Client-Efficient Online-Offline Private Information Retrieval

Hoang-Dung Nguyen
Virginia Tech
nhd@vt.edu

Jorge Guajardo
Robert Bosch LLC — RTC
jorge.guajardomerchan@us.bosch.com

Thang Hoang
Virginia Tech
thanghoang@vt.edu

Abstract

Private Information Retrieval (PIR) permits clients to query data entries from a public database hosted on untrusted servers while preserving client privacy. Traditional PIR models suffer from high computation and/or bandwidth overhead due to linear database processing for privacy. Recently, Online-Offline PIR (OO-PIR) has been proposed to improve PIR practicality by precomputing query-independent materials to accelerate online access. While state-of-the-art OO-PIR schemes (e.g., S&P'24, CRYPTO'23) successfully reduce online processing cost to sublinear levels, they still impose substantial bandwidth and storage burdens on the client, especially when operating on large databases.

In this paper, we propose Pirex, a new two-server OO-PIR with semi-honest security that offers minimal client inbound bandwidth and storage cost while retaining the sublinear processing efficiency. The Pirex design is simple with most operations are naturally low-cost and streamlined (e.g., XOR, PRF, modular arithmetic). We have fully implemented Pirex and evaluated its real-world performance using commodity hardware. Our results showed that Pirex outperforms existing OO-PIR schemes by at least two orders of magnitude. With a 1 TB database, Pirex takes 55ms to retrieve a 4 KB entry, compared with 9-30s by state-of-the-art. For practical databases with billions of 4 KB entries, Pirex only takes 16 KB of inbound bandwidth, which is up to three orders of magnitude more efficient.

Keywords

Private Information Retrieval; Distributed Computation

1 Introduction

Public databases provide the users with seamless access to diverse data resources, including entertainment (e.g., audio/video), social (e.g., news media), economic (e.g., stock market), healthcare (e.g., medical, pharmaceutical data), geospatial services (e.g., locations, directions). These databases eliminate the need for local storage and allow users to retrieve the latest information remotely. While these databases are not considered sensitive, privacy concerns arise as the users' queries on them can still, inadvertently reveal sensitive information, such as their personal preferences, current location, health conditions, or revenue streams [55, 56]. A database server can deliberately misuse the users' query behaviors (i.e. user locations [51], search frequency [69]) and expose them to malicious activities such as price and search discrimination [38, 57].

To preserve the user privacy, Chor et. al [26] proposed Private Information Retrieval (PIR), a cryptographic primitive that permits users to retrieve an entry in a public database without revealing to the adversarial server which entry has been accessed. Despite its strong privacy guarantee, PIR can be costly in terms of bandwidth and processing overhead. Various studies have managed to reduce the PIR bandwidth cost in single-server [17, 23, 25, 34, 42, 47, 49] and multi-server [16, 19–21, 32, 33, 35, 70] setting, but the processing cost remains a barrier to making PIR practical. Beimel et al. [21] proved that under the standard computation model, any secure PIR must incur at least linear processing cost (w.r.t the database size). Sion et al. [66] showed that, in certain conditions, streaming entire database is more efficient than such linear server processing.

To be more computationally efficient, PIR has been studied in different computation models such as preprocessing [21, 22, 24, 28, 29] or batching [17, 21, 40, 48, 54]. Patel et al. [58] proposed Online-Offline PIR (OO-PIR), a preprocessing paradigm where the client privately precomputes query-independent hints beforehand to accelerate online access. Corrigan-Gibbs and Kogan [29] designed the first OO-PIR scheme with $O(\sqrt{N})$ server computation per online query. Later works were proposed to improve the OO-PIR efficiency [41, 44, 72] or optimality [43, 65, 71] and achieved promising results.

Although OO-PIR achieves a sublinear server processing cost, it poses a serious bandwidth and storage burden to the client. Most OO-PIR schemes [28, 29, 41, 43, 44, 65, 71, 72] require the client to store a considerable amount of hints. Specifically, for a database with N entries of size B, the client storage is $\Omega(\lambda B\sqrt{N})$ (with λ is the security parameter). More critically, to privately read an entry, the client is required to download from $O(\lambda)$ to $O(\sqrt{N})$ extra entries. This cost is significant for practical public databases (e.g., [11–13]) with billions of entries, large-scale content distribution systems (e.g., [36, 37, 63]), or real data platforms (e.g., [1, 2, 5, 8–10, 59]) where the entry or page size granularity can be large (e.g., 4 KB-1 MB). For example, the most efficient OO-PIR scheme to date [72] requires nearly 160 GB of client storage and 525 MB of bandwidth cost to query a 16 KB entry from a database of 2^{32} entries.

Given the above limitations in client metrics of existing OO-PIR designs, we ask the following research question:

Can we design an OO-PIR scheme with low client bandwidth and storage overhead for large databases while retaining the sublinear client and server processing efficiency?

1.1 Our Contributions

We answer the question affirmatively by presenting an efficient two-server OO-PIR framework with semi-honest security called Pirex, which stands for Private Information Retrieval with Client Expedience. To our knowledge, Pirex is the first OO-PIR that offers $O(1)$ client inbound bandwidth blowup, and low client storage

Proceedings on Privacy Enhancing Technologies YYYY(X), 1–21

https://doi.org/XXXXXXX.XXXXXXX

Table 1: Our proposed Pirex/Pirex+ **schemes vs. prior (semi-honest) OO-PIR schemes.**

Scheme	**#S**[†]	**#R**[†]	**Client Online B/W**		**Online Computation**[‡]		**Storage**		**Total Client**	**Offline Computation**[‡]	
			In	**Out**	**Client**	**Server**	**Client**	**Server**	**Offline B/W**	**Client**	**Server**
CK20 (PRF) [29]	2	1	$O(\lambda B)$	$\tilde{O}(\lambda^2)$	$O(\lambda B)_\oplus + \tilde{O}(\lambda N)_\mathrm{p}$	$O(\lambda B\sqrt{N})_\oplus + O(\lambda\sqrt{N})_\mathrm{p}$	$\tilde{O}(\lambda B\sqrt{N})$	$O(BN)$	$\tilde{O}(\lambda B\sqrt{N})$	$\tilde{O}(\lambda\sqrt{N})_\mathrm{p}$	$\tilde{O}(\lambda BN)_\oplus + \tilde{O}(\lambda N)_\mathrm{p}$
CK20 (PRP) [29]				$\tilde{O}(\lambda\sqrt{N})$	$O(\lambda B)_\oplus + \tilde{O}(\lambda\sqrt{N})_\mathrm{p}$	$O(\lambda B\sqrt{N})_\oplus$					
Shi et al. [65]	2	1	$O(\lambda B)$	$\tilde{O}(\lambda^2)$	$O(\lambda B)_\oplus + \tilde{O}(\lambda\sqrt{N})_\mathrm{p}$	$\tilde{O}(\lambda B\sqrt{N})_\oplus + \tilde{O}(\lambda\sqrt{N})_\mathrm{p}$	$\tilde{O}(\lambda B\sqrt{N})$	$O(BN)$	$\tilde{O}(\lambda B\sqrt{N})$	$\tilde{O}(\lambda\sqrt{N})_\mathrm{p}$	$\tilde{O}(\lambda BN)_\oplus + \tilde{O}(\lambda N)_\mathrm{p}$
Checklist [41]	2	1	$O(B)$	$\tilde{O}(\lambda)$	$O(B)_\oplus + O(N)_\mathrm{p}$	$O(B\sqrt{N})_\oplus + O(\sqrt{N})_\mathrm{p}$	$O(\lambda B\sqrt{N})$	$O(BN)$	$O(\lambda B\sqrt{N})$	$O(\lambda\sqrt{N})_\mathrm{p}$	$O(\lambda BN)_\oplus + O(\lambda N)_\mathrm{p}$
TreePIR [44]	2	1	$O(B\sqrt{N})$	$\tilde{O}(\lambda)$	$O(B)_\oplus + \tilde{O}(\lambda\sqrt{N})_\mathrm{p}$	$O(B\sqrt{N})_\oplus + O(\sqrt{N})_\mathrm{p}$	$O(\lambda B\sqrt{N})$	$O(BN)$	$O(\lambda B\sqrt{N})$	$O(\lambda\sqrt{N})_\mathrm{p}$	$O(\lambda BN)_\oplus + O(\lambda N)_\mathrm{p}$
Piano [72]	1	1	$O(B\sqrt{N})$	$\tilde{O}(\sqrt{N})$	$O(B)_\oplus + O(\lambda\sqrt{N})_\mathrm{p}$	$O(B\sqrt{N})_\oplus$	$O(\lambda B\sqrt{N})$	$O(BN)$	$O(BN)$	$O(\lambda BN)_\oplus +$ $O(\lambda N)_\mathrm{p}$	–
SinglePass [45]	2	1	$O(B\sqrt{N})$	$\tilde{O}(\sqrt{N})$	$O(B\sqrt{N})_\oplus + O(\sqrt{N})_\mathrm{p}$	$O(B\sqrt{N})_\oplus$	$O(B\sqrt{N})$	$O(BN)$	$O(B\sqrt{N})$	$O(\sqrt{N})_\mathrm{p}$	$O(BN)_\oplus + O(N)_\mathrm{p}$
Pirex	2	1	$O(B)$	$\tilde{O}(\sqrt{N})$	$O(B)_\oplus + O(\lambda\sqrt{N})_\mathrm{p}$	$O(B\sqrt{N})_\oplus$	$O(\lambda B\sqrt{N})$	$O(BN)$	$O(\lambda B\sqrt{N})$	$O(\lambda\sqrt{N})_\mathrm{p}$	$O(\lambda BN)_\oplus + O(\lambda N)_\mathrm{p}$
Pirex+		1		$\tilde{O}(\sqrt{N}) +$ $O(B)$	$O(B)_{\mathbb{F}/\mathbb{G}} + O(\lambda\sqrt{N})_\mathrm{p}$	$O(B\sqrt{N})_\mathbb{F} + O(B\lambda\sqrt{N})_\oplus$	$O(\lambda^2\sqrt{N})$	$O(BN) +$ $O(\lambda B\sqrt{N})$	$O(\lambda B\sqrt{N})$	$O(\lambda B\sqrt{N})_\mathbb{G} +$ $O(\lambda\sqrt{N})_\mathrm{p}$	$O(\lambda BN)_\mathbb{F} + O(\lambda N)_\mathrm{p}$

‡ $\oplus$ denotes bitwise XOR operations, $\mathbb{F}$ denotes finite field arithmetic operations, p denotes PRF/PRP operations, $\mathbb{G}$ denotes group operations. For simplicity, we use the notation $\tilde{O}(\cdot)$ to hide the multiplicative polylog(N) terms in the asymptotic complexity. † #S denotes the number of servers. #R denotes the number of communication rounds.

with sublinear client and server processing time. In particular, Pirex offers desirable properties as follows:

- **Minimal client inbound bandwidth**: The main property of Pirex is the minimal client inbound bandwidth that is *independent* of the number of database entries. To retrieve an entry privately, Pirex only requires the client to download four entries. This cost is asymptotically (and concretely) lower than the state of the art (e.g., [44, 72]) which download $O(\sqrt{N})$ entries. The total client bandwidth of Pirex is $O(B+\sqrt{N}\log N)$ compared to $O(\sqrt{N}(\lambda+B))$ in other OO-PIR schemes [44, 72], with N, B, λ are the number of entries, entry size, and security parameter, respectively.
- **Low client storage overhead:** We present Pirex+, an extended Pirex scheme that offers the client a low storage cost, which is desirable for constrained client devices (e.g., mobile). Pirex+ only requires the client $O(\lambda^2\sqrt{N})$ bits for a precomputed hint that is *independent* of the entry size B, compared to $\tilde{O}(\lambda B\sqrt{N})$ in [29] and $O(\lambda B\sqrt{N})$ in [44, 72]. Prior works [45] also tried to reduce this cost by λ factors but requires downloading $O(\sqrt{N})$ entries. Concretely, for a 1 TB database of 256 KB entries, Pirex+ only requires 709 KB client storage, compared with 536 MB, 11.5 GB or 1.3 TB in others (i.e., two to six orders of magnitude smaller).
- **Sublinear computational overhead:** Pirex retains the sublinear processing efficiency from state-of-the-art OO-PIR. The servers perform $O(B\sqrt{N})$ low-cost operations (e.g., XOR, modular addition). The client only invokes sublinear low-cost PRF evaluations and performs a constant amount of XOR operations.
- **Extremely low end-to-end delay:** Thanks to the asymptotic bandwidth and computation costs, Pirex achieves a concretely low end-to-end delay for public database access. Pirex requires only simple cryptographic operations (e.g., XOR, PRF, modular arithmetic). Under real-world settings, Pirex is up to two orders of magnitudes faster, since it only takes 55ms to privately read a 4 KB entry in a 1 TB database, compared with 9s-30s in other schemes (see §6 for more comprehensive experiments).
- **Technique: Private Partition Retrieval.** As a core building block for Pirex, we design Private Partition Retrieval (PPR), a protocol that permits private retrieval of an arbitrary entry from a partitioned database with sublinear complexity. We developed a concrete instantiation for PPR and formally proved that it achieves the desired security.

Table 1 summarize the performance of Pirex/Pirex+ compared to state-of-the-art OO-PIR. We analyzed the security and rigorously proved they satisfy PIR security definition. We fully implemented our schemes and intensively evaluated their efficiency on commodity hardware. Experiments showed that Pirex/Pirex+ significantly outperforms state-of-the-art in all online metrics, especially in large database settings with large entry sizes. Our implementation source code is available at https://github.com/vt-asaplab/pirex.

1.2 Technical Highlights

Pirex relies on an elegant OO-PIR blueprint from Corrigan-Gibbs and Kogan [29] (we call it CK20 for brevity). We briefly present their high-level idea, along with TreePIR [44] as the follow-up attempt, and present our ideas to address the drawbacks in their designs.

CK20 [29]. CK20 operates on two non-colluding servers, Left and Right, with two phases: offline and online. Each server maintains a replica of a public database DB with N entries.

In the offline phase, the client precomputes $M = \tilde{O}(\sqrt{N})$ hints $\mathcal{H} = (h_1, \ldots, h_M)$. Each hint $h_i = (\mathcal{S}_i, \rho_i)$ contains a set of random indices $\mathcal{S}_i = \{s_0^{(i)}, \ldots, s_{\sqrt{N}-1}^{(i)}\}$ and a parity $\rho_i = \bigoplus_{j=0}^{\sqrt{N}-1} \mathsf{DB}[s_j^{(i)}]$ that is computed by sending the set $\mathcal{S}_i$ to the Left server. However, the storage cost for $\mathcal{H}$ is $O(N\log^2 N)$ as it takes $O(\sqrt{N}\log N)$ in space per set. To reduce this cost, each $\mathcal{S}_i$ is represented by a λ-bit PRF/PRP key sk_i, resulting in $O(M(\lambda+B))$, with B as the entry size. The offline bandwidth is $O(M(\lambda+B))$ as M keys are sent to the Left server to receive M parities.

To retrieve a desired data entry $\mathsf{DB}[x]$ in the online access, the idea is to recover $\mathsf{DB}[x]$ from $(\mathcal{S}_i, \rho_i)$, with $x \in \mathcal{S}_i$. To do this, the client sends a *punctured set* $\hat{\mathcal{S}} = \mathcal{S}_i \setminus \{x\}$ to the Right server, which in turn, replies $\hat{\rho} = \bigoplus_{j=0}^{\sqrt{N}-2} \mathsf{DB}[\hat{s}_j]$, with $\hat{s}_j \in \hat{\mathcal{S}}$. To recover $\mathsf{DB}[x]$, the client computes $\mathsf{DB}[x] = \hat{\rho} \oplus \rho_i$. Since $\mathcal{S}_i$ is partially exposed to both servers, h_i needs to be replaced with a new random hint $h' = (\mathcal{S}', \rho')$ using a *refresh* operation. The client samples $\mathcal{S}'$ with $x \in \mathcal{S}'$ using bias sampling. A new offline parity $\rho' = \mathsf{DB}[x] \oplus \hat{\rho}'$ is computed by sending $\hat{\mathcal{S}}' = \mathcal{S}' \setminus \{x\}$ to the Left server to obtain $\hat{\rho}' = \bigoplus_{j=0}^{\sqrt{N}-2} \mathsf{DB}[\hat{s}'_j]$. Note that $\hat{\mathcal{S}}$ or $\hat{\mathcal{S}}'$ is created by removing the desired index x. Receiving $\hat{\mathcal{S}}$ or $\hat{\mathcal{S}}'$, the servers certainly learn that the entry being accessed is *not* in $\hat{\mathcal{S}}$ or $\hat{\mathcal{S}}'$, thereby violating PIR security. Thus, the client performs a probabilistic puncture such

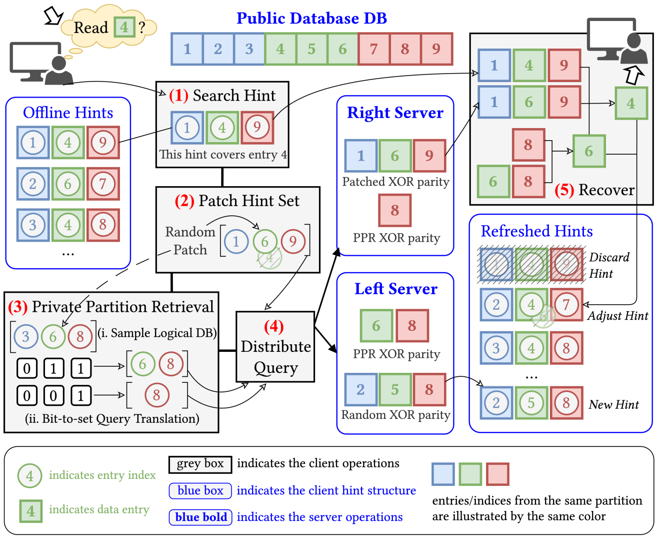


Figure 1: An illustration of the online phase in our proposed OO-PIR.

that a random index $x' \neq x$ is removed with probability $\frac{(\sqrt{N}-1)}{N}$, which results in non-negligible probability of failure. To ensure correctness, the client executes $O(\lambda)$ protocol instances in parallel. Note that there exists a trade-off in the online phase of CK20, where the client overhead depends on if the sets are represented by PRF or PRP keys. For PRF keys, since the PRF outputs a random index, it takes $O(M\sqrt{N})$ to find which set containing index x, but the outbound bandwidth is reduced to $O(\lambda \log N)$ by sending a punctured PRF key. For PRP keys, the lookup time is $O(M \log N)$, at the cost of $O(\sqrt{N} \log N)$ outbound bandwidth since PRP is not puncturable.

TreePIR [44]. To reduce both the client outbound bandwidth and lookup time to $O(\lambda \log N)$ and $O(M)$, respectively, Lazzaretti and Papamanthou proposed a *partitioning technique* combined with puncturable PRF. DB is divided into $\sqrt{N}$ partitions $(P_0, \ldots, P_{\sqrt{N}-1})$, with P_j covers indices in the range $(j\sqrt{N}, \ldots, (j+1)\sqrt{N} - 1)$. The idea is to have each set $\mathcal{S}_i = \{s_0^{(i)}, \ldots, s_{\sqrt{N}-1}^{(i)}\}$ represented by a PRF key sk_i, where index $s_j^{(i)}$ is generated by an offset $\delta_j^{(i)} \leftarrow \mathsf{PRF}(\mathsf{sk}_i, j)$ from partition P_j. Thus, an offline set $\mathcal{S}_i$ now corresponds to an offset vector $\Delta_i = (\delta_0^{(i)}, \ldots, \delta_{\sqrt{N}-1}^{(i)})$, with $s_j^{(i)} = j\sqrt{N} + \delta_j^{(i)}$. To check if $x \in \mathcal{S}_i$ in $O(1)$ for each hint in the online, the client computes $k = \lfloor \frac{x}{\sqrt{N}} \rfloor$, and check if $x = k\sqrt{N} + \mathsf{PRF}(\mathsf{sk}_i, k)$. To recover $\mathsf{DB}[x]$ from the offline parity $\rho_i = \bigoplus_{j=0}^{\sqrt{N}-1} \mathsf{DB}[s_j^{(i)}]$, the client needs the punctured parity of $\mathcal{S}_i \setminus \{x\}$ from the Right server. The client sends a corresponding punctured offset vector $\Delta = (\delta_0, \ldots, \delta_{\sqrt{N}-2})$ that is compressed under a punctured key of size $O(\lambda \log N)$ derived from sk_i. As Δ has $\sqrt{N} - 1$ offsets, there are $\sqrt{N}$ possible parities. For each $\Delta_{j^*} = (\delta_0, \ldots, \delta_{j-1}, \perp, \delta_{j+1}, \ldots, \delta_{\sqrt{N}-1})$, the Right server computes $\hat{\rho}_{j^*} = \bigoplus_{j \neq j^*}^{\sqrt{N}-1} \mathsf{DB}[j\sqrt{N} + \delta_j]$. To retrieve $\hat{\rho}_k$ for recovering $\mathsf{DB}[x] = \hat{\rho}_k \oplus \rho_i$, the client can download $\sqrt{N}$ values or execute a single-server PIR instance (which can be costly). To refresh the hint, the client samples a new key sk' so that $x - k\sqrt{N} = \mathsf{PRF}(\mathsf{sk}', k)$, then sends the punctured key to the Left server to obtain $\hat{\rho}'_k$ and compute a new parity $\rho' = \mathsf{DB}[x] \oplus \hat{\rho}'_k$. Note that query privacy is ensured as by observing Δ of $\sqrt{N} - 1$ random offsets, the servers only know the partition of x with $\frac{1}{\sqrt{N}}$ probability. Thus, TreePIR does not need probabilistic puncture as in CK20. However, the client's inbound bandwidth incurs to $O(B\sqrt{N})$ due to $\sqrt{N}$ parities transmission.

Idea 1: Patch the punctured vector using a random offset. To reduce the client's inbound bandwidth while retaining the efficient client lookup, our idea is to patch the punctured vector Δ generated from PRF to operate on the partitioned database. Figure 1 illustrates the high-level workflow of our proposed scheme to incorporate this idea. We observe that although the partitioning technique offers sublinear client lookup, it incurs high client bandwidth since the punctured query vector Δ removes one offset from a hidden partition, which requires the retrieval of $\sqrt{N}$ possible answers to hide that partition. Thus, we propose to patch Δ with a random offset $\bar{\delta}$ from the hidden partition k as $\Delta = (\delta_0, \ldots, \delta_{k-1}, \bar{\delta}, \delta_{k+1}, \ldots, \delta_{\sqrt{N}-1})$ (Step 2 Figure 1). The query vector now contains $\sqrt{N}$ offsets rather than $\sqrt{N}-1$ offsets represented by a PRF key. Let $\bar{z} \leftarrow k\sqrt{N}+\bar{\delta}$ be the index of the patching offset $\bar{\delta}$ in DB. In this case, the client obtains one patched parity $\bar{\rho} = (\bigoplus_{j \neq k} \mathsf{DB}[j\sqrt{N} + \delta_j]) \oplus \mathsf{DB}[\bar{z}]$ from the Right server. While this idea reduces the client inbound bandwidth to $O(1)$, it also impacts the reconstruction correctness since the client obtains $\bar{\rho} \oplus \rho_i = \mathsf{DB}[x] \oplus \mathsf{DB}[\bar{z}]$ instead of $\mathsf{DB}[x]$.

Idea 2: Retrieve the random patch with private partition retrieval. To address the reconstruction correctness issue due to patching, we need to somehow privately retrieve $\mathsf{DB}[\bar{z}]$ to compute $\bar{\rho} \oplus \hat{\rho}_i \oplus \mathsf{DB}[\bar{z}] = \mathsf{DB}[x]$. As there are $\sqrt{N}$ partitions and the offset of the patching entry from the desired partition (i.e., the value $\bar{\delta}$ of $\bar{z}$) is arbitrary, our initial idea is to execute the standard XOR-PIR protocol [26] on a $\sqrt{N}$-sized "logical" database containing $\mathsf{DB}[\bar{z}]$ plus $\sqrt{N} - 1$ entries $\mathsf{DB}[\mathfrak{z}]$ selected randomly from every other partition of DB (Step 3.i Figure 1). However, it is not trivial to apply XOR-PIR on this logical database. This is because standard XOR-PIR requires two servers to maintain the same database to evaluate the client queries. That means both servers must have access to the same logical database being created for correct evaluation. Meanwhile, it is insecure to reveal the entire logical database to both servers since it contains $\mathsf{DB}[\bar{z}]$, which has previously been revealed to the Right server in the patching step (Idea 1). Thus, after creating the XOR-PIR queries for the logical database, we perform an additional processing (Step 3.ii Figure 1) that permits the servers to evaluate XOR-PIR as usual without disclosing the entire logical database.

We observe that in standard XOR-PIR, only the active bits in the XOR-PIR bit queries matter as the servers only evaluate the database entries corresponding to these bits, while omitting all zero bits. Thus, it suffices to only reveal to each server the selected entries in the original database that corresponds to the active bits of XOR-PIR queries created on the logical database. Specifically, let $\mathbf{z} = (\mathfrak{z}_1, \ldots, \mathfrak{z}_{k-1}, \bar{z}, \mathfrak{z}_{k+1}, \ldots, \mathfrak{z}_{\sqrt{N}})$ be the index of the selected entries (including the patch $\bar{z}$), $\mathbf{e}$ be the unit vector representing the location (i.e., partition) of $\bar{z}$ in $\mathbf{z}$ (i.e., $\mathbf{e}[k] = 1$) and $\mathbf{v}_0, \mathbf{v}_1 \in \{0,1\}^{\sqrt{N}}$ be two random XOR-PIR queries such that $\mathbf{v}_0 \oplus \mathbf{v}_1 = \mathbf{e}$. We create two sets $\mathcal{T}_0 = \{\mathbf{z}[i] : \mathbf{v}_0[i] = 1\}$ and $\mathcal{T}_1 = \{\mathbf{z}[i] : \mathbf{v}_1[i] = 1\}$ (Step 3.ii Figure 1). Since $\mathbf{v}_0$ and $\mathbf{v}_1$ are the same, except the k-th position, $\bar{z}$ will only appear in either $\mathcal{T}_0$ or $\mathcal{T}_1$. In this case, we distribute the set ($\mathcal{T}_\mathsf{L}$) that contains $\bar{z}$ to the Left server and the other set ($\mathcal{T}_\mathsf{R}$) to the Right server (Step 4 Figure 1). This bit-to-set translation strategy allows each server to obtain a set of uniformly random elements that are *independent* to the patching step, thereby hiding what partition is privately retrieved. On receiving the query set, each server performs XOR-PIR evaluation as usual as $w_\mathsf{L} = \bigoplus_{j \in \mathcal{T}_\mathsf{L}} \mathsf{DB}[j]$ and

$w_{\mathsf{R}} = \bigoplus_{j \in \mathcal{T}_{\mathsf{R}}} \mathsf{DB}[j]$. Finally, the client recovers $\mathsf{DB}[\bar{z}]$ by computing $w_{\mathsf{L}} \oplus w_{\mathsf{R}} = \mathsf{DB}[\bar{z}]$ (Step 5 Figure 1). Since the set size is $O(\sqrt{N})$, the cost to privately retrieve $\mathsf{DB}[\bar{z}]$ is $O(\sqrt{N})$ and thus, does not asymptotically increase the complexity of OO-PIR's online query protocol overall. Finally, as the retrieved patch $\mathsf{DB}[\bar{z}]$ is chosen randomly, we show that $\mathsf{DB}[\bar{z}]$ can further be used to refresh the consumed hint directly without requiring to send another refresh query as in prior works [29, 44] (see §4.3).

Idea 3: Remote parities storage via additive homomorphic encryption and oblivious write. OO-PIR paradigm (e.g., [28, 29, 41, 43–45, 65, 71, 72]) requires the client to store $\Omega(\lambda\sqrt{N})$ offline parities ρ_i, the size of each ρ_i is equal to the database entry. To reduce the client storage, we propose to store the parities ρ_i remotely (under IND-CPA encryption) on the database server. As the number of ρ_i is sublinear, we can utilize the standard XOR-PIR protocol [26] to privately access any ρ_i upon a request, without worsening the overall complexity much (see the cost of our Pirex+ in Table 1). The challenge arises when we refresh the parities given that each parity can be used only once due to the OO-PIR design. To perform refresh securely, we develop an oblivious refresh buffer based on [62] that temporarily stores refresh parities and obliviously merges them with the offline parities over time. Another challenge when remotely maintaining the parities is to update them according to changes in the public database. While database updates are not captured in PIR security model, remote parities must be updated privately as they are individually formed by aggregating a set of random database entries. If an entry changes and the affected parities are updated insecurely, the server learns the index distribution per set, compromising the privacy of client's online queries. To update the parities obliviously, we use Additive Homomorphic Encryption to create an encrypted updated vector (i.e., a binary vector with active bit at update positions), and then delegate the secure update task to the servers via additive homomorphic property.

Putting it all together. By combining the first two ideas, the client inbound bandwidth overhead is now minimal since it only contains a single patched parity $\bar{\rho}$ and a random $\mathsf{DB}[\bar{z}]$ from the punctured partition, obtained by adopting the standard XOR-PIR. At the high level, our OO-PIR design is extremely simple and computationally efficient as the total server computational cost is still $O(\sqrt{N})$ XOR operations. We present two OO-PIR schemes. The first scheme (Pirex) combines the first two ideas. The second scheme (Pirex+) is an extension that combines all three ideas. Compared with Pirex, Pirex+ reduces the client storage cost from $O(\lambda B\sqrt{N})$ to $O(\lambda^2\sqrt{N})$ at the cost of having slightly heavier $\tilde{O}(\sqrt{N})$ server computation.

2 Preliminary and Models

Notation. $[n]$ denotes $\{0, 1, \ldots, n-1\}$. $\mathsf{negl}(\cdot)$ refers to negligible functions and λ denotes security parameters. $x \xleftarrow{\$} [n]$ indicates x is randomly chosen from $[n]$. PPT refers to Probabilistic Polynomial Time. We denote $\oplus$ as the bit-wise XOR operation between two binary strings a and b of size n, such that $c_i = a_i \oplus b_i$ for $i \in [n]$. We denote the negation of bit b as $\neg b$. We denote $\mathbb{Z}_n$ as the cyclic group formed by a set of integers modulo n under the addition. We denote $\mathbb{G}$ as an arbitrary cyclic group with the prime order p, where $\langle 1 \rangle \in \mathbb{G}$ is a random generator and $\langle x \rangle \in \mathbb{G}$ is a group element that has a discrete logarithm $x \in \mathbb{Z}_p$ with base $\langle 1 \rangle$.

Pseudorandom function. We denote $\mathsf{PRF} : \{0,1\}^\lambda \times [n] \rightarrow [n]$ as a pseudorandom function (PRF). $\mathsf{PRF}(\mathsf{sk}, s)$ outputs a pseudorandom value $y \in [n]$ given a PRF key $\mathsf{sk} \in \{0,1\}^\lambda$ and a seed $s \in [n]$. A PRF is secure if given security parameter λ, the output $y \in [n]$ is computationally indistinguishable from $y' \xleftarrow{\$} [n]$.

System model. Our system consists of a client and two servers S_0 and S_1. Each server maintains a replica of the database DB of N entries (each of size B) and allows the client to access an arbitrary entry in DB. Our system is a two-server OO-PIR scheme as follows:

Definition 1. A 2-server OO-PIR scheme is a tuple of PPT algorithms $\mathsf{OO\text{-}PIR} = (\mathsf{Prep}, \mathsf{Query}, \mathsf{Answer}, \mathsf{Recover})$*:*

- $\mathcal{H} \leftarrow \mathsf{Prep}(\mathsf{DB}, N)$*: Given database* DB*, with* N *as the number of entries, it outputs an offline hint* $\mathcal{H}$.
- $(Q_0, Q_1, \mathcal{H}^*) \leftarrow \mathsf{Query}(x, \mathcal{H})$*: Given index* x *and hint* $\mathcal{H}$*, it outputs query* Q_0 *for server* S_0*,* Q_1 *for* S_1*, and an updated hint* $\mathcal{H}^*$.
- $\mathcal{R}_l \leftarrow \mathsf{Answer}(Q_l, \mathsf{DB})$*: Given an online query* $Q_l \in \{Q_0, Q_1\}$ *and the database* DB*, it outputs a response* $\mathcal{R}_l$.
- $(b_x, \mathcal{H}') \leftarrow \mathsf{Recover}(\mathcal{R}_0, \mathcal{R}_1, \mathcal{H}^*)$*: Given hint* $\mathcal{H}^*$ *and responses* $\mathcal{R}_0, \mathcal{R}_1$*, it outputs the desired entry* b_x *and an updated hint* $\mathcal{H}'$.

Definition 2 (OO-PIR Correctness). A 2-server OO-PIR scheme is correct if for any DB *and* $\mathcal{H} \leftarrow \mathsf{Prep}(\mathsf{DB}, N)$*, given security parameter* λ *and an unbound number of prior queries, there exists a negligible function* $\mathsf{negl}(\lambda)$ *for any index* $x \in [N]$*:*

$$\Pr\left[b_x \neq \mathsf{DB}[x] \;\middle|\; \begin{array}{r} (Q_0, Q_1, \mathcal{H}^*) \leftarrow \mathsf{Query}(x, \mathcal{H}) \\ \mathcal{R}_0 \leftarrow \mathsf{Answer}(Q_0, \mathsf{DB}) \\ \mathcal{R}_1 \leftarrow \mathsf{Answer}(Q_1, \mathsf{DB}) \\ (b_x, \mathcal{H}') \leftarrow \mathsf{Recover}(\mathcal{R}_0, \mathcal{R}_1, \mathcal{H}^*) \end{array} \right] \leq \mathsf{negl}(\lambda)$$

Threat model. The client is trusted. The servers are semi-honest and follow the protocol but are curious on the entry being queried by the client. We consider static corruption, an adversary $\mathcal{A}$ can corrupt either server S_0 or S_1 but not both, and can not adaptively switch between two servers during the protocol execution.

Security model. We define the security of our scheme using the Ideal/Real paradigm, such that an adversary $\mathcal{A}$ statically corrupting one server learns nothing about the entry being retrieved. Let $\mathcal{F}$ be an ideal functionality that answers the query honestly. Let $\mathcal{S}$ be an ideal simulator that emulates the view of the real-world adversary. Let $\mathcal{Z}$ be the environment that provides inputs for all entities and receives corresponding outputs. $\mathcal{Z}$ can get any adversarial views at any time. We define the Ideal/Real world as follows:

- Ideal: In the offline, on receiving (DB, N), $\mathcal{F}$ notifies $\mathcal{S}$ about the content of DB. $\mathcal{S}$ emulates the adversarial view of offline execution and replies to $\mathcal{F}$ with ok or $\perp$. In the online, on receiving an index $x \in [N]$, $\mathcal{F}$ notifies $\mathcal{S}$ about the event (but not x). $\mathcal{S}$ emulates the adversarial view of online execution and replies to $\mathcal{F}$ with ok or $\perp$. If $\mathcal{S}$ says ok, $\mathcal{F}$ returns the entry $\mathsf{DB}[x]$.
- Real: In the offline, on receiving (DB, N), the client honestly executes $\mathcal{H} \leftarrow \mathsf{Prep}(\mathsf{DB}, N)$ with two servers to obtain a private hint $\mathcal{H}$. In the online, on receiving an index x, the client honestly executes $(Q_0, Q_1, \mathcal{H}^*) \leftarrow \mathsf{Query}(x, \mathcal{H})$ and sends Q_0 to server S_0,

Q_1 to server S_1. Each server S_l executes $\mathcal{R}_l \leftarrow \mathsf{Answer}(Q_l, \mathrm{DB})$ and responses $\mathcal{R}_l$. The client obtains the output b_x by execute $(b_x, \mathcal{H}') \leftarrow \mathsf{Recover}(\mathcal{R}_0, \mathcal{R}_1, \mathcal{H}^*)$.

Definition 3 (OO-PIR Security). A 2-server OO-PIR $\Pi_{\mathcal{F}}$ is secure in realizing $\mathcal{F}$ if for every PPT real adversary $\mathcal{A}$, there is a PPT simulator $\mathcal{S}$, such that for all non-uniform, polynomial-time environment $\mathcal{Z}$, the following distributions are computationally indistinguishable:

$$|\Pr[\textsc{Real}_{\Pi_{\mathcal{F}},\mathcal{A},\mathcal{Z}}(\lambda) = 1] - \Pr[\textsc{Ideal}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(\lambda) = 1]| \leq \mathsf{negl}(\lambda)$$

3 Private Partition Retrieval

We present Private Partition Retrieval (PPR), a technique that allows the client to privately read (a random entry from) a DB partition, without revealing the partition index of interest.

Definition 4 (Private Partition Retrieval). A 2-server PPR scheme is a tuple of PPT algorithms PPR = (Gen, Ret, Rec)*:*

- $\underline{(\bar{z}, \mathcal{T}_0, \mathcal{T}_1) \leftarrow \mathsf{Gen}(m, n, k)}$*: Given the partition size m, the number of partition n and a partition index $k \in [n]$, it outputs two partition queries $\mathcal{T}_0$, $\mathcal{T}_1$, and a random chosen index $\bar{z}$.*
- $\underline{r_l \leftarrow \mathsf{Ret}(\mathcal{T}_l, \mathrm{DB})}$*: Given a query $\mathcal{T}_l \in \{\mathcal{T}_0, \mathcal{T}_1\}$ and the partitioned database* DB*, it outputs a response r_l.*
- $\underline{b_{\bar{z}} \leftarrow \mathsf{Rec}(r_0, r_1)}$*: Given two responses r_0 and r_1, it outputs the data entry $b_{\bar{z}}$ at random index $\bar{z}$ from partition k.*

We define the PPR security using the Ideal/Real paradigm such that an adversary $\mathcal{A}$ statically corrupting one server learns nothing about the partition being accessed. Let $\mathcal{F}_P$ be an ideal functionality that honestly returns an arbitrary entry from the desired partition. Let $\mathcal{S}_P$ be an ideal simulator that emulates the view of the real-world adversary. Let $\mathcal{Z}$ be the environment that provides inputs for all entities and receives the corresponding outputs. $\mathcal{Z}$ can get the adversarial views at any time.

- Ideal: In the setup, on receiving (DB, N) and partition parameters (m, n), $\mathcal{F}_P$ notifies $\mathcal{S}_P$ about the content of DB and its partition size. $\mathcal{S}_P$ replies to $\mathcal{F}_P$ with ok or $\perp$. For each read access, on receiving a partition index k, $\mathcal{F}_P$ notifies $\mathcal{S}_P$ about the event (but not the partition range). $\mathcal{S}_P$ then emulates the adversarial view of execution and replies to $\mathcal{F}_P$ with ok or $\perp$. If $\mathcal{S}_P$ says ok, then $\mathcal{F}_P$ returns an arbitrary entry $\mathrm{DB}[\bar{z}]$ from partition k.
- Real: In the setup, on receiving (DB, N) and partition parameters (m, n), the servers divide DB into n partitions. For each access, on receiving a partition index k, the client honestly executes $(\bar{z}, \mathcal{T}_0, \mathcal{T}_1) \leftarrow \mathsf{Gen}(m, n, k)$ and sends $\mathcal{T}_0$ to server S_0, $\mathcal{T}_1$ to server S_1. Server S_l executes $r_l \leftarrow \mathsf{Ret}(\mathcal{T}_l, \mathrm{DB})$ and responses with r_l. The client executes $b_{\bar{z}} \leftarrow \mathsf{Rec}(r_0, r_1)$ to obtain an arbitrary data entry $b_{\bar{z}}$ from partition k.

Definition 5 (PPR Security). A 2-server PPR scheme $\Pi_{\mathcal{F}_P}$ is secure in realizing $\mathcal{F}_P$ if for every PPT real adversary $\mathcal{A}$, there is a PPT simulator $\mathcal{S}_P$, such that for all non-uniform, polynomial-time environment $\mathcal{Z}$, the following distributions are computationally indistinguishable:

$$|\Pr[\textsc{Real}_{\Pi_{\mathcal{F}_P},\mathcal{A},\mathcal{Z}}(\lambda) = 1] - \Pr[\textsc{Ideal}_{\mathcal{F}_P,\mathcal{S}_P,\mathcal{Z}}(\lambda) = 1]| \leq \mathsf{negl}(\lambda)$$

Protocol details. Figure 2 presents a concrete PPR protocol. Given the partition parameters (m offsets, n partitions) and the partition index $k \in [n]$ to be accessed, the protocol starts by invoking the PPR.Gen algorithm to create two queries $\mathcal{T}_0$ to server S_0 and $\mathcal{T}_1$ to server S_1. The client first samples n random offsets (line 1), where offset δ_k is the location of a random entry $\mathrm{DB}[\bar{z}]$ to be read from the desired partition k. It then creates two bit vectors (lines 3-4) where the only bits of difference is between e_k and e'_k. Given the sampled offsets, the client translates each bit vector into a corresponding set of indices (lines 5-6), which only reveals a random offset δ_i to be accessed in partition i if the bit e_i (or e'_i) is active. This is an important step as only the offset δ_k (of the random entry $\mathrm{DB}[\bar{z}]$) will be added to one set but not both, which can hide $\mathrm{DB}[\bar{z}]$ from one random server, allowing PPR to be employed in Pirex for secure queries. Hence, for each partition $i \in [n]$, the client adds the index $(i \cdot m + \delta_i)$ into the query set $\mathcal{T}_0$ (or $\mathcal{T}_1$) according to the bit e_i (or e'_i). If the bit is zero, no index for partition i will be added to the query, and partition i will not be accessed by the server. To this end, the client sends the query $\mathcal{T}_0$ to an arbitrary server $S_l \in \{S_0, S_1\}$ and query $\mathcal{T}_1$ to the remaining server. On receiving a query $\mathcal{T}_l$, the server S_l invokes the algorithm PPR.Ret with the public DB. The server accesses random entries on DB indicated by the set of indices $\mathcal{T}_l$ and aggregates them under XOR operation. To recover the randomly selected entry $\mathrm{DB}[\bar{z}]$, the client invokes PPR.Rec with the aggregated results r_0 and r_1 received from the servers. As the query set difference between $\mathcal{T}_0$ and $\mathcal{T}_1$ is the only index $\bar{z}$, combining r_0 and r_1 produces $b_{\bar{z}} = \mathrm{DB}[\bar{z}]$.

```
• $\underline{(\bar{z}, \mathcal{T}_0, \mathcal{T}_1) \leftarrow \mathsf{PPR.Gen}(m, n, k)}$:
 1: $(\delta_0, \ldots, \delta_k, \ldots, \delta_{n-1}) \xleftarrow{\$} [m]^n$
 2: let $\bar{z} \leftarrow k \cdot m + \delta_k$
 3: $(e_0, \ldots, e_k, \ldots, e_{n-1}) \xleftarrow{\$} \{0,1\}^n$
 4: $(e'_0, \ldots, e'_k, \ldots, e'_{n-1}) \leftarrow (e_0, \ldots, e_k \oplus 1, \ldots, e_n)$
 5: $\mathcal{T}_0 \leftarrow \{e_i \cdot (i \cdot m + \delta_i) \ \forall \ i \in [n]\}$
 6: $\mathcal{T}_1 \leftarrow \{e'_i \cdot (i \cdot m + \delta_i) \ \forall \ i \in [n]\}$
 7: **return** $(\bar{z}, \mathcal{T}_0, \mathcal{T}_1)$

• $\underline{r_l \leftarrow \mathsf{PPR.Ret}(\mathcal{T}_l, \mathrm{DB})}$:
 8: **parse** $\mathcal{T}_l = \{p_1^{(l)}, \ldots, p_{n'}^{(l)}\}$ with $n' = |\mathcal{T}_l|$
 9: **return** $r_l \leftarrow \bigoplus_{j=1}^{n'} \mathrm{DB}[p_j^{(l)}]$

• $\underline{b_{\bar{z}} \leftarrow \mathsf{PPR.Rec}(r_0, r_1)}$:
10: **return** $b_{\bar{z}} \leftarrow r_0 \oplus r_1$
```

Figure 2: Our proposed PPR protocol.

Lemma 1. *PPR scheme (Figure 2) is secure by Definition 5.*

Proof. See Appendix §A.1 □

4 The Proposed Scheme

4.1 Data Structure

Our scheme includes a database DB, and a hint buffer $\mathcal{H}$:

- DB is an array of N entries, divided into n partitions. Partition P_j covers m indices in range $[j \cdot m \ldots (j+1) \cdot m - 1]$, for $j \in [n]$. DB is replicated to 2 servers. For simplicity, we set $m = n = \sqrt{N}$.
- $\mathcal{H}$ includes M precomputed hints $h_i = (\ell_i, \mathsf{sk}_i, \rho_i, Y_i)$, where sk_i is a key representing a set $\mathcal{S}_i = (s_0, \ldots, s_{n-1})$, $\rho_i = \bigoplus_{j=0}^{n-1} \mathrm{DB}[s_j]$

```
• $\mathcal{H} \leftarrow \mathsf{Prep}(\mathsf{DB}, N)$:
1: for $i = 1$ to $M$ do
2:     $\ell_i \xleftarrow{\$} \{0,1\}$ and $\mathsf{sk}_i \leftarrow \mathsf{PRS.Gen}(1^\lambda)$
3:     $\mathcal{S}_i \leftarrow \mathsf{PRS.Eval}(\mathsf{sk}_i, \perp)$                          } Executed by Server $\mathsf{S}_{\ell_i}$
4:     $\rho_i \leftarrow \bigoplus_{j=0}^{n-1} \mathsf{DB}[s_j]$ for all $s_j \in \mathcal{S}_i$    } on receiving a PRF key
                                                                      } $\mathsf{sk}_i$ from the client
5:     $h_i \leftarrow (\ell_i, \mathsf{sk}_i, \rho_i, Y_i)$ with $Y_i \leftarrow \perp$
6: return $\mathcal{H} \leftarrow (h_1, \ldots, h_M)$
```

Figure 3: Pirex - **offline phase.**

```
• $(Q_0, Q_1, \mathcal{H}^*) \leftarrow \mathsf{Query}(x, \mathcal{H})$:
1: parse $\mathcal{H} = (h_1, \ldots, h_M)$
2: search $h_i = (\ell_i, \mathsf{sk}_i, \rho_i, Y_i)$ where $x \in \mathcal{S}_i \leftarrow \mathsf{PRS.Eval}(\mathsf{sk}_i, Y_i)$
3: $(\bar{z}, \mathcal{T}, \mathcal{T}^{(z)}) \leftarrow \mathsf{PPR.Gen}(m, n, k)$ where $k = \lfloor \frac{x}{m} \rfloor$
4: $\bar{\mathcal{S}} \leftarrow \mathcal{S}_i \setminus \{x\} \cup \{\bar{z}\}$
5: $\mathcal{S}' \leftarrow \mathsf{PRS.Eval}(\mathsf{sk}', \perp)$ where $\mathsf{sk}' \leftarrow \mathsf{PRS.Gen}(1^\lambda)$
6: $Q_{\ell_i} \leftarrow (\mathcal{S}', \mathcal{T}^{(\bar{z})})$, $Q_{\neg\ell_i} \leftarrow (\bar{\mathcal{S}}, \mathcal{T})$
7: $\mathcal{H}^* \leftarrow (h_1, \ldots, h_i, \ldots, h_M)$
8: return $(Q_0, Q_1, \mathcal{H}^*)$
```

Figure 4: Pirex - **online phase: query.**

is an offline parity, $\ell_i \in \{0,1\}$ denotes server identity (S_{ℓ_i}) that computes the parity ρ_i, and Y_i is an auxiliary value.

Pseudorandom set. For efficient storage and substitution of set elements in our scheme, we use a pseudorandom set (PRS). Given that the set elements are indices from DB parameterized by (m, n), our PRS is constructed from $\mathsf{PRF} : \{0,1\}^\lambda \times [m] \rightarrow [m]$ with the following algorithms $\mathsf{PRS} = (\mathsf{Gen}, \mathsf{Eval})$:

- $\underline{\mathsf{sk} \leftarrow \mathsf{Gen}(1^\lambda)}$: It outputs a PRF key $\mathsf{sk} \xleftarrow{\$} \{0,1\}^\lambda$.
- $\underline{\mathcal{S} \leftarrow \mathsf{Eval}(\mathsf{sk}, Y)}$: Given a PRF key $\mathsf{sk} \in \{0,1\}^\lambda$ and an auxiliary $Y = (y_1, \ldots, y_t)$, it outputs a set $\mathcal{S} = (s_0, \ldots, s_{n-1})$ such that $s_j = (j \cdot m) + \mathsf{PRF}(\mathsf{sk}, j)$ for $j \in [n]$ and some elements are replaced by the auxiliary as $s_{\gamma_i} = y_i$, where $\gamma_i = \lfloor \frac{y_i}{m} \rfloor$ for $i \in [t]$.

4.2 Offline Phase

Figure 3 illustrates how the offline phase works. Given a database DB of size N, the client runs a one-time setup with server S_0 and S_1 to prepare a set $\mathcal{H}$ of M hints. The idea is to have each hint $h_i \in \mathcal{H}$ contain a key sk_i representing a set $\mathcal{S}_i$ of n indices that has a corresponding offline parity ρ_i. To do this, the client samples M PRF keys $(\mathsf{sk}_1, \ldots, \mathsf{sk}_M)$, then sends each key sk_i to a random server $\mathsf{S}_{\ell_i} \in \{\mathsf{S}_0, \mathsf{S}_1\}$ to compute the according parity ρ_i. The client selects a random server identifier ℓ_i (line 2) for processing each hint h_i to ensure that no server in the online phase can distinguish whether it receives a query set for recovery purpose or for refresh operation. On receiving each key sk_i, server S_{ℓ_i} generates the set of indices $\mathcal{S}_i = \{s_0, \ldots, s_{n-1}\}$, where index $s_j \in P_j$ and $s_j = (j \cdot m) + \mathsf{PRF}(\mathsf{sk}_i, j)$ (line 3). Given the set $\mathcal{S}_i$, server S_{ℓ_i} computes and returns an offline parity $\rho_i \leftarrow \bigoplus_{j=0}^{n-1} \mathsf{DB}[s_j]$ (line 4). On receiving M offline parities, the client finalizes the set of hints $\mathcal{H}$, where hint $h_i = (\ell_i, \mathsf{sk}_i, \rho_i, Y_i)$ denotes server S_{ℓ_i} used the key sk_i to compute the offline parity ρ_i (lines 5-6) and Y_i is the auxiliary data that stores the index of entries being queried in the online phase for later hint refresh purposes. Obviously, as no online query has been made in the offline phase, the client sets Y_i as empty.

4.3 Online Phase

Figure 4 shows how the online phase works. To privately retrieve an entry $\mathsf{DB}[x]$, the client invokes the Query algorithm that uses the hint $\mathcal{H}$ to create queries Q_0 and Q_1 to server S_0 and S_1, respectively. The client performs two main actions as follows:

First, the client searches for a hint $h_i = (\ell_i, \mathsf{sk}_i, \rho_i, Y_i) \in \mathcal{H}$ where the parity ρ_i and the corresponding PRS set $\mathcal{S}_i$ contains information about the entry $\mathsf{DB}[x]$ (line 2). As each $\mathcal{S}_i$ is computed from PRF key sk_i and auxiliary data Y_i, this can be done efficiently by checking if $x \in Y_i$, or if $x \stackrel{?}{=} (k \cdot m) + \mathsf{PRF}(\mathsf{sk}_i, k)$ and there exists no element y in Y_i such that its partition is the same with the partition of x, i.e., $\lfloor \frac{y}{m} \rfloor = \lfloor \frac{x}{m} \rfloor$. Later, we will show that Y_i contains at most one element (see Lemma 2) so the cost of this membership test is $O(1)$.

Second, the client creates two queries Q_0 and Q_1 based on the set $\mathcal{S}_i$ such that when combining the responses with parity ρ_i, $\mathsf{DB}[x]$ is recovered and the hint buffer $\mathcal{H}$ is refreshed to preserve the hint distribution for future queries (lines 3-10). To recover $\mathsf{DB}[x]$ with $(\ell_i, \rho_i, \mathcal{S}_i)$, the client needs a punctured parity $\hat{\rho}_i$ such that $\mathsf{DB}[x] = \rho_i \oplus \hat{\rho}_i$. Since $\rho_i = \bigoplus_{j=0}^{n-1} \mathsf{DB}[s_j]$, for $s_j \in \mathcal{S}_i$, this only holds when $\hat{\rho}_i = \bigoplus_{j=0}^{n-2} \mathsf{DB}[\hat{s}_j]$, for $\hat{s}_j \in \mathcal{S}_i \setminus \{x\}$. However, revealing the punctured set $\hat{\mathcal{S}} = \mathcal{S}_i \setminus \{x\}$ (to obtain $\hat{\rho}_i$) permits the adversary to learn the partition P_k of x. This is because each index $s_j \in \mathcal{S}_i$ is belong to a distinct partition P_j. To prevent this leakage, our idea is to patch $\hat{\mathcal{S}}$ with a random index $\bar{z} \in P_k$. This results in a patched set $\bar{\mathcal{S}} = \{\bar{s}_1, \ldots, \bar{s}_n\} = \hat{\mathcal{S}} \cup \{\bar{z}\}$ which has a patched parity $\bar{\rho} = \bigoplus_{j=1}^{n} \mathsf{DB}[\bar{s}_j] = \hat{\rho}_i \oplus \mathsf{DB}[\bar{z}]$. To obtain $\hat{\rho}_i$, the client needs the patch $\mathsf{DB}[\bar{z}]$. We will incorporate our PPR protocol to create queries Q_0 and Q_1 so that $\bar{\rho}$ and $\mathsf{DB}[\bar{z}]$ can be securely retrieved without leaking the partition P_k as follows.

The client first samples $\bar{z} \in P_k$ and two PPR queries $\mathcal{T}, \mathcal{T}^{(z)}$. According to PPR, sending $\mathcal{T}$ and $\mathcal{T}^{(z)}$ to the servers permits the private retrieval of $\mathsf{DB}[\bar{z}]$. To obtain $\bar{\rho}$, the client also needs to send the patched set $\bar{\mathcal{S}} = \hat{\mathcal{S}} \cup \{\bar{z}\}$ to a server. Thus, we need to make sure the three sets $\mathcal{T}_0, \mathcal{T}_1$ and $\bar{\mathcal{S}}$ are distributed securely to two servers. Let $\mathcal{T}^{(\bar{z})} \in \{\mathcal{T}_0, \mathcal{T}_1\}$ be the PPR query that contains $\bar{z}$ and $\mathcal{T}$ be the other PPR query. Remark that for hint h_i, the identifier $\ell_i \in \{0,1\}$ reflects that server S_{ℓ_i} has observed the set $\mathcal{S}_i$ and its offline parity ρ_i in the offline phase. Therefore, the client must send the patched set $\bar{\mathcal{S}}$ to the other server $\mathsf{S}_{\neg\ell_i}$ in the online phase for security. Since $\bar{z} \in \bar{\mathcal{S}}$, it is critical to ensure the server $\mathsf{S}_{\neg\ell_i}$ will not receive $\mathcal{T}^{(\bar{z})}$ as there is a common $\bar{z}$ in $\bar{\mathcal{S}}$ and $\mathcal{T}^{(\bar{z})}$. In this case, $\bar{\mathcal{S}}$ must be paired with $\mathcal{T}$, meaning the server $\mathsf{S}_{\neg\ell_i}$ must receive $(\bar{\mathcal{S}}, \mathcal{T})$ and the other server S_{ℓ_i} must receive $\mathcal{T}^{(\bar{z})}$. However, a server can distinguish if it receives $(\bar{\mathcal{S}}, \mathcal{T})$ or $\mathcal{T}^{(\bar{z})}$. In either case, the servers learn a set of partitions that are certainly not client interest, which violates PIR security. Thus, the client must pair $\mathcal{T}^{(\bar{z})}$ with a dummy set $\mathcal{S}'$ that also contains $\sqrt{N}$ random indexes, thereby making $(\bar{\mathcal{S}}, \mathcal{T})$ and $(\mathcal{S}', \mathcal{T}^{(\bar{z})})$ indistinguishable from the server's perspective. To this end, the server $S_{\neg\ell_i}$ receives $Q_{\neg\ell_i} \leftarrow (\bar{\mathcal{S}}, \mathcal{T})$ and the other server S_{ℓ_i} receives $Q_{\ell_i} \leftarrow (\mathcal{S}', \mathcal{T}^{(\bar{z})})$.

```
• $\mathcal{R}_l \leftarrow$ Answer($Q_l$, DB):
1: parse $Q_l = (\mathcal{S}, \mathcal{T})$
2: $\bar{\rho} \leftarrow \bigoplus_{j=1}^{n} \mathsf{DB}[s_j]$ for all $s_j \in \mathcal{S}$
3: $w \leftarrow$ PPR.Ret($\mathcal{T}$, DB)
4: return $\mathcal{R}_l \leftarrow (\bar{\rho}, w)$
```

Figure 5: Pirex - **online phase: answer.**

```
• $(b_x, \mathcal{H}') \leftarrow$ Recover($\mathcal{R}_0, \mathcal{R}_1, \mathcal{H}^*$):
1: let $(\bar{z}, \mathsf{sk}')$ be the values from line 3 and 5 in Query algorithm
2: parse $\mathcal{H}^* = (h_1, \ldots, h_i, \ldots, h_M)$, $h_i = (\ell_i, \mathsf{sk}_i, \rho_i)$
3: parse $\mathcal{R}_{\ell_i} = (\rho', w^{(\bar{z})})$, $\mathcal{R}_{\neg \ell_i} = (\bar{\rho}, w)$
4: $b_{\bar{z}} \leftarrow$ PPR.Rec($w^{(\bar{z})}, w$)
5: $b_x \leftarrow \rho_i \oplus \bar{\rho} \oplus b_{\bar{z}}$
6: search $h_j = (\ell_j, \mathsf{sk}_j, \rho_j, Y_j)$ where $\bar{z} \in$ PRS.Eval($\mathsf{sk}_j, Y_j$)
7: $\rho'_j \leftarrow \rho_j \oplus b_{\bar{z}} \oplus b_x$
8: $Y'_j \leftarrow Y_j \cup \{x\} \setminus \{\bar{z}\}$
9: $h'_j \leftarrow (\ell_j, \mathsf{sk}_j, \rho'_j, Y'_j)$
10: $h' \leftarrow (\ell_i, \mathsf{sk}', \rho', \perp)$
11: $\mathcal{H}' \leftarrow (h_1, \ldots, h'_j, \ldots, h', \ldots, h_M)$
12: return $(b_x, \mathcal{H}')$
```

Figure 6: Pirex - **online phase: recover.**

On receiving a query $Q_l = (\mathcal{S}, \mathcal{T})$, each server computes the parity on $\mathcal{S}$ as $\bar{\rho} = \bigoplus_{j=1}^{n} \mathsf{DB}[s_j]\ \forall\, s_j \in \mathcal{S}$ and executes the PPR retrieval protocol on $\mathcal{T}$ to obtain the result w (lines 2-3 Figure 5). Each server returns $(\bar{\rho}, w)$ to the client. Let $(\rho', w^{(\bar{z})})$ and $(\bar{\rho}, w)$ be the answers the client receives from the servers S_{ℓ_i} and $\mathsf{S}_{\neg \ell_i}$, respectively. The client can reconstruct the desired entry $\mathsf{DB}[x]$ by executing the PPR reconstruction algorithm on $(w^{(\bar{z})}, w)$ to obtain the patched entry $b_{\bar{z}}$ followed by computing $\hat{\rho}_i = \bar{\rho} \oplus b_{\bar{z}}$, and then $\mathsf{DB}[x] = \rho_i \oplus \hat{\rho}_i$ (lines 4-5 Figure 6).

As the set $\mathcal{S}_i$ is exposed to both servers, the client must discard the hint h_i and replace it with another hint h'_j to preserve the hint distribution. As h_i was used to recover $\mathsf{DB}[x]$, the new hint h'_j to replace h_i must contain a parity subject to recovering $\mathsf{DB}[x]$. We show that the client can make use of all the materials in the online query (i.e., the patching entry $\mathsf{DB}[\bar{z}]$, the random set $\mathcal{S}'$ and its parity ρ') to refresh the hints without sending additional queries to the servers as prior works [29, 44].

To replace h_i, the client finds a random hint $h_j \in \mathcal{H}$ that covers $\mathsf{DB}[\bar{z}]$ and updates it to make it cover $\mathsf{DB}[x]$. Let $\mathcal{S}_j$ be the hint's representative set and ρ_j be its parity. As the entries $\mathsf{DB}[\bar{z}]$ and $\mathsf{DB}[x]$ are already obtained, the client can update the parity of h_j to $\rho'_j = \rho_j \oplus \mathsf{DB}[\bar{z}] \oplus \mathsf{DB}[x]$. In this case, the updated parity ρ'_j corresponds to the set $\mathcal{S}'_j = \mathcal{S}_j \setminus \{\bar{z}\} \cup \{x\}$, thus it can support the recovery of $\mathsf{DB}[x]$ in future online queries. Recall that $\mathcal{S}_j$ is represented by PRF key sk_j in the hint structure for small storage. Thus, to capture the updated elements in the set $\mathcal{S}'_j$ (i.e., replacing $\bar{z}$ with x), the client adds x to the auxiliary data Y_j of hint h_j and remove $\bar{z}$ (if any) as $Y'_j = Y_j \cup \{x\} \setminus \{\bar{z}\}$.

Since Y_j can already contain some indexes from prior protocol executions, adding x may increase the auxiliary size, impacting the membership test's time complexity in future queries. In Pirex, we prove the size of any auxiliary data (denoted Y_i) in the average case is constant regardless of the number of online queries in the following Lemma 2.

Lemma 2. *For every hint $h_i = (\ell_i, \mathsf{sk}_i, \rho_i, Y_i) \in \mathcal{H}$, let X_i be a random variable denoting the size of auxiliary data Y_i. We have $\mathbb{E}[X_i] = 1$.*

Proof. See Appendix §B.1 □

Since the client consumes an existing random hint (h_j) in the hint buffer to replace h_i, the client needs to add a random hint h' to the hint buffer to retain its size for future queries. This hint can be created using the dummy set $\mathcal{S}'$ (created by key sk') with its corresponding parity ρ' returned by the server S_{ℓ_i} from the online query Q_{ℓ_i} as $h' = (\ell_i, \mathsf{sk}', \rho', \perp)$ with an empty auxiliary data.

4.4 Analysis

We state the correctness and security of Pirex as follows.

Theorem 1. *By setting the hint buffer size $M = O(\alpha\sqrt{N})$, with $\alpha = \min(\lambda, \log N)$,* Pirex *achieves correctness by Definition 2.*

Proof. See Appendix §B.2 □

Theorem 2. Pirex *is secure by Definition 3.*

Proof. See Appendix §B.3. □

Complexity. We analyze the complexity of Pirex with parameters including number of DB entries (N), entry size (B) and security parameter (λ). We consider $M = O(\lambda\sqrt{N})$ for arbitrarily large N.

- Offline cost: For communication, the client sends $\lambda\sqrt{N}$ PRF keys to the servers for set representation and receives $\lambda\sqrt{N}$ parities correspondingly. As each PRF key is λ-bit and each parity is B-bit, the client inbound (*resp.* outbound) bandwidth cost is $O(B\lambda\sqrt{N})$ (*resp.* $O(\lambda^2\sqrt{N})$). The cost of sending $\lambda\sqrt{N}$ keys can be optimized by sending a single master key and let the server generate $\lambda\sqrt{N}$ PRF keys, leading to $O(\lambda B\sqrt{N})$ total outbound bandwidth.

For computation, the client performs $O(\lambda\sqrt{N})$ PRS invocations to generate the PRF keys. For each hint, the server performs $O(\sqrt{N})$ PRF evaluations and $O(\sqrt{N})$ XOR operations on B-bit data entries. Since there are $O(\lambda\sqrt{N})$ hints, the total server offline computation cost is $O(\lambda N)$ PRF evaluations and $O(B\lambda N)$ XOR operations.

- Online cost: For each online retrieval, the client sends one query to each server. Each query contains a set of $\sqrt{N}$ indices and a set of $O(\sqrt{N})$ partition indices. As each index is represented by $O(\log N)$ bits, the client outbound bandwidth is $O(\sqrt{N}\log N)$. The client inbound bandwidth is $O(B)$ as it receives four aggregated results. The total bandwidth is $O(\sqrt{N}\log N + B)$.

For computation, the client incurs $O(\lambda\sqrt{N})$ hint searches, each costs $O(1)$ PRF evaluations and $O(1)$ membership test to check if a hint h_i contains the desired entry index. To refresh, it also takes the client $O(1)$ PRS invocation to generate a new PRF key. To recover the desired entry (and maintain hint structure), the client incurs $O(1)$ XOR operation on three B-bit entries. Thus, the client incurs $O(\lambda\sqrt{N})$ PRF evaluations and $O(1)$ XOR operations.

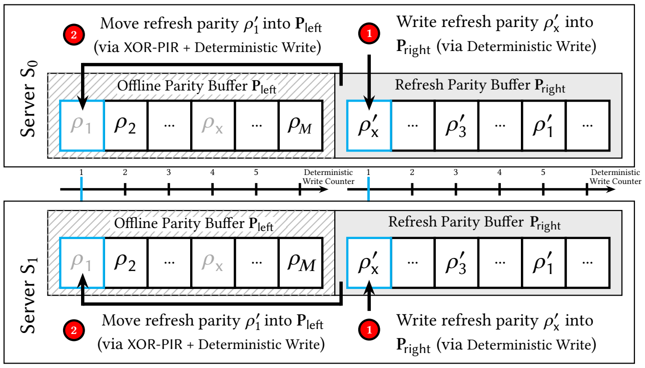

2
Move refresh parity ρ'_1 into $\mathbf{P}_{\text{left}}$
(via XOR-PIR + Deterministic Write)
1
Write refresh parity ρ'_x into
$\mathbf{P}_{\text{right}}$ (via Deterministic Write)
Server S_0
Offline Parity Buffer $\mathbf{P}_{\text{left}}$
ρ_1
ρ_2
...
ρ_x
...
ρ_M
Refresh Parity Buffer $\mathbf{P}_{\text{right}}$
ρ'_x
...
ρ'_3
...
ρ'_1
...
1
2
3
4
5
Deterministic Write Counter
1
2
3
4
5
Deterministic Write Counter
Server S_1
Offline Parity Buffer $\mathbf{P}_{\text{left}}$
ρ_1
ρ_2
...
ρ_x
...
ρ_M
Refresh Parity Buffer $\mathbf{P}_{\text{right}}$
ρ'_x
...
ρ'_3
...
ρ'_1
...
2
Move refresh parity ρ'_1 into $\mathbf{P}_{\text{left}}$
(via XOR-PIR + Deterministic Write)
1
Write refresh parity ρ'_x into
$\mathbf{P}_{\text{right}}$ (via Deterministic Write)

Figure 7: Remote oblivious refresh (w/ $c = 1$).

On the server side, each query (consisting of $O(\sqrt{N})$ indices) incurs $O(\sqrt{N})$ XOR operations on B-bit data entries to obtain two aggregated results. Thus, the server computation is $O(B\sqrt{N})$.

• Storage cost: The servers take no extra cost besides the $O(NB)$ DB storage. The client stores $O(\lambda\sqrt{N})$ precomputed hints, each contains a λ-bit PRF key, a server bit, an offline B-bit parity, and an auxiliary with $O(1)$ element. The client storage is $O(\lambda\sqrt{N}(\lambda + B))$.

5 Reducing Client Storage

Although our Pirex offers an efficient bandwidth overhead that is independent of the entry size, its client storage still depends on the entry size and thus, is significant. Specifically, the client storage cost is $O(\lambda\sqrt{N}(\lambda + B))$ because there are $O(\lambda\sqrt{N})$ hint entries, each contains a λ-bit PRF key and a B-bit parity. This cost may be significant for certain applications (e.g., mobile). In this section, we propose Pirex+, an extended Pirex scheme that provides an option for storage-limited clients to remotely store the parity components of the hints at the server, thereby reducing local storage overhead.

Remote parity storage. To reduce client storage, our idea is to maintain the offline parities (ρ_i) on the server and encrypt them with an IND-CPA encryption scheme to prevent the server from learning the private hint sets of indices from the parities in advance. As there are only $O(\lambda\sqrt{N})$ parities, the standard 2-server XOR-PIR can be used to privately read a desired parity in the online phase without much extra cost. Remark that the hint buffer needs to be refreshed per query so that the pseudorandom distribution of the sets is preserved. Given the parity parts are stored remotely, the refresh operation must be performed obliviously for security.

Oblivious refresh. To perform an oblivious refresh, we make use of oblivious write in [62]. Thus, we make the following changes to the data structures of the client and server in the Pirex scheme to support private remote parity maintenance:

- Server: Apart from maintaining DB as in Pirex, each server stores an additional replica of a $2M$-size parity buffer $\mathbf{P} = (\mathbf{P}_{\text{left}}, \mathbf{P}_{\text{right}})$. $\mathbf{P}_{\text{left}}$ is used to store M offline parities and $\mathbf{P}_{\text{right}}$ can temporarily store up to M refresh parities obtained in the online phase.
- Client: The client maintains a hint buffer $\mathbf{H} = (h_1, \ldots, h_M)$ as in Pirex. However, each hint $h_i = (\ell_i, \mathsf{sk}_i, Y_i, \pi_i)$ contains a new component $\pi_i \in [2M]$ denotes the location of the corresponding offline parity in the buffer $\mathbf{P}$ at the servers.

Let $\mathbf{H} = (h_1, \ldots, h_M)$ be the client hint buffer and $\mathbf{P}$ be the parity buffer the servers received after the offline phase. For each hint $h_i = (\ell_i, \mathsf{sk}_i, Y_i, \pi_i)$, the corresponding offline parity ρ_i is stored at $\mathbf{P}[\pi_i] = \mathbf{P}_{\text{left}}[i]$ for $i \in [M]$. In the online phase, suppose that a parity $\rho_i \in \mathbf{P}_{\text{left}}[i]$ corresponding to hint h_i is consumed to recover a desired data entry. Let ρ' be the new parity (line 6 Figure 6). To replace ρ_i with the new ρ', the idea is to perform two deterministic write operations on regions $(\mathbf{P}_{\text{left}}, \mathbf{P}_{\text{right}})$ per each refresh operation. Let c (mod M) be the refresh counter. At the c-th refresh, the client temporarily writes $\mathbf{P}_{\text{right}}[c] \leftarrow \rho'$ and stores its location $\pi_i := c+M$.

Due to the round-robin schedule, $\mathbf{P}_{\text{right}}$ will become full after M refreshes, making the next refresh overwrite some hints that were previously stored in $\mathbf{P}_{\text{right}}$. Thus, we let the client perform another deterministic write on $\mathbf{P}_{\text{left}}$ that obliviously transfers parities from $\mathbf{P}_{\text{right}}$ to $\mathbf{P}_{\text{left}}$. Specifically, at the c-th refresh, the client needs to write to $\mathbf{P}_{\text{left}}[c]$ a parity corresponds to hint $h_c = (\ell_c, \mathsf{sk}_c, Y_c, \pi_c)$. If $\pi_c = c$, it means the parity at $\mathbf{P}[\pi_c] = \mathbf{P}_{\text{left}}[c]$ was never refreshed in prior rounds. If $\pi_c \neq c$, the parity at $\mathbf{P}_{\text{left}}[c]$ was refreshed and it is now located at $\mathbf{P}[\pi_c] = \mathbf{P}_{\text{right}}[\pi_c - M]$. In either case, the client deterministically performs XOR-PIR to privately read the parity from $\mathbf{P}[\pi_c]$ and write it to $\mathbf{P}_{\text{left}}[c]$. To this end, the client updates the parity location of hint h_c to $\pi_c := c$.

The above strategy ensures every refresh parity located in $\mathbf{P}_{\text{right}}$ will be moved to $\mathbf{P}_{\text{left}}$ before it is overwritten. This is because it will take M additional refresh operations to revisit the same position in $\mathbf{P}_{\text{right}}$. By that time, all M positions in $\mathbf{P}_{\text{left}}$ will have been updated with the new parities. Thus, for any consumed parity $\rho_i \in \mathbf{P}_{\text{left}}[i]$, it will eventually be replaced by a new one ρ' after M rounds. Note that our scheme requires accessing two hints per online query. Thus, the client executes XOR-PIR and the oblivious write twice.

Supporting database update. In the real world, a public database can be updated. Although private database update is not captured in the PIR security, it is necessary to update the precomputed hints in OO-PIR to maintain the correctness. In Pirex+, since the parity buffer $\mathbf{P}$ is stored at the server, the update must be done obliviously. Otherwise, the server learns which parities are associated with the updated entry. After several updates, the server will learn the index distribution of each private hint set that constructed the parities, which violates the OO-PIR security that only holds if the hint sets are revealed *once* to each server in the online phase.

To privately update the parities according to database update, a simple method is to incorporate standard XOR-PIR and oblivious write similar to the oblivious refresh discussed above. Unlike the refresh operation which updates only a single parity, a database update requires multiple parities to be updated since each data entry contributes in $O(\lambda)$ offline hints. Thus, this method will incur high computation and communication costs (i.e., $O(B\lambda^2\sqrt{N})$ XOR operations and $O(B\lambda)$ bandwidth). To reduce this overhead, our solution is to incorporate Additive Homomorphic Encryption (AHE) so that the client can delegate oblivious update to the server.

Building block: Additive Homomorphic Encryption. AHE [30] permits the plaintexts to be encrypted such that their ciphertexts can be homomorphically evaluated. Given a cyclic group $\mathbb{G}$ of order p, an AHE scheme over $\mathbb{G}$ contains three PPT algorithms:

- $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{AHE.Gen}(1^\lambda)$: Given a security parameter λ, it outputs a pair of public and private keys $(\mathsf{pk}, \mathsf{sk})$.
- $\langle m \rangle \leftarrow \mathsf{AHE.Enc}(\mathsf{sk}, m)$: Given a message $m \in \mathbb{Z}_p$ and a public key pk, it outputs a ciphertext $\langle m \rangle$.
- $m \leftarrow \mathsf{AHE.Dec}(\mathsf{sk}, \langle m \rangle)$: Given a ciphertext $\langle m \rangle$ and the private key sk, it outputs a plaintext $m \in \mathbb{Z}_p$.

Let $\boxplus$ and $\boxdot$ be the group addition and scalar multiplication over the cyclic group $\mathbb{G}$. Given $m, m' \in \mathbb{Z}_p$, AHE offers the following additive homomorphic properties:

$$\mathsf{AHE.Enc}(\mathsf{pk}, m) \boxplus \mathsf{AHE.Enc}(\mathsf{pk}, m') = \mathsf{AHE.Enc}(\mathsf{pk}, m + m')$$
$$\mathsf{AHE.Enc}(\mathsf{pk}, m) \boxdot m' = \mathsf{AHE.Enc}(\mathsf{pk}, m \cdot m')$$

In Pirex+, we employ additive homomorphism in AHE to update the parities stored in the buffer $\mathbf{P}$ w.r.t database update as follows. Suppose the x-th entry $\mathsf{DB}[x]$ is being updated. As the buffer $\mathbf{P}$ is of size $2M$, the client first creates a binary vector $\mathbf{e} \in \{0, 1\}^{2M}$, where $\mathbf{e}[i] = 1$ if $\mathsf{DB}[x] \in \mathbf{P}[i]$, otherwise $\mathbf{e}[i] = 0$. The client encrypts vector $\mathbf{e}$ with the AHE public key pk as $\langle \mathbf{e} \rangle \leftarrow \mathsf{AHE.Enc}(\mathsf{pk}, \mathbf{e})$. Let $\langle \mathbf{p} \rangle = (\langle \rho_1 \rangle, \dots, \langle \rho_{2M} \rangle)$ be the vector of encrypted parities from the buffer, with $\langle \rho_i \rangle \leftarrow \mathsf{AHE.Enc}(\mathsf{pk}, \mathbf{P}[i])$. The client sends the encrypted vector $\langle \mathbf{e} \rangle$ to the server. Let b be the new data payload and $\epsilon = b - \mathsf{DB}[x]$. The server updates the parity buffer as follows:

$$\langle \mathbf{p}' \rangle := \langle \mathbf{p} \rangle \boxplus \langle \mathbf{e} \rangle \boxdot \epsilon$$

Since $\langle \mathbf{e} \rangle$ contains $2M$ group elements, the update bandwidth cost is $O(\lambda^2 \sqrt{N})$, independent of the entry size. The server computation includes $O(\lambda \sqrt{N})$ group additions and scalar multiplications.

Note that to fully incorporate AHE into Pirex+, we need to make necessary changes to the algebraic operations when computing the parities. Specifically, XOR operations are replaced with $\mathbb{Z}_p$ group additions. This is because the update ϵ is aggregated into a parity under homomorphic addition, where plaintexts are in $\mathbb{Z}_p$. Due to space constraints, we present the detailed algorithm of Pirex+ in Appendix C. Concretely, we instantiate Pirex+ with an efficient AHE scheme, e.g., Exponential ElGamal [30]. As Exponential ElGamal uses a discrete log solver for decryption, it only permits a small size of plaintext $m \in \mathbb{Z}_q$ with $q < p$ (e.g., $|q| = 32$ bits). This can be adapted by dividing a parity ρ_i into $|q|$-bit chunks and separately encrypting each chunk. Thus, each XOR operation (in Pirex) on B-bit entry is substituted with $\frac{B}{q}$ group additions of data chunks in $\mathbb{Z}_q$. To encrypt a B-bit parity, it now takes $\frac{B}{q}$ AHE encryption invocations. As each entry is now $|q|$-bit chunks, an update ϵ needs to be computed by $|q|$-bit chunks. Since the blocksize B is a fixed value in database settings, the number of chunks is always $\frac{B}{q}$. Each chunk's update will reside in $\mathbb{Z}_q$ with no overflow and is aggregated into a parity accordingly under homomorphic addition.

Reducing bandwidth impact of AHE ciphertext expansion. In Pirex+, although AHE permits the remote update of the encrypted parity buffer $\mathbf{P}$, its ciphertext expansion can incur the bandwidth overhead as the client accesses two AHE-encrypted parities per online query. To mitigate the bandwidth impact of AHE ciphertext expansion, we can slightly adjust the online query such that it only accesses one encrypted parity in the buffer. Specifically, to refresh the parity (ρ_i) that was used to recover $\mathsf{DB}[x]$, the client can directly sample a new random set $\mathcal{S}^*$ containing x and create additional online queries to obtain the punctured parity $\hat{\rho}$ of $\mathcal{S}^* \setminus \{x\}$, as similar to how we create and patch queries to recover $\mathsf{DB}[x]$. To this end, the client accesses the buffer $\mathbf{P}$ once to replace the consumed parity ρ_i with $\rho' = \hat{\rho} \oplus \mathsf{DB}[x]$. This strategy halves the bandwidth cost of the oblivious refresh, at the cost of making an extra online query to the servers (which is not impacted by the ciphertext expansion). We present the full algorithm of Pirex+ in Appendix C.

Complexity. We analyze the complexity of Pirex+ with constant chunk size q. For each offline hint, the server incurs $O(\sqrt{N})$ PRF evaluations and $O(\frac{B}{q}\sqrt{N})$ group additions (instead of XOR as in Pirex). The client invokes $O(\frac{B}{q}\lambda\sqrt{N})$ additional AHE encryptions to encrypt the parity buffer $\mathbf{P}$. In total, the client incurs $O(\lambda\sqrt{N})$ PRS invocations and $O(B\lambda\sqrt{N})$ AHE encryption. As $\mathbf{P}$ is maintained at the server, the client incurs $O(\lambda\sqrt{N}(\lambda + B))$ bandwidth to send the PRF keys and upload the parity buffer to the server.

In the online, each server incurs $O(\frac{B}{q}\sqrt{N})$ group additions per received set of indices. The server also performs $O(B\lambda\sqrt{N})$ XOR operations due to the 2-server XOR-PIR for offline parity retrieval. Meanwhile, the client executes $O(\lambda\sqrt{N})$ random bit generation, $O(\lambda\sqrt{N})$ PRF evaluations, and $O(\frac{B}{q})$ AHE decryptions plus $O(\frac{B}{q})$ group additions (instead of XOR as in Pirex) for data recovery. For client inbound bandwidth, as Pirex+ uses the above acceleration technique, it transmits eight aggregated entries to the client, along with eight additional encrypted parity for oblivious refresh (four received and sent), which is still $O(B)$ in total. The client can defer sending the new refreshed encrypted parities for oblivious write until executing the next online query, which results in one round for communication in total.

For storage, given the number of offline hints is $M = \lambda\sqrt{N}$, the client storage now contains only $O(\lambda\sqrt{N})$ PRF keys, since the client already offloads the parity part of the offline hints to the servers. Thus, the total client storage cost is $O(\lambda^2\sqrt{N})$. As a tradeoff, the server storage incurs an additional cost of $O(KB\lambda\sqrt{N})$ for storing the encrypted parities. Note that Pirex+ only provides this remote parity storage as an option for clients with limited storage.

For a database update, the client incurs an $O(\lambda^2\sqrt{N})$ outbound bandwidth for sending the AHE-encrypted binary vector. To update $\mathbf{P}$, the server performs $O(\frac{B}{q}\lambda\sqrt{N})$ scalar multiplications and group additions on $|q|$-bit parity chunks. The server cost is $O(B\lambda\sqrt{N})$.

Security. We state the security of Pirex+ as follows.

Theorem 3. Pirex+ *is secure by Definition 3.*

Proof. See Appendix §D.1. □

6 Experimental Evaluation

6.1 Implementation

We fully implemented our schemes in `rust`. We used libraries from `crates.io` to implement functionalities as follows: For PRF, we used `aes` crate with low-level AES-NI instruction for parallel block processing. We used `packed_simd` crate for XOR operations, which has SIMD instructions to load a 256-byte chunk onto the register per single XOR. For efficient memory access on server database and client storage, we used `memmap` to map all data files directly into OS memory. For client-server communication, we used `TcpStream` from `std::net` module. For Pirex+, we implemented exponential ElGamal using `libsecp256k1` [7], and adopted Shank's Baby-Step Giant-Step [64] for discrete log solver. Our code is publicly available at https://github.com/vt-asaplab/pirex.

6.2 Configuration

Hardware & network setting. For the client side, we used a 2023 MacBook Pro with M2 CPU @ 3.5 GHz, 32 GB RAM. For the server side, we created two virtual server instances on a Dell PowerEdge R750 with 48-core Intel Xeon 8360Y @ 2.4 GHz, 1 TB RAM. We only used a few physical cores for the virtual server process. To simulate average mobile LTE [4], we set client-server bandwidth to be around 40 Mbps with 11ms round-trip. For a comprehensive comparison, we also set up various bandwidth rates: 20 Mbps - 120 Mbps.

Database. To measure the performance, we used databases of sizes ranging from 1GB to 1TB, with three different entry sizes including 4 KB, 64 KB, and 256 KB. The number of entries N varies from 2^{12} to 2^{28} depending on the total size of the benchmarked database.

Counterpart comparison. We compared Pirex and Pirex+ with state-of-the-art OO-PIRs including CK20 [29], TreePIR [44], and Piano [72]. Note that the computation of all OO-PIR counterparts only involves PRF evaluations (for pseudorandom sets representation) and XOR-sum (for parity computation during online/offline queries). Therefore, for a fair comparison, we instantiated PRF in all OO-PIR counterparts with AES. We then applied the same hardware acceleration via AES-NI instructions to all schemes and measured their performance using the same hardware for both client and servers. For each scheme, we selected the parameters as follows:

- Pirex/Pirex+: We used 128-bit PRF keys. We set the number of hints $M = \sqrt{N} \log N$ for correctness (Theorem 1). For Pirex+, we used standard parameters for secp256k1 curve with 256-bit prime order and the base field $p = 2^{256} - 2^{32} - 977$. We divided each offline parity into 32-bit chunks for homomorphic encryption.
- CK20 [29]: We used 128-bit PRF keys for pseudorandom sets. We executed $\lambda = 128$ parallel instances of the protocol and set the number of hints $M = \sqrt{N} \log N$ for correctness.
- TreePIR [44]: We used 128-bit keys for puncturable PRF and set the number of hints $M = \sqrt{N} \log N$ for correctness.
- Piano [72]: We used 128-bit PRF keys. We set the number of primary hints $M_1 = \sqrt{N}(\ln(2)\kappa + \ln(Q))$, number of backup hints $M_2 = 3\sqrt{N} \ln N$ ($\kappa = 40$ is their statistical security parameter). Unlike TreePIR, CK20, or Pirex, Piano rebuilds their hints after $Q = \sqrt{N} \ln N$ queries. We measured their online performance with the amortized cost of rebuilding offline hints. We also compared with an extended Piano scheme [73] (denoted Piano$_{\text{ext}}$) that trades additional client storage for reduced online cost.

6.3 Results

Online bandwidth. Figure 8 reports the client online bandwidth overhead of our schemes compared with other works. Pirex achieves the lowest bandwidth among all, where it is around 120×-910× times smaller than Piano and TreePIR when performing on a 1 TB database, depending on the chosen entry sizes. This is because our online inbound bandwidth cost is independent of the number of entries. The client only downloads a constant of four parities, which is equivalent to 16/256/1024 KB to access a 4/64/256 KB entry, respectively. Meanwhile, Piano (*resp.* TreePIR) requires $O(\sqrt{N})$ parities to be downloaded, which takes from 35 MB (*resp.* 131 MB) to 281 MB (*resp.* 1 GB) of online bandwidth for a 1 TB database setting depending on the entry size. Thus, comparing to Piano and TreePIR in large database settings (2^{18} to 2^{24} entries of 64 KB, for example), Pirex saves the client more than 98% of the online bandwidth. For Piano$_{\text{ext}}$, the bandwidth cost of Pirex is still 24×-54× smaller on 1 TB settings. This is because Piano$_{\text{ext}}$ still requires an amortized cost of $O(\sqrt{N}/\log N)$ entries to be downloaded per online query. For real practical databases with billions of entries, the gap between Pirex and Piano/TreePIR will be at least three orders of magnitude. In CK20, although the client downloads a constant of 256 parities per online query (due to 128 online instances executing in parallel), its concrete cost is at least 65× larger than Pirex in all test cases. For DB with large entry sizes (2^{18} to 2^{24} entries of 64KB, and more), Pirex reduces around 96% of the client's inbound bandwidth.

Pirex+ incurs slightly higher bandwidth than Pirex. This is because the client needs to privately read two offline parities with XOR-PIR (i.e., one for online access, one for preventing buffer overflow) and rewrite a refresh parity to the parity buffer **P**. Compared with Pirex, Pirex+ requires transmitting eight extra parities. Similar to Pirex, the inbound bandwidth cost of Pirex+ is independent of the number of entries, and thus, is lower than other schemes on increasing database sizes. Figure 9a reports the online bandwidth cost of all schemes with varied entry sizes from 4 KB to 256 KB on a database of 2^{24} entries. Figure 9b further reports the network delay of all schemes on varied bandwidth rates from 20 Mbps to 120 Mbps, with a database of 2^{24} entries of 64 KB.

Online end-to-end delay. Figure 10 showed the concrete end-to-end delay of Pirex/Pirex+ compared to CK20, TreePIR, Piano and Piano$_{\text{ext}}$. Pirex incurs a minimal delay on varied database and entry sizes. Specifically, Pirex takes only 55ms to retrieve a 4 KB entry from a DB with 2^{28} records. This is around 165×, 565×, and 728× faster than Piano, TreePIR, and CK20, respectively, taking from 9s (Piano) to more than 30s (TreePIR, CK20). For 1 TB DB with large entry sizes such as 256 KB, Pirex only takes 260ms of end-to-end delay. This is around 191×, 845×, and 127× faster than Piano, TreePIR, and CK20, respectively, which takes more than 30s. For Piano$_{\text{ext}}$, although the high delay is reduced, it still takes more than 6s. The high delay in Piano and TreePIR is mainly due to retrieving $O(\sqrt{N})$ entries (compared with $O(1)$ in Pirex/Pirex+). Meanwhile, CK20 requires executing 128 protocol instances in parallel, which incurs high bandwidth and computation at both client and server. For example, with a 1 TB database of 2^{22} 256 KB entries, our client and server computation is merely 7.5ms and 31ms, respectively. CK20 takes more than 1s of client times and 30s of server times. On average, the client and server computation in Pirex is respectively 100×-150× and 80×-120× faster than CK20.

The end-to-end delay of Pirex+ is at most 17× higher than Pirex, yet it is 10×, 55×, 16×, 2× (e.g. on 64 KB entry size database with 2^{24} entries) lower than CK20, TreePIR, Piano, and Piano$_{\text{ext}}$ respectively. For increasing database sizes, the gap between Pirex+ and TreePIR/Piano (or Piano$_{\text{ext}}$) will be more significant. Note that the differences in delay between Pirex+ and Pirex are due to the extra

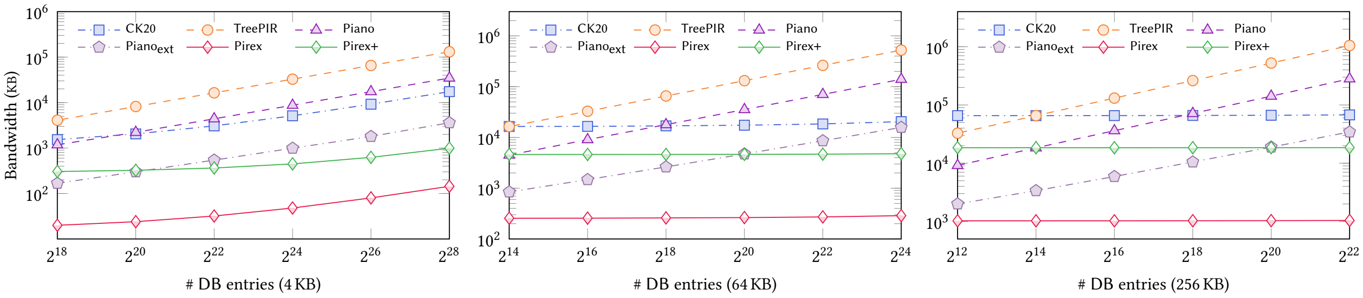
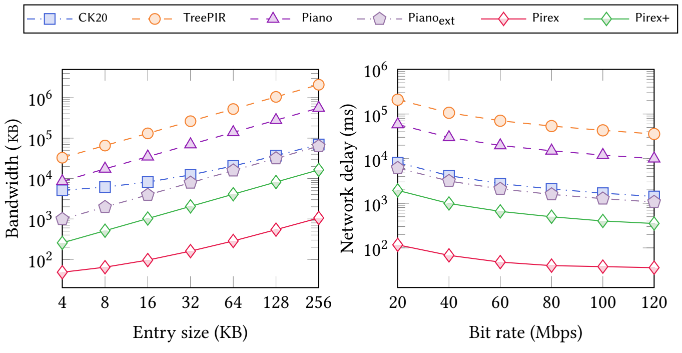


Figure 8: Client (amortized) online bandwidth cost.



(a) (w/ 2^{24} entries) **(b)** (w/ 2^{24} 64 KB entries)

Figure 9: (a) Online bandwidth / (b) Network delay.

operations, which include retrieving the offline parities and decrypting them by solving discrete logs. As each parity is chunked into 32 bits for encryption, Pirex+ takes under 426ms for decrypting a 256 KB parity (see cost breakdown below). Thus, the difference mostly stems from the extra four encrypted parities being downloaded (using XOR-PIR) and the XOR computation that servers perform on $O(N \log N)$ encrypted parities.

Cost breakdown. We dissect the end-to-end delay of Pirex and Pirex+ to investigate which performance factors impact the most.

- Pirex: Figure 11 presents the detailed cost of Pirex from 1 GB to 1 TB DB with 4 KB, 64 KB, and 256 KB entry sizes, respectively. The three main factors contributing to the delay of Pirex are the client computation, the server processing, and the communication latency. The client overhead in Pirex is efficient, taking up merely 10 ms, thus, only contributing 1%-18% to the total delay. The client performs three main operations: (1) looking up a hint, (2) creating an online query, and (3) recovering the desired entry. Looking up a hint is fast as the membership testing incurs only one PRF evaluation for each PRF key, and there are $\sqrt{N} \log N$ keys in total. Recovering the entry (and refreshing hint) only incurs XOR operations on four parities responded from the server.

The server processing in Pirex is also efficient. For 100 GB DB, the cost is smaller then 7ms and hence, is hard to observe in Figure 11. For larger DB (2^{24} entries of 64 KB and 2^{22} entries of 256 KB), Pirex only takes 30ms and 60ms, respectively, contributing up to 20% in average of the online delay. The cost mainly stems from performing XOR operations on $\sqrt{N}$ data entries. In Pirex, each query contains one patched set and one partition set with at most $\sqrt{N}$ indices. Thus, the amount of XOR operations per server is sublinear to the number of entries and linear to the entry size. For DB with 2^{28} entries of 4KB, Pirex takes only 7ms for end-to-end server computation.

Communication is the most dominating factor in the delay. As Pirex features constant bandwidth, this latency remains constant for each setting of entry sizes. Under the 40 Mbps bandwidth rate, Pirex takes around 55ms and 225ms to get four parities of 64 KB and 256 KB, respectively. For the databases with smaller entry sizes (4 KB), the query size can outweigh the parity transmission size. With 2^{28} entries, the client needs to send 2^{14} offsets to get a parity, where the concrete size of the offsets is approximately 30 KB. Thus, the communication latency incurs from 10ms to 60ms (respectively for databases with 2^{18} to 2^{28} entries).

- Pirex+: Figure 12 illustrates the detailed cost of Pirex+. Unlike Pirex, the client cost in Pirex+ is noticeable for large entry sizes (64 KB or 256 KB). This cost is mostly attributed to the re-encryption of two offline parities, which takes about 120ms for 64 KB parities and 420ms for 256 KB parities. The server processing cost is the most dominating factor for entry sizes of 4 KB and 64 KB, which takes from 55ms to 1.6s and attribute from 20% to 70% of the total delay, due to the extra oblivious refresh on the parity buffer that two servers perform. The process involves XOR operations on the encrypted parity buffer of size $\sqrt{N} \log N$, where each entry is 16× larger than a database entry size due to AHE ciphertext expansion. Thus, the amount of data to be processed is $16 \log N \times$ larger than Pirex. However, this gap is constant as shown by a growth with a small slope. For DB of 256 KB entry, the communication delay outweighs the server processing cost since it takes over 2s to download four extra encrypted parities.

Mobile client computation. To assess the performance of Pirex+ under resource-limited clients, we conducted an experiment using a 2021 iPad Pro with M1 CPU @ 8 GB RAM, and 128 GB storage as the client. Under the database with 2^{22} entries of size 256 KB, the mobile client only incurs an extra 446ms delay over the original resourceful client. Such overhead mostly stems from executing the discrete log solver on the mobile CPU.

Storage. We report the client storage of all schemes in Figure 14. Pirex+ permits extremely low client storage compared with others. Concretely, for DB of 2^{22} 256 KB entries, Pirex+ incurs four to six orders of magnitudes smaller client storage than CK20, TreePIR, and Piano/Piano$_{ext}$. The client in Pirex+ only stores 710 KB of PRF

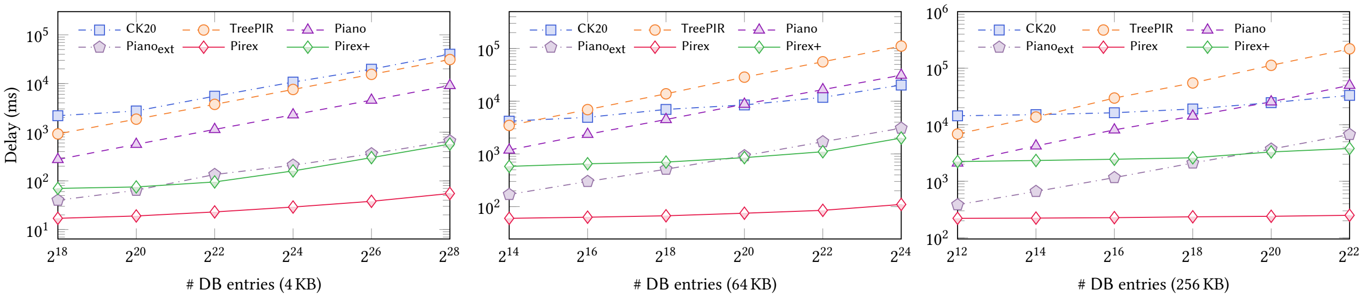


Figure 10: Client (amortized) online end-to-end delay.

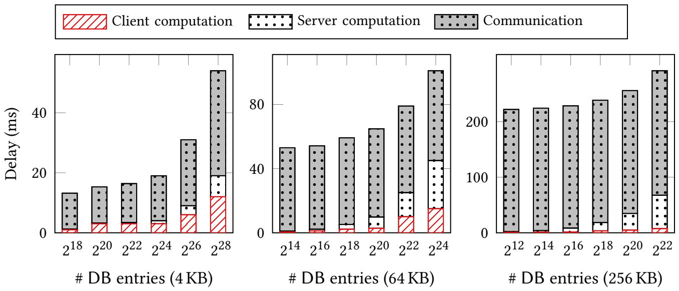


Figure 11: Cost breakdown of Pirex **(online phase).**

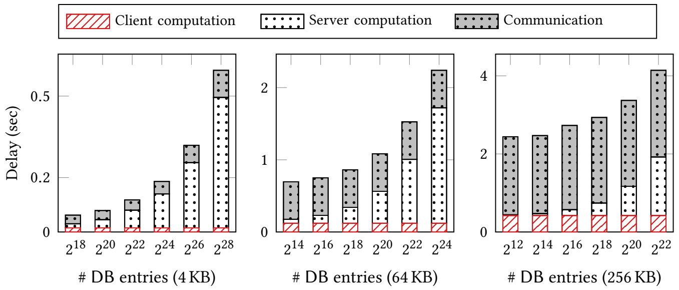


Figure 12: Cost breakdown of Pirex+ **(online phase).**

keys compared with 11 GB (in Pirex, TreePIR), 110 GB (in Piano, Piano$_{ext}$) and 1.3 TB (in CK20) due to the offline parities. In Pirex+, the extra server storage for the encrypted parities per client is 185 GB for 1 TB DB, due to the ciphertext expansion and oblivious write buffer. Note that Pirex+ provides this remote parity storage as an option, which is mostly desirable for limited-memory devices. As server storage is cheap [3, 6] and continually decreasing, the monthly cost to store these parities remotely is only $3.5-4.5.

Offline cost. Figure 13 reports the offline cost of Pirex/Pirex+ with other counterparts. Pirex features a comparable overhead to TreePIR, taking from 7s to 2600s to preprocess up to 1 TB DB (with varied entry sizes). Piano requires entire database streaming to compute the offline hints and, therefore, its offline delay is 78×-571× higher than Pirex and TreePIR. CK20 requires 128 instances in parallel so its preprocessing is 128× slower than Pirex and TreePIR. On the other hand, Pirex+ incurs 20×-30× higher offline delay than Pirex. This gap is mainly due to the AHE encryption and the network delay when sending encrypted offline parities to the server.

Database update. We report the server cost for Pirex+ to privately update the parity buffer per database entry update. Pirex+ takes from 4ms to 3s to update each entry chunk in databases with 2^{12} to 2^{28} entries. In other schemes (e.g., Pirex, Piano, TreePIR), the client stores the offline parities and thus, the update cost is negligible.

7 Related Work

Standard PIR. Chor et al. were the first to introduce PIR [26]. Their standard 2-server XOR-PIR achieves information-theoretic security with $O(N)$ bandwidth cost. To reduce the bandwidth cost to $O(N^{\frac{1}{3}})$, they proposed a variant based on covering codes. To enable single-server, Kushilevitz et al. [42] proposed an AHE-based PIR scheme with computational security and achieves $O(N^{\epsilon})$ bandwidth ($\epsilon > 0$). While later refinements reduced the bandwidth to sublinear [23, 25, 34, 47], Sion et al. [66] showed that evaluating AHE is more expensive than streaming the database itself. To reduce computation overhead, some lattice-based PIR schemes were proposed [14, 15, 17, 31, 39, 49, 50, 53]. These schemes, however, cannot surpass the $\Omega(N)$ computation lower bound [21] in the standard PIR model.

Global preprocessing PIR. Beimel et al. [21] showed that by preprocessing an $O(N)$-sized database to an encoded form of size $O(N^{3.2})$, the server time and communication cost in a 2-server PIR can be reduced to $O(N^{0.6})$. Several single-server PIR schemes were designed based on secretly permuted Reed-Muller codes [22, 24] which require superlinear server storage to store the encoded database per designated group of clients that holds a secret key. Boyle et al. [22] showed how to upgrade the secret-key scheme to a public-key variant using ideal obfuscation, where the key can be used by any client to execute the retrieval protocol. All these schemes do not rely on known standard assumptions. Lin et al. [46] thus presented a scheme based on standard Ring-LWE, where the server time and communication cost are polylogarithmic.

Client preprocessing PIR. Patel et al. [58] proposed PSIR, an OO-PIR model uses precomputed offline hint to achieve online queries with only linear PRF and sublinear public-key operations. Corrigan-Gibbs et al. [29] then proposed a two-server OO-PIR with $\tilde{O}(\lambda\sqrt{N})$ online server cost, and a single-server variant [28] that supports $\sqrt{N}$ queries with $\tilde{O}(\sqrt{N})$ bandwidth and $O(N^{3/4})$ server time using linearly HE. To reduce client query bandwidth to polylog(N), other works leveraged privately puncturable/programmable PRF [43, 65, 71]. The main bottleneck in these schemes is the λ parallel protocol instance executions for correctness. Kogan et al. [41] showed a trick

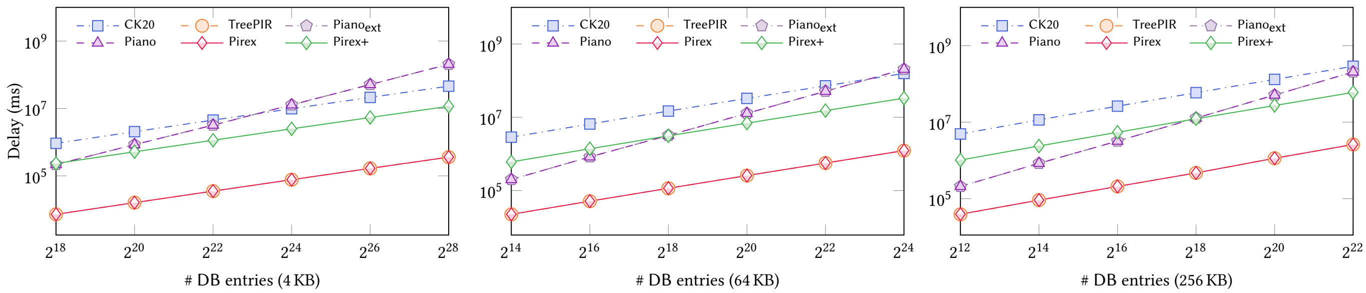


Figure 13: Client offline end-to-end delay.

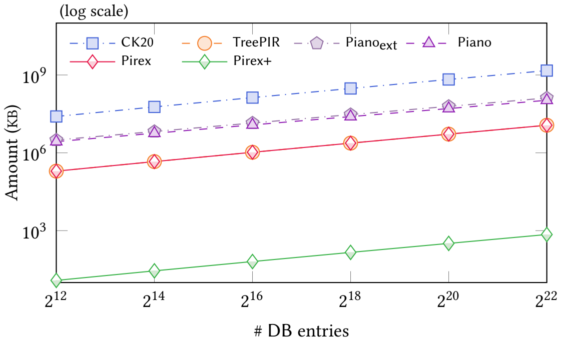


Figure 14: Client storage cost ($B = 256$ KB).

to remove the λ repetitions but requires $O(N)$ storage or $O(N)$ online time from the client for using non-private puncturable PRF. Lazzaretti et al. [44] suggested a novel partitioned OO-PIR with polylog(N) query size in $O(\lambda\sqrt{N})$ client time, but incurs $O(\sqrt{N})$ parities to be transmitted. Zhou et al. [72] proposed Piano, which adapts the scheme [44] to a single server setting. To create offline hints without needing a second server, Piano requires full database streaming per $O(\sqrt{N})$ queries for hint rebuild. To achieve a constant online inbound bandwidth comparable to our scheme, Piano makes use of the database streaming process to prepare upfront a patching element per hint. Their amortized inbound bandwidth is $O(\sqrt{N})$. In contrast, our schemes use PPR to efficiently and privately obtain a random online patching element with no preprocessing costs, by utilizing two non-colluding servers for PPR execution. Lazzaretti et al. [45] further reduces the hint size in [44] by a λ factor, yielding smaller preprocessing overhead. Ren et al. [61] recently proposed a concurrent and independent work that also concretely incurs four entries on the client's inbound bandwidth, similar to Pirex. Their scheme randomly divides each original OO-PIR hint set into two equal subsets, which requires a double hint size and a probabilistic median selection value per hint for efficient online hint searching.

Batched PIR. Pioneered by [40], batched PIR permits the server to process a batch of Q queries at a time. Using batch codes, the server time is linear to the number of codewords but will be smaller than executing a PIR protocol Q times. As the number of buckets in existing batch codes [18, 40, 60, 67] incurs a significant response overhead, Angel et al. [17] proposed a method that costs $O(N)$ server time for a large batch of size Q but incurs only $O(Q)$ ciphertext responses. Mughees et al. [54] later proposed a vectorized batch PIR that can fit as many database entries as a single ciphertext can hold. Some batched PIR schemes [21, 48] support multiple clients using efficient matrix multiplication techniques [27, 68].

8 Conclusion

We proposed Pirex, a new OO-PIR framework for large databases that incurs minimal client inbound bandwidth and storage overhead, while retaining a sublinear processing cost for the client and servers. Pirex offers constant client's inbound bandwidth regardless of the number of entries in the public database. It also offers clients the flexibility to securely store and access preprocessing hints remotely if necessary. This alleviates client storage requirements, and therefore, is beneficial to resource-limited clients (e.g., mobile).

Acknowledgments

The authors thank the revision editor and anonymous reviewers for their insightful comments and constructive feedback on improving the quality of this work. This work was supported in part by an unrestricted gift from Robert Bosch, 4-VA, and the Commonwealth Cyber Initiative (CCI), an investment in the advancement of cyber R&D, innovation, and workforce development. For more information about CCI, visit www.cyberinitiative.org.

References

[1] 01. 2024. Amazon DynamoDB - Item Size Limits. https://aws.amazon.com/dynamodb/faqs/.
[2] 01. 2024. Amazon Elastic Block Store - Block Size. https://docs.aws.amazon.com/ebs/latest/userguide/volume_constraints.html.
[3] 01. 2024. Amazon S3 Pricing. https://aws.amazon.com/s3/pricing/.
[4] 01. 2024. Average Mobile Network Speed. https://www.statista.com/statistics/896779/average-mobile-fixed-broadband-download-upload-speeds/.
[5] 01. 2024. Azure Blob Storage - Blob Size. https://learn.microsoft.com/en-us/azure/storage/blobs/scalability-targets.
[6] 01. 2024. Azure Blob Storage Pricing. https://azure.microsoft.com/en-us/pricing/details/storage/blobs/.
[7] 01. 2024. Bitcoin Core Secp256k1. https://github.com/bitcoin-core/secp256k1.
[8] 01. 2024. MongoDB Manual - Document Size Limit. https://www.mongodb.com/docs/manual/core/document/.
[9] 01. 2024. MySQL Limits - Row Size Limits. https://dev.mysql.com/doc/refman/8.4/en/innodb-file-space.html.
[10] 01. 2024. PostgreSQL - Database Page Layout. https://www.postgresql.org/docs/current/storage-page-layout.html.
[11] 01. 2024. Public Centers for Disease Control and Prevention. https://data.cdc.gov/.
[12] 01. 2024. Public Federal Reserve Economic Data. https://fred.stlouisfed.org/.
[13] 01. 2024. Public Inter-University Consortium for Political and Social Research. https://www.icpsr.umich.edu/.

[14] Ishtiyaque Ahmad, Yuntian Yang, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta. 2021. Addra: Metadata-Private Voice Communication Over Fully Untrusted Infrastructure. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*.

[15] Asra Ali, Tancrède Lepoint, Sarvar Patel, Mariana Raykova, Phillipp Schoppmann, Karn Seth, and Kevin Yeo. 2021. Communication-Computation Trade-offs in PIR. In *30th USENIX Security Symposium (USENIX Security 21)*. 1811–1828.

[16] Andris Ambainis. 1997. Upper Bound on The Communication Complexity of Private Information Retrieval. In *International Colloquium on Automata, Languages, and Programming*. Springer, 401–407.

[17] Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty. 2018. PIR with Compressed Queries and Amortized Query Processing. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 962–979.

[18] Sebastian Angel and Srinath Setty. 2016. Unobservable Communication Over Fully Untrusted Infrastructure. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 551–569.

[19] Amos Beimel and Yuval Ishai. 2001. Information-Theoretic Private Information Retrieval: A Unified Construction. In *Automata, Languages and Programming: 28th International Colloquium, ICALP 2001 Crete, Greece, July 8–12, 2001 Proceedings 28*. Springer, 912–926.

[20] Amos Beimel, Yuval Ishai, Eyal Kushilevitz, and J-F Raymond. 2002. Breaking The $O(n^{1/(2k-1)})$ Barrier for Information-theoretic Private Information Retrieval. In *Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002*. IEEE, 261–270.

[21] Amos Beimel, Yuval Ishai, and Tal Malkin. 2000. Reducing The Servers Computation In Private Information Retrieval: PIR with Preprocessing. In *Advances in Cryptology—CRYPTO 2000: 20th Annual International Cryptology Conference Santa Barbara, California, USA, August 20–24, 2000 Proceedings 20*. Springer, 55–73.

[22] Elette Boyle, Yuval Ishai, Rafael Pass, and Mary Wootters. 2017. Can We Access a Database both Locally and Privately?. In *Theory of Cryptography: 15th International Conference, TCC 2017, Baltimore, MD, USA, November 12-15, 2017, Proceedings, Part II 15*. Springer, 662–693.

[23] Christian Cachin, Silvio Micali, and Markus Stadler. 1999. Computationally Private Information Retrieval with Polylogarithmic Communication. In *Advances in Cryptology—EUROCRYPT'99: International Conference on the Theory and Application of Cryptographic Techniques Prague, Czech Republic, May 2–6, 1999 Proceedings 18*. Springer, 402–414.

[24] Ran Canetti, Justin Holmgren, and Silas Richelson. 2017. Towards Doubly Efficient Private Information Retrieval. In *Theory of Cryptography: 15th International Conference, TCC 2017, Baltimore, MD, USA, November 12-15, 2017, Proceedings, Part II 15*. Springer, 694–726.

[25] Yan-Cheng Chang. 2004. Single Database Private Information Retrieval with Logarithmic Communication. In *Information Security and Privacy: 9th Australasian Conference, ACISP 2004, Sydney, Australia, July 13-15, 2004. Proceedings 9*. Springer, 50–61.

[26] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. 1998. Private Information Retrieval. *Journal of the ACM (JACM)* 45, 6 (1998), 965–981.

[27] Don Coppersmith and Shmuel Winograd. 1987. Matrix Multiplication via Arithmetic Progressions. In *Proceedings of The Nineteenth Annual ACM Symposium on Theory of Computing*. 1–6.

[28] Henry Corrigan-Gibbs, Alexandra Henzinger, and Dmitry Kogan. 2022. Single-Server Private Information Retrieval with Sublinear Amortized Time. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 3–33.

[29] Henry Corrigan-Gibbs and Dmitry Kogan. 2020. Private Information Retrieval with Sublinear Online Time. In *Advances in Cryptology–EUROCRYPT 2020: 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10–14, 2020, Proceedings, Part I 39*. Springer, 44–75.

[30] Ronald Cramer, Rosario Gennaro, and Berry Schoenmakers. 1997. A Secure and Optimally Efficient Multi-Authority Election Scheme. *European Transactions on Telecommunications* 8, 5 (1997), 481–490.

[31] Alex Davidson, Gonçalo Pestana, and Sofía Celi. 2023. FrodoPIR: Simple, Scalable, Single-Server Private Information Retrieval. *Proceedings on Privacy Enhancing Technologies* 1 (2023), 365–383.

[32] Zeev Dvir and Sivakanth Gopi. 2016. 2-Server PIR With Subpolynomial Communication. *Journal of the ACM (JACM)* 63, 4 (2016), 1–15.

[33] Klim Efremenko. 2009. 3-Query Locally Decodable Codes of Subexponential Length. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing*. 39–44.

[34] Craig Gentry and Zulfikar Ramzan. 2005. Single-Database Private Information Retrieval with Constant Communication Rate. In *International Colloquium on Automata, Languages, and Programming*. Springer, 803–815.

[35] Niv Gilboa and Yuval Ishai. 2014. Distributed Point Functions and Their Applications. In *Advances in Cryptology–EUROCRYPT 2014: 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings 33*. Springer, 640–658.

[36] Matthew Green, Watson Ladd, and Ian Miers. 2016. A Protocol For Privately Reporting Ad Impressions At Scale. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 1591–1601.

[37] Trinabh Gupta, Natacha Crooks, Whitney Mulhern, Srinath Setty, Lorenzo Alvisi, and Michael Walfish. 2016. Scalable and Private Media Consumption with Popcorn. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. 91–107.

[38] Aniko Hannak, Gary Soeller, David Lazer, Alan Mislove, and Christo Wilson. 2014. Measuring Price Discrimination and Steering on E-Commerce Web Sites. In *Proceedings of the 2014 Conference on Internet Measurement Conference*. 305–318.

[39] Alexandra Henzinger, Matthew M Hong, Henry Corrigan-Gibbs, Sarah Meiklejohn, and Vinod Vaikuntanathan. 2023. One Server For The Price of Two: Simple and Fast Single-Server Private Information Petrieval. In *Usenix Security*, Vol. 23.

[40] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. 2004. Batch Codes and Their Applications. In *Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing*. 262–271.

[41] Dmitry Kogan and Henry Corrigan-Gibbs. 2021. Private Blocklist Lookups With Checklist. In *30th USENIX Security Symposium*. 875–892.

[42] Eyal Kushilevitz and Rafail Ostrovsky. 1997. Replication Is Not Needed: Single Database, Computationally-Private Information Retrieval. In *Proceedings 38th Annual Symposium on Foundations of Computer Science*. IEEE, 364–373.

[43] Arthur Lazzaretti and Charalampos Papamanthou. 2023. Near-Optimal Private Information Retrieval with Preprocessing. In *Theory of Cryptography Conference*. Springer, 406–435.

[44] Arthur Lazzaretti and Charalampos Papamanthou. 2023. TreePIR: Sublinear-Time and Polylog-Bandwidth Private Information Retrieval from DDH. In *Advances in Cryptology – CRYPTO 2023*, Helena Handschuh and Anna Lysyanskaya (Eds.). 284–314.

[45] Arthur Lazzaretti and Charalampos Papamanthou. 2024. Single-Pass Client Preprocess Private Information Retrieval. In *33rd USENIX Security Symposium*. 5967–5984.

[46] Wei-Kai Lin, Ethan Mook, and Daniel Wichs. 2023. Doubly Efficient Private Information Retrieval and Fully Homomorphic RAM Computation From Ring LWE. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing*. 595–608.

[47] Helger Lipmaa. 2005. An Oblivious Transfer Protocol with Log-Squared Communication. In *Information Security: 8th International Conference, ISC 2005, Singapore, September 20-23, 2005. Proceedings 8*. Springer, 314–328.

[48] Wouter Lueks and Ian Goldberg. 2015. Sublinear Scaling for Multi-Client Private Information Retrieval. In *Financial Cryptography and Data Security: 19th International Conference, FC 2015, San Juan, Puerto Rico, January 26-30, 2015, Revised Selected Papers 19*. Springer, 168–186.

[49] Carlos Aguilar Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. 2016. XPIR: Private Information Retrieval for Everyone. *Proceedings on Privacy Enhancing Technologies* (2016), 155–174.

[50] Samir Jordan Menon and David J Wu. 2022. Spiral: Fast, High-Rate Single-Server PIR via FHE Composition. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 930–947.

[51] Jakub Mikians, László Gyarmati, Vijay Erramilli, and Nikolaos Laoutaris. 2012. Detecting Price and Search Discrimination on The Internet. In *Proceedings of the 11th ACM workshop on hot topics in networks*. 79–84.

[52] Michael Mitzenmacher and Eli Upfal. 2017. *Probability and Computing: Randomization and Probabilistic Techniques in Algorithms and Data Analysis.*

[53] Muhammad Haris Mughees, Hao Chen, and Ling Ren. 2021. OnionPIR: Response Efficient Single-Server PIR. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2292–2306.

[54] Muhammad Haris Mughees and Ling Ren. 2023. Vectorized Batch Private Information Retrieval. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 437–452.

[55] Arvind Narayanan and Vitaly Shmatikov. 2008. Robust De-Anonymization of Large Sparse Datasets. In *2008 IEEE Symposium on Security and Privacy (SP 2008)*. IEEE, 111–125.

[56] Arvind Narayanan and Vitaly Shmatikov. 2010. Myths and Fallacies of Personally Identifiable Fnformation. *Commun. ACM* 53, 6 (2010), 24–26.

[57] Andrew Odlyzko. 2003. Privacy, Economics, and Price Discrimination on The Internet. In *Proceedings of the 5th international conference on Electronic commerce*. 355–366.

[58] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. 2018. Private Stateful Information Retrieval. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1002–1019.

[59] Raghu Ramakrishnan, Baskar Sridharan, John R Douceur, Pavan Kasturi, Balaji Krishnamachari-Sampath, Karthick Krishnamoorthy, Peng Li, Mitica Manu, Spiro Michaylov, Rogério Ramos, et al. 2017. Azure Data Lake Store: A Hyperscale Distributed File Service for Big Data Analytics. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 51–63.

[60] Ankit Singh Rawat, Zhao Song, Alexandros G Dimakis, and Anna Gál. 2016. Batch Codes Through Dense Graphs Without Short Cycles. *IEEE Transactions on Information Theory* 62, 4 (2016), 1592–1604.

[61] Ling Ren, Muhammad Haris Mughees, and I Sun. 2024. Simple and Practical Amortized Sublinear Private Information Retrieval using Dummy Subsets. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security.*

[62] Daniel S Roche, Adam Aviv, Seung Geol Choi, and Travis Mayberry. 2017. Deterministic, Stash-Free Write-Only ORAM. In *Proceedings of The 2017 ACM SIGSAC Conference on Computer and Communications Security.* 507–521.

[63] Sacha Servan-Schreiber, Kyle Hogan, and Srinivas Devadas. 2021. AdVeil: A Private Targeted Advertising Ecosystem. *Cryptology ePrint Archive* (2021).

[64] Daniel Shanks. 1971. Class Number, A Theory of Factorization, and Genera. In *Proc. Symp. Math. Soc., 1971*, Vol. 20. 415–440.

[65] Elaine Shi, Waqar Aqeel, Balakrishnan Chandrasekaran, and Bruce Maggs. 2021. Puncturable Pseudorandom Sets and Private Information Retrieval with Near-Optimal Online Bandwidth and Time. In *Proceedings of the 41st Annual International Cryptology Conference.* Springer, 641–669.

[66] Radu Sion and Bogdan Carbunar. 2007. On The Computational Practicality of Private Information Retrieval. In *Proceedings of The Network and Distributed Systems Security Symposium.* Internet Society Geneva, Switzerland, 2006–06.

[67] Douglas R Stinson, Ruizhong Wei, and Maura B Paterson. 2009. Combinatorial Batch Codes. *Advances in Mathematics of Communications* 3, 1 (2009), 13–27.

[68] Volker Strassen et al. 1969. Gaussian Elimination Is Not Optimal. *Numerische Mathematik* 13, 4 (1969), 354–356.

[69] Thomas Vissers, Nick Nikiforakis, Nataliia Bielova, and Wouter Joosen. 2014. Crying Wolf? On The Price Discrimination of Online Airline Tickets. In *7th Workshop on Hot Topics in Privacy Enhancing Technologies (HotPETs 2014).*

[70] Sergey Yekhanin. 2008. Towards 3-Query Locally Decodable Codes of Subexponential Length. *Journal of the ACM (JACM)* 55, 1 (2008), 1–16.

[71] Mingxun Zhou, Wei-Kai Lin, Yiannis Tselekounis, and Elaine Shi. 2023. Optimal Single-Server Private Information Retrieval. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques.* 395–425.

[72] M. Zhou, A. Park, W. Zheng, and E. Shi. 2024. PIANO: Extremely Simple, Single-Server PIR with Sublinear Server Computation. In *2024 IEEE Symposium on Security and Privacy.* 55–55.

[73] 2024. PIANO: Extremely Simple, Single-Server PIR with Sublinear Server Computation. Cryptology ePrint Archive, Paper 2023/452.

A PPR Deferred Proofs

A.1 Security (Lemma 1)

Proof. We construct a simulator $\mathcal{S}_P$ such that no PPT environment $\mathcal{Z}$ can distinguish between its view in the Ideal and Real. $\mathcal{Z}$ can statically corrupt one server to view the execution transcript. On receiving the notification from the ideal functionality $\mathcal{F}_P$, the simulator $\mathcal{S}_P$ functions as follows:

(1) $\mathcal{S}_P$ samples $(\delta_0, ..., \delta_{n-1}) \xleftarrow{\$} [m]^n$, $q \xleftarrow{\$} \{0,1\}^n$

(2) $\mathcal{S}_P$ outputs $\mathcal{T} \leftarrow \{q[i] \cdot (i \cdot m + \delta_i)\ \forall\ i \in [n] \wedge q[i] \neq 0\}$

In Ideal, the simulator $\mathcal{S}_P$ randomly samples a selection bit string to simulate a list of partition accesses (to arbitrary indices). For the PPR protocol (Figure 2) described in Real, the environment $\mathcal{Z}$ can infer the bit selection when viewing the partition query $\mathcal{T}_0$ (or $\mathcal{T}_1$), since each index belongs to a partition. Since bit flipping does not distort the distribution of random bit string, the partition access (by using bit selection), is uniformly random for both execution under the view of $\mathcal{Z}$. Note that for security, privately accessing a partition does not require the returned data item to be located at a random index. It is rather a functionality that we want to achieve. □

B Pirex Deferred Proofs

B.1 Auxiliary Size (Lemma 2)

Proof. Let $\mathcal{H} = (h_1, \ldots, h_M)$ be the generated hint buffer of size $M = O(\sqrt{N} \log N)$. For every hint $h_i = (\ell_i, \mathsf{sk}_i, \rho_i, Y_i) \in \mathcal{H}$, let X_i be a random variable that indicates the size of auxiliary Y_i. We present the operations that affects the state of X_i at each protocol execution time t as follows:

- In the offline: $X_i^{(t)} = X_i^{(0)} = 0$. Thus, $\mathbb{E}[X_i^{(0)}] = 0$.
- In the online:
  (1) If h_i is consumed and replaced with a new hint h_i':
  $$X_i^{(t+1)} = X_i^{(t)} - X_i^{(t)} = 0$$
  (2) If h_i is adjusted by expanding its auxiliary data Y_i:
  $$X_i^{(t+1)} = X_i^{(t)} + 1$$

Let $\Delta X_i^{(t)} = X_i^{(t+1)} - X_i^{(t)}$ be the transition from $X_i^{(t)}$ to $X_i^{(t+1)}$. As there are two online cases, we have $\Delta X_i^{(t)} = -X_i^{(t)}$ or $\Delta X_i^{(t)} = 1$ with an independent probability of $\frac{1}{M}$ for each case. Thus:

$$\mathbb{E}[\Delta X_i^{(t)}] = -\mathbb{E}[X_i^{(t)}] * \frac{1}{M} + \frac{1}{M}$$

We can then obtain a recurrence to compute the expected value of $X_i^{(t+1)}$ on knowing the prior expected value of $X_i^{(t)}$:

$$\begin{aligned}\mathbb{E}[X_i^{(t+1)}] &= \mathbb{E}[X_i^{(t)}] + \mathbb{E}[\Delta X_i^{(t)}] \\ &= \mathbb{E}[X_i^{(t)}] * \left(1 - \frac{1}{M}\right) + \frac{1}{M}\end{aligned}$$

By expanding this recurrence ($\forall\ t \geq 1$), we have that:

$$\mathbb{E}[X_i^{(t)}] = \mathbb{E}[X_i^{(0)}] * \left(1 - \frac{1}{M}\right)^t + \frac{1}{M} * \sum_{i=0}^{t-1} \left(1 - \frac{1}{M}\right)^i$$

It is easy to see that the summation is a geometric series with the common ratio $|r| = |1 - \frac{1}{M}| < 1$, meaning that the series will converge and $\mathbb{E}[X_i]$ will reach its steady state when $t \longrightarrow \infty$:

$$\mathbb{E}[X_i] = \mathbb{E}[X_i^{(\infty)}] = \frac{1}{M} * \sum_{i=0}^{\infty} \left(1 - \frac{1}{M}\right)^i = \frac{1}{M} * M = 1$$

To this end, we complete our proofs to show that in the average case, $|Y_i| = \mathbb{E}[X_i] = 1$ for any auxiliary data $Y_i \in \mathcal{H}$. □

B.2 Correctness (Theorem 1)

Proof. To ensure correctness, we must show that:

(1) By building a set $\mathcal{H}$ of $O(\sqrt{N} \log N)$ offline hints, the client can find a hint h_i in the online phase that contains its desired query index x with an overwhelming probability.

(2) By replacing the query index x in the chosen random hint h_i with another patching index $\bar{z}$, the client can recover the data item $\mathsf{DB}[x]$ with an overwhelming probability.

For the client to find a hint $h_i \in \mathcal{H}$ that contains x, the set $\mathcal{H}$ must cover all N indices. Each offline hint is created by aggregating $\sqrt{N}$ random indices from $\sqrt{N}$ partitions (one index per partition). To cover all N indices, the offline hints must cover all $\sqrt{N}$ offsets in each database partition. Since all partitions are independent, we can apply the classic Coupon Collector Problem [52] to each partition to prove the expected sampling number is $O(\sqrt{N} \log \sqrt{N})$ to cover all $\sqrt{N}$ offsets within a partition. (Lemma 2.10 in [52]). Therefore, it suffices to set the number of hints as $M = O(\sqrt{N} \log N)$ to cover all offsets in all partitions, thereby N indices. Lazzaretti et al. (Section 4.2 of [44]) proved that for large N, $M = O(\lambda\sqrt{N})$ suffices to cover all N indices except with negligible probability of failure.

To recover DB$[x]$ successfully, the client combines the following three components: (1) an existing offline parity $\rho_i = \hat{\rho} \bigoplus \mathsf{DB}[x]$ containing DB$[x]$ with $\hat{\rho}$ is a punctured parity, (2) a random entry DB$[\bar{z}]$ from the same partition with DB$[x]$, (3) a patched parity $\hat{\rho} \bigoplus \mathsf{DB}[\bar{z}]$. The correctness of data recovery holds when given the offline parity ρ_i that the client already owns, the client can retrieve the punctured parity $\hat{\rho}$ to compute $\mathsf{DB}[x] = \rho_i \bigoplus \hat{\rho}$. This is true as the client combines the patched parity $\hat{\rho} \bigoplus \mathsf{DB}[\bar{z}]$ with the random entry DB$[\bar{z}]$ to produce the punctured parity $\hat{\rho}$. The patched parity $\hat{\rho} \bigoplus \mathsf{DB}[\bar{z}]$ is computed from one server by sending the patched set $\mathcal{S}_i \setminus \{x\} \cup \{\bar{z}\}$, where $\mathcal{S}_i$ is the hint set previously corresponded to producing parity ρ_i and $\mathcal{S}_i \setminus \{x\}$ is the punctured set corresponded to the punctured parity $\hat{\rho}$. The random entry DB$[\bar{z}]$ is recovered due to the correctness of the PPR protocol. □

B.3 Security (Theorem 2)

PROOF. We construct a simulator $\mathcal{S}$ in the Ideal such that a PPT environment $\mathcal{Z}$ cannot distinguish between its view in Ideal and Real. Note that $\mathcal{Z}$ can statically corrupt one server to get the view of the transcript, which is either from the simulation by $\mathcal{S}$ in Ideal, or from the protocol execution in Real. We denote the distribution $\mathcal{D}_n \leftarrow [m]^n$ as sampling a random set, which draws one random index from each partition P_k for $k \in [n]$. The simulator $\mathcal{S}$ functions as follows:

Offline: On receiving input (DB, N):

(1) $\mathcal{S}$ samples a random bit string $\{0,1\}^M$, where M_0 counts the bits zero (or one), denoting the number of random sets it needs to simulate the adversarial view.
(2) $\mathcal{S}$ outputs M_0 dummy sets $(\mathcal{S}_1, ..., \mathcal{S}_{M_0}) \xleftarrow{\$} \mathcal{D}_n^{M_0}$.

Online: On receiving query notification:

(1) $\mathcal{S}$ outputs $\mathcal{S}^* \xleftarrow{\$} \mathcal{D}_n$.
(2) $\mathcal{S}$ outputs one partition set $\mathcal{T}$, by invoking $\mathcal{S}_P$ from the Ideal world of PPR.

In Ideal, $\mathcal{S}^*$, $\mathcal{T}$ and $\{\mathcal{S}_1, \ldots, \mathcal{S}_{M_0}\}$ are truly random sets and independent from each other.

We now define a sequence of hybrid experiments Hyb_i. The differences between Hyb_i and Hyb_{i+1} are highlighted in red. We show that the Real and Ideal are indistinguishable:

$$|\Pr[\textsc{Real}_{\Pi_{\mathcal{F}},\mathcal{A},\mathcal{Z}}(\lambda) = 1] - \Pr[\textsc{Ideal}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(\lambda) = 1]| \le \mathsf{negl}(\lambda)$$

Hybrid 0. We define Hyb_0 experiment as $\textsc{Real}_{\Pi_{\mathcal{F}},\mathcal{A},\mathcal{Z}}$ with an adversarial $\mathcal{A}$ and an environment $\mathcal{Z}$. We rewrite the real protocol of Pirex as in Figure 15.

Hybrid 1. Let Hyb_1 experiment be as in Figure 16. In Hyb_1, the difference is that the client samples each set $\mathcal{S}_i$ from $\mathcal{D}_n$ in the offline and stores all sets of indices in plain. The client does not use PRF key with auxiliary data and there is no more PRS evaluation.

We argue that the view of $\mathcal{Z}$ for the offline and online in Hyb_1 and Hyb_0 are computationally indistinguishable. This is because the distribution of online queries created by any $\mathcal{S}_i$ sampled from either PRS or $\mathcal{D}_n$ would have a negligible difference under $\mathcal{Z}$.

```
Offline: The servers receive a database DB as input from $\mathcal{Z}$:
  (1) The client samples $M$ identifier bits $(\ell_1, \ldots, \ell_M) \xleftarrow{\$} \{0,1\}^M$
  (2) The client samples $M$ PRF keys $(\mathsf{sk}_1, \ldots, \mathsf{sk}_M)$ with $\mathsf{PRS.Gen}(1^\lambda)$
  (3) On receiving a PRF key $\mathsf{sk}_i$, $\mathsf{S}_{\ell_i}$ computes $\rho_i \leftarrow \bigoplus_{j=0}^{n-1} \mathsf{DB}[s_j]$
      where $s_j \in \mathcal{S}_i \leftarrow \mathsf{PRS.Eval}(\mathsf{sk}_i, \perp)$
  (4) The client receives parity $\rho_i$ from server $\mathsf{S}_{\ell_i}$
Online: On receiving an index $x \in P_k$ from $\mathcal{Z}$, the client executes:
  (5) Search $h_i = (\ell_i, \mathsf{sk}_i, \rho_i, Y_i)$ where $x \in \mathcal{S}_i \leftarrow \mathsf{PRS.Eval}(\mathsf{sk}_i, Y_i)$
  (6) $(\bar{z}, \mathcal{T}, \mathcal{T}^{(z)}) \leftarrow \mathsf{PPR.Gen}(m, n, k)$
  (7) $\bar{\mathcal{S}} \leftarrow \mathcal{S}_i \setminus \{x\} \cup \{\bar{z}\}$
  (8) Sample $\mathcal{S}' \leftarrow \mathsf{PRS.Eval}(\mathsf{sk}', \perp)$ where $\mathsf{sk}' \leftarrow \mathsf{PRS.Gen}(1^\lambda)$
  (9) Send $Q_{\ell_i} \leftarrow (\mathcal{S}', \mathcal{T}^{(\bar{z})})$ to server $\mathsf{S}_{\ell_i}$ and receive $\mathcal{R}_{\ell_i} = (\rho', w^{(\bar{z})})$
  (10) Send $Q_{\neg\ell_i} \leftarrow (\bar{\mathcal{S}}, \mathcal{T})$ to server $\mathsf{S}_{\neg\ell_i}$ and receive $\mathcal{R}_{\neg\ell_i} = (\bar{\rho}, w)$
  (11) $\mathsf{DB}[\bar{z}] \leftarrow \mathsf{PPR.Rec}(w^{(\bar{z})}, w)$
  (12) $\mathsf{DB}[x] \leftarrow \rho_i \oplus \bar{\rho} \oplus \mathsf{DB}[\bar{z}]$
  (13) Search $h_j = (\ell_j, \mathsf{sk}_j, \rho_j, Y_j)$ where $\bar{z} \in \mathcal{S}_j \leftarrow \mathsf{PRS.Eval}(\mathsf{sk}_j, Y_j)$
  (14) $\rho'_j \leftarrow \rho_j \oplus \mathsf{DB}[\bar{z}] \oplus \mathsf{DB}[x]$
  (15) $Y'_j \leftarrow Y_j \cup \{x\} \setminus \{\bar{z}\}$
  (16) Replace $h_j$ with $h'_j \leftarrow (\ell_j, \mathsf{sk}_j, \rho'_j, Y'_j)$
  (17) Replace $h_i$ with $h'_i \leftarrow (\ell_i, \mathsf{sk}', \rho', Y')$ with $Y' \leftarrow \perp$
```

Figure 15: Hyb_0 **experiment.**

```
Offline: The servers receive a database DB as input from $\mathcal{Z}$:
  (1) The client samples $M$ identifier bits $(\ell_1, \ldots, \ell_M) \xleftarrow{\$} \{0,1\}^M$
  (2) The client samples $M$ sets of indices $(\mathcal{S}_1, \ldots, \mathcal{S}_M) \xleftarrow{\$} \mathcal{D}_n^M$
  (3) On receiving a set $\mathcal{S}_i$, $\mathsf{S}_{\ell_i}$ computes $\rho_i \leftarrow \bigoplus_{j=0}^{n-1} \mathsf{DB}[s_j] \, \forall s_j \in \mathcal{S}_i$
  (4) The client receives parity $\rho_i$ from server $\mathsf{S}_{\ell_i}$
Online: On receiving an index $x \in P_k$ from $\mathcal{Z}$, the client executes:
  (5) Search $h_i = (\ell_i, \mathcal{S}_i, \rho_i)$ where $x \in \mathcal{S}_i$
  (6) $(\bar{z}, \mathcal{T}, \mathcal{T}^{(z)}) \leftarrow \mathsf{PPR.Gen}(m, n, k)$
  (7) $\bar{\mathcal{S}} \leftarrow \mathcal{S}_i \setminus \{x\} \cup \{\bar{z}\}$
  (8) Sample $\mathcal{S}' \xleftarrow{\$} \mathcal{D}_n$
  (9) Send $Q_{\ell_i} \leftarrow (\mathcal{S}', \mathcal{T}^{(\bar{z})})$ to server $\mathsf{S}_{\ell_i}$ and receive $\mathcal{R}_{\ell_i} = (\rho', w^{(\bar{z})})$
  (10) Send $Q_{\neg\ell_i} \leftarrow (\bar{\mathcal{S}}, \mathcal{T})$ to server $\mathsf{S}_{\neg\ell_i}$ and receive $\mathcal{R}_{\neg\ell_i} = (\bar{\rho}, w)$
  (11) $\mathsf{DB}[\bar{z}] \leftarrow \mathsf{PPR.Rec}(w^{(\bar{z})}, w)$
  (12) $\mathsf{DB}[x] \leftarrow \rho_i \oplus \bar{\rho} \oplus \mathsf{DB}[\bar{z}]$
  (13) Search $h_j = (\ell_j, \mathcal{S}_j, \rho_j)$ where $\bar{z} \in \mathcal{S}_j$
  (14) $\rho'_j \leftarrow \rho_j \oplus \mathsf{DB}[\bar{z}] \oplus \mathsf{DB}[x]$
  (15) $\mathcal{S}'_j \leftarrow \mathcal{S}_j \cup \{x\} \setminus \{\bar{z}\}$
  (16) Replace $h_j$ with $h'_j \leftarrow (\ell_j, \mathcal{S}'_j, \rho'_j)$
  (17) Replace $h_i$ with $h'_i \leftarrow (\ell_i, \mathcal{S}', \rho')$
```

Figure 16: Hyb_1 **experiment.**

Hybrid 2. Let Hyb_2 experiment be as in Figure 17. In Hyb_2, the main difference is that in the online, the client finds the hint $h_i = (\ell_i, \mathcal{S}_i, \rho_i)$ but does not use $\mathcal{S}_i$ as the input to create the queries as in Hyb_1. The client instead uses a newly sampled set $\mathcal{S}^* \xleftarrow{\$} \mathcal{D}_n$, with $x \in \mathcal{S}^*$. Since $\mathcal{S}^*$ is not related to any precomputed offline parity, the client cannot recover the data item DB$[x]$. Thus, we introduce the ideal functionality $\mathcal{F}$, which can return the correct answer based on the query input x from $\mathcal{Z}$. Since no precomputed

```
Offline: The servers receive a database DB as input from $\mathcal{Z}$:
    (1) The client samples $M$ identifier bits $(\ell_1, \dots, \ell_M) \xleftarrow{\$} \{0,1\}^M$
    (2) The client samples $M$ sets of indices $(\mathcal{S}_1, \dots, \mathcal{S}_M) \xleftarrow{\$} \mathcal{D}_n^M$
    (3) On receiving a set $\mathcal{S}_i$, $\mathsf{S}_{\ell_i}$ computes $\rho_i \leftarrow \bigoplus_{j=0}^{n-1} \mathsf{DB}[s_j]\ \forall s_j \in \mathcal{S}_i$
    (4) The client receives parity $\rho_i$ from server $\mathsf{S}_{\ell_i}$
Online: On receiving an index $x \in P_k$ from $\mathcal{Z}$, the client executes:
    (5) Search $h_i = (\ell_i, \mathcal{S}_i, \rho_i)$ where $x \in \mathcal{S}_i$
    (6) Sample $\mathcal{S}^* \xleftarrow{\$} \mathcal{D}_n$ where $x \in \mathcal{S}^*$
    (7) $(\bar{z}, \mathcal{T}, \mathcal{T}^{(\bar{z})}) \leftarrow \mathsf{PPR.Gen}(m, n, k)$
    (8) $\bar{\mathcal{S}} \leftarrow \mathcal{S}^* \setminus \{x\} \cup \{\bar{z}\}$
    (9) Sample $\mathcal{S}' \xleftarrow{\$} \mathcal{D}_n$
    (10) Send $Q_{\ell_i} \leftarrow (\mathcal{S}', \mathcal{T}^{(\bar{z})})$ to server $\mathsf{S}_{\ell_i}$ and receive $\mathcal{R}_{\ell_i} = (\rho', w^{(\bar{z})})$
    (11) Send $Q_{\neg\ell_i} \leftarrow (\bar{\mathcal{S}}, \mathcal{T})$ to server $\mathsf{S}_{\neg\ell_i}$ and receive $\mathcal{R}_{\neg\ell_i} = (\bar{\rho}, w)$
    (12) $\mathsf{DB}[x] \leftarrow \mathcal{F}(x)$
```

Figure 17: Hyb_2 **experiment.**

hint is consumed to recover DB$[x]$, the client also does not need to perform any hint refresh (Step 13-17 as in Hyb_1).

We argue that the view of $\mathcal{Z}$ for the offline and online in Hyb_2 has the same distribution as in Hyb_1. In the offline, the operations are identical. In the online, using the newly sampled set $\mathcal{S}^*$ yields the same distribution of queries as using $\mathcal{S}_i$. This is because the selected set $\mathcal{S}_i$ from Hyb_1 is always guaranteed to be $\mathcal{S}_i \xleftarrow{\$} \mathcal{D}_n$, conditioned on $x \in \mathcal{S}_i$. Recall that in OO-PIR, when a hint h_i with set $\mathcal{S}_i$ is consumed subject to containing x, the client always refresh it with a new hint that is also subjected to containing x to preserve the total number of hints containing x, with respect to the random distribution $\mathcal{D}_n^M$ from the offline phase. In Hyb_1, the client chose a random local hint h_j subject to containing a random element $\bar{z}$ and replaced $\bar{z}$ with x, which preserves the total number of hints containing x with respect to the distribution $\mathcal{D}_n^M$, but not preserve the total number of random hints M. This is because another hint h_j is consumed and this hint is randomly selected based on the random element $\bar{z}$. Finally, the total number of random hints M is preserved by adding a completely new random hint h'_i, which preserves the distribution $\mathcal{D}_n^M$ exactly as it is from the offline phase.

Since the newly sampled set $\mathcal{S}^*$ and the hint set $\mathcal{S}_i$ (that was not previously revealed to the corrupted server in the offline) are indistinguishable, the resulting queries will have the same distribution under the view of $\mathcal{Z}$ as in Hyb_1.

Hybrid 3. Let Hyb_3 experiment be as in Figure 18. In Hyb_3, the difference is that in the offline, the client only requests server S_{ℓ_i} to compute the parity ρ_i (by sending the set $\mathcal{S}_i$) but does not store the returned result. In the online, the client samples a new server identifier $\ell^* \xleftarrow{\$} \{0,1\}$ when sampling the new set $\mathcal{S}^*$, instead of using ℓ_i from the hint h_i.

We argue that the view of $\mathcal{Z}$ for the offline and online in Hyb_3 has the same distribution as in Hyb_2. In the offline, the corrupted server (in $\mathcal{Z}$'s view) receives the same distribution of random sets as in Hyb_2. In the online, replacing ℓ_i with ℓ^* only affects how the queries are distributed to which server, where the queries are already independently random and indistinguishable from $\mathcal{S}_i$.

Hybrid 4. Let Hyb_4 experiment be as in Figure 19. In Hyb_4, the main difference is that in the online, the client samples two random

```
Offline: The servers receive a database DB as input from $\mathcal{Z}$:
    (1) The client samples $M$ identifier bits $(\ell_1, \dots, \ell_M) \xleftarrow{\$} \{0,1\}^M$
    (2) The client samples $M$ sets of indices $(\mathcal{S}_1, \dots, \mathcal{S}_M) \xleftarrow{\$} \mathcal{D}_n^M$
    (3) On receiving a set $\mathcal{S}_i$, $\mathsf{S}_{\ell_i}$ computes $\rho_i \leftarrow \bigoplus_{j=0}^{n-1} \mathsf{DB}[s_j]\ \forall s_j \in \mathcal{S}_i$
Online: On receiving an index $x \in P_k$ from $\mathcal{Z}$, the client executes:
    (4) Sample $\mathcal{S}^* \xleftarrow{\$} \mathcal{D}_n$ where $x \in \mathcal{S}^*$ and $\ell^* \xleftarrow{\$} \{0,1\}$
    (5) $(\bar{z}, \mathcal{T}, \mathcal{T}^{(\bar{z})}) \leftarrow \mathsf{PPR.Gen}(m, n, k)$
    (6) $\bar{\mathcal{S}} \leftarrow \mathcal{S}^* \setminus \{x\} \cup \{\bar{z}\}$
    (7) Sample $\mathcal{S}' \xleftarrow{\$} \mathcal{D}_n$
    (8) Send $Q_{\ell^*} \leftarrow (\mathcal{S}', \mathcal{T}^{(\bar{z})})$ to server $\mathsf{S}_{\ell^*}$
    (9) Send $Q_{\neg\ell^*} \leftarrow (\bar{\mathcal{S}}, \mathcal{T})$ to server $\mathsf{S}_{\neg\ell^*}$
    (10) $\mathsf{DB}[x] \leftarrow \mathcal{F}(x)$
```

Figure 18: Hyb_3 **experiment.**

```
Offline: The servers receive a database DB as input from $\mathcal{Z}$:
    (1) The client samples $M$ identifier bits $(\ell_1, \dots, \ell_M) \xleftarrow{\$} \{0,1\}^M$
    (2) The client samples $M$ sets of indices $(\mathcal{S}_1, \dots, \mathcal{S}_M) \xleftarrow{\$} \mathcal{D}_n^M$
    (3) On receiving a set $\mathcal{S}_i$, $\mathsf{S}_{\ell_i}$ computes $\rho_i \leftarrow \bigoplus_{j=0}^{n-1} \mathsf{DB}[s_j]\ \forall s_j \in \mathcal{S}_i$
Online: On receiving an index $x \in P_k$ from $\mathcal{Z}$, the client executes:
    (4) Sample $\mathcal{S}^* \xleftarrow{\$} \mathcal{D}_n$ and $\mathcal{S}' \xleftarrow{\$} \mathcal{D}_n$
    (5) $(\bar{z}, \mathcal{T}_0, \mathcal{T}_1) \leftarrow \mathsf{PPR.Gen}(m, n, k)$
    (6) Send $Q_0 \leftarrow (\mathcal{S}', \mathcal{T}_0)$ to server $\mathsf{S}_0$
    (7) Send $Q_1 \leftarrow (\mathcal{S}^*, \mathcal{T}_1)$ to server $\mathsf{S}_1$
    (8) $\mathsf{DB}[x] \leftarrow \mathcal{F}(x)$
```

Figure 19: Hyb_4 **experiment.**

sets $\mathcal{S}^*$ and $\mathcal{S}'$ that are entirely independent to the desired index x, instead of sampling a $\mathcal{S}^* \supset \{x\}$. The client directly sends them to the servers without any set modification.

We argue that the view of $\mathcal{Z}$ for the offline and online in Hyb_4 has the same distribution as in Hyb_3. In the offline, the operations are identical. In the online, $\mathcal{S}^*$ has the same distribution as the patched set $\bar{\mathcal{S}}$. This is because $x \in \mathcal{S}^*$ is replaced by a random z from the same partition, which yields $\bar{\mathcal{S}} \leftarrow \mathcal{S}^* \setminus \{x\} \cap \{z\}$ that matches the distribution $\mathcal{D}_n$, where each element is independently and uniformly sampled within its partition. For the remaining query set $\mathcal{S}'$, it is also randomly sampled from $\mathcal{D}_n$. Since both server S_0 and S_1 have no prior knowledge about $\mathcal{S}^*$ and $\mathcal{S}'$, there is no need to specify any server identifier (as shown in Hyb_3) for the security when distributing the online queries. As the client no longer use $\bar{z}$ for patching, we also do not need to care which PPR query contains $\bar{z}$ and can arbitrarily send them.

Hybrid 5. Let Hyb_5 experiment be as in Figure 20. In Hyb_5 the only difference is that the client create the partition sets $(\mathcal{T}_0, \mathcal{T}_1)$ using simulator $\mathcal{S}_P$, instead of using the real PPR protocol. In the view of $\mathcal{Z}$, this yields the same distribution of partition sets as in Hyb_4, according to the PPR security proof in Lemma 1.

Note that Hyb_5 is identical to the simulator $\mathcal{S}$ in Ideal, which shows that Ideal and Real are computationally indistinguishable under the view of $\mathcal{Z}$.

The indistinguishability between Ideal and Real under $\mathcal{Z}$'s view implies that the real-world adversary, given a client's online query containing a patched set $\mathcal{S}$ and a partition set $\mathcal{T}$, cannot guess

Offline: The servers receive a database DB as input from $\mathcal{Z}$:
(1) The client samples M identifier bits $(\ell_1, \dots, \ell_M) \xleftarrow{\$} \{0,1\}^M$
(2) The client samples M sets of indices $(\mathcal{S}_1, \dots, \mathcal{S}_M) \xleftarrow{\$} \mathcal{D}_n^M$
(3) On receiving a set $\mathcal{S}_i$, S_{ℓ_i} computes $\rho_i \leftarrow \bigoplus_{j=0}^{n-1} \mathsf{DB}[s_j]\ \forall s_j \in \mathcal{S}_i$

Online: On receiving an index $x \in P_k$ from $\mathcal{Z}$, the client executes:
(4) Sample $\mathcal{S}^* \xleftarrow{\$} \mathcal{D}_n$ and $\mathcal{S}' \xleftarrow{\$} \mathcal{D}_n$
(5) Sample $(\mathcal{T}_0, \mathcal{T}_1)$ by invoking $\mathcal{S}_P$ from PPR
(6) Send $Q_0 \leftarrow (\mathcal{S}', \mathcal{T}_0)$ to server S_0
(7) Send $Q_1 \leftarrow (\mathcal{S}^*, \mathcal{T}_1)$ to server S_1
(8) $\mathsf{DB}[x] \leftarrow \mathcal{F}(x)$

Figure 20: Hyb_5 **experiment.**

what data entry is being retrieved with probability better than $\frac{1}{N}$ for any public database containing N data entries, where the number of partitions $n = \sqrt{N}$ and the number of entries per partition $m = \sqrt{N}$. The patched set $\mathcal{S}$ includes exactly $\sqrt{N}$ random indices, each is from a distinct partition, so the adversary always see exactly one random entry being accessed for each partition and cannot distinguish which random entry index in $\mathcal{S}$ is the patching element to indicate the (punctured) partition of interest. The partition set $\mathcal{T}$ is a query generated by the PPR protocol, which also does not reveal the partition of interest (see §A.1). Thus, the adversary can only guess the partition of interest with probability $\frac{1}{\sqrt{N}}$. As there are $\sqrt{N}$ indices per partition, the adversary can only guess what index being queried with probability $\frac{1}{\sqrt{N}}$ as random index being accessed for this partition is independently sampled with no correlation to the actual index being queried. In overall, the adversary cannot guess the index being queried with probability better than $\frac{1}{N}$. □

C Pirex+ Detailed Algorithms

The detailed algorithms for Pirex+ is presented in Figure 21, Figure 22, Figure 23, Figure 24, Figure 25, Figure 26. To enable the remote storage for offline hint parities, Pirex+ has slightly different interfaces (marked as blue) over Pirex as follows:

- $\underline{(\mathcal{H}, \mathbf{P}) \leftarrow \mathsf{Prep}(\mathsf{DB}, N)}$: Given a database DB of N entries, it outputs a hint $\mathcal{H}$ and an encrypted parity buffer $\mathbf{P}$.
- $\underline{(Q_0, Q_1, \mathcal{H}^*) \leftarrow \mathsf{Query}(x, \mathcal{H})}$: Given an entry index x and the hint $\mathcal{H}$, it outputs two online queries Q_0, Q_1 for server S_0, and S_1, respecitvely and an updated hint $\mathcal{H}^*$.
- $\underline{\mathcal{R}_I \leftarrow \mathsf{Answer}(Q_I, \mathsf{DB}, \mathbf{P})}$: Given a query Q_I, the database DB and the parity buffer $\mathbf{P}$, it outputs a response $\mathcal{R}_i$.
- $\underline{(b_x, \langle \rho' \rangle, \mathcal{H}') \leftarrow \mathsf{Recover}(\mathcal{R}_0, \mathcal{R}_1, \mathcal{H}^*)}$: Given the hint $\mathcal{H}^*$ and two responses $\mathcal{R}_0, \mathcal{R}_1$, it outputs the desired data entry b_x, an encrypted refresh parity $\langle \rho' \rangle$ and an updated hint $\mathcal{H}'$.

Additionally, Pirex+ has new interfaces to enable remote offline parities updates due to refresh and database updates:

- $\underline{(\mathcal{H}', \mathbf{P}') \leftarrow \mathsf{Rewrite}(\langle \rho' \rangle, \mathcal{H}, \mathbf{P})}$: Given a new encrypted parity $\langle \rho' \rangle$, the hint $\mathcal{H}$ and the buffer $\mathbf{P}$, it outputs a new $\mathbf{P}'$ such that $\langle \rho' \rangle$ is written into $\mathbf{P}$, and a correspondingly updated hint $\mathcal{H}'$.
- $\underline{\mathbf{P}' \leftarrow \mathsf{Update}(x, b'_x, \mathbf{P})}$: Given an entry index x to be updated, its new content b'_x, and the buffer $\mathbf{P}$, it outputs a new buffer $\mathbf{P}'$.

```
• $(\mathcal{H}, \mathbf{P}) \leftarrow \mathsf{Prep}(\mathsf{DB}, N)$:
for $i = 1$ to $M$ do
    $\ell_i \xleftarrow{\$} \{0,1\}$ and $\mathsf{sk}_i \leftarrow \mathsf{PRS.Gen}(1^\lambda)$
    $\mathcal{S}_i \leftarrow \mathsf{PRS.Eval}(\mathsf{sk}_i, \perp)$                                  } Executed by
    $\rho_i \leftarrow \sum_{j=1}^{n} \mathsf{DB}[s_j] \pmod{p}$ for all $s_j \in \mathcal{S}_i$        } Server $\mathsf{S}_{\ell_i}$
    $\mathbf{P}[i] \leftarrow \mathsf{AHE.Enc}(\mathsf{pk}, \rho_i)$
    $h_i \leftarrow (\ell_i, \mathsf{sk}_i, i)$
return $(\mathcal{H} \leftarrow (h_1, \dots, h_M), \mathbf{P})$
```

Figure 21: Pirex+ **offline phase.**

```
• $(Q_0, Q_1, \mathcal{H}^*) \leftarrow \mathsf{Query}(x, \mathcal{H})$:
parse $\mathcal{H} = (h_1, \dots, h_M)$
search $h_i = (\ell_i, \mathsf{sk}_i, \pi_i)$ where $x \in \mathcal{S}_i \leftarrow \mathsf{PRS.Eval}(\mathsf{sk}_i, \perp)$
$(\mathbf{q}_0, \mathbf{q}_1) \leftarrow \mathsf{XOR\text{-}PIR.Gen}(\pi_i)$
$(\hat{Q}_0, \hat{Q}_1) \leftarrow \mathsf{SubQuery}(x, \ell_i, \mathcal{S}_i)$
sample $\ell' \xleftarrow{\$} \{0,1\}$
sample $\mathsf{sk}' \leftarrow \mathsf{PRS.Gen}(1^\lambda)$ where $x \in \mathcal{S}' \leftarrow \mathsf{PRS.Eval}(\mathsf{sk}', \perp)$
$(\hat{Q}'_0, \hat{Q}'_1) \leftarrow \mathsf{SubQuery}(x, \neg\ell', \mathcal{S}')$
$Q_0 \leftarrow (\hat{Q}_0, \hat{Q}'_0, \mathbf{q}_0)$,   $Q_1 \leftarrow (\hat{Q}_1, \hat{Q}'_1, \mathbf{q}_1)$
$\mathcal{H}^* \leftarrow (h_1, \dots, h_i, \dots, h_M, h')$ where $h' \leftarrow (\ell', \mathsf{sk}', \perp)$
return $(Q_0, Q_1, \mathcal{H}^*)$

• $(\hat{Q}_0, \hat{Q}_1) \leftarrow \mathsf{SubQuery}(x, \ell, \mathcal{S})$:
$(\bar{z}, \mathcal{T}, \mathcal{T}^{(\bar{z})}) \leftarrow \mathsf{PPR.Gen}(m, n, k)$ where $k \leftarrow \lfloor \frac{x}{m} \rfloor$
$\bar{\mathcal{S}} \leftarrow \mathcal{S} \setminus \{x\} \cup \{z\}$
$\tilde{\mathcal{S}} \leftarrow \mathsf{PRS.Eval}(\mathsf{k}, \perp)$ where $\mathsf{k} \leftarrow \mathsf{PRS.Gen}(1^\lambda)$
$\hat{Q}_\ell \leftarrow (\tilde{\mathcal{S}}, \mathcal{T}^{(\bar{z})})$,   $\hat{Q}_{\neg\ell} \leftarrow (\bar{\mathcal{S}}, \mathcal{T})$
return $(\hat{Q}_0, \hat{Q}_1)$
```

Figure 22: Pirex+ **online phase: query.**

```
• $\mathcal{R}_I \leftarrow \mathsf{Answer}(Q_I, \mathsf{DB}, \mathbf{P})$:
parse $Q_I = ((\mathcal{S}, \mathcal{T}), (\mathcal{S}', \mathcal{T}'), \mathbf{q})$
$\bar{\rho} \leftarrow \sum_{j=1}^{n} \mathsf{DB}[s_j]$ for all $s_j \in \mathcal{S}$
$\bar{\rho}' \leftarrow \sum_{j=1}^{n} \mathsf{DB}[s'_j]$ for all $s'_j \in \mathcal{S}'$
$w \leftarrow \mathsf{PPR.Ret}(\mathcal{T}, \mathsf{DB})$
$w' \leftarrow \mathsf{PPR.Ret}(\mathcal{T}', \mathsf{DB})$
$r \leftarrow \mathsf{XOR\text{-}PIR.Ret}(\mathbf{q}, \mathbf{P})$
return $\mathcal{R}_I \leftarrow ((\bar{\rho}, w), (\bar{\rho}', w'), r)$
```

Figure 23: Pirex+ **online phase: answer.**

We highlight the difference between Pirex+ and Pirex in blue. Pirex+ makes use of the following standard 2-server XOR-PIR [26] XOR-PIR = (Gen, Ret, Rec) on the parity buffer $\mathbf{P}$ of size $2M$:

- $\underline{(\mathbf{q}_0, \mathbf{q}_1) \leftarrow \mathsf{Gen}(x)}$: Given an index $x \in [2M]$, it samples two bit vectors $\mathsf{s}\mathbf{q}_0, \mathbf{q}_1 \leftarrow \{0,1\}^{2M}$ such that $\mathbf{q}_0 \oplus \mathbf{q}_1 = \mathbf{e}$, where $\mathbf{e}$ is a unit vector with $\mathbf{e}[x] = 1$.
- $\underline{r_i \leftarrow \mathsf{Ret}(\mathbf{q}_I, \mathbf{P})}$: Given query $\mathbf{q}_I$ and buffer $\mathbf{P}$, it outputs an XOR result $r = \bigoplus_{j \in \mathcal{J}} \mathbf{P}[j]$ where $\mathcal{J} = \{j : \mathbf{q}_I[j] = 1\}$.
- $\underline{b_x \leftarrow \mathsf{Rec}(r_0, r_1)}$: Given two results r_0, r_1, it outputs $b_x = r_0 \oplus r_1$.

```
• $(b_x, \langle\rho'\rangle, \mathcal{H}') \leftarrow \mathsf{Recover}(\mathcal{R}_0, \mathcal{R}_1, \mathcal{H})$:
  parse $\mathcal{R}_0 = ((\bar{\rho}_0, w_0), (\bar{\rho}'_0, w'_0), r_0)$,   $\mathcal{R}_1 = ((\bar{\rho}_1, w_1), (\bar{\rho}'_1, w'_1), r_1)$
  parse $\mathcal{H}^* = (h_1, \dots, h_i, \dots, h_M, h')$
  parse $h_i = (\ell_i, \mathsf{sk}_i, \pi_i)$,   $h' = (\ell', \mathsf{sk}', \perp)$
  $b_{\bar{z}} \leftarrow \mathsf{PPR.Rec}(w_0, w_1)$
  $b'_{\bar{z}} \leftarrow \mathsf{PPR.Rec}(w'_0, w'_1)$
  $\rho_i \leftarrow \mathsf{AHE.Dec}(\mathsf{sk}, \langle\rho_i\rangle)$ where $\langle\rho_i\rangle \leftarrow \mathsf{XOR\text{-}PIR.Rec}(r_0, r_1)$
  $b_x \leftarrow b_{\bar{z}} + \rho_i - \bar{\rho}_{\neg\ell_i} \pmod{p}$
  $\langle\rho'\rangle \leftarrow \mathsf{AHE.Enc}(\mathsf{pk}, \rho')$ where $\rho' \leftarrow b_x - b'_{\bar{z}} + \bar{\rho}'_{\ell'} \pmod{p}$
  $h' \leftarrow (\ell', \mathsf{sk}', c + M)$
  $\mathcal{H}' \leftarrow (h_1, \dots, h', \dots, h_M)$
  return $(b_x, \langle\rho'\rangle, \mathcal{H}')$
```

Figure 24: Pirex+ **online phase: recover.**

```
• $(\mathcal{H}', \mathbf{P}') \leftarrow \mathsf{Rewrite}(\langle\rho'\rangle, \mathcal{H}, \mathbf{P})$:
Client:
  parse $\mathcal{H} = (h_1, \dots, h_c, \dots, h_M)$ with $h_c = (\ell_c, \mathsf{sk}_c, \pi_c)$
  $(\mathsf{q}_0, \mathsf{q}_1) \leftarrow \mathsf{XOR\text{-}PIR.Gen}(\pi_c)$
Server: $\mathsf{S}_l \in \{\mathsf{S}_0, \mathsf{S}_1\}$
  return $r_l \leftarrow \mathsf{XOR\text{-}PIR.Ret}(q_l, \mathbf{P})$
Client:
  $\langle\rho_c\rangle \leftarrow \mathsf{XOR\text{-}PIR.Rec}(r_0, r_1)$
  $\langle\rho_c\rangle' \leftarrow \mathsf{AHE.Enc}(\mathsf{pk}, \rho_c)$
  count $c \leftarrow c + 1 \pmod{M}$
  return $\mathcal{H}' \leftarrow (h_1, \dots, h'_c, \dots, h_M)$ with $h'_c = (\ell_c, \mathsf{sk}_c, c)$
Server: $\mathsf{S}_l \in \{\mathsf{S}_0, \mathsf{S}_1\}$
  write $\mathbf{P}_{\mathsf{left}}[c] = \langle\rho_c\rangle'$,  $\mathbf{P}_{\mathsf{right}}[c] = \langle\rho'\rangle$
  return $\mathbf{P}$
```

Figure 25: Pirex+ **oblivious refresh.**

```
• $\mathbf{P}' \leftarrow \mathsf{Update}(x, b'_x, \mathbf{P})$:
Client:
  $\mathbf{e} \leftarrow \{0\}^{2M}, k \leftarrow \lfloor \frac{x}{m} \rfloor$
  for $h_i = (\ell_i, \mathsf{sk}_i, \pi_i) \in \mathcal{H}$ do
      $\mathbf{e}[\pi_i] = 1$  if  $x \in \mathsf{PRS.Eval}(\mathsf{sk}_i, \perp)$
  for $i = 1$ to $2M$ do
      $\langle e_i\rangle \leftarrow \mathsf{AHE.Enc}(\mathsf{sk}, \mathbf{e}[i])$
  send $(\langle e_1\rangle, \dots, \langle e_{2M}\rangle)$ to $\mathsf{S}_0$ and $\mathsf{S}_1$
Server: $\mathsf{S}_l \in \{\mathsf{S}_0, \mathsf{S}_1\}$
  $\epsilon \leftarrow b'_x - \mathsf{DB}[x]$
  for $i = 1$ to $2M$ do
      $\mathbf{P}[i] \leftarrow \mathbf{P}[i] \boxplus (\langle e_i\rangle \boxdot \epsilon)$
  return $\mathbf{P}$
```

Figure 26: Pirex+ **remote update parities.**

D Pirex+ Deferred proofs

D.1 Security (Theorem 3)

Proof. We extend the Ideal simulator $\mathcal{S}$ from Appendix §B.3:

(1) In the offline phase, the simulator $\mathcal{S}$ additionally outputs a dummy encrypted parity buffer $\mathbf{P}$, where $\mathbf{P}[i] \leftarrow \langle 0\rangle$

(2) In the online phase, the simulator $\mathcal{S}$ additionally outputs two random bit strings $\mathbf{q}$ and $\mathbf{q}'$

(3) At c-th oblivious refresh, $\mathcal{S}$ writes into $\mathbf{P}_{\mathsf{left}}[c]$ and $\mathbf{P}_{\mathsf{left}}[c]$ a dummy encrypted parity value $\langle 0\rangle$

Offline: The servers receive a database DB as input from $\mathcal{Z}$:

(1) The client samples M identifier bits $(\ell_1, \dots, \ell_M) \xleftarrow{\$} \{0,1\}^M$
(2) The client samples M PRF keys $(\mathsf{sk}_1, \dots, \mathsf{sk}_M)$
(3) On receiving a PRF key sk_i, S_{ℓ_i} computes $\rho_i \leftarrow \bigoplus_{j=0}^{n-1} \mathsf{DB}[s_j]$ where $s_j \in \mathcal{S}_i \leftarrow \mathsf{PRS.Eval}(\mathsf{sk}_i, \perp)$
(4) The client receives ρ_i from S_{ℓ_i} and sets hint $h_i = (\ell_i, \mathsf{sk}_i, \pi_i = i)$
(5) The client sends parity buffer $\mathbf{P}$ to $\mathsf{S}_0, \mathsf{S}_1$ with $\mathbf{P}[i] = \langle\rho_i\rangle$

Online: On receiving an index $x \in P_k$ from $\mathcal{Z}$, the client executes:

(6) Search $h_i = (\ell_i, \mathsf{sk}_i, \pi_i)$ where $x \in \mathcal{S}_i \leftarrow \mathsf{PRS.Eval}(\mathsf{sk}_i, \perp)$
(7) $(\hat{Q}_0, \hat{Q}_1) \leftarrow \mathsf{SubQuery}(x, \ell_i, \mathcal{S}_i)$
(8) Sample $\ell' \xleftarrow{\$} \{0,1\}, \mathsf{sk}' \xleftarrow{\$} \{0,1\}^\lambda$ where $x \in \mathcal{S}' \leftarrow \mathsf{PRS.Eval}(\mathsf{sk}', \perp)$
(9) $(\hat{Q}'_0, \hat{Q}'_1) \leftarrow \mathsf{SubQuery}(x, \neg\ell', \mathcal{S}')$
(10) Send $Q_0 \leftarrow (\hat{Q}_0, \hat{Q}'_0)$ to server S_0 and receive $\mathcal{R}_0$
(11) Send $Q_1 \leftarrow (\hat{Q}_1, \hat{Q}'_1)$ to server S_1 and receive $\mathcal{R}_1$
(12) Read $\langle\rho_i\rangle$ with XOR-PIR on position π_i
(13) Obtain $\mathsf{DB}[x]$ using $\mathcal{R}_0$, $\mathcal{R}_1$, and parity ρ_i
(14) Obtain parity ρ' using $\mathcal{R}_0$, $\mathcal{R}_1$, and $\mathsf{DB}[x]$

Refresh: On retrieving new parity ρ', the client executes:

(15) Write $\mathbf{P}_{\mathsf{right}}[c] = \langle\rho'\rangle$, $h' = (\ell', \mathsf{sk}', \pi' = c + M)$
(16) Read $\langle\rho_c\rangle$ with XOR-PIR on position $\pi_c \in h_c$
(17) Write $\mathbf{P}_{\mathsf{left}}[c] = \langle\rho_c\rangle'$, $h_c = (\ell_c, \mathsf{sk}_c, c)$

Figure 27: Hyb_0^+ **experiment.**

In addition to the online and offline phase, the simulator $\mathcal{S}$ can receive an update command, which outputs an IND-CPA encrypted random binary vector.

We now define a sequence of hybrid experiments Hyb_i^+. The differences between Hyb_i^+ and Hyb_{i+1}^+ are highlighted in red. We show that the Real and Ideal are indistinguishable:

$$|\Pr[\textsc{Real}_{\Pi_{\mathcal{F}},\mathcal{A},\mathcal{Z}}(\lambda) = 1] - \Pr[\textsc{Ideal}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(\lambda) = 1]| \leq \mathsf{negl}(\lambda)$$

Hybrid 0. We define Hyb_0^+ experiment as the $\textsc{Real}_{\Pi_{\mathcal{F}},\mathcal{A},\mathcal{Z}}$ with $\Pi_{\mathcal{F}} =$ Pirex+, adversarial $\mathcal{A}$ and environment $\mathcal{Z}$ in Figure 27

Hybrid 1. Let Hyb_1^+ experiment be as in Figure 28. Similar to Hyb_1, the client samples a set $\mathcal{S}_j \xleftarrow{\$} \mathcal{D}_n$ instead of using a PRF key. This adjustment does not affect the way a client retrieves a parity using standard XOR-PIR and thus, the view of $\mathcal{Z}$ in Hyb_1^+ and Hyb_0^+ are computationally indistinguishable.

Hybrid 2. Let Hyb_2^+ experiment be as in Figure 29. From Hyb_2, we know that the client uses a set $\mathcal{S}^*$ that is not related to any offline parity to create the queries. Thus, no offline parity can be used to recover $\mathsf{DB}[x]$ so the $\mathcal{F}$ is invoked to obtain $\mathsf{DB}[x]$. The client still invokes XOR-PIR to maintain the indistinguishability with Hyb_1^+. Since no offline parities are used in Hyb_2^+, we do not need to write any new refresh parity in the buffer $\mathbf{P}_{\mathsf{right}}$. Due to the IND-CPA property of the AHE encryption, the client can write $\langle 0\rangle$ into $\mathbf{P}_{\mathsf{right}}[c]$ as a dummy refresh parity, which makes the view of $\mathcal{Z}$ in Hyb_2^+ and Hyb_1^+ computationally indistinguishable.

Hybrid 3. Let Hyb_3^+ experiment be as in Figure 30. In Hyb_3^+, the additional difference (w.r.t experiment Hyb_3) is that the client does not use the offline parities computed by the servers to create the parity buffer $\mathbf{P}$. The client instead sends a dummy encrypted buffer

```
Offline: The servers receive a database DB as input from $\mathcal{Z}$:
    (1) The client samples $M$ identifier bits $(\ell_1, \ldots, \ell_M) \xleftarrow{\$} \{0,1\}^M$
    (2) The client samples $M$ sets of indices $(\mathcal{S}_1, \ldots, \mathcal{S}_M) \xleftarrow{\$} \mathcal{D}_n^M$
    (3) On receiving a set $\mathcal{S}_i$, $\mathsf{S}_{\ell_i}$ computes $\rho_i \leftarrow \bigoplus_{j=0}^{n-1} \mathsf{DB}[s_j]\ \forall s_j \in \mathcal{S}_i$
    (4) The client receives $\rho_i$ from $\mathsf{S}_{\ell_i}$ and set hint $h_i = (\ell_i, \mathcal{S}_i, \pi_i = i)$
    (5) The client sends parity buffer $\mathbf{P}$ to $\mathsf{S}_0, \mathsf{S}_1$ with $\mathbf{P}[i] = \langle \rho_i \rangle$
Online: On receiving an index $x \in P_k$ from $\mathcal{Z}$, the client executes:
    (6) Search $h_i = (\ell_i, \mathcal{S}_i, \rho_i)$ where $x \in \mathcal{S}_i$
    (7) $(\hat{Q}_0, \hat{Q}_1) \leftarrow \mathsf{SubQuery}(x, \ell_i, \mathcal{S}_i)$
    (8) Sample $\ell' \xleftarrow{\$} \{0,1\}$, $\mathcal{S}' \xleftarrow{\$} \mathcal{D}_n$ where $x \in \mathcal{S}'$
    (9) $(\hat{Q}_0', \hat{Q}_1') \leftarrow \mathsf{SubQuery}(x, \neg\ell', \mathcal{S}')$
    (10) Send $Q_0 \leftarrow (\hat{Q}_0, \hat{Q}_0')$ to server $\mathsf{S}_0$ and receive $\mathcal{R}_0$
    (11) Send $Q_1 \leftarrow (\hat{Q}_1, \hat{Q}_1')$ to server $\mathsf{S}_1$ and receive $\mathcal{R}_1$
    (12) Read $\langle \rho_i \rangle$ with XOR-PIR on position $\pi_i$
    (13) Obtain $\mathsf{DB}[x]$ using $\mathcal{R}_0$, $\mathcal{R}_1$, and parity $\rho_i$
    (14) Obtain parity $\rho'$ using $\mathcal{R}_0$, $\mathcal{R}_1$, and $\mathsf{DB}[x]$
Refresh: On retrieving new parity $\rho'$, the client executes:
    (15) Write $\mathbf{P}_{\mathsf{right}}[c] = \langle \rho' \rangle$,  $h' = (\ell', \mathcal{S}', \pi' = c + M)$
    (16) Read $\langle \rho_c \rangle$ with XOR-PIR on position $\pi_c \in h_c$
    (17) Write $\mathbf{P}_{\mathsf{left}}[c] = \langle \rho_c \rangle'$,  $h_c = (\ell_c, \mathcal{S}_c, c)$
```

Figure 28: Hyb_1^+ **experiment.**

```
Offline: The servers receive a database DB as input from $\mathcal{Z}$:
    (1) The client samples $M$ identifier bits $(\ell_1, \ldots, \ell_M) \xleftarrow{\$} \{0,1\}^M$
    (2) The client samples $M$ sets of indices $(\mathcal{S}_1, \ldots, \mathcal{S}_M) \xleftarrow{\$} \mathcal{D}_n^M$
    (3) On receiving a set $\mathcal{S}_i$, $\mathsf{S}_{\ell_i}$ computes $\rho_i \leftarrow \bigoplus_{j=0}^{n-1} \mathsf{DB}[s_j]\ \forall s_j \in \mathcal{S}_i$
    (4) The client receives $\rho_i$ from $\mathsf{S}_{\ell_i}$ and set hint $h_i = (\ell_i, \mathcal{S}_i, \pi_i = i)$
    (5) The client sends parity buffer $\mathbf{P}$ to $\mathsf{S}_0, \mathsf{S}_1$ with $\mathbf{P}[i] = \langle \rho_i \rangle$
Online: On receiving an index $x \in P_k$ from $\mathcal{Z}$, the client executes:
    (4) Sample $\mathcal{S}^* \xleftarrow{\$} \mathcal{D}_n$ where $x \in \mathcal{S}^*$
    (5) $(\hat{Q}_0, \hat{Q}_1) \leftarrow \mathsf{SubQuery}(x, \ell_i, \mathcal{S}^*)$
    (6) Sample $\ell' \xleftarrow{\$} \{0,1\}$, $\mathcal{S}' \xleftarrow{\$} \mathcal{D}_n$ where $x \in \mathcal{S}'$
    (7) $(\hat{Q}_0', \hat{Q}_1') \leftarrow \mathsf{SubQuery}(x, \neg\ell', \mathcal{S}')$
    (8) Send $Q_0 \leftarrow (\hat{Q}_0, \hat{Q}_0')$ to server $\mathsf{S}_0$ and receive $\mathcal{R}_0$
    (9) Send $Q_1 \leftarrow (\hat{Q}_1, \hat{Q}_1')$ to server $\mathsf{S}_1$ and receive $\mathcal{R}_1$
    (10) Invoke XOR-PIR on position $\pi_i$
    (11) $\mathsf{DB}[x] \leftarrow \mathcal{F}(x)$
Refresh: With dummy parity $\rho' = 0$, the client executes:
    (8) Write $\mathbf{P}_{\mathsf{right}}[c] = \langle \rho' \rangle$
    (9) Read $\langle \rho_c \rangle$ with XOR-PIR on position $\pi_c \in h_c$
    (10) Write $\mathbf{P}_{\mathsf{left}}[c] = \langle \rho_c \rangle'$,  $h_c = (\ell_c, \mathcal{S}_c, c)$
```

Figure 29: Hyb_2^+ **experiment.**

$\mathbf{P}$, with $\mathbf{P}[i] = \langle 0 \rangle$. Since AHE is an IND-CPA encryption, $\mathcal{Z}$ cannot distinguish between a dummy parity buffer and a buffer containing the offline parities computed by the servers. In addition, since there is no encrypted parity to be retrieved in the online phase for data recovery, the client invokes XOR-PIR on random input. Therefore, the view of $\mathcal{Z}$ in Hyb_3^+ and Hyb_2^+ are computationally indistinguishable.

Hybird 4. Let Hyb_4^+ experiment be as in Figure 31. In Hyb_4^+, the additional difference is that for oblivious refresh, the client already knows the value to be rewrite must be $\langle 0 \rangle$. The client can write $\mathbf{P}_{\mathsf{left}}[c] = \langle 0 \rangle$, regardless of the PIR query produced by XOR-PIR. To simulate a PIR query in this case, the client can invoke XOR-PIR on deterministic counter c. By the security of XOR-PIR, under a statically adversarial view $\mathcal{A}$, $\mathcal{Z}$ cannot distinguish between if the received queries are in a deterministic sequence. Thus, the view of $\mathcal{Z}$ in Hyb_4^+ and Hyb_3^+ are computationally indistinguishable.

```
Offline: The servers receive a database DB as input from $\mathcal{Z}$:
    (1) The client samples $M$ identifier bits $(\ell_1, \ldots, \ell_M) \xleftarrow{\$} \{0,1\}^M$
    (2) The client samples $M$ sets of indices $(\mathcal{S}_1, \ldots, \mathcal{S}_M) \xleftarrow{\$} \mathcal{D}_n^M$
    (3) On receiving a set $\mathcal{S}_i$, $\mathsf{S}_{\ell_i}$ computes $\rho_i \leftarrow \bigoplus_{j=0}^{n-1} \mathsf{DB}[s_j]\ \forall s_j \in \mathcal{S}_i$
    (4) The client sends parity buffer $\mathbf{P}$ to $\mathsf{S}_0, \mathsf{S}_1$ with $\mathbf{P}[i] = \langle 0 \rangle$
Online: On receiving an index $x \in P_k$ from $\mathcal{Z}$, the client executes:
    (3) Sample $\mathcal{S}^* \xleftarrow{\$} \mathcal{D}_n$ where $x \in \mathcal{S}^*$ and $\ell^* \xleftarrow{\$} \{0,1\}$
    (4) $(\hat{Q}_0, \hat{Q}_1) \leftarrow \mathsf{SubQuery}(x, \ell^*, \mathcal{S}^*)$
    (5) Sample $\ell' \xleftarrow{\$} \{0,1\}$, $\mathcal{S}' \xleftarrow{\$} \mathcal{D}_n$ where $x \in \mathcal{S}'$
    (6) $(\hat{Q}_0', \hat{Q}_1') \leftarrow \mathsf{SubQuery}(x, \neg\ell', \mathcal{S}')$
    (7) Send $Q_0 \leftarrow (\hat{Q}_0, \hat{Q}_0')$ to server $\mathsf{S}_0$
    (8) Send $Q_1 \leftarrow (\hat{Q}_1, \hat{Q}_1')$ to server $\mathsf{S}_1$
    (9) Invokes XOR-PIR on random input $\pi'$
    (10) $\mathsf{DB}[x] \leftarrow \mathcal{F}(x)$
Refresh: With dummy parity = 0, the client executes:
    (7) Write $\mathbf{P}_{\mathsf{right}}[c] = \langle 0 \rangle$
    (8) Read $\langle \rho_c \rangle$ with XOR-PIR on position $\pi_c \in h_c$
    (9) Write $\mathbf{P}_{\mathsf{left}}[c] = \langle 0 \rangle$
```

Figure 30: Hyb_3^+ **experiment.**

```
Offline: The servers receive a database DB as input from $\mathcal{Z}$:
    (1) The client samples $M$ identifier bits $(\ell_1, \ldots, \ell_M) \xleftarrow{\$} \{0,1\}^M$
    (2) The client samples $M$ sets of indices $(\mathcal{S}_1, \ldots, \mathcal{S}_M) \xleftarrow{\$} \mathcal{D}_n^M$
    (3) On receiving a set $\mathcal{S}_i$, $\mathsf{S}_{\ell_i}$ computes $\rho_i \leftarrow \bigoplus_{j=0}^{n-1} \mathsf{DB}[s_j]\ \forall s_j \in \mathcal{S}_i$
    (4) The client sends parity buffer $\mathbf{P}$ to $\mathsf{S}_0, \mathsf{S}_1$ with $\mathbf{P}[i] = \langle 0 \rangle$
Online: On receiving an index $x \in P_k$ from $\mathcal{Z}$, the client executes:
    (3) Sample $\mathcal{S}_0^* \xleftarrow{\$} \mathcal{D}_n$ and $\mathcal{S}_1^* \xleftarrow{\$} \mathcal{D}_n$
    (4) $(z, \mathcal{T}_0, \mathcal{T}_1) \leftarrow \mathsf{PPR.Gen}(m, n, k)$
    (5) Sample $\mathcal{S}_0' \xleftarrow{\$} \mathcal{D}_n$ and $\mathcal{S}_1' \xleftarrow{\$} \mathcal{D}_n$
    (6) $(z', \mathcal{T}_0', \mathcal{T}_1') \leftarrow \mathsf{PPR.Gen}(m, n, k)$
    (7) Send $Q_0 \leftarrow ((\mathcal{S}_0^*, \mathcal{T}_0), (\mathcal{S}_0', \mathcal{T}_0'))$ to server $\mathsf{S}_0$
    (8) Send $Q_1 \leftarrow ((\mathcal{S}_1^*, \mathcal{T}_1), (\mathcal{S}_1', \mathcal{T}_1'))$ to server $\mathsf{S}_1$
    (9) Invokes XOR-PIR on random input $\pi'$
    (10) $\mathsf{DB}[x] \leftarrow \mathcal{F}(x)$
Refresh: With dummy parity = 0, the client executes:
    (7) Write $\mathbf{P}_{\mathsf{right}}[c] = \langle 0 \rangle$
    (8) Invokes XOR-PIR on position $\pi_c = c$
    (9) Write $\mathbf{P}_{\mathsf{left}}[c] = \langle 0 \rangle$
```

Figure 31: Hyb_4^+ **experiment.**

Hybird 5. Let Hyb_5^+ experiment be as in Figure 32. In Hyb_5^+, the additional difference is that the client invokes XOR-PIR on random input, which is also indistinguishable from PIR query produced in Hyb_4^+. Thus, the view of $\mathcal{Z}$ in Hyb_5^+ and Hyb_4^+ are computationally indistinguishable.

Note that Hyb_5^+ is identical to the online and offline phase from the simulator $\mathcal{S}$ in the Ideal. It is left to show that for the parities' update algorithm, $\mathcal{Z}$ cannot distinguish between the encrypted

```
Offline: The servers receive a database DB as input from $\mathcal{Z}$:
    (1) The client samples $M$ identifier bits $(\ell_1, \ldots, \ell_M) \xleftarrow{\$} \{0,1\}^M$
    (2) The client samples $M$ sets of indices $(\mathcal{S}_1, \ldots, \mathcal{S}_M) \xleftarrow{\$} \mathcal{D}_n^M$
    (3) On receiving a set $\mathcal{S}_i$, $\mathsf{S}_{\ell_i}$ computes $\rho_i \leftarrow \bigoplus_{j=0}^{n-1} \mathsf{DB}[s_j]\ \forall s_j \in \mathcal{S}_i$
    (4) The client sends parity buffer $\mathbf{P}$ to $\mathsf{S}_0, \mathsf{S}_1$ with $\mathbf{P}[i] = \langle 0 \rangle$

Online: On receiving an index $x \in P_k$ from $\mathcal{Z}$, the client executes:
    (3) Sample $\mathcal{S}_0^* \xleftarrow{\$} \mathcal{D}_n$ and $\mathcal{S}_1^* \xleftarrow{\$} \mathcal{D}_n$
    (4) Sample $\mathcal{T}_0$, $\mathcal{T}_1$ by invoking $\mathcal{S}_P$ from PPR
    (5) Sample $\mathcal{S}_0' \xleftarrow{\$} \mathcal{D}_n$ and $\mathcal{S}_1' \xleftarrow{\$} \mathcal{D}_n$
    (6) Sample $\mathcal{T}_0'$, $\mathcal{T}_1'$ by invoking $\mathcal{S}_P$ from PPR
    (7) Send $Q_0 \leftarrow ((\mathcal{S}_0^*, \mathcal{T}_0), (\mathcal{S}_0', \mathcal{T}_0'))$ to server $\mathsf{S}_0$
    (8) Send $Q_1 \leftarrow ((\mathcal{S}_1^*, \mathcal{T}_1), (\mathcal{S}_1', \mathcal{T}_1'))$ to server $\mathsf{S}_1$
    (9) Invokes XOR-PIR on random input $\pi'$
   (10) $\mathsf{DB}[x] \leftarrow \mathcal{F}(x)$

Refresh: With dummy parity = 0, the client executes:
    (6) Write $\mathbf{P}_{\text{right}}[c] = \langle 0 \rangle$
    (7) Invokes XOR-PIR on random input $\pi_c$
    (8) Write $\mathbf{P}_{\text{left}}[c] = \langle 0 \rangle$
```

Figure 32: Hyb_5^+ **experiment.**

dummy binary vector from $\mathcal{S}$ and the real encrypted update vector from Figure 26, which is true due to the IND-CPA property of AHE. To this end, the Ideal and Real for Pirex+ are computationally indistinguishable.

□