Single-Server Client Preprocessing PIR with Tight Space-Time Trade-off

Zhikun Wang Ling Ren
University of Illinois Urbana-Champaign

November 10, 2024

Abstract

This paper partly solves the open problem of tight trade-off of client storage and server time in the client preprocessing setting of private information retrieval (PIR). In the client preprocessing setting of PIR, the client is allowed to store some hints generated from the database in a preprocessing phase and use the hints to assist online queries. We construct a new single-server client preprocessing PIR scheme. For a database with n entries of size w, our protocol uses $S = O((n/T) \cdot (\log n + w))$ bits of client storage and T amortized server probes over n/T queries, where T is a tunable online time parameter. Our scheme matches (up to constant factors) a $ST = \Omega(nw)$ lower bound generalized from a recent work by Yeo (EUROCRYPT 2023) and a communication barrier generalized from Ishai, Shi, and Wichs (CRYPTO 2024).

From a technical standpoint, we present a novel organization of hints where each PIR query consumes a hint, and entries in the consumed hint are relocated to other hints. We then present a new data structure to track the hint relocations and use small-domain pseudorandom permutations to make the hint storage sublinear while maintaining efficient lookups in the hints.

1 Introduction

Private Information Retrieval (PIR) [CGKS95,CKGS98] studies the following problem: Consider a server that holds a database DB of n entries indexed by $0, 1, \ldots, n-1$, each of size w; A client communicates with the server to retrieve an entry, without revealing any information about the index of the entry to the server.

In the most standard model of PIR, the server stores the unmodified database and nothing else, and the client has a single entry to retrieve from the server. There has been a long line of work on constructing private information retrieval schemes in this standard model [CG97, KO97, CMS99, KO00, Cha04, GR05, OSI07], eventually leading to several recent schemes that boast reasonable practical efficiency [ACLS18, MCR21, MW22].

All PIR schemes in the standard model inevitably have linear server computation. Intuitively, a secure PIR scheme must ask the server to touch every single entry in the database; otherwise, the server learns that the untouched entries are not the client's desired entry. Beimel, Ishai and Malkin [BIM00] formally proved the lower bound that linear server computation is necessary if the server stores only the unmodified database and the client has no state.

Since then, several avenues have been explored in an attempt to circumvent the linear server computation lower bound. A recent and promising avenue is the paradigm of *client preprocessing* PIR, which will be the focus of our paper. We will briefly mention other avenues in Section 1.3.

The first client preprocessing PIR scheme is proposed by Patel, Persiano, and Yeo [PPY18] (under the name private stateful information retrieval), though their goal at the time was not to circumvent the linear server computation bound. In the client preprocessing PIR paradigm, the client privately retrieves and persistently stores some hints about the database, and later uses

these hints to speed up subsequent PIR queries. The server still stores the original database unmodified and nothing else.

The breakthrough work of Corrigan-Gibbs and Kogan [CGK20] give the first client proprocessing PIR scheme that has sublinear online computation. They also presented the first lower bound showing that $ST = \tilde{\Omega}(n)$ where S is the client storage in bits, and T is the online number of "probes" that the server performs on the database for a single query. Here, $\tilde{\Omega}(\cdot)$ hides $\text{poly}(\log n, \lambda)$ factors where λ is a statistical security parameter.

Since then, both the lower bound and the upper bound have been improved. Yeo [Yeo23] managed to remove the $\text{poly}(\log n, \lambda)$ factors from the lower bound and showed the elegant lower bound of $ST = \Omega(n)$ for 1-bit entries. The lower bound also holds when T is the amortized number of server probes over multiple adaptively chosen queries [CGHK22] (rather than for a single query). We also show (in Appendix B.1 of this paper) that Yeo's lower bound for 1-bit entries can be extended to a $ST = \Omega(nw)$ lower bound for w-bit entries.

On the upper bound side, the original scheme of Corrigan-Gibbs and Kogan [CGK20] and many follow-up works [CGHK22,LP23b,SACM21,KCG21,ZLTS23,LP23a] had large $\text{poly}(\log n, \lambda)$ factors in both the client storage and the amortized server probes. Recent practical schemes [LP23b,ZPZS24,RMI23,GZS24,HPPY24] have improved them to a single λ factor in client storage. In other words, state-of-the-art client preprocessing PIR schemes achieve $S = O(\lambda\sqrt{n}w)$ and $T = O(\sqrt{n})$. We remark that some existing schemes require two non-colluding servers [CGK20, LP23b] while others work for a single server [CGHK22, ZPZS24, RMI23]. In both settings, the state-of-the-art schemes achieve $S = O(\lambda\sqrt{n}w)$ and $T = O(\sqrt{n})$.

The state of affairs motivates the following natural and fundamental question:

Can we construct a client preprocessing PIR scheme whose client storage and amortized server probes match the lower bound by a constant factor?

1.1 Our Contributions

This paper partly answers the above question in the affirmative. We present a single-server client preprocessing PIR scheme whose client storage S and amortized server probe T satisfy $ST = O(nw)$ when the entry size $w = \Omega(\log n)$. The scheme can be parameterized to achieve any S and T on the trade-off curve.

Theorem 1.1 (Main Result). *Assuming one-way functions exist, there exists a single-server client preprocessing PIR scheme that for any database size n, entry size w, time parameter T, over at least $Q = n/T$ queries, has*

- *client storage of $O(Qw + Q\log n)$ bits*
- *amortized communication of $O(Tw + T\log n)$ bits*
- *amortized server computation of $O(T)$ accesses to entries of size w*
- *amortized client computation of $O(T)$ XORs of w-bit entries and $O(T)$ small-domain PRP calls*

We also note that our protocol meets a communication barrier [ISW24] for client preprocessing PIR schemes that use only symmetric-key cryptography and are *database-oblivious*, i.e., where the client requests to the server do not depend on the database contents. In particular, they showed that a database-oblivious PIR scheme that uses S bits of client storage and $n/3$ bits of communication over $3S$ queries implies an average-case hard promise problem in the complexity class Statistical Zero-Knowledge (SZK). A hard problem in SZK is not known to exist from one-way functions, making it closer to cryptomania than minicrypt [Imp95]. We show in Appendix B.2 that their result extends to w-bit entries, and that symmetric-key data-oblivious

PIR schemes using S bits of client storage must have either $\Omega(nw/S)$ worst case online communication or $\Omega(nw/S)$ amortized communication, or the barrier will be breached. Our scheme meets this barrier with up to constant factors when $w = \Omega(\log n)$.

While our scheme does not meet the server probe lower bound or the communication barrier for very small entry sizes (e.g., $w = 1$), we note that $w = \Omega(\log n)$ is the most natural setting as it takes $\log n$ bits to represent the index of a database with n entries. We also note that this is the setting for most practical applications of PIR.

In addition to matching the lower bound for online probes and the communication barrier, our scheme also has efficient server and client computation. The server performs no extra computation aside from retrieving the T entries requested by the client. The client computation is $O(T)$ small-domain PRP calls each of cost $\text{poly}(\log n, \lambda)$ and T XORs on w-bit entries. This means that when the entry size is greater than $\text{poly}(\log n, \lambda)$, the overall amortized computation (client and server combined) is dominated by retrieving and XORing entries. We further note that our scheme supports updating an entry in expected $O(1)$ PRP calls because each entry contributes to only one hint.

1.2 Technical Overview

The last extra λ factor in existing schemes comes from the fact that the Corrigan-Gibbs-Kogan blueprint uses hint sets that are independently and randomly sampled. To fulfill a PIR query, the client must be able to find a hint that *includes* the queried entry. Therefore, a factor of λ duplication is needed to keep the correctness failure probability exponentially small in λ, similar to the situation in the "coupon collector" problem.

A recent work by Lazzaretti and Papamanthou [LP24] introduced a new way to organize the hints. In their work, an imaginary hint table is constructed by arranging the database as a matrix and randomly permuting each row. The hints are then the parities of all the columns. This construction eliminates the duplication because every database entry is now covered by exactly one hint. However, their scheme has two major drawbacks: it requires two non-colluding servers and linear client storage.

The main contribution of our work is to present new ideas to resolve these two drawbacks: we will adapt the scheme of [LP24] to a single server and remove the linear client storage. A key innovation is a new relocation data structure that, when utilizing small-domain pseudorandom permutations (PRPs), avoids linear client storage. In the remainder of this subsection, we present a technical overview of our scheme. We will begin with a review of the [LP24] scheme, as our construction uses its hint construction as a starting point.

Review of [LP24]. Throughout the paper, we consider a database DB of n entries each of size w. For all index $i \in [n]$, let $\mathsf{DB}[i] \in \{0,1\}^w$ denote the i-th entry. All indices start from 0. For a given parameter T (that will eventually be the amortized server probes), we divide DB into T rows of size $m = n/T$ and define $\mathsf{DB}_j = \mathsf{DB}[j \cdot m : (j+1) \cdot m]$ to be the j-th row of the database.

The scheme of [LP24] is illustrated in Fig. 1. It uses an imaginary *hint table* that organizes the n database entries in T rows and $m = n/T$ columns. Each row j of the hint table is a random permutation of the corresponding database row DB_j. Each column of the table corresponds to one hint, for which the parity (XOR sum) of all entries in the column is computed and stored. We note again that this hint construction covers the entire database without duplication as each database entry is in exactly one column.

The client storage in their scheme has the following two parts:

1. The T permutations (one per row) $p_0, p_1, \ldots, p_{T-1}$. The database entry at row $j \in [T]$ and column $c \in [m]$ of the hint table is $\mathsf{DB}_j[p_j(c)]$.

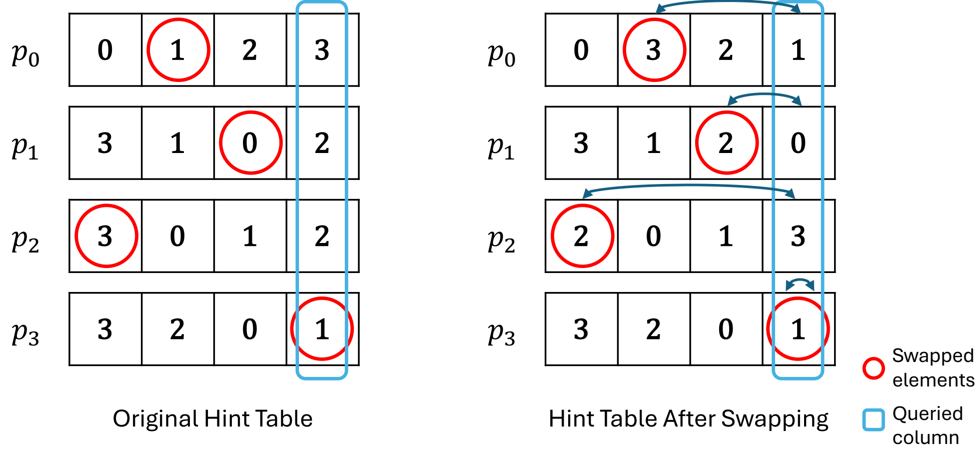

Figure 1: Hint table swapping in [LP24]

2. The m parities (one per column) $h_0, h_1, \ldots, h_{m-1}$. For each $c \in [m]$, $h_c = \bigoplus_{j=0}^{T-1} \mathsf{DB}_j[p_j(c)]$.

For the client to make a query for the i-th database entry $\mathsf{DB}[i]$, the client would first find the row j and column c of the hint table that $\mathsf{DB}[i]$ resides in. The client then sends the indices of the database entries in column c to the first server, with the exception that the desired index i is replaced by a random index in the same row. The server returns all the database entries requested by the client. The client now has all the database entries in column c except $\mathsf{DB}[i]$ and can calculate $\mathsf{DB}[i]$ by XORing the hint h_c with all the entries in column c (except $\mathsf{DB}[i]$).

The client also needs to refresh the hint table by swapping every entry in column c with a random entry in the same row. To do so, the client sends T uniformly random indices $(r_0, r_1, \ldots, r_{T-1}) \xleftarrow{\$} [m]^T$ to a second server to retrieve a random entry from each row. Now, the client can easily perform the swap and maintain the stored parities for all columns. For example, when swapping the c-th entry with the r_j-th entry in row j, the parities for columns c and r_j are updated to $h'_c \leftarrow h_c \oplus \mathsf{DB}_j[p_j(r_j)] \oplus \mathsf{DB}_j[p_j(c)]$ and $h'_{r_j} \leftarrow h_{r_j} \oplus \mathsf{DB}_j[p_j(r_j)] \oplus \mathsf{DB}_j[p_j(c)]$. After each entry in column c is swapped with a random entry, the client state is *reset*, and any information about the hint table indices revealed during the last PIR query is phased out. The refreshed hint table is identically distributed as it was prior to the query. Therefore, the request sent by the client to the first server is indistinguishable from a set of random indices. The second server also receives a set of random indices by construction.

From two non-colluding servers to a single server. Our first technical contribution is a new way to organize the hint table to make the scheme work with only a single server. The reason [LP24] requires two non-colluding servers is that in order to perform swaps in the hint table, the client needs to know the value of both entries. Yet, the first server should not learn the swapped entries, so the client has to retrieve them from a second server.

To solve this problem, we introduce empty cells in the hint table that we use to relocate the entries in the used column. Empty cells can be considered to contain 0^w, so there is no need to query a second server for their contents.

In more detail, the hint table is now a matrix with T rows where each row is of size $m' = 2m = 2n/T$. Out of the $2m$ positions in each row, a random subset of m positions contains the m database entries in that row, and the other m positions are empty. Each row j of the hint table is now represented by an array p_j of m' elements where each element is in $[m] \cup \{\perp\}$. Each value in $[m]$ appears once and only once in p_j. The cell at column $c \in [m']$ of row $j \in [T]$ in the hint table contains $\mathsf{DB}_j[p_j[c]]$ if $p_j(c) \in [m]$ and is empty if $p_j[c] = \perp$.

Suppose in a PIR query for database entry i, the column c is where $\mathsf{DB}[i]$ resides. After using

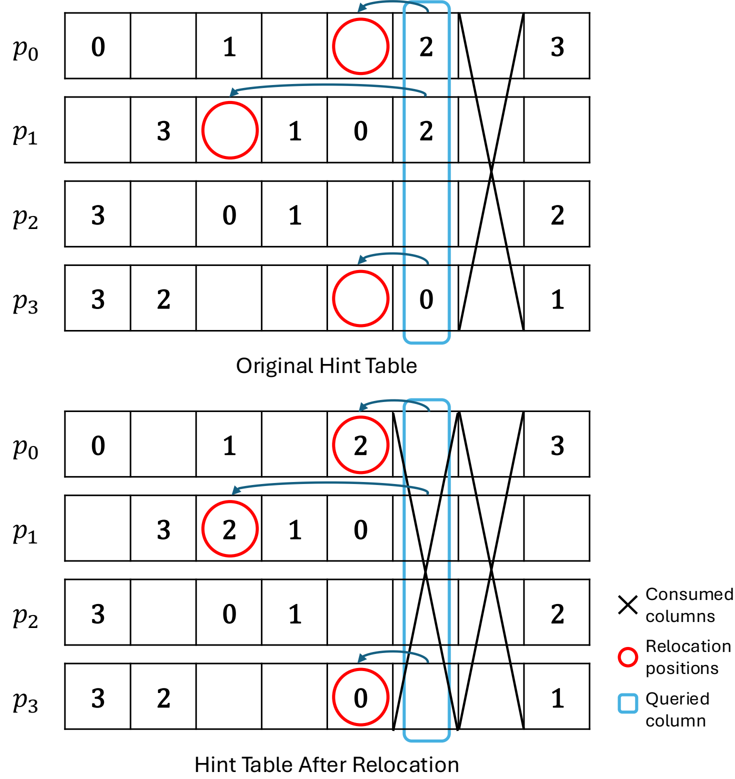

Figure 2: Entry relocation and hint updates in our PIR scheme.

h_c and column c to fulfill the query, the column c and the hint h_c are *consumed*. Thus, after the query, the client must relocate each entry in column c to a random empty column in the same row. For example, to move the entry in column c and row j to a random empty column r_j in that row, the client simply updates h_{r_j} to $h'_{r_j} \leftarrow h_{r_j} \oplus \mathsf{DB}_j[p_j(c)]$. This process is illustrated in Fig. 2.

An interesting difference between our scheme and all previous client preprocessing PIR schemes (including [LP24]) is that our scheme does *not* maintain the same distribution for the hints after each query. A consumed column will *not* be replenished as in all previous schemes, and the hint table now has one fewer column. Naturally, a hint table can only support a limited number of queries. (Previous schemes also exhibit this behavior but for slightly different reasons.) We will need a new hint table after every $m' - m$ queries, so the amortized communication and server computation are $O(n/m) = O(T)$.

We now present our single-server scheme with linear client storage in full.

1. *Offline hint construction phase.* The client computes the stored hints as follows:

   (a) The client initializes hints $(h_0, h_1, \ldots, h_{m'-1})$ to be an array of 0^w.

   (b) The client initializes T arrays $p_0, p_1, \ldots, p_{T-1}$ of m' elements in $[m] \cup \{\perp\}$ as follows: for each array, the client randomly sets m distinct positions in the array to $0, 1, \ldots, m-1$ and sets other positions to $\perp$.

   (c) The client streams the entire database one entry at a time. For e-th database entry $\mathsf{DB}_j[e]$ in row j, the client updates $h_c \leftarrow h_c \oplus \mathsf{DB}_j[e]$ where c is the column containing the entry (i.e., $p_j[c] = e$).
   After this step, the array h contains the parities (XOR sums) of all hint table columns.

   (d) The client initializes array C of consumed columns to be an empty array.

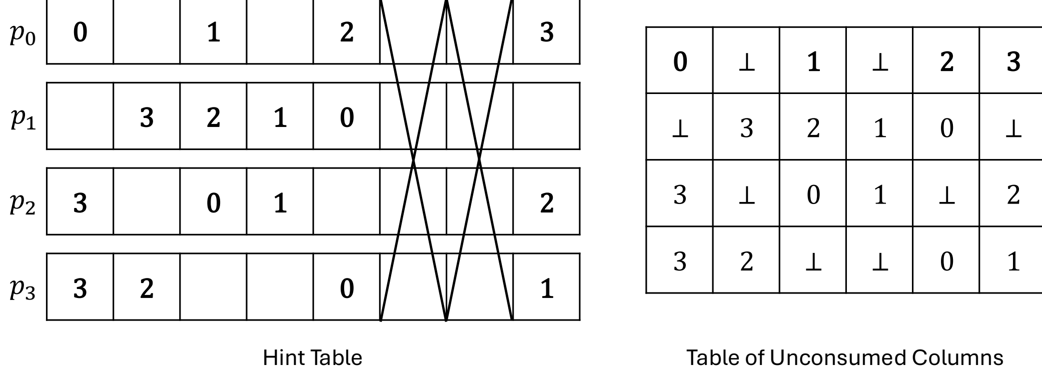

Figure 3: Table from unconsumed columns in the hint table.

2. *Online PIR query phase.* When the client wishes to retrieve i-th entry in the database, i.e., $\mathsf{DB}[i]$ for some $i \in [n]$:

   (a) The client calculates the row $j \leftarrow \lfloor i/m \rfloor$ and finds the column c in the hint table that contains the desired entry (i.e., $p_j[c] = i \mod m$).

   (b) The client computes all the indices in column c, $q \leftarrow (p_0[c], p_1[c], \ldots, p_{T-1}[c])$. The client then samples a random position $r \xleftarrow{\$} [m'] \setminus C$ and rewrites $q[j] \leftarrow p_j[r]$. The client sends q to the server.

   Essentially, the client wants to send the c-th column to the server but needs to redact the desired index i, so it replaces $p_j[c] = i \mod m$ with the value of a random unconsumed cell (possibly empty) in the same row as i.

   (c) The server returns all the database entries in the client's request q to the client.

   (d) The client computes the result as $\mathsf{DB}[i] = h_c \oplus \bigoplus_{j' \in [T] \setminus \{j\}} \mathsf{DB}_{j'}[q[j']]$.

   (e) The client appends column c to the array C of consumed columns.

   (f) For each row $j \in [T]$, if $p_j[c] \neq \perp$, the client samples a random unconsumed and empty cell r_j from that row, i.e., $r_j \xleftarrow{\$} [m'] \setminus C$ such that $p_j[r_j] = \perp$. The client then sets $p_j[r_j] \leftarrow p_j[c]$ and updates the corresponding parity $h[r_j] \leftarrow h[r_j] \oplus \mathsf{DB}_j[p_j[c]]$.

   Essentially, the client relocates each database entry in the consumed column to an unconsumed empty column in the same row.

We now give the intuition behind the security of our scheme. Consider the table formed by the unconsumed hint table columns as illustrated in Fig. 3. After k queries, the table will have T rows and $m' - k$ unconsumed columns.

Our proof depends on a key observation: after any number of queries, when conditioned on the adversary's view (what the client sent to the server during those queries), the table of unconsumed columns will have an equal probability of being any legitimate table where for each row, each element in $[m]$ appears exactly once. This can be proved inductively. At the beginning, each row is sampled according to the uniform distribution independently. Assuming that the table is uniformly random before a query, the column that is being consumed has an equal probability of being any column. For a fixed column c, when conditioned on the value of the column being consumed, every new table (with column c removed) corresponds to a pair of an original table and a vector of positions of all relocated entries. Therefore, each possible value of the resulting table has an equal likelihood, and the table will still be uniformly distributed. This means that, while the distribution of the table has changed, it is still a publicly known distribution independent of the PIR queries and the adversary's view thus far.

Given that the table of unconsumed columns is uniformly random, when conditioned on the desired entry i, every index in the client's request to the server will be $\perp$ with probability

$\frac{m'-k-m}{m'-k}$ and each value in $[m]$ with probability $\frac{1}{m'-k}$ as the hint table rows are independent and uniformly distributed, and that the j-th index have been replaced randomly. The security of our scheme follows.

Sublinear client storage using small-domain PRP. Our second technical contribution is a data structure DS that helps avoid the linear client storage in the above PIR scheme. Observe that the linear client storage comes from the T arrays $p_0, p_1, \dots, p_{T-1}$. Looking ahead, we will use T instances of the new data structure DS, one for each row of the hint table, to replace the T arrays.

Each instance of DS has m' positions, where each position corresponds to a hint table column. DS supports random accesses (Access), lookups by elements (Locate), and relocating elements ($\mathsf{Relocate}$). In more detail, $\mathsf{Access}(c)$ is the standard array access operation that takes a position c and returns the element at that position; $\mathsf{Locate}(e)$ is the inverse of Access: it takes an element e and returns the position c that e resides in; $\mathsf{Relocate}(c)$ relocates an element from a position c that is being consumed into a random unconsumed empty position.

We observe that each of the T arrays $p_0, p_1, \dots, p_{T-1}$ in our linear client storage PIR scheme is exactly implementing a copy of this data structure. In fact, it is not hard to rewrite the PIR scheme using T instances of DS instead of working directly with the T arrays. In the first step of the PIR online query algorithm, the client needs to find the column that contains the desired index i; this can be done by calling $\mathsf{Locate}(e)$ where $e = i \mod m$. Any occurrence of $p_j[c]$ can be replaced with $\mathsf{Access}(c)$ on j-th instance of DS. Finally, after consuming a column c, the client calls $\mathsf{Relocate}(c)$ for every instance of DS.

Now, removing the linear client storage from our previous PIR scheme boils down to an implementation of the DS data structure that is efficient in both time and space. We give such an implementation next.

Each DS instance uses a separate small-domain PRP. (The keys to the PRPs can be derived pseudorandomly via a PRF from a single master key.) The T DS instances will share a global array C that stores the consumed columns, along with a hash map to allow finding the index of any column in C (i.e., "invert" C) in constant time. This way, the total client storage needed to reconstruct the entire hint table is $O(\lambda + m \log m)$, avoiding the linear client storage.

We now focus on one instance of DS. DS has m' positions, where each position corresponds to a hint table column. The small-domain PRP P is used to determine the initial positions of elements as well as positions for subsequent relocations. Initially, the elements in $[m]$ are located at distinct positions $P(0), P(1), \dots, P(m-1)$. The rest of the positions $P(m), P(m+1), \dots, P(m'-1)$ are initially empty and are reserved for relocation operations. In particular, $P(m+t)$ is reserved for the t-th $\mathsf{Relocate}$ operation, regardless of which position is being consumed. In the event that the reserved position is already consumed by a previous $\mathsf{Relocate}$ operation, the algorithm would attempt to use the position reserved for that previous $\mathsf{Relocate}$ operation, and this process can repeat, similar to pointer chasing.

We now present the algorithm in detail as follows:

- $\mathsf{Access}(c) \to e$. To find the element at position c, calculate $P^{-1}(c)$. If $P^{-1}(c) = m + t$ for some $0 \le t < |C|$, then position c is the destination of a previous $\mathsf{Relocate}$ operation. Let $c \leftarrow C[t]$ be the relocated position and repeat the process. When the loop ends, if $P^{-1}(c) \ge m + |C|$, return $\perp$; otherwise, return $P^{-1}(c)$.

- $\mathsf{Locate}(e) \to c$. To find the position of element e, calculate $c = P(e)$. If the position has been consumed by t-th $\mathsf{Relocate}$ operation (i.e., $C[t] = c$), then set $c \leftarrow P(m+t)$, the reserved position for the t-th $\mathsf{Relocate}$. Repeat until $c \notin C$ and return c.

- $\mathsf{Relocate}(c)$. Simply append the consumed position c to array C.

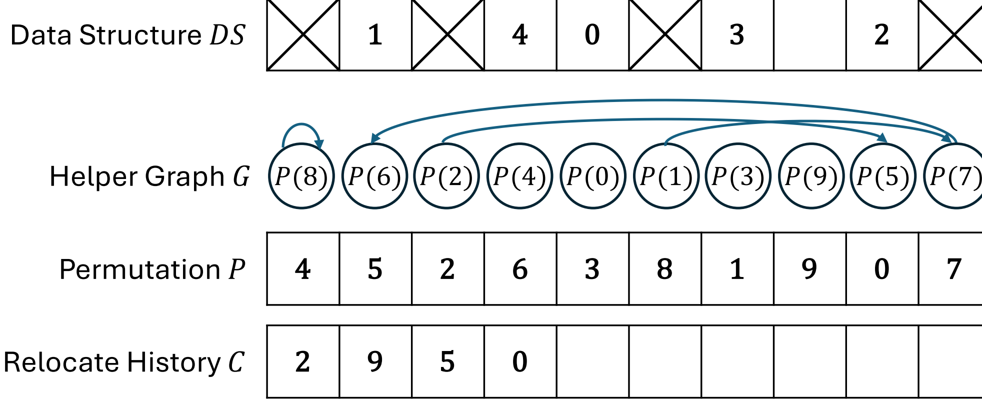

Figure 4: Illustration of the relocation data structure using PRP.

An intuitive way to think about the algorithm is to imagine a directed graph G with m' nodes, where node v corresponds to the v-th position in the data structure. This is illustrated in Fig. 4. Each consumed position $c \in C$ in G has a directed edge to its reserved position. More precisely, the t-th Relocate operation adds a directed edge from node $C[t]$ to node $P(m+t)$.

We can observe that every node in the graph has at most one in-edge and one out-edge. Thus, the graph consists of disjoint chains and cycles. (If a node has no in-edge and no out-edge, we consider the node to be both the start node and the end node of a chain that consists of only itself.) We can further observe that nodes $P(0), P(1), \ldots, P(m-1)$ have no in-edge and are start nodes of distinct chains and that all unconsumed nodes (i.e., not in C) have no out-edge and are end nodes of distinct chains. We can then verify that our algorithm is equivalent the the following:

- $\mathsf{Access}(c) \to e$. Given $c \notin C$, start from the end of chain c and follow the in-edge until reaching the start of the chain. Let that node be c', return $P^{-1}(c')$ if $P^{-1}(c') \in [m]$ and $\perp$ otherwise.
- $\mathsf{Locate}(e) \to c$. Start from node $P(e)$ and follow the out-edge until reaching the end of the chain. Return the corresponding position.
- $\mathsf{Relocate}(c)$. Add an edge from node c to node $P(m+t)$ in G.

We now give the intuition as to why the data structure is perfectly indistinguishable from the simple and ideal implementation using a linear-sized array. The initial positions of the m elements are clearly random. Because the nodes $P(0), P(1), \ldots, P(m-1)$ are start nodes of distinct chains and because the end node of each chain is an unconsumed position, we can verify that all m elements in $[m]$ are always in distinct unconsumed positions. From the one-to-one correspondence between the start and end of a chain, we can see that Locate is the inverse of Access as required. Finally, the most important step is to show that if a Relocate operation relocates an element, then the destination is uniformly random among the unconsumed empty positions.

Suppose the k-th Relocate operation consumes position c, which contains an element $e \in [m]$. The key observation is that the historical positions of all m elements up until this point (i.e., initially and after each of the previous $k-1$ Relocate operations) depend only on $P(0), P(1), \ldots, P(m+k-1)$. Therefore, conditioned on all the historical positions of all m elements so far, $P(m+k)$ has an equal probability among the $m'-m-k$ possible values in $[m'] \setminus \{P(0), P(1), \ldots, P(m+k-1)\}$. There are $m'-m-k$ unconsumed empty positions, each corresponding to the end of a chain whose starting node is not one of $P(0), P(1), \ldots, P(m-1)$. Therefore, there is a one-to-one correspondence between the $m'-m-k$ possible values of

$P(m+k)$ and the $m' - m - k$ unconsumed empty positions. Since c contains an element $e \in [m]$, the newly added edge from c to $P(m+k)$ will relocate e from c to the unconsumed empty position according to this correspondence.

1.3 Additional Related Work

Recent client preprocessing PIR schemes. Corrigan-Gibbs and Kogan [CGK20] give the first client preprocessing PIR scheme with amortized sublinear communication and server time. The initial scheme requires two servers and is later extended to the single-server setting [CGHK22].

Several subsequent works use privately puncturable and programmable pseudorandom functions to reduce the communication from $\Omega(\lambda\sqrt{N})$ to polylogarithmic in n, first in the two-server setting [SACM21] and later in the single-server setting [ZLTS23, LP23a].

A main source of the extra λ factors is that the blueprint of Corrigan-Gibbs and Kogan allows a small but non-negligible correctness failure, and thus requires parallel repetitions by a factor of λ. Recent works have found ways to avoid the correctness failure and remove the parallel repetitions, first in the two-server setting [KCG21, LP23b] and later in the single-server setting [ZPZS24, RMI23]. This leaves the λ factor in client storage due to hint duplication the only extra factor.

Recent works have also studied how to perform database updates efficiently in client preprocessing PIR [HPPY24, LP24] and how to support the full spectrum of trade-off between client storage and server time [HPPY24].

PIR with server preprocessing. In the same paper that established the $\Omega(N)$ server computation lower bound, Beimel, Ishai, and Malkin [BIM00] showed that the lower bound can be circumvented if the server preprocesses and encodes the database offline. This approach is also taken by a line of works known as doubly efficient PIR [CHR17, BIPW17, LMW23]. So far, these schemes have to significantly blow up server storage (superlinearly or by the number of clients) and/or require heavyweight theoretical tools (such as oblivious locally decodable codes or virtual black box obfuscation).

Batch PIR Batch PIR [IKOS04, ACLS18, MR23] is another way to circumvent the linear server computation lower bound through amortization. The difference between batch PIR and client preprocessing PIR is that batch PIR assumes the client has many queries to fetch in one go, while client preprocessing PIR allows the client to generate queries sequentially and possibly causally.

Small-domain PRP. Our construction uses a small-domain pseudorandom permutation (PRP) on the columns of the hint table. Constructing PRPs over small domains is a well-studied problem with a line of existing work [SS12, HMR12, RY13, MR14]. The construction of [MR14] achieved a permutation distribution that is within ε of uniform distribution using $\Theta(\log n - \log \varepsilon)$ rounds of true randomness. Their result immediately yields a small-domain PRP from a PRF.

2 Preliminaries

Notations. For a positive integer n, $[n]$ denotes the set $\{0, 1, \ldots, n-1\}$. 0^n denotes the all-zero binary string of length n. We ignore the integrality concerns and treat expressions like n/T and $\sqrt{n}$ as integers. For positive integers n, m where $m \le n$, $\mathbf{P}_n^m$ denotes $n!/(n-m)!$, the number of ways to choose m elements from n distinct elements with order. For a set S, we use $x \xleftarrow{\$} S$ to denote sampling x independently and uniformly at random from S. For a distribution $\mathcal{D}$ over a set S, $x \xleftarrow{\$} \mathcal{D}$ denotes sampling x from S according to $\mathcal{D}$.

We use the RAM model of computation with a word size of logarithm in a database of size n and linear in the security parameter λ. We measure our server and client computation by XORs of size w entries and PRP calls, which makes our result relatively independent of the computational model. We note that for larger (e.g., $\text{poly}(\log n, \lambda)$) entry size, the cost of manipulating entries of size w will dominate the computation cost.

2.1 Client Preprocessing Private Information Retrieval

In this section we provide the formal definition of client preprocessing PIR.

Definition 2.1 (Client preprocessing PIR)**.** A single-server client preprocessing PIR scheme for adaptive queries is a tuple of polynomial-time algorithms:

- $\mathsf{KeyGen}(1^\lambda) \to \mathsf{ck}$, a randomized algorithm that takes in a security parameter λ. The algorithm returns a client key ck.
- $\mathsf{HintConstruct}(\mathsf{ck}, \mathsf{DB}) \to h$, a deterministic algorithm that takes in client key ck and a database DB. The algorithm returns a hint state h.
- $\mathsf{Query}(\mathsf{ck}, i) \to (\mathsf{ck}', \mathsf{st}, q)$, a randomized algorithm at the client that takes in a client key ck and an index $i \in [n]$ for the desired database entry. The algorithm returns an updated client key ck', a client state query st, and a request q.
- $\mathsf{Answer}(q, \mathsf{DB}) \to a$, a deterministic algorithm on the server that takes in a request q and a database DB. The algorithm returns an answer a.
- $\mathsf{Reconstruct}(\mathsf{ck}, \mathsf{st}, h, a) \to (h', \mathsf{DB}[i])$, a deterministic algorithm on the client that takes in a client key ck, a client query state st, a hint state h and a server answer a. The algorithm returns an updated hint state h' and the answer $\mathsf{DB}[i]$.

Correctness. We require all recovered entries to be correct for all honest execution of the protocol. A PIR scheme is correct for Q queries if for all λ, w, n, database $\mathsf{DB} \in (\{0,1\}^w)^n$, and any sequence of queries $(i_1, i_2, \ldots, i_Q) \in [n]^Q$, the following experiment outputs 1 with probability 1:

- $\mathsf{ck} \leftarrow \mathsf{KeyGen}(1^\lambda)$
- $h \leftarrow \mathsf{HintConstruct}(\mathsf{ck}, \mathsf{DB})$
- For $t = 1, 2, \ldots, Q$:
  - $(\mathsf{ck}, \mathsf{st}, q) \leftarrow \mathsf{Query}(\mathsf{ck}, i_t)$
  - $a \leftarrow \mathsf{Answer}(q, \mathsf{DB})$
  - $(h, v_t) \leftarrow \mathsf{Reconstruct}(\mathsf{ck}, \mathsf{st}, h, a)$
- Output 1 if $v_t = \mathsf{DB}[i_t]$ for all t and 0 otherwise.

Security. We require that the server (adversary $\mathcal{A}$) is unable to learn anything about the client's queries even when the queries are chosen adaptively by the adversary. Formally, consider the following experiment:

- $b \xleftarrow{\$} \{0,1\}$
- $\mathsf{ck} \leftarrow \mathsf{KeyGen}(1^\lambda)$

- $\mathsf{st} \leftarrow \mathcal{A}(1^\lambda)$
- For $t = 1, 2, \ldots, Q$:
  - $(\mathsf{st}, i_0, i_1) \leftarrow \mathcal{A}(\mathsf{st})$
  - $(\mathsf{ck}, _, q) \leftarrow \mathsf{Query}(\mathsf{ck}, i_b)$
  - $\mathsf{st} \leftarrow \mathcal{A}(\mathsf{st}, q)$
- $b' \leftarrow \mathcal{A}(\mathsf{st})$
- Output 1 if $b = b'$ and 0 otherwise.

We define $W_{\mathcal{A},\lambda,Q,n}$ to be the event that the experiment outputs 1. We say that a client preprocessing PIR scheme is secure for Q queries if for all probabilistic and polynomial time adversaries $\mathcal{A}$, all polynomially bounded functions $n(\lambda)$, and all $\lambda \in \mathbb{N}$,

$$\Pr[W_{\mathcal{A},\lambda,Q,n}] \leq \frac{1}{2} + \mathrm{negl}(\lambda).$$

2.2 Small-domain Pseudorandom Permutation (PRP)

In this section we provide formal definitions for pseudorandom functions (PRF) and small-domain pseudorandom permutations (PRP). We also present the result of [MR14] which constructs a small-domain PRP from a PRF.

We use the standard definition of a pseudorandom function family [GGM86] where the polynomial time function is indistinguishable from a random function when the key is sampled uniformly at random.

Definition 2.2 ([GGM86])**.** A family of functions $\{f_k\}_{k\in\{0,1\}^*}$ where $f_k : \{0,1\}^{|k|} \to \{0,1\}^{|k|}$ is a pseudorandom function family if there is a polynomial time algorithm that computes $f_k(x)$ given $k \in \{0,1\}^*, x \in \{0,1\}^{|k|}$ and if for all polynomial time adversaries $\mathcal{A}$ there is a negligible function negl such that

$$\left| \Pr_{k \xleftarrow{\$} \{0,1\}^n} [\mathcal{A}^{f_k}(1^n) = 1] - \Pr_{g \xleftarrow{\$} \mathcal{F}_n} [\mathcal{A}^{g}(1^n) = 1] \right| \leq \mathrm{negl}(n)$$

for every n, where $\mathcal{F}_n$ is the set of all functions from $\{0,1\}^n$ to $\{0,1\}^n$.

We now recall the definition for a small-domain pseudorandom permutation.

Definition 2.3. A small-domain pseudorandom permutation over domain $[N]$ with key length λ is a tuple of polynomial-time algorithms $P : \{0,1\}^\lambda \times [N] \to [N]$ and $P^{-1} : \{0,1\}^\lambda \times [N] \to [N]$ such that for any $k \in \{0,1\}^\lambda$, P_k is a bijection from $[N]$ to $[N]$, P_k^{-1} is its inverse, and for all polynomial time adversaries $\mathcal{A}$ there is a negligible function negl such that

$$\left| \Pr_{k \xleftarrow{\$} \{0,1\}^\lambda} [\mathcal{A}^{P_k, P_k^{-1}}(1^\lambda) = 1] - \Pr_{g \xleftarrow{\$} \pi(N)} [\mathcal{A}^{g, g^{-1}}(1^\lambda) = 1] \right| \leq \mathrm{negl}(\lambda)$$

where $\pi(N)$ is the set of all permutations over $[N]$.

We use the construction of a shuffle algorithm by [MR14].

Theorem 2.4 ([MR14])**.** *For any $N \geq 1$ and $\varepsilon \in (0,1)$, there is an algorithm that evaluates a permutation and its inverse for domain $[N]$ in expected $O(\log N - \log \varepsilon)$ time given access to a uniformly random function (as an $O(1)$ time oracle). The distribution of the sampled permutation differs from the uniform distribution over all permutations with domain N by at most ε.*

Their result can be interpreted as a construction for a small-domain PRP from a secure PRF.

Corollary 2.5. *Assuming a computationally secure PRF, there exists a computationally secure small-domain PRP that, for domain* $[N]$ *and key length* λ*, uses expected* $\Theta(\log N + \text{poly} \log \lambda)$ *calls to the PRF.*

3 The relocation data structure

In this section, we construct a data structure that stores an array of size m' where each element is in $[m] \cup \{\perp\}$ and every element in $[m]$ appears once and only once. The data structure supports the regular array access by indices and lookups by element. In addition, the data structure supports *consuming* a position and relocating the element (if there is one) at that position to another random empty position. No element can reside at the consumed position in the future. The data structure will be later used to store hint table rows in our client preprocessing PIR protocol where each position in the array corresponds to a column.

In this section, we will construct the data structure using uniformly random permutations over domain $[m']$ and demonstrate perfect security. In Section 4, we will use small-domain PRP to obtain a computationally secure PIR scheme without linear storage.

3.1 Definition

A relocation data structure DS is parameterized by security parameter λ, size parameters $m, m' \in \mathbb{N}$ $(m' > m)$, and has internal state st. The data structure supports operations Access, Locate and $\mathsf{Relocate}$. We say each $\mathsf{Relocate}(c)$ operation *consumes* the position $c \in [m']$. Throughout this section, we define array C to contain all positions consumed by previous calls to $\mathsf{Relocate}$ in the order they are consumed. The operations are defined as follows:

- $\mathsf{DS.Access}(c) \to e$. Given a position $c \in [m'] \setminus C$, return the element at position c. The element is either in $[m]$ or $\perp$.

  This operation does not modify the state st of DS.

- $\mathsf{DS.Locate}(e) \to c$. Given an element $e \in [m]$, return the position that the element resides in. This can be viewed as the inverse of the Access operation.

  This operation does not modify the state st of DS.

- $\mathsf{DS.Relocate}(c)$. Given a position $c \in [m'] \setminus C$, consume the position so it cannot hold any element in the future. If there is an element at position c, relocate it to a random unconsumed and empty position.

  This operation modifies the state st of DS and appends c to C.

The data structure is initialized by randomized algorithm $\mathsf{DS.Init}(m, m')$ which takes size parameters m, m' as input and sets the initial state st of DS using randomness. The $\mathsf{DS.Relocate}$ operation can be called at most $m' - m$ times, and each time is called on a distinct position.

The relocation data structure must satisfy the following correctness and security properties.

Correctness. A relocation data structure is correct if the following properties hold:

- Initially and after each $\mathsf{Relocate}$ operation, each element in $[m]$ appears once and only once in the output of Access. Formally, for every element $e \in [m]$, there exists a unique position $c \in [m'] \setminus C$ such that $\mathsf{DS.Access}(c) = e$.

- Initially and after each $\mathsf{Relocate}$ operation, Access and Locate are inverses of each other. Formally, for any element $e \in [m]$ we have $\mathsf{DS.Locate}(e) = c$ if and only if $\mathsf{DS.Access}(c) = e$.

- After each Relocate operation, if there is an element $e \neq \bot$ in the position being consumed, then e is relocated to an empty, unconsumed position. All other elements remain in their positions. Formally, let DS' be the resulting data structure after calling $\mathsf{DS}.\mathsf{Relocate}(c)$ for some $c \in [m'] \setminus C$, and let $e = \mathsf{DS}.\mathsf{Access}(c)$. We require (i) $\mathsf{DS}'.\mathsf{Locate}(e') = \mathsf{DS}.\mathsf{Locate}(e')$ for all $e' \in [m] \setminus \{e\}$, and (ii) if $e \neq \bot$, $c' \notin C$ and $\mathsf{DS}.\mathsf{Access}(c') = \bot$ where $c' = \mathsf{DS}'.\mathsf{Locate}(e)$.

Perfect security. A relocation data structure is perfectly secure if the following two properties hold for any sequence of relocation positions:

- Initially, the positions of all m elements are distinct and uniformly random in $[m']$.
- After each Relocate operation that relocates an element in $[m]$, the element is relocated to a uniformly random empty position in $[m'] \setminus C$ when conditioned on the initial element positions and previous relocation operations and their destinations.

Formally, our definition of perfect security requires the experiments Experiment 3.1 and Experiment 3.2 to be identically distributed for all parameters.

Experiment 3.1. Parameterized by $m, m', 0 \leq Q \leq m' - m$, and $c_1, c_2, \ldots, c_Q \in [m']$:

- Let DS be initialized with randomized algorithm $\mathsf{DS}.\mathsf{Init}(m, m')$.
- Output $(\mathsf{DS}.\mathsf{Access}(0), \mathsf{DS}.\mathsf{Access}(1), \ldots, \mathsf{DS}.\mathsf{Access}(m-1))$.
- For $i = 1, 2, \ldots, Q$:
  - Let $e_i = \mathsf{DS}.\mathsf{Access}(c_i)$.
  - Call $\mathsf{DS}.\mathsf{Relocate}(c_i)$.
  - If $e_i \neq \bot$, output $\mathsf{DS}.\mathsf{Locate}(e_i)$. Otherwise, output $\bot$.

Experiment 3.2. Parameterized by $m, m', 0 \leq Q \leq m' - m$, and $c_1, c_2, \ldots, c_Q \in [m']$:

- Initialize A to be an array of size m' where m random positions contain $0, 1, \ldots, m-1$ and other positions contain $\bot$.
- Output the position of each element in $[m]$ in A.
- For $i = 1, 2, \ldots, Q$:
  - Find a random position $c' \in [m']$ satisfying $A[c'] = \bot$.
  - Update $A[c'] \leftarrow A[c_i]$ and $A[c_i] \leftarrow \varnothing$.
  - If $A[c'] \neq \bot$, output c'. Otherwise, output $\bot$.

3.2 Construction

In this section, we present our construction for a relocation data structure in Construction 3.5. We will prove the following theorem:

Theorem 3.3. *Construction 3.5 is a relocation data structure that is correct and secure. Additionally, the data structure satisfies the following efficiency properties:*

- Access *uses $O(1)$ time in expectation for a random input $c \in [m'] \setminus C$.*
- Locate *uses $O(1)$ time in expectation for a random input $e \in [m]$.*

Construction 3.4 (Relocate history)**.** Data structure Hist stores an array C of previously consumed positions and a hash map M that maps consumed position to their index in C. Hist supports the following operations:

- Hist.Init(). Initializes C to be an empty array and M to be an empty hash map.
- Hist.Append(c). Append c to C. Set $M[c] \leftarrow |C| - 1$.
- $\mathsf{Hist}[t] \to c$. Return $C[t]$ if $t < |C|$. Otherwise, return $\perp$.
- $\mathsf{Hist}^{-1}[c] \to t$. Return $M[c]$ if $c \in M$. Otherwise, return $\perp$.

Construction 3.5 (Relocation data structure)**.** Parameterized by size parameters m, m'. The internal state st of DS contains a permutation P and relocation history Hist.

DS.Init(ck, m).

- Initialize P to be a uniformly random permutation over $[m']$.
- Execute Hist.Init().

DS.Access(c) $\to e$.

- While $\mathsf{Hist}[P^{-1}(c) - m] \neq \perp$: *While c is the destination of a relocation*
  - Update $c \leftarrow \mathsf{Hist}[P^{-1}(c) - m]$. *Update c to the source of relocation*
- If $P^{-1}(c) < m$, return $P^{-1}(c)$. Otherwise, return $\perp$.

DS.Locate(e) $\to c$.

- Let $c = P(e)$. *Let c be the initial position of e*
- While $\mathsf{Hist}^{-1}[c] \neq \perp$: *While c is a consumed position*
  - Update $c \leftarrow P(m + \mathsf{Hist}^{-1}[c])$. *Update c to the relocation destination*
- Return c.

DS.Relocate(c).

- Call Hist.Append(c) to update Hist.

– $\mathsf{Relocate}(c)$ *and then* $\mathsf{Locate}(e)$ *(if relocated element* $e \neq \bot$ *exists) uses* $O(1)$ *expected time over position* $c \in [m'] \setminus C$ *and* $O(1)$ *amortized time.*

We use the simple data structure Hist in Construction 3.4 to store the relocation history C in a form that supports both indexing and lookup by value in expected constant time with the help of a hash map. For $m' = O(m)$, the array and hash map will take $O(m \log m)$ bits in total.

The construction of the relocation data structure in Construction 3.5 is based on a directed helper graph G where each node corresponds to a position in the array. The directed edges in the graph is determined by the array of consumed positions C and a uniformly random permutation $P : [m'] \to [m']$.

Definition 3.6 (The helper graph G)**.** The helper graph G is a directed graph with m' nodes, where the v-th node corresponds to the v-th position in the array. There is an edge going out of each position in C: for the t-th position in C, there is a directed edge from $C[t]$ to $P(m+t)$.

We now present some properties of the graph G.

Fact 3.7. *All nodes in* $\{P(0), P(1), \ldots, P(m-1)\}$ *have no in-edges.*

Fact 3.8. *All nodes in* $[m'] \setminus C$ *have no out-edges.*

Fact 3.7 is due to the fact that only nodes $P(m+t)$ where $t \geq 0$ have in-edges. Fact 3.8 is due to the fact that only nodes in C have out-edges.

Lemma 3.9. *Each node in* G *has at most one in-edge and one out-edge.*

Proof. To see that each node has at most one out-edge, note again that an out-edge is created only when a position is consumed and that each position is consumed at most once. □

Corollary 3.10. *The graph* G *consists of disjoint chains and cycles of nodes. Furthermore, all nodes in a cycle are in* C*.*

We now provide an alternative view for Locate and Access as walks on graph G to show that the input/output relations of Locate and Access always correspond to the start/end nodes of a chain in G.

We shall first examine Locate. From Construction 3.5, we can see that $\mathsf{Locate}(e)$ will start with $c = P(e)$, and in each iteration, c will be updated to $P(m+t)$ if c is the t-th position added to C. The procedure will terminate when $c \notin C$. This is equivalent to starting with $c = P(e)$ and following the out-edge until arriving at a node with no out-edge. From Fact 3.7 we know that $P(e)$ has no in-edge and is the start of a chain. Therefore we have the following:

Fact 3.11. *The function* $\mathsf{Locate}(e)$ *is equivalent to the following: start from the start node* $P(e)$ *of a chain, traverse the chain, and return the position* c *corresponding to the end of the chain.*

Similarly, from examining the process of $\mathsf{Access}(c)$, we have the following:

Fact 3.12. *The function* $\mathsf{Access}(c)$ *is equivalent to the following: start from node* c*, follow the in-edge of the node until there is no in-edge, and return* $P^{-1}(c)$ *if* $P^{-1}(c) < m$ *and* $\bot$ *otherwise.*

3.2.1 Correctness

We will now show that the construction is correct.

Lemma 3.13 (Elements appear once and only once)**.** *Initially and after each* $\mathsf{Relocate}$ *operation, for every element* $e \in [m]$*, there exists a unique position* $c \in [m'] \setminus C$ *such that* $\mathsf{DS.Access}(c) = e$*.*

Proof. From Corollary 3.10 and Fact 3.7, we know that nodes $P(0), \ldots, P(m-1)$ are start nodes of disjoint chains. For each element $e \in [m]$, $\mathsf{Access}(c)$ returns e only if the node c is on the chain starting from $P(e)$. As all nodes with out-edges are in C, the only possible input for Access that returns e is the end of the chain starting from $P(e)$. Therefore the lemma holds. □

Lemma 3.14 (Access and Locate are inverses)**.** *Initially and after each* Relocate *operation, for any element* $e \in [m]$*,* $\mathsf{DS.Locate}(e) = c$ *if and only if* $\mathsf{DS.Access}(c) = e$*.*

Proof. From Corollary 3.10, we know that the graph G consists of disjoint chains and cycles of nodes. For an element $e \in [m]$, from Fact 3.11 we know that $\mathsf{Locate}(e) = c$ implies that c is the end of the chain starting from $P(e)$, which implies $\mathsf{Access}(c) = e$ by Fact 3.12. For the other direction, $\mathsf{Access}(c) = e$ implies that $P(e)$ is the start of the chain where the end is node c, which implies $\mathsf{Locate}(e) = c$. □

Lemma 3.15 (Correctness of Relocate)**.** *After each* Relocate *operation, if there is an element* $e \neq \perp$ *in the position being consumed, then* e *is relocated to an empty, unconsumed position. All the other elements remain in their positions (see Section 3.1).*

Proof. For a data structure DS, let DS′ be the resulting data structure after calling $\mathsf{DS.Relocate}(c)$ for some $c \in [m'] \setminus C$. Consider the helper graph G. During a Relocate operation, the only change to the graph is to add an edge from node c to node $P(m + |C|)$.

We first show that if there is an element $e = \mathsf{DS.Access}(c) \neq \perp$ at position c, it will be relocated to an empty, unconsumed position. According to Fact 3.11, c will be the end of a chain where the start is node $P(e)$. Let $c' = \mathsf{DS'.Locate}(e)$ be the new position for e, we have c' is unconsumed ($c' \notin C$) as c' is the end of a chain. To see that c' is empty in DS (i.e., $\mathsf{DS.Access}(c') = \perp$), note that before the edge $(c, P(m + |C|))$ is added, the node $P(m + |C|)$ starts a chain that ends at c'. As $m + |C| \geq m$, we have $\mathsf{DS.Access}(c') = \perp$.

To show that no other element changes its position, we observe that since the new edge $(c, P(m + |C|))$ is part of the chain starting from $P(e)$ and the fact that chains are disjoint, no other chains starting from a node in $\{P(e') : e' \in [m] \setminus \{e\}\}$ will be affected. From Fact 3.11, this directly implies that $\mathsf{DS'.Locate}(e') = \mathsf{DS.Locate}(e')$ for all $e' \in [m] \setminus \{e\}$. □

3.2.2 Security

We will now show that the construction is secure.

Lemma 3.16. *For all* $t = 0, 1, \ldots, m$*, adversary* $\mathcal{A}$*'s views on the initial output and the output of first* t *relocation operations for Experiments 3.1 and 3.2 are identically distributed.*

Proof. The proof will use induction for t.

Base case. For $t = 0$, we show that the initial positions of all m elements are identically distributed. From definition, we know that for Experiment 3.2 the initial element positions are uniformly at random among all $\mathbf{P}^m_{m'}$ possibilities. For 3.1, we see that the initial positions are $(P(0), P(1), \ldots, P(m-1))$, which have the same distribution since P is a random permutation.

Inductive case. Assuming that the outputs up to the $t-1$-th relocation operation are identically distributed. We now show that the output of the t-th operation is also identically distributed when conditioned on all t inputs and previous outputs. For the t-th position c_t, no element resides in position c_t, then the output of both Experiments 3.1 and 3.2 will be $\perp$.

We now consider the case where there is an element at position c_t. By definition, Experiment 3.2 will output a uniformly random empty, unconsumed position. To see that Experiment 3.1 will output a uniformly random position, we first see that in the first t relocations, only $P(0), P(1), \ldots, P(m+t-1)$ are accessed in the experiment. Therefore, conditioned on the previous view, $P(m+t)$ have equal probability of being any value in $[m'] \setminus \{P(0), P(1), \ldots, P(m+t-1)\}$.

Consider the helper graph G. All empty positions are not consumed and are thus nodes without an in-edge. These nodes are end nodes of disjoint chains that do not start with a node in $\{P(0), \ldots, P(m-1)\}$. As t positions are consumed, there are $m'-m-t$ empty positions in the array. All $m'-m-t$ possible values for $P(m+t)$ correspond to nodes without an in-edge and are start nodes of disjoint chains where the end node corresponds to empty positions. Therefore, there is a bijection from possible values of $P(m+t)$ to empty positions in the array. Since P is a uniformly random permutation, the outputs of Experiment 3.1 and 3.2 are identically distributed. □

3.2.3 Efficiency

The main idea behind the efficiency analysis is as follows: from Fact 3.11 and Fact 3.12 we know that Locate and Access are equivalent to traversing on a chain. As all chains are distinct and there are m' nodes in total, the total cost of Locate and Access for all possible calls is $O(m')$ and the expected cost for a random call will be $O(1)$. We will defer the proof of efficiency stated in Theorem 3.3 to Appendix A.1.

Theorem 3.3 now follows directly from the efficiency, correctness and security of the construction.

4 Construction of the PIR scheme

In this section, we present our construction of the client preprocessing PIR scheme using the relocation data structure DS constructed in Section 3. We will prove the following theorem stating that Construction 4.2 satisfies all the desired efficiencies.

Theorem 4.1. *Assuming one-way functions exist, Construction 4.2 is a correct and secure single-server client preprocessing PIR scheme that for any database size n, entry size w, online time parameter T, over at least $Q = n/T$ queries, has*

- *client storage of $O(Qw + Q \log n)$ bits*
- *amortized communication of $O(Tw + T \log n)$ bits*
- *amortized server computation of $O(T)$ accesses to entries of size w*
- *amortized client computation of $O(T)$ XORs of size w elements and $O(T)$ small-domain PRP calls* [1]

We divide the database into T rows of m elements each, and define DB_j to be $\mathsf{DB}[j \cdot m : (j+1) \cdot m]$ for $j \in [T]$. The protocol stores m' hints organized by an imaginary hint table, where m' is defined to be $2m$. The hint table is a matrix of T rows and m' columns, where each row j is managed by a relocation data structure DS_j and contains the elements in DB_j. The element at the c-th column on the j-th row of the hint table is defined to be $\mathsf{DB}_j[\mathsf{DS}_j.\mathsf{Access}(c)]$ if $\mathsf{DS}_j.\mathsf{Access}(c) \neq \perp$; otherwise, no element resides in that position. For convenience in computing parities, we define $\mathsf{DB}_j[\perp] = 0$.

Each of the m' hints is the parity of elements in a column of the hint table. Concretely, $h_c = \bigoplus_{j \in [T]} \mathsf{DB}_j[\mathsf{DS}_j.\mathsf{Access}(c)]$ for $c \in [m']$ is the c-th hint. For every PIR query for desired entry i, the client finds the hint table row $j = \lfloor i/m \rfloor$ and column $c = \mathsf{DS}_j.\mathsf{Locate}(i \mod m)$ that contains the desired entry.

The client then constructs a request from the c-th column by replacing the j-th element with a random element in row j. The client retrieves these elements from the server and obtains

[1] We note that our stated client computation requires the PIR queries to be independent of the randomness of the scheme. This assumption is not required for correctness, security, or the other efficiency metrics.

Construction 4.2 (Single-server client preprocessing PIR using PRP)**.** Parameterized by database size n, time parameter T, we define $m = n/T$, $m' = 2m$. The client stores a key $\widehat{\mathsf{ck}}$, the history of consumed columns Hist, and hints $h = (h_0, h_1, \ldots, h_{m'-1})$. For each row j, an instance of DS_j in Construction 3.5 is instantiated with pseudorandom permutation $\mathsf{PRP}(\widehat{\mathsf{ck}}_j, \cdot)$ where $\widehat{\mathsf{ck}}_j = \mathsf{PRF}(\widehat{\mathsf{ck}}, j)$ and the globally shared Hist. We implicitly pass $\mathsf{st} = (i, c, j^*, q)$ from Query to $\mathsf{Reconstruct}$ and implicitly parse $\mathsf{ck} = (\widehat{\mathsf{ck}}, \mathsf{Hist})$.

$\mathsf{KeyGen}(1^\lambda) \to \mathsf{ck}$.

- Initialize $\widehat{\mathsf{ck}} \xleftarrow{\$} \{0,1\}^\lambda$, call $\mathsf{Hist.Init}()$.
- Return $\mathsf{ck} = (\widehat{\mathsf{ck}}, \mathsf{Hist})$.

$\mathsf{HintConstruct}(\mathsf{ck}, \mathsf{DB}) \to h$.

- Initialize XOR sums $h = (h_0, \ldots, h_{m'-1})$ as an array of 0^w.
- The client streams the database by each element. For e-th element $\mathsf{DB}_j[e]$ in row j:
  - Update $h_c = h_c \oplus \mathsf{DB}_j[e]$, where $c = \mathsf{DS}_j.\mathsf{Locate}(e)$.
- Return h.

$\mathsf{Query}(\mathsf{ck}, i) \to (\mathsf{ck}', \mathsf{st}, q)$.

- Let $j^* \leftarrow \lfloor i/m \rfloor$ and $c \leftarrow \mathsf{DS}_{j^*}.\mathsf{Locate}(i \mod m)$.
- Let $q \leftarrow (\mathsf{DS}_0.\mathsf{Access}(c), \mathsf{DS}_1.\mathsf{Access}(c), \ldots, \mathsf{DS}_{T-1}.\mathsf{Access}(c))$.
- Let $r^* \xleftarrow{\$} [m'] \setminus C$. Rewrite $q[j^*] \leftarrow \mathsf{DS}_{j^*}.\mathsf{Access}(r^*)$.
- Call $\mathsf{Hist.Append}(c)$.
- Return $\mathsf{ck}' = (\widehat{\mathsf{ck}}, \mathsf{Hist})$, q.

$\mathsf{Answer}(q, \mathsf{DB}) \to a$.

- Parse $(i_0, i_1, \ldots, i_{T-1}) \leftarrow q$.
- Return $a = (\mathsf{DB}_0[i_0], \mathsf{DB}_1[i_1], \ldots, \mathsf{DB}_{T-1}[i_{T-1}])$.

$\mathsf{Reconstruct}(\mathsf{ck}, \mathsf{st}, h, a) \to (h', \mathsf{DB}[i])$.

- Let $\mathsf{DB}[i] \leftarrow h[c] \oplus \bigoplus_{j \in [T], j \neq j^*} a[j]$.
- Let $a[j^*] \leftarrow \mathsf{DB}[i]$. Let $q[j^*] \leftarrow i \mod m$.
- For each row $j \in [T]$ such that $q[j] \neq \bot$:
  - Let $c = \mathsf{DS}_j.\mathsf{Locate}(q[j])$, update $h_c = h_c \oplus a[j]$.
- Return $(h' = h, \mathsf{DB}[i])$.

the desired entry by XORing all other elements in the column with the hint. Finally, the client updates the hint table by calling $\mathsf{Relocate}(c)$ for every row, and updating the relevant hints by XORing with the incoming relocated entries. The c-th hint and column c in the hint table are now *consumed*.

In the PIR construction, we make two modifications to the data structure: Firstly, we use pseudorandom permutations for the permutation P in the data structures. This is possible as in Construction 3.5, the permutation is read only. Secondly, we use only one copy of the relocation history Hist to be shared by all T DS instances. As all the rows share the same Relocate history, storing only one copy is sufficient.

4.1 Correctness

We will now show that the construction satisfies correctness. We will start by proving that the hints are always of correct value:

Lemma 4.3 (Correctness of the hints)**.** *Initially and after every query, the hint value is correct for every unconsumed column. Formally, for every $c \in [m'] \setminus C$, $h_c = \bigoplus_{j\in[T]} \mathsf{DB}_j[\mathsf{DS}_j.\mathsf{Access}(c)]$.*

Proof. We will use induction to show that the hints are correct after each query. Initially, we can see that the hint table is correct by inspecting HintConstruct.

After a query, an element is appended to Hist. From Construction 3.5 we can see that this is equivalent to calling Relocate for each data structure. For a row j, from the correctness of Relocate, we know that at most one element in the row could be relocated to a currently empty position. We can verify that in Reconstruct, for each row that an element is moved in DS, the value is added onto the parity value for the new column. Therefore, the lemma holds. ☐

Lemma 4.4. *Construction 4.2 satisfies the correctness definition of client preprocessing PIR.*

Proof. We will show that the scheme has perfect correctness. From Lemma 3.13, we know that the desired entry can always be found in some hint table column. From Lemma 4.3, we know that the parity value of the column is equal to the actual parity of the database entries in the column. As all other database entries in the column are retrieved from the server and XORed with the stored parity value, the result is the desired entry and the scheme is correct. ☐

4.2 Security

We begin by proving that the scheme is perfectly secure when we use truly uniformly random permutations for the relocation data structures (instead of pseudorandom permutations). In this case, the data structures will satisfy perfect security, and each relocated element will be relocated to a uniformly random position among all empty, unconsumed positions.

Consider the distribution of the hint table formed after removing the consumed columns as illustrated in Fig. 3. Throughout this section, we will use $\mathcal{D}_{m,m'}$ to denote the uniform distribution over arrays of m' elements in $[m] \cup \{\perp\}$ where each element in $[m]$ appears exactly once and the rest are $\perp$. This corresponds to one row of the hint table after removing the consumed columns. There are $\mathbf{P}^m_{m'}$ possible arrays in $\mathcal{D}_{m,m'}$ with equal probability. We will prove that after any sequence of $Q \leq m' - m$ queries, the resulting hint table will have distribution $\mathcal{D}^T_{m,m'-Q}$; in other words, each row in the resulting table has distribution $\mathcal{D}_{m,m'-Q}$ and is independent of other rows.

We will start with a helper experiment to show that, starting from a table with distribution $(\mathcal{D}_{m,m'})^T$, if we pick any element in the table, relocate the column c containing that element, and delete column c, the resulting table has distribution $(\mathcal{D}_{m,m'-1})^T$, even when conditioned on the contents of column c. The experiment is illustrated by Fig. 5.

Experiment 4.5. Parameterized by number of elements m, size of array m', number of rows T, and an index $i \in [m \cdot T]$, the experiment $\mathsf{RelocateColumn}_{m,m',T,i}$ is defined as follows:

0	⊥	1	⊥	2	3
⊥	3	2	1	0	⊥
3	⊥	0	1	⊥	2
3	2	⊥	⊥	0	1

Table H

1
2
0
⊥

0	1	⊥	2	3
2	3	1	0	⊥
3	⊥	1	0	2
3	2	⊥	0	1

Experiment outputs: column $H(\cdot, c)$ and table H'

Figure 5: Illustration for Experiment 4.5

- Sample a table with T rows and m' columns $H \xleftarrow{\$} (\mathcal{D}_{m,m'})^T$, let H' be an empty table with T rows and $m'-1$ columns.
- Let c be the column that contains element $(i \mod m)$ in row $\lfloor i/m \rfloor$ of H.
- Output $q = (H_0[c], H_1[c], \ldots, H_{T-1}[c])$.
- For $j = 0, 1, \ldots, T-1$:
  - If $H_j[c] \neq \perp$, let r be an random index such that $H_j[r] = \perp$ and set $H_j[r] \leftarrow H_j[c]$.
  - Let $H'_j \leftarrow (H_j[0], H_j[1], \ldots, H_j[c-1], H_j[c+1], \ldots, H_j[m'-1])$.
- Output H'.

Lemma 4.6. *The output H' of $\mathsf{RelocateColumn}_{m,m',T,i}$ follows distribution $(\mathcal{D}_{m,m'-1})^T$ even when conditioned on the column $q = (H_0[c], H_1[c], \ldots, H_{T-1}[c])$.*

Proof. We will show that when conditioned on a column $q \in ([m] \cup \{\perp\})^T$, H' have equal probability to be each of the $(\mathbf{P}^m_{m'-1})^T$ possible tables. This will directly prove that H' follows distribution $(\mathcal{D}_{m,m'-1})^T$.

From the assumption that H follows distribution $(\mathcal{D}_{m,m'})^T$, we know that i has an equal probability of being in any column c. When conditioned on q and c, H has an equal probability of being every possible table where the c-th column is equal to q. If k values in q are $\perp$, the table H has $(\mathbf{P}^m_{m'-1})^k \cdot (\mathbf{P}^{m-1}_{m'-1})^{T-k}$ possibilities.

For every possible arrangement of H, each of the $T-k$ elements in column c will be randomly relocated among the $m'-m$ empty positions in the same row. The key observation is that every possible arrangement of H' corresponds to a valid table H along with destinations for relocating elements in column c. To construct this mapping, we can use the positions of relocated elements (elements in q) to recreate the choice of relocation positions, and use other elements to recreate the other columns of H. This creates a one-to-one mapping between the $(\mathbf{P}^m_{m'-1})^k \cdot (\mathbf{P}^{m-1}_{m'-1})^{T-k} \cdot (m'-m)^{T-k} = (\mathbf{P}^m_{m'-1})^T$ possibilities. As every possible valid table of T rows and $m'-1$ columns have the same probability of being the output, H' has distribution $(\mathcal{D}_{m,m'-1})$.^{T} □

We now prove that the server's view is identically distributed when the permutations used are uniformly random. This implies that the scheme has perfect security when we use uniformly random permutations.

Lemma 4.7. *For any size parameter m, number of queries $Q \leq m'-m$, and query sequence $i_1, i_2, \ldots, i_Q$, the client requests to the server made by Construction 4.2 when $\mathsf{DS}_0, \mathsf{DS}_1, \ldots, \mathsf{DS}_{T-1}$ use uniformly random permutations are distributed as follows:*

All Q requests are independent. For the t-th client request, all T elements in the request are independently and identically distributed, and each element is sampled uniformly from $[m]$ with probability $\frac{m}{m'-t+1}$ and $\perp$ with probability $1 - \frac{m}{m'-t+1}$.

Proof. We will use induction on the number of queries Q. In addition to the distribution of the client requests, we will also show that by the end of the Q queries, the table formed by the unconsumed columns of the hint table has distribution $(\mathcal{D}_{m,m'-Q})^T$.

Base case. For $Q = 0$, no request has been sent. The hint table has distribution $(\mathcal{D}_{m,m'})^T$ as the data structure satisfies perfect security when the permutations are uniformly random.

Inductive case. Assuming that for any sequence of $Q - 1$ queries, the client requests to the server are distributed as described in the lemma and the hint table has distribution $(\mathcal{D}_{m,m'-Q+1})^T$, we will show that the lemma holds for Q queries.

To see that the Q-th client request is distributed as described, we note that from the induction hypothesis, the unconsumed columns of the hint table used for the Q-th request have distribution $(\mathcal{D}_{m,m'-Q+1})^T$. Without loss of generality, let c be the column that contains the desired element. Since the hint table rows are independently distributed, elements in the other rows of column c are distributed as an element of a random row in $\mathcal{D}_{m,m'-Q+1}$. For the row that contains the desired database entry, the distribution follows from the replacement of the desired entry in Construction 4.2.

We now focus on the distribution of unconsumed columns of the hint table after Q queries. As the unconsumed columns have distribution $(\mathcal{D}_{m,m'-Q+1})^T$ before the query, we have that the resulting columns have the same distribution $\mathsf{RelocateColumn}_{m,m'-Q+1,T,i_Q}$ when DS is perfectly secure. This is because Query relocates elements of the consumed column in the same way when DS satisfies perfect security. From Lemma 4.6, we know that the distribution of the output after the query is $(\mathcal{D}_{m,m'-Q})^T$. Therefore the lemma holds. □

Lemma 4.8. *Construction 4.2 satisfies the security definition of client preprocessing PIR.*

Proof. From Lemma 4.7, we know that the security experiment of Definition 2.1 outputs 1 with probability $\frac{1}{2}$ when the permutations used are truly random, because the adversary's view is independent of the query sequence. By the security of the pseudorandom permutation, we know that after replacing the truly random permutations with pseudorandom permutations with security parameter λ, the security experiment outputs 1 with probability $\frac{1}{2} + \mathrm{negl}(\lambda)$. Therefore, the scheme is computationally secure. □

4.3 Efficiency

We will now show that the construction is efficient.

Lemma 4.9. *Construction 4.2 satisfies the efficiency requirement of Theorem 4.1.*

Proof. We will show that the scheme satisfies the efficiency requirements for communication, server time and client time.

Communication. For each online query, the client sends T indices of size $\log n$ and the server sends back T elements of size w. Therefore, each online query has communication cost $O(Tw + T \log n)$. The client performs the $O(nw)$ preprocessing step per $m = n/T$ queries, so the amortized communication cost from preprocessing is $O(Tw)$ and overall amortized communication cost is $O(Tw + T \log n)$ bits.

Server time. The server only performs T accesses to the database per online query. The preprocessing phase requires the server to stream all n entries to the client every $\Theta(n/T)$ queries. Therefore the amortized server time is $O(T)$ accesses to entries of size w.

Client storage. The client storage consists of two parts: h and ck. h contains m' XOR sums of size w each and has size $m'w = O(Qw)$. ck contains λ bits of PRF key and the state for Hist. As all DS has the same access pattern, we can use the same Hist across rows. From Construction 3.4, we know that Hist stores an array of size no more than Q and a hash map containing no more than Q elements. Therefore, the size of ck is $O(Q \log n)$ bits and the total client storage is $O(Qw + Q \log n)$ bits.

Client time. We defer the analysis of client computation to Appendix A.2. □

Theorem 4.1 now follows directly from the efficiency, correctness, and security of the construction.

Handling database updates. Our scheme additionally supports efficient updates to the database, where updating a random database entry takes $O(1)$ expected PRP calls and two XORs of size w.

For an update for the i-th entry, the server sends the index i and the old and new values $\mathsf{DB}[i], \mathsf{DB}'[i]$ to the client. The client computes the row $j \leftarrow \lfloor i/m \rfloor$, and the column containing the hint $c \leftarrow \mathsf{DS}_j.\mathsf{Locate}(i \mod m)$. The client then updates the c-th hint $h_c \leftarrow h_c \oplus \mathsf{DB}[i] \oplus \mathsf{DB}'[i]$. After the update, the hints satisfy the correctness property Lemma 4.3 with respect to the updated database, since the updated database entry is contained in exactly one hint.

Acknowledgements

This work is funded in part by the National Science Foundation Award #2246386.

References

[ACLS18] Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty. Pir with compressed queries and amortized query processing. In *2018 IEEE symposium on security and privacy (SP)*, pages 962–979. IEEE, 2018. (cit. on p. 1, 9)

[BIM00] Amos Beimel, Yuval Ishai, and Tal Malkin. Reducing the servers computation in private information retrieval: PIR with preprocessing. In *Advances in Cryptology—CRYPTO 2000: 20th Annual International Cryptology Conference Santa Barbara, California, USA, August 20–24, 2000 Proceedings 20*, pages 55–73. Springer, 2000. (cit. on p. 1, 9)

[BIPW17] Elette Boyle, Yuval Ishai, Rafael Pass, and Mary Wootters. Can we access a database both locally and privately? In *Theory of Cryptography: 15th International Conference, TCC 2017, Baltimore, MD, USA, November 12-15, 2017, Proceedings, Part II 15*, pages 662–693. Springer, 2017. (cit. on p. 9)

[CG97] Benny Chor and Niv Gilboa. Computationally private information retrieval. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 304–313, 1997. (cit. on p. 1)

[CGHK22] Henry Corrigan-Gibbs, Alexandra Henzinger, and Dmitry Kogan. Single-server private information retrieval with sublinear amortized time. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 3–33. Springer, 2022. (cit. on p. 2, 9, 27, 28)

[CGK20] Henry Corrigan-Gibbs and Dmitry Kogan. Private information retrieval with sublinear online time. In *Advances in Cryptology–EUROCRYPT 2020: 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10–14, 2020, Proceedings, Part I 39*, pages 44–75. Springer, 2020. (cit. on p. 2, 9)

[CGKS95] B Chor, O Goldreich, E Kushilevitz, and M Sudan. Private information retrieval. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*, pages 41–50. IEEE, 1995. (cit. on p. 1)

[Cha04] Yan-Cheng Chang. Single database private information retrieval with logarithmic communication. In *Information Security and Privacy: 9th Australasian Conference, ACISP 2004, Sydney, Australia, July 13-15, 2004. Proceedings 9*, pages 50–61. Springer, 2004. (cit. on p. 1)

[CHR17] Ran Canetti, Justin Holmgren, and Silas Richelson. Towards doubly efficient private information retrieval. In *Theory of Cryptography: 15th International Conference, TCC 2017, Baltimore, MD, USA, November 12-15, 2017, Proceedings, Part II 15*, pages 694–726. Springer, 2017. (cit. on p. 9)

[CKGS98] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *Journal of the ACM (JACM)*, 45(6):965–981, 1998. (cit. on p. 1)

[CMS99] Christian Cachin, Silvio Micali, and Markus Stadler. Computationally private information retrieval with polylogarithmic communication. In *Advances in Cryptology—EUROCRYPT'99: International Conference on the Theory and Application of Cryptographic Techniques Prague, Czech Republic, May 2–6, 1999 Proceedings 18*, pages 402–414. Springer, 1999. (cit. on p. 1)

[GGM86] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *Journal of the ACM (JACM)*, 33(4):792–807, 1986. (cit. on p. 11)

[GR05] Craig Gentry and Zulfikar Ramzan. Single-database private information retrieval with constant communication rate. In *International Colloquium on Automata, Languages, and Programming*, pages 803–815. Springer, 2005. (cit. on p. 1)

[GZS24] Ashrujit Ghoshal, Mingxun Zhou, and Elaine Shi. Efficient pre-processing pir without public-key cryptography. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 210–240. Springer, 2024. (cit. on p. 2)

[HMR12] Viet Tung Hoang, Ben Morris, and Phillip Rogaway. An enciphering scheme based on a card shuffle. In *Advances in Cryptology–CRYPTO 2012: 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, pages 1–13. Springer, 2012. (cit. on p. 9)

[HPPY24] Alexander Hoover, Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. Plinko: Single-server pir with efficient updates via invertible prfs. *Cryptology ePrint Archive*, 2024. (cit. on p. 2, 9)

[IKOS04] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Batch codes and their applications. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 262–271, 2004. (cit. on p. 9)

[Imp95] Russell Impagliazzo. A personal view of average-case complexity. In *Proceedings of Structure in Complexity Theory. Tenth Annual IEEE Conference*, pages 134–147. IEEE, 1995. (cit. on p. 2)

[ISW24] Yuval Ishai, Elaine Shi, and Daniel Wichs. Pir with client-side preprocessing: Information-theoretic constructions and lower bounds. In *Annual International Cryptology Conference*, pages 148–182. Springer, 2024. (cit. on p. 2, 27, 28)

[KCG21] Dmitry Kogan and Henry Corrigan-Gibbs. Private blocklist lookups with checklist. In *30th USENIX security symposium (USENIX Security 21)*, pages 875–892, 2021. (cit. on p. 2, 9)

[KO97] Eyal Kushilevitz and Rafail Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *Proceedings 38th annual symposium on foundations of computer science*, pages 364–373. IEEE, 1997. (cit. on p. 1)

[KO00] Eyal Kushilevitz and Rafail Ostrovsky. One-way trapdoor permutations are sufficient for non-trivial single-server private information retrieval. In *Advances in Cryptology—EUROCRYPT 2000: International Conference on the Theory and Application of Cryptographic Techniques Bruges, Belgium, May 14–18, 2000 Proceedings 19*, pages 104–121. Springer, 2000. (cit. on p. 1)

[LMW23] Wei-Kai Lin, Ethan Mook, and Daniel Wichs. Doubly efficient private information retrieval and fully homomorphic ram computation from ring lwe. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing*, pages 595–608, 2023. (cit. on p. 9)

[LP23a] Arthur Lazzaretti and Charalampos Papamanthou. Near-optimal private information retrieval with preprocessing. In *Theory of Cryptography Conference*, pages 406–435. Springer, 2023. (cit. on p. 2, 9)

[LP23b] Arthur Lazzaretti and Charalampos Papamanthou. Treepir: Sublinear-time and polylog-bandwidth private information retrieval from ddh. In *Annual International Cryptology Conference*, pages 284–314. Springer, 2023. (cit. on p. 2, 9)

[LP24] Arthur Lazzaretti and Charalampos Papamanthou. Single pass Client-Preprocessing private information retrieval. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 5967–5984, Philadelphia, PA, August 2024. USENIX Association. (cit. on p. 3, 4, 5, 9)

[MCR21] Muhammad Haris Mughees, Hao Chen, and Ling Ren. Onionpir: Response efficient single-server pir. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2292–2306, 2021. (cit. on p. 1)

[MR14] Ben Morris and Phillip Rogaway. Sometimes-recurse shuffle: almost-random permutations in logarithmic expected time. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 311–326. Springer, 2014. (cit. on p. 9, 11)

[MR23] Muhammad Haris Mughees and Ling Ren. Vectorized batch private information retrieval. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 437–452. IEEE, 2023. (cit. on p. 9)

[MW22] Samir Jordan Menon and David J Wu. Spiral: Fast, high-rate single-server pir via fhe composition. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 930–947. IEEE, 2022. (cit. on p. 1)

[OSI07] Rafail Ostrovsky and William E Skeith III. A survey of single-database private information retrieval: Techniques and applications. In *International Workshop on Public Key Cryptography*, pages 393–411. Springer, 2007. (cit. on p. 1)

[PPY18] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. Private stateful information retrieval. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1002–1019, 2018. (cit. on p. 1)

[RMI23] Ling Ren, Muhammad Haris Mughees, and Sun I. Simple and practical amortized sublinear private information retrieval using dummy subsets. Cryptology ePrint Archive, Paper 2023/1072, 2023. (cit. on p. 2, 9, 29)

[RY13] Thomas Ristenpart and Scott Yilek. The mix-and-cut shuffle: small-domain encryption secure against n queries. In *Advances in Cryptology–CRYPTO 2013: 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, pages 392–409. Springer, 2013. (cit. on p. 9)

[SACM21] Elaine Shi, Waqar Aqeel, Balakrishnan Chandrasekaran, and Bruce Maggs. Puncturable pseudorandom sets and private information retrieval with near-optimal online bandwidth and time. In *41st Annual International Cryptology Conference (CRYPTO)*, pages 641–669. Springer, 2021. (cit. on p. 2, 9)

[SS12] Emil Stefanov and Elaine Shi. Fastprp: Fast pseudo-random permutations for small domains. *Cryptology ePrint Archive*, 2012. (cit. on p. 9)

[Yeo23] Kevin Yeo. Lower bounds for (batch) PIR with private preprocessing. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 518–550. Springer, 2023. (cit. on p. 2, 27, 28)

[ZLTS23] Mingxun Zhou, Wei-Kai Lin, Yiannis Tselekounis, and Elaine Shi. Optimal single-server private information retrieval. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 395–425. Springer, 2023. (cit. on p. 2, 9)

[ZPZS24] Mingxun Zhou, Andrew Park, Wenting Zheng, and Elaine Shi. Piano: extremely simple, single-server pir with sublinear server computation. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 4296–4314. IEEE, 2024. (cit. on p. 2, 9, 29)

A Deferred proofs on client efficiency

We now present deferred client efficiency proofs from Section 3 and Section 4.

A.1 Deferred proofs in Section 3

We will prove the following lemmas that capture the efficiency stated in Theorem 3.3.

Lemma A.1 (Efficiency of Locate)**.** *The expected number of PRP calls for* Locate *is $O(1)$ for a random input $e \in [m]$.*

Proof. To show that Locate takes $O(1)$ PRP calls in expectation, we will show that calling Locate for all m possible inputs in $[m]$ takes $O(m)$ PRP calls in total. From Fact 3.11, we know that for each element in $e \in [m]$, $c = P(e)$ are start of disjoint chains in G and that in each step of Locate, the algorithm uses one PRP call to traverse by the out-edge.

As chains are disjoint, the position variable c will not take the same value more than once accross all calls to Locate for all $e \in [m]$. As there are m' possible values for c, all calls to Locate will not take more than m' PRP calls. Therefore, the expected number of PRP calls for Locate is $O(m'/m) = O(1)$. □

Lemma A.2 (Efficiency of Access). *The expected number of PRP calls for* Access *is* $O(1)$ *for a random input* $c \in [m'] \setminus C$.

Proof. The proof is similar to Lemma A.1. We will show that calling Access for all possible inputs in $[m'] \setminus C$ takes $O(m')$ PRP calls in total. From Fact 3.12, we know that each call to Access start from ends of disjoint chains and in each step uses a PRP call to traverse the in-edge.

Therefore, the same node will not be visited more than once accross all calls to Access for all c. As the total cost of all calls to Access cannot exceed m' PRP calls, the expected number of PRP calls for Access is $O(m'/(m - |C|)) = O(1)$ since $m' - |C| \geq m$. □

Lemma A.3 (Efficiency of $\mathsf{Relocate}$). *For a uniformly random position* $c \in [m'] \setminus C$, $\mathsf{Relocate}(c)$ *and then* $\mathsf{Locate}(e)$ *(if relocated element* $e \neq \perp$ *exists) uses* $O(1)$ *expected PRP calls in addition to* $O(1)$ *amortized PRP calls across* m *calls with distinct* c.

Proof. The $\mathsf{Relocate}$ operation takes constant time and zero PRP calls. We shall now focus on the cost of $\mathsf{Locate}(e)$.

If there does not exist an element at position c, Locate is not called. In this case, the number of edges starting from an empty node in G increases by one.

If there exists an element at position c, that element has uniformly distribution in $[m]$ as c is uniformly random. From Lemma A.1, we know that $\mathsf{Access}(e)$ takes $O(1)$ expected PRP calls before $\mathsf{Relocate}(c)$. The relocate operation adds an edge from c to $P(m + |C|)$, which will increase the cost of $\mathsf{Access}(e)$. However, we can see that every additional edge traversed after edge $(c, P(m + |C|))$ is now in the chain starting from $P(e)$ instead of a chain starting from an empty node. As all such edges are created in prior $\mathsf{Relocate}$ operations, this part takes $O(1)$ amortized PRP calls. □

A.2 Deferred proofs in Section 4

In this subsection, we will present the proof that Construction 4.2 satisfies the client efficiency requirement of Theorem 4.1.

Lemma A.4. *Construction 4.2 uses amortized client computation of* $O(T)$ *XORs of size* w *elements and* $O(T)$ *small-domain PRP calls over* $Q = n/T$ *queries, assuming that the queries are independent of the randomness of the protocol.*

Proof. We will first prove the statement with the assumption that queries are uniformly random, and then show how to remove that assumption.

Efficiency for random queries. Assuming that queries are uniformly random, we will show that the amortized client time is $O(T)$ XORs of size w elements and $O(T)$ calls to PRP. $\mathsf{Reconstruct}$ is the only online phase where the client operates on entries, and the client performs T XORs of w-bit entries. We shall now focus on the number of online PRP calls.

When constructing a request to the server, the client calls Locate once to find the column of the desired entry, which takes $O(1)$ expected PRP calls according to Lemma A.1. In addition, the client uses T calls of Access to get elements in the column to be consumed and T calls to Locate to update the hints.

From Lemma A.3, we know that in finding the columns that elements are relocated to using Locate, the traversals on the newly added edges take $O(1)$ amortized PRP calls over $Q = n/T$ operations for each row. As there are T rows, this add $O(T)$ to the final amortized cost.

From Lemma A.2 and Lemma A.3, we know that the T Access calls and the traversals on preexisting edges in all T Locate calls takes $O(T)$ PRP calls in expectation for a random column. For a column c, let PRP calls in $e \leftarrow \mathsf{Access}(c)$ and $\mathsf{Locate}(e)$ across all T data structures be x_c. As the expectation is over unconsumed columns, we have $\sum_{c \in [m'] \setminus C} x_c / (m' - |C|) = O(T)$.

Let the likelihood of column c being chosen be p_c. As the desired entry i is uniformly random and the maximum number of elements in a column is T, the maximum likelihood of a column containing the desired entry is $T/n = 1/m$.

Therefore, the expected number of PRP calls is

$$\sum_{c\in[m']\setminus C} p_c \cdot x_c \leq \sum_{c\in[m']\setminus C} 1/m \cdot x_c = \frac{m' - |C|}{m} \sum_{c\in[m']} x_c/(m' - |C|) = O(T).$$

For preprocessing, n XORs and PRP calls are made every n/T queries, yielding an amortized cost of $O(T)$. Thus, the total amortized computation is $O(T)$ XORs of size w and $O(T)$ PRP calls.

Efficiency for arbitrary queries. In order to remove the assumption of random queries, we use an additional PRP to make the queries appear random to the protocol. At the beginning of the protocol, the client chooses a PRP $P : [n] \to [n]$ over domain $[n]$ of database indices and sends the PRP key to the server. The client and server then execute the protocol on a permuted database where indices are permuted with P. For every query, the client uses permutation P to transform the desired index into the index of the same entry in the permuted database. Assuming that the PRP is independent of the queries, the permuted sequence of queries will appear random. □

In the proof, we used the assumption that we can choose a PRP independent of the queries. This requires the queries to be independent of the randomness used in the protocol. Intuitively, when the client chooses the queries, there will be no incentive to sabotage its own efficiency by choosing the worst-case queries according to the client's own inner state. We note that this requirement is only required for the client's efficiency and is not needed for the correctness, security, storage, or server time of the protocol.

B Lower bounds and barriers

Existing lower bounds and barriers for client preprocessing PIR consider the case where each database entry is a single bit. In this section, we show that the lower bound on amortized probes [Yeo23] and the barrier on amortized communication [ISW24] both extend to w-bit entries.

B.1 Lower bound of [Yeo23]

In this subsection, we first extend the lower bound by Yeo [Yeo23] to w-bit entries. We then use techniques presented in [CGHK22] to obtain a lower bound on amortized server probes over multiple queries.

Theorem B.1 ([Yeo23] generalized)**.** *For any $\ell = O(1)$ and any ℓ-server computationally secure (against single compromised server) client preprocessing PIR scheme such that, on database size n and entry size w,*

- *the server stores the database in its original form,*
- *the client stores S bits before the query,*
- *the server probes T entries of the database, and*
- *the client retrieves the desired entry with error probability at most $1/15$*

must have $ST = \Omega(nw)$.

[Yeo23] proved the theorem for the $w = 1$ case by converting a PIR scheme that breaches the lower bound into an encoding of the entire database that is information-theoretically impossible. Their proof first presented properties of queried and probed entries and ways for finding *good* queries that uncover large fractions of queried entries without probing them. These results are independent of the entry size.

[Yeo23] then constructs an encoding scheme utilizing the fact that using the entries probed, the queried entries can be deduced "for free" without being probed. The encoding consists of three types of objects: the client storage of the protocol, database entries stored in plaintext, and metadata that depend only on the database size n and the protocol's probe pattern. After setting the entry size to w-bits, the encoded database will have nw bits instead of n bits. For a fixed number of probes, this will allow us to increase the client storage size by w times and still be able to derive the contradiction, as the size of each part of the encoding will increase by a factor of at most w (the metadata will not increase in size). Therefore, we have $ST = \Omega(nw)$.

Amortized server probes. The lower bound in Theorem B.1 can be extended to the amortized server probes over many PIR queries, using a reduction from [CGHK22] showing that any multi-query client preprocessing PIR scheme using T server probes on average implies a single-query scheme using $O(T)$ probes with the same storage. From a many-query PIR scheme, they proved that there exists a sequence of queries such that the subsequent query uses T probes in expectation. By having the server error when probes exceed a constant factor of T, it is possible to obtain a scheme with S bits of storage, makes most $O(T)$ probes, and has constant error probability. This contradicts the lower bound on a single query. We refer the reader to the proof of Theorem 6.2 in [CGHK22] for details.

Theorem B.2. *For any $\ell = O(1)$ and any ℓ-server computationally secure (against single compromised server) client preprocessing PIR scheme for many adaptive queries such that, on database size n and entry size w,*

- *the server stores the database in its original form,*
- *the client stores st most S bits between queries, and*
- *the server probes T entries of the database when amortized for all queries*

must have $ST = \Omega(nw)$.

We note that when w is asymptotically no larger than the word size, Theorem B.2 immediately yields the same $ST = \Omega(nw)$ bound where T is the amount of server computation, because each probe takes constant server computation.

B.2 SZK barrier of [ISW24]

In this subsection, we note that the SZK barrier of [ISW24] extends to the w-bit entry case. We then present a way to understand their result as a barrier to the amortized communication of client preprocessing PIR schemes.

Theorem B.3 ([ISW24] generalized)**.** *Any database-oblivious client preprocessing PIR scheme with S bits of storage, and a consecutive sequence of $3S/w$ queries consume less than $nw/3$ bandwidth, implies an average-case hard promise problem in SZK. In particular, it implies a separation of (promise) SZK from BPP.*

As a hard problem is not known to exist assuming one-way functions, the theorem can be seen as a barrier against PIR schemes in minicrypt. The original barrier immediately generalizes to the w-bit entry case as their proof actually proves a barrier for any scheme that can retrieve a single entry of $3S$ bits from a database of $n/3S$ entries. In their proof, queries for $3S$ consecutive

bits of the database are used for the reduction. For databases of n w-bit entries, the same reduction still works with $3S/w$ queries each of w bits.

Our protocol downloads $O(nw/S)$ entries which is $O(nw^2/S)$ bits per online query. This is more than the online communication cost of some previous protocols [RMI23, ZPZS24], which downloads $O(w)$ bits per query. However, we note with the following theorem that the barrier implies that schemes with low worst-case online communication must have high amortized communication due to the preprocessing phase.

Theorem B.4. *Any database-oblivious client preprocessing PIR scheme with S bits of storage, (worst-case) online communication cost of less than $1/9 \cdot nw^2/S$ bits, and amortized communication of less than $1/18 \cdot nw^2/S$ implies an average-case hard promise problem in SZK.*

Proof Sketch. If the PIR scheme supports more than $3S/w$ queries where each query costs less than $1/9 \cdot nw^2/S$ bits, then all $3S/w$ queries can be answered with less than $nw/3$ bits of communication. This breaches the barrier in Theorem B.3.

If the PIR scheme supports $Q < 3S/w$ queries and has amortized cost of less than $1/18 \cdot nw^2/S$ bits across all Q queries (including the preprocessing phase), then we can conduct the $3S/w$ queries in batches of Q queries and run the preprocessing phase once for each batch. This will allow us to answer all $3S/w$ queries with less than $nw/3$ bits of communication, as we will "waste" at most Q queries, and the overall cost will be less than $(6S/w) \cdot (1/18 \cdot nw^2/S) = nw/3$ bits. This again breaches the barrier in Theorem B.3. □