PIANO: Extremely Simple, Single-Server PIR with Sublinear Server Computation

Mingxun Zhou Andrew Park Elaine Shi Wenting Zheng

Carnegie Mellon University

Abstract

We construct a sublinear-time single-server pre-processing Private Information Retrieval (PIR) scheme with optimal client storage and server computation (up to poly-logarithmic factors), only relying on the assumption of the existence of One Way Functions (OWF). Our scheme achieves amortized $\tilde{O}(\sqrt{n})$ online server computation and client computation and $O(\sqrt{n})$ online communication per query, and requires $\widetilde{O}_\lambda(\sqrt{n})$ client storage. Unlike prior single-server PIR schemes that rely on heavy cryptographic machinery such as Homomorphic Encryption, our scheme only utilizes lightweight cryptography such as PRFs, which is easily instantiated in practice. To our knowledge, this is the first practical implementation of a single-server sublinear-time PIR scheme. Compared to existing linear time single-server solutions, our schemes are faster by $10-300\times$ and are comparable to the fastest two-server schemes. In particular, for a 100GB database of 1.6 billion entries, our experiments show that our scheme has less than 40ms online computation time on a single core.

1 Introduction

Suppose that a server has a large public database DB indexed by $0, 1, \ldots, n-1$, e.g., the repository of DNS entries or a list of blocklisted websites. A client wants to fetch the i-th entry of the database. Although the database is public, the client wants to hide which entry it is interested in. Chor, Goldreich, Kushilevitz, and Sudan [CGKS95, CKGS98] first investigated how to solve this problem, and they came up with cryptographic construction called Private Information Retrieval (PIR). Since then, a long line of work focused on improving the asymptotical and concrete performance of PIR [CG97, Cha04, GR05, CMS99, KO97, Lip09, OS07, Gas04, DG16, PR93, DCIO98, BLW17, BGI16, PPY18, IKOS04, Hen16, HH17, ACLS18, IKOS06, LG15, DHS14, ACLS18, MR22, CK20, CHK22, KCG21, dCP22, LMW22, ZLTS23, LP22, HHCG+22, MW22, LP23, DPC22].

Two classes of PIR schemes. There are two main classes of PIR schemes, depending on whether they rely on preprocessing. Classical PIR schemes do not perform any preprocessing of the database, and the server simply stores an original copy of the database DB. In this setting, although we can achieve polylogarithmic communication per query, the server's computation overhead must be *linear* in the size of the database. Beimel, Ishai, and Malkin [BIM00] showed that the linear server computation overhead is inherent — intuitively, if there is some entry that is not touched during some query, then the server learns that the client is not interested in this entry. To overcome this prohibitive linear server computation barrier, Beimel et al. introduced the *preprocessing* model [BIM00], which was further explored in several subsequent works [CK20, SACM21, LP23, CHK22, KCG21, ZLTS23, LP22, LMW22]. In the *client-specific preprocessing* model (also called the *subscription* model), we have each client download and store a

"hint" from the server during preprocessing. In this model, it is known that with $\widetilde{O}_\lambda(\sqrt{n})$ client-side storage[1], each online query can be accomplished with polylogarithmic communication and $\widetilde{O}_\lambda(\sqrt{n})$ server and client computation [ZLTS23, LP22]. Another possible model is the *global preprocessing* model, in which the server performs a global preprocessing and computes an encoding of the database upfront for all clients. In this model, the most recent breakthrough work by Mook, Lin, and Wichs [LMW22] showed that for any constant $\varepsilon > 0$, with $O(n^{1+\varepsilon})$ amount of server storage, each query can be accomplished with $(\mathsf{poly}\log n)^{1/\varepsilon}$ communication and $(\mathsf{poly}\log n)^{1/\varepsilon}$ server computation.

Practical landscape for PIR. The community have made various attempts to implement and optimize PIR for practical applications [MW22, HHCG+22, DPC22, ACLS18, MR22]. In the single-server setting, to the best of our knowledge, only classical-style PIR schemes with *linear* server computation have been implemented. Although recent works [HHCG+22] managed to achieve server-computation throughput comparable to the native memory bandwidth, the linear amount of computation severely limits the scalability to larger databases, and precludes various killer applications such as private DNS (where the database can be as large as 100GB). Unsurprisingly, prior works ran experiments for databases of size up to 8GB [MW22, HHCG+22], and the server time per query is more than one second for this data size.

A natural question is why prior implementation efforts did not choose schemes with preprocessing despite their better asymptotical performance. The reason is that known single-server, preprocessing sublinear PIR schemes are theoretical in nature. In particular, known schemes with polylogarithmic communication [ZLTS23,LP22,LMW22] require one or more of the following heavyweight cryptographic primitives: Fully Homomorphic Encryption (FHE), Privately Programmable PRFs [BLW17, PS18, KW21], and polynomial encoding data structures [KU11], which introduce astronomical constants in the concrete performance. Unfortunately, within the limits of known techniques, we are still very far from making these cryptographic primitives practical (or even implementable)! Finally, although the work of Corrigan-Gibbs et al. [CHK22] showed how to get sublinear server computation using only linear homomorphic encryption, they pay the price of much worse asymptotics, that is, $\widetilde{O}_\lambda(n^{2/3})$ for client storage and server computation. Consequently, their scheme is also not a sweetspot for practical implementation.

Time for a paradigm shift for practical PIR? Given the status quo, we ask the following natural question:

Can we have a concretely efficient, single-server PIR scheme with sublinear server computation?

An affirmative answer to the above question promises to bring a paradigm shift for the practical landscape of single-server PIR! Specifically, our dream is to eventually eschew the linear server computation regime for practical implementations, and thus allow scaling to large database sizes.

1.1 Our Contributions

We propose a novel single-server PIR scheme called Piano (short for "Private Information Access NOw"). Piano adopts the client-specific pre-processing model. With roughly $\widetilde{O}(\sqrt{n})$ client-side storage, we achieve $\widetilde{O}(\sqrt{n})$ online communication and computation per query. The most notable feature of Piano lies in its *simplicity*. Unlike prior sublinear PIR schemes [CK20, CHK22, ZLTS23,

[1]Throughout the paper, we use $\widetilde{O}(\cdot)$ or $\widetilde{\Theta}(\cdot)$ to hide polylogarithmic terms, and the subscript in $O_\lambda(\cdot)$ hides terms related to some computational security parameter λ.

LP22], we do not need any form of homomorphic encryption or other heavy-weight cryptographic primitives such as privately puncturable PRFs [BKM17, CC17, BTVW17]. In fact, *the only cryptographic primitive we need is pseudorandom functions (PRFs)*, which can be accelerated through the AES-NI instruction sets available in most modern processors. Moreover, our construction is completely *self-contained* and we need not invoke any existing PIR scheme as a building block.

Optimality. Corrigan-Gibbs, Henzinger and Kogan [CHK22] showed a lower bound for any adaptive PIR scheme without server-side preprocessing. In particular, their lower bound states that if the client stores S bits and the amortized server computation time is T, it must be that $\mathsf{ST} = \Omega(n)$. Our scheme matches this lower bound (up to poly-logarithmic factors). However, the per-query $\Theta(\sqrt{n})$ communication cost in our scheme is not theoretically optimal – previous theoretical work [ZLTS23, LP22] can achieve poly-logarithmic communication per query.

Open-source implementation and evaluation results. We implemented Piano in Go. Given its simplicity, the core implementation contains only around 800 lines of code. We also provide a reference implementation (for tutorial purposes) that contains only 153 lines of code. Both our full implementation and the tutorial implementation are open sourced at `https://github.com/pianopir/Piano-PIR`.

In our evaluation, we mainly compare with SimplePIR [HHCG+22] and the non-private baseline. SimplePIR is the state of the art for practical single-server PIR schemes, and has been shown to outperform all other practical single-server PIR schemes. They also incur roughly $O_\lambda(\sqrt{n})$ bandwidth, but their server computation is linear in n. SimplePIR pushed linear-computation PIR schemes to the very limit: their server performs fewer than one 32-bit multiplication and one 32-bit addition per database byte. Thus, they were able to fully saturate the memory bandwidth for the server computation. Nonetheless, the linear computation severely limits their scalability. For this reason, all prior works on single-server PIR only ran experiments for databases that are at most 8GB in size [MW22, HHCG+22].

We conducted an experiment on a 100GB database with a 60ms RTT coast-to-coast connection. In particular, we chose an 100GB database to roughly match the size of a typical DNS database. Our scheme achieves **99ms** response time, whereas SimplePIR suffers from 11s or higher response time[2]. This represents over **111**× speedup relative to SimplePIR. Since our improvement is asymptotical in nature, the speedup will only become larger as the database size grows. We also ran a non-private baseline for the same scenario, and the response time is 61ms. Therefore, our slowdown w.r.t. the non-private baseline is only **1.62**×.

Additional related work. Although our work focuses on making PIR practical, our result may be of interest from a theoretical perspective, since this is the first time we know how to construct single-server PIR with sublinear server computation from only one-way functions (OWF). Section 5 provides a more detailed comparison with additional related work.

[2]Their implementation itself cannot support network connections or a database as large as 100GB, so this number is a conservatively extrapolated lower bound estimate of their performance.

2 Technical Roadmap

2.1 Starting Point: Inefficient 2-Server PIR Scheme

A general paradigm for constructing PIR. The 2-server setting assumes two non-colluding servers (denoted as the left and right server) both store a copy of the database, which is a stronger assumption than the single-server setting. As a warmup, we first describe an inefficient 2-server PIR scheme. Corrigan-Gibbs and Kogan [CK20] proposed an elegant paradigm for constructing PIR, and the high-level idea is as follows. During the preprocessing phase, the client generates $M := \widetilde{O}(\sqrt{n})$ random sets denoted $S_1, \ldots, S_M$. It sends these sets to the left server, and the left server responds with the parity p_j of every set j, where $p_j := \oplus_{k \in S_j} \mathsf{DB}[k]$, i.e., the xor-sum. The client now stores the hint table $\{(S_j, p_j)\}_{j \in [M]}$. During each online query, let x be the index desired by the client. The client finds in its hint table some set S_j that contains the index x, along with the parity p_j — we can choose parameters such that the client can find such a set with all but negligible probability. The client now sends $S_j \backslash \{x\}$ to the right server where x is removed from the set S_i. The right server responds with the parity $p^* := \oplus_{k \in S_j \backslash \{x\}} \mathsf{DB}[k]$, and the client can now recover $\mathsf{DB}[j]$ by computing the difference of the parities $p^* \oplus p_j$. To make the above idea fully work, we have to additionally address the following two challenges:

- *Privacy.* First, we want to make sure that the set $S_j \backslash \{x\}$ sent to the server during an online query does not reveal any information about the desired index x.
- *Refresh.* Second, we want the client to have a way to replenish entries that have been consumed in the hint table, such that we can support an unbounded number of queries.

Distribution of random sets. To instantiate the above paradigm, we need to decide the distribution for sampling random sets. To this end, we will use a clever idea first introduced by Lazzaretti and Papamanthou [LP23]. Given a database of size n, we divide it into $\sqrt{n}$ chunks of consecutive indices, each of size $\sqrt{n}$. To sample a random set, we sample exactly one random index from each chunk. Therefore, we can encode such a random set using an offset vector defined below.

Definition 2.1 (Offset vector and random set)**.** Given an offset vector of the form $\Delta := (\delta_0, \ldots, \delta_{\sqrt{n}-1}) \in \{0, 1, \ldots, \sqrt{n} - 1\}^{\sqrt{n}}$, we can compute an ordered set of $\sqrt{n}$ indices as follows:

$$\mathbf{Set}(\Delta) := \left\{ i \cdot \sqrt{n} + \delta_i \right\}_{i \in \{0,1,\ldots,\sqrt{n}-1\}}$$

We use the notation $\mathbf{Set}(\Delta)[i] := i \cdot \sqrt{n} + \delta_i$ to mean the i-th index of the set.

Inefficient 2-server PIR: a single query. We shall instantiate the paradigm using the aforementioned distribution for constructing random sets. As mentioned, a random set can be expressed as an offset vector. Henceforth, we may assume that the client stores offset vectors in its hint table of the form $T := \{(\Delta_j, p_j)\}_{j \in M}$.

Recall that during an online query for the database index x, the client finds a hint (Δ^*, p^*) in its hint table, such that $x \in \mathbf{Set}(\Delta^*)$. It wants to learn the parity $\oplus_{k \in \mathbf{Set}(\Delta^*) \backslash \{x\}} \mathsf{DB}[k]$. To achieve this, the client removes x from the offset vector Δ^*, and computes a punctured offset vector

$$\Delta^*_{-i} := (\delta_0, \delta_{i-1}, \delta_{i+1}, \delta_{\sqrt{n}-1}) \text{ where } i := \left\lfloor \frac{x}{\sqrt{n}} \right\rfloor$$

The client sends the punctured offset vector Δ^*_{-i} to the right server. Since the right server does not know which index was punctured, it basically sees a random vector of length $\sqrt{n} - 1$ where

each coordinate is sampled independently at random from $\{0, 1, \ldots, \sqrt{n}-1\}$. Therefore, the right server cannot learn any information about the client's query x. Also because the right server does not know which chunk index was punctured, it is unable to directly compute $\oplus_{k \in \mathbf{Set}(\Delta^*) \setminus \{x\}} \mathsf{DB}[k]$. However, the right server can guess all $\sqrt{n}$ possibilities of the punctured index, and compute all $\sqrt{n}$ corresponding parities. When it sends all possible answers back the the client, the client can read the relevant one herself.

Refresh. So far, we have focused on performing only a single query. To support an unbounded number of queries, the client needs to replenish consumed entries in the hint table. For privacy, it is important that the replenished entry does not alter the distribution of the hint table (even when conditioned on the right server's view). Observe that the entry consumed is a random set subject to containing the current query $x \in \{0, 1, \ldots, n-1\}$. The client wants to replace it with an entry sampled from the same distribution.

Therefore, the client samples a random offset vector $\Delta' := (\delta_0, \ldots, \delta_{\sqrt{n}-1}) \xleftarrow{\$} \{0, \ldots, \sqrt{n}-1\}^{\sqrt{n}}$, and let $i := \left\lfloor \frac{x}{\sqrt{n}} \right\rfloor$ be the chunk index corresponding to the current query x. Replacing the i-th entry in $\mathbf{Set}(\Delta')$ with x would give a new set sampled according to the desired distribution. We can use the following notation to denote the new set's offset vector:

$$\Delta'_{i \to x} := (\delta_0, \ldots, \delta_{i-1},\ x - i \cdot \sqrt{n},\ \delta_{i+1}, \ldots, \delta_{\sqrt{n}-1}) \tag{1}$$

To obtain the parity of this new set $\mathbf{Set}(\Delta'_{i \to x})$, the client performs the following. It sends the offset vector punctured at i, i.e., $\Delta'_{-i} := (\delta_0, \ldots, \delta_{i-1}, \delta_{i+1}, \ldots, \delta_{\sqrt{n}-1})$ to the left server, which responds with all $\sqrt{n}$ possible parities. The client then picks out the relevant one, that is, $p' := \oplus_{k \in \mathbf{Set}(\Delta') \setminus \{x\}} \mathsf{DB}[k]$. The client can now compute the parity for $\mathbf{Set}(\Delta'_{i \to x})$ as $p' \oplus \beta$ where $\beta := \mathsf{DB}[x]$ is the answer to the current query x that the client just learned. The client replaces the consumed entry with $(\Delta'_{i \to x}, p' \oplus \beta)$.

Full description of 2-server scheme. For convenience, we introduce the following notation for describing an offset vector with a hole, where the hole represents the server's guessed location of the puncturing.

Definition 2.2 (Offset vector with a hole)**.** Given an offset vector $\Delta^{i^*} := (\delta_0, \ldots, \delta_{i^*-1}, \perp, \delta_{i^*+1}, \ldots, \delta_{\sqrt{n}-1})$, we define

$$\mathbf{Set}(\Delta^{i^*}) := \left\{ i \cdot \sqrt{n} + \delta_i \right\}_{i \in \{0,1,\ldots,\sqrt{n}-1\}, i \neq i^*}$$

We now give a more formal description of the 2-server scheme.

An Inefficient 2-Server Scheme

Offline preprocessing. ($\mathsf{DB}[k]$ *denotes the* k*-th entry of the database*)

- Client samples M random offset vectors $\Delta_1, \Delta_2, \ldots, \Delta_M \xleftarrow{\$} \{0, 1, \ldots, \sqrt{n}-1\}^{\lceil \sqrt{n} \rceil}$ where $M = \widetilde{\Theta}(\sqrt{n} \log n)$, and sends them to Left.
- For each set $j \in [M]$, Left responds with the parity bit $p_j := \oplus_{k \in \mathsf{Set}(\Delta_j)} \mathsf{DB}[k]$.
- Client stores the hint table $T := \{T_j := (\Delta_j, p_j)\}_{j \in [\sqrt{n}]}$.

Online query for index $x \in \{0, 1, \ldots, n-1\}$.

- **Client Side:** (Client $\Rightarrow$ Right)
  1. Find a hint $T_j := (\Delta_j, p_j)$ in its hint table T such that $x \in \mathbf{Set}(\Delta_j)$. Let $\Delta^* := \Delta_j$ if found, else let Δ^* be a fresh random offset-vector subject to the constraint that $x \in \mathbf{Set}(\Delta^*)$.
  2. Write $\Delta^* := \{\delta_0, \ldots, \delta_{\sqrt{n}-1}\}$, and let $i^* := \left\lfloor \frac{x}{\sqrt{n}} \right\rfloor$ be the chunk index corresponding to x. The client removes index i^* from Δ^*, and sends to Right the resulting offset vector $\Delta^*_{-i^*} := (\delta_0, \ldots, \delta_{i^*-1}, \delta_{i^*+1}, \ldots, \delta_{\sqrt{n}-1})$.
- **Server Side** (Client $\Leftarrow$ Right)
  1. Upon receiving $\Delta = (\delta_0, \ldots, \delta_{\sqrt{n}-2})$, the server Right returns $(q_0, \ldots, q_{\sqrt{n}-1}) \leftarrow \mathbf{PossibleParities}(\Delta)$ which is defined below:
     - For index $i \in \{0, 1, \ldots, \sqrt{n}-1\}$, guessing that i was the index punctured earlier, let $\Delta^i = (\delta_0, \ldots, \delta_{i-1}, \perp, \delta_i, \ldots, \delta_{\sqrt{n}-2})$.
     - Compute the parity $q_i = \oplus_{x \in \mathbf{Set}(\Delta^i)} \mathsf{DB}[x]$.

     As explained later, it only takes $O(\sqrt{n})$ time to compute all $\sqrt{n}$ parities.
- **Client Side:** Upon receiving $(q_0, \ldots, q_{\sqrt{n}-1})$ from Right, the client obtains the answer by $\beta = q_{i^*} \oplus p_j$ (recall x is in i^*-th chunk and j is the hint's index).
- **Refresh** (Client $\Leftrightarrow$ Left)
  1. Client samples a random vector of offsets $\Delta' = \{\delta'_0, \ldots, \delta'_{\sqrt{n}-1}\}$ and sends the vector Δ'_{-i^*} punctured at i^* to Left.
  2. Left responds with $(q'_0, \ldots, q'_{\sqrt{n}-1}) \leftarrow \mathbf{PossibleParities}(\Delta')$.
  3. If a hint $T_j = (\Delta_j, p_j)$ such that $x \in \mathsf{Set}(\Delta_j)$ was found and consumed earlier, let $\Delta_{i^* \to x}$ be the same as Δ' but replacing the i^*-th offset with $x - i^* \cdot \sqrt{n}$. The client replaces the hint T_j with $(\Delta_{i^* \to x}, q'_{i^*} \oplus \beta)$.

Performance of the 2-server scheme. We now analyze the performance of the 2-server scheme. The offline costs can be amortized over an unbounded number of queries, so we focus on the online phase.

- *Communication:* $O(\sqrt{n})$. The online communication per query is $O(\sqrt{n})$.
- *Server computation:* $O(\sqrt{n})$. For each query, the server must compute all $\sqrt{n}$ possible parities. Naïvely computing all parities would take linear in n time. However, observe that for two adjacent guessed vectors Δ^i and Δ^{i+1}, i.e., when the guessed hole (see Definition 2.2) moves from index i to index $i+1$, the corresponding sets $\mathbf{Set}(\Delta^i)$ and $\mathbf{Set}(\Delta^{i+1})$ differ only in two elements, i.e., $\mathbf{Set}(\Delta^i)[i+1]$ and $\mathbf{Set}(\Delta^{i+1})[i]$. This observation allows the server to compute all $\sqrt{n}$ possible parities in $O(\sqrt{n})$ time: the server computes $q_0 = \oplus_{x \in \mathbf{Set}(\Delta^0)} \mathsf{DB}[x]$ first, then computes $q_1, \ldots, q_{\sqrt{n}-1}$ sequentially by xor'ing the two different elements each time.
- *Client storage and computation.* The above 2-server scheme is inefficient in terms of client storage and client computation per query. The client needs to store $\widetilde{O}(\sqrt{n} \log n)$ random sets each of $\sqrt{n}$ size, and this takes super-linear in n space! For each query x, the client must search its hint table for a set that contains x. A naïve scan can take linear time in expectation.

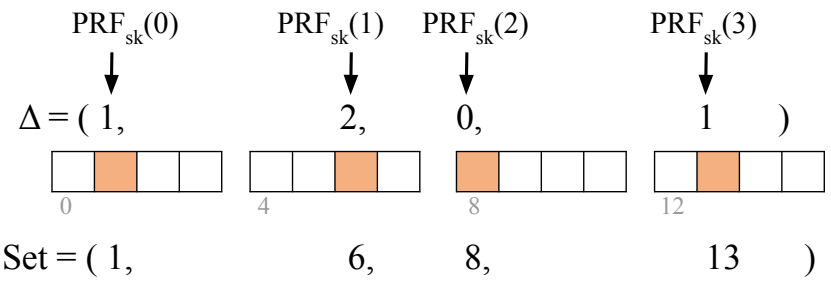

Figure 1: PIANO: From PRF key sk, to offset vector Δ, and to expanded set.

2.2 Compressing Client Storage and Computation with Pseudo-Randomness

Compressing client storage. The idea is to use a PRF key of λ bits (where λ denotes the security parameter) to succinctly represent each set of size $\sqrt{n}$. More specifically, given a PRF key $\mathsf{sk} \in \{0,1\}^\lambda$, we can compute a pseudorandom offset vector $\Delta(\mathsf{sk})$ as follows:

$$\Delta(\mathsf{sk}) := \left(\mathsf{PRF}_{\mathsf{sk}}(0), \mathsf{PRF}_{\mathsf{sk}}(1), \ldots, \mathsf{PRF}_{\mathsf{sk}}(\sqrt{n}-1)\right)$$

where the PRF's output range is $\{0, 1, \ldots, \sqrt{n}-1\}$. We can then compute the set from the offset vector as before.

During the refresh operation, the client samples a fresh PRF key sk', and it wants to force the $\left\lfloor \frac{x}{\sqrt{n}} \right\rfloor$-th element in the set to be the current query x. This can be achieved simply by recording x next to the new sk'. Thus, for each set in the hint table, the client only needs to store $(\mathsf{sk}, \perp)$ or (sk, x'), where the latter form appears if the hint is generated during the refresh phase of some query for the database index x'. In this way, the client can compress its local storage to $\widetilde{O}_\lambda(\sqrt{n})$. Note that although the client is now storing succinct keys, the client still sends the expanded and punctured offset vector to the servers, and we are not compressing the communication.

Compressing client computation. Henceforth, we use the notation $\mathbf{Set}(\mathsf{sk}, x')$ to denote the set expanded by the tuple (sk, x'), where x' is possibly $\perp$. The i-th element of the set $\mathbf{Set}(\mathsf{sk}, x')$ can be computed in $O_\lambda(1)$ time as follows:

$$\mathbf{Set}(\mathsf{sk}, x')[i] := \begin{cases} x' & \text{if } x' \neq \perp \text{ and } \left\lfloor \frac{x'}{\sqrt{n}} \right\rfloor = i \\ i \cdot \sqrt{n} + \mathsf{PRF}_{\mathsf{sk}}(i) & \text{o.w.} \end{cases} \tag{2}$$

It is easy to see that testing whether $x \in \mathbf{Set}(\mathsf{sk}, x')$ takes $O_\lambda(1)$ time, by checking whether $\mathbf{Set}(\mathsf{sk}, x')[i] = x$ where $i := \left\lfloor \frac{x}{\sqrt{n}} \right\rfloor$. Therefore, the client can find a set containing the current query x in $\widetilde{O}(\sqrt{n})$ time.

2.3 Coalescing the Two Servers into One

How to coalesce the two servers into one is technically the most interesting part of our construction. Prior works [CK20, CHK22, ZLTS23, LP22] suggested to have the right server realize the left server, but now wrapped under a fully homomorphic encryption (FHE) scheme. The main drawback of this approach is that FHE is expensive. Although having the same asymptotics as the RAM-based algorithm, the circuit representation of the left server algorithm will have an enormous constant

factor. Furthermore, directly applying FHE-based transformations [CHK22] requires the client to run λ parallel instances of the scheme to get $1 - \mathsf{negl}(\lambda)$ correctness probability. In practice, this incurs unaffordable overhead. For the same reason, it is far from ideal even when we use the FHE-based transformation to convert the state-of-the-art two-server scheme, Tree-PIR by Lazzaretti and Papamanthou [LP23], into a single-server scheme.

Our approach fundamentally departs from the prior works, and we do not need any form of homomorphic encryption.

Simplifying assumptions. Before describing our scheme, we first describe a couple simplifying assumptions — we stress all of these assumptions can be made **without loss of generality** as explained below.

- **Bounded scheme supporting $\tilde{O}(\sqrt{n})$ queries implies an unbounded scheme.** At a very high level, our idea is for the client to not only download the hint table (henceforth referred to as the *primary* table), but also some *backup* hints during the preprocessing. Later, when the client consumes an entry in the primary table, it can use a backup hint to replenish the consumed entry. In particular, we want each preprocessing to prepare for the next $Q := \widetilde{\Theta}(\sqrt{n})$ window of queries. If we can achieve this, we can get an *unbounded* scheme simply by rerunning the preprocessing every Q queries, and amortizing the periodic pre-processing cost over the Q queries. In fact, the preprocessing work can also be deamortized in a straightforward manner to avoid having periodic bursts of workload, similar to previous work [ZLTS23]. Further, the deamortization does not require additional storage on the server.
- **Duplicate suppression.** We may assume that during each window of Q queries, each client does not make any duplicate queries. This is also without loss of generality, since the client can always save the most recent Q queries and their answers. In case it wants to make a duplicate query, it can simply look up the answer locally, and make a filler query (that is not a duplicate) instead. Note we allow two different clients to query the same database index.
- **Random queries.** We may also assume that during each Q-query window, each client queries indices that are sampled at random without replacement. This assumption is also without loss of generality, since the server can simply randomly shuffle the database upfront, and the randomness used in this shuffle is independent of the client's query generation process. For this idea to work, the client must know where the desired query index is in the shuffled database. Therefore, we can have the server sample a pseudorandom permutation (PRP) key upfront which is used to shuffle the database. The server can publish the PRP key, and then the client will be able to compute the index of the query in the shuffled database. If the PRP key is not sampled honestly, it will not affect privacy, but may affect correctness — keep in mind that correctness is impossible anyway if the server is malicious. This assumption is also used in previous PIR constructions [CK20, CHK22].

  Note also that although we assume the queries of each client in a Q-window are random, the queries of two different clients are allowed to be correlated.

Idea 1: Grouped backup hints. Recall that during a query for database index $x \in \{0, 1, \ldots, n-1\}$, we want to replace the consumed hint with a random set but forcing the $\left\lfloor \frac{x}{\sqrt{n}} \right\rfloor$-th element in the set to be x. To achieve this, the client can download roughly $\widetilde{O}(\sqrt{n})$ backup hints during the preprocessing, divided into $\sqrt{n}$ groups each with polylogarithmically many backup hints. The i-th group of backup hints will be used for refresh whenever the client queries x such that $\left\lfloor \frac{x}{\sqrt{n}} \right\rfloor = i$.

Each backup hint is represented by a pseudorandom key $\overline{\mathsf{sk}}$. For an $\overline{\mathsf{sk}}$ that belongs to the i-th group, the client stores with it the parity $\overline{p} := \oplus_{k \in \mathbf{Set}(\overline{\mathsf{sk}})_{-i}} \mathsf{DB}[k]$ where $\mathbf{Set}(\overline{\mathsf{sk}})_{-i}$ denotes the set expanded by sk but *removing the i-th element*. Here, the i-th element is removed because later the client will want to force this location to be the query x.

Now, suppose that the client has just queried a database index x such that $\left\lfloor \frac{x}{\sqrt{n}} \right\rfloor = i$, and it got back the answer β. The client now finds a backup hint $(\overline{\mathsf{sk}}, \overline{p})$ of the i-th group. The client replaces the consumed hint with $\big((\overline{\mathsf{sk}}, x), \beta \oplus \overline{p}\big)$ where the term x signals that the i-th element of the set is forced to be x. Note also that the client can compute the correct parity for this set as $\beta \oplus \overline{p}$ since it just learned that $\mathsf{DB}[x] = \beta$.

Recall that without loss of generality, we can assume that a client's queries are sampled randomly without replacement. One can show that except with negligible probability, we will consume at most polylogarthmically many hints per group during each $\sqrt{n} \ln n$-window. In other words, the client will not run out of backup hints except with negligible probability.

Idea 2: Preprocessing with a streaming algorithm. The client stores in total $\widetilde{O}(\sqrt{n})$ primary and backup hints. The client can set up the hints using only $\widetilde{O}(\sqrt{n})$ extra storage by making only one streaming pass over the database. The algorithm is surprisingly simple. First, the client samples all $\widetilde{O}(\sqrt{n})$ keys for both primary and backup hints, and initializes all of their parities to be 0. At each time step, the client downloads the next $\sqrt{n}$-sized chunk from the server. It then scans through all hints, and for each hint, if some database element in the current chunk should be xor'ed into the parity, it updates the parity accordingly. This preprocessing requires $O(n)$ communication and $\widetilde{O}(n)$ client computation, but when amortized over a window of $\widetilde{O}(\sqrt{n})$ queries, each query only costs $O(\sqrt{n})$ communication and $\widetilde{O}(\sqrt{n})$ client computation. Note that the streaming property of our preprocessing stage does not require the client to have $O(n)$ storage. In particular, the client storage cost is only $\widetilde{O}(\sqrt{n})$ because the client can delete a chunk once the relevant preprocessing is done. This departs from prior work such as [PPY18], which also requires downloading the whole DB during the preprocessing phase, but requires $O(n)$ transient client storage or $O(n)$ amortized computation per query.

2.4 Putting Everything Together

We now give a formal description of our final scheme. As mentioned, our unbounded scheme is obtained from a bounded scheme that supports $Q = \widetilde{\Theta}(\sqrt{n})$ queries (described below), and we can spread the work of the next pre-processing over the current window of Q queries.

We briefly review some convenient notations before giving the formal algorithm description. Given a PRF key sk, earlier in Equation (2), we defined the convenient notation of $\mathbf{Set}(\mathsf{sk}, x')$ where x' is possibly $\perp$. We also define $\mathbf{Set}(\mathsf{sk}) := \mathbf{Set}(\mathsf{sk}, \perp)$. Recall also that Equation (2) gave an $O_\lambda(1)$-time algorithm for computing $\mathbf{Set}(\mathsf{sk})[i]$, i.e., the i-th element of $\mathbf{Set}(\mathsf{sk})$.

Single-Server Scheme for $Q = \Theta(\sqrt{n} \log \kappa \cdot \alpha(\kappa))$ queries

Notation. κ denotes a *statistical* security parameter, λ denotes a computational security parameter. We use $\alpha(\kappa)$ to denote an arbitrarily small super-constant function.

Offline preprocessing.

- Client samples M_1 PRF keys $\mathsf{sk}_1, \ldots, \mathsf{sk}_{M_1} \in \{0,1\}^\lambda$ for the primary table, where $M_1 =$

$\Theta(\sqrt{n}\log\kappa\cdot\alpha(\kappa))$. Initialize the parities $p_1,\ldots,p_{M_1}$ to zeros.

- For $i\in\{0,1,\ldots,\sqrt{n}-1\}$, Client samples $M_2=\Theta(\log\kappa\cdot\alpha(\kappa))$ PRF keys $\overline{\mathsf{sk}}_{i,1},\ldots,\overline{\mathsf{sk}}_{i,M_2}$ — together they form the i-th group of backup keys. Intitialize the parities $\overline{p}_{i,1},\ldots,\overline{p}_{i,M_2}$ to zeros.
- Client downloads the whole DB from the server in a streaming way: when the client has the i-th chunk $\mathsf{DB}[i\sqrt{n}:(i+1)\sqrt{n}]$:
  - For $j\in[M_1]$, let $p_j\leftarrow p_j\oplus\mathsf{DB}[\mathbf{Set}(\mathsf{sk}_j)[i]]$.
  - For $j\in\{0,1,\ldots,\sqrt{n}-1\}/\{i\}$ and $k\in[M_2]$, let $\overline{p}_{j,k}\leftarrow\overline{p}_{j,k}\oplus\mathsf{DB}[\mathbf{Set}(\overline{\mathsf{sk}}_{j,k})[i]]$.
  - Delete $\mathsf{DB}[i\sqrt{n}:(i+1)\sqrt{n}]$ from the local storage.
- At this moment, let $T:=\{((\mathsf{sk}_j,\perp),p_j)\}_{j\in[M]}$ denote the client's *primary table*, and let $\{(\overline{\mathsf{sk}}_{i,j},\overline{p}_{i,j})\}_{j\in[M_2]}$ denote the *i-th group of backup hints*.

Online query for index $x\in\{0,1,\ldots,n-1\}$.

1. **Query:**
   - Client finds a hint $T_j:=((\mathsf{sk}_j,x'),p_j)$ in its primary table T such that $x\in\mathbf{Set}(\mathsf{sk}_j,x')$.
   - Let $\Delta^*:=\Delta(\mathsf{sk}_j,x')$. If no such hint is found, sample a uniform random offset vector $\Delta^*\xleftarrow{\$}\{0,1,\ldots,\sqrt{n}-1\}^{\sqrt{n}}$.
   - Write $\Delta^*:=\{\delta_0,\ldots,\delta_{\sqrt{n}-1}\}$, and let $i^*=\left\lfloor\frac{x}{\sqrt{n}}\right\rfloor$ be the index of x's chunk . The client removes index i^* from Δ^*, and sends to the server the resulting offset vector $\Delta^*_{-i^*}:=(\delta_0,\ldots,\delta_{i^*-1},\delta_{i^*+1},\ldots,\delta_{\sqrt{n}-1})$.
   - The server returns $(q_0,\ldots,q_{\sqrt{n}-1})\leftarrow\mathbf{PossibleParities}(\Delta^*_{-i^*})$.
   - The client sets the answer as $\beta=p_j\oplus q_{i^*}$ if a hint is found earlier. Otherwise the client sets the answer as 0.
2. **Refresh:**
   - Client finds the next unconsumed hint $\left(\overline{\mathsf{sk}}_{i^*,k},\overline{p}_{i^*,k}\right)$ from the i^*-th group. If no unconsumed hint exists in the group, Client generates a random $\overline{\mathsf{sk}}_{i^*,k}$ and lets $\overline{p}_{i^*,k}=0$.
   - If a hint $T_j=((\mathsf{sk}_j,x'),p_j)$ such that $x\in\mathbf{Set}(\mathsf{sk}_j,x')$ was found earlier in the query phase, Client replaces T_j with $\left((\overline{\mathsf{sk}}_{i^*,k},x),\overline{p}_{i^*,k}\oplus\beta\right)$.

As mentioned earlier, $\mathbf{PossibleParities}(\Delta)$ can be evaluated in $O(\sqrt{n})$ time by observing that every pair of adjacent parities differ in only two terms. Therefore, it takes $O(1)$ time to modify the previous parity into the next one. The detailed algorithm is described below.

$O(\sqrt{n})$-time algorithm for $\mathbf{PossibleParities}(\Delta)$

1. Parse Δ as $(\delta_0,\ldots,\delta_{\sqrt{n}-2})$.
2. Compute $q_0=\oplus_{i\in\{1,\ldots,\sqrt{n}-1\}}\mathsf{DB}\left[\delta_{i-1}+i\cdot\sqrt{n}\right]$.

3. For $i = 0, \ldots, \sqrt{n} - 2$, compute $q_{i+1} = q_i \oplus \mathsf{DB}[\delta_i + (i+1) \cdot \sqrt{n}] \oplus \mathsf{DB}[\delta_i + i \cdot \sqrt{n}]$.
4. Return $(q_0, \ldots, q_{\sqrt{n}-1})$.

When n is not a perfect square. So far, we assume that n is a perfect square. If not, we can simply divide it into $\lceil \sqrt{n} \rceil$ chunks, each of size $\left\lceil \frac{n}{\lceil \sqrt{n} \rceil} \right\rceil$. We can round up n to the nearest multiple of $\lceil \sqrt{n} \rceil$ by introducing filler entries. In this way, the amount of padded entries we introduce is at most $O(\sqrt{n})$.

Supporting Unbounded Number of Queries. Our scheme can be easily extended to support unbounded number queries. One straightforward idea is to re-run the preprocessing every $Q = \sqrt{n} \ln \kappa\alpha(\kappa)$ queries, like Corrigan-Gibbs, Henzinger, and Kogan [CHK22], but it incurs long waiting time between each Q-size window of queries. Zhou et al. [ZLTS23] proposes the amortization idea to convert a query-bounded scheme to an unbounded scheme without long waiting time. This idea naturally fits our scheme.

We can distribute the preprocessing of the $(i+1)$-th window among the online phase of the i-th window. The scheme starts by running the preprocessing for the first window of Q queries. When the client starts the online phase of the i-th window, it also intitializes the local hints for the $(i+1)$-th window. Then, when the client receives the response of the j-th query in the i-th batch, the server also sends the j-th chunk of the DB. The client then executes the exact proprocesing algorithm as in the Q-bounded scheme and updates the local hint for the $(i+1)$-th window. At the end of the online phase of the i-th window, the client already makes a pass of all $\sqrt{n}$ chunks and finishes the preprocessing for the $(i+1)$-th window. Therefore, the client can start the online queries for the $(i+1)$-th window immediately after it finishes the i-th window!

From the efficiency side, the amortized time and communication cost are not changed. The client only doubles its local stroage and the server has no extra storage requirement.

2.5 Organization of the Remaining Sections

In the rest of the paper, we shall prove the above 1-server PIR scheme secure. To this end, we first give formal security definitions in Section 3. Next, in Section 4, we describe our implementation and show experimental results.

3 Formal Definitions and Security Proofs

3.1 Definitions

We define a single-server private information retrieval (PIR) scheme in the pre-processing setting. In a single-server PIR scheme, we have two stateful machines called the client and the server. The scheme consists of two phases:

- **Offline setup.** The offline setup phase is run only once up front. The client receives nothing as input, and the server receives a database $\mathsf{DB} \in \{0,1\}^n$ as input. The client sends a single message to the server, and the server responds with a single message. For simplicity, we assume the entries in DB are 1-bit[3].

[3]Our scheme can directly work with multi-bit entries.

- **Online queries.** This phase can be repeated multiple times. Upon receiving an index $x \in \{0, 1, \ldots, n-1\}$, the client sends a single message to the server, and the server responds with a single message. The client performs some computation and outputs an answer $\beta \in \{0, 1\}$.

Correctness. Given a database $\mathsf{DB} \in \{0,1\}^n$, where the bits are indexed $0, 1, \ldots, n-1$, the correct answer for a query $x \in \{0, 1, \ldots, n-1\}$ is the x-th bit of DB.

For correctness, we require that given a statistical security parameter κ and a computational security parameter λ, for any sufficiently large n and any Q, there exists a negligible function $\mathsf{negl}(\kappa)$, such that for any database $\mathsf{DB} \in \{0,1\}^n$, for any sequence of queries $x_1, x_2, \ldots, x_Q \in \{0, 1, ..., n-1\}$, an honest execution of the PIR scheme with DB and queries $x_1, x_2, \ldots, x_Q$, returns all correct answers with a probability at least $1 - \mathsf{negl}(\kappa) - \mathsf{negl}(\lambda)$.

Privacy. We now formally define privacy in the following experiment.

Definition 3.1 (Privacy of PIR)**.** We say that a single-server PIR scheme satisfies privacy iff there exists a probabilistic polynomial-time simulator $\mathsf{Sim}(1^\lambda, n)$, such that for any probabilistic polynomial-time adversary $\mathcal{A}$ acting as the server, any polynomially bounded n and Q, any $\mathsf{DB} \in \{0,1\}^n$, $\mathcal{A}$'s views in the following two experiments are computationally indistinguishable:

- Real: an honest client interacts with $\mathcal{A}(1^\lambda, n, \mathsf{DB})$ who acts as the server and may arbitrarily deviate from the prescribed protocol. In every online step $t \in [Q]$, $\mathcal{A}$ may adaptively choose the next query $x_t \in \{0, 1, \ldots, n-1\}$ for the client, and the client is invoked with x_t;
- Ideal: the simulated client $\mathsf{Sim}(1^\lambda, n)$ interacts with $\mathcal{A}(1^\lambda, n, \mathsf{DB})$ who acts as the server. In every online step, $\mathcal{A}$ may adaptively choose the next query $x_t \in \{0, 1, \ldots, n-1\}$, and Sim is invoked without receiving x_t.

3.2 Proofs

We now prove the privacy and correctness of our PIR scheme.

Theorem 3.2 (Privacy)**.** *Our PIR scheme satisfies privacy (i.e., Definition 3.1).*

Proof. Denote the distribution $\mathcal{D}_n$ as sampling a random set that draws a random element from each $\sqrt{n}$ chunk. We define the following simulator $\mathsf{Sim}(1^\lambda, n)$. Basically, whenever Sim is invoked, it outputs a random offset vector $\Delta \xleftarrow{\$} \{0, \ldots, \sqrt{n}-1\}^{\sqrt{n}-1}$. In other words, the Ideal experiment is as follows.

- *Offline.* $\mathcal{A}$ receives only the streaming starting signal.
- *Online.* for query i, the simulated client sends a uniformly random vector $\Delta \xleftarrow{\$} \{0, 1, \ldots, \sqrt{n}-1\}^{\sqrt{n}-1}$ to $\mathcal{A}$.

We define a hybrid experiment Hyb_1 as following:

- *Offline.* $\mathcal{A}$ receives only the streaming starting signal.
- *Online.* For each online round t, $\mathcal{A}$ chooses the query x_t. The client samples a random set $S \xleftarrow{\$} \mathcal{D}_n$ conditioned on $x_t \in S$. Let Δ be the offset vector of S and it sends $\Delta_{-\lfloor x_t/\sqrt{n} \rfloor}$ to the server (received by $\mathcal{A}$).

It should be straightforward to prove the distribution of $\mathcal{A}$'s views in Ideal and Hyb_1 are identical. In Hyb_1, since the $\mathcal{D}_n$ chooses a random element in each chunk independently, even conditioned on containing any particular x, the remaining elements are still independent and uniformly random within their chunks. Therefore, after puncturing x, the compacted offset vector's distribution is exactly uniformly random in $\{0, 1, \ldots, \sqrt{n}-1\}^{\sqrt{n}-1}$. So the views are indeed indentically distributed in these two experiments.

Now we define a hybrid experiment Hyb_2 as following:

- *Offline.* $\mathcal{A}$ receives only the streaming starting signal. The client samples random sets $S_1, \ldots, S_{M_1} \xleftarrow{\$} \mathcal{D}_n^{[M_1]}$.
- *Online.* For each online round t, $\mathcal{A}$ chooses the query x_t:
  1. The client finds the smallest index $j \in [M_1]$ that $x_t \in S_j$. Denote the set as S^*. If no such index is found, the client samples a set $S^* \xleftarrow{\$} \mathcal{D}_n$ conditioned on $x_t \in S^*$.
  2. Denote Δ as S^*'s offset vector. The client sends $\Delta_{\lfloor x_t/\sqrt{n} \rfloor}$ to the server.
  3. The client samples $S' \xleftarrow{\$} \mathcal{D}_n$ conditioned on $x_t \in S'$. If the client finds a set that contains x_t earlier, replace the j-th set in the local sets with S'.

We will prove the view of $\mathcal{A}$ in Hyb_1 and Hyb_2 are identically distributed by proving the following lemma.

Lemma 3.3. *In Hyb_2, for every online queries x_t, even conditioned on $\mathcal{A}$'s view over the first $t-1$ queries,*

- *The message $\mathcal{A}$ receives in this step is distributed as: sample $S \xleftarrow{\$} \mathcal{D}_n$ conditioned on $x_t \in S$ and output $\Delta_{-\lfloor \frac{x_t}{\sqrt{n}} \rfloor}$, where Δ is the offset vector of S.*
- *At the end of the t-th query, the client local sets $S_1, \ldots, S_{M_1}$ are identically distributed as $S_1, \ldots, S_{M_1} \xleftarrow{\$} \mathcal{D}_n^{M_1}$ even condtioned on the messages received by $\mathcal{A}$ during the first t-th queries.*

Proof. The proof is similar to the proof of Fact 7.3 in [SACM21].

Base case. At the end of the offline phase, indeed $S_1, \ldots, S_{M_1}$ are indeed distributed as $S_1, \ldots, S_{M_1} \xleftarrow{\$} \mathcal{D}_n^{[M_1]}$. The set found by the client are indeed distributed as $S \xleftarrow{\$} \mathcal{D}_n$ subject to $x \in S$ (even when the client does not find it in the first M_1 sets and generates it on-the-fly).

Inductive case. Suppose that at the end of the $t-1$-th step, the client's local sets $S_1, \ldots, S_{M_1}$ are distributed as $S_1, \ldots, S_{M_1} \xleftarrow{\$} \mathcal{D}_n^{[M_1]}$ even when conditioned on $\mathcal{A}$'s view in the first $t-1$ steps. We now prove that the stated claims hold for t. Let x_t be the query chosen by $\mathcal{A}$ depending on the first $t-1$ queries' messages. For $i \in [M_1]$, define α_i be the probability that if $S_1, \ldots, S_{M_1}$ are i.i.d sampled from $\mathcal{D}_n$, the first set that contains x is i. Let $\alpha_{M_1+1} = 1 - \sum_{i \in [M_1]} \alpha_i$.

Consider the following experiment Expt:

- The client samples $u \in [M_1 + 1]$ such that $u = i$ with probability α_i.
- $\forall j < u$, the client samples $S_j \xleftarrow{\$} \mathcal{D}_n$ subject to $x_t \notin S_j$.
- For u, the client samples $S_u \xleftarrow{\$} \mathcal{D}_n$ subject to $x_t \in S$. The client sends $\Delta_{-\lfloor x_t/\sqrt{n} \rfloor}$ to the server, where Δ is the offset vector of S.

- For $j \in [u+1, M_1]$, the client samples $S_j \xleftarrow{\$} \mathcal{D}_n$.
- The client samples $S'_u \xleftarrow{\$} \mathcal{D}_n$ subject to $x_t \in S'_u$. If $u \le M_1$, the client replaces S_u with S'_u.

The main random variables sampled in those two cases are $(S_1, \ldots, S_{M_1}, u, S'_u)$ where $S_1, \ldots, S_{M_1}$ are the sets at the beginning of the t-th query, u is the index of the first set containing x_t, and $S_1, \ldots, S_{u-1}, S'_u, S_{u+1}, \ldots, S_{M_1}$ will be the local sets at the end of the t-th query. In Hyb_2, by the induction hypothesis, $S_1, \ldots, S_{M_1}$ are i.i.d. sampled from $\mathcal{D}_n$. Then u is selected as the first set's index that contains x_t and its distribution will follow $\Pr[u = i] = \alpha_i$. Finally, it samples $S'_u \xleftarrow{\$} \mathcal{D}_n$. In Expt, the sampling order is changed: it first samples u, then samples $S_1, \ldots, S_{M_1}$ conditioned on u, then samples S'_u. By the definition of $\alpha_1, \ldots, \alpha_{M+1}$, we know the joint distribution of $(S_1, \ldots, S_{M_1}, u)$ are the same in both experiments. Also, S'_u is always just sampled from $\mathcal{D}_n$ subject to $x_t \in S'_u$. Therefore, the marginal distributions of $(S_1, \ldots, S_{M_1}, S'_u)$ are the same in both experiment. Now we look at Expt. The message received by $\mathcal{A}$ fully depends on S_u (with no dependency on u) and S_u's marginal distribution is exactly $S_u \xleftarrow{\$} \mathcal{D}_n$ subject to $x \in S_u$. So we prove that Hyb_2 satisfies the first property in the statement. From the definition of Expt, the marginal distribution of $(S_1, \ldots, S_{u-1}, S'_u, S_{u+1}, \ldots, S_{M_1})$ will actually be $\mathcal{D}_n^{M_1}$. Thus, we prove Hyb_2 also satisfies the second property in the statements. □

Notice that Hyb_2 is already very close to the real experiment. We now define the final hybrid experiment Real^* as following:

- *Offline.* $\mathcal{A}$ receives only the streaming starting signal. The client samples random sets $S_1, \ldots, S_{M_1} \xleftarrow{\$} \mathcal{D}_n^{[M_1]}$ and also $\overline{S}_{i,j} \xleftarrow{\$} \mathcal{D}_n$ for $i \in \{0, 1, \ldots \sqrt{n} - 1\}$, $j \in [M_2]$.
- *Online.* For each round t, $\mathcal{A}$ chooses the query x_t:
  1. The client finds the smallest index $j \in [M_1]$ that $x_t \in S_j$. Denote the set as S^*. If no such index is found, the client samples a set $S^* \xleftarrow{\$} \mathcal{D}_n$ conditioned on $x_t \in S^*$.
  2. Denote $i^* = \lfloor x_t / \sqrt{n} \rfloor$. Let Δ be S^*'s offset vector. The client sends Δ_{-i^*} to the server.
  3. If there is an unconsumed set in $\overline{S}_{i^*,1}, \ldots, \overline{S}_{i^*,M_2}$, say $\overline{S}_{i^*,j}$, the client consumes it and set $S' = (\overline{S}_{i^*,j} / \{\overline{S}_{i^*,j}[i^*]\}) \cup \{x_t\}$. Otherwise, The client samples $S' \xleftarrow{\$} \mathcal{D}_n$ conditioned on $x_t \in S'$. If the client finds a set that contains x_t earlier, replace the j-th set in the local sets with S'.

The view of $\mathcal{A}$ in Hyb_2 and Real^* is identically distributed – the experiments only differ in the refreshing phase. In Hyb_2, the client always replaces the set with a freshly generated set S' subject to the query index x_t is contained. In Real^*, the client first tries to find an unconsumed local backup set S' (which has distribution $\mathcal{D}_n$) and manually forces x_t into it. Otherwise it is the same as Hyb_2. Notice that $\mathcal{D}_n$ samples the element in each chunk independently. Therefore, even in Real^* that x_t is forced into the set, the elements in other chunks are still uniformly random. Therefore, the distribution of S' is identical in both experiments, and $\mathcal{A}$ has the same view in these experiments.

Finally, Real^* is just a rewrite of Real throwing out the terms that we are not interested and replacing the PRF with real randomness when picking the elements. By a straightforwad reduction to the pseudorandomness of the PRF, Real^* and Real are computationally indistinguishable. □

Theorem 3.4 (Correctness)**.** *Assume n is bounded by $\mathsf{poly}(\lambda)$ and $\mathsf{poly}(\kappa)$. Let $\alpha(\kappa)$ be any super-constant function, i.e., $\alpha(\kappa) = \omega(1)$. Setting $M_1 = \sqrt{n} \ln \kappa\alpha(\kappa)$, $M_2 = 3 \ln \kappa\alpha(\kappa)$, all the*

$Q = \sqrt{n} \ln \kappa\alpha(\kappa)$ *queries will be answered correctly with probability at least* $1 - \mathsf{negl}(\lambda) - \mathsf{negl}(\kappa)$ *for some negligible function* $\mathsf{negl}(\cdot)$.

Proof. Recall that in our full scheme, the server will first sample a pseudorandom permutation(PRP) to permute the database upfront and the client will download the key from the server. Replacing the PRP with a true random permutation only affects the failure probability by a negligible amount, $\mathsf{negl}(\lambda)$. We also assume that the client does not make any duplicative queries for those Q queries. Therefore, taking the randomness of the permutation, we can view all Q queries are randomly sampled from $\{0, 1, \ldots, n-1\}$ without replacement.

We assume the client uses a true random oracle to sample the sets, instead of a PRF. Due to the pseudorandomness of the PRF, this assumption will not affect the failure probability by a negligible amount, $\mathsf{negl}(\lambda)$.

There are only two types of events that causes failures: 1) the client cannot find a set that contains the online query index. 2) the client runs out of hints in a backup group;

We analyze the second type of failure events – it only happens when the client makes more than M_2 queries in one group. Since the client is making $\sqrt{n} \ln \kappa\alpha(\kappa)$ queries and there are $\sqrt{n}$ groups, we can use a standard balls-into-bins argument. For $t \in [Q], i \in \{0, 1, \ldots, \sqrt{n} - 1\}$, define the random variables $Y_{t,i} \in \{0, 1\}$ such that $Y_{t,i} = 1$ if and only if the t-th query locates in the i-th chunk. Denote $X_i = Y_{1,i} + \cdots + Y_{t,i}$ be the number of queries located in the i-th chunk. We know $\mathbb{E}[Y_{t,i}] = 1/\sqrt{n}$ and $\mathbb{E}[X_i] = \ln \kappa\alpha(\kappa)$. Notice that we are taking the randomness of the permutation and the queries do not have duplication, so $Y_{1,1}, \ldots, Y_{Q,1}$ are negatively correlated. With the Chernoff bound for negatively correlated variables, we know that

$$\Pr[X_1 \geq (1+2) \ln \kappa\alpha(\kappa)] \leq \exp\left(\frac{-2^2}{2+2} \ln \kappa\alpha(\kappa)\right) = \kappa^{-\Theta(\alpha(\kappa))}.$$

Taking the union bound over all $\sqrt{n}$ chunks, the failure probability is bounded by $\sqrt{n} \cdot \kappa^{-\Theta(\alpha(\kappa))}$, which is a negligible function of κ.

For the first type of failure events, by Lemma 3.3, the local sets $S_1, \ldots, S_{M_1}$ (i.e., the set represented by the keys) will be identically distributed as $\mathcal{D}_n^{M_1}$ and each set will contain the query with probability $1/\sqrt{n}$. So for a particular query x, the probability of no set containing x is

$$\begin{aligned}
&\left(1 - 1/\sqrt{n}\right)^{M_1} \\
=&\left(1 - 1/\sqrt{n}\right)^{\sqrt{n} \ln \kappa \cdot \alpha(\kappa)} \\
\leq&\left(1/e\right)^{\ln \kappa\alpha(\kappa)} \\
=&\kappa^{-\alpha(\kappa)}.
\end{aligned}$$

With the union bound, for all $\sqrt{n} \ln \kappa\alpha(\kappa)$ queries, the probability of any query cannot find a set is bounded by $\sqrt{n} \ln \kappa\alpha(\kappa) \cdot \kappa^{-\alpha(\kappa)}$, which is a negligible function of κ since n is bounded by $\mathsf{poly}(\kappa)$.

Then, there exists some negligible function $\mathsf{negl}(\cdot)$ that all the $\sqrt{n} \ln \kappa\alpha(\kappa)$ queries are answered correctly with probability at least $1 - \mathsf{negl}(\lambda) - \mathsf{negl}(\kappa)$.

□

With the amortization technique we discussed before, we can show the following efficiency theorem:

Theorem 3.5 (Efficiency)**.** *Let $\alpha(\kappa)$ be any super-constant function, i.e., $\alpha(\kappa) = \omega(1)$. The single-server PIR scheme only need an one-time offline phase and supports unbounded number of queries. It achieves the following performance bounds:*

- $O_\lambda(\sqrt{n}\log\kappa\cdot\alpha(\kappa))$ *client storage and no additional server storage;*
- *Offline Phase:*
  - $O_\lambda(n\log\kappa\cdot\alpha(\kappa))$ *client time and* $O(n)$ *server time;*
  - $O(n)$ *communication;*
- *Each Online Query:*
  - *Expected* $O_\lambda(\sqrt{n})$ *client time and* $O(\sqrt{n})$ *server time;*
  - $O(\sqrt{n})$ *communication.*

Proof. Let's first consider the scheme that supports $Q = \sqrt{n}\ln\kappa\alpha(\kappa)$ online queries.

The client has $O(\sqrt{n}\log\kappa\cdot\alpha(\kappa))$ local hints and each hint stores a parity and a PRF key. Also, during the offline phase, the client will only store one $\sqrt{n}$-size chunk of the DB at any time. So the client's storage is $O_\lambda(\sqrt{n}\log\kappa\cdot\alpha(\kappa))$. The server only needs to store the DB.

For the offline phase, the client downloads the whole DB, so the communication cost is $O(n)$. For each chunk, the client needs to enumerate all $O(\sqrt{n}\log\kappa\cdot\alpha(\kappa))$ local hints and update them. So in total, the client offline computation time is $O(n\log\kappa\cdot\alpha(\kappa))$.

For the online phase, the client needs to search for the hint that contains the query. Since each set contains the query with probability $1/\sqrt{n}$ and each membership testing takes $O_\lambda(1)$ time, the expected searching time is $O_\lambda(\sqrt{n})$. The expanding and puncturing operations all take $O_\lambda(\sqrt{n})$ time. The client then sends an $O(\sqrt{n})$-sized vector to the server. The server uses the **PossibleParities** algorithms to compute the response and takes $O(\sqrt{n})$ time. The client downloads the response of size $O(\sqrt{n})$ and gets the answer. The refreshing time for each query is $O(1)$.

Finally, to support unbounded queries, the client needs to amortize the work of the next offline phase of the original Q-bounded scheme to the last Q queries. Since $Q = \sqrt{n}\ln\kappa\alpha(\kappa)$, the amortized asymptotic computation and communication costs remain the same. Also, the client stores two sets of local hints that it uses one for the current online phase and it prepares another one for the next Q queries. It only brings 2x cost to the local storage. The server does not need to have any additional storage.

□

4 Evaluation

Our evaluation aims to answer the following questions:

1. How does Piano perform compared to a state-of-art single server PIR scheme (SimplePIR [HHCG+22])? (Section 4.3)
2. How does Piano perform compared to a non-private retrieval baseline? (Section 4.4)

In particular, we compare to the SimplePIR protocol [HHCG+22] which is the current state-of-art single-server PIR implementation and is faster than all other single-server PIR schemes by at least an order of magnitude. We refer the reader to their paper for details, but crucially, their scheme requires a linear scan on the server for each server, like many other practical PIR implementations.

4.1 Implementation and Practical Considerations

We implement PIANO in Golang in approximately 800 lines of code. We utilize AES-NI hardware instruction for fast PRF evaluations.

Parallelization. We parallelize the preprocessing on the client side, which is the main bottleneck of the setup phase. All server-side and online computation is performed on a single thread.

Parameters. Finally, we note that the performance of our scheme is more affected by the size of each set rather than the size of each chunk. To this end, we set the chunk size to be $2\sqrt{n}$ and round it up to the nearest power of 2, which makes the modulo operation more efficient. The set size is computed accordingly. It does not affect the theoretical asymptotics of our protocol. We set $Q = \sqrt{n} \ln n$. We set the statistical security parameter κ to 40 and computational security parameter λ to 128. We adjust M_1, M_2 accordingly that the failure probability for all Q queries is bounded by $2^{-\kappa} = 2^{-40}$, matching the same failure probability as SimplePIR [HHCG+22]. We use AES-128 to instantiate the PRF and use 128-bit PRF keys.

4.2 Evaluation Setup

We evaluate PIANO and baseline schemes on 2 AWS m5.8xlarge instances with 128GB of RAM. For our local area network experiments, we run the PIR scheme on a single machine. This simulates a scenario where the network is not the bottleneck. We also evaluate our scheme over a wide-area network. In this case, we place the client server on the west coast, and the server machine on the east coast. All communication is performed over TLS on a 2Gbps network latency. The round-trip-time is around 60ms. All query costs are computed as the average over one thousand queries. We use the open-source implementation of SimplePIR provided by Henzinger et al. [HHCG+22]. We also implement a non-private database access scheme as the baseline that does not include caching nor load balancing.

4.3 Experiments in a Local-Area Network

We first compare our protocol to the SimplePIR protocol for 1GB and 2GB databases of 8-byte entries. Because the open-source SimplePIR implementation does not support parallelization nor connections across servers, we only compare the protocols run on a single machine. We analyze the effect of network latency on our protocol in the following section. We also run our scheme with a 100GB database with 1.6 billion 64-byte entries. To the best of our knowledge, this is by far the largest database ever supported by any implementation of a single-server PIR scheme. Because the implementation of SimplePIR does not support databases of this size, we extrapolate the results by running their scheme for 1GB and 2GB sized databases of 64-byte entries, and extrapolating their performance to 100GB based on the asymptotic performance discussed in their paper.

Metrics and two modes of operation. Table 1 shows the costs of the queries as well as the one-time pre-processing. For the query costs, we divide it into two parts, the online costs and the amortized offline costs. The former is on the critical path of the perceived response time, and the latter is the additional maintainence work needed when we deamortize the periodic preprocessing costs over the queries. In practice, there are two ways to run our scheme. The first method is to perform the preprocessing upfront *only once*, and the subsequent periodic pre-processing costs are deamortized to the queries in each window. The second method is to periodically rerun the

	1GB($n = 2^{27}$)		**2GB**($n = 2^{28}$)		**100GB**($n \approx 1.68 \times 10^9$)	
	SimplePIR	Piano	SimplePIR	Piano	SimplePIR(*)	Piano
			Preprocessing			
Client time	293s	1438s/243s	608s	3369s/563s	425min	356min/59min
Communication	123MB	1GB	173MB	2GB	1.2GB	100GB
			Per query			
Online Time	131.6ms	7.9ms	219.5ms	9.0ms	10.9s	33.3ms
Online Comm.	238KB	64KB	338KB	128KB	2.3MB	900KB
Am. Offline Time	1.4 ms	6.6/1.1ms	2.9ms	10.6/1.7ms	29.6ms	24.6ms/4.1ms
Am. Offline Comm.	0.6KB	4.9KB	0.6KB	6.6KB	1.4KB	120.5KB
Client Storage	123MB	66MB	173MB	75MB	1.2GB	719MB

Table 1: Performance of our scheme and SimplePIR on 1GB, 2GB and 100GB sized databases. The 1GB and 2GB databases have 8-byte entries and the 100GB database has 64-byte entries. For preprocessing times in the format of 1438s/243s, the former is with a single thread, and the latter is with 8 threads. "Am." is an abbreviation of "Amortized". "Comm." stands for communication cost. We report the online costs as well as the offline costs amortized over $Q = \sqrt{n} \ln n$ queries. *The results for SimplePIR with the 100GB database are extrapolated since their implementation cannot directly support such a large database.

pre-processing phase, e.g., at night or during periods of inactivity — in this case, the query phase need not pay the "amortized offline time" and "amortized offline communication".

Query costs. As seen in Table 1, our protocol outperforms SimplePIR in all online metrics, including client storage, communication, and online querying time. In particular, for medium-sized databases (1GB/2GB), we outperform SimplePIR by 16.7x - 24.4x in terms of online querying latency. This performance gain stems from the fact that our online computation is sublinear in the size of the database, while SimplePIR is fundamentally limited by the linear scan required of their protocol. As the database grows larger, the performance gap further increases. For the 100GB database, Piano only takes 33.3ms for the online query. On the other hand, the extrapolated online time for SimplePIR is 10.9s, such that we see nearly a 330× performance gap.

Preprocessing costs. Preprocessing costs depend on the size of each entry. When the per-entry size is bigger, our preprocessing is faster than SimplePIR (see the 100GB case where the entry size is 64 bytes). When the per-entry size is smaller, our preprocessing is slower than SimplePIR. In particular, Piano has a quasi-linear preprocessing phase that it takes $O(n \log \kappa\alpha(\kappa))$ PRF evaluations and $O(n \log \kappa\alpha(\kappa))$ xor operations between database entries. We observe that the PRF evaluations are the computation bottleneck when the entry size is not too big (e.g., 64 bytes or less). Therefore, our scheme's concrete performance depends more on the number of entries, rather than the per-entry size.

The table also shows the effect of the parallelization for preprocessing costs (and similarly for amortized offline costs during the query phase). Since client computation is the bottleneck for the preprocessing, parallelizing the work using 8 threads significantly improves the running time. For example, for a 2GB database, parallelization with 8 threads improves the client's preprocessing time from 3369s to 563s.

4.4 Experiments over a Wide-Area Network

Next, we report our results for an experiment conducted over a wide-area network. Recall that the round-trip network latency is around 60ms and the network bandwidth is 2Gbps (see Section 4.2). The effects of this added network latency are seen in Table 2. Because the open-source SimplePIR implementation does not support connections across multiple machines, we extrapolate a *lower bound* for their querying time based on the summation of thee extrapolated numbers in the previous section and the network latency.

	2GB($n = 2^{28}$)			**100GB**($n \approx 1.68 \times 10^9$)		
	Non-Private	SimplePIR	Piano	Non-Private	SimplePIR	Piano
			Preprocessing			
Client Time	-	608s	3331s/565s	-	425min	395min/76min
Communication	-	173MB	2GB	-	1.2GB	100GB
			Per query			
Online Time	59.8ms	279.3ms	70.7ms	61.0ms	10.9s	99.0ms
Online Comm.	16B	338KB	128KB	72B	2.3MB	900KB
Am. Offline Time	-	1.9ms	10.4ms/1.8ms	-	29.6ms	27.2ms/5.3ms
Am. Offline Comm.	-	0.6KB	6.6KB	-	1.4KB	120.5KB
Client Storage	-	173MB	75MB	-	1.2GB	719MB

Table 2: Performance of our scheme, SimplePIR and the non-private baseline on 2GB and 100GB sized databases. The 2GB database has 8-byte entries and the 100GB database has 64-byte entries. Numbers of the format 10.4ms/1.8ms denote the performance with a single thread and 8 threads, respectively. "Comm." stands for communication cost. "Am." stands for "amortized".

When compared to the non-private baseline, our protocol has a $18\% - 62\%$ latency overhead. SimplePIR, on the other hand, has a 4.6x - 178.7x latency overhead.

5 Additional Related Work

In this section, we provide some additional comparisons with related work.

Single-server PIR schemes. Table 3 compares Piano with existing single-server PIR schemes. Although our paper primarily focuses on enhancing the practical performance of Private Information Retrieval (PIR), our proposed scheme is also of interest from a theoretical perspective. Notably, it is the first single-server PIR scheme that relies solely on one-way functions (OWF) and has sublinear server computation.

Early theoretical works aimed at improving the communication cost of PIR, such as the studies by Chachin, Micali and Stadler [CMS99], Yan-Cheng Chang [Cha04], and Genry and Ramzan [GR05].

Beimel, Ishai and Malkin [BIM00] proved an important lower bound that dictates the per-query time of any PIR scheme without preprocessing to be $\Omega(n)$. Inspired by their work, many subsequent studies followed the "pre-processing" model to achieve amortized sublinear per-query time.

In the "global-preprocessing" model, also known as Doubly Efficient PIR (DEPIR), the server first preprocesses the database, and subsequently, it can answer queries with sublinear computation time. However, early works [CHR17, BIPW17] in this area relied on non-standard assumptions or even ideal obfuscation. A recent breakthrough study by Lin, Mook and Wichs presented a method to construct DEPIR based on the standard RingLWE assumption. In their approach, for any constant $\varepsilon > 0$, the server preprocesses the database and stores a data structure of size $\widetilde{O}_\lambda(n^{1+\varepsilon})$.

Table 3: **Comparison of single-server PIR schemes.** m is the number of clients, n is the database size, d is the dimension of the hypercube representation of the DB, $\epsilon \in (0,1)$ is some suitable constant. 'Comm." means communication per query. "CRA" means the composite residuosity assumption, ϕ-hiding is a number-theoretic assumption described in [CMS99], "OLDC" means oblivious locally decodable codes, "VBB" means virtual-blackbox obfuscation, and "OWF" means one-way function. The extra space denotes the client's extra storage, except for the schemes based on OLDC and also Lin, Mook and Wichs [LMW22], where the server stores the extra storage.

Scheme	**Assumpt.**	**Comm.**	**Per-query time**	**Extra space**
Theoretical Single-server PIR Schemes				
Standard [Cha04, CMS99, GR05]	CRA or ϕ-hiding or LWE	$\widetilde{O}(1)$	$O(n)$	0
[CHR17, BIPW17]	OLDC	n^ϵ	n^ϵ	mn
[BIPW17]	OLDC, VBB	n^ϵ	n^ϵ	n
[LMW22]	RingLWE	$\mathsf{poly}((\log n)^{1/\varepsilon})$	$\mathsf{poly}((\log n)^{1/\varepsilon})$	$\widetilde{O}_\lambda(n^{1+\varepsilon})$
[CK20]	LWE	$\widetilde{O}_\lambda(\sqrt{n})$	$\widetilde{O}_\lambda(n)$	$\widetilde{O}_\lambda(\sqrt{n})$
[CHK22]	LWE	$\widetilde{O}_\lambda(\sqrt{n})$	$\widetilde{O}_\lambda(\sqrt{n})$	$\widetilde{O}_\lambda(\sqrt{n})$
[ZLTS23, LP22]	LWE	$\widetilde{O}_\lambda(1)$	$\widetilde{O}_\lambda(\sqrt{n})$	$\widetilde{O}_\lambda(\sqrt{n})$
Practical Single-server PIR Schemes (with implementations)				
XPIR($d=2$) [MBFK16]	LWE	$\widetilde{O}_\lambda(\sqrt{n})$	$O(n)$	$\widetilde{O}_\lambda(1)$
PSIR [PPY18]	LWE	$\widetilde{O}_\lambda(\sqrt{n})$	$O(n)$	$\widetilde{O}_\lambda(\sqrt{n})$
FastPIR [AYA+21]	LWE	$\widetilde{O}_\lambda(n)$	$O(n)$	$O_\lambda(1)$
OnionPIR [MCR21]	LWE	$\widetilde{O}_\lambda(1)$	$O(n)$	$\widetilde{O}_\lambda(\sqrt{n})$
Spiral [MW22]	LWE	$\widetilde{O}_\lambda(1)$	$O(n)$	$O(1)$
FrodoPIR [DPC22]	LWE	$\widetilde{O}_\lambda(\sqrt{n})$	$O(n)$	$\widetilde{O}_\lambda(\sqrt{n})$
SimplePIR [HHCG+22]	LWE	$\widetilde{O}_\lambda(\sqrt{n})$	$O(n)$	$\widetilde{O}_\lambda(\sqrt{n})$
Ours	OWF	$O(\sqrt{n})$	$\widetilde{O}_\lambda(\sqrt{n})$	$\widetilde{O}_\lambda(\sqrt{n})$

Later, the server can answer queries in $\mathsf{poly}((\log n)^{1/\varepsilon})$ time, and the communication cost is also $\mathsf{poly}((\log n)^{1/\varepsilon})$, where poly is a fixed polynomial for all n and ε.

Our scheme falls under the "client-preprocessing" model, also known as the subscription model. Corrigan-Gibbs and Kogan [CK20] were the first to present a construction under this model with $O(n)$ offline time and $\widetilde{O}_\lambda(\sqrt{n})$ online time. However, their scheme only supports a single query. Later, Corrigan-Gibbs, Henzinger and Kogan [CHK22] showed how to transform a single-query scheme into a $\sqrt{n}$-query scheme with polylogarithmic overhead using fully homomorphic encryption(FHE). Zhou et al. [ZLTS23] and Lazzaretti and Papamanthou [LP22] further extended the idea and compressed the communication cost to polylogarithmic levels.

In terms of practical schemes, all the previous works supporting adaptive queries had $\tilde{O}(n)$ per-query computation time. Furthermore, most of them relied on homomorphic encryption (usually not just linear homomorphic encryption) and required the LWE assumption. XPIR [MBFK16] implemented a single-server PIR scheme that only consumes $\widetilde{O}_\lambda(\sqrt{n})$ per-query communication cost. PSIR [PPY18] utilized client-side preprocessing to reduce the online cryptographic operation number to $\widetilde{O}(\sqrt{n})$, but the server still needs to perform $O(n)$ plaintext operations. FastPIR [AYA+21] made concrete improvements to online time, but it comes at the cost of $O(n)$ per-query communication for the client. OnionPIR [MCR21], on the other hand, choosed homomorphic encryption parameters carefully to compress communication. Among the state-of-the-art single PIR schemes, Spiral [MW22], FrodoPIR [DPC22], and SimplePIR [HHCG+22] are noteworthy. Spiral [MW22] combined two different homomorphic encryption schemes to control noise in the ciphertext, thereby achieving polylogarithmic communication. Meanwhile, FrodoPIR [DPC22] and SimplePIR [HHCG+22] shared a similar idea that in the evaluation of the Regev's HE scheme, most of the computation can be performed without knowing the message upfront and thus can be preprocessed. Their schemes require an one-time offline setup where the client downloads and stores the query-irrelevant parts of the HE evaluation in advance. As a result, for each online query, the server only needs to compute the query-relevant part, and the cost is almost the same as plaintext evaluation over the entire DB. SimplePIR [HHCG+22] had the best performance and claimed that online query time is already limited by the server's memory I/O speed. However, all these schemes still use linear time in the online phase.

Batch PIR schemes. Pioneered by Ishai et. al [IKOS04], Batch PIR schemes [AS16, ACLS18, MR22, LLWR22] are designed for batched queries. If a client submits a batch of Q parallel queries to the server, the server's computation cost can be amortized to $\tilde{O}(n/Q)$ per query, even though the server still performs $O(n)$ computation for the entire batch. However, batch PIR schemes have two main drawbacks. First, a client must make many parallel queries simultaneously to amortize the computation cost. In many practical applications, the client may wish to adaptively decide their following queries based on previous querying results. Asymptotically, only when the client makes $O(\sqrt{n})$ parallel queries, their amortized computation time can match our scheme. Second, these schemes still require the server to run some form of homomorphic encryption evaluations on the entire DB. This still incurs $O(n)$ computation per batch and the overall latency is significant. In contrast, our scheme only requires the server to perform $O(\sqrt{n})$ plaintext evaluation.

From a practical standpoint, as mentioned in Henzinger et al. [HHCG+22], the state-of-the-art batch PIR scheme, SealPIR [ACLS18], has 100 times worse throughput than SimplePIR [HHCG+22].

Multi-server PIR schemes. The multi-server PIR schemes assume there are mulitple non-colluding servers and each one of them stores a copy of the DB. This assumption was first shown to improve the communication cost to $O\left(n^{1/3}\right)$ [CGKS95, CKGS98] and later schemes based

on [GI14, BGI16] further improved the cost to be polylogarithmic.

Corrigan-Gibbs and Kogan [CK20,KCG21] proposed the client-side preprocessing idea to achieve $\widetilde{O}_\lambda(\sqrt{n})$ amortized per-query time under the two-server model with $\widetilde{O}_\lambda(\sqrt{n})$-size per-query communication and $\widetilde{O}_\lambda(\sqrt{n})$ client side storage. Shi et al. [SACM21] and TreePIR [LP23] by Lazzaretti and Papamathou further extended this idea and achieved polylogarithmic per-query communication.

The sublinear schemes in the multi-server model are practical. The PRP-based PIR [CK20] is implemented by Ma et al. [MZRA22]. Checklist [KCG21] and TreePIR [LP23] also provided implmentations.

TreePIR reported the best performance among these implementations, providing two implementations, one with polylogarithmic per-query communication cost by invoking a recursive scheme and another with $O(\sqrt{n})$ per-query communication cost without the recursion. For an 8GB database with 2^{28} entries, the best amortized online time results reported in TreePIR are 23ms for the non-recursive scheme and 84ms for the recursive scheme. For comparison, our scheme has an amortized 20ms per-query time under the same setting with four times more local storage. The blowup of the local storage is mainly because of the backup hints and the deamortization of the setup phase, which are inherently required for the single-server setting.

6 Conclusion

We propose an extremely simple single-server PIR scheme called Piano. Unlike previous *practical* single-server PIR schemes, Piano achieves sublinear server computation. This allows us to scale Piano to database sizes that are much larger than previous implementations. For example, for a 100GB database over a coast-to-coast link, Piano achieves 99ms response time, which is only 1.62× slowdown w.r.t. a non-private baseline. Our work pushes the frontier of the practical PIR landscape, and opens up possibilities for new applications, especially ones that involve larger databases.

Acknowledgment

We gratefully acknowledge Waqar Aqeel, Zakir Durumeric, Marwan Fayed, Geoff Huston, Bruce Maggs, Paul Pearce, and Justine Sherry for helpful discussions about DNS. We thank Shuoshuo Chen for providing us computation resources. This work is in part supported by a grant from ONR, a grant from the DARPA SIEVE program under a subcontract from SRI, a gift from Cisco, Samsung MSL, NSF awards under grant numbers CIF-1705007, 2128519 and 2044679, and a Packard Fellowship.

References

[ACLS18] Sebastian Angel, Hao Chen, Kim Laine, and Srinath T. V. Setty. PIR with compressed queries and amortized query processing. In *S&P*, 2018.

[AS16] Sebastian Angel and Srinath TV Setty. Unobservable communication over fully untrusted infrastructure. In *OSDI*, volume 16, pages 551–569, 2016.

[AYA+21] Ishtiyaque Ahmad, Yuntian Yang, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta. Addra: Metadata-private voice communication over fully untrusted infrastructure. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI2021, July 14-16, 2021*, 2021.

[BGI16] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In *CCS*, 2016.

[BIM00] Amos Beimel, Yuval Ishai, and Tal Malkin. Reducing the servers computation in private information retrieval: Pir with preprocessing. In *CRYPTO*, pages 55–73, 2000.

[BIPW17] Elette Boyle, Yuval Ishai, Rafael Pass, and Mary Wootters. Can we access a database both locally and privately? In *TCC*, 2017.

[BKM17] Dan Boneh, Sam Kim, and Hart William Montgomery. Private puncturable PRFs from standard lattice assumptions. In *EUROCRYPT*, pages 415–445, 2017.

[BLW17] Dan Boneh, Kevin Lewi, and David J. Wu. Constraining pseudorandom functions privately. In *PKC*, 2017.

[BS80] Jon Louis Bentley and James B. Saxe. Decomposable searching problems I: static-to-dynamic transformation. *J. Algorithms*, 1(4):301–358, 1980.

[BTVW17] Zvika Brakerski, Rotem Tsabary, Vinod Vaikuntanathan, and Hoeteck Wee. Private constrained prfs (and more) from LWE. In *TCC*, 2017.

[CC17] Ran Canetti and Yilei Chen. Constraint-hiding constrained PRFs for NC^1 from LWE. In *EUROCRYPT*, pages 446–476, 2017.

[CG97] Benny Chor and Niv Gilboa. Computationally private information retrieval. In *STOC*, 1997.

[CGKS95] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *FOCS*, 1995.

[Cha04] Yan-Cheng Chang. Single database private information retrieval with logarithmic communication. In *ACISP*, 2004.

[CHK22] Henry Corrigan-Gibbs, Alexandra Henzinger, and Dmitry Kogan. Single-server private information retrieval with sublinear amortized time. In *Eurocrypt*, 2022.

[CHR17] Ran Canetti, Justin Holmgren, and Silas Richelson. Towards doubly efficient private information retrieval. In *TCC*, 2017.

[CK20] Henry Corrigan-Gibbs and Dmitry Kogan. Private information retrieval with sublinear online time. In *EUROCRYPT*, 2020.

[CKGS98] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *J. ACM*, 45(6):965–981, November 1998.

[CMS99] Christian Cachin, Silvio Micali, and Markus Stadler. Computationally private information retrieval with polylogarithmic communication. In *EUROCRYPT*, pages 402–414, 1999.

[DCIO98] Giovanni Di-Crescenzo, Yuval Ishai, and Rafail Ostrovsky. Universal service-providers for database private information retrieval. In *PODC*, 1998.

[dCP22] Leo de Castro and Antigoni Polychroniadou. Lightweight, maliciously secure verifiable function secret sharing. In *Eurocrypt*, 2022.

[DG16] Zeev Dvir and Sivakanth Gopi. 2-server pir with subpolynomial communication. *J. ACM*, 63(4), 2016.

[DHS14] Daniel Demmler, Amir Herzberg, and Thomas Schneider. Raid-pir: Practical multi-server pir. In *CCSW*, 2014.

[DPC22] Alex Davidson, Gonçalo Pestana, and Sofía Celi. Frodopir: Simple, scalable, single-server private information retrieval. Cryptology ePrint Archive, Paper 2022/981, 2022. https://eprint.iacr.org/2022/981.

[Gas04] William I. Gasarch. A survey on private information retrieval. *Bulletin of the EATCS*, 82:72–107, 2004.

[GI14] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In *Advances in Cryptology – EUROCRYPT 2014*, 2014.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 1996.

[Gol87] O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *STOC*, 1987.

[GR05] Craig Gentry and Zulfikar Ramzan. Single-database private information retrieval with constant communication rate. In *ICALP*, 2005.

[Hen16] Ryan Henry. Polynomial batch codes for efficient IT-PIR. *Proc. Priv. Enhancing Technol.*, 2016(4):202–218, 2016.

[HH17] Syed Mahbub Hafiz and Ryan Henry. Querying for queries: Indexes of queries for efficient and expressive IT-PIR. In *CCS*, 2017.

[HHCG+22] Alexandra Henzinger, Matthew M. Hong, Henry Corrigan-Gibbs, Sarah Meiklejohn, and Vinod Vaikuntanathan. One server for the price of two: Simple and fast single-server private information retrieval. Cryptology ePrint Archive, Paper 2022/949, 2022. https://eprint.iacr.org/2022/949.

[IKOS04] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Batch codes and their applications. In *STOC*, 2004.

[IKOS06] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Cryptography from anonymity. In *FOCS*, pages 239–248, 2006.

[KCG21] Dmitry Kogan and Henry Corrigan-Gibbs. Private blocklist lookups with checklist. In *Usenix Security*, 2021.

[KO97] E. Kushilevitz and R. Ostrovsky. Replication is not needed: single database, computationally-private information retrieval. In *FOCS*, 1997.

[KU11] Kiran S Kedlaya and Christopher Umans. Fast polynomial factorization and modular composition. *SIAM Journal on Computing*, 40(6):1767–1802, 2011.

[KW21] Sam Kim and David J. Wu. Watermarking cryptographic functionalities from standard lattice assumptions. *J. Cryptol.*, 34(3), jul 2021.

[LG15] Wouter Lueks and Ian Goldberg. Sublinear scaling for multi-client private information retrieval. In *FC*, 2015.

[Lip09] Helger Lipmaa. First CPIR protocol with data-dependent computation. In *ICISC*, 2009.

[LLWR22] Jian Liu, Jingyu Li, Di Wu, and Kui Ren. Pirana: Faster (multi-query) pir via constant-weight codes. Cryptology ePrint Archive, Paper 2022/1401, 2022. https://eprint.iacr.org/2022/1401.

[LMW22] Wei-Kai Lin, Ethan Mook, and Daniel Wichs. Doubly efficient private information retrieval and fully homomorphic ram computation from ring lwe. Cryptology ePrint Archive, Paper 2022/1703, 2022. https://eprint.iacr.org/2022/1703.

[LP22] Arthur Lazzaretti and Charalampos Papamanthou. Single server pir with sublinear amortized time and polylogarithmic bandwidth. Cryptology ePrint Archive, Paper 2022/830, 2022. https://eprint.iacr.org/2022/830.

[LP23] Arthur Lazzaretti and Charalampos Papamanthou. Treepir: Sublinear-time and polylog-bandwidth private information retrieval from ddh. Cryptology ePrint Archive, Paper 2023/204, 2023. https://eprint.iacr.org/2023/204.

[MBFK16] Carlos Aguilar Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. Xpir: Private information retrieval for everyone. *Proceedings on Privacy Enhancing Technologies*, pages 155–174, 2016.

[MCR21] Muhammad Haris Mughees, Hao Chen, and Ling Ren. Onionpir: Response efficient single-server pir. In *CCS*. Association for Computing Machinery, 2021.

[MR22] Muhammad Haris Mughees and Ling Ren. Vectorized batch private information retrieval. Cryptology ePrint Archive, Paper 2022/1262, 2022. https://eprint.iacr.org/2022/1262.

[MW22] Samir Jordan Menon and David J. Wu. Spiral: Fast, high-rate single-server PIR via FHE composition. In *IEEE S&P*, 2022.

[MZRA22] Yiping Ma, Ke Zhong, Tal Rabin, and Sebastian Angel. Incremental {Offline/Online}{PIR}. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1741–1758, 2022.

[OS07] Rafail Ostrovsky and William E. Skeith, III. A survey of single-database private information retrieval: techniques and applications. In *PKC*, pages 393–411, 2007.

[PPY18] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. Private stateful information retrieval. In *CCS*, 2018.

[PR93] P. Pudlák and V. Rödl. Modified ranks of tensors and the size of circuits. In *STOC*, 1993.

[pro] Project Sonar. https://www.rapid7.com/research/project-sonar/. Accessed: 2023-3-28.

[PS18] Chris Peikert and Sina Shiehian. Privately constraining and programming PRFs, the LWE way. In *Public Key Cryptography (2)*, volume 10770 of *Lecture Notes in Computer Science*, pages 675–701. Springer, 2018.

[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 *CRYPTO*, 2021.

[SPS14] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. Practical dynamic searchable symmetric encryption with small leakage. In *Network and Distributed System Security Symposium (NDSS)*, 2014.

[SSP13] Elaine Shi, Emil Stefanov, and Charalampos Papamanthou. Practical dynamic proofs of retrievability. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[ZLTS23] Mingxun Zhou, Wei-Kai Lin, Yiannis Tselekounis, and Elaine Shi. Optimal single-server private information retrieval. In *EUROCRYPT*, 2023.

A Extensions

A.1 Supporting Key-Value Queries

Our PIR scheme so far supports memory lookup queries, where the client wants to query some index x into some database. In some real-world applications such as private DNS, the client wants to query some search key rather than an index. Our scheme can easily be modified to support a key-value interface as follows. First, the server can use a Cuckoo hashing scheme to hash all n keys into a table D of size $O(n)$, along with an overflow pile F which is logarithmic in size except with negligibly small probability. The server publishes the randomness seed used in the Cuckoo hashing as well as the overflow pile F. The client will store the overflow table F locally. Moreover, using the randomness seed, given any key, the client can compute the two relevant indices x_0 and x_1 in the table D to look for key. It is guaranteed that key exists in either $D[x_0]$ or $D[x_1]$, or in the overflow pile F. The client can retrieve both $D[x_0]$ and $D[x_1]$ using our PIR scheme that works for memory lookup.

A.2 Supporting Dynamic Databases

So far, we have focused on a stataic database. In some applications such as private DNS, the database will evolve over time. It is not hard to transform our static scheme into a dynamic one using a standard technique called "hierarchical data structures". This technique was originally proposed by Bentley and Saxe [BS80]. Since then, it has been used in various cryptographic applications to transform static schemes into dynamic ones, such as Oblivious RAM [GO96, Gol87], proof of retrievability [SSP13], searchable encryption [SPS14], and PIR [KCG21].

Below we describe how to use this approach in our context to make the scheme dynamic.

Syntax. Specifically, we want to have a PIR scheme for key-value queries, supporting the following operations:

- $\mathsf{Init}(1^\lambda, \mathsf{DB})$: given a key-value store DB, initialize a PIR scheme.
- $\mathsf{Query}(\mathsf{key})$: the client wants to look up the value associated with some key key.

- $\mathsf{Insert}(\mathsf{key}, \mathsf{val})$: add a new entry $(\mathsf{key}, \mathsf{val})$ to the key-value store.
- $\mathsf{Update}(\mathsf{key}, \mathsf{val})$: update the value of an existing key to the specified new value.
- $\mathsf{Delete}(\mathsf{key})$: delete key from the key-value store.

Construction. Let n be the maximum size of the database. Let $Q = \sqrt{n} \log n \cdot \alpha(n)$ where $\alpha(\cdot)$ is an arbitrarily small super-constant function. We assume that $n = 2^L \cdot Q$. We will use a hierachical data structure Γ with logarithmically many levels denoted $\Gamma_0, \Gamma_1, \ldots, \Gamma_L$, where each level ℓ may either be empty or have a PIR scheme of size $2^\ell \cdot Q$.

Let t be the number of update operations (including insertions, updates, or deletions) that have taken place so far including the current operation. We assume that at any point of time, the client always locally stores the most recent Q updates (including insertions, updates, or deletions). Further, these most recent Q updates are also stored at the server, in a separate array called Γ_{-1}.

- $\mathsf{Init}(1^\lambda, \mathsf{DB})$: Suppose that the size of the database $|\mathsf{DB}| = 2^\ell \cdot Q$. Run the preprocessing phase of the PIR scheme with each client, using the key-value store DB. At this moment, we have only one PIR instance corresponding to the level Γ_ℓ. Every other level is empty.
- $\mathsf{Insert}(\mathsf{key}, \mathsf{val})$: Record the operation including the type of the operation in Γ_{-1}. If t is a multiple of Q. Let ℓ^* be the first empty level. At this moment, we want to merge all PIR schemes in levels $\Gamma_{-1}, \Gamma_0, \ldots, \Gamma_{\ell^*-1}$ into a new PIR scheme in Γ_{ℓ^*}. If no empty level is found, then we want to merge levels $\Gamma_{-1}, \Gamma_0, \ldots, \Gamma_{\ell^*}$ into level Γ_{ℓ^*}.

  The merge is done as follows: first, we examine all the update operations in the levels to be merged, and perform a duplicate suppression. During the duplicate suppression, the most recent update to some key should override old ones. Unless we are rebuilding the last level L, if some key has been deleted, we will explicitly record that its corresponding value is $\perp$. Only when we are rebuilding the last level L, can we actually delete this key.

  After the duplicate suppression, we get a key-value store with at most $2^{\ell^*} \cdot Q$ entries — this will become the new database at level ℓ^*. The server now runs the preprocessing stage of the PIR scheme with every client for this key-value store.
- $\mathsf{Update}(\mathsf{key}, \mathsf{val})$: Same as $\mathsf{Insert}(\mathsf{key}, \mathsf{val})$.
- $\mathsf{Delete}(\mathsf{key})$: Same as $\mathsf{Insert}(\mathsf{key}, \perp)$.
- $\mathsf{Query}(\mathsf{key}, \mathsf{val})$: For $\ell = 0, 1, \ldots, L$, if Γ_ℓ is not empty, invoke the PIR scheme of level Γ_ℓ to query the value corresponding to key. Let v_ℓ be the answer obtained from level ℓ. Further, the client also looks up its local table of the most recent Q updates, and obtains another answer v_{-1}.

  Each answer v_i may be of the form, "`not found`" , $\perp$ (which indicates that the key is deleted), or some actual value. If all levels report "`not found`", the client outputs "`not found`". Otherwise, it outputs the freshest value found that is possibly $\perp$.

In practice, the client need not be constantly online. For the periodic rebuilds that stem from updates, the client can defer the rebuild work to the next time it comes online and makes queries. The cost of the periodic rebuilds need to be amortized to the total number of updates — see our performance analysis later.

Removing known-n assumption. So far, we assumed that we know an upper bound n on the maximum number of entries in the key-value store. This assumption can easily be removed as

follows. When we are rebuildling the last level L, if we discover that the number of entries has exceeded n, we update $n \leftarrow 2n$ as the new upper bound, i.e., increase the number of levels by 1.

Similarly, when we are rebuilding the last level L, if we discover that the actual number of entries is less than $n/2$, we can also update the new upper bound to be $n \leftarrow n/2$, i.e., reduce the number of levels by 1.

Performance analysis. We now analyze the cost of the scheme. In the analysis below, we will amortize the cost of periodic rebuilds (i.e., preprocessing) to the updates. The initial preprocessing is only one-time and will be amortized to an unbounded number of queries, so the amortized cost is arbitrarily small.

- *Online query costs.* For each query, the online cost is the sum of the costs of querying $O(\log n)$ PIR schemes, each of size Q, $2Q$, ..., n. The total amortized communication is $O_\lambda(Q^{\frac{1}{2}} + (2Q)^{\frac{1}{2}} + \ldots + n^{\frac{1}{2}}) = O_\lambda(\sqrt{n})$. Using a similar calculation, the amortized online server computation is $O(\sqrt{n})$. The amortized client online computation is $O_\lambda(\sqrt{n})$.

- *Update costs.* Every Q updates, we need to perform the preprocessing phase for a Q-sized database. The amortized communication is $C_\lambda \cdot Q/Q = C_\lambda$, the amortized server time is C, and the amortized client time is $C_\lambda \cdot \log \kappa \cdot \alpha(\kappa)$ for some constant C and another parameter C_λ related to the security parameter λ. Every $2Q$ updates, we need to perform the preprocessing phase for a $2Q$-sized database. The amortized communication is C_λ, the amortized server time is C, and the amortized client time is $C_\lambda \log \kappa \cdot \alpha(\kappa)$. Every $4Q$ updates, we need to perform the preprocessing phase for a $4Q$-sized database, and so on. Therefore, in total, the amortized communication per update is $O_\lambda(\log n)$, the amortized server computation per update is $O(\log n)$, the amortized client computation per update is $O_\lambda(\log n \log \kappa \cdot \alpha(\kappa))$.

- *Space.* The client space is $O_\lambda(\sqrt{n} \log \kappa \cdot \alpha(\kappa))$. The server's storage is $O(n)$.