Client-Efficient Online-Offline Private Information Retrieval

Hoang-Dung Nguyen
Virginia Tech

Jorge Guajardo
Robert Bosch LLC – RTC

Thang Hoang
Virginia Tech

Abstract

Private Information Retrieval (PIR) permits clients to query entries from a public database hosted on untrusted servers in a privacy-preserving manner. Traditional PIR model suffers from high computation and/or bandwidth cost due to entire database processing for privacy. Recently, Online-Offline PIR (OO-PIR) has been suggested to improve the practicality of PIR, where query-independent materials are precomputed beforehand to accelerate online access. While state-of-the-art OO-PIR schemes (e.g., S&P'24, CRYPTO'23) successfully reduce the online processing overhead to sublinear, they still impose sustainable bandwidth and storage burdens on the client, especially when operating on large databases.

In this paper, we propose Pirex, a new OO-PIR scheme with eminent client performance while maintaining the sublinear server processing efficiency. Specifically, Pirex offers clients with sublinear processing, minimal inbound bandwidth, and low storage requirements. Our Pirex design is fairly simple yet efficient, where the majority of 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 experimental results demonstrated that Pirex outperforms existing OO-PIR schemes by at least two orders of magnitude. Concretely, with a 1 TB database, Pirex only takes 0.8s to query a 256-KB entry, compared with 30-220s by the state-of-the-art.

1 INTRODUCTION

Public databases provide users with seamless access to data resources across diverse domains: social (e.g., news media), entertainment (e.g., audio/video archives), economical (e.g., stock market), healthcare (e.g., medical, pharmaceutical data), geospatial services (e.g., locations, directions). These public databases not only eliminate the need for local storage, but also enables users to retrieve the latest information remotely at their convenience. Despite their usefulness, using public database services may create privacy concerns. While these databases are not considered sensitive, the users' queries on them can still, inadvertently reveal sensitive or personally identifiable information, such as their personal preferences, current location, health conditions, or revenue streams [46, 47]. A malicious database server can misuse the users' query behaviors (i.e. user locations [41], search frequency [58]) and expose them to malicious activities such as price and search discrimination [29, 48].

To protect user privacy, Chor et. al [17] proposed Private Information Retrieval (PIR), a cryptographic primitive that permits users to retrieve a public data entry without revealing to the adversarial server what entry has been accessed. Despite its strong privacy guarantee, PIR can be expensive in terms of bandwidth and processing overhead. Various attempts have successfully reduced the PIR bandwidth cost in centralized [8, 14, 16, 25, 33, 37, 39] and distributed database settings [7, 10–12, 23, 24, 26, 59], but the high processing cost remains a barrier to making PIR practical. Beimel et al. [12] proved that, under the standard computation model, any secure PIR scheme must incur a lower bound of linear server processing (w.r.t the database size). Sion et al. [55] showed that, in certain conditions, streaming the database is more efficient than such a linear processing.

To be more computationally efficient, PIR has been studied in different computation models such as preprocessing [12, 13, 15, 19, 20] or batching [8, 12, 31, 38, 44]. Patel et al. [49] recently proposed an eminent preprocessing paradigm called Online-Offline PIR (OO-PIR), where the client can privately precompute query-independent hints beforehand to accelerate online access. Corrigan-Gibbs et al. [20] designed the first OO-PIR scheme that achieved *sublinear* server computation for the online query. Several works were later proposed to improve the OO-PIR efficiency [32, 35, 60] or optimality [34, 54, 61] and achieve promising results.

Although OO-PIR achieves a sublinear server processing cost, it poses a critical bandwidth and storage burden to the client. Specifically, most OO-PIR schemes [19, 20, 32, 34, 35, 54, 60, 61] require a client to store a considerable amount of

Table 1: Our proposed Pirex/Pirex+ schemes vs. prior OO-PIR schemes.

Scheme	Client Online B/W		Online Computation [‡]		Client	Total Client	Offline Computation[‡]	
	In	Out	Client	Server	Storage	Offline B/W	Client	Server
CK20 (PRF) [20]	$O(\lambda B)$	$\tilde{O}(\lambda^2)$	$O(\lambda B)_\oplus + \tilde{O}(\lambda N)_\mathsf{p}$	$O(\lambda B\sqrt{N})_\oplus + O(\lambda\sqrt{N})_\mathsf{p}$	$\tilde{O}(\lambda B\sqrt{N})$	$\tilde{O}(\lambda B\sqrt{N})$	$\tilde{O}(\lambda\sqrt{N})_\mathsf{p}$	$\tilde{O}(\lambda BN)_\oplus + \tilde{O}(\lambda N)_\mathsf{p}$
CK20 (PRP) [20]		$\tilde{O}(\lambda\sqrt{N})$	$O(\lambda B)_\oplus + \tilde{O}(\lambda\sqrt{N})_\mathsf{p}$	$O(\lambda B\sqrt{N})_\oplus$				
Shi et al. [54]	$O(\lambda B)$	$\tilde{O}(\lambda^2)$	$O(\lambda B)_\oplus + \tilde{O}(\lambda\sqrt{N})_\mathsf{p}$	$\tilde{O}(\lambda B\sqrt{N})_\oplus + \tilde{O}(\lambda\sqrt{N})_\mathsf{p}$	$\tilde{O}(\lambda B\sqrt{N})$	$\tilde{O}(\lambda B\sqrt{N})$	$\tilde{O}(\lambda\sqrt{N})_\mathsf{p}$	$\tilde{O}(\lambda BN)_\oplus + \tilde{O}(\lambda N)_\mathsf{p}$
Checklist [32]	$O(B)$	$\tilde{O}(\lambda)$	$O(B)_\oplus + O(N)_\mathsf{p}$	$O(B\sqrt{N})_\oplus + O(\sqrt{N})_\mathsf{p}$	$O(\lambda B\sqrt{N})$	$O(\lambda B\sqrt{N})$	$O(\lambda\sqrt{N})_\mathsf{p}$	$O(\lambda BN)_\oplus + O(\lambda N)_\mathsf{p}$
TreePIR [35]	$O(B\sqrt{N})$	$\tilde{O}(\lambda)$	$O(B)_\oplus + \tilde{O}(\lambda\sqrt{N})_\mathsf{p}$	$O(B\sqrt{N})_\oplus + O(\sqrt{N})_\mathsf{p}$	$O(\lambda B\sqrt{N})$	$O(\lambda B\sqrt{N})$	$O(\lambda\sqrt{N})_\mathsf{p}$	$O(\lambda BN)_\oplus + O(\lambda N)_\mathsf{p}$
Piano [60]	$O(B\sqrt{N})$	$\tilde{O}(\sqrt{N})$	$O(B)_\oplus + O(\lambda\sqrt{N})_\mathsf{p}$	$O(B\sqrt{N})_\oplus$	$O(\lambda B\sqrt{N})$	$O(BN)$	$O(\lambda BN)_\oplus + O(\lambda N)_\mathsf{p}$	—
Pirex	$O(B)$	$\tilde{O}(\sqrt{N})$	$O(B)_\oplus + O(\lambda\sqrt{N})_\mathsf{p}$	$O(B\sqrt{N})_\oplus$	$O(\lambda B\sqrt{N})$	$O(\lambda B\sqrt{N})$	$O(\lambda\sqrt{N})_\mathsf{p}$	$O(\lambda BN)_\oplus + O(\lambda N)_\mathsf{p}$
Pirex+			$O(B)_{\mathbb{F}/\mathbb{G}} + O(\lambda\sqrt{N})_\mathsf{p}$	$O(B\sqrt{N})_\mathbb{F} + O(B\lambda\sqrt{N})_\oplus$	$O(\lambda^2\sqrt{N})$		$O(\lambda B\sqrt{N})_\mathbb{G} + O(\lambda\sqrt{N})_\mathsf{p}$	$O(\lambda BN)_\mathbb{F} + O(\lambda N)_\mathsf{p}$

‡ $\oplus$ 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 $\mathsf{polylog}(N)$ terms in the asymptotic complexity.

the pre-computed hints. For a database with N entries each of size B, the total client storage is at least $\Omega(\lambda B\sqrt{N})$ (where λ is the security parameter). More importantly, to privately read a database entry, the client is required to download from $O(\lambda)$ to $O(\sqrt{N})$ extra entries. This cost can be significant in real data platforms (e.g., Amazon EBS [1], Azure Blob Storage [2], PostgreSQL [4], and content distribution systems (e.g., [27, 28, 52]), where the entry size granularity is large (e.g., 64 KB-1 MB). For example, the most efficient OO-PIR scheme to date [60] takes the client approximately 11.5 GB storage and 512 MB bandwidth to read a 256-KB entry from a 1 TB Azure Storage.

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 that features low client bandwidth and storage for large databases while retaining sublinear client and server processing complexity?

1.1 Our Contributions

We answer the question affirmatively by presenting a highly efficient OO-PIR framework called Pirex, which stands for Private Information Retrieval with Client Expedience. To our knowledge, Pirex is the first OO-PIR scheme that offers $O(1)$ client inbound bandwidth blowup, and low client storage with sublinear client and server processing time. In particular, our Pirex offers specific desirable properties as follows:

- **Minimal client (inbound) bandwidth**: The most critical and desirable property of Pirex is that it offers a minimal client inbound bandwidth overhead, which is *independent* of the remote database entry size. Specifically, to privately read an entry, Pirex only requires the client to download an amount of data that is equal to eight database entries. This cost is asymptotically (and concretely) much lower than most state-of-the-art OO-PIR schemes (e.g., [35, 60], which transmits a square-root number of entries. The total client bandwidth cost of Pirex is only $O(B + \sqrt{N}\log N)$ compared to $O(\sqrt{N}(\lambda + B))$ in other schemes [35, 60], with N, B, and λ are the number of database entries, the entry size, and the security parameter, respectively.
- **Low client storage overhead:** We introduce Pirex+, an extended Pirex scheme that features minimal client storage. In Pirex+, the client only stores $O(\lambda^2\sqrt{N})$bits of precomputed data that is *independent* of the entry size. Thus, it requires much lower client storage than prior OO-PIR schemes that require either $O(\lambda B\sqrt{N})$ or $O(\lambda B\sqrt{N}\log N)$. Concretely, for 1 TB database with 256 KB entries, Pirex+ requires 709 KB client storage, compared with 11.5 GB-1.3 TB in prior OO-PIR schemes [20, 35, 60] (i.e., four to six orders of magnitude smaller). Therefore, Pirex+ is desirable for clients with limited storage (e.g., mobile).
- **Sublinear computational complexity:** Pirex retains the sublinear computation efficiency as in state-of-the-art OO-PIR schemes. The server only performs $O(B\sqrt{N})$ low-cost operations (e.g., XOR, modular addition). Also, the client computation is efficient, where it only invokes a sublinear number of PRF evaluations and some XOR operations.
- **Extremely low end-to-end delay:** Thanks to the asymptotic communication and computation properties it achieves, Pirex offers a concretely low end-to-end delay for public database access. The Pirex design is also simple, where it only requires simple cryptographic operations (e.g., XOR, PRF invocations, modular arithmetics). Under real-world settings, Pirex is up to two orders of magnitudes faster, where it only takes 0.8s to privately read a 256-KB entry in 1 TB database compared with 30s-200s in other schemes (see §7 for more comprehensive experiments).
- **Technique: Private Partition Retrieval.** An important building block of Pirex is the design of a Private Partition Retrieval (PPR) protocol, which permits private retrieval of an arbitrary entry in a partitioned database in sublinear time. We developed a concrete PPR instantiation and formally proved that it achieves the desired security.

Table 1 summarize the performance of our Pirex framework compared to state-of-the-art OO-PIR schemes. We analyzed the security of our proposed techniques and rigorously prove that they satisfy the standard PIR security definition. We fully implemented all the schemes in our framework and intensively evaluated their performance on commodity hardware. Experimental results showed that our proposed schemes significantly outperforms state-of-the-art approaches in all online metrics, especially in the context of large databases with large entry sizes. Our implementation source code is publicly available at https://anonymous.4open.science/r/pirex.

1.2 Technical Highlights

Pirex relies on an elegant OO-PIR design blueprint proposed by Corrigan-Gibbs et al. in [20] (we call it CK20 for brevity). We briefly present the high-level idea of their scheme, along with some follow-up attempts (i.e., TreePIR [35]) to improve the OO-PIR performance. We then outline the limitations of these works and present our ideas to address such drawbacks.

CK20 [20]. It 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 $\mathcal{H} = (h_1, \ldots, h_M)$, where each hint $h_i = (\mathcal{S}_i, \rho_i)$ contains a set of indices $\mathcal{S}_i = (s_0^{(i)}, \ldots, s_{\sqrt{N}-1}^{(i)})$, $s_j^{(i)} \xleftarrow{\$} [N]$ and an offline 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, storing $\mathcal{H}$ takes $O(N \log^2 N)$ in space as it takes $O(\sqrt{N} \log N)$ 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 is the entry size. The offline bandwidth cost is $O(M(\lambda + B))$ as M keys are sent to receive M parities.

To retrieve a desired 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 $\mathsf{DB}[x] = \hat{\rho} \bigoplus \rho_i$. Since $\mathcal{S}_i$ is partially exposed to both servers, the hint h_i needs to be replaced with new 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 then computed by sending $\hat{\mathcal{S}}' = \mathcal{S}' \setminus \{x\}$ to the Left server to obtain $\hat{\rho}' = \bigoplus_{j=0}^{\sqrt{N}-2} \mathsf{DB}[\hat{s}'_j]$. Notice that $\hat{\mathcal{S}}$ or $\hat{\mathcal{S}}'$ is always created by puncturing out the desired index x. By receiving $\hat{\mathcal{S}}$ or $\hat{\mathcal{S}}'$, the servers certainly learn that the entry being privately retrieved by the client is *not* the one in $\hat{\mathcal{S}}$ or $\hat{\mathcal{S}}'$, thereby violating PIR security. Therefore, the client performs a probabilistic puncture where a random index $x' \neq x$ is removed with probability $\frac{(\sqrt{N}-1)}{N}$, which results in non-negligible failure probability. To ensure correctness, the client executes $O(\lambda)$ protocol instances in parallel. Note that there exists a trade-off in this online protocol, where the client overhead depends on if the sets are represented by PRF or PRP keys. For PRF keys, since the PRF outputs are random in $[N]$, it takes $O(M\sqrt{N})$ to find a 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 client lookup time is reduced to $O(M \log N)$, at the cost of $O(\sqrt{N} \log N)$ outbound bandwidth since PRP is not puncturable.

TreePIR [35]. To reduce the client outbound bandwidth and lookup time to $O(\lambda \log N)$ and $O(M)$, respectively, Lazzaretti and Papamanthou proposed a *partition* technique combined with puncturable PRF. DB is divided into $\sqrt{N}$ partitions P_j, $j \in [\sqrt{N}]$ and P_j covers the range $\{j\sqrt{N}, \ldots, (j+1)\sqrt{N} - 1\}$. Each set of indices $\mathcal{S}_i = (s_0^{(i)}, \ldots, s_{\sqrt{N}-1}^{(i)})$ is generated by PRF on the output domain $[\sqrt{N}]$. The idea is to sample one index per partition using a random offset returned by PRF, so that when checking for an index, the client only needs to look at a specific partition. To do this, the client creates each $\mathcal{S}_i$ using an offset vector $\mathbf{\Delta}_i = (\delta_0^{(i)}, \ldots, \delta_{\sqrt{N}-1}^{(i)})$, with $\delta_j^{(i)} \leftarrow \mathsf{PRF}(sk_i, j)$, to compute $s_j^{(i)} = j\sqrt{N} + \delta_j^{(i)}$. The offline parity ρ_i is obtained by sending sk_i to the Left server, where $\rho_i = \bigoplus_{j=0}^{\sqrt{N}-1} \mathsf{DB}[s_j^{(i)}]$. In the online, to find which $\mathcal{S}_i$ contains index x in $O(M)$, the client computes $k = \lfloor \frac{x}{\sqrt{N}} \rfloor$, and uses the key sk_i to check if $x = k\sqrt{N} + \mathsf{PRF}(sk_i, k)$. To recover $\mathsf{DB}[x]$, the client needs the parity of $\mathcal{S}_i \setminus \{x\}$, which is obtained by sending a punctured vector $\mathbf{\Delta} = (\delta_0, \ldots, \delta_{\sqrt{N}-2})$ to the Right server, represented by a punctured key of size $O(\lambda \log N)$ derived from sk_i. Since $\mathbf{\Delta}$ has $\sqrt{N} - 1$ offsets, there are $\sqrt{N}$ possible partitions of x each corresponding to $\mathbf{\Delta}_{j^*} = (\delta_0, \ldots, \delta_{j-1}, \perp, \delta_{j+1}, \ldots, \delta_{\sqrt{N}-1})$. The Right server computes $\hat{\rho}_{j^*} = \bigoplus_{j \neq j^*}^{\sqrt{N}-1} \mathsf{DB}[j\sqrt{N} + \delta_j]$, for each possible $j^* \in [\sqrt{N}]$, where $\hat{\rho}_k$ is the punctured parity the client needs to recover $\mathsf{DB}[x] = \hat{\rho}_k \oplus \rho_i$. To privately retrieve $\hat{\rho}_k$, the client can download all $\sqrt{N}$ values, or execute another single-server PIR instance on them (which can be costly). To refresh the hint, the client samples a new sk' where $x - k\sqrt{N} = \mathsf{PRF}(sk', k)$, then sends the punctured key to the Left server to obtain $\hat{\rho}'_k$ and compute a new offline parity $\rho' = \mathsf{DB}[x] \oplus \hat{\rho}'_k$. Note that query privacy is guaranteed because the two servers, by observing a vector $\mathbf{\Delta}$ of $\sqrt{N} - 1$ random offsets, only know the partition of x with $\frac{1}{\sqrt{N}}$ probability. Thus, the scheme only needs one-time execution, instead of $O(\lambda)$ times as CK20. However, the client's inbound bandwidth increases to $O(B\sqrt{N})$ due to $\sqrt{N}$ parities transmission.

Idea 1: Patch the punctured set with a random offset. To reduce the client's inbound bandwidth while retaining the efficient lookup time, we propose a novel *patching* trick to support the puncturable set and partition composition. We observe that although partition offers efficient lookup, it incurs high client bandwidth because the punctured set removes one offset from a hidden partition, which requires transmitting $\sqrt{N}$ possible cases to hide that partition. Thus, our idea is to patch the punctured set with a random offset selected for the hidden partition as $\mathbf{\Delta} = (\delta_0, \ldots, \delta_{j-1}, s, \delta_{j+1}, \ldots, \delta_{\sqrt{N}-1})$, where $s \xleftarrow{\$} \{0, \ldots, \sqrt{N} - 1\}$. In this case, the server replies to the client with a single patched parity $\bar{\rho} = \left(\bigoplus_j \mathsf{DB}[j\sqrt{N} + \delta_j]\right) \oplus \mathsf{DB}[s + j'\sqrt{N}]$, where j' is the index of s in $\mathbf{\Delta}$. While this strategy reduces the client bandwidth to $O(1)$, it also impacts the reconstruction correctness because the client will obtain $\bar{\rho} \oplus \rho_i = \mathsf{DB}[x] \oplus \mathsf{DB}[s]$, instead of $\mathsf{DB}[x]$.

Idea 2: Retrieve the random patching entry via private partition retrieval. To address the reconstruction correctness issue due to patching, we need to somehow privately read $\mathsf{DB}[s]$ to compute $\bar{\rho} \oplus \hat{\rho}_i \oplus \mathsf{DB}[s] = \mathsf{DB}[x]$. As there are only

$\sqrt{N}$ partitions and the offset value s in $\mathbf{\Delta}$ is not important (as long as it is random since it is only used to hide the index of the punctured partition), we can treat $\sqrt{N}$ partitions as a (logical) database of size $\sqrt{N}$ and then employ a standard XOR-PIR protocol (e.g., [17]) to privately access (an arbitrary entry in) the punctured partition. Intuitively, suppose the index of the punctured partition is k and it is represented by a unit vector $\mathbf{e} \in \{0,1\}^{\sqrt{N}}$ s.t. $\mathbf{e}[k] = 1$. Let $\mathbf{v}_0, \mathbf{v}_1$ be two random binary vectors so that $\mathbf{v}_0 \oplus \mathbf{v}_1 = \mathbf{e}$. The client sends a set of indices $\mathcal{T}_l = \{j\sqrt{N} + s : v_l[j] = 1, j \in [\sqrt{N}]\}$ to each server S_l and receives a corresponding response $w_l = \bigoplus_{\gamma \in \mathcal{T}_l} \mathsf{DB}[\gamma]$, thereby obtaining $\mathsf{DB}[s] = w_0 \oplus w_1$. Note that we simplified how the queries $\mathcal{T}_l$ are created to highlight the key idea of using standard XOR-PIR to privately obtain $\mathsf{DB}[s]$. In fact, $\mathcal{T}_i$ must be created more carefully in a way that when combined with the query $\mathbf{\Delta}$ in Idea 1, it should not leak any information about what partition was punctured, thereby violating PIR security (see §5.2.2 for details). As standard XOR-PIR incurs linear processing (w.r.t the database of size $\sqrt{N}$ in this case), the additional server cost is $O(\sqrt{N})$, and hence, this strategy does not asymptotically increase the complexity of OO-PIR's online query protocol overall. Note that the refresh procedure in OO-PIR follows the similar idea to reduce the inbound bandwidth, since it also involves sending a punctured set, which can make uses of our patching technique.

Idea 3: Remote parities storage via oblivious write and additive homomorphic encryption. OO-PIR paradigm (e.g., [19, 20, 32, 34, 35, 54, 60, 61]) requires the client to maintain at least $\Omega(\lambda\sqrt{N})$ parities ρ_i computed in offline, where each ρ_i has an equal size to the database entry. To reduce client storage, we propose to store ρ_i (under IND-CPA encrypted) remotely on the database server. Since the number of ρ_i is sublinear, we can utilize a standard XOR-PIR protocol [17] to privately read it when needed 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 paradigm. To refresh securely, we develop an oblivious refresh buffer based on [51], which temporarily stores refresh parities and obliviously merges them with the original parities over time. Another challenge when maintaining the parities remotely is that they have to be updated when there is a change in the public database. While private database update is not captured in PIR security model, the parities must be updated privately because they are individually formed by aggregating a set of random database entries together. If an entry changes and the affected parities are updated insecurely, the server will learn the index distribution per set, thereby compromising the privacy of future client's online queries. To obliviously update parities, we use Additive Homomorphic Encryption (AHE) to create an encrypted updated vector (i.e., a binary vector with elements 1 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, we can see that the client's inbound bandwidth now only contains a single patched parity $\bar{\rho}$ and a random patching entry $\mathsf{DB}[s]$ from the punctured partition, obtained by using standard XOR-PIR, and therefore, the cost is minimal. At the high level, we can also see that our OO-PIR design is extremely simple and computationally efficient, where it only incurs simple XOR operations. We present two OO-PIR schemes. The first scheme (called Pirex) combines the first two ideas, while the second scheme (called Pirex+) is an extension that combines all three ideas together. 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 server computation (arithmetic operations instead of XOR, and an extra of λ factor in the complexity due to the number of offline parity).

2 PRELIMINARIES

Notation. $[n]$ denotes $\{0,1,\ldots,n-1\}$. λ denotes security parameters and $\mathsf{negl}(\cdot)$ stands for the negligible function. $x \xleftarrow{\$} [n]$ indicates that x is chosen randomly from the set $[n]$. 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 $\ominus$ as the symmetric difference of two sets $\mathcal{A}$ and $\mathcal{B}$ such that $\mathcal{A} \ominus \mathcal{B} = (\mathcal{A} \cup \mathcal{B}) \setminus (\mathcal{A} \cap \mathcal{B})$. We denote $\mathsf{PRF} : \mathcal{K} \times [n] \to [n]$ as the pseudorandom function (PRF), where $y \leftarrow \mathsf{PRF}(sk, s)$ outputs a pseudorandom bit string $y \in [n]$ given a PRF key $sk \in \mathcal{K}$ and a seed $s \in [n]$. We denote $\mathbb{Z}_n$ as the finite 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 $\langle 1 \rangle$.

2.1 Pseudorandom Set

Pseudorandom Set (PRS) [20] permits the generation and representation of a set with s (pseudorandom) elements using a set key sk. A PRS contains the following PPT algorithms:

- $\underline{sk \leftarrow \mathsf{PRS.Gen}(1^\lambda, n)}$: Given a security parameter λ and a range n for the elements, it outputs a set key $sk \in \mathcal{K}$.
- $\underline{\mathcal{S} \leftarrow \mathsf{PRS.Eval}(sk, s)}$: Given a set key sk, it outputs a set $\mathcal{S}$ with s elements.

A PRS scheme is secure (w.r.t pseudorandomness) if given security parameter λ, range n and $sk \leftarrow \mathsf{PRS.Gen}(1^\lambda, n)$, the algorithm $\mathsf{PRS.Eval}(sk, s)$ outputs an associated set $\mathcal{S} \in [n]^s$ that is computationally indistinguishable from $\mathcal{S}' \xleftarrow{\$} [n]^s$.

PRS can be constructed from PRF [20]. Specifically, the algorithm $\mathsf{PRS.Gen}(1^\lambda, n)$ generates and outputs a PRF key sk for a PRF function: $\mathcal{K} \times [n] \to [n]$. The algorithm $\mathsf{PRS.Eval}(sk, s)$ invokes and outputs PRF evaluations at s points as $\mathcal{S} = \{\mathsf{PRF}(sk, 0), \ldots, \mathsf{PRF}(sk, s-1)\}$.

3 MODELS

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. The servers allow the client to access an arbitrary entry in DB. Our system is a two-server OO-PIR scheme defined as follows.

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

- $\underline{\mathcal{H} \leftarrow \mathsf{Prep}(\mathsf{DB}, N)}$*: Given a database* DB *and the number of entries N, it outputs a hint* $\mathcal{H}$*.*
- $\underline{(Q_0, Q_1, \mathcal{H}^*) \leftarrow \mathsf{Query}(x, \mathcal{H})}$*: 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* $\mathcal{H}^*$*.*
- $\underline{\mathcal{R}_i \leftarrow \mathsf{Answer}(Q_i, \mathsf{DB})}$*: Given an online query* Q_i *and the database* DB*, it outputs a response* $\mathcal{R}_i$*.*
- $\underline{(b_x, \mathcal{H}') \leftarrow \mathsf{Recover}(\mathcal{R}_0, \mathcal{R}_1, \mathcal{H}^*)}$*: Given the hint* $\mathcal{H}^*$ *and two responses* $\mathcal{R}_0$ *and* $\mathcal{R}_1$*, it outputs the desired data entry* b_x *and an updated hint* $\mathcal{H}'$*.*

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

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

Threat model. The client is trusted and the two servers are semi-honest, meaning that they follow the protocol but are curious about which database entry is being retrieved by the client. We assume the two servers do not collude. We consider static corruption, where an adversary $\mathcal{A}$ can corrupt either server S_0 or S_1 but can not adaptively switch between two servers during protocol execution.

Security model. We define the security of our system using the Ideal/Real simulation 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 client's query honestly. Let $\mathcal{Z}$ be the environment that provides inputs for the client and receives corresponding outputs. $\mathcal{Z}$ can also get the view of the adversary at any time. We define the Ideal world and Real world as follows:

- Ideal: In the offline, $\mathcal{Z}$ provides a database DB to the client. The client sends DB to $\mathcal{F}$, which notifies the simulator $\mathcal{S}$ about the size of DB (but not the content). If $\mathcal{S}$ says ok, $\mathcal{F}$ returns ok to the client. For each online query, $\mathcal{Z}$ specifies an entry index x for the client. The client forwards x to $\mathcal{F}$. $\mathcal{F}$ notifies the simulator $\mathcal{S}$ about the query event (but not x). If $\mathcal{S}$ says ok, $\mathcal{F}$ returns $\mathsf{DB}[x]$. The client forwards $\mathsf{DB}[x]$ to $\mathcal{Z}$. Otherwise, $\mathcal{F}$ returns $\perp$ and the client forwards $\perp$ to $\mathcal{Z}$.
- Real: In the offline, $\mathcal{Z}$ provides a database DB to the client. The client honestly execute the $\mathsf{Prep}(\mathsf{DB})$ algorithm with two servers to obtain the hint $\mathcal{H}$. For each online query, $\mathcal{Z}$ specifies an entry index x to the client. The client honestly executes $(Q_0, Q_1, \mathcal{H}') \leftarrow \mathsf{Query}(x, \mathcal{H})$ on the input x. The client sends Q_0 to server S_0 and Q_1 to server S_1. Each server S_i executes $\mathcal{R}_i \leftarrow \mathsf{Answer}(Q_i, \mathsf{DB})$ and returns response $\mathcal{R}_i$. The client executes $(b_x, \mathcal{H}') \leftarrow \mathsf{Recover}(\mathcal{R}_0, \mathcal{R}_1, \mathcal{H})$ to get the entry b_x and forwards b_x to $\mathcal{Z}$.

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

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

4 PRIVATE PARTITION RETRIEVAL

We present Private Partition Retrieval (PPR), a technique that allows the client to privately read (an arbitrary data entry from) a partition containing m entries in a n-partitioned database DB, without revealing which partition is of interest.

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

- $\underline{(\mathcal{T}_0, \mathcal{T}_1) \leftarrow \mathsf{Gen}(k)}$*: Given a partition index* $k \in [n]$*, it outputs two partition queries* $\mathcal{T}_0$*,* $\mathcal{T}_1$*.*
- $\underline{r_i \leftarrow \mathsf{Ret}(\mathcal{T}_i, \mathsf{DB})}$*: Given a query* $\mathcal{T}_i$ *and the partitioned database* DB*, it outputs a response* r_i*.*
- $\underline{b \leftarrow \mathsf{Rec}(r_0, r_1)}$*: Given two responses* r_0 *and* r_1*, it outputs an arbitrary data entry b from partition k.*

We define the PPR security using the Ideal/Real simulation paradigm such that an adversary $\mathcal{A}$ statically corrupting one server learns nothing about the partition being read. Let $\mathcal{F}_P$ be the ideal functionality that honestly returns an arbitrary entry from the client-chosen partition. Let $\mathcal{Z}$ be the environment that provides inputs for the client and receives corresponding outputs. We define the Ideal world and Real world as follows:

- Ideal: In the setup, $\mathcal{Z}$ provides a database DB to the client. The client sends DB to $\mathcal{F}_P$. $\mathcal{F}_P$ notifies the simulator $\mathcal{S}_P$ about the partition size of DB (but not the content). If $\mathcal{S}$ says ok, $\mathcal{F}_P$ returns ok to the client. For each read access, $\mathcal{Z}$ specifies a partition number k. The client forwards k to $\mathcal{F}_P$. $\mathcal{F}_P$ then notifies the simulator $\mathcal{S}_P$ about the event (but not the partition range). If $\mathcal{S}_P$ says ok, $\mathcal{F}_P$ returns an arbitrary b from partition k. Otherwise, $\mathcal{F}_P$ returns $\perp$ to the client.
- Real: In the setup, $\mathcal{Z}$ provides a database to the client. The client sends DB to two servers. For each access, $\mathcal{Z}$ specifies a partition index k. The client executes $(Q_0, Q_1) \leftarrow \mathsf{Gen}(k)$ algorithm honestly on input k. The client sends Q_0 to server S_0 and Q_1 to server S_1. Server S_i executes $r_i \leftarrow \mathsf{Ret}(Q_i, \mathsf{DB})$ and return response r_i. The client executes $b \leftarrow \mathsf{Rec}(r_0, r_1)$ to get an arbitrary entry b and forwards b to $\mathcal{Z}$.

```
• $(\mathcal{T}_0, \mathcal{T}_1) \leftarrow \mathsf{PPR.Gen}(k)$:
1: $(\delta_0, ..., \delta_{n-1}) \xleftarrow{\$} [m]^n$
2: $(e_0, ..., e_{n-1}) \xleftarrow{\$} \{0,1\}^n$
3: $(e'_0, ..., e'_{n-1}) \leftarrow (e_1, ..., e_n),\ e'_k = e'_k \oplus 1$
4: $\mathcal{T}_0 \leftarrow \{e_i \cdot (i \cdot m + \delta_i)\ \forall\, i \in [n]\}$
5: $\mathcal{T}_1 \leftarrow \{e'_i \cdot (i \cdot m + \delta_i)\ \forall\, i \in [n]\}$
6: return $(\mathcal{T}_0, \mathcal{T}_1)$

• $r_i \leftarrow \mathsf{PPR.Ret}(\mathcal{T}_i, \mathsf{DB})$:
1: parse $\mathcal{T}_i = \{p_1^{(i)}, \ldots, p_{n'}^{(i)}\}$
2: return $r_i \leftarrow \bigoplus_{j=1}^{n'} \mathsf{DB}[p_j^{(i)}]$

• $b \leftarrow \mathsf{PPR.Rec}(r_0, r_1)$:
1: return $b \leftarrow r_0 \oplus r_1$
```

Figure 1: Our proposed PPR scheme.

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

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

We present a concrete PPR scheme in Figure 1.

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

Proof. See Appendix §A.1. □

5 The Proposed Pirex Scheme

5.1 Data Structures

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

- The database DB is an array of N entries and is divided into n partitions. Each 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 assume $m = n = \sqrt{N}$.
- The buffer $\mathcal{H}$ consists of M entries to store the precomputed hints. Each hint h_i is a tuple (ℓ_i, sk_i, ρ_i), where $sk_i \in \{0,1\}^\lambda$ is a PRS key that represents a set of indices $\mathcal{S}_i = (s_1, \ldots, s_n)$, $\rho_i = \bigoplus_{j=1}^{n} \mathsf{DB}[s_j]$ is an offline parity, and $\ell_i \in \{0,1\}$ is the identifier of the server that computed ρ_i.

5.2 Protocol Details

5.2.1 Offline Phase

Figure 2 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 offsets that has a corresponding offline parity ρ_i. To do this, the client samples M PRS keys $(sk_1, \ldots, sk_M)$, then sends each key sk_i to a random server $\mathsf{S}_{\ell_i} \in \{\mathsf{S}_0, \mathsf{S}_1\}$. This is because in our online phase, the client sends two queries to two servers to privately retrieve a data entry, where one server receives a real query, while the other server receives a dummy query (if it is the one that created the offline hint for that desired online entry previously). Thus, selecting a random server to compute the hint in the offline phase ensures each server can not distinguish whether it receives the real or dummy query in the online phase. Upon receiving each key sk_i, server S_{ℓ_i} generates the set $\mathcal{S}_i = (s_0, \ldots, s_{n-1})$, where each offset s_j is in partition P_j and $s_j = (j \cdot m) + \mathsf{PRS}(sk_i, j)$ (line 4). Given the set $\mathcal{S}_i$, server S_{ℓ_i} computes and returns an offline parity $\rho_i \leftarrow \bigoplus_{j=1}^{n} \mathsf{DB}[s_j]$ (line 5). On receiving M offline parities, the client finalizes the set of hints $\mathcal{H}$, where hint $h_i = (\ell_i, sk_i, \rho_i)$ reflects that server S_{ℓ_i} used the key sk_i to compute the offline parity ρ_i (lines 6-7).

```
• $\mathcal{H} \leftarrow \mathsf{Prep}(\mathsf{DB}, N)$:
1: for $i = 1$ to $M$ do
2:     $\ell_i \xleftarrow{\$} \{0,1\}$
3:     $sk_i \leftarrow \mathsf{PRS.Gen}(1^\lambda)$
4:     $\{s_0, ..., s_{n-1}\} \leftarrow \mathsf{PRS.Eval}(sk_i)$                 } executed by server $\mathsf{S}_{\ell_i}$
5:     $\rho_i \leftarrow \bigoplus_{j=0}^{n-1} \mathsf{DB}[j\sqrt{N} + s_j]$     }
6:     $h_i \leftarrow (\ell_i, sk_i, \rho_i)$
7: return $\mathcal{H} \leftarrow (h_1, \ldots, h_M)$
```

Figure 2: Pirex - Offline Phase.

5.2.2 Online Phase

To privately retrieve a data 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. The algorithm performs two actions. It first creates two data queries $\hat{Q}_0$ and $\hat{Q}_1$ such that when combining the responses with a particular hint $h_i \in \mathcal{H}$, $\mathsf{DB}[x]$ will be recovered (lines 2-4). It then creates two refresh queries $\hat{Q}'_0$ and $\hat{Q}'_1$ that permits the client to refresh h_i by creating a new hint h' in place of h_i (lines 5-7). This is because after h_i is used to recover $\mathsf{DB}[x]$, it cannot be reused for the security of future queries.

To search for the hint $h_i = (\ell_i, sk_i, \rho_i)$, the client computes the partition $k = \lfloor \frac{x}{m} \rfloor$ and checks if $x \overset{?}{=} k\sqrt{N} + \mathsf{PRF}(sk_i, k)$. To recover $\mathsf{DB}[x]$, the client needs a punctured parity $\hat{\rho}_i$ so that $\mathsf{DB}[x] = \rho_i \oplus \hat{\rho}_i$. Since $\rho_i = \bigoplus_{j=1}^{n} \mathsf{DB}[s_j]$, for $s_j \in \mathcal{S}_i$ generated by sk_i, this only holds if $\hat{\rho}_i = \bigoplus_{j=1}^{n-1} \mathsf{DB}[\hat{s}_j]$, with $\hat{s}_j \in \hat{\mathcal{S}}$ and $\hat{\mathcal{S}} = \mathcal{S}_i \setminus \{x\}$. However, sending the punctured set $\hat{\mathcal{S}}$ of $(n-1)$ offsets leaks the partition P_k, which permits the adversary to learn the range of x. To prevent this leakage, our idea is to patch $\hat{\mathcal{S}}$ with a random offset $z \in P_k$, such that the client can still obtain the punctured parity $\hat{\rho}_i$. This results in a patched set $\bar{\mathcal{S}} = \{\bar{s}_1, \ldots, \bar{s}_n\} = \hat{\mathcal{S}} \cup \{z\}$ which has a patched parity $\bar{\rho} = \bigoplus_{j=1}^{n} \mathsf{DB}[\bar{s}_j] = \hat{\rho}_i \oplus \mathsf{DB}[z]$. To obtain $\hat{\rho}_i$, the client needs the random patch $\mathsf{DB}[z]$. We will use our PPR protocol to privately read $\mathsf{DB}[z]$ without leaking the partition P_k.

```
• $(Q_0, Q_1, \mathcal{H}^*) \leftarrow \mathsf{Query}(x, \mathcal{H})$:
 1: parse $\mathcal{H} = (h_1, \ldots, h_M)$, $k \leftarrow \lfloor \frac{x}{m} \rfloor$
 2: Find $h_i = (\ell_i, sk_i, \rho_i) \in \mathcal{H}$ such that $x = k\sqrt{N} + \mathsf{PRF}(sk_i, k)$
 3: $\mathcal{S}_i \leftarrow \mathsf{PRS.Set}(sk_i)$
 4: $(\hat{Q}_0, \hat{Q}_1) \leftarrow \mathsf{SubQuery}(x, \ell_i, \mathcal{S}_i)$
 5: $sk' \leftarrow \mathsf{PRS.Gen}(1^\lambda)$ where $x = k\sqrt{N} + \mathsf{PRF}(sk', k)$
 6: $\mathcal{S}' \leftarrow \mathsf{PRS.Set}(sk')$ and $\ell' \xleftarrow{\$} \{0,1\}$
 7: $(\hat{Q}'_0, \hat{Q}'_1) \leftarrow \mathsf{SubQuery}(x, \neg\ell', \mathcal{S}')$
 8: $Q_0 \leftarrow (\hat{Q}_0, \hat{Q}'_0)$, $Q_1 \leftarrow (\hat{Q}_1, \hat{Q}'_1)$
 9: $\mathcal{H}^* \leftarrow (h_1, \ldots, h_i, \ldots, h_M, h')$ where $h' \leftarrow (\ell', sk', \perp)$
10: return $(Q_0, Q_1, \mathcal{H}^*)$

• $(\hat{Q}_0, \hat{Q}_1) \leftarrow \mathsf{SubQuery}(x, \ell, \mathcal{S})$:
 1: $(\mathcal{T}^{(z)}, \mathcal{T}) \leftarrow \mathsf{PPR.Gen}(k)$ where $k \leftarrow \lfloor \frac{x}{m} \rfloor$
 2: $\bar{\mathcal{S}} \leftarrow \mathcal{S} \setminus \{x\} \cup \{z\}$ where $\{z\} \leftarrow \mathcal{T}^{(z)} \ominus \mathcal{T}$
 3: $\tilde{\mathcal{S}} \leftarrow \{i \cdot m + \delta_i \ \forall\, i \in [n]\}$ where $(\delta_0, \ldots, \delta_{n-1}) \xleftarrow{\$} [m]^n$
 4: $\hat{Q}_\ell \leftarrow (\tilde{\mathcal{S}}, \mathcal{T}^{(z)})$, $\hat{Q}_{\neg\ell} \leftarrow (\bar{\mathcal{S}}, \mathcal{T})$
 5: return $(\hat{Q}_0, \hat{Q}_1)$
```

Figure 3: Pirex - Online Phase: Query

```
• $\mathcal{R}_i \leftarrow \mathsf{Answer}(Q_i, \mathsf{DB})$:
 1: parse $Q_i = ((\mathcal{S}, \mathcal{T}), (\mathcal{S}', \mathcal{T}'))$
 2: $\bar{\rho} \leftarrow \bigoplus_{j=1}^{n} \mathsf{DB}[s_j]$ for all $s_j \in \mathcal{S}$
 3: $\bar{\rho}' \leftarrow \bigoplus_{j=1}^{n} \mathsf{DB}[s'_j]$ for all $s'_j \in \mathcal{S}'$
 4: $w \leftarrow \mathsf{PPR.Ret}(\mathcal{T}, \mathsf{DB})$
 5: $w' \leftarrow \mathsf{PPR.Ret}(\mathcal{T}', \mathsf{DB})$
 6: return $\mathcal{R}_i \leftarrow ((\bar{\rho}, w), (\bar{\rho}', w'))$
```

Figure 4: Pirex - Online Phase: Answer

To patch $\hat{\mathcal{S}}$ so that $\bar{\rho}$ and $\mathsf{DB}[z]$ can be obtained, the client invokes the $\mathsf{SubQuery}$ algorithm. The client computes two partition sets $(\mathcal{T}^{(z)}, \mathcal{T}) \leftarrow \mathsf{PPR.Gen}(k)$ (line 1), where $z \in \mathcal{T}^{(z)}$ and $\mathcal{T}^{(z)} \ominus \mathcal{T} = \{z\}$. By PPR correctness, sending $\mathcal{T}^{(z)}$ and $\mathcal{T}$ to the servers permits private retrieval of $\mathsf{DB}[z]$. To obtain $\bar{\rho}$, the client needs to send the patched set $\bar{\mathcal{S}} = \hat{\mathcal{S}} \cup \{z\}$ to a server. We show how to distribute three sets $\mathcal{T}^{(z)}, \mathcal{T}$ and $\bar{\mathcal{S}}$ to two servers. Remark that for hint h_i, the identifier $\ell_i \in \{0,1\}$ reflects that server S_{ℓ_i} knew the set $\mathcal{S}_i$ that was used to create the offline parity ρ_i. For security, the client must send the patched set $\bar{\mathcal{S}}$ to the other server $\mathsf{S}_{\neg\ell_i}$. Since $z \in \bar{\mathcal{S}}$, it is vital to ensure $\mathsf{S}_{\neg\ell_i}$ receives $\mathcal{T}$ since there is a common z in $\bar{\mathcal{S}}$ and $\mathcal{T}^{(z)}$. In this case, server $\mathsf{S}_{\neg\ell_i}$ receives $(\bar{\mathcal{S}}, \mathcal{T})$ and server S_{ℓ_i} receives $\mathcal{T}^{(z)}$. However, the server can distinguish if it receives $(\bar{\mathcal{S}}, \mathcal{T})$ or $\mathcal{T}^{(z)}$, which leaks information about z that indicates the partition P_k. Thus, the client creates a random set $\tilde{\mathcal{S}}$ to send along with $\mathcal{T}^{(z)}$, which makes $(\bar{\mathcal{S}}, \mathcal{T})$ and $(\tilde{\mathcal{S}}, \mathcal{T}^{(z)})$ indistinguishable. To this end, the queries to servers $S_{\neg\ell_i}$ and S_{ℓ_i} are $\hat{Q}_{\neg\ell_i} = (\bar{\mathcal{S}}, \mathcal{T})$ and $\hat{Q}_{\ell_i} = (\tilde{\mathcal{S}}, \mathcal{T}^{(z)})$, respectively.

To create the new hint h', the client samples a new PRS key sk' such that $x \in \mathcal{S}'$, with $\mathcal{S}' = (s'_1, \ldots, s'_n)$ is generated by the key sk'. This is to preserve the distribution of the set $\mathcal{H}$, since h_i was consumed subject to recovering $\mathsf{DB}[x]$. To create a new offline parity $\rho' = \bigoplus_{j=1}^{n} \mathsf{DB}[s'_j]$, the client needs a punctured parity $\hat{\rho}'$ so that $\rho' = \hat{\rho}' \oplus \mathsf{DB}[x]$. Since $\mathsf{DB}[x]$ will be obtained from the data queries ($\hat{Q}_{\ell_i}$ and $\hat{Q}_{\neg\ell_i}$), the above only holds if $\hat{\rho}' = \bigoplus_{j=1}^{n} \mathsf{DB}[\hat{s}'_j]$ with $\hat{s}'_j \in \hat{\mathcal{S}}'$ and $\hat{\mathcal{S}}' = \mathcal{S}' \setminus \{x\}$. To obtain $\hat{\rho}'$, the client needs to send the punctured set $\hat{\mathcal{S}}'$ to the server. Since this is similar to sending $\hat{\mathcal{S}}$ above, the client invokes $\mathsf{SubQuery}$. It creates two refresh queries that contain a patched set $\bar{\mathcal{S}}'$ and two partition sets, which permits the client to obtain $\hat{\rho}'$. Remark that similar to the offline phase, the set $\bar{\mathcal{S}}'$ must be sent to a random server $\mathsf{S}_{\ell'} \in \{\mathsf{S}_0, \mathsf{S}_1\}$ when computing the new hint. Thus, the refresh queries are $\hat{Q}'_{\ell'}$ and $\hat{Q}'_{\neg\ell'}$ such that $\bar{\mathcal{S}}' \in \hat{Q}'_{\ell'}$. To this end, the client distributes the data queries $\hat{Q}_{\ell_i}$, $\hat{Q}_{\neg\ell_i}$ and the refresh queries $\hat{Q}'_{\ell'}$, $\hat{Q}'_{\neg\ell'}$ to the corresponding server.

On receiving a query $Q_i = ((\mathcal{S}, \mathcal{T}), (\mathcal{S}', \mathcal{T}'))$, each server $\mathsf{S}_i \in \{\mathsf{S}_0, \mathsf{S}_1\}$ parses the sets $\mathcal{S}$ and $\mathcal{S}'$ to compute the patched parity $\bar{\rho}$ and $\bar{\rho}'$ respectively (lines 1-3). For partition sets $\mathcal{T}$ and $\mathcal{T}'$, server S_i computes $w \leftarrow \mathsf{PPR.Ret}(\mathcal{T}, \mathsf{DB})$ and $w' \leftarrow \mathsf{PPR.Ret}(\mathcal{T}', \mathsf{DB})$ (lines 4-5). Server S_i then returns the final answer $\mathcal{R}_i = ((\bar{\rho}, w), (\bar{\rho}', w'))$ (line 5).

```
• $(b_x, \mathcal{H}') \leftarrow \mathsf{Recover}(\mathcal{R}_0, \mathcal{R}_1, \mathcal{H}^*)$:
 1: parse $\mathcal{R}_0 = ((\bar{\rho}_0, w_0), (\bar{\rho}'_0, w'_0))$, $\mathcal{R}_1 = ((\bar{\rho}_1, w_1), (\bar{\rho}'_1, w'_1))$
 2: parse $\mathcal{H}^* = (h_1, \ldots, h_i, \ldots, h_M, h')$
Reconstruct:
 3: parse $h_i = (\ell_i, sk_i, \rho_i)$
 4: $b_x \leftarrow b \oplus \rho_i \oplus \bar{\rho}_{\neg\ell_i}$ where $b \leftarrow \mathsf{PPR.Rec}(w_0, w_1)$
Refresh:
 5: parse $h' = (\ell', sk', \perp)$
 6: $\rho' \leftarrow b' \oplus b_x \oplus \bar{\rho}'_{\ell'}$ where $b' \leftarrow \mathsf{PPR.Rec}(w'_0, w'_1)$
 7: $h' \leftarrow (\ell', sk', \rho')$
 8: $\mathcal{H}' \leftarrow (h_1, \ldots, h_{i-1}, h', h_{i+1}, \ldots, h_M)$
 9: return $(b_x, \mathcal{H}')$
```

Figure 5: Pirex - Online Phase: Recover

On receiving responses $\mathcal{R}_0$ and $\mathcal{R}_1$, the client can recover $\mathsf{DB}[x]$ using the tuples $(\bar{\rho}_0, w_0)$ and $(\bar{\rho}_1, w_1)$. The client first computes $b \leftarrow \mathsf{PPR.Rec}(w_0, w_1)$. Using hint $h_i = (\ell_i, sk_i, \rho_i)$, the client knows the punctured parity $\hat{\rho}_i$ must be derived from the patched parity $\bar{\rho}_{\neg\ell_i} \in \mathcal{R}_{\neg\ell_i}$ (since ρ_i was created by server S_{ℓ_i}). The client computes $\hat{\rho}_i = \bar{\rho}_{\neg\ell_i} \oplus b$, and retrieves $\mathsf{DB}[x] = \rho_i \oplus \hat{\rho}_i$. Similarly, to get a new offline parity ρ', the client uses the remaining tuples $(\bar{\rho}'_0, w'_0)$ and $(\bar{\rho}'_1, w'_1)$ from the responses. The client obtains another $b' = w'_0 \oplus w'_1$. Since h' contains the identifier ℓ' that reflects the patched parity $\bar{\rho}'_{\ell'}$ was created by server $\mathsf{S}_{\ell'}$, the client uses $\bar{\rho}'_{\ell'}$ to derive $\rho' = \mathsf{DB}[x] \oplus b' \oplus \bar{\rho}'_{\ell'}$. The client forms (ℓ', sk', ρ') as the new hint h' in place of the consumed h_i.

5.3 Analysis

We state the correctness and security of Pirex as follows.

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

Proof (sketch). To ensure correctness, the offline hints must cover all N database entries. Each offline hint is created by aggregating $\sqrt{N}$ random entries sampled from $\sqrt{N}$ partitions (one entry per partition). That means to cover all N entries, the offline hints must cover all $\sqrt{N}$ offsets in each partition of the database. Since all partitions are independent, we can apply the classic Coupon Collector Problem [42] to each partition to prove that the expected sampling number is $O(\sqrt{N}\log\sqrt{N})$ to cover $\sqrt{N}$ offsets within the partition (Lemma 2.10 in [42]). Therefore, in general, it suffices to set the number of hints as $M = O(\sqrt{N}\log N)$ to cover all offsets in all partitions, thereby N database entries. Lazzaretti et al. (Section 4.2 of [35]) proved that for large N, $M = O(\lambda\sqrt{N})$ suffices to cover all N database entries except with negligible failure probability. □

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

Proof. See Appendix §A.2 □

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

• Offline cost: For communication, the client sends $\lambda\sqrt{N}$ PRS keys to the servers for PRS representation and receives $\lambda\sqrt{N}$ parities correspondingly. As each PRS key is of size λ-bit and the parity size equals the data entry size, the client inbound (*resp.* outbound) bandwidth cost is $O(B\lambda\sqrt{N})$ (*resp.* $O(\lambda^2\sqrt{N})$). The total bandwidth is $O(\lambda\sqrt{N}(\lambda+B))$.

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

• Online cost: For each online retrieval, the client sends two queries (data and refresh) to each server. Each query contains a vector of $\sqrt{N}$ offsets and a set of $O(\sqrt{N})$ partition indices. As each offset/index can be represented by $O(\log N)$ bits, the client outbound bandwidth is $O(\sqrt{N}\log N)$. Each server responds to each client query with two aggregated results, each of size B-bit. Thus, the client inbound bandwidth cost is $O(B)$. The total bandwidth is $O(\sqrt{N}\log N + B)$.

For computation, the client performs at most $O(\lambda\sqrt{N})$ PRS evaluations, since there are $O(\lambda\sqrt{N})$ hints and each hint h_i needs $O(1)$ PRS evaluation and $O(1)$ comparison to check if h_i contains the desired entry index. To refresh, it also takes the client $O(1)$ PRS invocation to generate a new PRS key in place of the consumed one. To recover the desired entry (or refresh a new parity), the client incurs $O(1)$ XOR operation on three B-bit entries. Thus, the total client computation is $O(\lambda\sqrt{N})$ PRS evaluations and $O(1)$ XOR operations.

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

• Storage cost: Each server takes no extra cost besides the database storage, so the server storage is $O(NB)$. The client stores $\lambda\sqrt{N}$ precomputed hints, where each hint contains a λ-bit PRS key, a server indicator variable, and an offline B-bit parity. Thus, the total client storage is $O(\lambda\sqrt{N}(\lambda+B))$.

6 Reducing Client Storage

Although our Pirex offers an efficient bandwidth overhead that is independent of the data entry size, its client storage still depends on the entry size and, therefore, is significant. Specifically, the client storage cost is $O(\lambda\sqrt{N}(\lambda+B))$ since there are $M = O(\lambda\sqrt{N})$ hint entries, each contains a λ-bit PRS key and a B-bit parity. In this section, we propose Pirex+, an extended scheme based on Pirex that reduces the client storage cost with a slight impact on the overall complexity.

Remote parity storage. To reduce client storage, our idea is to maintain the offline parity components (ρ_i) on the server. In this context, we encrypt the parities using an IND-CPA encryption scheme to prevent the server from learning the private pseudorandom set from the parities in advance. Since there are only $O(\lambda\sqrt{N})$ offline parities, the standard 2-server XOR-PIR can be used to privately read the desired parity during the online phase of OO-PIR without incurring much extra overhead. Remark that the consumed hint also needs to be refreshed to maintain the distribution of the pseudorandom sets for future requests. Given part of the hints (parity) is stored remotely, the refresh operation must be performed obliviously for security.

Oblivious refresh. To perform oblivious refresh, we make use of oblivious write in [51]. 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 the database DB as in Pirex, each server maintains a replica of a $2M$-sized parity buffer $\mathbf{P} = (\mathbf{P}_{\text{left}}, \mathbf{P}_{\text{right}})$, where $\mathbf{P}_{\text{left}}$ 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 $h_i = (\ell_i, sk_i, \pi_i)$ contains a new component $\pi \in [2M]$, denoting 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 that the server maintains after offline processing. Each offline parity ρ_i corresponding to h_i is stored at $\mathbf{P}[h_i.\pi] = \mathbf{P}_{\text{left}}[i]$ for $i \in [M]$. In the online phase, suppose that a parity $\rho_i \in \mathbf{P}_{\text{left}}[i]$ corresponding to $h_i = (\ell_i, sk_i, \pi_i)$ is consumed to recover a desired data entry. Let ρ' be the refresh parity. To replace ρ_i, the idea is to have the client perform deterministic write operations on the two regions ($\mathbf{P}_{\text{left}}$, $\mathbf{P}_{\text{right}}$) of the buffer $\mathbf{P}$. Suppose the current refresh is the c-th refresh operation (mod M). The client writes ρ' to the c-th location in $\mathbf{P}_{\text{right}}$ as

$\mathbf{P}_{\mathsf{right}}[c] \leftarrow \rho'$, and stores its temporary location $\pi_i := c + M$.

Due to the round-robin schedule, $\mathbf{P}_{\mathsf{right}}$ can be full after M refreshes, which makes the next refresh operation overwrite some hints that were previously stored in $\mathbf{P}_{\mathsf{right}}$. To overcome this issue, we let the client perform another deterministic write on $\mathbf{P}_{\mathsf{left}}$, which obliviously transfers parities from $\mathbf{P}_{\mathsf{right}}$ to $\mathbf{P}_{\mathsf{left}}$. Specifically, at the c-th refresh round, the client also writes to $\mathbf{P}_{\mathsf{left}}[c]$ a parity corresponds to the hint $h_c = (\ell_c, sk_c, \pi_c)$. Such parity is located at either $\mathbf{P}_{\mathsf{left}}[c]$ or $\mathbf{P}[\pi_c] = \mathbf{P}_{\mathsf{right}}[\pi_c - M]$. Thus, the client performs a standard 2-server XOR-PIR to privately read the parity in $\mathbf{P}[\pi_c]$ without revealing its location and then write it to $\mathbf{P}_{\mathsf{left}}[c]$. To this end, the client updates its new location to $\pi_c := c$.

The above strategy ensures that, for every position in $\mathbf{P}_{\mathsf{right}}$, the refresh parity will be moved to $\mathbf{P}_{\mathsf{left}}$ before it is overwritten. This is because it will take M additional refresh operations to revisit the same position in $\mathbf{P}_{\mathsf{right}}$ again, which, by that time, all M positions in the $\mathbf{P}_{\mathsf{left}}$ have already been updated with the new parities. Thus, for any consumed parity $\rho_i \in \mathbf{P}_{\mathsf{left}}[i]$, it will eventually be replaced by a new one ρ' after M rounds.

Supporting database update. In real-world scenarios, the 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 components of the hints are 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 pseudorandom set that was used to construct the offline parities, thereby violating the security of OO-PIR, which only holds if the pseudorandom sets are revealed *once* to each server in the online phase.

To privately update the parities according to the database update, a simple method is to incorporate standard XOR-PIR and oblivious write similar to the oblivious refresh discussed above. However, unlike the refresh operation which updates only a single parity, a database update can require 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), in which the client can delegate oblivious update to the server.

Building block: Additive Homomorphic Encryption. AHE [21] permits the messages to be encrypted in a way that their ciphertexts can be homomorphically evaluated. Given a cyclic group $\mathbb{G}$ of order p, an AHE scheme over $\mathbb{G}$ contains the following PPT algorithms:

- $\underline{(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{AHE.Gen}(1^\lambda)}$: Given a security parameter λ, it outputs a pair of public and private keys $(\mathsf{pk}, \mathsf{sk})$.
- $\underline{\langle m \rangle \leftarrow \mathsf{AHE.Enc}(\mathsf{sk}, m)}$: Given a message $m \in \mathbb{Z}_p$ and a public key pk, it outputs a ciphertext $\langle m \rangle$.
- $\underline{m \leftarrow \mathsf{AHE.Dec}(\mathsf{sk}, \langle m \rangle)}$: 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 AHE as $\langle \mathbf{e} \rangle \leftarrow \mathsf{AHE.Enc}(\mathsf{pk}, \mathbf{e})$, where pk is its public key. Let $\langle \mathbf{p} \rangle = (\langle \rho_1 \rangle, \ldots, \langle \rho_{2M} \rangle)$ be the vector that contains 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 $\varepsilon = b - \mathsf{DB}[x]$. The server updates the parity buffer by evaluating the homomorphic addition and scalar multiplication as

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

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

Note that to fully incorporate AHE into Pirex+, we also make necessary changes to the algebraic operations when computing the offline parities (e.g., XOR to modular addition). Due to space constraints, we present the detailed algorithm of Pirex+ in Appendix §A.3. Concretely, we can instantiate Pirex+ with an efficient AHE scheme, e.g., exponential ElGamal [21]. Since exponential ElGamal uses a discrete log solver for decryption, it only permits a small size of plaintext (e.g., $|q| = 16$ or 32 bits). This can be adapted by splitting a parity ρ_i into $|q|$-bit chunks and separately encrypting each chunk. Therefore, each XOR operation on B-bit data entry (when computing the parity in Pirex) will be substituted with $\frac{B}{q}$ modular additions on $|q|$-bit data chunks. To encrypt a B-bit parity, it also takes $\frac{B}{q}$ AHE encryption invocations.

Complexity. We analyze the complexity of Pirex+, where the chunk size q is assumed to be constant. In the offline, the server incurs $O(\lambda N)$ PRS evaluations and $O(\frac{B}{q}\lambda N)$ modular 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 generation 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 PRS keys and upload the parity buffer to the server.

In the online phase, the server in Pirex+ incurs $O(\frac{B}{q}\sqrt{N})$ modular additions on $|q|$-bit parity chunks (instead of B-bit XOR operations as in Pirex). In addition, the server performs $O(\lambda\sqrt{N})$ XOR operations due to the 2-server XOR-PIR for offline parity retrieval. In total, the server performs $O(B\sqrt{N})$ modular additions and $O(B\lambda\sqrt{N})$ XOR operations. Meanwhile, the client executes $O(\lambda\sqrt{N})$ random bit generation, $O(\lambda\sqrt{N})$ PRS evaluations, and $O(\frac{B}{q})$ AHE decryptions plus $O(\frac{B}{q})$ modular additions (instead of XOR as in Pirex) for data entry recovery. The client bandwidth is $O(B)$ since only eight extra parities are transmitted (four received/sent).

For a database update, the client incurs $O(\lambda^2\sqrt{N})$ outbound bandwidth for the AHE-encrypted binary vector. To update **P**, the server performs $O(\frac{B}{q}\lambda\sqrt{N})$ scalar multiplications and group additions on $|q|$-bit parity chunks. Thus, the server cost per database item update 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 §A.4. □

7 Experimental Evaluation

7.1 Implementation

We fully implemented all our proposed schemes in `rust`. We used libraries from the `crates.io` registry to implement the following functionalities: For PRF, we used `aes` crate, which supports AES-NI instruction set for low-level AES parallel block processing. To optimize bitwise operations, we used `packed_simd` crate, which uses SIMD instruction to load a data chunk of 256 bytes onto the register per single XOR. For efficient memory access in the server database and client storage, we used `memmap` crate, which allows us to map the entire data files directly into the OS memory. For network communication between the client and the servers, we used `TcpStream` from the standard `std::net` module. For Pirex+, we implemented exponential ElGamal using `libsecp256k1` [3], which has `secp256k1` crate as the standard rust wrapper. For exponential ElGamal decryption, we used Shank's Baby-Step Giant-Step [53] to implement the discrete log solver. Our code is anonymously available at `https://anonymous.4open.science/r/pirex`.

7.2 Configuration

Hardware & network setting. For the client side, we used a 2023 Macbook Pro with Apple M2 CPU @ 3.5 GHz and 32 GB RAM. For the server side, we created two virtual server instances on a physical Dell PowerEdge Rack RX750 that has a 48-core Intel Xeon 8360Y CPU @ 2.4 GHz, 1TB RAM, and 1.5TB SSD. Note that we did not use all physical resources (only a few physical cores for each virtual server). The network bandwidth between the client and each server is around 40 Mbps with 7ms round-trip.

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

Multi-threading parallelization. We used up to 8 threads in the server for the offline phase to match the implementation of our counterpart (i.e. Piano [60]). In the online phase, all the server computation was processed by a single thread.

Counterpart comparison. We compare the performance of Pirex and Pirex+ with state-of-the-art OO-PIRs including CK20 [20], TreePIR [35], and Piano [60]. We selected the parameters for each scheme as follows.

- Pirex/Pirex+ : For our PRS, we used 128-bit PRF keys. For correctness, we set $M = \sqrt{N}\log N$ as the number of offline hints by Theorem 1. For Pirex+, we selected standard parameters for secp256k1 curve including 256-bit prime order and the base field $p = 2^{256} - 2^{32} - 977$. We divided each offline parity into 16-bit chunks for homomorphic encryption.
- CK20 [20]: We used 128-bit PRF keys for its PRS building block. We consider $\lambda = 128$ parallel executions of protocol instance and the number of offline hints to be $M = \sqrt{N}\log N$ for correctness as originally suggested.
- TreePIR [35]: We used 128-bit keys for puncturable PRF and set the number of hints $M = \sqrt{N}\log N$ for correctness.
- Piano [60]: We used 128-bit PRF keys. Unlike TreePIR, CK20, or Pirex, Piano requires rebuilding offline hints after $Q = \sqrt{N}\ln N$ online phases. We follow its suggestion to set the number of primary hints $M_1 = \sqrt{N}(\ln(2)\kappa + \ln(Q))$, the number of backup hints $M_2 = 3\sqrt{N}\ln N$, where $\kappa = 40$ is the chosen statistical security parameter for negligible failure probability. We measure its online performance with the amortized rebuild cost.

7.3 Results

Online bandwidth. Figure 6 illustrates the client online bandwidth cost of our schemes compared with other works. Pirex achieves the lowest bandwidth among all, where it is around $164\times - 542\times$ times smaller than Piano and TreePIR when performing on a 1TB database, depending on the chosen entry sizes. This is because our online bandwidth overhead is independent of database size. The client downloads a constant of eight parities, which is equivalent to 32/512/2048-KB to access a 4/64/256-KB entry, respectively. Meanwhile, Piano (*resp.* TreePIR) requires $O(\sqrt{N})$ parities to be downloaded, which costs from 69MB (*resp.* 131MB) to 558 MB (*resp.* 1 GB) of bandwidth for a 1TB database depending on the entry size. Thus, when comparing with Piano and TreePIR in terms of larger databases with larger entry sizes (2^{18} to 2^{24} entries of 64KB, for example), Pirex saves the client more than 98% of the inbound bandwidth. In CK20, although the client downloads a constant of 256 parities per online query (due to

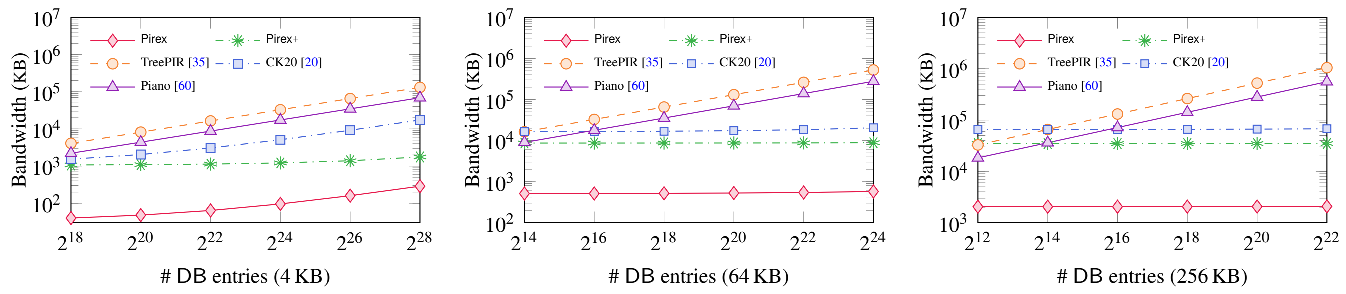

Figure 6: Client online bandwidth overhead.

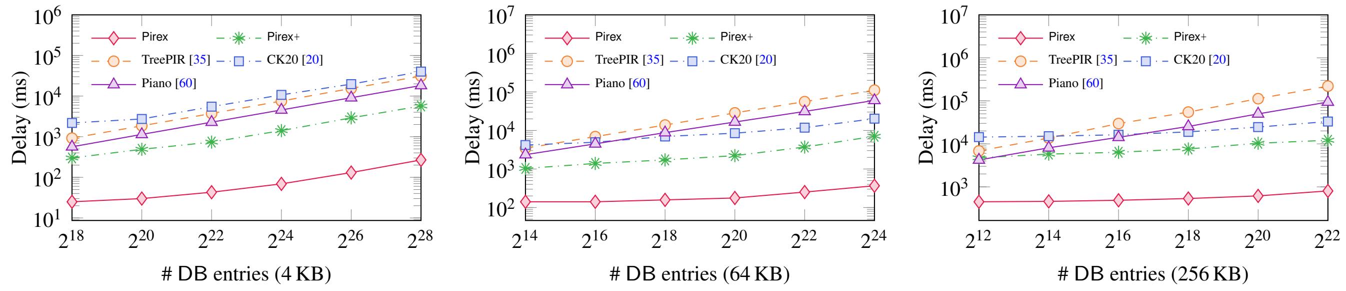

Figure 7: Online end-to-end delay.

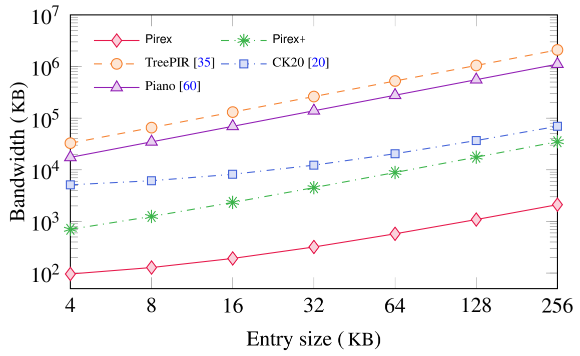

Figure 8: Online bandwidth (w/ fixed 2^{24} entries).

128 online instances executing in parallel), its concrete cost is about 35× larger than Pirex for all test cases. For databases 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 client 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 $\mathbf{P}$. Compared with Pirex, Pirex+ requires transmitting eight extra parities. Similar to Pirex, the inbound bandwidth cost of Pirex+ does not depend on the number of database entries, and thus, is significantly lower than other schemes. We report the client bandwidth of all schemes in Figure 8 with varied entry sizes on a database with 2^{24} entries.

Online end-to-end delay. Figure 7 illustrates the concrete end-to-end delay of Pirex and Pirex+ compared with CK20, TreePIR, and Piano. We can see that Pirex incurs a minimal delay with varied database and entry sizes. Specifically, it takes only 804ms for Pirex to access a 256 KB entry on a DB with 2^{22} records. This is approximately 116×, 273×, and 41× times faster than Piano, TreePIR, and CK20, respectively, which can take from 30s (CK20) to more than 90s (Piano, TreePIR) to access an entry. The high delay of Piano and TreePIR is mainly due to the cost of downloading $O(\sqrt{N})$ entries (compared with $O(1)$ in Pirex and 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 with 2^{22} entries of size 256 KB, our client and server computation is only around 7.5ms and 353ms, respectively. Meanwhile, this takes about 1s of client times and more than 30s of server times for CK20. On average, the client computation in Pirex is about 100×-150× faster than CK20, and the server computation is about 20×-30× faster than CK20.

The end-to-end delay of Pirex+ is at most 17× higher than Pirex yet it is 3×, 32×, 60× (e.g. on 64 KB entry size database with 2^{24} entries) lower than CK20, Piano, and TreePIR, respectively. For increasing database size, the gap between Pirex+ and Piano/TreePIR will be more significant. Note that the differences in delay between Pirex+ and Pirex are due to the extra operations, which include retrieving the offline parities and decrypting them by solving the discrete logs. However, since each parity is encrypted with 16-bit chunks, the cost to solve discrete log with baby-step giant-step is not significant, where it takes under 1s (for decrypting a 256 KB parity) (see cost breakdown below). Thus, the main difference

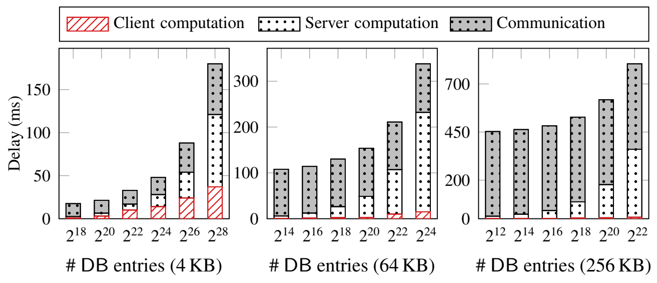

Figure 9: Cost breakdown of Pirex (online phase).

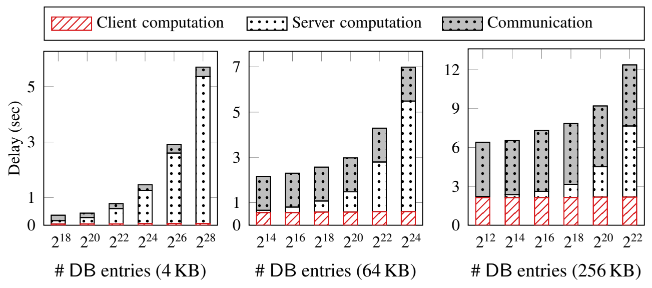

Figure 10: Cost breakdown of Pirex+ (online phase).

mostly stems from the extra four encrypted parities being downloaded (by using XOR-PIR) and the XOR computation that the servers perform on $O(N \log N)$ encrypted parities.

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

• Pirex: Figure 9 illustrates the detailed cost of Pirex for 1 GB-1 TB databases with 4 KB, 64 KB, and 256 KB entry sizes, respectively. There are three main factors contributing to the delay of Pirex including the client computation, the server processing, and the communication latency. The client overhead in Pirex is negligible, taking up merely 35 ms, thus, only contributing 1%-16% to the total delay (which therefore is hard to observe in Figure 9). The client performs three main operations: (i) looking up a hint, (2) creating an online query, and (3) recovering the database entry. Looking up a hint is fast as the client only performs 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 processing on eight parities responded from the server.

The server computation in Pirex is efficient, where it only takes 200ms to 350ms in large databases (2^{24} entries of 64 KB and 2^{22} entries of 256 KB, respectively), thus contributing up to approximately 45% in average of the online delay. The server cost mainly stems from performing XOR operations on $\sqrt{N}$ data entries. In Pirex, each query contains patched sets and partition sets, for which the set sizes are at most $\sqrt{N}$. Therefore, the amount of XOR operations performed by the server is sublinear to the database size and linear to the entry size. For a database with 2^{28} entries of size 4KB, Pirex takes only 84ms for end-to-end server computation.

Communication is the most dominating factor in the delay. However, as Pirex features constant bandwidth, this latency remains constant for each setting of entry sizes. With 40 Mbps network bandwidth configuration, Pirex takes around 105ms and 440ms to get eight parities of sizes 64 KB and 256 KB, respectively. For the databases with a small entry size (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. Therefore, the communication latency incurs approximately from 10ms to 60ms (respectively for databases with 2^{18} to 2^{28} entries) when the entry size is small (e.g. 4 KB).

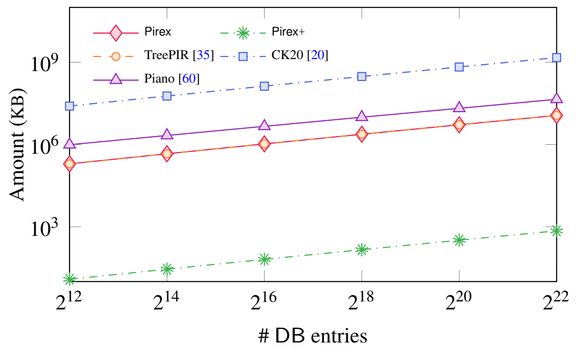

Figure 11: Client storage cost ($B = 256$ KB).

• Pirex+: Figure 10 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 555ms for 64 KB parities and 2s for 256 KB parities. The server processing is the most dominating factor, where it takes from 100ms to 5s, attributing 20%-60% of the total delay. This is because the two servers perform an extra oblivious refresh on the offline hints. The computation 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 illustrated by a growth with a small slope.

Storage. We report the client storage cost of Pirex /Pirex+ and counterparts in Figure 11. Pirex+ permits extremely low client storage compared with other schemes. With 256 KB entries, Pirex+ incurs four to six orders of magnitudes smaller client storage than Pirex, TreePIR, Piano, and CK20. Concretely, the client in Pirex+ only stores about 12 KB-700 KB sizes of PRF keys (for 2^{22} 256-KB DB entries) compared with 11 GB-1.3 TB in other schemes due to PRF keys and offline parities. Due to ciphertext expansion and oblivious write buffer, the extra server storage to maintain the encrypted offline parities in Pirex+ is approximately 369 GB for 1 TB DB.

Offline cost. Figure 12 reports the offline cost of Pirex and Pirex+ with other counterparts. Pirex features a comparable overhead to TreePIR, taking 7s-2600s to preprocess up to 1 TB database (with varied entry sizes). Piano requires entire

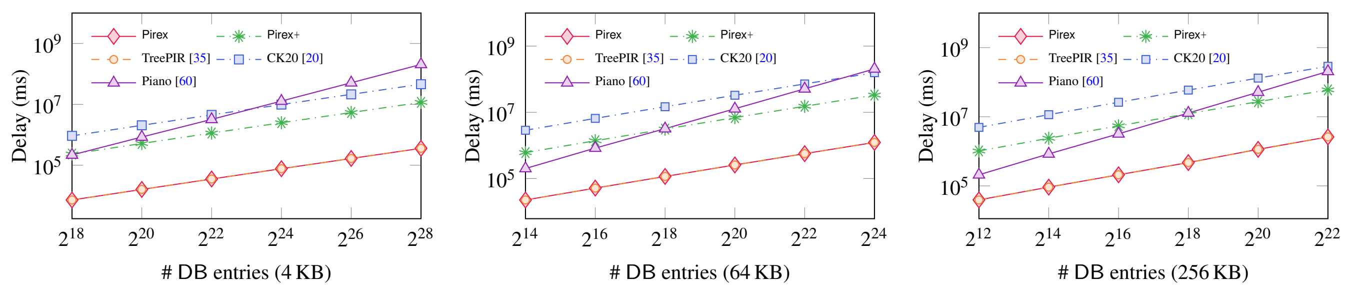

Figure 12: Offline end-to-end delay.

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 cost to privately update the offline parities stored on the server in Pirex+ when a database entry is updated. Pirex+ takes from 4ms to 3s to update an 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.

8 Related Work

Standard PIR. Chor et al. were the first to introduce PIR [17]. 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. [33] proposed an AHE-based PIR scheme with computational security and achieves $O(N^{\varepsilon})$ bandwidth ($\varepsilon > 0$). While later refinements reduced the bandwidth to sublinear [14,16,25,37], Sion et al. [55] showed that evaluating AHE is more expensive than streaming the database itself. To reduce computation overhead, some PIR schemes using lattice-based fully HE (FHE) were proposed [5,6,8,22,30,39,40,43]. However, all these schemes suffer a $\Omega(N)$ computation lower bound [12] in the standard PIR model. Thus, other settings have been explored to make PIR more efficient.

Global preprocessing PIR. Beimel et al. [12] 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 [13, 15] which require superlinear server storage to store the encoded database per designated group of clients that holds a secret key. Boyle et al. [13] 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. [36] thus presented a scheme based on standard Ring-LWE, where the server time and communication cost are polylogarithmic.

Client preprocessing PIR. Patel et al. [49] proposed PSIR, an OO-PIR model, where a precomputed offline private hint is used to accelerate online queries that involve linear PRF and sublinear public-key operations. Corrigan-Gibbs et al. [20] later proposed a two-server OO-PIR scheme with $\tilde{O}(\lambda\sqrt{N})$ online server cost, and a single-server variant using FHE. The scheme was improved in [19] to support $\sqrt{N}$ queries with $\tilde{O}(\sqrt{N})$ bandwidth and $O(N^{3/4})$ server time by using linearly HE. To reduce client query bandwidth cost to $\text{polylog}(N)$, other works leveraged privately puncturable/programmable PRF [34,54,61]. The main bottleneck in these schemes is the λ parallel protocol instance executions for correctness. Thus, Kogan et al. [32] showed a trick to remove the λ repetitions. It, however, requires $O(N)$ storage or $O(N)$ online time from the client for using non-private puncturable PRF. Lazzaretti et al. [35] suggested a novel partition paradigm for OO-PIR to obtain $\text{polylog}(N)$ query size in $O(\lambda\sqrt{N})$ client time. Their scheme, however, incurs $O(\sqrt{N})$ parities to be transmitted. Zhou et al. [60] adapts the scheme [35] to a single server setting but requires database streaming per $O(\sqrt{N})$ queries to rebuild the private offline hint. Mughees et al. [45] have recently proposed a concurrent and independent work that uses a different approach to create the offline hints to achieve the same objective as Pirex (i.e., small client bandwidth).

Batched PIR. Pioneered by [31], 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 [9,31,50,56] incurs a significant response overhead, Angel et al. [8] 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. [44] later proposed a vectorized batch PIR that can fit as many database entries as a single ciphertext can hold. Some batched PIR schemes [12,38] support multiple clients using efficient matrix multiplication techniques [18,57].

References

[1] Amazon Elastic Block Store - Block Size. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/volume_constraints.html#block_size.

[2] Azure Blob Storage - Blob Size. https://learn.microsoft.com/en-us/azure/storage/blobs/scalability-targets.

[3] Bitcoin Core Secp256k1. https://github.com/bitcoin-core/secp256k1.

[4] PostgreSQL - Block / Page Size Optimization. https://www.postgresql.org/message-id/3c840f8b-73f0-aae7-6bcf-e22d2a0a6a40%40gusw.net.

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

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

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

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

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

[10] Amos Beimel and Yuval Ishai. 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*, pages 912–926. Springer, 2001.

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

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

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

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

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

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

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

[18] Don Coppersmith and Shmuel Winograd. Matrix Multiplication via Arithmetic Progressions. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 1–6, 1987.

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

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

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

[22] Alex Davidson, Gonçalo Pestana, and Sofía Celi. Frodopir: Simple, scalable, single-server private information retrieval. *Proceedings on Privacy Enhancing Technologies*, 1:365–383, 2023.

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

[24] Klim Efremenko. 3-Query Locally Decodable Codes of Subexponential Length. In *Proceedings of The Forty-first Annual ACM symposium on Theory of Computing*, pages 39–44, 2009.

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

[26] Niv Gilboa and Yuval Ishai. 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*, pages 640–658. Springer, 2014.

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

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

[29] Aniko Hannak, Gary Soeller, David Lazer, Alan Mislove, and Christo Wilson. Measuring Price Discrimination and Steering on E-Commerce Web Sites. In *Proceedings of the 2014 conference on internet measurement conference*, pages 305–318, 2014.

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

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

[32] Dmitry Kogan and Henry Corrigan-Gibbs. Private Blocklist Lookups With Checklist. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 875–892, 2021.

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

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

[35] Arthur Lazzaretti and Charalampos Papamanthou. TreePIR: Sublinear-Time and Polylog-Bandwidth Private Information Retrieval from DDH. In Helena Handschuh and Anna Lysyanskaya, editors, *Advances in Cryptology – CRYPTO 2023*, pages 284–314, Cham, 2023. Springer Nature Switzerland.

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

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

[38] Wouter Lueks and Ian Goldberg. 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*, pages 168–186. Springer, 2015.

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

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

[41] Jakub Mikians, László Gyarmati, Vijay Erramilli, and Nikolaos Laoutaris. Detecting price and search discrimination on the internet. In *Proceedings of the 11th ACM workshop on hot topics in networks*, pages 79–84, 2012.

[42] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomization and Probabilistic Techniques in Algorithms and Data Analysis*. Cambridge University Press, 2017.

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

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

[45] Muhammad Haris Mughees, I Sun, and Ling Ren. Simple and Practical Amortized Sublinear Private Information Retrieval. *Cryptology ePrint Archive*, 2023.

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

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

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

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

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

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

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

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

[54] Elaine Shi, Waqar Aqeel, Balakrishnan Chandrasekaran, and Bruce Maggs. Puncturable Pseudorandom Sets and Private Information Retrieval with Near-Optimal Online Bandwidth and Time. In *Advances in Cryptology–CRYPTO 2021: 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16–20, 2021, Proceedings, Part IV 41*, pages 641–669. Springer, 2021.

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

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

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

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

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

[60] M. Zhou, A. Park, W. Zheng, and E. Shi. PIANO: Extremely Simple, Single-Server PIR with Sublinear Server Computation. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 55–55, Los Alamitos, CA, USA, may 2024. IEEE Computer Society.

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

A Proofs

A.1 Proof of Lemma 1

Proof. We will construct the simulator $\mathcal{S}_P$ such that all PPT environment $\mathcal{Z}$ cannot distinguish between its view in the Ideal and Real. Note that $\mathcal{Z}$ can statically corrupt one server and examine the execution transcript.

The simulator $\mathcal{S}$ functions as follows:

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

A.2 Proof of 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: $\mathcal{Z}$ provides a database DB to the client:

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 $(\mathcal{S}_1,\ldots,\mathcal{S}_{M_0}) \xleftarrow{\$} \mathcal{D}_n^{M_0}$.

Online: $\mathcal{Z}$ specifies an query index x to the client:

1. $\mathcal{S}$ outputs $\mathcal{S}^* \xleftarrow{\$} \mathcal{D}_n$ and $\mathcal{S}' \xleftarrow{\$} \mathcal{D}_n$.
2. $\mathcal{S}$ outputs two partition sets $\mathcal{T}$ and $\mathcal{T}'$, by invoking $\mathcal{S}_P$ from the Ideal world of PPR, two times.

In Ideal, everything is random. $\mathcal{S}^*, \mathcal{S}', \mathcal{T}, \mathcal{T}'$ are independent from each other and also independent from $\{\mathcal{S}_1,\ldots,\mathcal{S}_{M_0}\}$.

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:

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

Hybrid 0. We define Hyb_0 experiment as $\mathrm{REAL}_{\Pi_{\mathcal{F}},\mathcal{A},\mathcal{Z}}$ with an adversarial $\mathcal{A}$ and an environment $\mathcal{Z}$. We rewrite the real protocol of Pirex as in Figure 13.

Hybrid 1. Let Hyb_1 experiment be as in Figure 14. In Hyb_1, the difference is that the client samples each set $\mathcal{S}_j$ from $\mathcal{D}_n$ in the offline, instead of using a PRS key. The client stores all sets of indices in plain, so there is no more PRS operation.

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 the online queries created by any $\mathcal{S}_i$ sampled from either PRS or $\mathcal{D}_n$ would have a negligible difference under the view of $\mathcal{Z}$.

```
Offline: On receiving a database DB from $\mathcal{Z}$, the client executes:
  1. Sample $M$ identifier bits $(\ell_1,\ldots,\ell_M) \xleftarrow{\$} \{0,1\}^M$
  2. Sample $M$ PRS keys $(sk_1,\ldots,sk_M)$
  3. Request server $\mathsf{S}_{\ell_i}$ to compute $\rho_i \leftarrow \bigoplus_{j=1}^{n} \mathsf{DB}[s_j]$ for $s_j \in \mathcal{S}_i$
     where $\mathcal{S}_i \leftarrow \mathsf{PRS.Set}(sk_i)$
  4. Receive 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. Find $h_i = (\ell_i, sk_i, \rho_i)$ where $x = \mathsf{PRS.Eval}(sk_i, k)$
  6. Get $\mathcal{S}_i \leftarrow \mathsf{PRS.Set}(sk_i)$
  7. Get $(\hat{Q}_0, \hat{Q}_1) \leftarrow \mathsf{SubQuery}(x, \ell_i, \mathcal{S}_i)$
  8. Find $sk' \leftarrow \mathsf{PRS.Gen}(1^\lambda)$ where $x = \mathsf{PRS.Eval}(sk', k)$
  9. Get $\mathcal{S}' \leftarrow \mathsf{PRS.Set}(sk')$ and $\ell' \xleftarrow{\$} \{0,1\}$
  10. Get $(\hat{Q}'_0, \hat{Q}'_1) \leftarrow \mathsf{SubQuery}(x, \neg\ell', \mathcal{S}')$
  11. Send $Q_0 \leftarrow (\hat{Q}_0, \hat{Q}'_0)$ to server $\mathsf{S}_0$ and receive $\mathcal{R}_0$
  12. Send $Q_1 \leftarrow (\hat{Q}_1, \hat{Q}'_1)$ to server $\mathsf{S}_1$ and receive $\mathcal{R}_1$
  13. Obtain data item $\mathsf{DB}[x]$ using $\mathcal{R}_0$, $\mathcal{R}_1$, and parity $\rho_i$
  14. Obtain new parity $\rho'$ using $\mathcal{R}_0$, $\mathcal{R}_1$, and data $\mathsf{DB}[x]$
  15. Replace $h_i$ with new hint $h' = (\ell', sk', \rho')$
```

Figure 13: Hyb_0 Experiment

```
Offline: On receiving a database DB from $\mathcal{Z}$, the client executes:
  1. Sample $M$ identifier bits $(\ell_1,\ldots,\ell_M) \xleftarrow{\$} \{0,1\}^M$
  2. Sample $M$ index sets $(\mathcal{S}_1,\ldots,\mathcal{S}_M) \xleftarrow{\$} \mathcal{D}_n^M$
  3. Request server $\mathsf{S}_{\ell_i}$ to compute $\rho_i \leftarrow \bigoplus_{j=1}^{n} \mathsf{DB}[s_j]$ for $s_j \in \mathcal{S}_i$
  4. Receive 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. Find $h_i = (\ell_i, \mathcal{S}_i, \rho_i)$ where $x \in \mathcal{S}_i$
  6. Get $(\hat{Q}_0, \hat{Q}_1) \leftarrow \mathsf{SubQuery}(x, \ell_i, \mathcal{S}_i)$
  7. Find $\mathcal{S}' \xleftarrow{\$} \mathcal{D}_n$ where $x \in \mathcal{S}'$ and $\ell' \xleftarrow{\$} \{0,1\}$
  8. Get $(\hat{Q}'_0, \hat{Q}'_1) \leftarrow \mathsf{SubQuery}(x, \neg\ell', \mathcal{S}')$
  9. Send $Q_0 \leftarrow (\hat{Q}_0, \hat{Q}'_0)$ to server $\mathsf{S}_0$ and receive $\mathcal{R}_0$
  10. Send $Q_1 \leftarrow (\hat{Q}_1, \hat{Q}'_1)$ to server $\mathsf{S}_1$ and receive $\mathcal{R}_1$
  11. Obtain data item $\mathsf{DB}[x]$ using $\mathcal{R}_0$, $\mathcal{R}_1$, and parity $\rho_i$
  12. Obtain new parity $\rho'$ using $\mathcal{R}_0$, $\mathcal{R}_1$, and data $\mathsf{DB}[x]$
  13. Replace $h_i$ with new hint $h' = (\ell', \mathcal{S}', \rho')$
```

Figure 14: Hyb_1 Experiment

```
Offline: On receiving a database DB from $\mathcal{Z}$, the client executes:
  1. Sample $M$ identifier bits $(\ell_1,\ldots,\ell_M) \xleftarrow{\$} \{0,1\}^M$
  2. Sample $M$ index sets $(\mathcal{S}_1,\ldots,\mathcal{S}_M) \xleftarrow{\$} \mathcal{D}_n^M$
  3. Request server $\mathsf{S}_{\ell_i}$ to compute $\rho_i \leftarrow \bigoplus_{j=1}^{n} \mathsf{DB}[s_j]$ for $s_j \in \mathcal{S}_i$
  4. Receive 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. Find $h_i = (\ell_i, \mathcal{S}_i, \rho_i)$ where $x \in \mathcal{S}_i$
  6. Find $\mathcal{S}^* \xleftarrow{\$} \mathcal{D}_n$ where $x \in \mathcal{S}^*$
  7. Get $(\hat{Q}_0, \hat{Q}_1) \leftarrow \mathsf{SubQuery}(x, \ell_i, \mathcal{S}^*)$
  8. Find $\mathcal{S}' \xleftarrow{\$} \mathcal{D}_n$ where $x \in \mathcal{S}'$ and $\ell' \xleftarrow{\$} \{0,1\}$
  9. Get $(\hat{Q}'_0, \hat{Q}'_1) \leftarrow \mathsf{SubQuery}(x, \neg\ell', \mathcal{S}')$
  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. Get $\mathsf{DB}[x]$ using the ideal functionality $\mathcal{F}$
```

Figure 15: Hyb_2 Experiment

Hybrid 2. Let Hyb_2 experiment be as in Figure 15. 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 data queries as in Hyb_0. The client instead uses a newly sampled set $\mathcal{S}^* \leftarrow \mathcal{D}_n$, with $x \in \mathcal{S}^*$. Since $\mathcal{S}^*$ is not related to any precomputed offline parity, the client cannot recover the data item $\mathsf{DB}[x]$. Thus, we introduce the ideal functionality $\mathcal{F}$, which can return the correct answer based on the query input x from $\mathcal{Z}$. Note that in Hyb_2, we do not need to obtain the new parity (from using the responses) for hint replacement, since no precomputed hint is consumed in the online phase.

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 a new set $\mathcal{S}^*$ yields the same distribution of data queries as using $\mathcal{S}_i$. By Lemma 2, for each online query in Hyb_1, the selected set $\mathcal{S}_i$ is always guaranteed to be in the distribution $\mathcal{D}_n$. 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 data queries will have the same distribution under the view of $\mathcal{Z}$ as in Hyb_1.

Lemma 2. *For every online query x, even conditioned on $\mathcal{Z}$'s view over the previous queries, the local sets are still identically distributed as $\{\mathcal{S}_1,\dots,\mathcal{S}_M\} \xleftarrow{\$} \mathcal{D}_n^M$. Thus the set $\mathcal{S}_i$, with $x \in \mathcal{S}_i$, satisfied $\mathcal{S}_i \xleftarrow{\$} \mathcal{D}_n$.*

Proof. We refer to the proof of Lemma 3.3 in [60]. □

```
Offline: On receiving a database DB from $\mathcal{Z}$, the client executes:
  1. Sample $M$ identifier bits $(\ell_1,\dots,\ell_M) \xleftarrow{\$} \{0,1\}^M$
  2. Sample $M$ index sets $(\mathcal{S}_1,\dots,\mathcal{S}_M) \xleftarrow{\$} \mathcal{D}_n^M$
  3. Send each set $\mathcal{S}_i$ to server $\mathsf{S}_{\ell_i}$
Online: On receiving an index $x \in P_k$ from $\mathcal{Z}$, the client executes:
  4. Find $\mathcal{S}^* \xleftarrow{\$} \mathcal{D}_n$ where $x \in \mathcal{S}^*$ and $\ell^* \xleftarrow{\$} \{0,1\}$
  5. Get $(\hat{Q}_0,\hat{Q}_1) \leftarrow \mathsf{SubQuery}(x,\ell^*,\mathcal{S}^*)$
  6. Find $\mathcal{S}' \xleftarrow{\$} \mathcal{D}_n$ where $x \in \mathcal{S}'$ and $\ell' \xleftarrow{\$} \{0,1\}$
  7. Get $(\hat{Q}'_0,\hat{Q}'_1) \leftarrow \mathsf{SubQuery}(x,\neg\ell',\mathcal{S}')$
  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. Get $\mathsf{DB}[x]$ using the ideal functionality $\mathcal{F}$
```

Figure 16: Hyb_3 Experiment

Hybrid 3. Let Hyb_3 experiment be as in Figure 16. 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 in how the data queries $\hat{Q}_0$ or $\hat{Q}_1$ (derived from $\mathcal{S}^*$) is distributed to which server, where $\hat{Q}_0$ and $\hat{Q}_1$ are indistinguishable and independent from $\mathcal{S}_i$. This is because both server S_0 and S_1 have no prior knowledge about the set $\mathcal{S}^*$, which has no correlation to $(\ell_i, \mathcal{S}_i) \in h_i$. Thus, sampling a random server identifier ℓ^* is just for the input requirement of the $\mathsf{SubQuery}$ algorithm, rather than for the security of the created data queries.

```
Offline: On receiving a database DB from $\mathcal{Z}$, the client executes:
  1. Sample $M$ identifier bits $(\ell_1,\dots,\ell_M) \xleftarrow{\$} \{0,1\}^M$
  2. Sample $M$ index sets $(\mathcal{S}_1,\dots,\mathcal{S}_M) \xleftarrow{\$} \mathcal{D}_n^M$
  3. Send each set $\mathcal{S}_i$ to server $\mathsf{S}_{\ell_i}$
Online: On receiving an index $x \in P_k$ from $\mathcal{Z}$, the client executes:
  4. Sample $\mathcal{S}_0^* \xleftarrow{\$} \mathcal{D}_n$ and $\mathcal{S}_1^* \xleftarrow{\$} \mathcal{D}_n$
  5. Sample $(\mathcal{T}_0,\mathcal{T}_1) \leftarrow \mathsf{PPR.Gen}(k)$
  6. Sample $\mathcal{S}'_0 \xleftarrow{\$} \mathcal{D}_n$ and $\mathcal{S}'_1 \xleftarrow{\$} \mathcal{D}_n$
  7. Sample $(\mathcal{T}'_0,\mathcal{T}'_1) \leftarrow \mathsf{PPR.Gen}(k)$
  8. Send $Q_0 \leftarrow ((\mathcal{S}_0^*,\mathcal{T}_0),(\mathcal{S}'_0,\mathcal{T}'_0))$ to server $\mathsf{S}_0$
  9. Send $Q_1 \leftarrow ((\mathcal{S}_1^*,\mathcal{T}_1),(\mathcal{S}'_1,\mathcal{T}'_1))$ to server $\mathsf{S}_1$
 10. Get $\mathsf{DB}[x]$ using the ideal functionality $\mathcal{F}$
```

Figure 17: Hyb_4 Experiment

```
Offline: On receiving a database DB from $\mathcal{Z}$, the client executes:
  1. Sample $M$ identifier bits $(\ell_1,\dots,\ell_M) \xleftarrow{\$} \{0,1\}^M$
  2. Sample $M$ index sets $(\mathcal{S}_1,\dots,\mathcal{S}_M) \xleftarrow{\$} \mathcal{D}_n^M$
  3. Send each set $\mathcal{S}_i$ to server $\mathsf{S}_{\ell_i}$
Online: On receiving an index $x \in P_k$ from $\mathcal{Z}$, the client executes:
  4. Sample $\mathcal{S}_0^* \xleftarrow{\$} \mathcal{D}_n$ and $\mathcal{S}_1^* \xleftarrow{\$} \mathcal{D}_n$
  5. Sample $\mathcal{T}_0$, $\mathcal{T}_1$ by invoking $\mathcal{S}_P$ from PPR
  6. Sample $\mathcal{S}'_0 \xleftarrow{\$} \mathcal{D}_n$ and $\mathcal{S}'_1 \xleftarrow{\$} \mathcal{D}_n$
  7. Sample $\mathcal{T}'_0$, $\mathcal{T}'_1$ by invoking $\mathcal{S}_P$ from PPR
  8. Send $Q_0 \leftarrow ((\mathcal{S}_0^*,\mathcal{T}_0),(\mathcal{S}'_0,\mathcal{T}'_0))$ to server $\mathsf{S}_0$
  9. Send $Q_1 \leftarrow ((\mathcal{S}_1^*,\mathcal{T}_1),(\mathcal{S}'_1,\mathcal{T}'_1))$ to server $\mathsf{S}_1$
 10. Get $\mathsf{DB}[x]$ using the ideal functionality $\mathcal{F}$
```

Figure 18: Hyb_5 Experiment

Hybrid 4. Let Hyb_4 experiment be as in Figure 17. In Hyb_4, the main difference is that in the online, the client directly samples a random set $\mathcal{S}_0^*$ (or $\mathcal{S}_1^*$) as the patch sets to be sent to the servers, instead of sampling a new set $\mathcal{S}^* \supset \{x\}$ and a new server identifier to create and distribute the data queries using $\mathsf{SubQuery}$ algorithm.

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, $\mathcal{S}_0^*$ (or $\mathcal{S}_1^*$) has the same distribution as the query sets (included in $\hat{Q}_0$ and $\hat{Q}_1$ respectively) returned by $\mathsf{SubQuery}$. 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 $\tilde{\mathcal{S}}$, it is also randomly sampled from $\mathcal{D}_n$. Thus, $\mathcal{S}_0$ and $\mathcal{S}_1$ have the same distribution as $\bar{\mathcal{S}}$ and $\tilde{\mathcal{S}}$ in Hyb_3. Since both server S_0 and S_1 have no prior knowledge about $\mathcal{S}^*$ or $\tilde{\mathcal{S}}$, there is no need to specify any server identifier (as shown in Hyb_3) for the security when distributing the online queries.

Hybrid 5. Let Hyb_5 experiment be as in Figure 18. In Hyb_5 the only difference is that the client create the partition sets $(\mathcal{T}_0, \mathcal{T}_1)$ and $(\mathcal{T}'_0, \mathcal{T}'_1)$ using simulator $\mathcal{S}_P$ from PRR, instead

```
• $(\mathcal{H},\mathbf{P}) \leftarrow \mathsf{Prep}(\mathsf{DB},N)$:
1: for $i = 1$ to $M$ do
2:     $\ell_i \xleftarrow{\$} \{0,1\}$ and $sk_i \leftarrow \mathsf{PRS.Gen}(1^\lambda)$
3:     $\{s_1,\dots,s_n\} \leftarrow \mathsf{PRS.Set}(sk_i)$               }
4:     $\rho^i \leftarrow \sum_{j=1}^{n} \mathsf{DB}[s_j] \pmod{p}$          } executed by server $\mathsf{S}_{\ell_i}$
5:     $\mathbf{P}[i] \leftarrow \mathsf{AHE.Enc}(\mathsf{pk},\rho^i)$
6:     $h_i \leftarrow (\ell_i, sk_i, i)$
7: return $(\mathcal{H} \leftarrow (h_1,\dots,h_M), \mathbf{P})$
```

Figure 19: Pirex+ Offline Phase

```
• $(Q_0,Q_1,\mathcal{H}^*) \leftarrow \mathsf{Query}(x,\mathcal{H})$:
1: parse $\mathcal{H} = (h_1,\dots,h_M)$, $k \leftarrow \lfloor \frac{x}{m} \rfloor$
2: Find $h_i = (\ell_i, sk_i, \pi_i)$ where $x = \mathsf{PRS.Eval}(sk_i,k)$
3: $(\mathbf{q}_0,\mathbf{q}_1) \leftarrow \mathsf{XOR\text{-}PIR.Gen}(\pi_i)$
4: Let $\mathcal{S}_i \leftarrow \mathsf{PRS.Set}(sk_i)$
5: $(\hat{Q}_0,\hat{Q}_1) \leftarrow \mathsf{SubQuery}(x,\ell_i,\mathcal{S}_i)$
6: Let $sk' \leftarrow \mathsf{PRS.Gen}(1^\lambda)$ where $x = \mathsf{PRS.Eval}(sk',k)$
7: Let $\mathcal{S}' \leftarrow \mathsf{PRS.Set}(sk')$ and $\ell' \xleftarrow{\$} \{0,1\}$
8: $(\hat{Q}'_0,\hat{Q}'_1) \leftarrow \mathsf{SubQuery}(x,\neg\ell',\mathcal{S}')$
9: $Q_0 \leftarrow (\hat{Q}_0,\hat{Q}'_0,\mathbf{q}_0)$, $Q_1 \leftarrow (\hat{Q}_1,\hat{Q}'_1,\mathbf{q}_1)$
10: $\mathcal{H}^* \leftarrow (h_1,\dots,h_i,\dots,h_M,h')$ where $h' \leftarrow (\ell',sk',\perp)$
11: return $(Q_0,Q_1,\mathcal{H}^*)$

• $(\hat{Q}_0,\hat{Q}_1) \leftarrow \mathsf{SubQuery}(x,\ell,\mathcal{S})$:
1: $(\mathcal{T},\mathcal{T}^{(z)}) \leftarrow \mathsf{PPR.Gen}(k)$ where $k \leftarrow \lfloor \frac{x}{m} \rfloor$
2: $\bar{\mathcal{S}} \leftarrow \mathcal{S} \setminus \{x\} \cup \{z\}$ where $\{z\} \leftarrow \mathcal{T} \ominus \mathcal{T}^{(z)}$
3: $\tilde{\mathcal{S}} \leftarrow \{i \cdot m + \delta_i \ \forall\, i \in [n]\}$ where $(\delta_0,\dots,\delta_{n-1}) \xleftarrow{\$} [m]^n$
4: $\hat{Q}_\ell \leftarrow (\bar{\mathcal{S}},\mathcal{T}^{(z)})$, $\hat{Q}_{\neg\ell} \leftarrow (\tilde{\mathcal{S}},\mathcal{T})$
5: return $(\hat{Q}_0,\hat{Q}_1)$
```

Figure 20: Pirex+ Online Phase: Query

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

Figure 21: Pirex+ Online Phase: Answer

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 the Ideal, which completes our proof to show that Ideal and Real are computationally indistinguishable. □

A.3 Pirex+ Detailed Algorithm

We present Pirex+ in detail. To enable partial remote offline hint storage, Pirex+ has slightly different interfaces (marked as blue) over Pirex as follows:

```
• $(b_x,\langle\rho'\rangle,\mathcal{H}') \leftarrow \mathsf{Recover}(\mathcal{R}_0,\mathcal{R}_1,\mathcal{H})$:
1: parse $\mathcal{R}_0 = ((\bar{\rho}_0,w_0),(\bar{\rho}'_0,w'_0),r_0)$
2: parse $\mathcal{R}_1 = ((\bar{\rho}_1,w_1),(\bar{\rho}'_1,w'_1),r_1)$
3: parse $\mathcal{H}^* = (h_1,\dots,h_i,\dots,h_M,h')$
Reconstruct:
4: parse $h_i = (\ell_i, sk_i, \pi_i)$
5: $b \leftarrow \mathsf{PPR.Rec}(w_0,w_1)$
6: $\langle\rho\rangle \leftarrow \mathsf{XOR\text{-}PIR.Rec}(r_0,r_1)$
7: $\rho \leftarrow \mathsf{AHE.Dec}(\mathsf{sk},\langle\rho\rangle)$
8: $b_x \leftarrow b + \rho - \bar{\rho}_{\neg\ell_i} \pmod{p}$
Refresh:
9: parse $h' = (\ell', sk', \perp)$
10: $b' \leftarrow \mathsf{PPR.Rec}(w'_0,w'_1)$
11: $\rho' \leftarrow b_x - b'_t + \bar{\rho}'_{\ell'} \pmod{p}$
12: $\langle\rho'\rangle \leftarrow \mathsf{AHE.Enc}(\mathsf{pk},\rho')$
13: $h' \leftarrow (\ell', sk', \pi')$ with $\pi' \leftarrow c + M$
14: $\mathcal{H}' \leftarrow (h_1,\dots,h_{i-1},h',h_{i+1},\dots,h_M)$
15: return $(b_x,\langle\rho'\rangle,\mathcal{H}')$
```

Figure 22: Pirex+ Online Phase: Recover

- $(\mathcal{H},\mathbf{P}) \leftarrow \mathsf{Prep}(\mathsf{DB},N)$: Given a database DB and the number of entries N, it outputs a hint $\mathcal{H}$ and an encrypted parity buffer $\mathbf{P}$ to be stored at the servers.
- $(Q_0,Q_1,\mathcal{H}^*) \leftarrow \mathsf{Query}(x,\mathcal{H})$: 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 $\mathcal{H}^*$.
- $\mathcal{R}_i \leftarrow \mathsf{Answer}(Q_i,\mathsf{DB},\mathbf{P})$: Given a query Q_i, the database DB and the parity buffer $\mathbf{P}$, it outputs a response $\mathcal{R}_i$.
- $(b_x,\langle\rho'\rangle,\mathcal{H}') \leftarrow \mathsf{Recover}(\mathcal{R}_0,\mathcal{R}_1,\mathcal{H}^*)$: Given the hint $\mathcal{H}^*$ and two responses $\mathcal{R}_0$ and $\mathcal{R}_1$, it outputs the desired data entry b_x, a new 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:

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

We present the detailed algorithm of Pirex+ in Figure 19, Figure 20, Figure 21, Figure 22, Figure 23, Figure 24. We highlight the difference between Pirex+ and Pirex in blue. Pirex+ makes use of the following standard 2-server XOR-PIR scheme [17] on the parity buffer $\mathbf{P}$ of size $2M$:

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

```
• $(\mathcal{H}', \mathbf{P}') \leftarrow \mathsf{Rewrite}(\langle\rho'\rangle, \mathcal{H}, \mathbf{P})$:
Client:
 1: parse $\mathcal{H} = (h_1, \dots, h_i, \dots, h_M)$
 2: parse $h_i = (\ell_i, sk_i, \pi_i)$
 3: $(\mathbf{q}_0, \mathbf{q}_1) \leftarrow \mathsf{XOR\text{-}PIR.Gen}(\pi_i)$
 4: Send $\mathbf{q}_0$ to $\mathsf{S}_0$ and $\mathbf{q}_1$ to $\mathsf{S}_1$
Server: On receiving query $q_l$, $\mathsf{S}_l$ executes:
 5: $r_l \leftarrow \mathsf{XOR\text{-}PIR.Ret}(q_l, \mathbf{P})$
Client: On receiving $r_0$ and $r_1$, client executes:
 6: $\langle\rho\rangle \leftarrow \mathsf{XOR\text{-}PIR.Rec}(r_0, r_1)$
 7: $\rho \leftarrow \mathsf{AHE.Dec}(\mathsf{sk}, \langle\rho\rangle)$
 8: $\langle\rho\rangle \leftarrow \mathsf{AHE.Enc}(\mathsf{pk}, \rho)$
 9: Write $\mathbf{P}_{\mathsf{left}}[c] = \langle\rho\rangle$ to servers $\mathsf{S}_0$ and $\mathsf{S}_1$
10: Write $\mathbf{P}_{\mathsf{right}}[c] = \langle\rho'\rangle$ to servers $\mathsf{S}_0$ and $\mathsf{S}_1$
11: Update counter $c \leftarrow c + 1 \pmod{M}$
12: $\mathcal{H}' \leftarrow (h_1, \dots, h'_i, \dots, h_M)$ where $h'_i = (\ell_i, sk_i, i)$
13: return $(\mathcal{H}', \mathbf{P})$
```

Figure 23: Pirex+ Oblivious Refresh

```
• $\mathbf{P}' \leftarrow \mathsf{Update}(x, b'_x, \mathbf{P})$:
Client:
 1: Let $\mathbf{e} \leftarrow \{0\}^{2M}$, $k \leftarrow \lfloor \frac{x}{m} \rfloor$
 2: for each $h_i = (\ell_i, sk_i, \pi_i)$ do
 3:     $\mathbf{e}[\pi_i] = 1$ if $x = \mathsf{PRS.Eval}(sk_i, k)$
 4: for $i = 1$ to $2M$ do
 5:     $\langle e_i\rangle \leftarrow \mathsf{AHE.Enc}(\mathsf{sk}, \mathbf{e}[i])$
 6: Send $(\langle e_1\rangle, \dots, \langle e_{2M}\rangle)$ to $\mathsf{S}_0$ and $\mathsf{S}_1$
Server:
 7: $\varepsilon \leftarrow b'_x - \mathsf{DB}[x]$
 8: for $i = 1$ to $2M$ do
 9:     $\mathbf{P}[i] \leftarrow \mathbf{P}[i] \boxplus (\langle e_i\rangle \boxdot \varepsilon)$
10: return $\mathbf{P}$
```

Figure 24: Pirex+ Remote Update Parities

A.4 Proof of Theorem 3

Proof. We extend the simulator $\mathcal{S}$ from §A.2 as follows:

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 q and 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 $\langle 0\rangle$

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

We now define a sequence of hybrid experiments Hyb^+_i to show that the Real and Ideal are indistinguishable:

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

Hybrid 0. We define Hyb^+_0 experiment as $\mathrm{REAL}_{\Pi_{\mathcal{F}},\mathcal{A},\mathcal{Z}}$ with an adversarial $\mathcal{A}$ and an environment $\mathcal{Z}$. We simplify the real protocol of Pirex+ as in Figure 25.

Hybrid 1. Let Hyb^+_1 experiment be as in Figure 26. We highlight the difference between hybrid games Hyb^+_i and Hyb^+_{i+1} in red. Similar to Hyb_1, the client samples a set $\mathcal{S}_j \xleftarrow{\$} \mathcal{D}_n$ instead of using a PRS 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.

```
Offline: On receiving a database DB from $\mathcal{Z}$, the client executes:
  1. Execute steps (1) → (3) as in $\mathsf{Hyb}_0$ (Figure 13)
  2. Receive parity $\rho_i$ from $\mathsf{S}_{\ell_i}$ and set hint $h_i = (\ell_i, sk_i, i)$
  3. Send 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. Execute steps (5) → (12) as in $\mathsf{Hyb}_0$ to obtain $\mathcal{R}_0$, $\mathcal{R}_1$, $(\ell', sk')$
  5. Get $(\mathbf{q}_0, \mathbf{q}_1) \leftarrow \mathsf{XOR\text{-}PIR.Gen}(\pi_i)$
  6. Send $\mathbf{q}_0, \mathbf{q}_1$ to server $\mathsf{S}_0, \mathsf{S}_1$ and receive $r_0, r_1$
  7. Obtain parity $\rho_i$ using $r_0$ and $r_1$
  8. Execute steps (13) → (14) as in $\mathsf{Hyb}_0$ to obtain $\mathsf{DB}[x]$ and $\rho'$
Oblivious Refresh:
  9. Write $\mathbf{P}_{\mathsf{right}}[c] = \langle\rho'\rangle$ and add $h' = (\ell', sk', \pi' = c + M)$
 10. Get $(\mathbf{q}_0, \mathbf{q}_1) \leftarrow \mathsf{XOR\text{-}PIR.Gen}(\pi_c)$ with $\pi_c \in h_c$
 11. Send $\mathbf{q}_0, \mathbf{q}_1$ to server $\mathsf{S}_0, \mathsf{S}_1$ and receive $r_0, r_1$
 12. Obtain $\langle\rho\rangle$ using $r_0$ and $r_1$ and re-encrypt $\langle\rho\rangle$
 13. Write $\mathbf{P}_{\mathsf{left}}[c] = \langle\rho\rangle$ and update $h_c = (\ell_c, sk_c, c)$
```

Figure 25: Hyb^+_0 Experiment

```
Offline: On receiving a database DB from $\mathcal{Z}$, the client executes:
  1. Execute steps (1) → (3) as in $\mathsf{Hyb}_1$ (Figure 14)
  2. Receive parity $\rho_i$ from $\mathsf{S}_{\ell_i}$ and set hint $h_i = (\ell_i, \mathcal{S}_i, i)$
  3. Send 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. Execute steps (5) → (10) as in $\mathsf{Hyb}_1$ to obtain $\mathcal{R}_0$, $\mathcal{R}_1$
  5. Get $(\mathbf{q}_0, \mathbf{q}_1) \leftarrow \mathsf{XOR\text{-}PIR.Gen}(\pi_i)$
  6. Send $\mathbf{q}_0$ to server $\mathsf{S}_0$ and receive $r_0$
  7. Send $\mathbf{q}_1$ to server $\mathsf{S}_1$ and receive $r_1$
  8. Obtain parity $\rho_i$ using $r_0$ and $r_1$
  9. Execute steps (11) → (12) as in $\mathsf{Hyb}_1$ to obtain $\mathsf{DB}[x]$ and $\rho'$
Oblivious Refresh:
  9. Write $\mathbf{P}_{\mathsf{right}}[c] = \langle\rho'\rangle$ and add $h' = (\ell', \mathcal{S}', \pi' = c + M)$
 10. Get $(\mathbf{q}_0, \mathbf{q}_1) \leftarrow \mathsf{XOR\text{-}PIR.Gen}(\pi_c)$ with $\pi_c \in h_c$
 11. Send $\mathbf{q}_0$ to server $\mathsf{S}_0$ and receive $r_0$
 12. Send $\mathbf{q}_1$ to server $\mathsf{S}_1$ and receive $r_1$
 13. Obtain $\langle\rho\rangle$ using $r_0$ and $r_1$ and re-encrypt $\langle\rho\rangle$
 14. Write $\mathbf{P}_{\mathsf{left}}[c] = \langle\rho\rangle$ and update $h_c = (\ell_c, \mathcal{S}_c, c)$
```

Figure 26: Hyb^+_1 Experiment

Hybrid 2. Let Hyb^+_2 experiment be as in Figure 27. From Hyb_2, we know that the client uses a set $\mathcal{S}^*$ that is not related to any precomputed offline parity to create the data query. Therefore, no offline parity can be used to recover $\mathsf{DB}[x]$. The client thus does not need to obtain an offline parity using r_0 and r_1 as in Hyb^+_1. Similar to Hyb_2, the ideal functionality is invoked to obtain $\mathsf{DB}[x]$. Here in Hyb^+_2, we do not need to write a new refresh parity in the buffer $\mathbf{P}_{\mathsf{right}}$ since no precomputed hint (and the corresponding offline parity) is consumed in the online phase. 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.

```
Offline: On receiving a database DB from $\mathcal{Z}$, the client executes:
  1. Execute steps (1) $\rightarrow$ (3) as in $\mathsf{Hyb}_2$ (Figure 15)
  2. Receive parity $\rho_i$ from $\mathsf{S}_{\ell_i}$ and set hint $h_i = (\ell_i, \mathcal{S}_i, i)$
  3. Send 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. Execute steps (5) $\rightarrow$ (11) as in $\mathsf{Hyb}_2$ to obtain $\mathcal{R}_0$, $\mathcal{R}_1$
  5. Get $(\mathbf{q}_0, \mathbf{q}_1) \leftarrow \mathsf{XOR\text{-}PIR.Gen}(\pi_i)$
  6. Send $\mathbf{q}_0$ to server $\mathsf{S}_0$ and receive $r_0$
  7. Send $\mathbf{q}_1$ to server $\mathsf{S}_1$ and receive $r_1$
  8. Execute step (12) as in $\mathsf{Hyb}_2$ to obtain $\mathsf{DB}[x]$
Oblivious Refresh:
  8. Write $\mathbf{P}_{\mathsf{right}}[c] = \langle 0 \rangle$
  9. Get $(\mathbf{q}_0, \mathbf{q}_1) \leftarrow \mathsf{XOR\text{-}PIR.Gen}(\pi_c)$
 10. Send $\mathbf{q}_0$ to server $\mathsf{S}_0$ and receive $r_0$
 11. Send $\mathbf{q}_1$ to server $\mathsf{S}_1$ and receive $r_1$
 12. Obtain $\langle \rho \rangle$ using $r_0$ and $r_1$ and re-encrypt $\langle \rho \rangle$
 13. Write $\mathbf{P}_{\mathsf{left}}[c] = \langle \rho \rangle$ and update $h_c = (\ell_c, \mathcal{S}_c, c)$
```

Figure 27: Hyb_2^+ Experiment

```
Offline: On receiving a database DB from $\mathcal{Z}$, the client executes:
  1. Execute steps (1) $\rightarrow$ (3) as in $\mathsf{Hyb}_3$ (Figure 16)
  2. Send 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. Execute steps (4) $\rightarrow$ (9) as in $\mathsf{Hyb}_3$ to obtain $\mathcal{R}_0$, $\mathcal{R}_1$
  4. Get $(\mathbf{q}_0, \mathbf{q}_1) \leftarrow \mathsf{XOR\text{-}PIR.Gen}(\pi_i)$
  5. Send $\mathbf{q}_0$ to server $\mathsf{S}_0$ and $\mathbf{q}_1$ to server $\mathsf{S}_1$
  6. Execute step (10) as in $\mathsf{Hyb}_3$ to obtain $\mathsf{DB}[x]$
Oblivious Refresh:
  7. Write $\mathbf{P}_{\mathsf{right}}[c] = \langle 0 \rangle$
  8. Get $(\mathbf{q}_0, \mathbf{q}_1) \leftarrow \mathsf{XOR\text{-}PIR.Gen}(\pi_c)$
  9. Send $\mathbf{q}_0$ to server $\mathsf{S}_0$ and $\mathbf{q}_1$ to server $\mathsf{S}_1$
 10. Write $\mathbf{P}_{\mathsf{left}}[c] = \langle 0 \rangle$
```

Figure 28: Hyb_3^+ Experiment

Hybird 3. Let Hyb_3^+ experiment be as in Figure 28. 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 encrypts and sends a dummy buffer $\mathbf{P}$, with $\mathbf{P}[i] = \langle 0 \rangle$. Due to the IND-CPA property of AHE encryption, $\mathcal{Z}$ cannot distinguish between a dummy parity buffer and a buffer containing the offline parities computed by the servers. Therefore, the view of $\mathcal{Z}$ in Hyb_3^+ and Hyb_2^+ are computationally indistinguishable.

Note that since the parity buffer contains all zero, the client does not need to recover the offline parity retrieved by PIR when performing oblivious refresh, since the client already knows the value 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.Gen.

Hybird 4. Let Hyb_4^+ experiment be as in Figure 29. In Hyb_4^+, the additional difference is that for oblivious refresh, the client replaces the query strings produced by XOR-PIR.Gen, with two random bit strings of the same lengths. By the security of XOR-PIR, under a statically adversarial view $\mathcal{A}$, $\mathcal{Z}$ cannot distinguish between the two received bit strings. Thus, the view of $\mathcal{Z}$ in Hyb_4^+ and Hyb_3^+ are computationally indistinguishable.

```
Offline: On receiving a database DB from $\mathcal{Z}$, the client executes:
  1. Execute steps (1) $\rightarrow$ (3) as in $\mathsf{Hyb}_4$ (Figure 17)
  2. Send 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. Execute steps (4) $\rightarrow$ (9) as in $\mathsf{Hyb}_4$ to obtain $\mathcal{R}_0$, $\mathcal{R}_1$
  4. Get $(\mathbf{q}_0, \mathbf{q}_1) \leftarrow \mathsf{XOR\text{-}PIR.Gen}(\pi_i)$
  5. Send $\mathbf{q}_0$ to server $\mathsf{S}_0$ and $\mathbf{q}_1$ to server $\mathsf{S}_1$
  6. Execute step (10) as in $\mathsf{Hyb}_4$ to obtain $\mathsf{DB}[x]$
Oblivious Refresh:
  7. Write $\mathbf{P}_{\mathsf{right}}[c] = \langle 0 \rangle$
  8. Send random bit strings $\mathbf{q}_0$ to server $\mathsf{S}_0$ and $\mathbf{q}_1$ to server $\mathsf{S}_1$
  9. Write $\mathbf{P}_{\mathsf{left}}[c] = \langle 0 \rangle$
```

Figure 29: Hyb_4^+ Experiment

Hybird 5. Let Hyb_5^+ experiment be as in Figure 30. In Hyb_5^+, the additional difference is that to simulate the PIR query for retrieving an offline parity (to be used for $\mathsf{DB}[x]$ reconstruction originally), the client replaces the query bit strings produced by XOR-PIR.Gen, with two random bit strings of the same length. By the security of XOR-PIR, under a statically adversarial view $\mathcal{A}$, $\mathcal{Z}$ cannot distinguish between the two received bit strings. Therefore, the view of $\mathcal{Z}$ in Hyb_5^+ and Hyb_4^+ are computationally indistinguishable.

```
Offline: On receiving a database DB from $\mathcal{Z}$, the client executes:
  1. Execute steps (1) $\rightarrow$ (3) as in $\mathsf{Hyb}_5$ (Figure 18)
  2. Send 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. Execute steps (3) $\rightarrow$ (9) as in $\mathsf{Hyb}_5$ to obtain $\mathcal{R}_0$, $\mathcal{R}_1$
  4. Send random bit strings $\mathbf{q}_0$ to server $\mathsf{S}_0$ and $\mathbf{q}_1$ to server $\mathsf{S}_1$
  5. Execute step (10) as in $\mathsf{Hyb}_5$ to obtain $\mathsf{DB}[x]$
Oblivious Refresh:
  6. Write $\mathbf{P}_{\mathsf{right}}[c] = \langle 0 \rangle$
  7. Send random bit strings $\mathbf{q}_0$ to server $\mathsf{S}_0$ and $\mathbf{q}_1$ to server $\mathsf{S}_1$
  8. Write $\mathbf{P}_{\mathsf{left}}[c] = \langle 0 \rangle$
```

Figure 30: Hyb_5^+ Experiment

Note that Hyb_5^+ is identical to the online and offline phases simulated by the simulator $\mathcal{S}$ in the Ideal. The remaining part is to show that for the parities' update algorithm, $\mathcal{Z}$ cannot distinguish between the encrypted dummy binary vector from $\mathcal{S}$ and the real encrypted update vector from Figure 24. This is true due to the IND-CPA property of AHE, the encryption of a random binary vector is indistinguishable from the encryption of a specific binary vector (where the elements are 1 at the update positions on the parity buffer). To this end, the Ideal and Real for Pirex+ are computationally indistinguishable. □