Achieving Tight Space-Time Tradeoff and Practical Performance in Preprocessing PIR with Multi-level Recursion

Chang Shi
Peking University
2300017795@stu.pku.edu.cn

Bo Peng
Peking University
bo.peng@stu.pku.edu.cn

Zhechen Li
Ant Group
lizhechen@pku.edu.cn

Cheng Hong
Ant Group
vince.hc@antgroup.com

Mingxun Zhou*
The Hong Kong University of Science and Technology
mingxunz@ust.hk

***Abstract*—**

Client-specific preprocessing PIR supports sublinear online private queries after a linear-time offline phase that prepares client-specific hints. The relevant lower bound is tight: any scheme with S bits of client storage and online cost T must satisfy $S \cdot T = \Omega(n)$. Most practical random-set schemes fall short by a κ factor in client storage, while the known constant-factor-optimal schemes, WR-PIR (Eurocrypt 2025) and Balanced PIR (S&P 2026), rely on complex hint-management machinery and incur high concrete costs.

We present MULTI-LEVEL PIR, a preprocessing PIR scheme that matches this tight space-time tradeoff using only simple random-set components. The main idea is a multi-level composition: early levels are allowed to fail with noticeable probability, and later levels are invoked only when these query-independent failures occur. This "waterfall" structure drives the overall failure probability down to negligible while keeping expected online cost and client storage at $O(\sqrt{n})$.

Our implementation shows that this simpler structure gives competitive concrete performance. Compared with Piano and S3PIR, two prior state-of-the-art practical schemes, MULTI-LEVEL PIR achieves a 9–$20\times$ client space reduction; compared with Balanced PIR, it reduces preprocessing time by about 8–$45\times$ and online communication by about 5–$67\times$ in our evaluated settings, while remaining competitive in other online metrics.

As an additional theoretical result, we give a more involved variant with $O(n^{1/4})$ online communication, the first constant-factor-optimal preprocessing PIR scheme with sub-$\sqrt{n}$ online communication.

* *Corresponding author.*

1. Introduction

Private Information Retrieval (PIR) [1, 2] lets a client fetch an entry $\mathsf{DB}[i]$ from a server holding a database DB without revealing the index i. It is a basic tool for privacy-preserving systems, with applications in private recommendation [3], certificate transparency auditing [4], and biomedical matching [5].

We study *single-server client-specific preprocessing PIR* [6]. In this model, the client and server first spend $\Theta(n)$ offline work to prepare client-specific hints. The client later uses these hints to answer many private queries with sublinear online computation, typically $T = \tilde{O}(\sqrt{n})$. Such sublinear online cost is impossible for classical single-server PIR without preprocessing [7].

Random-set coverage in preprocessing PIR. Most practical schemes follow the same random-set coverage framework [6, 8, 9, 10, 11, 12]. Each hint consists of a compact description of a random set of database indices, such as a PRF seed, together with the parity of the database entries in that set. To retrieve $\mathsf{DB}[x]$, the client finds a stored set that contains x, queries a carefully edited version of the set whose distribution hides x, and combines the server's response with the stored parity to solve for $\mathsf{DB}[x]$. Different schemes realize this private editing step in different ways; Piano and S3PIR [9, 10], for example, mask the queried index by swapping in an independently random index.

The space-time tradeoff and lower bounds. Several prior works [6, 8, 13] proved that any preprocessing PIR with S bits of client storage and T online computation must satisfy $S \cdot T = \Omega(n)$. Prior practical schemes choose parameters near the balanced $\sqrt{n}$ point, yielding the common regime of having $\kappa\sqrt{n}$ sets, each containing $\sqrt{n}$ indices. The extra statistical security parameter κ ensures each index in the database is covered by κ times, achieving full-coverage except a $\mathsf{negl}(\kappa)$ failure probability. This gives $T = \tilde{O}(\sqrt{n})$ online computation and $S = \tilde{O}(\kappa\sqrt{n})$ client storage [9, 11, 14, 10, 15]. However, the κ overhead is substantial in practice: for a 128GB database, it often

translates to a heavy $40\times$ blowup and results in 1GB of client storage, which is prohibitive for resource-constrained devices such as smartphones and browser extensions.

Optimal-tradeoff prior work. Wang and Ren [16] (WR-PIR) achieve the tight tradeoff up to constant factors. Instead of covering each entry many times with independent random sets, WR-PIR organizes the hints so that each database entry is covered exactly once. This removes the κ-factor overhead, but the construction relies on a sophisticated relocation mechanism based on small-domain PRPs, which introduces high concrete costs.

Balanced PIR [12] improves the concrete online cost by using WR-PIR only as a fallback behind a faster primary random-set scheme. However, WR-PIR remains the bottleneck in preprocessing, and privacy requires hiding when the fallback is used. This forces every online response to download $O(\sqrt{n})$ database entries.

These works show that the κ factor is not inherent, but leave open whether one can remove it with a simpler approach while keeping communication low. This motivates the following question:

Can preprocessing PIR match the tight space-time tradeoff using simple random-set techniques, while preserving low online communication and practical concrete costs?

1.1. Our Contributions

In this paper, we give a positive answer through two constructions. The first construction, MULTI-LEVEL PIR, matches the tight storage-computation tradeoff using only simple random-set components. The second construction, MULTI-LEVEL++ PIR, preserves the same storage and computation bounds while reducing online communication to $O_\lambda(n^{1/4})$.

Main idea: use unreliable levels deliberately. Random-set preprocessing PIR has two natural knobs: more sets improve coverage but increase client storage; larger sets also improve coverage but increase online computation time, since the set must be expanded for the query. No single parameter setting removes the κ-factor storage overhead without paying elsewhere. Our key idea is to compose several imperfect schemes in a waterfall: the cheap, early level handles most queries, while progressively heavier levels are invoked only after earlier levels fail. With a geometric parameter schedule, the client storage remains a convergent series of size $O(\sqrt{n})$, and the expected online work remains $O(\sqrt{n})$ because expensive levels are reached with rapidly decreasing probability.

MULTI-LEVEL PIR: optimal space-time tradeoff PIR. Using this multi-level composition, MULTI-LEVEL PIR achieves $S = O(\sqrt{n})$ client storage and expected $T = O(\sqrt{n})$ online computation, matching the lower bound $S \cdot T = \Omega(n)$ up to constant factors. At a high level, the client stores hints for all levels and queries them in order: if the current level has no set containing the queried index, the client proceeds to the next level. We halve the number of sets and quadruple the set size from one level to the next, so each later level has twice the expected coverage but is invoked much less often. After $\lceil \log_2 \kappa \rceil$ levels, the remaining failure probability becomes negligible.

Theorem 1 (MULTI-LEVEL PIR: Optimal Space-Time Tradeoff PIR)**.** *Assuming one-way functions, there exists a single-server client-specific preprocessing PIR scheme for a database of n entries with constant entry size that achieves:*

- ***Storage:*** *$O(\sqrt{n})$ client, no additional server storage;*
- ***Preprocessing:*** *$\widetilde{O}_\lambda(n)$ client time, $O(n)$ server time, $O(n)$ communication;*
- ***Online Query:*** *Expected $O(\sqrt{n})$ client/server time (counted in PRF evaluations), $O(\sqrt{n})$ communication, negligible failure probability.*

Key technical issue: privacy under observable failures. A naive multi-level composition is not private. If the server sees that the client failed in early levels and succeeded only later, the server learns predicates about the client's current hint state. Those predicates can become harmful to privacy if the affected hints remain correlated with future queries. Balanced PIR avoids this problem by hiding whether the fallback is used, but this forces every online transcript to have the same large download shape. We take the opposite approach: we allow the server to observe which levels fail, and then refresh the invoked levels so that the observed failure pattern cannot be reused. The refresh uses the recovered database value to resample precisely the hint state correlated with the observed failure, restoring the distribution needed for future queries.

MULTI-LEVEL++ PIR: reducing online communication to $O_\lambda(n^{1/4})$. Our second construction reduces the latency-critical online communication by replacing ordinary random sets with a QuarterPIR-style privately programmable pseudorandom set (PPPS) representation [11]. It turns out to be highly challenging, as direct use of QuarterPIR would require many chunk-specific backup hints and would reintroduce the κ-factor storage overhead. We redesign the programmable backup and refresh mechanism so that the multi-level construction keeps $O(\sqrt{n})$ client storage while sending only $O_\lambda(n^{1/4})$ online communication. This gives, to our knowledge, the first constant-factor-optimal preprocessing PIR construction with sub-$\sqrt{n}$ online communication. The point is that compressing the primary query alone is insufficient: Balanced PIR, for example, would have to download $O(\sqrt{n})$ entries in every fallback-hiding transcript.

Theorem 2 (MULTI-LEVEL++ PIR: Optimal Tradeoff PIR with $O_\lambda(n^{1/4})$ Online Communication)**.** *Assuming one-way functions, there exists a single-server client-specific preprocessing PIR scheme for a database of n entries with constant entry size that achieves:*

- ***Storage:*** *$O(\sqrt{n})$ client, no additional server storage;*
- ***Preprocessing:*** *$\widetilde{O}_\lambda(n)$ client time, $O(n)$ server time, $O(n)$ communication;*
- ***Online Query:*** *Expected $O(\sqrt{n})$ client/server time (counted in PRF evaluations), $O_\lambda(n^{1/4})$ communication, negligible failure probability.*

TABLE 1: **Comparison of notable client-specific preprocessing PIR schemes and our results**. **"Assumption"** is the cryptographic assumption. **"Online Time"** denotes online computation per query. **"Online Communication"** denotes latency-critical query communication. **"Amortized Offline Communication"** denotes offline/background communication amortized per query. **"Servers"** is the number of servers used by the stated construction, and any single-server result immediately implies a two-server result. **"Evaluation"** is ✓ if the corresponding practical evaluation is provided.
†: Tightly match the known storage-computation tradeoff up to constant factors [8, 13].

Scheme	Assumption	Online Time	Client Space	Online Communication	Amortized Offline Communication	Servers	Evaluation
[6]	OWF	$\widetilde{O}(\sqrt{n})$	$\widetilde{O}(\sqrt{n})$	$\widetilde{O}(\sqrt{n})$	$\mathsf{polylog}(n)$	2	–
[8]	LWE	$\widetilde{O}(\sqrt{n})$	$\widetilde{O}(\sqrt{n})$	$\widetilde{O}(\sqrt{n})$	$O(\sqrt{n})$	1	–
[17, 18]	LWE	$\widetilde{O}(\sqrt{n})$	$\widetilde{O}(\sqrt{n})$	$\widetilde{O}(1)$	$O(\sqrt{n})$	1	–
TreePIR [14]	Various	$\widetilde{O}(\sqrt{n})$	$\widetilde{O}(\sqrt{n})$	$\widetilde{O}(1)$	$\mathsf{polylog}(n)$	2	✓
Piano [9] S3PIR [10] Plinko [15]	OWF	$\widetilde{O}(\sqrt{n})$	$\widetilde{O}(\sqrt{n})$	$O(\sqrt{n})$	$O(\sqrt{n})$	1	✓
QuarterPIR [11]	OWF	$\widetilde{O}(\sqrt{n})$	$\widetilde{O}(\sqrt{n})$	$\widetilde{O}(n^{1/4})$	$O(\sqrt{n})$	1	✓
WR PIR [16]†	OWF	$O(\sqrt{n})$	$O(\sqrt{n})$	$O(\sqrt{n})$	$O(\sqrt{n})$	1	–
Balanced PIR [12]†	OWF	$O(\sqrt{n})$	$O(\sqrt{n})$	$O(\sqrt{n})$	$O(\sqrt{n})$	1	✓
MULTI-LEVEL PIR †	OWF	$O(\sqrt{n})$	$O(\sqrt{n})$	$O(\sqrt{n})$	$O(\sqrt{n})$	1	✓
MULTI-LEVEL++ PIR †	OWF	$O(\sqrt{n})$	$O(\sqrt{n})$	$O(n^{1/4})$	$O(\sqrt{n})$	1	–

Empirical evaluation. We implement MULTI-LEVEL PIR and compare it with Piano, S3PIR, and Balanced PIR [9, 10, 12]. The evaluation supports the intended asymptotic behaviors: MULTI-LEVEL PIR reaches the low-storage regime of Balanced PIR while avoiding its expensive WR-PIR fallback machinery. Across our evaluated settings, MULTI-LEVEL PIR reduces preprocessing time by about 8–$45\times$ and online communication by about 5–$67\times$ compared with Balanced PIR. For a 64 GiB database with 2^{28} entries, MULTI-LEVEL PIR uses 75 MB of client storage, compared with 837.75 MB for Piano, 664.25 MB for S3PIR, and 70.22 MB for Balanced PIR. In the same setting, MULTI-LEVEL PIR completes preprocessing $8.27\times$ faster than Balanced PIR and uses $66.6\times$ less online bandwidth (62 KB versus 4.13 MB), while keeping comparable online time (7.4 ms versus 6.4 ms). The implementation is also compact: our MULTI-LEVEL PIR prototype is about 400 lines of C++ code; for comparison, Balanced PIR reports close to 500 lines for its primary/verifiable-primary implementation and close to 800 lines for its backup/verifiable-backup implementation [12]. Taken together, MULTI-LEVEL PIR is a well-rounded candidate for practical deployment: among the evaluated schemes, it is either the best or among the most competitive choices in client storage, preprocessing time, online bandwidth, latency, and implementation complexity.

In summary, our contributions are as follows.

- **MULTI-LEVEL PIR.** We construct a preprocessing PIR scheme with $O(\sqrt{n})$ client storage and expected $O(\sqrt{n})$ online computation, matching the tight $S \cdot T = \Theta(n)$ tradeoff up to constant factors.
- **MULTI-LEVEL++ PIR.** We construct a second scheme that preserves the same storage and computation bounds while reducing online communication to $O_\lambda(n^{1/4})$.
- **Empirical evaluation.** We implement MULTI-LEVEL PIR and compare it with Piano, S3PIR, and Balanced PIR. The experiment shows that MULTI-LEVEL PIR is highly competitive for deployment. Our code is available at https://anonymous.4open.science/r/Multi-level-PIR-3B6E.

1.2. Organization of the Paper

The rest of the paper is organized as follows. In Section 2, we give an overview of the technical challenges and our approaches. In Section 3, we give the formal definitions of client-specific preprocessing PIR and some preliminaries. In Section 4, we present our first construction that matches the storage-computation lower bound with conceptually simple techniques. In Section 5, we present our second construction that optimizes the online communication cost to $O_\lambda(n^{1/4})$ per query. In Section 6, we present our implementation and evaluation results. In Section 7, we review some related works. In Section 8, we present our conclusion.

2. Technical Overview

In this section, we give a high-level technical overview of our constructions. We first introduce the necessary background on preprocessing PIR, and then describe the main ideas behind our schemes.

2.1. Background: Client-Specific Preprocessing PIR

Basic Construction. Most prior works on preprocessing PIR [8, 19, 14, 17, 9, 10, 15, 16] follow the general framework of random-set coverage, pioneered by Stateful PIR [20] and Corrigan-Gibbs and Kogan [6]. For simplicity of exposition, we use Piano PIR scheme [9] as our starting point.

In Piano, the database is partitioned into $\sqrt{n}$ chunks, each containing $\sqrt{n}$ entries. During preprocessing, the client samples $M = O(\kappa\sqrt{n})$ random sets $\mathsf{Set}_1, \mathsf{Set}_2, \ldots, \mathsf{Set}_M$, each containing exactly one random index from each chunk.

The client also computes the "parity" for each set, i.e., $p_i = \bigoplus_{j \in \mathsf{Set}_i} \mathsf{DB}[j]$, and stores all these parities locally. In the query phase, to retrieve the entry at index x, the client finds a set Set_i containing x. To hide the actual query x from the server, the client replaces x with a random index y from the same chunk and queries the parity of the "edited" set $\mathsf{Set}_i \cup \{y\} \setminus \{x\}$. From the server's view, this queried set is indistinguishable from a uniformly random set, so privacy is preserved. XORing the edited set's parity with p_i recovers $\mathsf{DB}[x] \oplus \mathsf{DB}[y]$. Since the replacement index y and its corresponding entry $\mathsf{DB}[y]$ are prepared during the preprocessing phase, the client can fully recover $\mathsf{DB}[x]$.

To handle multiple queries, one must be careful about *replacing consumed random sets*: a single random set cannot be reused for multiple entries due to a simple linkage attack. Thus, the client prepares $\tilde{O}(\sqrt{n})$ backup sets during preprocessing and replaces each consumed set with a backup after every query. However, the replacement process involves several subtleties, because the client must *maintain the distribution* of the primary table's random sets to avoid statistical leakage. In particular, replacing a set known to be containing x requires a backup set to also contain x. This can be done by making "edits" to random backup sets, instead of actually preparing backups to actually cover x.

The lower bound of $S \cdot T = \Omega(n)$. The lower bound proved by Corrigan-Gibbs and Kogan [6] states that for any single-server preprocessing PIR scheme with S-bit per-client hint and T query time, it must hold that $S \cdot T = \Omega(n)$. This was further generalized by subsequent works to the batched and multiple adaptive query settings [8, 13]. We provide the intuition behind this lower bound in Appendix A.

2.2. Matching the Lower Bound: $O(\sqrt{n})$ Client Storage with $O(\sqrt{n})$ Online Computation

Understanding $O(\kappa\sqrt{n})$ client-storage overhead in prior works. Prior practical works using the random-set framework, including Piano [9], S3PIR [10], and QuarterPIR [11], require $\tilde{O}(\kappa\sqrt{n})$ client storage to achieve $\tilde{O}(\sqrt{n})$ online computation. Here κ is the statistical security parameter (i.e., achieving $\mathsf{negl}(\kappa)$ failure probability). In practice, κ is typically set to 40 or higher.

This storage blowup stems from the correctness requirement that *all* n database indices must be "covered" by at least one random set with overwhelming probability, e.g., probability at least $1-2^{-\kappa}$. When each set contains C random indices, a coupon-collector argument shows that $\Theta(\kappa \cdot n/C)$ sets are needed to cover all n indices with overwhelming probability. Since each set requires at least one stored parity value, the κ factor in storage is unavoidable in this approach.

Our insight. Rather than preventing failures, we allow them to occur with noticeable probability but ensure that they reveal *no information* about the queried index. Our key observation is that in PRF-based random-set PIR schemes, increasing the computation (by enlarging the random-set size) improves correctness without increasing the storage per set, since each random set still requires only a PRF key plus a parity value. This decoupling enables us to build PIR instances with varying computation-correctness tradeoffs while maintaining low storage costs.

We exploit this by designing a multi-level composition. Computationally efficient but less reliable PIR instances handle queries with high probability, while computationally expensive but highly reliable instances are invoked only after earlier levels fail. This yields optimal *expected* online costs.

First attempt: two-level recursion with $O(\log \kappa)$ storage and computation overhead. We first describe a two-level construction. A lightweight first-level scheme handles most queries efficiently, while a heavier second-level scheme serves as a fallback for the rare cases in which the first level fails.

- **First-level scheme (invoked first):** A random-set based scheme with set size $C = \sqrt{n}$ and $O(\sqrt{n}\log\kappa)$ sets. This achieves $1 - 1/\kappa$ correctness probability.
- **Second-level scheme (heavier, full correctness):** A fallback scheme with set size $C' = \kappa\sqrt{n}$ and $O(\sqrt{n})$ sets. This achieves negligible failure probability and is only invoked when the first level fails.

This two-level approach reduces the client storage overhead from κ to $\log\kappa$. The detailed parameters are listed in Table 2. The efficiency analysis is as follows:

- **Computation:** The first-level scheme requires amortized $O(\sqrt{n}\log\kappa)$ computation per query. The second-level scheme requires $O(\kappa\sqrt{n})$ computation but is only invoked with probability $1/\kappa$. Thus, the expected online computation remains $O(\sqrt{n}\log\kappa)$.
- **Storage:** Since the storage per set is fixed (one PRF key plus one parity bit) regardless of set size, the total storage is $O(\sqrt{n}\log\kappa) + O(\sqrt{n}) = O(\sqrt{n}\log\kappa)$.

Generalization to multi-level recursion with $O(\sqrt{n})$ client storage. The above two-level recursion generalizes naturally to multiple levels, achieving $O(\sqrt{n})$ client storage. Specifically, we use $d = \lceil \log_2 \kappa \rceil$ levels, with geometrically fewer sets and larger sets per level. This doubles the expected coverage at each level while keeping the total number of stored sets bounded by a geometric series. The detailed parameters are listed in Table 3. This achieves expected $O(\sqrt{n})$ online computation and client storage, while still attaining negligible failure probability.

The privacy challenge: dealing with potential information leakage from early level failure. A technical subtlety is how to preserve privacy when an early level fails. If the server can observe whether each level succeeds or fails, then a failure event may reveal information about the client's hint structure. This issue also appears in Balanced PIR [12], which uses a similar two-level technique. Balanced PIR addresses it by carefully designing the query protocols of PIR-PRIMARY and PIR-BACKUP so that the server sees identically distributed messages regardless of which layer is invoked. This design is *workable but inefficiency*. For each query, the client downloads $O(\sqrt{n})$ entries from the server. When entries are large, this becomes expensive. For example,

Level	# Sets	Set Size	Fail Prob.	Exec. Prob.	Storage	Comp.	Comm.
1	$\sqrt{n}\log\kappa$	$\sqrt{n}$	$1/\kappa$	1	$\sqrt{n}\log\kappa$	$\sqrt{n}\log\kappa$	$\sqrt{n}$
2	$\sqrt{n}$	$\sqrt{n}\kappa$	$\mathsf{negl}(\kappa)$	$1/\kappa$	$\sqrt{n}$	$\sqrt{n}\kappa$	$\sqrt{n}\kappa$
Total	–	–	$\mathsf{negl}(\kappa)$	–	$O(\sqrt{n}\log\kappa)$	$O(\sqrt{n}\log\kappa)$	$O(\sqrt{n})$

TABLE 2: Parameters for the two-level recursive PIR composition. Level 1 always executes; Level 2 executes with probability $1/\kappa$ upon a Level 1 failure. **Fail Prob.** is the conditioned per-level failure probability when that level is executed. **Exec. Prob.** is the per-query execution probability. **Comp./Comm.** are the expected per-query online computation/communication.

Level	# Sets	Set Size	Fail Prob.	Exec. Prob.	Storage	Comp.	Comm.
1	$\sqrt{n}$	$\sqrt{n}$	$1/e$	1	$\sqrt{n}$	$\sqrt{n}$	$\sqrt{n}$
2	$\frac{1}{2}\sqrt{n}$	$4\sqrt{n}$	e^{-2}	$1/e$	$\frac{1}{2}\sqrt{n}$	$4\sqrt{n}$	$4\sqrt{n}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
i	$2^{1-i}\sqrt{n}$	$4^{i-1}\sqrt{n}$	$e^{-2^{i-1}}$	$e^{-(2^{i-1}-1)}$	$2^{1-i}\sqrt{n}$	$4^{i-1}\sqrt{n}$	$4^{i-1}\sqrt{n}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\lceil\log_2\kappa\rceil$	$\Theta(\sqrt{n}/\kappa)$	$\Theta(\kappa^2\sqrt{n})$	$\mathsf{negl}(\kappa)$	$e^{-\Theta(\kappa)}$	$\Theta(\sqrt{n}/\kappa)$	$\Theta(\kappa^2\sqrt{n})$	$\Theta(\kappa^2\sqrt{n})$
Total	$< 2\sqrt{n}$	–	$\mathsf{negl}(\kappa)$	–	$O(\sqrt{n})$	$O(\sqrt{n})$	$O(\sqrt{n})$

TABLE 3: Parameters for a multi-level recursive PIR composition with $d = \lceil\log_2\kappa\rceil$ levels. Each level i has M_i sets with expected selected-set size C_i. Level 1 always executes, and each subsequent level i executes only upon a Level $i-1$ failure. Notations **Fail Prob.**, **Exec. Prob.** and **Comp./Comm.** have the same meanings as Table 2. For better readability, we present the expressions without the big O notations.

in a 128 GB database with 2^{28} entries of size 512 B, the online bandwidth per query can be up to 8 MB.

Thus, we adopt a different approach. The key observation is that a failure at index x does not immediately compromise x itself; rather, it reveals a predicate about the relationship between x and the client's current hint table. Such information can become useful to the adversary only if it remains correlated with the client's *future* hint state.

We therefore introduce a novel refresh mechanism for random-set PIR that *maintains privacy even under observable failures* – a strictly stronger guarantee than standard PIR privacy definitions. After each query, the client runs the refresh mechanism for the invoked level and all prior ones. In this way, the refreshed hint distribution is statistically independent of this observation, even though the adversary observes whether a query succeeded or failed at each level. Thus, the server will not learn any reusable information about the hint table or the queried indices. The communication cost in our scheme is only $O(\sqrt{n})$ indices plus one parity value. In the same 128 GB database, this corresponds to roughly 32.5 KB of online bandwidth per query.

The simplified refresh mechanism. For technical completeness, we describe a simplified version of our refresh mechanism below for a single level, which is slightly inefficient but remains secure.

For a query at index x, the client partitions its local hint sets (suppose their number is M) into $\bar{S}_{+x} = \{\mathsf{Set}_i : x \in \mathsf{Set}_i\}$ (matching sets) and $\bar{S}_{-x} = \{\mathsf{Set}_i : x \notin \mathsf{Set}_i\}$ (non-matching sets). If $\bar{S}_{+x} \neq \emptyset$, the client selects a random set from $\bar{S}_{+x}$ to answer the query; otherwise, it outputs $\perp$.

The critical observation is that the adversary learns the binary predicate $|\bar{S}_{+x}| > 0$ from observing success or failure. Our refresh procedure eliminates this information leakage as follows. After the query, the client resamples the cardinality $K \leftarrow \mathrm{Binom}(M, p)$ where $p = \Pr_{S\leftarrow\mathcal{D}}[x \in S]$, i.e., the probability of an index being included in a freshly-generated random set. This is precisely the distribution of $|\bar{S}_{+x}|$ for freshly sampled sets. The client then generates K fresh sets conditioned on containing x (forming $\bar{S}'_{+x}$) and adjusts $\bar{S}_{-x}$ to maintain exactly M total sets (forming $\bar{S}'_{-x}$). This ensures that the post-refresh distribution $\bar{S}'_{+x} \cup \bar{S}'_{-x}$ is identical to freshly sampled sets, *regardless* of whether the query succeeded.

The above mechanism and guaranteed privacy demonstrate the security of our multi-level construction. However, it is slightly inefficient since it consumes too many backup hints per query. In Section 4, we will describe a detailed and more efficient version, in which the hint replenishment mechanism is optimized.

Further reducing the online communication to $O_\lambda(n^{1/4})$. The privately programmable pseudorandom sets (PPPS) proposed in Quarter PIR [11] provides a tool to compress the communication content. Their key idea is to construct a two-layer PRF, so that the client only sends $O(n^{1/4})$ PRF keys and $O(n^{1/4})$ programmed indices to the server. The server computes all $O(n^{1/4})$ possibilities and sends back all of them. However, directly applying the method of Quarter PIR does not work in our setting.

- **Challenge:** In these settings, the client promotes a backup hint to replace a consumed primary one to preserve the distribution of the hint table. The promoted hint should also have a short representation: $O(n^{1/4})$ superblock keys and explicit local descriptors. Quarter-PIR handles this problem with chunk-specific backup hints, where a backup hint can only be promoted for one designed chunk. Thus, each chunk requires $\Theta(\kappa)$ reserved backup hints to avoid backup exhaustion with high probability, which leads to $O(\kappa\sqrt{n})$ client storage.

Therefore, we make a non-blackbox use of PPPS and carefully redesign both the query protocol and the refresh mechanism. We will describe the details in Section 5.

3. Notation, Definitions, and Preliminaries

3.1. Notation

The computational security parameter is denoted by λ, and the statistical correctness parameter by κ. A function $f(\lambda)$ is negligible, written $\mathsf{negl}(\lambda)$, if $f(\lambda) = o(\lambda^{-c})$ for every constant $c > 0$. We write $\mathsf{poly}(\cdot)$ for an unspecified polynomial.

For an integer n, we write $[n] = \{0, \ldots, n-1\}$. Unless otherwise stated, the database is $\mathsf{DB} \in \{0,1\}^n$. The parameter S denotes client storage in bits, T denotes online computation per query, and q denotes the number of online queries. We use $\widetilde{O}(\cdot)$ to suppress polylogarithmic factors in n. The notation $O_\lambda(\cdot)$ hides polynomial factors in λ, and $\widetilde{O}_\lambda(\cdot)$ hides both such factors and polylogarithmic factors.

We use PRF and PRP for pseudorandom functions and pseudorandom permutations, with secret key sk and master key msk when needed. The symbols Sim and $\mathcal{A}$ denote a simulator and an adversary, respectively. By default, the server is modeled as a probabilistic polynomial-time semi-honest adversary.

3.2. Definitions

Single-server preprocessing PIR. A single-server preprocessing PIR scheme is an interactive protocol between a stateful client and a stateful server. It has two phases:

- **Preprocessing phase.** The server starts with the database $\mathsf{DB} \in \{0,1\}^n$, while the client starts with no database input. The parties may interact and then retain local state for future queries.
- **Query phase.** For each requested index $x \in [n]$, the client interacts with the server using the state produced by preprocessing and outputs a value β.

Correctness. Correctness requires that honest executions answer the requested database entries except with negligible probability. Formally, for any $n, q = \mathsf{poly}(\lambda)$, any database $\mathsf{DB} \in \{0,1\}^n$, and any query sequence $x_1, \ldots, x_q \in [n]$, the client's outputs $\beta_1, \ldots, \beta_q$ satisfy

$$(\beta_1, \ldots, \beta_q) = (\mathsf{DB}[x_1], \ldots, \mathsf{DB}[x_q])$$

with probability at least $1 - \mathsf{negl}(\lambda) - \mathsf{negl}(\kappa)$, where the probability is over the honest parties' randomness.

Privacy. Privacy says that the server's transcript can be simulated without knowing the client's query sequence. We use the following simulation-based formulation. For every p.p.t. server adversary $\mathcal{A}$, every $n, q = \mathsf{poly}(\lambda)$, and every database DB, there exists a p.p.t. simulator $\mathsf{Sim}(1^\lambda, n, q)$ such that the server views in the following experiments are computationally indistinguishable:

- Real: an honest client executes the protocol with $\mathcal{A}(1^\lambda, n, \mathsf{DB})$, which plays the server. The adversary may adaptively choose the next client query in each online step.
- Ideal: $\mathsf{Sim}(1^\lambda, n, q)$ interacts with $\mathcal{A}(1^\lambda, n, \mathsf{DB})$ and produces a simulated server view without receiving the chosen query indices.

4. Matching the Space-Time Lower Bound with Multi-Level PIRs

This section gives our first construction: a client-specific preprocessing PIR scheme with $S = O(\sqrt{n})$ client storage and expected $T = O(\sqrt{n})$ online computation, matching the known $S \cdot T = \Omega(n)$ lower bound up to constants.

The main difficulty is failure privacy. A level may fail, and the server observes this failure event. After each query x, the client must therefore refresh the primary table so that it is again distributed as freshly sampled, even conditioned on whether the level succeeded. We first build a single level with this fresh-table invariant, then compose levels so that a lower level repairs failed upper levels.

4.1. Single-Level Hints and One Private Query

Hint representation. Each level stores many *primary hints*. A primary hint is a compact description of a random set, together with the XOR parity of the database entries in that set. For any set $S \subseteq [n]$, write $\mathsf{Parity}(S) = \bigoplus_{a \in S} \mathsf{DB}[a]$. Fix a chunk size B and split DB into n/B consecutive chunks. A PRF key determines one offset $\Delta_i \in [B]$ in each chunk i, so the hint samples one database index from every chunk. It also samples a color bit $s_i \in \{0,1\}$ per chunk and a hidden *primary color selector* $s^* \in \{0,1\}$. The selected set is the sampled indices with color s^*:

$$\mathsf{Set}(H) = \{iB + \Delta_i \mid s_i = s^*\}.$$

The client stores only the compact hint description and the parity $p = \mathsf{Parity}(\mathsf{Set}(H))$. The hidden selector is what lets the client edit a query without revealing which color class is real.

Making a single private query. Suppose the client wants to query an index $x = jB + k$ that appears in a primary hint H. The client edits only chunk j: it flips the color bit, replaces the offset by a fresh random offset, and sends the edited vectors while keeping s^* secret. The server returns the two color-class parities of the edited hint. For the hidden

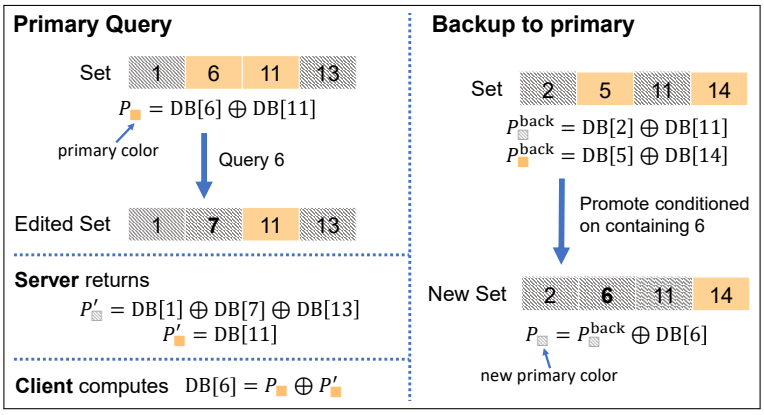


Figure 1: Demonstration of our single-level PIR construction in MULTI-LEVEL PIR. Left: a primary hint samples indices $(1, 6, 11, 13)$ with color vector $(0, 1, 1, 0)$, and stores its parity. The primary color selector is 1, so the set should be $\{6, 11\}$. To query 6, the client edits only that chunk by replacing 6 with a fresh offset and flipping its color. The server returns both edited color parities, and the client recovers $\mathsf{DB}[6]$ by taking the XOR with the stored parity. Right: a backup hint stores both color parities. To promote it into a primary hint conditioned on containing 6, the client overwrites the index in the queried chunk to 6 and flips the color bit, then picks the primary color and adjusts the parity using $\mathsf{DB}[6]$.

selector, this parity is exactly $\mathsf{Parity}(\mathsf{Set}(H) \setminus \{x\})$, so the client recovers

$$\mathsf{DB}[x] = p \oplus \mathsf{Parity}(\mathsf{Set}(H) \setminus \{x\}).$$

Figure 1 gives a concrete example, and Figure 2 gives the precise query procedure.

Primary hint and single-query procedure

SamplePrimary(B)**.**

1) Sample a PRF key sk and expand it into one color bit and one offset per chunk: $(s_i, \Delta_i) \leftarrow \mathsf{PRF}(\mathsf{sk}, i)$ for every chunk i.
2) Sample a hidden primary selector $s^* \xleftarrow{\$} \{0, 1\}$ and define $\mathsf{Set}(H) = \{iB + \Delta_i \mid s_i = s^*\}$.
3) Store the primary hint $H = (\mathsf{sk}, s^*, p)$, where $p = \mathsf{Parity}(\mathsf{Set}(H))$.

QueryPrimary(H, x) **for** $x = jB + k$**.**

1) If $x \notin \mathsf{Set}(H)$, return $\perp$.
2) Expand sk to (s, Δ) and enforce the stored overwrite of H, if any. Set $s'[j] \leftarrow 1 - s[j]$ and $\Delta'[j] \xleftarrow{\$} [B]$; keep all other chunks unchanged.
3) Send (s', Δ') to the server, but keep s^* local.
4) The server returns $r_b = \mathsf{Parity}(\{iB + \Delta'_i \mid s'_i = b\})$ for $b \in \{0, 1\}$.
5) Output $\mathsf{DB}[x] = p \oplus r_{s^*}$ and mark H as consumed.

Figure 2: The primary hint structure and how a single query is made in a single-level PIR. The underlined step is relevant only when H was promoted from a backup hint; for a freshly sampled primary hint, it has no effect. Figure 3 explains the overwrite in detail.

4.2. Backups and Failure-Recoverable Refresh

As in prior random-set PIR schemes [9, 10], a primary hint can be consumed only once. After a successful query, the matching hint has been used to recover $\mathsf{DB}[x]$ and must be removed. After a failed query, the server learns that the current table contains no matching primary hint. Both branches bias the table distribution, so the client must repair the table before the next query.

Backup hint structure. A backup hint uses the same sampled offsets and chunk colors as a primary hint, but it does not choose a primary selector yet. Instead, it stores both color parities:

$$q_b = \bigoplus_{i:s_i=b} \mathsf{DB}[iB + \Delta_i] \qquad \text{for } b \in \{0, 1\}.$$

Thus a backup is an uncommitted random set: during refresh, the client can choose which color class becomes primary and can promote the backup into a fresh primary hint conditioned either on containing x or on excluding x.

Failure-recoverable refresh. For a fixed query index x, let

$$\mathcal{D}_{\mathsf{in}}(x) := (H \mid x \in \mathsf{Set}(H)),$$
$$\mathcal{D}_{\mathsf{out}}(x) := (H \mid x \notin \mathsf{Set}(H)),$$

where H is a freshly sampled primary hint. Refresh must return the primary table to the distribution of independent fresh hints, not the distribution conditioned on the observed success or failure event.

Promotion to $\mathcal{D}_{\mathsf{in}}(x)$ is direct: overwrite the queried chunk with x, select the color that includes it, and adjust the stored parity using $\mathsf{DB}[x]$. Promotion to $\mathcal{D}_{\mathsf{out}}(x)$ samples uniformly from the $4B - 2$ local triples that exclude x; Lemma 8 verifies the distribution. The table-level refresh uses these samplers to restore the fresh-table invariant. After a failure, it inserts a freshly sampled number of containing

Backup promotion and single-level refresh

SampleBackup(B)**.** Sample sk and expand it into (s_i, Δ_i) for every chunk. Store $\bar{H} = (\mathsf{sk}, \perp, q_0, q_1)$, where $q_b = \mathsf{Parity}(\{iB + \Delta_i : s_i = b\})$.

PromoteIn$(\bar{H}, x, v)$ **for** $x = jB + k$ **and** $v = \mathsf{DB}[x]$**.**

1) Expand sk to (s, Δ); overwrite $(s[j], \Delta[j]) \leftarrow (1 - s[j], k)$.
2) Set the selected color $s^* \leftarrow s[j]$ after the overwrite.
3) Output $(\mathsf{sk}, s^*, q_{s^*} \oplus v)$ with this overwrite.

PromoteOut$(\bar{H}, x)$ **for** $x = jB + k$**.**

1) Expand sk to (s, Δ).
2) If $\Delta[j] = k$, set $s^* \leftarrow 1 - s[j]$. With probability $(B-1)/(2B-1)$, also resample $\Delta[j] \xleftarrow{\$} [B] \setminus \{k\}$.
3) If $\Delta[j] \neq k$, set $s^* \leftarrow s[j]$ with probability $B/(2B-1)$; otherwise set $s^* \leftarrow 1 - s[j]$.
4) Output $(\mathsf{sk}, s^*, q_{s^*})$ with the chosen overwrite, if any.

Refresh(x, v, status) **for** $v = \mathsf{DB}[x]$**.**

1) If $\mathsf{status} = \mathsf{fail}$, sample $K \leftarrow \mathrm{Binom}(M, 1/(2B))$. Replace K uniform primary slots by PromoteIn$(\cdot, x, v)$ outputs.
2) If $\mathsf{status} = \mathsf{success}$, replace the consumed hint by PromoteIn$(\cdot, x, v)$. With probability $(1-1/(2B))^M$, replace every primary hint that now contains x by PromoteOut$(\cdot, x)$.
3) Delete every backup hint used for promotion.

Figure 3: Pseudocode for backup hints, conditional promotion, and single-level refresh.

hints. After a success, it replenishes the consumed hint and, with the probability shown in Figure 3, converts all currently matching hints into excluding hints. Figure 3 gives the concrete routines.

4.3. Oracle-Relative Single-Level PIR

We now package the preceding primitives into one oracle-relative level of PIR instance. The level can answer a query when some primary hint contains x. If it fails, it still needs $\mathsf{DB}[x]$ to repair the table, but the PIR instance itself does not have this capability. In our full construction, the $\mathsf{DB}[x]$ comes from later level's successful query. Here, we abstract it as an ideal oracle $\mathcal{O}_{\mathsf{DB}}$ that returns the correct database value, and present the single-level PIR instance with respect to this oracle.

Theorem 3 (Oracle-relative single-level guarantee)**.** *Fix any adaptive query sequence that is independent of the client's preprocessing randomness. Conditioned on the backup pool not being exhausted, the oracle-relative level in Figure 4 satisfies:*

Oracle-relative single-level PIR

Preprocessing(M, B)**.** Sample M primary hints and a backup pool, then compute all primary and backup parities in one streaming pass over DB.

Probe(x) **for** $x = jB + k$**.**

1) Let $\mathsf{Cand} \leftarrow \{H \in \mathsf{Primary} : x \in \mathsf{Set}(H)\}$. If $\mathsf{Cand} = \emptyset$, return $(\mathsf{fail}, \perp, \perp)$.
2) Choose $H^* \xleftarrow{\$} \mathsf{Cand}$; run QueryPrimary$(H^*, x)$ to recover v. Mark H^* consumed and return $(\mathsf{success}, v, H^*)$.

Repair$(x, v, \mathsf{status}, H^*)$**.** Run Refresh$(x, v, \mathsf{status})$ from Figure 3. If $\mathsf{status} = \mathsf{success}$, use H^* as the consumed slot.

$\mathsf{Query}^{\mathcal{O}_{\mathsf{DB}}}(x)$**.**

1) Run $(\mathsf{status}, v, H^*) \leftarrow \mathsf{Probe}(x)$.
2) If $\mathsf{status} = \mathsf{fail}$, set $v \leftarrow \mathcal{O}_{\mathsf{DB}}(x)$.
3) Run Repair$(x, v, \mathsf{status}, H^*)$ and output v.

Figure 4: Oracle-relative single-level PIR. $\mathcal{O}_{\mathsf{DB}}(x)$ returns $\mathsf{DB}[x]$ and is used only after level failure, to obtain the value needed for repair.

1) ***Correctness.*** *For every query x, the client outputs $\mathsf{DB}[x]$.*
2) ***Hint freshness.*** *After every completed query, the primary table is distributed as a fresh table of M independent primary hints, even conditioned on the observable success/failure bit of that query.*
3) ***Privacy.*** *Let $C = n/(2B)$ be the expected selected-set size, so the hint has $2C$ chunks, and let*

$$p_{\mathsf{fail}} = (1 - 1/(2B))^M.$$

The following simulator, which uses only public parameters and does not query $\mathcal{O}_{\mathsf{DB}}$, simulates the server's view. For each query t, sample

$$F_t \leftarrow \mathrm{Bernoulli}(p_{\mathsf{fail}}).$$

If $F_t = 1$, output fail. If $F_t = 0$, sample

$$(s_t, \Delta_t) \xleftarrow{\$} \{0,1\}^{2C} \times [B]^{2C}$$

and output $(\mathsf{success}, s_t, \Delta_t)$. The simulated transcript is indistinguishable from the real server view.

Proof sketch. Maintain the invariant that each query starts with a fresh primary table and fresh unused backups. On success, QueryPrimary hides x by sending a random-looking edited descriptor and recovers $\mathsf{DB}[x]$ from the stored parity. On failure, the oracle supplies $\mathsf{DB}[x]$. The only remaining issue is that success consumes a matching hint, while failure reveals that all primary hints exclude x. Repair removes this bias using two exact conditional samplers: PromoteIn samples a fresh hint conditioned on containing x, and PromoteOut

samples one conditioned on excluding x. The refresh rule recombines these conditional distributions into the original fresh-table distribution, and backup exhaustion is negligible. Thus the simulator samples failure with probability $(1 - 1/(2B))^M$; on failure it outputs only the failure event, and on success it outputs a uniformly random query descriptor of the right format. Induction over the query sequence gives the theorem. The full proof appears in Appendix C. □

4.4. Multi-Level Composition

The full scheme removes the oracle by stacking levels. Light levels run first. If they fail, the client probes heavier levels until one returns $\mathsf{DB}[x]$. The first successful level then supplies the value needed to repair all earlier failed levels.

Parametrized PIR instances. Parameterize a level by the number M of primary hints and the expected selected-set size C. Recall that a hint samples one index from each of the n/B chunks and then keeps the half whose color matches the hidden selector. Thus the expected selected-set size is $n/(2B)$. To obtain expected size C, we set $B = n/(2C)$; equivalently, a fixed index is included in a fresh primary hint with probability $(1/B) \cdot (1/2) = C/n$. This configuration yields the following complexity profile:

- **Failure probability:** $(1 - C/n)^M \approx e^{-MC/n}$.
- **Client storage:** $O(M)$ hints.
- **Online computation and communication:** $O(M + C)$ computation and $O(C)$ communication when the level is invoked.

We instantiate $d = \lceil \log_2 \kappa \rceil$ independent levels with

$$M_i = 2^{1-i}\sqrt{n}, \qquad C_i = 4^{i-1}\sqrt{n}.$$

Level i has failure probability $e^{-2^{i-1}}$ and is invoked with probability $e^{-(2^{i-1}-1)}$ (the product of all previous failure probabilities). The exponential growth in C_i is balanced by exponentially decreasing invocation probability, yielding $O(\sqrt{n})$ expected cost despite the final level having $\Theta(\kappa^2\sqrt{n})$ set size.

Theorem 4. *Instantiating the multi-level composition with $d = \lceil \log_2 \kappa \rceil$ levels, where level i has parameters $M_i = 2^{1-i}\sqrt{n}$ and $C_i = 4^{i-1}\sqrt{n}$, yields a client-specific preprocessing PIR scheme with negligible failure probability $\mathsf{negl}(\kappa)$ and the following efficiency after an $O(n)$-time preprocessing phase:*

- *Client storage: $S = O(\sqrt{n})$;*
- *Expected online computation: $T = O(\sqrt{n})$;*
- *Expected online communication: $O(\sqrt{n})$.*

Proof. The probability that all levels fail is

$$\prod_{i=1}^{d} e^{-2^{i-1}} = e^{-(2^d-1)} = e^{-\Omega(\kappa)}.$$

Storage is $\sum_i O(M_i) = O(\sqrt{n})$. Let

$$A_d = \sum_{i=1}^{d} 4^{i-1} e^{-(2^{i-1}-1)} = O(1).$$

Multi-level PIR composition

Preprocessing. Let $d = \lceil \log_2 \kappa \rceil$. For each $i \in [d]$, set $M_i = 2^{1-i}\sqrt{n}$, $C_i = 4^{i-1}\sqrt{n}$, and $B_i = n/(2C_i)$; preprocess an independent level Π_i with these parameters.

Online query for $x \in [n]$.

1) Set $\mathsf{Failed} \leftarrow \emptyset$.
2) For $i = 1, \ldots, d$, run $\Pi_i.\mathrm{Probe}(x)$ and parse the output as $(\mathsf{status}_i, v_i, H_i^*)$. If $\mathsf{status}_i = \mathsf{fail}$, append i to Failed and continue. Otherwise set $(j, v) \leftarrow (i, v_i)$ and stop probing.
3) If all levels fail, output $\perp$ and abort.
4) For every $i \in \mathsf{Failed}$, run $\Pi_i.\mathrm{Repair}(x, v, \mathsf{fail}, \perp)$. Then repair the successful level by running $\Pi_j.\mathrm{Repair}(x, v, \mathsf{success}, H_j^*)$.
5) Output v.

Figure 5: Multi-level composition with $d = \lceil \log_2 \kappa \rceil$ levels of PIR instances. The first successful level supplies the value needed to repair all earlier failed levels.

The expected online computation is therefore

$$\sum_{i=1}^{d} e^{-(2^{i-1}-1)} \cdot O(C_i + M_i) = O(\sqrt{n})(A_d + 1) = O(\sqrt{n}).$$

Communication has the same bound with C_i in place of $C_i + M_i$. □

Remark 5. This result matches the lower bound $S \cdot T = \Omega(n)$ up to constant factors, achieving $S = O(\sqrt{n})$ and $T = O(\sqrt{n})$ [6, 8, 13]. The multi-level schedule removes the κ-factor storage blowup in prior practical random-set schemes.

Theorem 6. *Assume each level satisfies the oracle-relative guarantee of Theorem 3. Then the multi-level composition in Figure 5 is private under adaptively chosen queries, with observable per-level success/failure events. More concretely, for each level i, let*

$$p_i = (1 - 1/(2B_i))^{M_i} = (1 - C_i/n)^{M_i}$$

be its failure probability. For each query t, the simulator runs the following procedure:

for $i = 1, \ldots, d$:
  $F_{t,i} \leftarrow \mathrm{Bernoulli}(p_i)$;
  if $F_{t,i} = 1$, output (i, fail) and continue;
  if $F_{t,i} = 0$, sample $(s_{t,i}, \Delta_{t,i}) \xleftarrow{\$} \{0,1\}^{2C_i} \times [B_i]^{2C_i}$,
  output $(i, \mathsf{success}, s_{t,i}, \Delta_{t,i})$ and stop.

If all sampled levels fail, the simulator outputs failures for all levels and the abort event. The resulting transcript is indistinguishable from the real server view.

Proof. The simulator above reproduces the public schedule because the real protocol invokes levels sequentially and

stops at the first successful level. By Theorem 3, each failed level is simulated by its failure event, and the first successful level is simulated by an independent random descriptor of the correct format. No descriptor is generated for later levels because they are not invoked in the real protocol. The value recovered by the first successful level is used only locally to repair failed upper levels. By Theorem 3, those repaired states are distributed as fresh tables even conditioned on failure, so replacing the oracle call by the lower-level answer adds no server-visible leakage. A standard level-by-level hybrid gives the theorem. □

5. MULTI-LEVEL++ PIR: Achieving $O_\lambda(n^{1/4})$ Online Communication

MULTI-LEVEL PIR gives $O(\sqrt{n})$ client storage and amortized $O(\sqrt{n})$ online computation. Its online message, however, still contains one offset and one color bit per chunk, giving $O(\sqrt{n})$ online communication. This is also the main communication bottleneck in practical random-set preprocessing PIR schemes such as Piano [9] and S3PIR [10]. WR PIR [16] and Balanced PIR [12] require downloading $O(\sqrt{n})$ entries, which is costly when entries are large.

Quarter PIR [11] uses privately programmable pseudorandom sets (PPPS) with list decoding from one-way functions to compress the online message to $O(n^{1/4})$ communication.

However, applying PPPS as a black box to the basic construction saves communication but introduces an $O(\kappa)$ blowup in online time, communication, and local storage. In QuarterPIR, the former two blowups are avoided by using PPPS non-black-box: the query message is carefully edited according to some history-dependent constraints.

MULTI-LEVEL++ PIR combines this communication saving technique with the refresh mechanism of MULTI-LEVEL PIR. This requires a new hint structure and a new backup-to-primary promotion algorithm. We first explain the core single-query idea that reduces communication from $O(\sqrt{n})$ to $O(n^{1/4})$. Section 5.2 then explains the additional issues that arise for many adaptive queries, with the complete construction deferred to Appendix D.

5.1. Two-Level Hints and One Query with $O_\lambda(n^{1/4})$ Communication

We first explain the single-query communication reduction. The construction keeps the same parity-editing idea as MULTI-LEVEL PIR, but changes how the edited descriptor is sent. Fix a chunk size B and write $L = n/B$ for the number of consecutive chunks. We then split the L chunks into R superblocks, each containing D consecutive chunks, so

$$R \cdot D = L, \qquad R = D = \sqrt{L} = n^{1/4}$$

in the balanced setting. Superblock i contains chunks $iD, \ldots, (i+1)D - 1$.

The two-level pseudorandom set representation. In MULTI-LEVEL PIR, one PRF key expands directly into the full length-L offset and color vectors. Here, the root key first expands into one subkey per superblock. Subkey k_i then expands into the D offsets and color bits for superblock i. Thus the expanded descriptor is the same as in MULTI-LEVEL PIR, namely one offset $\Delta[j] \in [B]$ and one color bit $s[j] \in \{0,1\}$ per chunk j, but it has a two-level representation.

Primary hint sampling

SamplePrimary(B, R, D).

1) Sample a PRF key sk and expand it into R subkeys: $k_i \leftarrow \mathsf{PRF}_1(\mathsf{sk}, i)$ for every $i \in [R]$. For every local chunk $\ell \in [D]$ in superblock i, set $(s[iD+\ell], \Delta[iD+\ell]) \leftarrow \mathsf{PRF}_2(k_i, \ell)$. Sample $s^* \xleftarrow{\$} \{0,1\}$.
2) Define $\mathsf{Set}(H) = \{jB + \Delta[j] \mid s[j] = s^*,\ j \in [L]\}$.
3) Store the primary hint $H = (\mathsf{sk}, s^*, p)$, where $p = \mathsf{Parity}(\mathsf{Set}(H))$.

Figure 6: Sampling primary hints with two-level representation.

Making a single private query with $n^{1/4}$ communication cost. The protocol in Figure 7 is the same edit-and-cancel trick as MULTI-LEVEL PIR, but with a compressed query message. Figure 8 shows a concrete example.

The client sends two objects. First, it sends one subkey per superblock, replacing the queried superblock's real subkey by a fresh random one. Second, it sends the explicitly edited length-D descriptor of the queried superblock: the target chunk's offset is replaced by a fresh one in $[B]$, and its color bit is flipped. The server evaluates the descriptor against every possible superblock.

To see the correctness of this query protocol: the values P_{i,s^*} reconstruct the selected-set parity represented by the transmitted subkeys. The client should ignore P_{j_1,s^*} because the j_1-th subkey was already replaced by k, a freshly random subkey only used for masking. Then, the term Q_{j_1,s^*} inserts the parity of the edited descriptor in the true queried superblock. Since the edit removes x and keeps all other selected entries in that superblock unchanged, the decoded value is exactly $p \oplus \mathsf{Parity}(\mathsf{Set}(H) \setminus \{x\}) = \mathsf{DB}[x]$.

Privacy follows from the same local freshness used in MULTI-LEVEL PIR. In the PRF-hybrid view, all nonqueried subkeys are uniform and the queried subkey is replaced by an independent uniform subkey, so $\boldsymbol{k}$ does not identify j_1. Inside the queried superblock, the matched condition fixes the target local pair before editing, but after the client resamples the offset and flips the hidden selected color, the sent pair is again uniform. All other local pairs in the descriptor are already uniform. Thus $(\boldsymbol{\delta}, \boldsymbol{t})$ does not identify j_2.

The online communication is therefore $O_\lambda(R+D)$, rather than one offset/color pair for each of the n/B chunks. With the balanced choice $B = \sqrt{n}$ and $R = D = \sqrt{n/B} = n^{1/4}$, this gives $O_\lambda(n^{1/4})$ communication cost.

s **Query with primary hints**

Client. Let $H = (\mathsf{sk}, s^*, p)$ be a matching primary hint for $x = jB + k$, and write $j = j_1 D + j_2$. Expand sk into subkeys $(k_0, \ldots, k_{R-1})$ and the target-superblock descriptor.

1) Set the transmitted subkeys by

$$\boldsymbol{k}[i] = \begin{cases} \tilde{k} & i = j_1, \\ k_i & i \neq j_1, \end{cases}$$

where $\tilde{k} \leftarrow \mathbf{SubkeyGen}(1^\lambda)$ is fresh.

2) For local chunk ℓ, write $g_\ell = j_1 D + \ell$. Sample $\tilde{\delta} \xleftarrow{\$} [B]$ and set the explicit edited descriptor as $\boldsymbol{\delta} = (\delta_0, \ldots, \delta_{D-1})$ and $\boldsymbol{t} = (t_0, \ldots, t_{D-1})$, where

$$(\delta_\ell, t_\ell) = \begin{cases} (\tilde{\delta}, 1 - s[j]) & \ell = j_2, \\ (\Delta[g_\ell], s[g_\ell]) & \ell \neq j_2. \end{cases}$$

3) Send $(\boldsymbol{k}, \boldsymbol{\delta}, \boldsymbol{t})$ to the server.

Server. Upon input $(\boldsymbol{k}, \boldsymbol{\delta}, \boldsymbol{t})$:

1) For every $i \in [R]$, expand k_i into $(s_i[\ell], \Delta_i[\ell])_{\ell \in [D]}$ and define

$$\mathsf{Set}_b(k_i) = \{(iD+\ell)B + \Delta_i[\ell] \mid \ell \in [D],\ s_i[\ell] = b\}.$$

2) For every $i \in [R]$ and $b \in \{0, 1\}$, define

$$E_{i,b} = \{(iD + \ell)B + \delta_\ell \mid \ell \in [D],\ t_\ell = b\}.$$

3) Return

$$P_{i,b} = \mathsf{Parity}(\mathsf{Set}_b(k_i)), \qquad Q_{i,b} = \mathsf{Parity}(E_{i,b})$$

for every $i \in [R]$ and $b \in \{0, 1\}$.

Client decoding. Recall $x = jB + k$, and $j = j_1 D + j_2$. Output

$$\mathsf{DB}[x] = p \oplus \left(\bigoplus_{i \in [R]/\{j_1\}} P_{i,s^*} \right) \oplus Q_{j_1,s^*}.$$

Figure 7: MULTI-LEVEL++ PIR: Query protocol for a single private query (simplified).

5.2. Towards Multiple Queries

What must be preserved. The single-query protocol above consumes a matching primary hint. For adaptive queries, the client must repair the table after every public branch: after a success, one matching hint has been used; after a failure, the server learns that no primary hint matched the queried index in that level. As in MULTI-LEVEL PIR, the repair step must restore the table to the distribution of fresh primary hints before the next query. In MULTI-LEVEL++ PIR, it must also preserve the compressed two-level representation; otherwise the next query would fall back to sending a full chunk descriptor and lose the $O_\lambda(n^{1/4})$ communication bound.

Why the compressed representation is delicate. In the uncompressed construction, promotion is local: the client overwrites one chunk and updates the stored parity. The overwrites are done lazily: the client just leaves a marker and executes it during the full set-expansion in future queries. The compressed representation in MULTI-LEVEL++ PIR does not allow us to perform this "full-expand-then-overwrite" trick. A promoted hint must later be queried using short superblock keys plus one explicit local descriptor. Thus the promotion step cannot leave behind arbitrary chunk-specific state; that state would either have to be revealed, sent explicitly, or supported by extra stored parities. QuarterPIR handles this by making backups chunk-specific: a backup carries auxiliary parities for one designated chunk and is used only for queries landing in that chunk. This restores correctness, but it also makes the backup pool location-dependent. Over $\Theta(\sqrt{n})$ queries spread across $\sqrt{n}$ chunks, and each chunk needs $\Theta(\kappa)$ reserved backups to avoid depletion with high probability, leading to a κ-blowup in backup storage.

How MULTI-LEVEL++ PIR repairs the table. MULTI-LEVEL++ PIR avoids this location-specific backup pool by enriching the hint representation instead. Chunks still have local colors, but now superblocks also carry colors. This extra layer lets one backup hint be promoted for any query location while keeping the resulting primary hint compatible with the compressed query format. When such a promoted hint is later used, the client reconstructs the query history and, if necessary, resamples one subkey from the right conditional distribution. The history dependence introduced by promotion is therefore localized at query time.

Putting the pieces together. The full scheme combines these promoted hints with similar refresh principle to that in MULTI-LEVEL PIR. The refresh procedure samples fresh-looking containing and excluding hints, so the server's observed branch does not bias the next table. The multi-level composition then gives negligible overall failure probability without increasing the asymptotic client storage. Appendix D gives the concrete promoted-hint structure, the localized resampling procedure, and the privacy argument for the resulting server view.

6. Evaluation

We implement our single-server PIR scheme MULTI-LEVEL PIR in Section 4 and compare it with several state-of-the-art preprocessing single-server PIR schemes, including Piano [9], S3PIR [10] and Balanced PIR [12].

We set three baselines. Piano and S3PIR are conventional random-set preprocessing PIR schemes with strong practical online latency, but they retain the κ-factor client-storage overhead. Balanced PIR is the closest low-storage baseline, which also targets the tight $S \cdot T = \Theta(n)$ storage-computation tradeoff. Thus, our comparison emphasizes Balanced PIR, while still reporting Piano and S3PIR as practical online-latency baselines later in "Comparison with other PIR schemes".

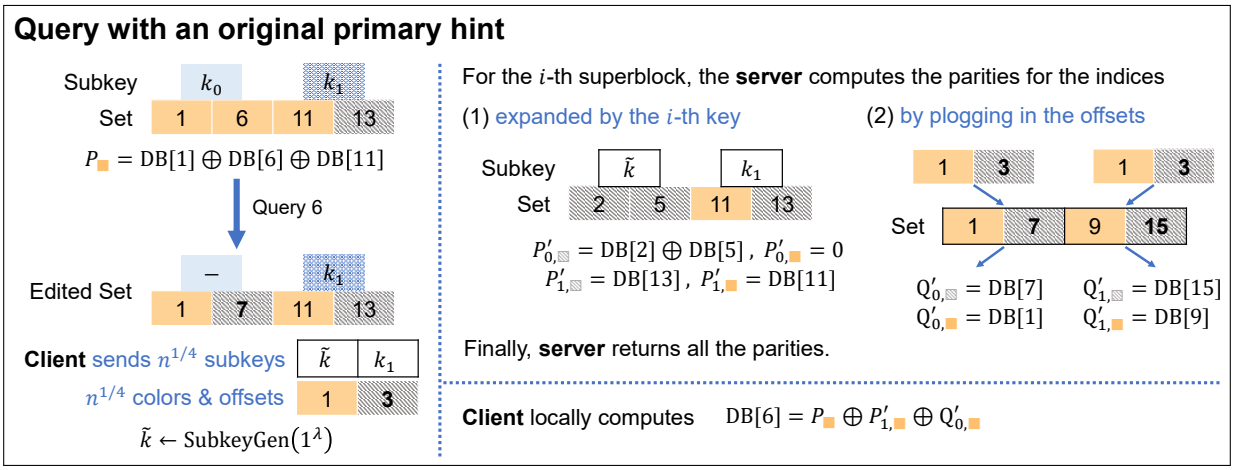


Figure 8: Demonstration of our single-level PIR construction in MULTI-LEVEL++ PIR.

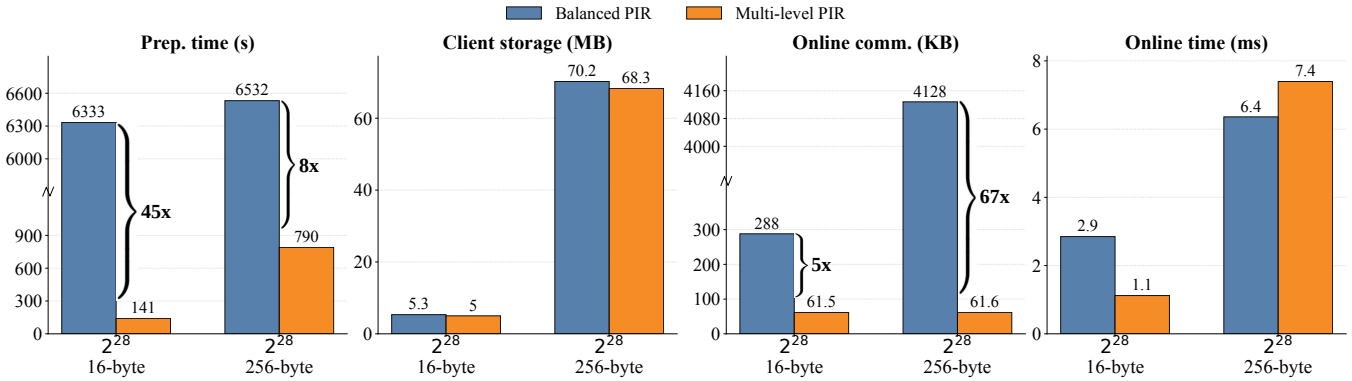


Figure 9: Performance comparison of MULTI-LEVEL PIR and Balanced PIR on 4 GB and 64 GB databases.

Our experiments show that MULTI-LEVEL PIR preserves practical online performance while substantially reducing the costs that make low-storage preprocessing PIR difficult to deploy. Compared with Piano and S3PIR, MULTI-LEVEL PIR removes most of the client-storage overhead while keeping similar online bandwidth. Compared with Balanced PIR, MULTI-LEVEL PIR reduces preprocessing time by about 8–45× and reduces per-query communication by about 5–67×.

6.1. Implementation and Evaluation Setup

We implement MULTI-LEVEL PIR in C++. The implementation consists of approximately 400 lines of code. Our code base is open-sourced[1].

We set $Q = \sqrt{n}$, the statistical security parameter to $\kappa = 80$, and the computational security parameter to $\lambda = 128$. Our implementation uses AES from Crypto++ for PRF evaluations. All schemes run with a single thread. All experiments run on an Alibaba Cloud ECS instance (ecs.u1-c1m8.8xlarge) with 256 GB of RAM and Ubuntu 24.04 (64-bit). The software environment uses GCC 13.3.0 and Go 1.23.6. We evaluate databases with 2^{28} entries, using entry sizes of 16 bytes and 256 bytes. We also report end-to-end performance on a 128 GB database with 2^{32} 32-byte entries.

1. https://anonymous.4open.science/r/Multi-level-PIR-3B6E

6.2. Experiment Results

Preprocessing time. Figure 9(a) compares preprocessing time between MULTI-LEVEL PIR and Balanced PIR. MULTI-LEVEL PIR avoids the expensive small-domain PRP evaluations used by Balanced PIR, so its preprocessing cost is much lower across the evaluated database sizes.

For a 4 GB database, Balanced PIR requires 6333.01 s of preprocessing. In comparison, MULTI-LEVEL PIR completes preprocessing in 140.84 s. Thus, MULTI-LEVEL PIR reduces preprocessing time by about 45×.

Client storage. Figure 9(b) reports client storage. Balanced PIR is the closest baseline because it targets the same low-storage regime implied by the $S \cdot T = \Theta(n)$ tradeoff. MULTI-LEVEL PIR matches this low-storage target and uses slightly less client storage than Balanced PIR.

For a 4 GB database, Balanced PIR requires 5.4 MB of client storage. MULTI-LEVEL PIR requires 5.01 MB, giving a modest reduction while preserving the same asymptotic storage-computation tradeoff.

TABLE 4: Comparison between single-server stateful PIR schemes: MULTI-LEVEL PIR achieves strong overall performance.

Scheme	Database Parameters	Client Storage (MB)	Offline		Online	
			Comm. (GB)	Computation (s)	Comm. (KB)	Computation (ms)
Piano	2^{28} 256-byte entries 64 GB database	837.75	64	2598	64.25	6.1
S3PIR		664.25	64	2310	34.50	2.8
Balanced PIR		70.22	64	6532	4128	6.4
MULTI-LEVEL PIR		68.29	64	790	61.62	7.4

Online bandwidth. Figure 9(c) compares per-query online bandwidth. Balanced PIR has high communication cost because its query protocol downloads up to $O(\sqrt{n})$ entries. This cost is the main practical drawback of using Balanced PIR to realize the $S \cdot T = \Theta(n)$ tradeoff. In contrast, MULTI-LEVEL PIR keeps the online download small while maintaining low client storage. Across our evaluated settings, MULTI-LEVEL PIR substantially reduces online communication by about 5–67$\times$ compared with Balanced PIR.

For a 4 GB database with 2^{28} entries, Balanced PIR requires 288 KB per query, while MULTI-LEVEL PIR requires 61.5 KB per query.

Online end-to-end delay. Figure 9(d) reports client-perceived online latency in a LAN setting. The latency comparison shows that our schemes keep the online phase practical while avoiding the preprocessing and communication overheads of Balanced PIR. For a 4 GB database with 2^{28} entries, our schemes and Balanced PIR require 1.5–3.5 ms per query in this environment.

Comparison with other PIR schemes. In Table 4, we present the practical performance of MULTI-LEVEL PIR and other three baselines in a 64 GB database. From the table, we see that MULTI-LEVEL PIR reduces client storage by approximately $10\times$, in comparison with Piano and S3PIR, while preserving the similar efficiency in online communication bandwidth and computation time.

Overall performance on 128 GB databases. We also evaluate MULTI-LEVEL PIR on 128 GB databases to measure scaling at a deployment-relevant size. For a database with 2^{32} 32-byte entries, MULTI-LEVEL PIR requires only 36.91 MB of client storage. The corresponding online end-to-end delay in the LAN setting is 8.76 ms. The online communication per query is only 246.18 KB.

7. Related Work

Client-specific preprocessing PIR. Stateful PIR [20] and the client-specific preprocessing model of Corrigan-Gibbs and Kogan [6] reduce online query cost by letting the client store hints prepared in an offline phase. TreePIR [14] and Piano [9] showed that this model can be practical in the two-server and single-server settings. S3PIR [10], Pirex [21], Plinko [15], and QuarterPIR [11] further improve concrete performance, support updates, or reduce online communication. LWE-based schemes [8, 17, 18] achieve near-optimal asymptotic tradeoffs with very small online communication, while Ghoshal et al. [22] obtain polylogarithmic online communication from one-way functions. ThorPIR [23] and Onepass PIR [24] use homomorphic techniques to reduce preprocessing cost, which is largely orthogonal to our focus on the client-storage and online-communication tradeoff. Other works study preprocessing PIR with many servers or information-theoretic security [25, 26, 27].

Our work stays within the practical client-specific preprocessing-PIR line, but focuses specifically on removing the κ-factor client-storage overhead while preserving the simple random-set structure.

Homomorphic Encryption-Based PIR. Homomorphic single-server PIR schemes such as Spiral [28], SimplePIR [29], Respire [30], HintlessPIR [31], and YPIR [32] achieve strong concrete performance using encrypted linear queries. They optimize a different regime: the server performs essentially linear work per query or relies on database-dependent preprocessing, without the linear preprocessing communication cost like in those practical single-server preprocessing PIR.

Doubly Efficient PIR. Doubly efficient PIR (DEPIR) [7, 33, 34, 35, 36, 37, 38] achieves sublinear queries through a global preprocessed and encoded server data structure rather than client-specific preprocessing, and therefore targets a different deployment tradeoff.

8. Conclusion

We revisited client-specific preprocessing PIR through the lens of the tight storage-computation tradeoff and concrete deployability. MULTI-LEVEL PIR shows that a simple multi-level random-set construction can remove the κ-factor client-storage overhead while preserving expected $O(\sqrt{n})$ online computation. MULTI-LEVEL++ PIR further shows that the same optimal-tradeoff regime is compatible with $O(n^{1/4})$ online communication. Our implementation of MULTI-LEVEL PIR suggests that this approach gives a well-rounded practical point: it reaches the low-storage regime of Balanced PIR while retaining the simplicity and low online bandwidth of random-set PIR.

Future work. The current construction exposes the database to the client without additional protection. To achieve this stronger "symmetric privacy" notion, oblivious PRF-based approach may be adopted. Another direction is to provide verifiability or stronger malicious-server protection (similar to verifiable Balanced PIR), while preserving the low storage, low bandwidth, and simple implementation profile of MULTI-LEVEL PIR.

Ethics considerations

This work studies cryptographic protocols and does not involve human-subject experiments, user data, or deployment on real users.

Generative AI Usage Considerations

Generative AI was used for editorial purposes in this manuscript, and all outputs were inspected by the authors to ensure accuracy and originality.

References

[1] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan, "Private information retrieval," in *FOCS*, 1995.

[2] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan, "Private information retrieval," *J. ACM*, vol. 45, no. 6, p. 965–981, Nov. 1998.

[3] A. Vadapalli, F. Bayatbabolghani, and R. Henry, "You may also like... privacy: Recommendation systems meet pir," *Proceedings on Privacy Enhancing Technologies*, 2021.

[4] D. Kogan and H. Corrigan-Gibbs, "Private blocklist lookups with checklist," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 875–892.

[5] C. Bee, Y.-J. Chen, M. Queen, D. Ward, X. Liu, L. Organick, G. Seelig, K. Strauss, and L. Ceze, "Molecular-level similarity search brings computing to dna data storage," *Nature communications*, vol. 12, no. 1, p. 4764, 2021.

[6] H. Corrigan-Gibbs and D. Kogan, "Private information retrieval with sublinear online time," in *EUROCRYPT*, 2020.

[7] A. Beimel, Y. Ishai, and T. Malkin, "Reducing the servers computation in private information retrieval: Pir with preprocessing," in *CRYPTO*, 2000, pp. 55–73.

[8] H. Corrigan-Gibbs, A. Henzinger, and D. Kogan, "Single-server private information retrieval with sublinear amortized time," in *Eurocrypt*, 2022.

[9] M. Zhou, A. Park, E. Shi, and W. Zheng, "Piano: Extremely simple, single-server pir with sublinear server computation," in *IEEE S& P*, 2024.

[10] L. Ren, M. H. Mughees, and I. Sun, "Simple and practical amortized sublinear private information retrieval using dummy subsets," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024.

[11] A. Ghoshal, M. Zhou, and E. Shi, "Efficient pre-processing pir without public-key cryptography," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2024, pp. 210–240.

[12] P. S. Arunachalaramanan and L. Ren, "Single-server stateful pir with verifiability and balanced efficiency," in *IEEE S&P*, 2026.

[13] K. Yeo, "Lower bounds for (batch) pir with private preprocessing," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2023, pp. 518–550.

[14] A. Lazzaretti and C. Papamanthou, "Treepir: Sublinear-time and polylog-bandwidth private information retrieval from ddh," in *CRYPTO*, 2023.

[15] A. Hoover, S. Patel, G. Persiano, and K. Yeo, "Plinko: Single-server pir with efficient updates via invertible prfs," *Eurocrypt*, 2024.

[16] Z. Wang and L. Ren, "Single-server client preprocessing pir with tight space-time trade-off," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2025, pp. 94–122.

[17] M. Zhou, W.-K. Lin, Y. Tselekounis, and E. Shi, "Optimal single-server private information retrieval," in *EUROCRYPT*, 2023.

[18] A. Lazzaretti and C. Papamanthou, "Near-optimal private information retrieval with preprocessing," in *Theory of Cryptography Conference*. Springer, 2023, pp. 406–435.

[19] E. Shi, W. Aqeel, B. Chandrasekaran, and B. Maggs, "Puncturable pseudorandom sets and private information retrieval with near-optimal online bandwidth and time," in *CRYPTO*, 2021.

[20] S. Patel, G. Persiano, and K. Yeo, "Private stateful information retrieval," in *CCS*, 2018.

[21] H.-D. Nguyen, J. Guajardo, and T. Hoang, "Client-efficient online-offline private information retrieval," Cryptology ePrint Archive, Paper 2024/719, 2024. [Online]. Available: https://eprint.iacr.org/2024/719

[22] A. Ghoshal, M. Zhou, E. Shi, and B. Peng, "Pseudorandom functions with weak programming privacy and applications to private information retrieval," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2025, pp. 284–313.

[23] B. Fisch, A. Lazzaretti, Z. Liu, and C. Papamanthou, "Thorpir: single server pir via homomorphic thorp shuffles," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 1448–1462.

[24] A. Lazzaretti and C. Papamanthou, "Single pass {Client-Preprocessing} private information retrieval," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 5967–5984.

[25] Y. Ishai, E. Shi, and D. Wichs, "Pir with client-side preprocessing: information-theoretic constructions and lower bounds," in *Annual International Cryptology Conference*. Springer, 2024, pp. 148–182.

[26] J. Singh, Y. Wei, and V. Zikas, "Information-theoretic multi-server private information retrieval with client preprocessing," in *Theory of Cryptography Conference*. Springer, 2025, pp. 423–450.

[27] A. Ghoshal, M. Zhou, B. Peng, and E. Shi, "Zelda: Efficient multi-server preprocessing pir with unconditional security," in *IEEE S&P*, 2025.

[28] S. J. Menon and D. J. Wu, "SPIRAL: Fast, high-rate single-server PIR via FHE composition," in *IEEE S&P*, 2022.

[29] A. Henzinger, M. M. Hong, H. Corrigan-Gibbs, S. Meiklejohn, and V. 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. [Online]. Available: https://eprint.iacr.org/2022/949

[30] A. Burton, S. J. Menon, and D. J. Wu, "RESPIRE: High-rate PIR for databases with small records," in *ACM CCS*, 2024.

[31] B. Li, D. Micciancio, M. Raykova, and M. Schultz-Wu, "Hintless single-server private information retrieval," in *Annual International Cryptology Conference*. Springer, 2024, pp. 183–217.

[32] S. J. Menon and D. J. Wu, "Ypir: High-throughput single-server pir with silent preprocessing," *Cryptology ePrint Archive*, 2024.

[33] R. Canetti, J. Holmgren, and S. Richelson, "Towards doubly efficient private information retrieval," in *TCC*, 2017.

[34] E. Boyle, Y. Ishai, R. Pass, and M. Wootters, "Can we access a database both locally and privately?" in *TCC*, 2017.

[35] W.-K. Lin, E. Mook, and D. Wichs, "Doubly efficient private information retrieval and fully homomorphic ram computation from ring lwe," in *STOC*, 2023.

[36] A. Ghoshal, B. Li, Y. Ma, C. Dai, and E. Shi, "Scalable multi-server private information retrieval," in *Theory of Cryptography Conference*. Springer, 2025, pp. 582–610.

[37] A. Lazzaretti, Z. Liu, B. Fisch, P. Miao, and C. Papamanthou, "Multi-server doubly efficient pir in the classical model and beyond," in *Theory of Cryptography Conference*. Springer, 2025, pp. 611–641.

[38] A. Henzinger and S. Ragavan, "Two-server private information retrieval in sublinear time and quasilinear space," *Cryptology ePrint Archive*, 2025.

Appendix A.
Intuition Behind the Preprocessing PIR Lower Bound

This appendix recalls the storage-time lower bound that motivates our target parameters. Consider a single-server preprocessing PIR scheme for an n-bit database in the cell-probe model, where the database is stored without encoding or additional server-side data. During preprocessing, the client stores an S-bit client-specific hint; during each online query, the protocol probes at most T database bits. Corrigan-Gibbs and Kogan [6] show that any such scheme must satisfy $ST = \Omega(n)$ in the non-trivial regime where $S, T = o(n)$. Intuitively, this rules out simultaneously sub-$\sqrt{n}$ client state and sub-$\sqrt{n}$ online query work in this model.

We provide an informal overview of the intuition behind this lower bound. Fix any preprocessing PIR scheme. For any index i in the database, since the server probes only T bits per query and the access pattern must be roughly "uniform" across all queries (to preserve privacy), the average probability that the i-th bit is probed during a query is only $T/n = o(1)$. In some sense, each query transcript yields $T + 1$ bits of information about the database – we probe T bits and also decode the i-th bit without probing it.

Now view this as a compression procedure for a uniform $\mathsf{DB} \in \{0,1\}^n$: repeatedly query the next unrecovered index and record only the values of newly probed bits. Each query reveals at most T new probed bits *plus* the target bit $\mathsf{DB}[i]$ "for free," so one query can recover at most $T+1$ previously unknown database bits, implying at least $n/(T+1)$ queries to recover all n bits. Thus the encoding length (the hint plus all probed bits throughout the process) satisfies $\mathsf{len} \le S + (n - n/(T+1))$, but Shannon's source coding theorem states that any lossless encoding of a uniform n-bit string has expected length at least n, forcing $n \le \mathsf{len} \le S + (n - n/(T+1))$. This implies $S(T+1) \ge n$, and thus $ST = \Omega(n)$.

Appendix B.
Detailed Proofs of MULTI-LEVEL PIR

In this section, we provide necessary lemmas and proofs for the framework given in Section 4.

B.1. Hint structure

Throughout this subsection, let $C = n/(2B)$ be the expected selected-set size; equivalently, the database is divided into $2C$ chunks of size B.

Lemma 7 (Primary hint editing)**.** *For a uniformly random primary hint $H = (s, c, \Delta, p)$ conditioned on containing x, let $Q = (s', \perp, \Delta') \leftarrow \mathsf{Edit}(H, x)$. Then:*

1) *(s', Δ') is uniformly distributed over $\{0,1\}^{2C} \times [B]^{2C}$, independent of x.*
2) *$\mathsf{Set}_c(Q) = \mathsf{Set}(H) \setminus \{x\}$.*

Proof. Conditioned on containing x, the coordinates $(s[j'], \Delta[j'])$ for $j' \neq j$ remain uniform and independent, and $(s[j], \Delta[j]) = (c, k)$. The transformation sets $s'[j] = 1 - s[j]$ and samples $\Delta'[j] \leftarrow [B]$ uniformly, leaving all other coordinates unchanged. Since c is uniform in $\{0,1\}$, the bit $s'[j] = 1 - c$ is uniform, and it is independent of $\Delta'[j]$ and of the remaining coordinates. Therefore every coordinate of (s', Δ') is uniform and the joint distribution is uniform over $\{0,1\}^{2C} \times [B]^{2C}$.

For the set identity, fix any chunk $j' \neq j$. The edit operation does not modify $(s[j'], \Delta[j'])$, hence j' contributes the same element $j'B + \Delta[j']$ to $\mathsf{Set}(H)$ and to $\mathsf{Set}_c(Q)$ iff $s[j'] = c$. For the distinguished chunk j, we have $s[j] = c$ (since $x \in \mathsf{Set}(H)$) and thus $jB + \Delta[j] = x$ is the unique element contributed by chunk j to $\mathsf{Set}(H)$. After editing, $s'[j] = 1 - s[j] \neq c$, so chunk j contributes no element to $\mathsf{Set}_c(Q)$. It follows that $\mathsf{Set}_c(Q)$ contains exactly the elements of $\mathsf{Set}(H)$ coming from chunks $j' \neq j$, i.e., $\mathsf{Set}_c(Q) = \mathsf{Set}(H) \setminus \{x\}$. □

Lemma 8 (Backup-to-primary transformations)**.** *Fix an index $x = jB + k$. Let $\bar{H}$ be a uniformly random backup hint and let H be a fresh primary hint. Let H^{in} (resp., H^{out}) be the output of the backup-to-primary transformation from Figure 3 conditioning on containing x (resp., not containing x). Then*

$$H^{\mathsf{in}} \overset{d}{=} H \mid (x \in \mathsf{Set}(H)) \text{ and } H^{\mathsf{out}} \overset{d}{=} H \mid (x \notin \mathsf{Set}(H)).$$

Proof. We analyze the two transformations separately.

Conditioning on containing x. Write $\bar{H} = (s^{\mathrm{old}}, \perp, \Delta^{\mathrm{old}}, q_0, q_1)$. The procedure defines (s, Δ) by setting $(s[j], \Delta[j]) \leftarrow (1 - s^{\mathrm{old}}[j], k)$ and $(s[j], \Delta[j]) \leftarrow (s^{\mathrm{old}}[j], \Delta^{\mathrm{old}}[j])$ for every $j \neq j$. It then sets $c \leftarrow s[j]$ and $p \leftarrow q_c \oplus \mathsf{DB}[x]$. Denote the resulting expanded primary hint by $H^{\mathsf{in}} = (s, c, \Delta, p)$.

Conditioned on containing x, a fresh primary hint satisfies the constraints $\Delta[j] = k$ and $s[j] = c$, while all other coordinates are uniform and independent. In the transformation, the overwrite forces $\Delta[j] = k$, and since flipping a fixed bit is a bijection on $\{0,1\}^{2C}$, the resulting string s is still uniform. Finally $c = s[j]$ is therefore uniform in $\{0,1\}$, and the other coordinates remain uniform and independent, matching the conditional distribution.

Conditioning on not containing x. The transformation never changes s and only possibly overwrites $\Delta[j]$; all other coordinates remain uniform and independent in both distributions. Thus it suffices to match the joint distribution of $(s[j], \Delta[j], c)$. Denote the resulting expanded primary hint by $H^{\mathsf{out}} = (s, c, \Delta, p)$, where $p = q_c$.

For a fresh primary hint, $(s[j], \Delta[j], c)$ is uniform over $\{0,1\} \times [B] \times \{0,1\}$, conditioned on the event $x \notin \mathsf{Set}(H)$, i.e., conditioned on not having simultaneously $\Delta[j] = k$ and $s[j] = c$. Equivalently, each admissible triple has probability $1/(4B - 2) = 1/(2(2B - 1))$.

Now consider the transformation on a uniform backup hint, where initially $s[j]$ is uniform in $\{0,1\}$ and $\Delta[j]$ is uniform in $[B]$.

- If initially $\Delta[j] = k$, it sets $c \leftarrow 1 - s[j]$, and keeps $\Delta[j] = k$ with probability $B/(2B - 1)$; otherwise it overwrites $\Delta[j]$ to a uniform value in $[B] \setminus \{k\}$.

- If initially $\Delta[j] = \delta \neq k$, it keeps $\Delta[j] = \delta$ and sets $c \leftarrow s[j]$ with probability $B/(2B-1)$ and $c \leftarrow 1 - s[j]$ otherwise.

It follows that every admissible triple has the desired probability:

1) For $(a, k, 1-a)$: $\Pr[(s[j], \Delta[j], c) = (a, k, 1-a)] = \frac{1}{2} \cdot \frac{1}{B} \cdot \frac{B}{2B-1} = \frac{1}{2(2B-1)}$.
2) For (a, δ, a) with $\delta \in [B] \setminus \{k\}$: $\Pr[(s[j], \Delta[j], c) = (a, \delta, a)] = \frac{1}{2} \cdot \frac{1}{B} \cdot \frac{B}{2B-1} = \frac{1}{2(2B-1)}$.
3) For $(a, \delta, 1-a)$ with $\delta \in [B] \setminus \{k\}$, there are two disjoint ways to obtain it: either initially $\Delta[j] = \delta$ and we set $c \leftarrow 1 - a$, or initially $\Delta[j] = k$ and we overwrite to δ. Summing the two probabilities gives

$$\Pr\big[(s[j], \Delta[j], c) = (a, \delta, 1-a)\big] = \tfrac{1}{2(2B-1)}.$$

Hence the output has exactly the same conditional distribution as a fresh primary hint conditioned on $x \notin \mathsf{Set}(H)$. □

B.2. PIR scheme with failure privacy

Lemma 9 (Primary hint table distribution is preserved after refresh)**.** *Fix a query index x. Let $\mathcal{T}$ be the primary hint table at the start of the query, consisting of M independently sampled primary hints (as in the preprocessing phase). Let* Succ *denote the event that the query succeeds, i.e., that $\mathcal{T}$ contains at least one hint containing x. Let $\mathcal{T}'$ be the primary hint table after executing the online step (consuming one uniformly random matching hint when* Succ *occurs) followed by the refresh procedure above. Then*

$$\mathcal{T}' \mid (\mathsf{Succ} = 0) \overset{d}{=} \mathcal{T}' \mid (\mathsf{Succ} = 1) \overset{d}{=} \mathcal{T},$$

where $\overset{d}{=}$ denotes equality in distribution; equivalently, the distribution of the refreshed primary hint table is identical to the original distribution, even when conditioned on whether the query succeeded or failed.

Proof. Let H be a fresh primary hint (distributed as in preprocessing). For the fixed index x, define the events $\mathsf{in}(H)$ and $\mathsf{out}(H)$ to be that H contains x or does not contain x, respectively, and let

$$\mathcal{D}_{\mathsf{in}} \coloneqq (H \mid \mathsf{in}(H)) \quad \text{and} \quad \mathcal{D}_{\mathsf{out}} \coloneqq (H \mid \mathsf{out}(H)).$$

By construction, for a fresh table $\mathcal{T} = (H_1, \ldots, H_M)$ the indicators $\mathbf{1}[\mathsf{in}(H_i)]$ are i.i.d. $\mathrm{Bernoulli}(1/(2B))$, and conditioned on these indicators the hints are independent with $H_i \sim \mathcal{D}_{\mathsf{in}}$ or $H_i \sim \mathcal{D}_{\mathsf{out}}$ accordingly. Let Z be the number of hints in $\mathcal{T}$ that contain x, so $Z \sim \mathrm{Binom}(M, 1/(2B))$ and let $p_0 \coloneqq \Pr[Z = 0] = (1 - 1/(2B))^M$.

Case 1: $\mathsf{Succ} = 0$ (equivalently $Z = 0$). Then every hint in $\mathcal{T}$ is distributed as $\mathcal{D}_{\mathsf{out}}$, independently. The refresh procedure samples $K \leftarrow \mathrm{Binom}(M, 1/(2B))$, chooses a uniformly random subset $S \subseteq [M]$ of size K, and replaces the hints in positions S by fresh independent samples from $\mathcal{D}_{\mathsf{in}}$ (using the backup-to-primary transformation), leaving the remaining positions unchanged. This is exactly an alternative way to sample a fresh table from the unconditional distribution: sample $K \sim \mathrm{Binom}(M, 1/(2B))$, choose S uniformly of size K, sample $H_i \sim \mathcal{D}_{\mathsf{in}}$ for $i \in S$ and $H_i \sim \mathcal{D}_{\mathsf{out}}$ for $i \notin S$. Therefore $\mathcal{T}' \mid (\mathsf{Succ} = 0) \overset{d}{=} \mathcal{T}$.

Case 2: $\mathsf{Succ} = 1$ (equivalently $Z > 0$). The online step consumes one uniformly random hint among those containing x, and the refresh immediately replenishes it with a fresh independent sample from $\mathcal{D}_{\mathsf{in}}$. Since, conditioned on $(Z > 0)$, the collection of "in" hints is i.i.d. $\mathcal{D}_{\mathsf{in}}$, replacing a uniformly random one by a fresh $\mathcal{D}_{\mathsf{in}}$ sample leaves conditional distribution of the table unchanged; thus after this replenishment the table is still distributed as $(\mathcal{T} \mid \mathsf{Succ} = 1)$.

Next, the refresh samples an independent coin $I \sim \mathrm{Bernoulli}(p_0)$. If $I = 1$, it replaces every hint that now contains x—including the freshly replenished slot—by a fresh independent sample from $\mathcal{D}_{\mathsf{out}}$, yielding a table distributed as $(\mathcal{T} \mid \mathsf{Succ} = 0)$; if $I = 0$, it leaves the table as-is, yielding a table distributed as $(\mathcal{T} \mid \mathsf{Succ} = 1)$. Thus, conditioned on $(\mathsf{Succ} = 1)$, the refreshed table equals $(\mathcal{T} \mid \mathsf{Succ} = 0)$ in distribution with probability p_0 and equals $(\mathcal{T} \mid \mathsf{Succ} = 1)$ in distribution with probability $1 - p_0$ (with an independent choice between these two branches). Since $p_0 = \Pr[\mathsf{Succ} = 0]$, this is exactly the law of total probability for $\mathcal{T}$, and hence equals the unconditional distribution of $\mathcal{T}$. Thus $\mathcal{T}' \mid (\mathsf{Succ} = 1) \overset{d}{=} \mathcal{T}$, completing the proof. □

Lemma 10 (Backup-hint usage)**.** *Let C be the number of backup hints consumed in one query, and let $p_0 = (1 - 1/(2B))^M$. Let $Z \sim \mathrm{Binom}(M, 1/(2B))$ be the number of primary hints containing x at the start of the query, let $K \sim \mathrm{Binom}(M, 1/(2B))$ be independent, and let $I \sim \mathrm{Bernoulli}(p_0)$ be independent. Then*

$$C \leq_{\mathrm{st}} = 1 + I \cdot Z + K \cdot \mathbf{1}[Z = 0].$$

Moreover, let $C_1, \ldots, C_B$ be the number of backup hints consumed in the first B queries (with a refresh after each query) and let $C_{\mathrm{tot}} \coloneqq \sum_{t=1}^{B} C_t$. Then

$$\Pr[C_{\mathrm{tot}} \geq 3B] \leq (2/3)^B$$

In particular, if $B = \Omega(\kappa)$, then $\Pr[C_{\mathrm{tot}} \geq 3B] = \mathsf{negl}(\kappa)$, i.e., $3B$ backup hints are sufficient for B queries with overwhelming probability.

Proof. Let Z be the number of primary hints containing x at the start of the query. If $Z = 0$, the query fails and the refresh step samples $K \sim \mathrm{Binom}(M, 1/(2B))$ new hints containing x, consuming exactly K backup hints. If $Z \geq 1$, the client consumes one hint containing x and replenishes it using one backup hint. With probability $p_0 = (1 - 1/(2B))^M$, it additionally replaces all Z hints currently containing x by fresh hints conditioned on not containing x, consuming Z backup hints. This yields the stated stochastic domination.

For any $\lambda > 0$, let $\widetilde{C} = 1 + I \cdot Z + K \cdot \mathbf{1}[Z = 0]$, then

$$\mathbb{E}[e^{\lambda C_{\mathrm{tot}}}] = \prod_{t=1}^{B} \mathbb{E}[e^{\lambda C_t}] \leq \left(\mathbb{E}[e^{\lambda \widetilde{C}}]\right)^B,$$

Query Protocol

Server. The same as that in Figure 7.

Client. Upon query x, where $x = jB + h$ and $j = j_1 D + j_2$, Client first searches for a matched hint. Then,

- Client sets $\boldsymbol{k}, \boldsymbol{\Delta}, \boldsymbol{s}$ the same way as in Figure 7.
- Additionally for a *new* primary hint $\tilde{H} = (S, S^*, s, s^*, \Delta, \mathsf{mark}, \boldsymbol{p})$, where $\mathsf{mark} = \tilde{j}B + \tilde{h}$ and $\tilde{j} = \tilde{j_1}D + \tilde{j_2}$.
  - If $j_1 = \tilde{j_1}$, Client sets $\tilde{s} \leftarrow 1 - s[j]$ and overwrites $\Delta[j_1 D + \tilde{j_2}] \leftarrow \tilde{h}$.
  - If $j_1 \neq \tilde{j_1}$, Client first rejection-samples a subkey conditioned on containing mark and global negative constraints. Then, Client sets $\tilde{s} \leftarrow 1 - s[j]$ and overwrites the reject-sampled subkey to $k_{\tilde{j_1}}$.
- Client sends $(\boldsymbol{k}, \boldsymbol{\Delta}, \boldsymbol{s})$ to the server.

Client local computation. Upon receving a reply $(\boldsymbol{P}, \boldsymbol{Q})$ from the server,

- For an *original* hint H, Client computes $\beta = \left(\bigoplus_{i \in [C]} P_{i,s^*}\right) \oplus P_{j_1,s^*} \oplus Q_{j_1,s^*} \oplus p$.
- For a *new* primary hint $\tilde{H}$,
  - If $j_1 = \tilde{j_1}$, Client computes $\beta = \left(\bigoplus_{i \in [C], S[i] = 1 - S^*} P_{i,s^*}\right) \oplus P_{j_1,s^*} \oplus Q_{j_1,s^*} \oplus p_{1-S^*,s^*}$.
  - If $j_1 \neq \tilde{j_1}$, Client computes $\beta = \left(\bigoplus_{i \in [C], S[i] = S^*} P_{i,s[j]}\right) \oplus P_{j_1,s[j]} \oplus Q_{j_1,s[j]} \oplus p_{S^*,s[j]}$.

Figure 10: Detailed query protocol.

and by Markov's inequality,

$$\Pr[C_{\mathrm{tot}} \geq 3B] = \Pr[e^{\lambda C_{\mathrm{tot}}} \geq e^{3\lambda B}] \leq \left(\frac{\mathbb{E}[e^{\lambda \tilde{C}}]}{e^{3\lambda}}\right)^B.$$

Choose $\lambda = \ln(3/2)$, and by some standard inequality manipulations, we can bound that

$$\Pr[C_{\mathrm{tot}} \geq 3B] \leq (2/3)^B.$$

□

Finally, we consider the complexity profile of this scheme. Letting $C = n/(2B)$ be the expected set size, it yields the following (which matches the description in Section 4.4):

- **Correctness**: The failure probability is $(1 - C/n)^M + \mathsf{negl}(\kappa) < e^{-MC/n}$ for any query at any time.
- **Storage**: $O(M)$. The client stores M parity values plus compact PRF-based set representations.
- **Computation**: $O(M + C)$ PRF evaluations per query. $O(M)$ for hint lookup and refresh, $O(C)$ for set expansion and parity computation, and amortized $O(C)$ for preprocessing.
- **Communication**: $O(C)$ per query. The client transmits a single set description per query, and the preprocessing phase amortizes to $n/B = O(C)$ parities per query.

Appendix C. Proof of Theorem 3

Proof. We argue by induction over the adaptive query sequence, conditioned on the event that the backup pool is not exhausted. The invariant is that, before each query, the primary table consists of M independent fresh primary hints, and all unused backup hints are fresh and independent of the table. This holds after preprocessing by construction.

Consider a query $x = jB + k$ under this invariant. If Probe succeeds, it selects a primary hint H^* containing x. By Lemma 7, the edited descriptor sent to the server is uniform over descriptors of the same size and is independent of x, even after conditioning on success. The same lemma gives the parity identity used by the client: the server's response is the old selected-set parity with x removed. Xoring this value with the stored primary parity therefore yields $\mathsf{DB}[x]$. If Probe fails, no primary hint contains x, and the client obtains the correct value directly from the oracle $v = \mathcal{O}_{\mathsf{DB}}(x) = \mathsf{DB}[x]$.

It remains to verify that Repair restores the invariant. The two branches induce different conditional distributions: after failure, every primary hint is conditioned on excluding x; after success, one containing hint has been consumed and the remaining table is conditioned on the fact that at least one hint contained x. Lemma 8 provides the exact conditional samplers required in these cases: PromoteIn samples a fresh primary hint conditioned on containing x, while PromoteOut samples one conditioned on excluding x. Lemma 9 lifts these samplers to the whole table and shows that repair returns the table to M independent fresh primary hints in both branches, even conditioned on the branch observed by the server. Lemma 10 ensures that the necessary independent backup hints are available except with negligible probability. Hence the invariant is restored after each completed query.

The transcript simulation follows from the invariant and Lemma 7. For each query, the simulator samples failure with probability $p_{\mathsf{fail}} = (1 - 1/(2B))^M$, the probability that none of the M fresh primary hints contains the queried index. This probability is independent of the adaptive query value. On failure, the simulator outputs only the failure event; on success, it outputs a success event together with a fresh random descriptor of the edited-query format. These outputs match the real server view: the success probability is as above, and the success descriptor is query-independent and uniform by Lemma 7. Values used only for client repair,

Backup hint sampling and backup promotion

SampleBackup(B, R).

1) Sample a PRF key sk, expand it into R superblock color bits and subkeys: $(S[i], k_i) \leftarrow \mathsf{PRF}_1(\mathsf{sk}, i)$ for every $i \in [R]$. Then, expand the subkeys into $(s[j], \Delta[j])$ for every chunk.
2) For every $a, b \in \{0,1\}$, define

$$\mathsf{Set}_{ab}(\hat{H}) = \left\{ jB + \Delta[j] \;\middle|\; \begin{array}{l} S[j_1] = a,\ s[j] = b, \\ j = j_1 D + j_2 \in [m] \end{array} \right\},$$

and compute $q_{ab} \leftarrow \bigoplus_{x \in \mathsf{Set}_{ab}(\hat{H})} \mathsf{DB}[x]$.
3) Store the backup hint $\hat{H} = (S, \mathsf{sk}, \boldsymbol{q})$, where $\boldsymbol{q} = (q_{00}, q_{01}, q_{10}, q_{11})$.

PromoteIn$(\hat{H}, x)$ **for** $x = jB + k$**, where** $j = j_1 D + j_2$**.**

1) Set $S^* \leftarrow 1 - S[j_1]$, $s^* \leftarrow 1 - s[j]$, and $\mathsf{mark} \leftarrow x$.
2) Overwrite $s[j] \leftarrow 1 - s[j]$.
3) Run FinalizePromotion$(\hat{H}, x, S^*, s^*, \mathsf{mark})$.

PromoteOut$(\hat{H}, x)$ **for** $x = jB + k$**.**

1) If $\Delta[j] = k$, set $S^* \leftarrow 1 - S[j_1]$ and $s^* \leftarrow 1 - s[j]$. With probability $(B-1)/(2B-1)$, set $\mathsf{mark} \leftarrow \{jB + h : h \in [B] \setminus \{k\}\}$; otherwise set $\mathsf{mark} \leftarrow x$.
2) If $\Delta[j] \neq k$, set $S^* \leftarrow S[j_1]$. Set $s^* \leftarrow s[j]$ with probability $B/(2B-1)$, and set $s^* \leftarrow 1 - s[j]$ otherwise. Set $\mathsf{mark} \leftarrow jB + \Delta[j]$.
3) Run FinalizePromotion$(\hat{H}, x, S^*, s^*, \mathsf{mark})$.

FinalizePromotion$(\hat{H}, x, S^*, s^*, \mathsf{mark})$**.**

1) For every $a, b \in \{0,1\}$, set

$$p_{ab} \leftarrow \begin{cases} q_{ab} \oplus \mathsf{DB}[x], & \text{if } (a,b) = (1-S^*, s^*), \\ \perp, & \text{if } (a,b) = (1-S^*, 1-s^*), \\ q_{ab}, & \text{otherwise.} \end{cases}$$

2) Output $\tilde{H} = (S, S^*, s, s^*, \Delta, \mathsf{mark}, \boldsymbol{p})$, where $\boldsymbol{p} = (p_{00}, p_{01}, p_{10}, p_{11})$.

Figure 11: Backup hints sampling and conditional promotion algorithm.

including the oracle value on failure, are never sent to the server. Since repair restores a fresh state in either branch, the same simulation applies to every subsequent adaptive query. Induction gives correctness and transcript simulation for the entire sequence. □

Appendix D. Detailed Algorithms of MULTI-LEVEL++ PIR

In this section, we provide details of our construction given in Section 5 and necessary lemmas.

Preprocessing protocol.

Refresh and rejection sampling

Initialization.

- Initialize a list Hist to empty.
- For each original primary hint H, initialize $\mathsf{time}(H) \leftarrow 0$.

Upon query x_t, refresh and record the modification history.

- The refresh algorithm is the same as that in MULTI-LEVEL PIR, except that when a matched hint is found and the elimination of all hints that contain x_t does not occur, additionally replace all hints containing x_t into fresh *new* hints conditioned on containing x_t. Whenever a replacement on a hint $\tilde{H}$ happens, additionally record $\mathsf{time}(\tilde{H}) \leftarrow t$.
- Append x_t to Hist.

Rejection sampling for $\tilde{H}$ on superblock j_1.

- Sample a subkey of superblock j_1 that satisfies the positive constraint $+\mathsf{mark}$.
- Visit $\mathsf{Hist}[i]$ for $i > \mathsf{time}(\tilde{H})$. If the modification $\mathsf{Hist}[i] = x$ happens in superblock j_1 and x is in the expanded set, then reject.
- Accept the subkey.

Figure 12: Single level: refresh and rejection sampling.

Primary hints have two types, *original* ones and *new* ones. *Original* primary hints and backup hints are sampled during preprocessing. A *new* primary hint is generated from a backup hint through backup promotion to replace a consumed primary hint. In Figure 6, we have presented how original primary hints are sampled in the preprocessing phase. A new primary hint $\tilde{H}$ has structure $(S, S^*, s, s^*, \Delta, \mathsf{mark}, \boldsymbol{p})$. Suppose mark is in the r-th superblock, we then define

$$\mathsf{Set}(\tilde{H}) = \left\{ jB + \Delta[j] \;\middle|\; \begin{array}{l} \text{either } j_1 \neq r, S[j_1] = S^* \\ \text{or } (j_1 = r, s[j] = s^*) \end{array} \right\},$$

where $j = j_1 D + j_2$. We will describe it in detail later.

When a primary hint is consumed or due to refresh requirement, the client promotes a backup hint to a *new* primary one. In Figure 11, we will give the details of backup hint sampling and promotion algorithm, which ensures that *every backup hint can be editted in any position.*

Main query protocol and refresh. Figure 7 illustrates our query protocol based on PPPS. Sometimes a subkey should be rejection sampled according to some constraints. We add a list Hist to record the query history for reconstruction of right distribution. Bias from the original distribution can be observed by the server and leads to information leakage. Figure 12 presents the detailed algorithm.

Privacy analysis. The privacy comes from the following lemmas. Due to space limit, we will provide the proofs in the full version.

Lemma 11. *Before each query, for any fixed index x, the events that x is available in the primary hints are independent and identically distributed. Moreover, after each completed*

query and refreshment, the hint table restricted to the index arrays and color vectors has the same distribution as before the query; this distribution is uniform.

Lemma 12. *The rejection-sampling preserves the distribution of the indices.*

Lemma 13 (Privacy for MULTI-LEVEL++ PIR)**.** *Conditioned on no backup exhaustion, the real* MULTI-LEVEL++ PIR *server view is computationally indistinguishable from the simulator that, for each executed level and query, samples the public success/failure bit using the level's failure probability and, on success, outputs fresh random descriptors.*