bucket $\mathbf{A}[i]$ that has overflown **then return** FAIL;
23: **else** Pad every bucket $\mathbf{A}[i]$ to C elements using dummy values;
24: **return** $(\mathbf{A}, (\mathbf{B}_{2\mathsf{min}}, \mathbf{B}_{4\mathsf{min}}, \mathbf{B}_{8\mathsf{min}}, \mathbf{B}_{16\mathsf{min}} \ldots, \mathbf{B}_{\mathsf{max}}), \mathsf{Tab})$;

[a] Constant c is derived by Theorem 1.

Fig. 6. Allocation algorithm for medium sizes.

$$\Pr[T_s > c \cdot \log^2 N] \leq \left(\frac{e}{\log N}\right)^{c \cdot \log N} + \left(\frac{c_1}{b_s}\right)^{\log N} + c_2\sqrt{b_s} \cdot 0.9^{b_s} = \mathsf{neg}(N)\,,$$

as $b_s = B/s = N/s \log^\gamma N \geq \log^{2-\gamma} N = \omega(\log N)$ as $s \leq \mathsf{max} = N/\log^2 N$. □

Theorem 7. *Algorithm* AllocateMedium *in Fig. 6 outputs* FAIL *with probability* $\mathsf{neg}(N)$. *Moreover, the size of the output array* $\boldsymbol{A}$ *and the stashes* $\mathcal{B}$ *is* $O(N)$.

Proof. AllocateMedium can fail either because a bucket $\mathbf{A}[i]$ overflows, which by Lemma 4 happens with probability 0, or because some stash $\mathbf{B}_s$ ends up having to store more than $\log^2 N$ elements for some $s = 2\mathsf{min}, 4\mathsf{min}, \ldots, \mathsf{max}$, which by Lemma 5 happens with probability $\mathsf{neg}(N)$. For the space complexity, since no bucket $\mathbf{A}[i]$ overflows, array $\mathbf{A}$ occupies space $O(N)$. Also each stash $\mathbf{B}_s$ contains $\log^2 N$ buckets of size s each so the total size required by the stashes is $c \cdot \log^2 N(\mathsf{min} + 2\mathsf{min} + 4\mathsf{min} + \ldots + \mathsf{max})$. Since $\mathsf{max} = N/\log^2 N$, the above is $\leq 2c \log^2 N \mathsf{max} = O(N)$. □

Algorithm $(A, \mathsf{Tab}) \leftarrow \mathsf{AllocateLarge}(\mathcal{D}, N, \mathsf{min}, \mathsf{max})$:

1: Initialize an empty hash table Tab;
2: Let **A** be an array of $t = N/\mathsf{max}$ buckets—each bucket has capacity $2\mathsf{max}$;
3: **for** each keyword w such that $\mathsf{min} < |\mathcal{D}(w)| \leq \mathsf{max}$ **do**
4: **if** there exists a bucket $\mathbf{A}[k]$ with at least $|\mathcal{D}(w)|$ available space **then**
5: Store $\mathcal{D}(w)$ in bucket $\mathbf{A}[k]$;
6: $\mathsf{Tab}[w] \leftarrow (|\mathcal{D}(w)|, k, \perp, \perp)$;
7: **else return** FAIL;
8: **return** (A, Tab);

Fig. 7. Allocation algorithm for large sizes.

5.4 Allocating Large Lists

Recall that we call a keyword list large, if its size is in the range $N/\log^2 N$ and $N/\log^\gamma N$ (recall $\gamma = 2/3 + \delta$). Algorithm AllocateLarge in Fig. 7 is used to allocate lists whose size falls within a specific subrange $(\mathsf{min}, \mathsf{max}]$ of the above range. Let step be an appropriately chosen parameter such that $\mathsf{step} < 3\delta/2$ and partition the range $(N/\log^2 N, N/\log^\gamma N]$ into $\frac{2-\gamma}{\mathsf{step}}$ consecutive subranges[12]

$$\left(\frac{N}{\log^2 N}, \frac{N}{\log^{2-\mathsf{step}} N}\right], \left(\frac{N}{\log^{2-\mathsf{step}} N}, \frac{N}{\log^{2-2\cdot\mathsf{step}} N}\right], \ldots, \left(\frac{N}{\log^{\gamma-\mathsf{step}} N}, \frac{N}{\log^\gamma N}\right].$$

[12] If $\frac{2-\gamma}{\mathsf{step}}$ is not an integer, we round up. Without loss of generality, the last subrange may be of smaller size than the previous ones in order to stop at $N/\log^\gamma N$. Note that, this can only make allocation easier (since it may only reduce the number of lists in the last subrange).

For a given subrange $(\mathsf{min}, \mathsf{max}]$, AllocateLarge stores all keyword lists in an array $\mathbf{A}$ of $t = N/\mathsf{max}$ buckets of capacity $2\mathsf{max}$ each. In particular, for a large keyword list $\mathcal{D}(w)$ of size s, the algorithm places the list in the first bucket that it can find with available space. We later prove that there will always be such a bucket, and therefore no overflow will ever happen. The formal description of the algorithm is shown in Fig. 7.

Theorem 8. *Algorithm* AllocateLarge *in Fig. 7 never outputs* FAIL.

Proof. Assume AllocateLarge fails. This means that at the time some list $\mathcal{D}(w)$ is considered, *all* buckets of $\mathbf{A}$ store at least $2\mathsf{max} - s + 1$ elements each. Therefore the total number of elements considered so far is $\frac{N}{\mathsf{max}}(2\mathsf{max} - s + 1) \geq \frac{N}{\mathsf{max}}(\mathsf{max} + 1) \geq N + \frac{N}{\mathsf{max}} \geq N + \log^{\gamma} N$, since $s \leq \mathsf{max} \leq N/\log^{\gamma} N$. This is a contradiction, however, since the number of entries of our dataset is exactly N. □

5.5 Allocating Huge Lists with a Read-All Algorithm

Keyword lists that have size greater than $N/\log^{\gamma} N$ up to N are stored directly in an array $\mathbf{A}$ of N entries, one after the other—see Fig. 8. To read a huge list in our actual construction, one would have to read the whole array $\mathbf{A}$—however, due to the huge size of the list, the read efficiency would still be small.

Algorithm $(A, \mathsf{Tab}) \leftarrow \mathsf{AllocateHuge}(\mathcal{D}, N)$:

1: Let $\mathsf{min} \leftarrow N/\log^{\gamma} N$;
2: Initialize an empty hash table Tab;
3: Let $\mathbf{A}$ be an array of N entries; $\mathsf{count} \leftarrow 1$;
4: **for** all keywords w such that $|\mathcal{D}(w)| > \mathsf{min}$ **do**
5: Store $\mathcal{D}(w)$ in positions $\mathsf{count}, \mathsf{count}+1, \ldots, \mathsf{count}+|\mathcal{D}(w)|-1$ of array $\mathbf{A}$;
6: $\mathsf{count} \leftarrow \mathsf{count} + |\mathcal{D}(w)|$;
7: $\mathsf{Tab}[w] \leftarrow (|\mathcal{D}(w)|, \perp, \perp, \perp)$;
8: **return** $(\mathbf{A}, \mathsf{Tab})$;

Fig. 8. Allocation algorithm for huge sizes.

6 Our SE Construction

We now present our main construction that uses the ORAM scheme presented in Sect. 4 and the allocation algorithms presented in Sect. 5 as black boxes. Our formal protocols are shown in Figs. 9 and 10.

6.1 Setup Protocol of SE Scheme

Our setup algorithm allocates lists depending on whether they are small, medium, large or huge, as defined in Sect. 5. We describe the details below.

Small Keyword Lists. These are allocated to superbuckets using AllocateSmall from Sect. 5.2. The allocation algorithm outputs an array of buckets $\mathbf{S}$ storing

the small keyword lists and the instructions hash table Tab_S storing, for each small keyword list $\mathcal{D}(w)$, its size s and the superbuckets α and β assigned for this keyword list by the allocation algorithm. The setup protocol of the SE scheme finally encrypts and writes bucket array $\mathbf{S}$ and stores it remotely—see Line 5 in Fig. 9. It stores Tab_S locally.

Medium Keyword Lists. These are allocated to superbuckets using AllocateMedium from Sect. 5.3. AllocateMedium outputs (i) an array of buckets $\mathbf{M}$; (b) the set of stashes $\{\mathbf{B}_s\}_s$ that handle the overflows, for all sizes s in the range; (iii) the instructions hash table Tab_M storing, for each keyword list $\mathcal{D}(w)$ that falls into this range, its size s, the superbuckets α and β assigned for this keyword list and a stash position x in the stash $\mathbf{B}_s$ where the specific keyword list could have been potentially stored, had it caused an overflow (otherwise a dummy position is stored). The setup protocol finally encrypts and writes $\mathbf{M}$ and stores it remotely—see Line 8 in Fig. 9. It also builds an ORAM per stash $\mathbf{B}_s$—see Line 15 in Fig. 9. Finally, it stores Tab_M locally.

Large Keyword Lists. These are allocated to buckets using AllocateLarge from Sect. 5.4. To keep read efficiency small, we run AllocateLarge for $\frac{2-\gamma}{\mathsf{step}}$ distinct subranges, as we detailed in Sect. 5. For the subrange of $(N/\log^{2-(h-1)\cdot\mathsf{step}} N, N/\log^{2-h\cdot\mathsf{step}} N]$, AllocateLarge outputs an array of buckets $\mathbf{L}_h$ and a hash table $\mathsf{Tab}_{\mathbf{L}_h}$. The setup protocol builds an ORAM for the array $\mathbf{L}_h$ and it stores $\mathsf{Tab}_{\mathbf{L}_h}$ locally.

Huge Keyword Lists. For these lists, we use AllocateHuge from Sect. 5.5. This algorithm outputs an array $\mathbf{H}$ and a hash table Tab_H. Our setup protocol encrypts and writes $\mathbf{H}$ remotely and stores Tab_H locally.

Local State and Using Tokens. For the sake of simplicity and readability of Fig. 9, we assume that the client keeps locally the hash table Tab—see Line 13. This occupies linear space $O(N)$ but can be securely outsourced using standard SE techniques [31], and without affecting the efficiency (read efficiency and locality): For every hash table entry $w \rightarrow [s, \alpha, \beta, x]$, store at the server the "encrypted" hash table entry $t_w \rightarrow \mathsf{ENC}_{k_w}(s||\alpha||\beta||x)$, where t_w and k_w comprise the *tokens* for keyword w (these are the outputs of a PRF applied on w with two different secret keys that the client stores) and ENC is a CPA-secure encryption scheme. To search for keyword w, the client just needs to send to the server the tokens t_w and k_w and the server can then search the encrypted hash table and retrieve the information $s||\alpha||\beta||x$ by decrypting.

Handling ORAM State and Failures. Our setup protocol does not store locally the ORAM states σ_s and σ_h of the stashes $\mathbf{B}_s$ and the arrays $\mathbf{L}_h$ for which we build an ORAM. Instead, it encrypts and writes them remotely and downloads them when needed—see Line 17 in Fig. 9. Also, our setup algorithm fails whenever any of the allocation algorithms fail. By Theorems 6, 7 and 8 we have the following:

Lemma 6. *Protocol* Setup *in Fig. 9 fails with probability* $\mathsf{neg}(N)$.

Protocol $\langle st, \mathcal{I} \rangle \leftrightarrow \text{SETUP}\langle (1^\kappa, \mathcal{D}), \perp \rangle$:

1: Let $N \leftarrow \sum_{w \in \mathbf{W}} |\mathcal{D}(w)|$; Set $\mathsf{step} < 3\delta/2$;
2: Let Tab be an empty hash table of capacity N;
3: $(\mathbf{S}, \mathsf{Tab}_S) \leftarrow \mathsf{AllocateSmall}(\mathcal{D}, N)$;
4: **for** all buckets $\mathbf{S}[i] \in \mathbf{S}$ **do**
5: **`Encrypt-And-Write`** bucket $\mathbf{S}[i]$ and add encrypted $\mathbf{S}[i]$ to server index $\mathcal{I}$;
6: $(\mathbf{M}, \mathbf{B}_M, \mathsf{Tab}_M) \leftarrow \mathsf{AllocateMedium}(\mathcal{D}, N)$;
7: **for** all buckets $\mathbf{M}[i] \in \mathbf{M}$ **do**
8: **`Encrypt-And-Write`** bucket $\mathbf{M}[i]$ and add encrypted $\mathbf{M}[i]$ to server index $\mathcal{I}$;
9: **for** $h = 1, \ldots, \frac{2-\gamma}{\mathsf{step}}$ **do**
10: $(\mathbf{L}_h, \mathsf{Tab}_{\mathbf{L}_h}) \leftarrow \mathsf{AllocateLarge}(\mathcal{D}, N, N/\log^{2-(h-1)\cdot\mathsf{step}} N, N/\log^{2-h\cdot\mathsf{step}} N)$;
11: $(\mathbf{H}, \mathsf{Tab}_H) \leftarrow \mathsf{AllocateHuge}(\mathcal{D}, N)$;
12: **`Encrypt-And-Write`** array $\mathbf{H}$ and add encrypted $\mathbf{H}$ to server index $\mathcal{I}$;
13: Set $\mathsf{Tab} \leftarrow \mathsf{Tab}_S \cup \mathsf{Tab}_M \cup \left(\bigcup_{h=1}^{\frac{2-\gamma}{\mathsf{step}}} \mathsf{Tab}_{\mathbf{L}_h} \right)$;
14: $st \leftarrow \mathsf{Tab}$;
15: **for** every stash $\mathbf{B}_s \in \mathbf{B}_M$ corresponding to size s **do**
16: $\langle \sigma_s, \mathsf{EM}_s \rangle \leftrightarrow \text{ORAMINITIALIZE}\langle (1^\kappa, \mathbf{B}_s), \perp \rangle$;
17: **`Encrypt-And-Write`** σ_s and add σ_s and EM_s to server index $\mathcal{I}$;
18: **for** $h = 1, \ldots, \frac{2-\gamma}{\mathsf{step}}$ **do**
19: $\langle \sigma_h, \mathsf{EM}_h \rangle \leftrightarrow \text{ORAMINITIALIZE}\langle (1^\kappa, \mathbf{L}_h), \perp \rangle$;
20: **`Encrypt-And-Write`** σ_h and add σ_h and EM_h to server index $\mathcal{I}$;
21: **if** AllocateSmall or AllocateMedium or AllocateLarge called above output FAIL **then**
22: **return** FAIL;
23: **return** $\langle st, \mathcal{I} \rangle$;

Fig. 9. The setup protocol of our SE construction.

Lemma 7. *Protocol* SETUP *in Fig. 9 outputs an encrypted index* $\mathcal{I}$ *that has* $O(N)$ *size and runs in* $O(N)$ *time.*

Proof. The space complexity follows from Theorems 6 and 7, by the fact that array $\mathbf{H}$ output by AllocateHuge has size $O(N)$, by the fact that we keep a number of arrays for large keyword lists that is independent of N, and by the fact that the ORAM states σ_s and σ_h, being asymptotically less than the ORAM themselves, occupy at most linear space. For the running time, note that AllocateSmall, AllocateLarge, AllocateHuge run in linear time and the ORAM setup algorithms also run in linear time (same analysis with the space can be made). By Lemma 1, AllocateMedium must perform a costly $O(n^3)$ offline allocation (a maximum flow computation) where n is the number of superbuckets defined for every size s in the range. The maximum number of superbuckets M is achieved for the smallest size handled by AllocateMedium and is equal to $M = \frac{N}{N^{1-1/\log^{1-\gamma} N} \cdot \log^\gamma N} = N^{1/\log^{1-\gamma} N} / \log^\gamma N$.

Recall that there are at most $\log^\gamma N$ sizes handled by AllocateMedium and therefore the time required to do the offline allocation is at most $O\left(\log^\gamma N \cdot M^3\right)$

Protocol $\langle(\mathcal{D}(w), st'), \mathcal{I}'\rangle \leftrightarrow \textsc{Search}\langle(st, w), \mathcal{I}\rangle$:

Parse st as Tab and $\mathcal{I}$ as $(\mathbf{S}, \mathbf{M}, \mathbf{H}, \{\sigma_s, \mathsf{EM}_s\}, \{\sigma_h, \mathsf{EM}_h\})$;
Let $(s, \alpha, \beta, x) \leftarrow \mathsf{Tab}[w]$; Set $\mathsf{step} < 3\delta/2$;
if $s > N/\log^{\gamma} N$ **then** ▷ Huge sizes
  Read-And-Decrypt array $\mathbf{H}$;
  Retrieve $\mathcal{D}(w)$ from $\mathbf{H}$;
else
  if $s \leq N^{1-1/\log^{1-\gamma} N}$ **then** ▷ Small sizes
    Read-And-Decrypt superbuckets $\mathbf{S}\{\alpha, s\}$ and $\mathbf{S}\{\beta, s\}$;
    Retrieve $\mathcal{D}(w)$ from $\mathbf{S}\{\alpha, s\}$ and $\mathbf{S}\{\beta, s\}$;
  else
    if $N^{1-1/\log^{1-\gamma} N} N < s \leq N/\log^2 N$ **then** ▷ Medium sizes
      Read-And-Decrypt σ_s;
      $\langle(v_x, \sigma_s), \mathsf{EM}_s\rangle \leftrightarrow \textsc{OramAccess}\langle(\sigma_s, x), \mathsf{EM}_s\rangle$;
      Encrypt-And-Write σ_s;
      Read-And-Decrypt superbuckets $\mathbf{M}\{\alpha, s\}$ and $\mathbf{M}\{\beta, s\}$;
      Retrieve $\mathcal{D}(w)$ from $\mathbf{M}\{\alpha, s\}$ and $\mathbf{M}\{\beta, s\}$ or v_x;
    else ▷ Large sizes
      Find $h \in \{1, \ldots, \frac{2-\gamma}{\mathsf{step}}\}$ s.t. $N/\log^{2-(h-1)\cdot\mathsf{step}} N < s \leq N/\log^{2-h\cdot\mathsf{step}} N$;
      Read-And-Decrypt σ_h;
      $\langle(v_\alpha, \sigma_h), \mathsf{EM}_h\rangle \leftrightarrow \textsc{OramAccess}\langle(\sigma_h, \alpha), \mathsf{EM}_h\rangle$;
      Encrypt-And-Write σ_h;
      Retrieve $\mathcal{D}(w)$ from v_α;
return $\langle(\mathcal{D}(w), st), \mathcal{I}\rangle$;

Fig. 10. The search protocol of our SE construction.

which is equal to $O(N^{3/\log^{1-\gamma} N}/\log^{2\gamma} N) = O(N)$. Therefore, the running time is $O(N)$. □

6.2 Search Protocol of SE Scheme

Given a keyword w, the client first retrieves information (s, α, β, x) from $\mathsf{Tab}[w]$. Depending on the size s of $\mathcal{D}(w)$ the client takes the following actions (see Fig. 10):

- If the list $\mathcal{D}(w)$ is *small*, the client reads two superbuckets $\mathbf{S}\{\alpha, s\}$ and $\mathbf{S}\{\beta, s\}$ and decrypts them. Since the size of the buckets $\mathbf{S}[i]$ is $\log^{\gamma} N$ and each superbucket contains s of them, it follows that the read efficiency for small sizes is $\Theta(\log^{\gamma} N)$. Also, since only two superbuckets are read, the locality for small lists is $O(1)$.
- If the list $\mathcal{D}(w)$ is *medium*, the client reads two superbuckets $\mathbf{M}\{\alpha, s\}$ and $\mathbf{M}\{\beta, s\}$ and decrypts them. Also he performs an ORAM access in the stash $\mathbf{B}_s$ for location x. Since the size of the buckets $\mathbf{M}[i]$ is $O(\log^{\gamma} N)$ and each superbucket has s of them, the read efficiency for medium sizes due to accessing array $\mathbf{M}$ is $O(\log^{\gamma} N)$.
  For the ORAM access, note that in our case it is $n = c \cdot \log^2 N$. Therefore, by Corollary 1, and since our block size is at least $N^{1-1/\log\log N}$

which is $\Omega(\log^{2/3} N)$, the bandwidth required is $O(n^{1/3} \log^2 n \cdot s) = O(\log^{2/3} N \log^2 \log N \cdot s)$ and therefore the read efficiency due to the ORAM access is $O(\log^{2/3} N \log^2 \log N) = o(\log^\gamma N)$, since $\gamma = 2/3 + \delta$. Therefore, the overall read efficiency for medium sizes is $O(\log^\gamma N)$. Again, since only two superbuckets are read and the ORAM locality is $O(1)$ (Corollary 1), it follows that the locality for medium lists is $O(1)$.

– Suppose now the list $\mathcal{D}(w)$ is large such that $\mathsf{min} < |\mathcal{D}(w)| \leq \mathsf{max}$ where $\mathsf{min} = N/\log^{2-(h-1)\cdot\mathsf{step}} N$ and $\mathsf{max} = N/\log^{2-h\cdot\mathsf{step}} N$ for some $h \in \{1, 2, \ldots, \frac{2-\gamma}{\mathsf{step}}\}$. To retrieve the list, our search algorithm performs our ORAM access on an array on N/max blocks of size $2 \cdot \mathsf{max}$ each. By Corollary 1, we have that the worst-case bandwidth for this access is

$$O\left(\left(\frac{N}{\mathsf{max}}\right)^{1/3} \log^2\left(\frac{N}{\mathsf{max}}\right) \mathsf{max}\right) = O\left(N \left(\log^{2-h\cdot\mathsf{step}} N\right)^{-2/3} \log^2 \log N\right).$$

For read efficiency, note that the client must use this bandwidth to read a keyword list of size $s \geq \mathsf{min} = N/\log^{2-(h-1)\cdot\mathsf{step}} N$. Thus, the read efficiency is at most

$$O\left(\log^{2-(h-1)\cdot\mathsf{step}} N \cdot \left(\log^{2-h\cdot\mathsf{step}} N\right)^{-2/3} \log^2 \log N\right) = O(\log^{\gamma'} N \log^2 \log N),$$

where for all $h \geq 1$ it is $\gamma' \leq 2/3 + 2 \cdot \mathsf{step}/3 < 2/3 + \delta = \gamma$ since $\mathsf{step} < 3\delta/2$. Therefore, the above is $o(\log^\gamma N)$ as required.

– For huge sizes, the read efficiency is at most $O(\log^\gamma N)$ and the locality is constant since the whole array $\mathbf{H}$ is read.

Therefore, overall, the locality is $O(1)$, the read efficiency is $O(\log^\gamma N)$ and the space required at the server is $O(N)$.

Rounds of Interaction. Our protocol requires $O(1)$ rounds for interaction for each query. In particular, for small and huge list sizes our construction requires a single round of interaction, as can be easily inferred from Fig. 10. For medium and large sizes, the deamortized version of our protocol which uses the deamortized ORAM from the extended version [13], requires four rounds of interaction.

Client Space. Finally, we measure the storage at the client (assuming, as discussed in Sect. 6.1 that Tab is stored at the server). For small lists, it follows from our above analysis for read efficiency that the storage at the client is $O(\log^\gamma N \cdot s)$. Note that, from Corollary 1, for medium and large list sizes the necessary space at the client due to the ORAM protocol is $O(n^{1/3} \log^2 n \cdot s)$, where n is the number of ORAM indices and s is the result list size (this result uses the deamortized version of our ORAM from the extended version [13]). Since $n \leq \log^2 N$, this becomes $O(\log^{2/3} N \log^2 \log N \cdot s)$. Specifically for medium lists, the client also needs to download two superbuckets for total storage $O(\log^\gamma N \cdot s)$. For huge list sizes, recall that the client downloads the entire array $\mathbf{H}$ which results in space $O(N)$. However, note that in this case $s > N/\log^\gamma N$, therefore $N < s \cdot \log^\gamma N$

and the client storage can be written as $O(\log^{\gamma} N \cdot s)$. We stress that any searchable encryption scheme requires $\Omega(s)$ space at the client simply to download the result of a query. Thus, in all cases our scheme imposes just a multiplicative overhead for the client storage that is sub-logarithmic in the database size, compared to the minimum requirement. Moreover, we stress that this storage is *transient*, i.e., it is only necessary when issuing a query; between queries, the client requires $O(1)$ space.

6.3 Security of Our Construction

We now prove the security of our construction. For this, we build a simulator SimSetup and SimSearch in Figs. 11 and 12 respectively.

Algorithm $(st_{\mathcal{S}}, \mathcal{I}_0) \leftarrow \text{SimSetup}(1^{\kappa}, \mathcal{L}_1(\mathcal{D}_0))$:

Parse $\mathcal{L}_1(\mathcal{D}_0)$ as N;
Let $\mathbf{S}$ to be an array that contains N dummy elements; **Encrypt-And-Write** $\mathbf{S}$;
Let $\mathbf{M}$ to be an array of N dummy elements; **Encrypt-And-Write** $\mathbf{M}$;
for $h = 1, 2, \ldots, \frac{2-\gamma}{\mathsf{step}}$ **do**
    Set $\mathsf{max} = N/\log^{2-h\cdot\mathsf{step}} N$;
    $(st^h_{\mathcal{S}}, \mathsf{EM}_h) \leftarrow \text{SimOramInitialize}(1^{\kappa}, (N/\mathsf{max}, \mathsf{max}))$;
    Parse $st^h_{\mathcal{S}}$ as σ_h; **Encrypt-And-Write** σ_h;
Let $\mathbf{H}$ be an array of N dummy elements; **Encrypt-And-Write** $\mathbf{H}$;
Let $\mathsf{min} = N^{1-1/\log^{1-\gamma} N}$ and $\mathsf{max} = N/\log^2 N$
for $s = 2\mathsf{min}, 4\mathsf{min}, 8\mathsf{min}, \ldots, \mathsf{max}$ **do**
    Set $\mathbf{B}_s$ to be an array of $c \cdot \log^2 N$ entries of s dummy elements each;
    $(st^s_{\mathcal{S}}, \mathsf{EM}_s) \leftarrow \text{SimOramInitialize}(1^{\kappa}, (c \cdot \log^2 N, s))$;
    Parse $st^s_{\mathcal{S}}$ as σ_s; **Encrypt-And-Write** σ_s;
Let $\mathsf{messages}$ be an empty hash table;
Set $\mathcal{I}_0 = (\mathbf{S}, \mathbf{M}, \mathbf{H}, \{\sigma_s, \mathsf{EM}_s\}, \{\sigma_h, \mathsf{EM}_h\})$;
return $((N, \mathsf{messages}, \{st^s_{\mathcal{S}}\}, \{st^h_{\mathcal{S}}\}, \mathcal{I}_0)$;

Fig. 11. The simulator of the setup protocol of our SE scheme.

Simulation of the Setup Protocol. To simulate the setup protocol, our simulator must output $\mathcal{I}_0$ by just using the leakage $\mathcal{L}_1(D_0) = N$. Our SimSetup algorithm outputs $\mathcal{I}_0$ as CPA-secure encryptions of arrays $(\mathbf{S}, \mathbf{M}, \mathbf{H})$ that contain dummy values and have the same dimensions with the arrays of the actual setup algorithm. Also, it calls the ORAM simulator and also outputs $\{\sigma_s, \mathsf{EM}_s\}$) and $\{\sigma_h, \mathsf{EM}_h\}$). Due to the security of the underlying ORAM scheme and the CPA-security of the underlying encryption scheme, the adversary cannot distinguish between the two outputs.

One potential problem, however, is the fact that SimSetup always succeeds while there is a chance that the setup algorithm can fail, which will enable the adversary to distinguish between the two. However, by Lemma 6, this happens with probability $\mathsf{neg}(N) = \mathsf{neg}(\kappa)$, as required by our security definition, Definition 1.

Simulation of the Search Protocol. The simulator of the SEARCH protocol is shown in Fig. 12. For a keyword query w_k, the simulator takes as input the leakage $\mathcal{L}_2(w_k) = (s, b)$, as defined in Relation 1.

If the query on w_k was performed before (thus $b \neq \perp$), the simulator just outputs the previous messages M_b plus the messages that were output by the ORAM simulator.

If the query on w_k was not performed before, then the simulator generates the messages M_k depending on the size s of the list $\mathcal{D}(w_k)$. In particular note that all accesses on $(\mathbf{S}, \mathbf{M}, \mathbf{H}, \mathbf{L}_h)$ are independent of the dataset and therefore can be simulated by repeating the same process with the real execution.

7 Conclusions and Observations

Basing the Entire Scheme on ORAM. Our construction is using ORAM as a black box and therefore one could wonder why not use ORAM from the very beginning and on the whole dataset. While ORAM can provide much better security guarantees, it suffers from high read efficiency. E.g., to the best of our knowledge, there is no ORAM that we could use that yields sublogarithmic read efficiency (irrespective of the locality).

Avoiding the Lower Bound of [6]. We note that Proposition 4.6 by Asharov et al. [6] states that one could not expect to construct an allocation algorithm where the square of the locality $\times$ the read efficiency is $O(\log N/\log\log N)$. This is the case with our construction! The reason this proposition does not apply to our approach is because our allocation algorithm is using multiple structures for storage, e.g., stashes and multiple arrays, and therefore does not fall into the model used to prove the negative result.

Reducing the ORAM Read Efficiency. Our technique for building our ORAM in Sect. 4 relies on one hierarchical application of the method of square-root ORAM [18]. We believe this approach can be generalized to yield read efficiency $O(n^{1/k}\log^2 n \cdot \lambda)$ for general k. The necessary analysis, while tedious, seems technically non-challenging and we leave it for future work (e.g., we could revisit some ideas from [34]). Such an ORAM could also help us decrease the number of subranges on which we apply our AllocateLarge algorithm.

Using Online Two-Choice Allocation. Our construction uses the offline variant of the two-choice allocation problem. This allows us to achieve low bounds on both the number of overflowing bins and the total overflow size in Sect. 3. However it requires executing a maximum flow algorithm during our construction's setup. A natural question is whether we can use instead the (more efficient) *online* two-choice allocation problem. The best known result [9] for the online version yields a maximum load of $O(\log\log n)$ beyond the expected value m/n, which suffices to bound the maximum number of overflowing bins with our technique. However, deriving a similar bound for the total overflow size would require entirely different techniques and we leave it as an open problem. Still, it seems that even if we could get the same bound for the overflow size as in the offline

Algorithm $(st_{\mathcal{S}}, M_k, \mathcal{I}_k) \leftarrow \textsc{SimSearch}(st_{\mathcal{S}}, \mathcal{L}_2(w_k), \mathcal{I}_{k-1})$:

Parse $st_{\mathcal{S}}$ as $(N, \mathsf{messages}, \{st^s_{\mathcal{S}}\}, \{st^h_{\mathcal{S}}\})$;
Parse $\mathcal{I}_{k-1}$ as $(\mathbf{S}, \mathbf{M}, \mathbf{H}, \{\sigma_s, \mathsf{EM}_s\}, \{\sigma_h, \mathsf{EM}_h\})$;
Parse $\mathcal{L}_2(w_k)$ as (s, b);
Set $m_k = null$; $m_1 = null$; $m_2 = null$;
if $N/\log^2 N < s \leq N/\log^{\gamma} N$ **then** ▷ For large sizes, perform a fresh ORAM access
    Find $h \in \{1, 2, \ldots, \frac{2-\gamma}{\mathsf{step}}\}$ such that $N/\log^{2-(h-1)\cdot\mathsf{step}} N < s \leq N/\log^{2-h\cdot\mathsf{step}} N$;
    Read-And-Decrypt σ_h. Let m_1 be this message;
    $(st^h_{\mathcal{S}}, \mathsf{EM}_h, m_k) \leftarrow \textsc{SimORAMAccess}(st^h_{\mathcal{S}}, \mathsf{EM}_h)$;
    Encrypt-And-Write σ_h. Let m_2 be this message;
    Set $\mathsf{messages}[k] \leftarrow null$;
    Set $st_{\mathcal{S}} \leftarrow (N, \mathsf{messages}, \{st^s_{\mathcal{S}}\}, \{st^h_{\mathcal{S}}\})$;
    Set $\mathcal{I}_k \leftarrow (\mathbf{S}, \mathbf{M}, \mathbf{H}, \{\sigma_s, \mathsf{EM}_s\}, \{\sigma_h, \mathsf{EM}_h\})$;
    return $(st_{\mathcal{S}}, (m_k, m_1, m_2), \mathcal{I}_k)$;
if $b \neq \perp$ **then** ▷ Query has been asked before
    if $N^{1-1/\log^{1-\gamma} N} < s \leq N/\log^2 N$ **then**
        Read-And-Decrypt σ_h. Let m_1 be this message;
        $(st^s_{\mathcal{S}}, \mathsf{EM}_s, m_k) \leftarrow \textsc{SimORAMAccess}(st^s_{\mathcal{S}}, \mathsf{EM}_s)$;
        Encrypt-And-Write σ_h. Let m_2 be this message;
    Set $st_{\mathcal{S}} \leftarrow (N, \mathsf{messages}, \{st^s_{\mathcal{S}}\}, \{st^h_{\mathcal{S}}\})$;
    Set $M_k \leftarrow (\mathsf{messages}[b], (m_k, m_1, m_2))$;
    Set $\mathcal{I}_k \leftarrow (\mathbf{S}, \mathbf{M}, \mathbf{H}, \{\sigma_s, \mathsf{EM}_s\}, \{\sigma_h, \mathsf{EM}_h\})$;
    return $(st_{\mathcal{S}}, M_k, \mathcal{I}_k)$;
if $s > N/\log^{\gamma} N$ **then** ▷ Huge sizes
    Read-And-Decrypt array $\mathbf{H}$;
    Add the above message to $\mathsf{messages}[k]$;
if $s \leq N^{1-1/\log^{1-\gamma} N}$ **then** ▷ Small sizes
    Set $C \leftarrow c_s \cdot \log^{\gamma} N$[a] and $B \leftarrow N/C$;
    Pick α and β independently and uniformly at random from $\{1, 2, \ldots, \frac{B}{s}\}$;
    Read-And-Decrypt superbuckets $\mathbf{S}\{\alpha, s\}$ and $\mathbf{S}\{\beta, s\}$;
    Add the above message to $\mathsf{messages}[k]$;
if $N^{1-1/\log^{1-\gamma} N} < s \leq N/\log^2 N$ **then** ▷ Medium sizes
    Set $C = 3 \cdot \log^{\gamma} N$ and $B \leftarrow N/C$;
    Pick α and β independently and uniformly at random from $\{1, 2, \ldots, \frac{B}{s}\}$;
    Read-And-Decrypt superbuckets $\mathbf{M}\{\alpha, s\}$ and $\mathbf{M}\{\beta, s\}$;
    Add the above message to $\mathsf{messages}[k]$;
    Read-And-Decrypt σ_h. Let m_1 be this message;
    $(st^s_{\mathcal{S}}, \mathsf{EM}_s, m_k) \leftarrow \textsc{SimORAMAccess}(st^s_{\mathcal{S}}, \mathsf{EM}_s)$;
    Encrypt-And-Write σ_h. Let m_2 be this message;
Set $st_{\mathcal{S}} \leftarrow (N, \mathsf{messages}, \{st^s_{\mathcal{S}}\}, \{st^h_{\mathcal{S}}\})$;
Set $M_k \leftarrow (\mathsf{messages}[k], (m_k, m_1, m_2))$;
Set $\mathcal{I}_k \leftarrow (\mathbf{S}, \mathbf{M}, \mathbf{H}, \{\sigma_s, \mathsf{EM}_s\}, \{\sigma_h, \mathsf{EM}_h\})$;
return $(st_{\mathcal{S}}, M_k, \mathcal{I}_k)$;

[a] Constant c_s is appropriately chosen in [6].

Fig. 12. The simulator of the search protocol of our SE scheme.

case, the read efficiency would be $O(\log^{\gamma} N \log\log N)$, as opposed to the better $O(\log^{\gamma} N)$, which is what we achieve here.

Reducing the Read Efficiency for Small Lists. The read efficiency of our scheme for small lists can be strictly improved if instead of using [6], we use the construction of Asharov et al. [7] that was proposed in concurrent work. In this manner, the read efficiency for a small keyword list with size $N^{1-\epsilon}$ would be $\omega(1) \cdot \epsilon^{-1} + O(\log\log\log N)$.

Acknowledgments. We thank Jiaheng Zhang for indicating a tighter analysis for Theorem 6 and for his feedback on the algorithm for allocating large keyword lists, and the reviewers for their comments. Work supported in part by NSF awards #1526950, #1514261 and #1652259, HKUST award IGN16EG16, a Symantec PhD fellowship, and a NIST award.

References

1. Crimes 2001 to present (City of Chicago). https://data.cityofchicago.org/public-safety/crimes-2001-to-present/ijzp-q8t2
2. Enron Email Dataset. https://www.cs.cmu.edu/./enron/
3. TPC-H Dataset. http://www.tpc.org/tpch/
4. USPS Dataset. http://www.app.com
5. Asharov, G., Chan, T.H., Nayak, K., Pass, R., Ren, L., Shi, E.: Oblivious computation with data locality. IACR Cryptology ePrint (2017)
6. Asharov, G., Naor, M., Segev, G., Shahaf, I.: Searchable symmetric encryption: optimal locality in linear space via two-dimensional balanced allocations. In: STOC (2016)
7. Asharov, G., Segev, G., Shahaf, I.: Tight tradeoffs in searchable symmetric encryption. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018. LNCS, vol. 10991, pp. 407–436. Springer, Heidelberg (2018)
8. Batcher, K.E.: Sorting networks and their applications. In: AFIPS (1968)
9. Berenbrink, P., Czumaj, A., Steger, A., Vöcking, B.: Balanced allocations: the heavily loaded case. In: STOC (2000)
10. Cash, D., et al.: Dynamic searchable encryption in very-large databases: data structures and implementation. In: NDSS (2014)
11. Cash, D., Tessaro, S.: The locality of searchable symmetric encryption. In: Nguyen, P.Q., Oswald, E. (eds.) EUROCRYPT 2014. LNCS, vol. 8441, pp. 351–368. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-55220-5_20
12. Curtmola, R., Garay, J.A., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: improved definitions and efficient constructions. JCS **9**(5), 895–934 (2011)
13. Demertzis, I., Papadopoulos, D., Papamanthou, C.: Searchable encryption with optimal locality: achieving sublogarithmic read efficiency. In: CRYPTO 2018 (2018). https://eprint.iacr.org/2017/749
14. Demertzis, I., Papadopoulos, S., Papapetrou, O., Deligiannakis, A., Garofalakis, M.: Practical private range search revisited. In: SIGMOD (2016)
15. Demertzis, I., Papadopoulos, S., Papapetrou, O., Deligiannakis, A., Garofalakis, M., Papamanthou, C.: Practical private range search in depth. In: TODS (2018)
16. Demertzis, I., Papamanthou, C.: Fast searchable encryption with tunable locality. In: SIGMOD (2017)

17. Dubhashi, D.P., Ranjan, D.: Balls and bins: a study in negative dependence. Random Struct. Algorithms **13**(2), 99–124 (1998)
18. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious rams. J. ACM **43**(3), 431–473 (1996)
19. Goodrich, M.T.: Data-oblivious external-memory algorithms for the compaction, selection, and sorting of outsourced data. In: SPAA (2011)
20. Goodrich, M.T., Mitzenmacher, M.: Privacy-preserving access of outsourced data via oblivious RAM simulation. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011. LNCS, vol. 6756, pp. 576–587. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22012-8_46
21. Goodrich, M.T., Mitzenmacher, M., Ohrimenko, O., Tamassia, R.: Oblivious RAM simulation with efficient worst-case access overhead. In: CCSW (2011)
22. Granboulan, L., Pornin, T.: Perfect block ciphers with small blocks. In: Biryukov, A. (ed.) FSE 2007. LNCS, vol. 4593, pp. 452–465. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74619-5_28
23. Kamara, S., Papamanthou, C.: Parallel and dynamic searchable symmetric encryption. In: Sadeghi, A.-R. (ed.) FC 2013. LNCS, vol. 7859, pp. 258–274. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39884-1_22
24. Kamara, S., Papamanthou, C., Roeder, T.: Dynamic searchable symmetric encryption. In: CCS (2012)
25. Miers, I., Mohassel, P.: IO-DSSE: scaling dynamic searchable encryption to millions of indexes by improving locality. In: NDSS (2017)
26. Morris, B., Rogaway, P.: Sometimes-recurse shuffle - almost-random permutations in logarithmic expected time. In: Nguyen, P.Q., Oswald, E. (eds.) EUROCRYPT 2014. LNCS, vol. 8441, pp. 311–326. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-55220-5_18
27. Ohrimenko, O., Goodrich, M.T., Tamassia, R., Upfal, E.: The Melbourne shuffle: improving oblivious storage in the cloud. In: Esparza, J., Fraigniaud, P., Husfeldt, T., Koutsoupias, E. (eds.) ICALP 2014. LNCS, vol. 8573, pp. 556–567. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43951-7_47
28. Sanders, P., Egner, S., Korst, J.H.M.: Fast concurrent access to parallel disks. Algorithmica **35**(1), 21–55 (2003)
29. Schoenmakers, L.A.: A new algorithm for the recognition of series parallel graphs. Technical report, Amsterdam, The Netherlands (1995)
30. Song, D.X., Wagner, D., Perrig, A.: Practical techniques for searches on encrypted data. In: SP (2000)
31. Stefanov, E., Papamanthou, C., Shi, E.: Practical dynamic searchable encryption with small leakage. In: NDSS (2014)
32. Stefanov, E., Shi, E.: FastPRP: fast pseudo-random permutations for small domains. IACR Cryptology ePrint (2012)
33. Stefanov, E., et al.: Path ORAM: an extremely simple oblivious RAM protocol. In: CCS (2013)
34. Zahur, S., et al.: Revisiting square-root ORAM: efficient random access in multi-party computation. In: SP (2016)

Appendix

Definition 2 (Correctness of ORAM). *Let* (OramInitialize, OramAccess) *be an ORAM scheme. Let* $\langle \sigma_0, \mathsf{EM}_0 \rangle \leftrightarrow \text{OramInitialize}\langle (1^\kappa, \mathsf{M}_0), 1^\kappa \rangle)$ *for some initial memory* M_0 *of* n *indexed values* $(1, v_1), (2, v_2), \ldots, (n, v_n)$. *Consider* q *arbitrary requests* $i_1, \ldots, i_q$. *We say that the ORAM scheme is correct if* $\langle (v_{i_k}, \sigma_k), \mathsf{EM}_k \rangle$ *are the final outputs of the protocol* $\text{OramAccess}\langle (\sigma_{k-1}, i_k), \mathsf{EM}_{k-1} \rangle$ *for any* $1 \leq k \leq q$, *where* M_k, EM_k, σ_k *are the memory array, the encrypted memory array and the secret state, respectively, after the* k*-th access operation, and* OramAccess *is run between an honest client and server.*

Definition 3 (Security of ORAM). *Assume* (OramInitialize, OramAccess) *is an ORAM scheme. The ORAM scheme is secure if for any PPT adversary* Adv, *there exists a stateful PPT simulator* (SimOramInitialize, SimOramAccess) *such that* $|\Pr[\mathbf{Real}^{\mathsf{ORAM}}(\kappa) = 1] - \Pr[\mathbf{Ideal}^{\mathsf{ORAM}}(\kappa) = 1]| \leq \mathsf{neg}(\kappa)$, *where experiments* $\mathbf{Real}^{\mathsf{ORAM}}(\kappa)$ *and* $\mathbf{Ideal}^{\mathsf{ORAM}}(\kappa)$ *are defined in Fig. 14 and where the randomness is taken over the random bits used by the algorithms of the ORAM scheme, the algorithms of the simulator and* Adv.

Definition 4 (Dubhashi and Ranjan [17]). *A set of random variables* $\{X_1, \ldots, X_n\}$ *is negatively associated if for every two disjoint index sets* $I \in [n]$ *and* $J \subseteq [n]$ *it is*

$$\mathbb{E}[f(X_i, i \in I)g(X_j, j \in J)] \leq \mathbb{E}[f(X_i, i \in I)]\mathbb{E}[g(X_j, j \in J)]$$

(chosen, alternative) ← MaxFlowSchedule$(m, n, \mathsf{A}, \mathsf{B})$

1: Let G be a graph that has n nodes and the following m unit-capacity directed edges $\{(\mathsf{A}[1], \mathsf{B}[1]), (\mathsf{A}[2], \mathsf{B}[2]) \ldots, (\mathsf{A}[m], \mathsf{B}[m])\}$;
2: Let s and t be two new nodes added to G serving as the source and the sink;
3: For all $v \in G$ such that $indeg(v) > \lceil m/n \rceil + 1$, add a directed edge (s, v) of capacity $indeg(v) - (\lceil m/n \rceil + 1)$;
4: For all $v \in G$ such that $indeg(v) < \lceil m/n \rceil + 1$, add a directed edge (v, t) of capacity $(\lceil m/n \rceil + 1) - indeg(v)$;
5: Compute the maximum flow in G from s to t;
6: **if** the maximum flow in G from s to t saturates all the edges having s as origin **then**
7: Change the direction of all edges $(\mathsf{A}[i], \mathsf{B}[i])$ by calling swap$(\mathsf{A}[i], \mathsf{B}[i])$ that carry flow;
8: Let chosen and alternative be empty arrays of m entries;
9: **for** $i = 1$ to m **do**
10: Set chosen$[i] \leftarrow \mathsf{B}[i]$ and alternative$[i] \leftarrow \mathsf{A}[i]$;
11: **return** (chosen, alternative);

Fig. 13. Maximum flow algorithm for finding allocation.

for all $f : \mathbb{R}^{|I|} \to \mathbb{R}$, $g : \mathbb{R}^{|J|} \to \mathbb{R}$ *that are both non-increasing or non-decreasing*[13].

The following lemmas are used when proving Theorems 1 and 2. Proofs appear in the extended version [13].

Lemma 8. *Let* $\{X_1, \ldots, X_n\}$ *be negatively associated* 0-1 *random variables and* X *be their sum. Let* $\mu = \mathbb{E}[X]$ *and* $\mu_H \in \mathbb{R}$ *such that* $\mu < \mu_H$*. Then, for any* $\delta > 0$*, the following version of the Chernoff bound holds:* $\Pr[X \geq (1+\delta)\mu_H] \leq \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mu_H}$.

Lemma 9. *For any set* $U \subseteq \{1, \ldots, n\}$ *and for any* $\tau \geq 2$ *it holds that* $\sum_{1 \leq |U| \leq \frac{n}{8}} \binom{n}{|U|} P_U \leq \left(\frac{|U|}{n}\right)^{(b+\tau-1)|U|+1} \cdot e^{(b+1)|U|+1} = O(1/n)^{b+\tau}$*, where* $P_U = \Pr[L_U \geq (b+\tau)|U| + 1]$ *and* L_U *is the unavoidable load of a subset of bins* U*, where the unavoidable load* L_U *is defined in Sect. 3.2.*

Correctness proof for our ORAM construction. It is enough to prove that for all indices i, (i, v_i) will be always stored either in C or in $A[\pi_a[i]]$ or in $B[\pi_b[\mathsf{Tab}[i]]]$—these are the values from which we retrieve v_i in Line 16 of our construction in Fig. 3. We consider the following disjoint cases.

1. **(i has been accessed since the last reshuffle):** Then, (i, v_i) can be found in C since it was stored there during the last access to it and C has not been emptied since.

$bit \leftarrow \mathbf{Real}^{\mathsf{ORAM}}(\kappa)$:
1: $\mathsf{M}_0 \leftarrow \mathsf{Adv}(1^\kappa)$; $\langle \sigma_0, \mathsf{EM}_0 \rangle \leftrightarrow \textsc{OramInitialize}\langle (1^\kappa, \mathsf{M}_0), 1^\kappa \rangle$;
2: **for** $k = 1$ to q **do**
3: $\quad i_k \leftarrow \mathsf{Adv}(1^\kappa, \mathsf{EM}_0, m_1, \ldots, m_{k-1})$;
4: $\quad \langle (v_{i_k}, \sigma_k), \mathsf{EM}_k \rangle \leftrightarrow \textsc{OramAccess}\langle (\sigma_{k-1}, i_k), \mathsf{EM}_{k-1} \rangle$;
5: $\quad$ Let m_k be the messages from client to server in the $\textsc{OramAccess}$ protocol above;
6: $bit \leftarrow \mathsf{Adv}(1^k, \mathsf{EM}_0, m_1, m_2, \ldots, m_q)$;
7: **return** bit;

$bit \leftarrow \mathbf{Ideal}^{\mathsf{ORAM}}(\kappa)$:
1: $\mathsf{M}_0 \leftarrow \mathsf{Adv}(1^\kappa)$; $(st_{\mathcal{S}}, \mathsf{EM}_0) \leftarrow \textsc{SimOramInitialize}(1^\kappa, |\mathsf{M}_0|)$;
2: **for** $k = 1$ to q **do**
3: $\quad (st_{\mathcal{S}}, \mathsf{EM}_k, m_k) \leftarrow \textsc{SimOramAccess}(st_{\mathcal{S}}, \mathsf{EM}_{k-1})$;
4: $bit \leftarrow \mathsf{Adv}(1^k, \mathsf{EM}_0, m_1, m_2, \ldots, m_q)$;
5: **return** bit;

Fig. 14. Real and ideal experiments for the ORAM scheme.

[13] A function $h : \mathbb{R}^k \to \mathbb{R}$ is non-decreasing when $h(\mathbf{x}) \leq h(\mathbf{y})$ whenever $\mathbf{x} \leq \mathbf{y}$ in the component-wise ordering on $\mathbb{R}^k$.

2. **(i has not been accessed since the last large reshuffle):** Then, (i, v_i) can be found in $A[\pi[i]]$ since during a large reshuffle all the elements of the dataset are reshuffled into A (and stay there if not accessed afterwards).
3. **(i has been accessed since the last large reshuffle but not since the last small reshuffle):** Then, the element can be found in $B[\pi_b[\mathsf{Tab}[i]]]$. This is because, after its first access that occurred after the large reshuffle element i moved to C and after the small reshuffle element i moved to B with a new index $\mathsf{Tab}[i]$ in B and it was stored at location $\pi_b[\mathsf{Tab}[i]]$ during the small reshuffle. Since it was never accessed after the small reshuffle, it remained in B. □

Security proof for our ORAM construction. Our simulator is shown in Fig. 15. Note that all EM_i are trivially indistinguishable from the EM_i output by the real game due to the CPA-security of the encryption scheme that is used—recall that whatever is being written on the server by our protocols is always freshly encrypted. We now argue that the messages $m_1, m_2, \ldots, m_q$ in the real game are indistinguishable from the messages $m_1, m_2, \ldots, m_q$ output by the simulator. This is because for each $1 \leq k \leq q$, the set of message m_k is entirely independent of the queried value i_k had we used truly random permutations for π_a and π_b. This follows from the following facts:

- When accessing i_k, array C is accessed in its entirety. Also $(\mathsf{Tab}[i_k], v_{i_k})$ is uploaded encrypted at a fixed position count_a in $\mathsf{SCRATCH}$ (see Line 20). So both memory accesses are independent of the index i_k.
- When accessing i_k within a specific superepoch, a location $x = \pi_a[y]$ from array A is accessed for the first and last time within the specific superepoch. Since x is the output of a truly random permutation and is accessed only once within the specific superepoch, x is independent of i_k. The same argument applies for the accesses made to array B. Now if we replace the truly random permutation with the pseudorandom permutation of our construction, the adversary can gain a negligible advantage which is acceptable.
- When accessing i_k at the end of the current superepoch, an oblivious sorting is executed whose memory accesses do not depend on the actual data that are being sorted, but only on the size of the array that is being sorted. The same argument applies for the case when i_k is accessed at the end of an epoch. □

Asymptotic complexity or our ORAM scheme. Over the course of n accesses, each access $1 \leq i \leq n$ incurs the following:

- $O(n^{1/3} \cdot \lambda)$ bandwidth and $O(1)$ locality due to access of A, B, C and $\mathsf{SCRATCH}$;
- $O(n^{2/3} \log^2 n \cdot \lambda)$ bandwidth and $O(n^{1/3})$ locality due to the small rebuilding which happens only when $i \bmod n^{1/3} = 0$ (i.e., $n^{2/3}$ times);
- $O(n \log^2 n \cdot \lambda)$ bandwidth and $O(n^{2/3})$ locality due to the large rebuilding which happens only when $i \bmod n^{2/3} = 0$ (i.e., $n^{1/3}$ times).

Algorithm $(st_{\mathcal{S}}, \mathsf{EM}_0) \leftarrow$ SimOramInitialize$(1^\kappa, |\mathsf{M}_0|)$:

1: Let $(n, \lambda) = |\mathsf{M}_0|$; ▷ Recall λ is the size of the ORAM block
2: **for** $i = 1$ to n **do**
3: Set $v_i = \mathbf{0}^\lambda$; $\mathsf{M}_0[i] = (i, v_i)$;
4: $\langle \sigma_0, \mathsf{EM}_0 \rangle \leftrightarrow$ OramInitialize$\langle (1^\kappa, \mathsf{M}_0), \perp \rangle$;
5: **return** $(\sigma_0, \mathsf{EM}_0)$;

Algorithm $(st_{\mathcal{S}}, \mathsf{EM}_k, m_k) \leftarrow$ SimOramAccess$(st_{\mathcal{S}}, \mathsf{EM}_{k-1})$:

Parse $st_{\mathcal{S}}$ as σ_{k-1};
Choose $i_k \in [n]$;
$\langle (v_{i_k}, \sigma_k), \mathsf{EM}_k \rangle \leftrightarrow$ OramAccess$\langle (\sigma_{k-1}, i_k), \mathsf{EM}_{k-1} \rangle$;
Let m_k be the messages sent from client to server during the above OramAccess protocol;
return $(\sigma_k, \mathsf{EM}_k, m_k)$;

Fig. 15. The simulator for the ORAM scheme of Fig. 3

Note that in order to derive the locality of the rebuilding above, we used Theorem 4 for $b = n^{1/3} \log^2 n$. Now, the amortized bandwidth is

$$\lambda \cdot \frac{n \cdot O(n^{1/3}) + n^{2/3} \cdot O(n^{2/3} \log^2 n) + n^{1/3} \cdot O(n \log^2 n)}{n} = O(n^{1/3} \log^2 n \cdot \lambda)$$

and the amortized locality is $\frac{n \cdot O(1) + n^{2/3} \cdot O(n^{1/3}) + n^{1/3} \cdot O(n^{2/3})}{n} = O(1)$. Finally, the client must store Tab locally, that consists of $n^{2/3}$ entries of $\log n$ bits each and also needs to have $O(n^{1/3} \log^2 n \cdot \lambda)$ space locally for the oblivious sorting—see Theorem 4. □

Protocol $\langle \perp, Y \rangle \leftrightarrow$ ObliviousSorting$\langle(\pi, n, b), X\rangle$:

▷ Assume n and b are powers of 2 ▷ Also assume that $X[i]$ also stores the respective index i, so that comparisons using π are possible while elements are being moved around

1: **if** $n \leq b$ **then**
2: **Read-And-Decrypt** array X. Set Y to be the sorted version of X[a];
3: **else**
4: $\langle \perp, Y_1 \rangle \leftrightarrow$ ObliviousSorting$\langle(\pi, n/2, b), X[1, \ldots, n/2]\rangle$;
5: $\langle \perp, Y_2 \rangle \leftrightarrow$ ObliviousSorting$\langle(\pi, n/2, b), X[n/2+1, \ldots, n]\rangle$;
6: $\langle \perp, Y \rangle \leftrightarrow$ ObliviousMerge$\langle(\pi, n, b), (Y_1, Y_2)\rangle$;
7: **Encrypt-And-Write** array Y;
8: **return** $\langle \perp, Y \rangle$;

Protocol $\langle \perp, Y \rangle \leftrightarrow$ ObliviousMerge$\langle(\pi, n, b), (Y_1, Y_2)\rangle$: ▷ Y_1, Y_2 must be sorted

1: **if** $n \leq b$ **then**
2: **Read-And-Decrypt** array Y_1;
3: **Read-And-Decrypt** array Y_2;
4: Set Y to be the merged array of Y_1 and Y_2;
5: **else**
6: Let D be a $2 \times n/2$ matrix and Y be a length n array stored at the server;
7: $j = 0$;
8: **for** $i = 1, 2b+1, 4b+1, \ldots, n/2 - 2b + 1$ **do**
9: Initialize arrays D_1, D_2, D_3, D_4 of size b;
10: Store $Y_1[i], Y_1[i+2], \ldots, Y_1[i+2b-2]$ at the first available position of D_1;
11: Store $Y_1[i+1], Y_1[i+3], \ldots, Y_1[i+2b-1]$ at the first available position of D_3;
12: Store $Y_2[i], Y_2[i+2], \ldots, Y_2[i+2b-2]$ at the first available position of D_2;
13: Store $Y_2[i+1], Y_2[i+3], \ldots, Y_2[i+2b-1]$ at the first available position of D_4;
14: Store D_1 in D's row 1, from position $1 + j \cdot b$ onwards;
15: Store D_2 in D's row 1, from position $n/4 + 1 + j \cdot b$ onwards;
16: Store D_3 in D's row 2, from position $1 + j \cdot b$ onwards;
17: Store D_4 in D's row 2, from position $n/4 + 1 + j \cdot b$ onwards;
18: $j \leftarrow j + 1$;
19: $\langle \perp, D[1,:] \rangle \leftrightarrow$ ObliviousMerge$\langle(\pi, n/2, b), (D[1, 1:n/4], D[1, n/4+1:n/2])\rangle$;
20: $\langle \perp, D[2,:] \rangle \leftrightarrow$ ObliviousMerge$\langle(\pi, n/2, b), (D[2, 1:n/4], D[2, n/4+1:n/2])\rangle$;
21: Let $Z_1, \ldots, Z_{n/2b}$ be the $2 \times b$ submatrices that result from partitioning D horizontally;
22: **for** $i = 1$ **to** $n/2b - 1$ **do**
23: **Read-And-Decrypt** Z_i;
24: **Read-And-Decrypt** Z_{i+1};
25: Sort $Z_i \cup Z_{i+1}$ and let $y_1, \ldots, y_{2b}$ be the smallest resulting elements;
26: **Encrypt-And-Write** $[y_1, \ldots, y_b]$ starting at the first available position of Y;
27: **Encrypt-And-Write** $[y_{b+1}, \ldots, y_{2b}]$ starting at the first available position of Y;
28: Sort $Z_{n/2b}$ and let $y_1, \ldots, y_{2b}$ be the sorted sequence;
29: **Encrypt-And-Write** $[y_1, \ldots, y_b]$ starting at the first available position of Y;
30: **Encrypt-And-Write** $[y_{b+1}, \ldots, y_{2b}]$ starting at the first available position of Y;
31: **return** $\langle \perp, Y \rangle$;

[a] We use π to perform comparisons between two elements of X, i.e., $X[i]$ isLessThan $X[j]$ iff $p[i] < p[j]$.

Fig. 16. Data-oblivious and I/O efficient sorting by Goodrich and Mitchenmacher [20].