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 an optimal tradeoff between client storage and server computation (up to poly-logarithmic factors). Our scheme achieves amortized $\widetilde{O}(\sqrt{n})$ server 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 relies only on Pseudo-Random Functions (PRF). To the best of our knowledge, Piano is the first practical single-server sublinear-time PIR scheme, and we outperform the state-of-the-art single-server PIR by $10\times - 300\times$. In comparison with the best known two-server PIR scheme, Piano enjoys comparable performance but our construction is considerably simpler. Experimental results show that for a 100GB database and with 60ms round-trip latency, Piano achieves 93ms response time, while the best known prior scheme requires 11s or more.

1 Introduction

Suppose that a server has a 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 this problem, and they came up with a cryptographic construction called Private Information Retrieval (PIR). Later, 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, 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.

Note: This is a major revision of the conference version to appear in IEEE S&P 2024. We present a slightly different but conceptually simpler scheme in this revised version. The initial scheme is presented in the appendix.

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 question:

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

An affirmative answer to the above question promises 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

[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.

feature of Piano lies in its *simplicity*. Unlike prior sublinear PIR schemes [CK20, CHK22, ZLTS23, 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 **93ms** response time, whereas SimplePIR suffers from 11s or higher response time[2]. This represents over **111**$\times$ 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 **1.53**$\times$.

Theoretical Contributions. 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 6 provides theoretical comparison with additional related work.

2 Main Construction

Suppose the database $\mathbf{DB}[0 \dots n-1]$ contains n bits. We divide the indices $\{0, 1, \dots, n-1\}$ into $\sqrt{n}$ chunks each of size $\sqrt{n}$. Specifically, the j-th chunk where $j \in \{0, 1, \dots, \sqrt{n}-1\}$ contains the indices $\{j \cdot \sqrt{n}, \dots, (j+1) \cdot \sqrt{n} - 1\}$.

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

Distribution of a random set. We will use the following simple strategy to sample a random set of indices of size exactly $\sqrt{n}$: simply sample one random index from every chunk. Henceforth, we use the notation S to denote such a random set and we use $S[j]$ to denote the index in S belonging to the j-th chunk.

Client's hint. Suppose that the client stores the following hint data structure:

1. **Primary table**: contains $\widetilde{O}(\sqrt{n})$ entries, where $\widetilde{O}(\cdot)$ hides polylogarithmic factors. The i-th entry in the hint table contains
   - A random set S of $\sqrt{n}$ indices chosen according to the aforementioned distribution;
   - The parity bit $p = \oplus_{i \in S} \mathbf{DB}[i]$.
2. **Replacement entries**: for each chunk $j \in \{0, 1, \sqrt{n} - 1\}$, store $\widetilde{O}(1)$ entries of the form $(i, \mathbf{DB}[i])$ where each i is a randomly sampled index from chunk j.
3. **Backup table** (needed for multiple queries): for each chunk $j \in \{0, 1, \sqrt{n} - 1\}$, store $\widetilde{O}(1)$ entries of the form (S, p), where S is a random set sampled according to the aforementioned distribution, and $p = \oplus_{i \in S \setminus \{S[j]\}} \mathbf{DB}[i]$. In other words, p is the parity of the database bits at all indices in S except the index corresponding to the j-th chunk.

Compressing client storage using PRFs. For the primary and backup table, if the client stores the full sets, the storage overhead will be $\widetilde{O}(n)$. However, in our full scheme we will use a PRF key to succinctly represent each set in the primary and backup tables, and thus the client's storage can be reduced to $\widetilde{O}(\sqrt{n})$.

Learning the hint in a single streaming pass. For the time being, we may assume that the client can somehow magically learn this hint table. Later, we will show how the client can learn this hint table using a streaming algorithm which makes a single linear scan over the database, while consuming only $\widetilde{O}(\sqrt{n})$ local storage. The communication and computational overhead of this preprocessing step is $\widetilde{O}(n)$. Later in our full scheme, this preprocessing step needs to be performed every $\widetilde{O}(\sqrt{n})$ queries. If we spread the $\widetilde{O}(n)$ work across $\widetilde{O}(\sqrt{n})$ queries, the amortized cost per query will be $O(\sqrt{n})$.

We stress that the preprocessing phase does not leak any information to the server, since the server only observes a linear scan over the database.

Making a single query. To support a single query, we only need to make use of the primary table and the replacement entries. Specifically, suppose the client wants to learn $\mathbf{DB}[x]$. It will perform the following:

1. Find an entry (S, p) in the primary table such that $x \in S$. This succeeds with all but negligible probability.
2. Let $j = \mathsf{chunk}(x)$ denote the chunk that x belongs to, and let $(r, \mathbf{DB}[r])$ be the next unconsumed replacement entry belonging to chunk j.
3. Replace the j-th entry in S with r; let S' denote the modified set, send S' to the server.
4. The server sends back $p' = \oplus_{i \in S'} \mathbf{DB}[i]$, and the client computes $\mathbf{DB}[x] = p' \oplus \mathbf{DB}[r] \oplus p$.

Clearly, the communication overhead is $O(\sqrt{n})$. Further, the set sent to the server has the same distribution as a freshly sampled random set. Thus, the server learns no information about the client's query.

Supporting $\widetilde{O}(\sqrt{n})$ random, distinct queries. We now discuss how to extend the scheme to support $Q = \widetilde{O}(\sqrt{n})$ random, distinct queries. After making a query, the (S, p) consumed should be removed from the primary table, since if the same entry is used again, it will leak information. However, simply removing the entry (S, p) is also not secure, since it skews the distribution of the random sets in the primary table. For example, if the client has made a query for the index 5, it will consume a set S containing 5. This means the remaining sets in the primary table will less likely contain 5, which skews the distribution of the future sets sent to the server.

To support multiple queries while ensuring security, we can make the following simple modification to the scheme. Whenever an entry (S, p) is consumed from the primary table during a query for $\mathbf{DB}[x]$, the client grabs the next unconsumed entry (S', p') from the backup table corresponding to $\mathsf{chunk}(x)$. It replaces the consumed entry with $(S'\langle\mathsf{chunk}(x) \to x\rangle, p' \oplus \mathbf{DB}[x])$ where[3] $S'\langle\mathsf{chunk}(x) \to x\rangle$ is otherwise the same as S' except for replacing $S'[\mathsf{chunk}(x)]$ with x. Observe that the consumed entry is a random set subject to containing x, and its replacement is also a random set subject to containing x. Therefore, the distribution of the sets in the primary table is unaffected.

The scheme so far can support $Q = \widetilde{O}(\sqrt{n})$ random distinct queries, because we provisioned polylogarithmically many replacement entries and backup table entries per chunk. With $Q = \widetilde{O}(\sqrt{n})$ random distinct queries, with all but negligible probability, each chunk will only be hit at most polylogarithmically many times. This means that we will not run out of replacement entries and backup table entries except with negligible probability.

Supporting unbounded, arbitrary queries. We can easily get rid of the "distinct query" assumption in the following way: suppose that the client stores the result of the most recent $Q = \widetilde{O}(\sqrt{n})$ queries. If a duplicate query is made, it simply looks up the answer locally, and it sends another random distinct query to the server to mask the fact that it is a duplicate query.

Next, we can get rid of the "random query" assumption in the following way, and the resulting scheme supports $Q = \widetilde{O}(\sqrt{n})$ arbitrary queries. Observe that the "random query" assumption is needed only for load balancing. Imagine that upfront, the server applies a pseudorandom permutation (PRP) to all indices of the database. We may assume that this permutation is independent of the queries. The server publishes the PRP key, and the client is now 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 (which is impossible anyway if the server is malicious).

Finally, we can get rid of the Q-bounded query assumption through a simple pipelining trick: during the current window of Q queries, we run the preprocessing phase of the next Q queries. As mentioned earlier, the total communication and computation overhead of the preprocessing is $\widetilde{O}(n)$. Thus, in our final scheme, we have a *one-time* preprocessing phase with $\widetilde{O}(n)$ communcation and computation, while consuming only $\widetilde{O}(\sqrt{n})$ client storage. After the one-time preprocessing, we can support an *unbounded* number of *arbitrary* queries. The communication and computation cost of each query is $O(\sqrt{n})$.

Detailed description. Figure 1 gives a detailed description of our scheme for supporting $\widetilde{O}(\sqrt{n})$ random, distinct queries. As mentioned, it is easy to upgrade such a scheme to one that supports

[3] Note that since the client just queried the index x, it knows what $\mathbf{DB}[x]$ is.

Single-Server Scheme for $Q = \sqrt{n} \log \kappa \cdot \alpha(\kappa)$ Queries [a]

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 $M_1 = \sqrt{n} \log \kappa \cdot \alpha(\kappa)$ PRF keys denoted $\mathsf{sk}_1, \ldots, \mathsf{sk}_{M_1} \in \{0,1\}^\lambda$ for the primary table. Initialize the parities $p_1, \ldots, p_{M_1}$ to zeros.
- For $j \in \{0, 1, \ldots, \sqrt{n}-1\}$, Client samples $M_2 = \log \kappa \cdot \alpha(\kappa)$ PRF keys denoted $\overline{\mathsf{sk}}_{j,1}, \ldots, \overline{\mathsf{sk}}_{j,M_2}$, representing all the backup keys for the j-chunk. Initialize the parities $\overline{p}_{j,1}, \ldots, \overline{p}_{j,M_2}$ to zeros.
- Client downloads the whole DB from the server in a streaming way: when the client has the j-th chunk $\mathbf{DB}[j\sqrt{n} : (j+1)\sqrt{n}]$:
  - *Update primary table*: for $i \in [M_1]$, let $p_i \leftarrow p_i \oplus \mathbf{DB}[\mathbf{Set}(\mathsf{sk}_i)[j]]$.
  - *Store replacement entries*: sample and store M_2 tuples of the form $(r, \mathbf{DB}[r])$ where r is a random index from the j-th chunk.
  - *Update backup table*: for $i \in \{0, 1, \ldots, \sqrt{n} - 1\}/\{j\}$ and $k \in [M_2]$, let $\overline{p}_{i,k} \leftarrow \overline{p}_{i,k} \oplus \mathbf{DB}[\mathbf{Set}(\overline{\mathsf{sk}}_{i,k})[j]]$.
  - Delete $\mathbf{DB}[j\sqrt{n} : (j+1)\sqrt{n}]$ from the local storage.
- At this moment, let $T := \{((\mathsf{sk}_i, \perp), p_i)\}_{i \in [M_1]}$ denote the client's *primary table*, and let $\{(\overline{\mathsf{sk}}_{j,i}, \overline{p}_{j,i})\}_{i \in [M_2]}$ denote the backup entries for the j-th chunk.

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

1. **Query:**
   (a) Client finds a hint $T_i := ((\mathsf{sk}_i, x'), p_i)$ in its primary table T such that $x \in \mathbf{Set}(\mathsf{sk}_i, x')$. Let $S = \mathbf{Set}(\mathsf{sk}_i, x')$.
   (b) Let $j^* = \mathsf{chunk}(x)$, Client finds the first unconsumed replacement entry from the j^*-th chunk, denoted $(r, \mathbf{DB}[r])$.
   (c) Client sends $S' = S\langle j^* \to r\rangle$ to the server if the previous two steps succeeded. Otherwise, send a random set.
   (d) Upon receiving a set S', the server returns $q = \oplus_{k \in S'} \mathbf{DB}[k]$.
   (e) Client computes the answer $\beta = q \oplus p_i \oplus \mathbf{DB}[r]$ if steps (a) and (b) succeeded. Otherwise, Client sets the answer $\beta = 0$.
2. **Refresh:**
   - Client finds the next unconsumed backup entry $\left(\overline{\mathsf{sk}}_{j^*,k}, \overline{p}_{j^*,k}\right)$ belonging to the j^*-th chunk. If not found, Client generates a random $\overline{\mathsf{sk}}_{j^*,k}$ and lets $\overline{p}_{j^*,k} = 0$.
   - If steps (a) and (b) of the query phase succeeded, then Client replaces the matched entry in the primary table with $\left((\overline{\mathsf{sk}}_{j^*,k}, x), \overline{p}_{j^*,k} \oplus \beta\right)$.

[a] For clarity, we present the scheme supporting distinct and random queries. As mentioned before, these restrictions can be removed by applying PRP and local caching.

Figure 1: Detailed description of Piano.

unbounded number of arbitrary queries.

In Figure 1, we use the following notation for describing pseudorandom sets. Henceforth let PRF denote a pseudorandom function whose output is in the range $\{0, 1, \ldots, \sqrt{n} - 1\}$, and let sk denote a PRF key.

- For $i \in \{0, 1, \ldots, n\}$, let $\mathsf{chunk}(i) = \lfloor i/\sqrt{n} \rfloor$ be the chunk i belongs to.
- $\mathbf{Set}(\mathsf{sk}) := \{j \cdot \sqrt{n} + \mathsf{PRF}_{\mathsf{sk}}(j)\}_{j \in \{0, \ldots, \sqrt{n}-1\}}$; It is easy to see that given $x \in \{0, 1, \ldots, n-1\}$, and sk, it takes $O(1)$ time to test whether $x \in \mathbf{Set}(\mathsf{sk})$.
- $\mathbf{Set}(\mathsf{sk}, x)$ where $x \in \{\perp\} \cup \{0, 1, \ldots, n-1\}$ is defined as follows:

$$\mathbf{Set}(\mathsf{sk}, x) = \begin{cases} \mathbf{Set}(\mathsf{sk}) & \text{if } x = \perp \\ \mathbf{Set}(\mathsf{sk})\langle \mathsf{chunk}(x) \to x \rangle & \text{o.w.} \end{cases}$$

  Recall that the notation $S\langle \mathsf{chunk}(x) \to x \rangle$ means the set obtained by replacing the index pertaining to $\mathsf{chunk}(x)$ in S with x.

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 $\mathbf{DB} \in \{0, 1\}^n$ as input. The client may interacts with the server and stored some hints in its local stroage. For simplicity, we assume the entries in $\mathbf{DB}$ are 1-bit[4].
- **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 $\mathbf{DB} \in \{0, 1\}^n$, 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 $\mathbf{DB}$.

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 $\mathbf{DB} \in \{0, 1\}^n$, 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 $\mathbf{DB}$ 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 $\mathbf{DB} \in \{0, 1\}^n$, $\mathcal{A}$'s views in the following two experiments are computationally indistinguishable:

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

- Real: an honest client interacts with $\mathcal{A}(1^\lambda, n, \mathbf{DB})$ 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 $\mathcal{A}(1^\lambda, n, \mathbf{DB})$ 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 provide the (sketched) proofs of privacy and correctness of our PIR scheme. We defer the full proofs to the appendices. We also provide the performance analysis.

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

Proof. The full proof is deferred to Appendix C. We provide some high-level ideas here.

We can first replace the PRF with true randomness and due to the pseudorandomness of the PRF, it is indistinguishable to the adversary.

With respect to the view of the server, the edited set can be simulated by just generating a random set containing $\sqrt{n}$ indices, where each one chunk contains exactly one random index.

Therefore, we only need to prove that the distribution of the client's primary hints are always "uniformly random" in the view of the adversary. After querying for x, we always replace the hint with a new hint that contains the current query x. This maintains the distribution of the client's primary hint table *in the adversary's view.* For intuition, consider a simplified case where the client just has one local set. Let the query be x. There are two cases:

1. With probability $1 - 1/\sqrt{n}$, the set does not contain x. The client sends a random set to the server. The local set stays the same.
2. With probability $1/\sqrt{n}$, the set contains x. The client sends an edited set by replacing x with a random index from the same chunk. The client samples a new local set containing x.

The key insight is that the adversary does not know which case happens. It only sees some random set independent of the remaining local set and the client's new local set is distributed as:

1. With probability $1 - 1/\sqrt{n}$, a random set does not contain x;
2. With probability $1/\sqrt{n}$, a random set contains x.

This is identically distributed as a uniformly random set.

The Lemma C.2 essentially just extends this calculation to the actual case where the client has multiple local sets. ☐

Theorem 3.3 (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. The full proof is deferred to Appendix C. We provide some high-level ideas here.

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.

Since each hint contains the querying index with probability exactly $1/\sqrt{n}$, applying union bound, we can directly bound the first type of error by $Q(1-1/\sqrt{n})^{M_1}$. Plugging in the numbers and the probability is negligible in κ.

The probability of the second type of error can be bounded by a simple "Q balls into $\sqrt{n}$ bins" argument with the Chernoff bound.

□

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

Theorem 3.4 (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. The client also stores $O(\sqrt{n}\log\kappa \cdot \alpha(\kappa))$ replacement index-value pairs. 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 $O_\lambda(\sqrt{n}\log\kappa \cdot \alpha(\kappa))$.

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 $O_\lambda(\sqrt{n})$. Other operations all take $O_\lambda(\sqrt{n})$ time. The client then sends an $O(\sqrt{n})$-sized set to the server. The server computation time is $O(\sqrt{n})$. The client downloads the response of size $O(\sqrt{n})$ and gets the answer. The refreshing time for each query is $O(1)$.

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. 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 brings 2x cost to the local storage. The server does not need to have any additional storage.

□

4 Evaluation

We implemented a slightly different variant of the scheme which we describe fully in **??**. In comparison with the version in our main body, the variant in **??** has slightly less client storage, but slightly more communication overhead.

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, like many other practical single-server PIR implementations.

4.1 Implementation

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. 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 is bounded by $2^{-\kappa} = 2^{-40}$ for all Q queries, matching the same failure probability as SimplePIR [HHCG+22]. We use 128-bit keys and use AES to instantiate the PRF.

4.2 Evaluation Setup

We evaluate Piano and the baseline schemes on two 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 server machine on the west coast, and the client 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.

	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.

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 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	68.4ms	61.0ms	10.9s	93.4ms
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 14% − 53% latency overhead. SimplePIR, on the other hand, has a 4.6x - 178.7x latency overhead.

4.5 Performance Breakdown

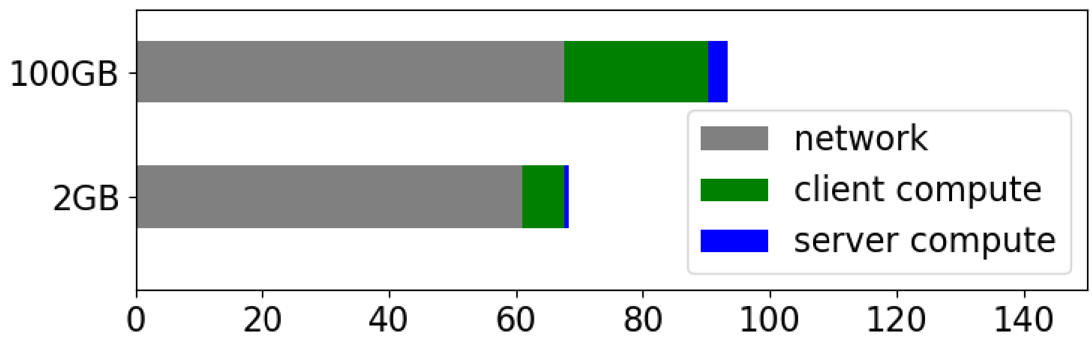

Figure 2: Cost breakdown

In Figure 2, we provide a detailed performance breakdown of the online time when running Piano on wide-area-network (the same setup as in Table 2). We see that the online time is mostly dominated by the network transmission time. The network transmission time is bounded by the physical distance – a 60ms RTT is required even the payload is negligible. The overhead of more data being transmitted is small since the bandwidth is 2GB/s in our setup and we transmit at most 900KB per query. The client computation time is the second largest factor and it is dominated by the time to find a matched hint, which requires expected $O(\sqrt{n})$ PRFs evaluations. The server computation time is much faster since the server-side algorithm only requires some RAM accesses with some non-cryptographic computation.

5 Additional Optimizations

We provide some additional optimization ideas.

Compressing local storage. Instead of generating a λ-bit key for each hint, the client just needs to generate a master secret key msk and a unique short key sk_i (e.g. 32 bits) for the i-th hint. This idea could save the local storage by up to 50%.

Compressing offline bandwidth. During the offline phase, the server will streamingly send the whole database to the client. The server can compress each chunk using loseless compression algorithms to reduce communication.

6 Additional Related Work

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

Single-server PIR schemes. Section 6 compares Piano with existing single-server PIR schemes. Although our paper primarily focuses on enhancing the practical performance of 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].

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	$O(\sqrt{n})$	$\widetilde{O}_\lambda(\sqrt{n})$	$\widetilde{O}_\lambda(\sqrt{n})$

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.

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] relied on non-standard assumptions or VBB obfuscation. A recent breakthrough work by Lin, Mook and Wichs [LMW22] 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 $\widetilde{O}_\lambda(n^{1+\varepsilon})$. Later, the server can answer queries in $\mathsf{poly}((\log n)^{1/\varepsilon})$ time with $\mathsf{poly}((\log n)^{1/\varepsilon})$ communication cost, where poly is a fixed polynomial.

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 FHE. Zhou et al. [ZLTS23] and Lazzaretti and Papamanthou [LP22] further extended the idea and achieved polylogarithmic communication cost.

In terms of practical schemes, all the previous works supporting adaptive queries had $\tilde{O}(n)$ per-query computation time. Most of them relied on homomorphic encryption (usually not just linear homomorphic encryption) and required the LWE assumption. XPIR [MBFK16] was among the first to implement 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. Nonetheless, all these schemes have linear online server time.

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. Batch PIR schemes have two main limitations. First, a client must make many parallel queries simultaneously to amortize the computation cost. In 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, the amortized computation time can match Piano. Second, these schemes require the server to run some form of homomorphic encryption evaluations on the entire **DB** and incur $O(n)$ computation per batch, making the overall latency 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 100x 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 implementations.

TreePIR reported the best performance among these implementations, providing one implementation with polylogarithmic per-query communication cost by invoking a recursive scheme and another one 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 4x local storage. The blowup of the local storage comes from the backup hints and the deamortization of the setup phase, which are inherently required for the single-server setting.

7 Limitations and Suitable Use Cases

The main limitation of Piano is its communication cost: 1) the client has to download the whole database during the setup phase; 2) the online communication cost per query is $O(\sqrt{n})$. Compared to previous solutions like Zhou et al. [ZLTS23] and Lazzaretti and Papamanthou [LP22] which have $\widetilde{O}_\lambda(1)$ communication overhead per query, the cost of Piano is $O(\sqrt{n})$. However, we argue that this sacrifice is actually what makes our solutions practical. The streaming preprocessing avoids the need of using FHE during the offline phase. Also, private programming of PRF is required to achieve $\widetilde{O}_\lambda(1)$ online bandwidth in previous solutions [ZLTS23, LP22] and this primitive is only known in theory. By sending the whole edited set,we can do puncturing or programming without need of complicated constructions. That being said, designing a truly practical single-server PIR with $\widetilde{O}_\lambda(1)$ communication overhead is one of the major future questions to be explored. We provide two possible use cases for Piano.

- **Private Light-weight Blockchain Node**. When a light-weight blockchain node needs to fetch data from the blockchain, it makes queries to other full nodes. A light-weight node needs to make a verification pass over the blockchain history, and it has frequent queries, which makes Piano a suitable privacy-preserving solution.

- **Private DNS Service**. DNS queries are usually made frequently and usually made during a period periodically (e.g., daytime). Piano is also suitable for building a private DNS service. Users can preprocess during rest time and make queries during the online phase.

8 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 93.4ms response time, which is only 1.53× 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 specifically thank Arthur Lazzaretti for introducing TreePIR to us and discussing general PIR ideas with us. We thank Jesko Dujmovic for suggesting an improvement to our previous version of the scheme (the variant in Appendix A), which made our scheme even simpler. 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 *OSDI*, 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*, 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, 2022.

[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 *EUROCRYPT*, 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. 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, 2022.

[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, 2022.

[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, 2022.

[LP22] Arthur Lazzaretti and Charalampos Papamanthou. Single server pir with sublinear amortized time and polylogarithmic bandwidth. Cryptology ePrint Archive, 2022.

[LP23] Arthur Lazzaretti and Charalampos Papamanthou. Treepir: Sublinear-time and polylog-bandwidth private information retrieval from ddh. Cryptology ePrint Archive, 2023.

[MBFK16] Carlos Aguilar Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. Xpir: Private information retrieval for everyone. *PETS*, 2016.

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

[MR22] Muhammad Haris Mughees and Ling Ren. Vectorized batch private information retrieval. Cryptology ePrint Archive, 2022.

[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 *Usenix Security*, 2022.

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

[PS18] Chris Peikert and Sina Shiehian. Privately constraining and programming PRFs, the LWE way. In *PKC*, 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 A Variant of Piano

We present a variant of Piano in this section[5]. This variant has less storage but comes with more online communication. The only difference will be in the online stage.

[5]This is the initial scheme when we publicized the paper in March 2023. Although both being easy to implement, we consider the new main scheme is conceptually simpler. Henceforce, we decided to present the initial scheme as a variant.

Client side: A different approach to hide the query point. Recall that for each query x, the client will find a preprocessed set S containing x and the client wants to learn the parity $S/\{x\}$ (which is enough for it to learn $\mathbf{DB}[x]$). The main point is to learn the parity $S/\{x\}$ while preserving privacy. In the scheme presented in Section 2, the client will replace the query point x by some preprocessed replacement indices r from the same chunk to ensure privacy. The query set will look random in the view of the server and the client can learn the parity of $(S/\{x\}) \cup \{r\}$.

The variant takes a different method to hide the query point. Given a set S that contains one index from each chunk, we first write its "offset vector" as $\Delta = (S[0] \bmod \sqrt{n}, S[1] \bmod \sqrt{n}, \ldots, S[\sqrt{n}-1] \bmod \sqrt{n})$. For example, if the DB size is 16 and the set is $\{2, 4, 11, 13\}$, the offset vector will be $(2, 0, 3, 1)$. Let j be x's chunk index. Consider the following offset vector that removes the offset of x and compacts the remaining offsets:

$$\Delta_{-x} = \left(S[0] \bmod \sqrt{n}, \ldots, S[j-1] \bmod \sqrt{n}, S[j+1] \bmod \sqrt{n}, \ldots, S[\sqrt{n}-1] \bmod \sqrt{n}\right)$$

For example, removing the offset of 4 from the offset vector of the last example will result in a vector of $(2, 3, 1)$. We observe that after removing the offset of x from the vector, the compacted remaining vector completely hides the information of x.

Therefore, the client can directly send Δ_{-x} to the server.

Server side: Returning the correct parity efficiently. The server cannot directly recover the set $S/\{x\}$ from the vector $\Delta' = (\delta_0, \ldots, \delta_{\sqrt{n}-2})$ it receives, because the chunk index of the removed index is unknown. However, the server can guess all $\sqrt{n}$ possible cases and reconstruct a possible set for each guess. For example, if the server guesses the removed point is from the i-th chunk, the server can reconstruct a set as

$$S_i = \{\delta'_0, \delta'_1 + \sqrt{n}, \ldots, \delta_{i-1} + (i-1)\cdot\sqrt{n}, \perp, \delta_i + (i+1)\cdot\sqrt{n}, \ldots, \delta_{\sqrt{n}-2} + (\sqrt{n}-1)\cdot\sqrt{n}\},$$

where $\perp$ is simply a placeholder for the removed index. Denote the parity for S_i as $q_i = \oplus_{k\in S_i}\mathbf{DB}[k]$. Suppose the server can compute all $q_0, \ldots, q_{\sqrt{n}-1}$ efficiently, it can return all the guessed parities to the client with $O(\sqrt{n})$ communication cost. The client can directly pick up the correct guess q_j because it knows exactly where the removed point is!

Now we show that computing $q_0, \ldots, q_{\sqrt{n}-1}$ only takes $O(\sqrt{n})$ time. The naive approach is to recover the whole set for each guess and compute their parities directly. Since we have $\sqrt{n}$ guesses and each guess reconstructs a $(\sqrt{n}-1)$-size set, the computation time will be $O(n)$. However, observe that the symmetric difference between each two consecutive reconstructed set S_i and S_{i+1} will only be two elements: $\Delta'_i + (i+1)\cdot\sqrt{n}$ and $\Delta'_i + i\cdot\sqrt{n}$. Therefore, computing q_{i+1} from q_j only takes two extra xor operations. The algorithm can just compute q_0 directly in $O(\sqrt{n})$ time, and compute $q_1, q_2, \ldots, q_{\sqrt{n}-1}$ in sequence, each takes $O(1)$ time. So the total computation time is $O(\sqrt{n})$. We list the algorithm in Figure 3.

Comparison with the main scheme. The efficiency for this variant is nearly the same as the scheme presented before. The difference as follows. On the one hand, in this variant, the client does not have to store all the replacement index-value pairs. However, since other parts of the storage already consume $O(\sqrt{n}\log\kappa\alpha(\kappa))$ space, the asymptotic stroage cost stay the same. On the other hand, this variant has $O(\sqrt{n})$ download cost per query whereas the main scheme has $O(1)$ download cost. However, the upload costs are both $O(\sqrt{n})$ for the two schemes. So the asymptotic communication cost stays the same as $O(\sqrt{n})$. In short, the two schemes have the same asymptotic behaviors and they provide a tradeoff between the local storage and the online communication (up to a constant factor).

1. Upon receiving the offset vector Δ', parse Δ' as $(\delta_0, \ldots, \delta_{\sqrt{n}-2})$.
2. $q_0 = \oplus_{i \in \{1,\ldots,\sqrt{n}-1\}} \mathbf{DB}[\delta_{i-1} + i \cdot \sqrt{n}]$.
3. For $i = 0, \ldots, \sqrt{n} - 2$, compute $q_{i+1} = q_i \oplus \mathbf{DB}[\delta_i + (i+1) \cdot \sqrt{n}] \oplus \mathbf{DB}[\delta_i + i \cdot \sqrt{n}]$.
4. Return $(q_0, \ldots, q_{\sqrt{n}-1})$.

Figure 3: $O(\sqrt{n})$**-time server-side algorithm for one query.**

Correctness and Privacy Proof. The correctness proof and the privacy proof are nearly the same as the proofs presented in Appendix C. The failure probability analysis remains the same for the correctness proof. For the privacy proof, the only difference is that the simulation strategy for the simulator. It now sends a uniform vector $\Delta' \xleftarrow{\$} \{0, 1, \ldots, \sqrt{n} - 1\}^{\sqrt{n}-1}$ instead of a random set. These two simulation strategies are both indepedent of the query index. Other parts of the privacy proof stay the same.

B Extensions

B.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.

B.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:

- $\mathsf{Init}(1^\lambda, \mathbf{DB})$: given a key-value store $\mathbf{DB}$, 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}.

- $\mathsf{Init}(1^\lambda, \mathbf{DB})$: Suppose that the size of the database $|\mathbf{DB}| = 2^\ell \cdot Q$. Run the preprocessing phase of the PIR scheme with each client, using the key-value store $\mathbf{DB}$. 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, $2Q$, $\ldots$, 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 $O(n)$.

C Deferred Proofs

Theorem C.1 (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. The Ideal experiment is as follows.

- *Offline.* $\mathcal{A}$ receives the streaming signal.
- *Online.* for query i, the simulated client sends a set sampled from $\mathcal{D}_n$ to $\mathcal{A}$.

We define a hybrid experiment Hyb_1 as following:

- *Offline.* $\mathcal{A}$ receives the streaming 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 $x_t \in S$ and also a random index r from x_t's chunk. The client sends $(S/\{x_t\}) \cup \{r\}$ 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 replacing x with a random index from the same chunk, the edited set $(S/\{x_t\}) \cup \{r\}$ is identitcally distributed as $\mathcal{D}_n$. So the views are indeed indentically distributed in these two experiments.

Now we define a hybrid experiment Hyb_2:

- *Offline.* $\mathcal{A}$ receives the streaming signal. The client samples random sets $S_1, \dots, S_{M_1} \xleftarrow{\$} \mathcal{D}_n^{[M_1]}$.
- *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. The client samples a random index r from x_t's chunk. The client sends $(S^*/\{x_t\}) \cup \{r\}$ to the server (received by $\mathcal{A}$).
  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'.

The following lemm shows that the view of $\mathcal{A}$ in Hyb_1 and Hyb_2 are identically distributed.

Lemma C.2. *In Hyb_2, for every online queries x_t, even conditioned on $\mathcal{A}$'s view over the first $t-1$ queries,*

- *The set S' received by $\mathcal{A}$ is distributed as follows. Sample $S \xleftarrow{\$} \mathcal{D}_n$ conditioned on $x_t \in S$. Sample a random index r from x_t's chunk. Let $S' = (S/\{x_t\}) \cup \{r\}$.*
- *At the end of the t-th query, the client local sets $S_1, \dots, S_{M_1}$ are identically distributed as $S_1, \dots, S_{M_1} \xleftarrow{\$} \mathcal{D}_n^{M_1}$ even condtioned on the messages received by $\mathcal{A}$ during the first t-th queries.*

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

Base case. At the end of the offline phase, indeed $S_1, \dots, S_{M_1}$ are indeed distributed as $S_1, \dots, S_{M_1} \xleftarrow{\$} \mathcal{D}_n^{[M_1]}$. 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 $S_1, \dots, S_{M_1}$ are distributed as $S_1, \dots, S_{M_1} \xleftarrow{\$} \mathcal{D}_n^{[M_1]}$ 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 $S_1, \dots, S_{M_1}$ 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 samples a random index r from x_t's chunk. The client sends $(S_u/\{x_t\}) \cup \{r\}$ to the server (received by $\mathcal{A}$).
- 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 $u \leq M_1$, the client replaces S_u with S'_u.

The main random variables sampled in those two cases are $(S_1, \dots, S_{M_1}, u, S'_u)$ where $S_1, \dots, S_{M_1}$ are the sets at the beginning of the t-th query, u is the index of the first set containing x_t, and $S_1, \dots, S_{u-1}, S'_u, S_{u+1}, \dots, S_{M_1}$ will be the local sets at the end of the t-th query. In Hyb_2, by the induction hypothesis, $S_1, \dots, S_{M_1}$ 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 close to the real experiment. We define hybrid experiment Real^* as following:

- *Offline.* $\mathcal{A}$ receives the streaming 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. The client samples a random index r from x_t's chunk.
  3. The client sends $(S^*/\{x_t\}) \cup \{r\}$ to the server.
  4. 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 C.3 (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

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

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 C.2, 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

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

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 in κ since n is bounded by $\mathsf{poly}(\kappa)$.

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

□