Private Information Retrieval and Searching with Sublinear Costs

Mingxun Zhou

CMU-CS-25-115

May 2025

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Elaine Shi, Co-Chair
Giulia Fanti, Co-Chair
Bryan Parno
David J. Wu (The University of Texas at Austin)

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by the National Science Foundation under award numbers OIA2040675, CNS2325477 and CNS2044679; the David and Lucille Packard Foundation under award number 202071730; the Office of Naval Research under award number N000142212064; the Algorand Foundation Ltd. under award number 1031489; the University of Illinois at Urbana-Champaign under award number 08662216839; the Air Force Office of Scientific Research under award number FA95502110090; and Cornell University under award number PO1415822. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Private Information Retrieval, Private Information Searching

For my parents

Abstract

In this thesis, we investigate Private Information Retrieval (PIR), a cryptographic protocol that enables clients to access information from a database without revealing their queries to the server. As a fundamental building block for privacy-preserving applications, PIR has been extensively studied in both theory and practice for decades.

However, practical implementations have been limited to small-scale use cases due to the linear computation barrier of PIR, which requires the server to process the entire database for each query. The seminal works of Beimel, Ishai, and Malkin (Crypto 2000) and Corrigan-Gibbs and Kogan (Eurocrypt 2022) introduced Preprocessing PIR to overcome this barrier. While theoretically efficient, previous constructions remained impractical due to their reliance on expensive cryptographic operations.

To address this limitation, we propose two new PIR schemes: Piano and Quarter-PIR. Both achieve sublinear server computation and communication while remaining efficient in practice. These constructions transform the practical PIR landscape by providing near real-time responses for databases with billions of entries, while maintaining reasonable communication and storage requirements.

Furthermore, we demonstrate the practical utility of our PIR schemes through an important application – private information searching. We develop Pacmann, a new private approximate nearest neighbor search algorithm that delivers both high search quality and fast response times for databases with hundreds of millions of records.

Our work makes a significant step toward bridging the gap between theory and practice in PIR research. These contributions not only advance the state of the art in PIR designs, but also open new avenues for developing privacy-preserving applications in real-world and large-scale settings.

Acknowledgments

All of the work presented in this thesis was made possible with the help of many people who deserve my sincere thanks.

First and foremost, I want to thank my advisors, Elaine Shi and Giulia Fanti, for their unwavering guidance and support throughout this journey. They are truly the best advisors I could have hoped for.

My connection with them began in 2019 when I was still an undergraduate at Peking University. After discovering a simple incentive loophole in the Bitcoin mining process in a blockchain course, I sent cold emails to several professors seeking feedback, expecting no responses. To my surprise, Giulia was the first to reply, offering me a summer internship opportunity. I traveled to Pittsburgh, and though the project ultimately incorporated many of Giulia's ideas, it became my first research paper in security and privacy. Interestingly, Elaine was the second professor to respond to my email, also offering to work with me. By then, I had already committed to CMU (while Elaine was at Cornell). In 2020, I was fortunate to be accepted into CMU's PhD program based on Giulia's recommendation. By a wonderful coincidence, Elaine also moved to CMU that same year, and my research journey with both of them began.

Working with Elaine and Giulia has been truly rewarding. Elaine's expertise in cryptography continues to amaze me. I cherish those long hours we spent together tackling difficult security proofs, watching as she effortlessly mapped out perfect solutions. Her insight and dedication have inspired me deeply. Giulia has been an incredible mentor who taught me how to become an independent researcher. Her patience, encouragement, and our thoughtful discussions have given me new perspectives when I needed them most. Her support has been my safe harbor when research challenges seemed overwhelming. What I value most from both of them is the freedom they gave me to explore my own research interests. They consistently helped develop my ideas while providing the right balance of guidance and support. Their trust in me has been a gift I truly treasure.

I would like to thank my committee members, Bryan Parno and David Wu, for their valuable feedback and support. Much of my research draws inspiration from David's work, and I'm grateful for the thoughtful feedback and suggestions he's provided throughout these years. Bryan's meticulous attention to detail has strengthened my work immensely, and his constructive criticism always pushed me to think more deeply. I feel fortunate to have had such engaged and supportive committee members guiding my research.

I'm also deeply thankful to Prof. Hubert Chan, Prof. Chen Qian, and Prof. Yunhuai Liu for their steadfast support during both my undergraduate and PhD studies. Hubert has been an extraordinary collaborator – someone I could always rely on when facing the most challenging problems. The way he approaches complex

questions has taught me so much about problem-solving and perseverance. Prof. Qian was the primary supporter of my undergraduate research journey, and I'm truly grateful for his recommendation that helped open the doors to CMU. His belief in my potential came at a crucial time in my academic development. Prof. Liu holds a special place in my heart as the person who first introduced me to the world of academia and research, sparking a passion that has defined my path. All three also provided invaluable guidance during my job search, offering wisdom and connections when I needed them most.

I would like to thank my collaborators for their time and effort in working with me. I'm deeply appreciative of Wenting Zheng, Tianhao Wang, Wei-Kai Lin, Andrew Park, Mengshi Zhao, Ashrujit Ghoshal, Yiannis Tselekounis, Bo Peng, and Charlie Hou for their collaboration on multiple projects during my PhD. Each brought unique perspectives and skills that enriched our work together, turning challenging problems into exciting opportunities for discovery. Changrui Mu and Justin Zhang also helped the Pacmann project, and I am grateful for their time and effort.

I would also like to thank Arthur Lazzaretti, Jesko Dujmovic, Quang Dao, and Abhiram Kothapalli who selflessly shared their insights with me and greatly improved our work. Their generosity with their time and knowledge made a meaningful difference in the quality of our research, and I'm touched by their willingness to help even when they weren't formal collaborators.

My friends are the ones who inspired my work, helped me through the hard times, and made my life in Pittsburgh amazing. I would like to thank Tian Li, Jiaheng Zhang, and Yan Ji for their help during my PhD application process – their guidance gave me the confidence to pursue this path when it seemed daunting. Weizhao Tang and Zinan Lin helped me navigate my early years at CMU when everything felt new and challenging. Tianyuan Zhang and Hao Zhu were my wonderful roommates, and I'm grateful for their patience with me and for creating a home away from home. Ke Wu brought countless happy moments to our lives, and I will always view her as my role model – her energy and outlook on life continue to inspire me. I treasured every conversation with Hao Chung, who was always such a thoughtful listener when I needed someone most. Wei Dong shared so much knowledge with me during his short stay in Pittsburgh and was instrumental during my job search. Matt Hong, Jiaqi Yang, Yuxi Liu, and Lianke Qin were always there when I needed to vent about life's challenges, offering support without judgment. Shuoshuo Chen and Yi Zhou became like older brothers to me during my PhD years. We spent countless hours biking, camping, lifting weights, and traveling together. They not only supported my research but cared for me during the hardest times when I needed it most. Weichen Yu was my best labmate, bringing so much happiness and emotional support during the final two years of my PhD when the pressure was greatest. Chenyang Yang and Qi Pang were two friends at CMU with whom I spent the most time. I can't count how many conversations we had about research, life, and everything in between and beyond. They were the most reliable partners I could ever ask for, and I feel fortunate to have them in my life. There are so many more friends I don't have space to mention here, but I want to thank all of them for standing by me throughout this journey.

I would also like to thank all the members in CyLab for creating such a friendly and supportive environment where I could thrive. I'm also grateful to all the faculty and staff members in the Computer Science Department, the Electrical and Computer Engineering Department and also in CMU for their help during my PhD.

Last but not least, I would like to thank my family for their unconditional love and support. My parents are the ones who brought me into this world and raised me to be who I am today. They have stood by me through every challenge, celebrated every achievement, and provided a foundation of love that has given me the courage to pursue my dreams. Their belief in me has been unwavering, even when we were separated by oceans and continents. Through the words of encouragement during those late-night, they have been my constant source of strength. They truly mean everything to me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Information retrieval and search constitute the cornerstone functionalities of the modern Internet. From checking weather forecasts on our smartphones as we wake up, to navigating commute routes, staying informed with current news, and searching for crucial reference documents for our professional work, these systems permeate our daily existence. Throughout the evolution of the Internet, researchers and engineers have invested tremendous intellectual capital into developing information retrieval systems that are increasingly efficient, cost-effective, and precise – fundamentally transforming our interactions with the digital landscape.

Nonetheless, privacy – recognized as a fundamental human right [A+48] and enshrined in legal frameworks [Byg14] – remains critically underaddressed in information retrieval systems. Mainstream search engines and retrieval platforms typically require users to transmit queries directly to service providers, potentially exposing sensitive personal information. When a user searches for "flights from New York to London on August 1st", the search engine acquires knowledge of their travel intentions, which may subsequently be exploited for targeted advertising or behavioral tracking. More concerning scenarios arise when users search for highly sensitive topics such as "HIV treatment" or "witness protection program" – queries containing information whose unauthorized disclosure could result in severe consequences for the individual. Despite the development of privacy policies and regulatory frameworks, data breach incidents continue to happen in real-world retrieval systems, with documented cases affecting millions of users across more than 70 countries [aol06, bin20].

Is it possible to design an information retrieval system that enables users to access database content without exposing their queries to service providers? While a trivial approach is to download the entire database for each query, such a method would impose prohibitive communication costs. Another widely adopted privacy-enhancing technique is *anonymization* that obscures users' identities when they make queries, which can be achieved through popular techniques such as Virtual Private Networks (VPNs) [KK04] and the Tor Network [MBG+08]. Nonetheless, anonymization techniques do not prevent service providers from inferring sensitive information from the queries themselves, and deanonymization attacks are frequently discovered by researchers [SSGN17, JTJS14].

A cryptographic solution: Private Information Retrieval (PIR). In their seminal 1995 work, Chor, Goldreich, Kushilevitz, and Sudan [CGKS95] coined the term *"Private Information Retrieval"* (PIR) and established its formal definition from a cryptographic view: given a database

comprising n bits stored on a server and a client's query for index i, a PIR protocol must allow the retrieval of the i-th bit without revealing i to the server; i.e., the server's view in a PIR protocol should be indistinguishable from the case where the client queries for a random index. Notably, Chor et al. demonstrated the first PIR construction achieving sublinear communication complexity – where 'sublinear' indicates that the per-query communication overhead is $o(n)$, i.e., asymptotically smaller than the database size.

Following the work by Chor et al. [CGKS95], the PIR problem has attracted substantial research interest in the cryptography community. An extensive corpus of research spanning more than two decades [CGKS95, Cha04, GR05, CMS99, CG97, KO97, Lip09, OS07, Gas04, BFG03, SC07, OG11, MCG$^+$08, MG07] has established various approaches for constructing PIR protocols with non-trivial communication efficiency. The rise of Fully Homomorphic Encryption techniques [Gen09] has led to a wave of new PIR architectures [HHCG$^+$22, DGI$^+$19, MW22, BMW24] that achieve near-optimal communication complexity, where the per-query communication cost is (poly-)logarithmic in the database size. However, despite the great promise of PIR techniques in addressing the privacy concerns in information retrieval applications and the significant achievements in the theoretical study of PIR, the practical deployment of PIR schemes has remained limited. This raises the question: what are the reasons behind this gap between theory and practice?

Key challenges in practical PIR. Within the scope of this thesis, we discuss the following two key challenges that have hindered the practical deployment of PIR schemes:

1. *Computation Efficiency.* In a standard setting where a single server maintains the database, prior practical PIR schemes [MW22, HHCG$^+$22, DPC22, ACLS18, MR22] require a *linear computation cost per query*, which means that the server must process the entire database for each query. This poses a significant barrier to the practical deployment of PIR, as the scale of databases in real-world applications could easily reach billions of records. Unfortunately, it has been shown that the linear computation barrier is intrinsic to the classical PIR framework [BIM00] – which has been the primary focus of the PIR research community – thus necessitating exploration of alternative models.

   To get around this linear computation barrier, Beimel, Ishai, and Malkin proposed the *"preprocessing PIR"* model [BIM00], which requires the server and the client to preprocess the database upfront, with the computation results subsequently facilitating the online query process. Their work showed the first preprocessing PIR scheme that achieves sublinear computation per query after a one-time preprocessing phase. Given the potential of this preprocessing model, the research community has achieved significant advancements in this domain [BIM00, CHR17, CK20, KCG21, CHK22, LP23a, LP23b, LMW23, OPPW23, LLFP24], including our contributions [ZLTS23]. Nevertheless, existing schemes have either relied on computationally intensive cryptographic techniques or required multiple copies of the database hosted across multiple servers, rendering them infeasible outside a theoretical context.

2. *Supporting complex queries.* Current PIR implementations predominantly support only basic point-access queries, requiring clients to retrieve records by explicitly specifying their indices. Several research efforts have explored private key-value retrieval [CGN97, PSY23, CD24], enabling clients to access records using keyword identifiers. However, mainstream information retrieval architectures, including search engines and recommendation systems, demand capabili-

ties for similarity search or even semantic search. For instance, when a user queries "What's the cause of my sore throat?", they reasonably expect to receive results relating to "common cold" or "influenza", despite these terms not appearing explicitly in the original query. Unfortunately, such queries fall beyond the scope of both single-point access PIR protocols and key-value PIR frameworks. Existing approaches to private semantic search [SSLD22, HDCG+23, ABG+24] have substantial limitations in computational efficiency and retrieval quality.

Given the status quo, we ask the following questions:

1. *Can we have practically efficient single-server PIR schemes that achieve sublinear computation and communication?*

2. *Can we have practically efficient PIR schemes that support similarity and semantic search?*

Affirmative answers to these questions would not only be major progress over the previous state-of-the-art practical PIR schemes [MW22, HHCG+22, DPC22, ACLS18, MR22], but also open up new possibilities for many more privacy-preserving applications including private contact discovery [DRRT18, Con22], privacy-preserving stateless clients for blockchains [Per24], private DNS queries [SACM21, ZPSZ24, Fea], private medical information search [HFM+24], and various other domains where information retrieval intersects with privacy concerns.

Contributions. This thesis presents a novel pathway toward positive responses to these critical questions through a series of original research contributions.

First, we introduce two new preprocessing PIR schemes – Piano [ZPSZ24] and QuarterPIR [GZS24], which work effectively in the single-server setting while achieving (amortized) sublinear computation and communication query costs. Piano attains superior performance through a simple, self-contained algorithm design that eliminates dependence on computationally intensive cryptographic primitives and incorporates novel preprocessing techniques, requiring only minimal cryptographic assumptions. Building upon Piano's foundation, QuarterPIR substantially reduces the online query communication overhead by introducing an new cryptographic primitive named *"Privately Programmable Pseudorandom Function with List Decoding"*, which is a weaker variant of *"Privately Programmable Pseudorandom Function"* [BKM17, CC17, BTVW17]. We demonstrate that this weaker primitive can be efficiently constructed using simple cryptographic building blocks such as standard pseudorandom functions, and we show that it is sufficient to achieve a communication-efficient preprocessing PIR scheme, while the existing PIR constructions [ZLTS23, LP23a] require the stronger version of the primitive. We implement and benchmark both schemes against previous state-of-the-art single-server PIR implementations on databases containing billions of records. Notably, our implementations represent the first sublinear PIR protocols delivering practical performance in the single-server setting, achieving improvements in online query computation cost of two to three orders of magnitude relative to previous works. These contributions, together with the follow-up works [RMS24, HPPY24, WLZ+23, NGH24], fundamentally transform the practical PIR landscape by enabling sub-100ms response times for queries on billion-record databases.

Second, we develop Pacmann [ZSF25], a novel private semantic search algorithm that simultaneously addresses the trifecta challenges of privacy constraints, computational efficiency, and

search quality. Pacmann is built on a combination of our Piano PIR scheme and a graph-based vector search algorithm. Pacmann achieves substantial improvement in retrieval accuracy – over 90% accuracy for top-10 results on benchmark datasets containing 100 million records, nearly tripling the accuracy metrics of previous leading private search implementations [HDCG$^+$23, ABG$^+$24]. This performance is achieved while maintaining a response time of 3.1 seconds, which is 25% faster than the previous work.

Through the development of practically efficient sublinear PIR protocols and their applications in private semantic search implementations, we hope to bridge the critical gap between theoretical constructs and practical applications in the PIR domain. Our contributions establish a new frontier for the continued advancement of privacy-preserving information retrieval applications that maintain both performance viability and privacy guarantees.

1.1 Overview of results

This thesis consists of two main parts. In the first part, we mainly focus on the standard PIR problem and introduce our results on constructing practical PIR schemes with sublinear computation and communication, covering two sequential research results, Piano [ZPSZ24] and QuarterPIR [GZS24]. In the second part, we show the applicability of our PIR schemes in the context of private semantic search, focusing on our Pacmann [ZSF25] solution. The following overview captures the core ideas and the key results of our work.

Part I: PIR with Sublinear Computation

Problem setting. Consider a server maintaining a public database DB consisting of n records, each of constant size (typically a few bytes). Given an upper bound Q on the number of queries, a client issues sequential (even adaptive) queries $x_1, x_2, \ldots, x_Q$ to the server, where each query x_i is an index $x_i \in [n]$. For each query, the client aims to retrieve the record $\mathsf{DB}[x_i]$ from the server, without revealing the index x_i to the server.

This thesis focuses on the *client-specific preprocessing PIR model*, building upon the framework established by Corrigan-Gibbs and Kogan [CK20]. Within this model, the client and server engage in a one-time preprocessing phase prior to the first query, during which they exchange messages and perform necessary computations. The client is responsible for storing the results of this preprocessing phase. Critically, the preprocessing should be agnostic to the future queries.

Piano: A Practical Single-server Preprocessing PIR Scheme. We introduce Piano [ZPSZ24] (short for Private Information Access NOw), a novel single-server PIR scheme in the client-specific preprocessing model. Piano requires a one-time preprocessing cost of $O(n)$ communication and computation, after which the client stores $\widetilde{O}(\sqrt{n})$ bits of the preprocessing results. Following this preprocessing, Piano supports an unbounded number of queries with $\widetilde{O}(\sqrt{n})$ communication and computation per query, making it a sublinear PIR scheme when amortized over multiple queries. This result can be summarized in the following theorem:

Table 1.1: **Comparison of single-server and two-server preprocessing PIR schemes (for unbounded queries)**. Any single-server scheme immediately implies a two-server result with the same performance bounds. n is the size of the database and m is the number of clients. The computation overhead counts both the client's and the server's computation, and here we report the expected computation. The server space counts only the extra storage needed on top of storing the original database. Piano [ZPSZ24] and QuarterPIR [GZS24] are our proposed schemes.

Scheme	Assumpt.	Compute	Comm.	Space		#	Concrete
				client	server	servers	eff.
With public-key cryptography							
[CHK22]	LWE	$\widetilde{O}_\lambda(\sqrt{n})$	$\widetilde{O}_\lambda(\sqrt{n})$	$\widetilde{O}_\lambda(\sqrt{n})$	$\widetilde{O}_\lambda(m \cdot n)^*$	1	✗
Ours [ZLTS23], [LP23a]	LWE	$\widetilde{O}_\lambda(\sqrt{n})$	$\widetilde{O}_\lambda(1)$	$\widetilde{O}_\lambda(\sqrt{n})$	$\widetilde{O}_\lambda(m \cdot n)^*$	1	✗
[LMW23]	Ring-LWE	$\mathsf{poly}((\log n)^{1/\epsilon})$	$\mathsf{poly}((\log n)^{1/\epsilon})$	0	$n^{1+\epsilon}$	1	✗
[SACM21]	LWE	$\widetilde{O}_\lambda(\sqrt{n})$	$\widetilde{O}_\lambda(1)$	$\widetilde{O}_\lambda(\sqrt{n})$	0	2	✗
[LP23b]	Various	$\widetilde{O}_\lambda(\sqrt{n})$	$\widetilde{O}_\lambda(1)$	$\widetilde{O}_\lambda(\sqrt{n})$	0	2	✓
QuarterPIR	Various	$\widetilde{O}_\lambda(\sqrt{n})$	$\widetilde{O}(\sqrt{n})$ offline $\widetilde{O}_\lambda(1)$ online	$\widetilde{O}_\lambda(\sqrt{n})$	0	1	✓
Without public-key cryptography							
[BIM00]	None	$O(n/\log^2 n)$	$O(n^{1/3})$	0	$O(n^2)$	2	✗
[BIM00]	None	$O(n^{1/2+\epsilon})$	$O(n^{1/2+\epsilon})$	0	$O(n^{1+\epsilon'})^{**}$	2	✗
[CK20]	OWF	$\widetilde{O}_\lambda(\sqrt{n})$	$\widetilde{O}(\sqrt{n})$	$\widetilde{O}_\lambda(\sqrt{n})$	0	2	✓
[KCG21]	OWF	$O(n)$	$\widetilde{O}_\lambda(1)$	$\widetilde{O}_\lambda(\sqrt{n})$	0	2	✓
Piano, [RMS24]	OWF	$\widetilde{O}_\lambda(\sqrt{n})$	$O(\sqrt{n})$	$\widetilde{O}_\lambda(\sqrt{n})$	0	1	✓
QuarterPIR	OWF	$O_\lambda(\sqrt{n})$	$O_\lambda(n^{1/4})$	$\widetilde{O}_\lambda(\sqrt{n})$	0	2	✓
QuarterPIR	OWF	$O_\lambda(\sqrt{n})$	$O(\sqrt{n})$ offline $O_\lambda(n^{1/4})$ online	$\widetilde{O}_\lambda(\sqrt{n})$	0	1	✓

$*$: In the unbounded query setting, some earlier works [ZLTS23, LP23a, CHK22] require that the next preprocessing is persistently attached to the current window of $O(\sqrt{n})$ operations. The preprocessing consumes $O_\lambda(n)$ server space per client to evaluate under FHE an $\widetilde{O}(n)$-sized circuit containing a sorting network.

$**$: $\epsilon' > 0$ depends on ϵ.

Theorem 1.1.1 (Single-server preprocessing PIR with sublinear computation and communication)**.** *Assume the existence of one-way functions. There exists a single-server preprocessing PIR scheme with (amortized) $O_\lambda(n^{1/2})$ communication, $O_\lambda(n^{1/2})$ server computation and $\widetilde{O}_\lambda(n^{1/2})$ client computation per query, while incurring $\widetilde{O}(n^{1/2})$ client storage.*

The most notable characteristic of Piano is its remarkable *simplicity*. Unlike previous sublinear PIR schemes [CK20, CHK22, ZLTS23, LP23a], our approach eliminates the need for homomorphic encryption or other computationally intensive cryptographic primitives such as privately puncturable PRFs [BKM17, CC17, BTVW17]. In fact, *the only cryptographic primitive we need is pseudorandom functions (PRFs)*, which can be efficiently implemented using the AES-NI instruction sets available in most modern processors. Furthermore, our construction is entirely *self-contained*, without relying on any existing PIR scheme as a building block.

The emphasis on simplicity translates directly to **ease of adoption**. Our parallelized implementation of Piano comprises around 800 lines of Golang code. We have also developed a tutorial

implementation that captures the scheme's core concepts in approximately 160 lines of code, making the fundamental ideas accessible to researchers and practitioners alike.

We conducted an experiment on a 100GB database with a 60ms RTT coast-to-coast connection – by far the largest database size evaluated for any PIR scheme in the literature. Our implementation achieved a response time of **73ms**, while the prior state-of-the-art solutions required 11s or higher. This represents over **150×** speedup compared to previous works. Moreover, since our improvement is asymptotic in nature, the performance gap will continue to widen as database sizes increase.

Apart from its contribution to practical efficiency, Piano is also of significant theoretical interest. It achieves the optimal theoretical tradeoff between client storage and server computation, matching the lower bound established by Corrigan-Gibbs, Henzinger, and Kogan [CHK22]. Moreover, Piano represents the first single-server sublinear PIR scheme that operates without any form of public-key cryptography, which has triggered the research interest in understanding sublinear PIR schemes in the Minicrypt model. For example, a recent lower bound result by Ishai et al. [ISW24] demonstrates that Piano's communication cost is nearly optimal subject to the black-box use of one-way functions.

QuarterPIR: A Communication-efficient Preprocessing PIR Scheme. In this work [GZS24], we propose a new cryptographic primitive called *"Privately Programmable Pseudorandom Function with List Decoding"*, and demonstrate its applicability by constructing QuarterPIR, a communication-optimized preprocessing PIR scheme. QuarterPIR adheres to the same client-specific preprocessing model as Piano, while significantly reducing the asymptotic complexity of the *online* communication cost – a critical metric for client experience – while maintaining the same asymptotic efficiency as Piano for all other performance parameters.

Theorem 1.1.2 (Single-server preprocessing PIR with improved online communication)**.** *Assume the existence of one-way functions. There exists a single-server preprocessing PIR scheme with $O_\lambda(n^{1/4})$ online communication, $O(n^{1/2})$ offline communication, $O_\lambda(n^{1/2})$ server computation and $\widetilde{O}_\lambda(n^{1/2})$ client computation per query, while incurring $\widetilde{O}(n^{1/2})$ client storage.*

In comparison with the construction of Piano (Theorem 1.1.1), QuarterPIR (Theorem 1.1.2) improves the online communication cost from $\widetilde{O}(\sqrt{n})$ to $\widetilde{O}_\lambda(n^{1/4})$, while maintaining equivalent asymptotic costs for all other parameters. This scheme also achieves the same optimal tradeoff between client storage and server computation. Our experimental results demonstrate that QuarterPIR delivers a 12- to 50-times improvement in communication cost, while incurring computation costs approximately three times higher per query compared to Piano.

Interestingly, as a side effect of the techniques we developed for QuarterPIR, we demonstrate that if we assume the existence of a classical PIR scheme with $\widetilde{O}_\lambda(1)$ communication (established under various cryptographic assumptions such as LWE, Φ-hiding, Damgård-Jurik, DDH, QR) [CMS99, HHCG+22, MW22, DGI+19, BMW24], our new techniques yield a concretely efficient single-server PIR scheme with $\widetilde{O}_\lambda(1)$ *online* communication, $\widetilde{O}(\sqrt{n})$ offline communication and computation per query, utilizing $\widetilde{O}_\lambda(\sqrt{n})$ client storage. This represents the first PIR construction in the unbounded query model that achieves polylogarithmic online communication without requiring the server to maintain any client-specific state. We summarize this additional result in the following theorem:

Theorem 1.1.3. *Assume the existence of a classical single-server PIR scheme (i.e., without*

preprocessing) that enjoys $\widetilde{O}_\lambda(1)$ communication per query. Then, there exists a single-server preprocessing PIR scheme with $\widetilde{O}_\lambda(1)$ online communication, $\widetilde{O}_\lambda(\sqrt{n})$ computation, $\widetilde{O}(\sqrt{n})$ offline communication, requiring $\widetilde{O}_\lambda(\sqrt{n})$ client storage and no additional server storage except the original database.

We provide a comparison of our schemes with the previous PIR schemes in Table 1.1.

Part II: Private Information Search.

Problem setting. Private information searching can be viewed as a generalization of the PIR problem in the previous part. Specifically, consider a server that stores a public database DB of n records $\mathsf{DB}[1], \mathsf{DB}[2], \ldots, \mathsf{DB}[n]$, where the records may represent documents, images, or other data formats. Given an upper bound Q on the client's number of queries, the client issues a sequence of potentially adaptive queries $x_1, \ldots, x_Q$ to the server, where each query x_i could be a keyword, a natural language sentence, or even an image or video. For the i-th query, the client aims to retrieve the indices of the top K most relevant records to query x_i from database DB, while preserving the privacy of x_i. This thesis specifically focuses on *vector search*, where both the database records and client queries are represented as high-dimensional vectors. Given a specific distance metric (e.g., Euclidean distance), the client tries to retrieve the indices of the top K *approximate nearest neighbors* for each query x_i. We concentrate on vector search because semantic search and similarity search can be effectively reduced to vector search through the application of "embedding techniques" [RKH+21, RG19], which is a standard practice in modern search services [LKLJ18, LLJ+21, HSS+20].

Pacmann: Graph-based Private Nearest Neighbor Search. Building upon our previous results [ZPSZ24, GZS24], in this work [ZSF25], we propose a novel private vector search scheme called Pacmann (Private ACcess to More Approximate Nearest Neighbors). Pacmann represents the first private vector search scheme that simultaneously achieves practical efficiency and high search quality. Specifically, Pacmann attains nearly 90% top-10 search accuracy on a benchmark containing 100 million records with a response time of only 3.1 seconds. This performance nearly triples the accuracy of the previous state-of-the-art private vector search solution [HDCG+23], while reducing response time by approximately 25%.

The key innovation of Pacmann's design lies in our integration of powerful *graph-based vector search* into the private setting through our newly developed PIR schemes. Previous approaches [HDCG+23, ABG+24] were limited to relatively simple search algorithms due to privacy constraints, resulting in suboptimal search quality. Our private implementation of graph-based vector search inherits the advantages of our earlier work on Piano and QuarterPIR, requiring only sublinear computation and communication cost per query, thus achieving an optimal balance of privacy and performance.

Conceptually, graph-based vector search constructs a graph structure over database DB, allowing the search algorithm to traverse this graph to identify nearest neighbors for queries. While being effective in non-private contexts, adapting this approach to private settings presents significant challenges, as the access pattern during graph traversal potentially exposes sensitive information about the query. We demonstrate that our novel PIR schemes, Piano and QuarterPIR, provide an elegant solution to this challenge.

Our fundamental insight is to *execute the graph traversal algorithm on the client side*, thereby

completely eliminating privacy concerns. To avoid the impractical requirement of storing the entire graph client-side, we enable the client to dynamically retrieve necessary graph information using our advanced PIR schemes. This approach allows the client to comprehensively traverse the graph without compromising the queries' privacy, then identify the top K nearest neighbors after minimal communication rounds with the server. We further improve practical performance through optimizations in both the graph traversal algorithms and PIR query processes.

The results of Pacmann represent not merely a technical advancement in private vector search, but also demonstrate a novel paradigm for privacy-preserving computation. For general computational tasks where clients need to query server-stored data structures (such as hash tables, trees, or graphs), we can effectively hide client intentions by moving computation to the client side while using our PIR schemes to retrieve only the necessary information. Importantly, this "separation of computation and storage for privacy" approach is viable only when the underlying PIR schemes achieve sufficient efficiency (i.e., sublinear complexity); otherwise, implementations would remain computationally impractical. We hope that this new paradigm can significantly influence the future evolution of privacy-preserving systems.

1.2 Thesis Organization

The remainder of this thesis is organized as follows.

- Part I includes the following two main constructions for PIR with sublinear computation:
  - Chapter 2 presents the Piano PIR scheme. The chapter is based on a jointly authored paper with Andrew Park, Elaine Shi, Wenting Zheng: *"Piano: Extremely Simple, Single-Server PIR with Sublinear Server Computation"*, published in the IEEE Symposium on Security and Privacy in 2024 [ZPSZ24].
  - Chapter 3 introduces the QuarterPIR scheme. This chapter is based on a jointly authored paper with Ashrujit Ghoshal, Elaine Shi: *"Efficient Pre-processing PIR Without Public-Key Cryptography"*, published in the IACR Annual International Conference on the Theory and Applications of Cryptographic Techniques ("Eurocrypt") in 2024 [GZS24].
- Part II focuses on the application of PIR in private semantic search:
  - Chapter 4 presents the Pacmann private semantic search algorithm. This chapter is based on a jointly authored paper with Elaine Shi and Giulia Fanti: *"Pacmann: Private Approximate Nearest Neighbor Search with Graph-based Vector Search"*, published in the International Conference on Learning Representations ("ICLR") in 2025 [ZSF25].
- Part III concludes the thesis and discusses future research directions.

Part I

Private Information Retrieval with Sublinear Computation

Chapter 2

Piano: Simple Single-Server PIR with Sublinear Computation

2.1 Introduction

Two classes of PIR schemes. There are two main classes of PIR schemes, depending on whether they rely on preprocessing. Classical PIR schemes do not perform any preprocessing of the database, and the server simply stores an original copy of the database DB. In this setting, although we can achieve polylogarithmic communication per query, the server's computation overhead must be *linear* in the size of the database. Beimel, Ishai, and Malkin [BIM00] showed that the linear server computation overhead is inherent — intuitively, if there is some entry that is not touched during some query, then the server learns that the client is not interested in this entry. To overcome this prohibitive linear server computation barrier, Beimel et al. introduced the *preprocessing* model [BIM00], which was further explored in several subsequent works [CK20, SACM21, LP23b, CHK22, KCG21, ZLTS23, LP23a, LMW23]. In the *client-specific preprocessing* model (also called the *subscription* model), we have each client download and store a "hint" from the server during preprocessing. In this model, it is known that with $\widetilde{O}_\lambda(\sqrt{n})$ client-side storage, each online query can be accomplished with polylogarithmic communication and $\widetilde{O}_\lambda(\sqrt{n})$ server and client computation [ZLTS23, LP23a]. Another possible model is the *global preprocessing* model, in which the server performs a global preprocessing and computes an encoding of the database upfront for all clients. In this model, the most recent breakthrough work by Lin, Mook, and Wichs [LMW23] showed that for any constant $\epsilon > 0$, with $O(n^{1+\epsilon})$ amount of server storage, each query can be accomplished with $(\mathsf{poly}\log n)^{1/\epsilon}$ communication and $(\mathsf{poly}\log n)^{1/\epsilon}$ server computation.

Practical landscape for PIR. The community has made various attempts to implement and optimize PIR for practical applications [MW22, HHCG$^+$22, DPC22, ACLS18, MR22]. In the single-server setting, to the best of our knowledge, only PIR schemes with *linear* server computation have been implemented prior to our work. Although recent works [HHCG$^+$22] managed to achieve server-computation throughput comparable to the native memory bandwidth, the linear amount of computation severely limits the scalability to larger databases, and precludes various killer applications such as private DNS (where the database can be as large as 100GB). Unsurpris-

ingly, prior works conducted experiments for databases of size up to 8GB [MW22, HHCG+22], and the server time per query is more than one second for this data size.

A natural question is why prior implementation efforts did not choose schemes with preprocessing despite their better asymptotic performance. The reason is that prior single-server, preprocessing sublinear PIR schemes are theoretical in nature. In particular, prior schemes with polylogarithmic communication [ZLTS23, LP23a, LMW23] require one or more of the following heavy-weight cryptographic primitives: Fully Homomorphic Encryption (FHE), Privately Programmable PRFs [BLW17, PS18, KW21], and polynomial encoding data structures [KU11], which introduce astronomical constants in the concrete performance. Unfortunately, within the limits of known techniques, we are still very far from making these cryptographic primitives practical when handling large amount of data (or even implementable)! Finally, although the work of Corrigan-Gibbs et al. [CHK22] showed how to get sublinear server computation using only linear homomorphic encryption, they pay the price of much worse asymptotics, that is, $\widetilde{O}_\lambda(n^{2/3})$ for client storage and server computation. Consequently, their scheme is also not a sweetspot for practical implementation.

Time for a paradigm shift for practical PIR? Can we have a concretely efficient, single-server PIR scheme with sublinear server computation? An affirmative answer to the above question promises a paradigm shift for the practical landscape of single-server PIR! Specifically, our goal is to eventually eschew the linear server computation regime for practical implementations, and thus allow scaling to large database sizes.

2.1.1 Our Contributions

We propose a novel single-server PIR scheme called PIANO (short for "Private Information Access NOw"). PIANO adopts the client-specific preprocessing model. With roughly $\widetilde{O}(\sqrt{n})$ client-side storage, we achieve $\widetilde{O}(\sqrt{n})$ online communication and computation per query. The most notable feature of PIANO lies in its *simplicity*. Unlike prior sublinear PIR schemes [CK20, CHK22, ZLTS23, LP23a], we do not need any form of homomorphic encryption or other heavy-weight cryptographic primitives such as privately puncturable PRFs [BKM17, CC17, BTVW17]. In fact, *the only cryptographic primitive we need is pseudorandom functions (PRFs)*, which can be accelerated through the AES-NI instruction sets available in most modern processors. Moreover, our construction is completely *self-contained* and we need not invoke any existing PIR scheme as a building block.

Optimality. Corrigan-Gibbs, Henzinger and Kogan [CHK22] showed a lower bound for any adaptive PIR scheme without server-side preprocessing. In particular, their lower bound states that if the client stores S bits and the amortized server computation time is T, it must be that $\mathsf{ST} = \Omega(n)$. Our scheme matches this lower bound (up to poly-logarithmic factors). However, the per-query $\Theta(\sqrt{n})$ communication cost in our scheme is not theoretically optimal – previous theoretical work [ZLTS23, LP23a] can achieve poly-logarithmic communication per query.

Open-source implementation and evaluation results. We implemented PIANO in Go. Given its simplicity, the core contains only around 800 lines of code. We also provide a reference implementation (for tutorial purposes) that contains only around 160 lines of code. Both our full

implementation and the tutorial implementation are open sourced [1].

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

We conducted an experiment on a 100GB database with a 60ms RTT coast-to-coast connection. In particular, we chose an 100GB database to roughly match the size of a typical DNS database. Our scheme achieves **73ms** response time, whereas SimplePIR suffers from 11s or higher response time[2]. This represents over $\mathbf{150}\times$ speedup relative to SimplePIR. Since our improvement is asymptotical in nature, the speedup will only become larger as the database size grows. We also ran a non-private baseline for the same scenario, and the response time is 61ms. Therefore, our slowdown w.r.t. the non-private baseline is only 20%.

Theoretical Contributions. Although our work focuses on making PIR practical, our result may be of interest from a theoretical perspective, since this is the first time we know how to construct single-server PIR with sublinear server computation from only one-way functions (OWF). Section 2.6 provides theoretical comparison with additional related work.

2.2 Main Construction

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

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

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

1. **Primary table**: contains $\widetilde{O}(\sqrt{n})$ entries, where $\widetilde{O}(\cdot)$ hides polylogarithmic factors. The i-th entry in the hint table contains
   - A random set S of $\sqrt{n}$ indices chosen according to the aforementioned distribution;
   - The parity bit $p = \oplus_{i \in S} \mathsf{DB}[i]$.
2. **Replacement entries**: for each chunk $j \in \{0, 1, \sqrt{n}-1\}$, store $\widetilde{O}(1)$ entries of the form $(i, \mathsf{DB}[i])$ where each i is a randomly sampled index from chunk j.

[1] `https://githubimplementation.com/wuwuz/Piano-PIR-new`

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

3. **Backup table** (needed for multiple queries): for each chunk $j \in \{0, 1, \sqrt{n} - 1\}$, store $\widetilde{O}(1)$ entries of the form (S, p), where S is a random set sampled according to the aforementioned distribution, and $p = \oplus_{i \in S \setminus \{S[j]\}} \mathsf{DB}[i]$. In other words, p is the parity of the database bits at all indices in S except the index corresponding to the j-th chunk.

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

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

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

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

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

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

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

To support multiple queries while ensuring security, we can make the following simple modification to the scheme. Whenever an entry (S, p) is consumed from the primary table during a query for $\mathsf{DB}[x]$, the client grabs the next unconsumed entry (S', p') from the backup table corresponding to $\mathsf{chunk}(x)$. It replaces the consumed entry with $(S'\langle \mathsf{chunk}(x) \to x \rangle, p' \oplus \mathsf{DB}[x])$

where[3] $S'\langle \mathsf{chunk}(x) \to x \rangle$ is otherwise the same as S' except for replacing $S'[\mathsf{chunk}(x)]$ with x. Observe that the consumed entry is a random set subject to containing x, and its replacement is also a random set subject to containing x. Therefore, the distribution of the sets in the primary table is unaffected.

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

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

Next, we can get rid of the "random query" assumption in the following way, and the resulting scheme supports $Q = \widetilde{O}(\sqrt{n})$ arbitrary queries. Observe that the "random query" assumption is needed only for load balancing. Imagin the server applies a pseudorandom permutation (PRP) to all indices of the database upfront. We may assume that this permutation is independent of the queries. The server publishes the PRP key, and the client is now able to compute the index of the query in the shuffled database. If the PRP key is not sampled honestly, it will not affect privacy, but may affect correctness (which is impossible anyway if the server is malicious).

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

Detailed description. Figure 2.1 gives a detailed description of our scheme for supporting $\widetilde{O}(\sqrt{n})$ random, distinct queries. As mentioned, it is easy to upgrade such a scheme to one that supports unbounded number of arbitrary queries.

In Figure 2.1, we use the following notation for describing pseudorandom sets. Henceforth let PRF denote a pseudorandom function whose output is in the range $\{0, 1, \ldots, \sqrt{n} - 1\}$, and let sk denote a PRF key.

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

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

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

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

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

Offline preprocessing.

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

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

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

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

Figure 2.1: Detailed description of PIANO .

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

2.3 Formal Definitions and Security Proofs

2.3.1 Notations

We denote the set $\{1, \ldots, n\}$ as $[n]$. DB denotes a database and we keep the convention that n is its entry number. The indices of DB start at 0 and end at $n-1$. Denote $\mathsf{DB}[x]$ for the x-th entry. For any i, j, $\mathsf{DB}[i:j]$ denotes a slice of the database that contains $(\mathsf{DB}[i], \mathsf{DB}[i+1], \ldots, \mathsf{DB}[j-1])$. Denote $\oplus$ as the xor operation.

For any distribution $\mathbf{D}$, we write $x \xleftarrow{\$} \mathbf{D}$ as sampling x according to $\mathbf{D}$. For any set S, we write $x \xleftarrow{\$} S$ to denote that x is sampled uniform-randomly from S. For $i \in \{0, 1, \ldots, |S|-1\}$, we write $S[i]$ as the $(i+1)$-th smallest element in the set.

For a vector $\Delta = (\delta_0, \ldots, \delta_{m-1})$, we write Δ_{-i} as the vector $(\delta_0, \ldots, \delta_{i-1}, \delta_{i+1}, \ldots, \delta_{m-1})$ that removes δ_i and compacts the remaining coordinates.

2.3.2 Pseudorandom Function

A Pseudorandom Function $PRF : \{0,1\}^\lambda \times \{0,1\}^\ell \to \{0,1\}^m$ takes in a λ-bit length key sk and then $\mathsf{PRF}_{\mathsf{sk}}(\cdot)$ takes in an ℓ-bit string input and outputs an m-bit pseudorandom string. We list its syntax below.

- $\mathsf{PRF.Gen}(1^\lambda)$: given the security parameter λ, output a secret key sk sampled uniformly random from $\{0,1\}^\lambda$.
- $\mathsf{PRF}_{\mathsf{sk}}(x)$: given the secret key sk and an input $x \in \{0,1\}^\ell$, output an m-bit pseudorandom string.

Definition 2.3.1. A Pseudorandom Function $PRF : \{0,1\}^\lambda \times \{0,1\}^\ell \to \{0,1\}^m$ satisfies pseudorandomness, if for sk sampled uniformly random from $\{0,1\}^\lambda$, for any function $\mathcal{F}$ sampled uniformly at random from the set of functions mapping $\{0,1\}^\ell \to \{0,1\}^m$, for any PPT adversary $\mathcal{A}$, there exists a negligible function $\mathsf{negl}(\lambda)$ such that

$$\left| \Pr\left[\mathcal{A}^{\mathcal{O}_{\mathcal{F}}(\cdot)} = 1\right] - \Pr\left[\mathcal{A}^{\mathcal{O}_{\mathsf{PRF}_{\mathsf{sk}}}(\cdot)} = 1 : \mathsf{sk} \xleftarrow{\$} \{0,1\}^\lambda\right] \right| \leq \mathsf{negl}(\lambda).$$

Here, $\mathcal{O}_{\mathcal{F}}(\cdot)$ denotes oracle access to the function $\mathcal{F}$.

2.3.3 Definitions

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

- **Offline setup.** The offline setup phase is run only once upfront. The client receives nothing as input, and the server receives a database $\mathsf{DB} \in \{0,1\}^n$ as input. The client may interact with the server and store some hints in its local storage. For simplicity, we assume the entries in DB are 1-bit[4].
- **Online queries.** This phase can be repeated multiple times. Upon receiving an index $x \in \{0, 1, \ldots, n-1\}$, the client sends a single message to the server, and the server responds with a single message. The client performs some computation and outputs an answer $\beta \in \{0,1\}$.

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

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

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

Definition 2.3.2 (Privacy of PIR)**.** We say that a single-server PIR scheme satisfies privacy iff there exists a probabilistic polynomial-time simulator $\mathsf{Sim}(1^\lambda, n)$, such that for any probabilistic polynomial-time adversary $\mathcal{A}$ acting as the server, any polynomially bounded n and Q, any $\mathsf{DB} \in \{0,1\}^n$, $\mathcal{A}$'s views in the following two experiments are computationally indistinguishable:

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

2.3.4 Proofs

We now provide the proofs of privacy and correctness of our PIR scheme. We also provide the performance analysis.

Theorem 2.3.3 (Privacy)**.** *Our PIR scheme satisfies privacy (i.e., Theorem 2.3.2).*

Proof Sketch. We can first replace the PRFs with true random functions. Due to the pseudorandomness of the PRF, it is indistinguishable to the adversary.

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

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

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

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

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

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

This is identically distributed as a uniformly random set.

The crux of the proof is to extend this simple calculation to the case where the client has multiple local sets.

Proof. Denote the distribution $\mathbf{D}_n$ as sampling a random set that draws a random element from each of the $\sqrt{n}$ chunks. The Ideal experiment is as follows.

Ideal:

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

We define a hybrid experiment Hyb_1 as follows:

Hyb_1:

- *Offline*. $\mathcal{A}$ receives the streaming signal.
- *Online*. For each online round t, $\mathcal{A}$ chooses the query x_t. The client samples a random set $S \xleftarrow{\$} \mathbf{D}_n$ conditioned on $x_t \in S$ and also a random index r from x_t's chunk. The client sends $(S/\{x_t\}) \cup \{r\}$ to the server (received by $\mathcal{A}$).

It should be straightforward to prove the distributions of $\mathcal{A}$'s views in Ideal and Hyb_1 are identical. In Hyb_1, since the $\mathbf{D}_n$ chooses a random element in each chunk independently, even conditioned on containing any particular x, the remaining elements are still independent and uniformly random within their chunks. Therefore, after replacing x with a random index from the same chunk, the edited set $(S/\{x_t\}) \cup \{r\}$ is identically distributed as $\mathbf{D}_n$. So the views are indeed indentically distributed in these two experiments.

Now we define a hybrid experiment Hyb_2:

Hyb_2:

- *Offline*. $\mathcal{A}$ receives the streaming signal. The client samples random sets $S_1, \ldots, S_{M_1} \xleftarrow{\$} \mathbf{D}_n^{M_1}$.
- *Online*. For each online round t, $\mathcal{A}$ chooses the query x_t:
  1. The client finds the smallest index $j \in [M_1]$ that $x_t \in S_j$. Denote the set as S^*. If no such index is found, the client samples a set $S^* \xleftarrow{\$} \mathbf{D}_n$ conditioned on $x_t \in S^*$.
  2. The client samples a random index r from x_t's chunk. The client sends $(S^*/\{x_t\}) \cup \{r\}$ to the server (received by $\mathcal{A}$).
  3. The client samples $S' \xleftarrow{\$} \mathbf{D}_n$ conditioned on $x_t \in S'$. If the client finds a set that contains x_t earlier, replace the j-th set in the local sets with S'.

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

Lemma 2.3.4. *In Hyb_2, for every online query x_t, even conditioned on $\mathcal{A}$'s view over the first $t-1$ queries,*

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

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

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

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

Consider the following experiment Expt:

- The client samples $u \in [M_1 + 1]$ such that $u = i$ with probability α_i.
- $\forall j < u$, the client samples $S_j \xleftarrow{\$} \mathbf{D}_n$ subject to $x_t \notin S_j$.
- For u, the client samples $S_u \xleftarrow{\$} \mathbf{D}_n$ subject to $x_t \in S$. The client samples a random index r from x_t's chunk. The client sends $(S_u/\{x_t\}) \cup \{r\}$ to the server (received by $\mathcal{A}$).
- For $j \in [u+1, M_1]$, the client samples $S_j \xleftarrow{\$} \mathbf{D}_n$.
- The client samples $S'_u \xleftarrow{\$} \mathbf{D}_n$ subject to $x_t \in S'_u$. If $u \leq M_1$, the client replaces S_u with S'_u.

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

Notice that Hyb_2 is close to the real experiment. We define hybrid experiment Real^* as follows:

- *Offline*. $\mathcal{A}$ receives the streaming signal. The client samples random sets $S_1, \ldots, S_{M_1} \xleftarrow{\$} \mathbf{D}_n^{M_1}$ and also $\overline{S}_{i,j} \xleftarrow{\$} \mathbf{D}_n$ for $i \in \{0, 1, \ldots \sqrt{n} - 1\}$, $j \in [M_2]$.
- *Online*. For each round t, $\mathcal{A}$ chooses the query x_t:

1. The client finds the smallest index $j \in [M_1]$ that $x_t \in S_j$. Denote the set as S^*. If no such index is found, the client samples a set $S^* \xleftarrow{\$} \mathbf{D}_n$ conditioned on $x_t \in S^*$.
2. The client samples a random index r from x_t's chunk.
3. The client sends $(S^*/\{x_t\}) \cup \{r\}$ to the server.
4. Let $i^* = \mathsf{chunk}(x_t)$. If there is an unconsumed set in $\overline{S}_{i^*,1}, \ldots, \overline{S}_{i^*,M_2}$, say $\overline{S}_{i^*,j}$, the client consumes it and set $S' = (\overline{S}_{i^*,j}/\{\overline{S}_{i^*,j}[i^*]\}) \cup \{x_t\}$. Otherwise, the client samples $S' \xleftarrow{\$} \mathbf{D}_n$ conditioned on $x_t \in S'$. If the client finds a set that contains x_t earlier, replace the j-th set in the local sets with S'.

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

Finally, Real^* is just a rewrite of Real removing irrelevant terms and replacing the PRF with real randomness. By a reduction to the pseudorandomness of the PRF, Real^* and Real are computationally indistinguishable. ■

Theorem 2.3.5 (Correctness)**.** *Assume n is bounded by $\mathsf{poly}(\lambda)$ and $\mathsf{poly}(\kappa)$. Let $\alpha(\kappa)$ be any super-constant function, i.e., $\alpha(\kappa) = \omega(1)$. Further, assume the queries are chosen independently of the PRP. Given $M_1 = \sqrt{n}\ln\kappa\alpha(\kappa)$, $M_2 = 3\ln\kappa\alpha(\kappa)$, all the $Q = \sqrt{n}\ln\kappa\alpha(\kappa)$ queries will be answered correctly with probability at least $1 - \mathsf{negl}(\lambda) - \mathsf{negl}(\kappa)$ for some negligible function $\mathsf{negl}(\cdot)$.*

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

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

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

We analyze the second type of failure events – it only happens when the client makes more than M_2 queries in one group. Since the client is making $\sqrt{n}\ln\kappa\alpha(\kappa)$ queries and there are $\sqrt{n}$ groups, we can use a standard balls-into-bins argument. For $t \in [Q], i \in \{0, 1, \ldots, \sqrt{n}-1\}$, define the random variables $Y_{t,i} \in \{0, 1\}$ such that $Y_{t,i} = 1$ if and only if the t-th query locates in the i-th chunk. Denote $X_i = Y_{1,i} + \cdots + Y_{t,i}$ as the number of queries located in the i-th chunk. We know $\mathbf{E}[Y_{t,i}] = 1/\sqrt{n}$ and $\mathbf{E}[X_i] = \ln\kappa\alpha(\kappa)$. Since we are taking the randomness of the permutation and assuming the queries have no duplication, $Y_{1,1}, \ldots, Y_{Q,1}$ are negatively

correlated. With the Chernoff bound for negatively correlated variables, we know that

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

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

For the first type of failure events, by Lemma 2.3.4, the local sets $S_1, \ldots, S_{M_1}$ (i.e., the set represented by the keys) will be identically distributed as $\mathbf{D}_n^{M_1}$ and each set will contain the query with probability $1/\sqrt{n}$. So for a particular query x, the probability of no set containing x is

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

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

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

■

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

Theorem 2.3.6 (Efficiency)**.** *Let $\alpha(\kappa)$ be any super-constant function, i.e., $\alpha(\kappa)=\omega(1)$. The single-server PIR scheme only needs a one-time offline phase and supports an unbounded number of queries. It achieves the following performance bounds:*

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

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

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

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

For the online phase, the client needs to search for the hint that contains the query. Since each set contains the query with probability $1/\sqrt{n}$ and each membership testing takes $O_\lambda(1)$ time, the expected searching time is $O_\lambda(\sqrt{n})$. Other operations all take $O_\lambda(\sqrt{n})$ time. The client then sends

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

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

■

Discussion on the correctness guarantee. A follow-up work by Ren et al. [RMS24] points out an attack exploiting the correctness failure in our PIR scheme. They mentioned that in a case where the client is "adversarially influenced" and chooses the queries repeatedly in one single chunk (with knowledge of the PRP), the client will run out of backup hints and have a correctness failure.

Indeed, our correctness guarantee is based on the assumption that the client's queries are independent to the PRP. However, their definition of "adversarially influenced" client is not well-defined, and the motivation behind such a Denial-of-Service (DoS) attack is unclear. We make the following clarifications:

First, we assume the server is semi-honest. If the server is malicious, it can cause a correctness failure by simply returning incorrect answers to the client, which also applies to their proposed PIR scheme.

Second, they assume the PRP is public knowledge, which is not the case in our scheme. We clarify that the PRP should be a secret known only between the server and the client and should not be known to a third party, in case such failure attack is a concern.

Finally, they argue that if the client's behaviors depend on the correctness of the returned answers and could be observed by an adversary, a correctness attack may result in a privacy failure. However, this is not an issue in our scheme, as we show that even when the client's queries are arbitrarily and adaptively chosen, privacy still holds (i.e., the messages received by the server can be fully simulated). If there are other side effects on the correctness failure (e.g. the timing of the next query depends on the correctness of the previous query), it is beyond the scope of our work and the standard PIR research literature.[5]

2.4 Evaluation

Our evaluation aims to answer the following questions:

1. How does PIANO perform compared to a state-of-art single server PIR scheme (SimplePIR [HHCG$^+$22])? (Section 2.4.3)
2. How does PIANO perform compared to a non-private retrieval baseline? (Section 2.4.4)

[5]In fact, there are many other side effects that could happen in a real-world deployment and might lead to privacy issues. Consider a scenario where the client will wait for one minute until the next query if the current query is on the first half of the database, and will wait for 10 minutes if the current query is on the second half. This behavior could indeed lead to privacy issues, and side effects like this should be considered in future research.

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

2.4.1 Implementation

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

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

Parameters. We note that the performance of our scheme is more affected by the size of each set rather than the size of each chunk. To this end, we set the chunk size to be $2\sqrt{n}$ and round it up to the nearest power of 2, which makes the modulo operation more efficient. The set size is computed accordingly. It does not affect the theoretical asymptotics of our protocol. We set $Q = \sqrt{n} \ln n$. We set the statistical security parameter κ to 40 and computational security parameter λ to 128. We adjust M_1, M_2 accordingly so that the failure probability is bounded by $2^{-\kappa} = 2^{-40}$ for all Q queries, matching the same failure probability as SimplePIR [HHCG+22]. We use 128-bit keys and use AES to instantiate the PRF. Our implementation uses a 64-bit integer to denote a database index and thus we can support sufficiently large databases.

Optimization. Instead of generating a λ-bit key for each hint, the client just needs to generate a λ-bit master secret key msk and a unique short tag tag_i (e.g. 32 bits) for the i-th hint. Then, the j-th offset of this hint will be $\mathsf{PRF}(\mathsf{msk}, \mathsf{tag}_i || j)$. In practice, we observe that this optimization reduces the storage by 30%.

2.4.2 Evaluation Setup

We evaluate PIANO and the baseline schemes on two AWS m5.8xlarge instances with 128GB of RAM. For our local area network experiments, we run the PIR scheme on a single machine. This simulates a scenario where the network is not the bottleneck. We also evaluate our scheme over a wide-area network. In this case, we place the server machine on the west coast and the client machine on the east coast. All communication is performed over TLS connections with a 2 Gbps network bandwidth. The round-trip time is around 60ms. All query costs are computed as the average over one thousand queries. We use the open-source implementation of SimplePIR provided by Henzinger et al. [HHCG+22]. We also implement a non-private database access scheme as the baseline that does not include caching or load balancing.

2.4.3 Experiments in a Local-Area Network

We first compare our protocol to the SimplePIR protocol for 1GB and 2GB databases of 8-byte entries. Because the open-source SimplePIR implementation does not support parallelization or

connections across servers, we only compare the protocols run on a single machine. We analyze the effect of network latency on our protocol in the following section. We also run our scheme with a 100GB database with 1.6 billion 64-byte entries. To the best of our knowledge, this is by far the largest database ever supported by any implementation of a single-server PIR scheme. Because the implementation of SimplePIR does not support databases of this size, we extrapolate the results by running their scheme for 1GB and 2GB sized databases of 64-byte entries, and extrapolating their performance to 100GB based on the asymptotic performance discussed in their paper.

	1GB($n = 2^{27}$)		**2GB**($n = 2^{28}$)		**100GB**($n \approx 1.68 \times 10^9$)	
	SimplePIR	PIANO	SimplePIR	PIANO	SimplePIR(*)	PIANO
			Preprocessing			
Client time	293s	629s/111s	608s	1471s/257s	425min	192min/32min
Communication	123MB	1GB	173MB	2GB	1.2GB	100GB
			Per query			
Online Time	131.6ms	3.0ms	219.5ms	3.4ms	10.9s	11.9ms
Online Comm.	238KB	32KB	338KB	64KB	2.3MB	100KB
Am. Offline Time	1.4 ms	2.9/0.5ms	2.9ms	4.6/0.8ms	29.6ms	13.2ms/2.2ms
Am. Offline Comm.	0.6KB	4.9KB	0.6KB	6.6KB	1.4KB	120.5KB
Client Storage	123MB	61MB	173MB	71MB	1.2GB	839MB

Table 2.1: Performance of our scheme and SimplePIR on 1GB, 2GB, and 100GB sized databases. The 1GB and 2GB databases have 8-byte entries, and the 100GB database has 64-byte entries. For preprocessing time in the format of 629s/111s, the former is with a single thread, and the latter is with 8 threads. "Am." is an abbreviation of "Amortized". "Comm." stands for communication cost. We report the online costs as well as the offline costs amortized over $Q = \sqrt{n} \ln n$ queries. *The results for SimplePIR with the 100GB database are extrapolated since their implementation cannot directly support such a large database.

Metrics and two modes of operation. Table 2.1 shows the costs of the queries as well as the one-time preprocessing. We divide the query costs into two parts, the online costs and the amortized offline costs. The former are on the critical path of the perceived response time, and the latter are the additional maintenance work needed when we deamortize the periodic preprocessing costs over the queries. In practice, there are two ways to run our scheme. The first method is to perform the preprocessing upfront *only once*, and the subsequent periodic preprocessing costs are deamortized to the queries in each window. The second method is to periodically rerun the preprocessing phase, e.g., at night or during periods of inactivity — in this case, the query phase need not pay the "amortized offline time" and "amortized offline communication".

Query costs. As seen in Table 2.1, our protocol outperforms SimplePIR in all online metrics, including client storage, communication, and online querying time. In particular, for medium-sized databases (1GB/2GB), we outperform SimplePIR by 43.9x - 64.6x in terms of online querying latency. This performance gain stems from the fact that our online computation is sublinear in the size of the database, while SimplePIR is fundamentally limited by the linear scan required by their protocol. As the database grows larger, the performance gap further increases.

For the 100GB database, PIANO only takes 11.9ms for the online query. On the other hand, the extrapolated online time for SimplePIR is 10.9s, resulting in a nearly $915\times$ performance gap.

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

The table also shows the effect of the parallelization for preprocessing costs (and similarly for amortized offline costs during the query phase). Since client computation is the bottleneck for the preprocessing, parallelizing the work using 8 threads significantly improves the running time. For example, for a 2GB database, parallelization with 8 threads improves the client's preprocessing time from 1471s to 257s.

2.4.4 Experiments over a Wide-Area Network

Next, we report our results for experiments conducted over a wide-area network. Recall that the round-trip network latency is around 60ms and the network bandwidth is 2Gbps (see Section 2.4.2). The effects of this greater network latency are seen in Table 2.2. Because the open-source SimplePIR implementation does not support connections across multiple machines, we extrapolate a *lower bound* for their querying time based on the summation of the extrapolated numbers in the previous section and the network latency.

	2GB($n = 2^{28}$)			**100GB**($n \approx 1.68 \times 10^9$)		
	Non-Private	SimplePIR	PIANO	Non-Private	SimplePIR	PIANO
			Preprocessing			
Client Time	-	608s	1472s/248s	-	425min	205min/45min
Communication	-	173MB	2GB	-	1.2GB	100GB
			Per query			
Online Time	59.8ms	279.3ms	64.0ms	61.0ms	10.9s	72.6ms
Online Comm.	16B	338KB	64KB	72B	2.3MB	100KB
Am. Offline Time	-	1.9ms	4.6ms/0.7ms	-	29.6ms	14.1ms/3.1ms
Am. Offline Comm.	-	0.6KB	6.6KB	-	1.4KB	120.5KB
Client Storage	-	173MB	72MB	-	1.2GB	839MB

Table 2.2: Performance of our scheme, SimplePIR and the non-private baseline on 2GB and 100GB sized databases. The 2GB database has 8-byte entries and the 100GB database has 64-byte entries. Numbers of the format 1472s/248s denote the performance with a single thread and 8 threads, respectively. "Comm." stands for communication cost. "Am." stands for "amortized".

When compared to the non-private baseline, our protocol has a $7\% - 20\%$ latency overhead. SimplePIR, on the other hand, has a 4.6x - 178.7x latency overhead.

2.4.5 Performance Breakdown

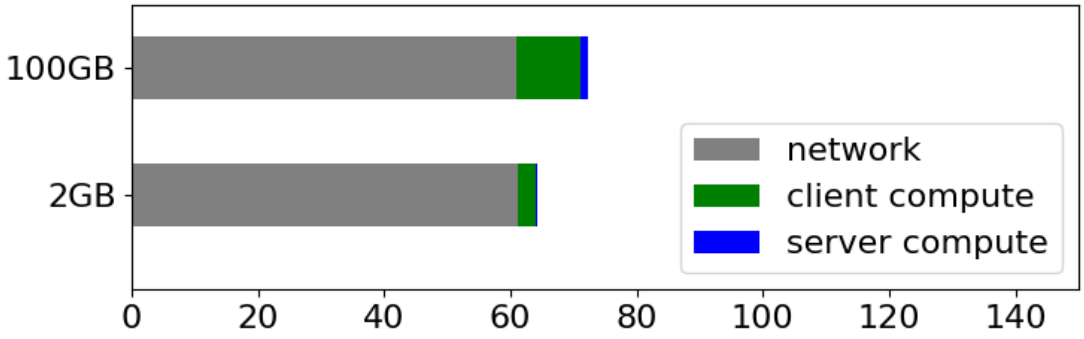


Figure 2.2: Cost breakdown

In Figure 2.2, we provide a detailed performance breakdown of the online time when running PIANO on a wide-area-network (the same setup as in Table 2.2). We see that the online time is mostly dominated by the network transmission time. The network transmission time is bounded by the physical distance – a 60ms RTT is required even when the payload is negligible. The client computation time is the second largest factor and it is dominated by the time to find a matched hint, which requires expected $O(\sqrt{n})$ PRF evaluations. The server computation time is much faster since the server-side algorithm only requires some RAM accesses with some non-cryptographic computation.

2.5 Variants and Extensions

2.5.1 A Variant of PIANO

We present a variant of PIANO in this section[6]. This variant has less storage but comes with more online communication.

Client side: A different approach to hide the query point. Recall that for each query x, the client will find a preprocessed set S containing x and the client wants to learn the parity $S/\{x\}$ (which is enough for it to learn $\mathsf{DB}[x]$). The main point is to learn the parity $S/\{x\}$ while preserving privacy. In the scheme presented in Section 2.2, the client will replace the query point x with some preprocessed replacement index r from the same chunk to ensure privacy. The query set will look random in the view of the server, and the client can learn the parity of $(S/\{x\}) \cup \{r\}$.

The variant takes a different method to hide the query point. Given a set S that contains one index from each chunk, we first write its "offset vector" as $\Delta = (S[0] \bmod \sqrt{n}, S[1] \bmod \sqrt{n}, \ldots, S[\sqrt{n}-1] \bmod \sqrt{n})$. For example, if the DB size is 16 and the set is $\{2, 4, 11, 13\}$, the

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

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

Figure 2.3: $O(\sqrt{n})$**-time server-side algorithm for one query.**

offset vector will be $(2, 0, 3, 1)$. Let j be x's chunk index. Consider the following offset vector that removes the offset of x and compacts the remaining offsets:

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

For example, removing the second offset from the offset vector of the last example will result in a vector of $(2, 3, 1)$. We observe that after removing the offset of x from the vector, the compacted remaining vector completely hides the information of x. Therefore, the client can directly send Δ_{-x} to the server.

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

$$S_i = \{\delta_0, \delta_1 + \sqrt{n}, \ldots, \delta_{i-1} + (i-1) \cdot \sqrt{n}, \perp, \delta_i + (i+1) \cdot \sqrt{n}, \ldots, \delta_{\sqrt{n}-2} + (\sqrt{n}-1) \cdot \sqrt{n}\},$$

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

Now we show that computing $q_0, \ldots, q_{\sqrt{n}-1}$ only takes $O(\sqrt{n})$ time. The naive approach is to recover the whole set for each guess and compute their parities directly. Since we have $\sqrt{n}$ guesses and each guess reconstructs a $(\sqrt{n}-1)$-size set, the computation time will be $O(n)$. However, observe that the symmetric difference between each two consecutive reconstructed sets S_i and S_{i+1} will only be two elements: $\delta'_i + (i+1) \cdot \sqrt{n}$ and $\delta'_i + i \cdot \sqrt{n}$. Therefore, computing q_{i+1} from q_i only takes two extra XOR operations. The algorithm can just compute q_0 directly in $O(\sqrt{n})$ time, and compute $q_1, q_2, \ldots, q_{\sqrt{n}-1}$ in sequence, each taking $O(1)$ time. So the total computation time is $O(\sqrt{n})$. We list the algorithm in Figure 2.3.

Comparison with the main scheme. The efficiency for this variant is nearly the same as the scheme presented before. The differences are as follows. On the one hand, in this variant, the client does not have to store all the replacement index-value pairs. However, since other parts of the storage already consume $O(\sqrt{n} \log \kappa \alpha(\kappa))$ space, the asymptotic storage cost stays the same. On the other hand, this variant has $O(\sqrt{n})$ download cost per query whereas the main scheme has $O(1)$ download cost. However, the upload costs are both $O(\sqrt{n})$ for the two schemes. So the

asymptotic communication cost stays the same as $O(\sqrt{n})$. In short, the two schemes have the same asymptotic behaviors and they provide a tradeoff between the local storage and the online communication (up to a constant factor).

Correctness and Privacy Proof. The correctness proof and the privacy proof are nearly the same as the proofs presented in Section 2.3.4. The failure probability analysis remains the same for the correctness proof. For the privacy proof, the only difference is in that the simulation strategy for the simulator. It now sends a uniform vector $\Delta' \xleftarrow{\$} \{0, 1, \ldots, \sqrt{n} - 1\}^{\sqrt{n}-1}$ instead of a random set. These two simulation strategies are both independent of the query index. Other parts of the privacy proof stay the same.

2.5.2 Supporting Key-Value Queries

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

2.5.3 Supporting Dynamic Databases

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

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

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

- $\mathsf{Init}(1^\lambda, \mathsf{DB})$: given a key-value store DB, initialize a PIR scheme.
- $\mathsf{Query}(\mathsf{key})$: the client wants to look up the value associated with some key key.
- $\mathsf{Insert}(\mathsf{key}, \mathsf{val})$: add a new entry $(\mathsf{key}, \mathsf{val})$ to the key-value store.
- $\mathsf{Update}(\mathsf{key}, \mathsf{val})$: update the value of an existing key to the specified new value.
- $\mathsf{Delete}(\mathsf{key})$: delete key from the key-value store.

Construction. Let n be the maximum size of the database. Let $Q = \sqrt{n} \log n \cdot \alpha(n)$ where $\alpha(\cdot)$ is an arbitrarily small super-constant function. We assume that $n = 2^L \cdot Q$. We will use a

hierarchical data structure Γ with logarithmically many levels denoted $\Gamma_0, \Gamma_1, \ldots, \Gamma_L$, where each level ℓ may either be empty or have a PIR scheme of size $2^\ell \cdot Q$.

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

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

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

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

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

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

Removing the known-n assumption. So far, we assumed that we know an upper bound n on the maximum number of entries in the key-value store. This assumption can easily be removed as follows. When we are rebuildling the last level L, if we discover that the number of entries has exceeded n, we update $n \leftarrow 2n$ as the new upper bound, i.e., increase the number of levels by 1.

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

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

- *Online query costs.* For each query, the online cost is the sum of the costs of querying $O(\log n)$ PIR schemes, each of size Q, $2Q$, ..., n. The total amortized communication is $O_\lambda(Q^{\frac{1}{2}} + (2Q)^{\frac{1}{2}} + \ldots + n^{\frac{1}{2}}) = O_\lambda(\sqrt{n})$. Using a similar calculation, the amortized online server computation is $O(\sqrt{n})$. The amortized client online computation is $O_\lambda(\sqrt{n})$.
- *Update costs.* Every Q updates, we need to perform the preprocessing phase for a Q-sized database. The amortized communication is $C_\lambda \cdot Q/Q = C_\lambda$, the amortized server time is C, and the amortized client time is $C_\lambda \cdot \log \kappa \cdot \alpha(\kappa)$ for some constant C and another parameter C_λ related to the security parameter λ. Every $2Q$ updates, we need to perform the preprocessing phase for a $2Q$-sized database. The amortized communication is C_λ, the amortized server time is C, and the amortized client time is $C_\lambda \log \kappa \cdot \alpha(\kappa)$. Every $4Q$ updates, we need to perform the preprocessing phase for a $4Q$-sized database, and so on. Therefore, in total, the amortized communication per update is $O_\lambda(\log n)$, the amortized server computation per update is $O(\log n)$, the amortized client computation per update is $O_\lambda(\log n \log \kappa \cdot \alpha(\kappa))$.
- *Space.* The client space is $O_\lambda(\sqrt{n} \log \kappa \cdot \alpha(\kappa))$. The server's storage is $O(n)$.

2.6 Additional Related Work

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

Single-server PIR schemes. Table 2.3 compares PIANO with existing single-server PIR schemes. Although we primarily focus on enhancing the practical performance of PIR, our proposed scheme is also of interest from a theoretical perspective. Notably, it is the first single-server PIR scheme that relies solely on one-way functions (OWF) and has sublinear server computation.

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

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

In the "global-preprocessing" model, also known as Doubly Efficient PIR (DEPIR), the server first preprocesses the database, and subsequently, it can answer queries with sublinear computation time. However, early works [CHR17, BIPW17] relied on non-standard assumptions or VBB obfuscation. A recent breakthrough work by Lin, Mook and Wichs [LMW23] presented a method to construct DEPIR based on the standard RingLWE assumption. In their approach, for any constant $\epsilon > 0$, the server preprocesses the database and stores a data structure of size $\widetilde{O}_\lambda(n^{1+\epsilon})$. Later, the server can answer queries in $\mathsf{poly}((\log n)^{1/\epsilon})$ time with $\mathsf{poly}((\log n)^{1/\epsilon})$ communication cost, where poly is a fixed polynomial.

Table 2.3: **Comparison of single-server PIR schemes.** m is the number of clients. n is the database size. d is the dimension of the hypercube representation of the DB. $\epsilon \in (0,1)$ is some suitable constant. "Comm." means communication per query. "CRA" means the composite residuosity assumption. ϕ-hiding is a number-theoretic assumption described in [CMS99]. "OLDC" means oblivious locally decodable codes. "VBB" means virtual-blackbox obfuscation. "OWF" means one-way function. The extra space denotes the client's extra storage, except for the schemes based on OLDC and also Lin, Mook and Wichs [LMW23], where the server stores the extra storage.

Scheme	**Assumpt.**	**Comm.**	**Per-query time**	**Extra space**
Theoretical Single-server PIR Schemes				
Standard [Cha04, CMS99, GR05]	CRA or ϕ-hiding or LWE	$\widetilde{O}(1)$	$O(n)$	0
[CHR17, BIPW17]	OLDC	n^ϵ	n^ϵ	mn
[BIPW17]	OLDC, VBB	n^ϵ	n^ϵ	n
[LMW23]	RingLWE	$\mathsf{poly}((\log n)^{1/\epsilon})$	$\mathsf{poly}((\log n)^{1/\epsilon})$	$\widetilde{O}_\lambda(n^{1+\epsilon})$
[CK20]	LWE	$\widetilde{O}_\lambda(\sqrt{n})$	$\widetilde{O}_\lambda(n)$	$\widetilde{O}_\lambda(\sqrt{n})$
[CHK22]	LWE	$\widetilde{O}_\lambda(\sqrt{n})$	$\widetilde{O}_\lambda(\sqrt{n})$	$\widetilde{O}_\lambda(\sqrt{n})$
[ZLTS23, LP23a]	LWE	$\widetilde{O}_\lambda(1)$	$\widetilde{O}_\lambda(\sqrt{n})$	$\widetilde{O}_\lambda(\sqrt{n})$
Practical Single-server PIR Schemes (with implementations)				
XPIR($d=2$)[MBFK16]	LWE	$\widetilde{O}_\lambda(\sqrt{n})$	$O(n)$	$\widetilde{O}_\lambda(1)$
PSIR[PPY18]	LWE	$\widetilde{O}_\lambda(\sqrt{n})$	$O(n)$	$\widetilde{O}_\lambda(\sqrt{n})$
FastPIR[AYA+21]	LWE	$\widetilde{O}_\lambda(n)$	$O(n)$	$O_\lambda(1)$
OnionPIR[MCR21]	LWE	$\widetilde{O}_\lambda(1)$	$O(n)$	$\widetilde{O}_\lambda(\sqrt{n})$
Spiral[MW22]	LWE	$\widetilde{O}_\lambda(1)$	$O(n)$	$O(1)$
FrodoPIR[DPC22]	LWE	$\widetilde{O}_\lambda(\sqrt{n})$	$O(n)$	$\widetilde{O}_\lambda(\sqrt{n})$
SimplePIR[HHCG+22]	LWE	$\widetilde{O}_\lambda(\sqrt{n})$	$O(n)$	$\widetilde{O}_\lambda(\sqrt{n})$
Ours	OWF	$O(\sqrt{n})$	$\widetilde{O}_\lambda(\sqrt{n})$	$\widetilde{O}_\lambda(\sqrt{n})$

Our scheme falls under the "client-preprocessing" model, also known as the subscription model. Corrigan-Gibbs and Kogan [CK20] were the first to present a construction under this model with $O(n)$ offline time and $\widetilde{O}_\lambda(\sqrt{n})$ online time. However, their scheme only supports a single query. Later, Corrigan-Gibbs, Henzinger and Kogan [CHK22] showed how to transform a single-query scheme into a $\sqrt{n}$-query scheme with polylogarithmic overhead using FHE. Zhou et al. [ZLTS23] and Lazzaretti and Papamanthou [LP23a] further extended the idea and achieved polylogarithmic communication cost.

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

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

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

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

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

Corrigan-Gibbs and Kogan [CK20, KCG21] proposed the client-side preprocessing idea to achieve $\widetilde{O}_\lambda(\sqrt{n})$ amortized per-query time under the two-server model with $\widetilde{O}_\lambda(\sqrt{n})$-size per-query communication and $\widetilde{O}_\lambda(\sqrt{n})$ client side storage. Shi et al. [SACM21] and TreePIR [LP23b] by Lazzaretti and Papamanthou further extended this idea and achieved polylogarithmic per-query communication.

The sublinear schemes in the multi-server model are practical. The PRP-based PIR [CK20] is implemented by Ma et al. [MZRA22]. Checklist [KCG21] and TreePIR [LP23b] also provided implementations.

TreePIR reported the best performance among these implementations, providing one implementation with polylogarithmic per-query communication cost by invoking a recursive scheme and another one with $O(\sqrt{n})$ per-query communication cost without the recursion. For an 8GB database with 2^{28} entries, the best amortized online time results reported in TreePIR are 23ms for the non-recursive scheme and 84ms for the recursive scheme. For comparison, our scheme has an amortized 20ms per-query time under the same setting with 4x local storage. The blowup of the local storage comes from the backup hints and the deamortization of the setup phase, which are inherently required for the single-server setting.

2.7 Limitations and Suitable Use Cases

The main limitation of PIANO is its communication cost: 1) the client has to download the whole database during the setup phase; 2) the online communication cost per query is $O(\sqrt{n})$. Compared to previous solutions like Zhou et al. [ZLTS23] and Lazzaretti and Papamanthou [LP23a] which have $\widetilde{O}_\lambda(1)$ communication overhead per query, the cost of PIANO is $O(\sqrt{n})$. However, we argue that this sacrifice is actually what makes our solution practical. The streaming preprocessing avoids the need for using FHE during the offline phase. Also, private programming of PRF is required to achieve $\widetilde{O}_\lambda(1)$ online bandwidth in previous solutions [ZLTS23, LP23a] and this primitive is only known in theory. By sending the whole edited set, we can do puncturing or programming without the need of complicated constructions. That being said, designing a truly practical single-server PIR with $\widetilde{O}_\lambda(1)$ communication overhead is one of the major future directions to be explored. We provide two possible use cases for PIANO .

- **Private Light-weight Blockchain Node**. When a light-weight blockchain node needs to fetch data from the blockchain, it makes queries to other full nodes. A light-weight node needs to make a verification pass over the blockchain history, and it has frequent queries, which makes PIANO a suitable privacy-preserving solution.
- **Private DNS Service**. DNS queries are frequently made, typically during specific periods (e.g., daytime) PIANO is also suitable for building a private DNS service. Users can preprocess during rest time and make queries during the online phase.

Chapter 3

QuarterPIR: Efficient Preprocessing PIR from List-Decodable Privately Programmable Pseudorandom Set

3.1 Overview

In this chapter, we still focus on the client-specific preprocessing PIR model introduced by Corrigan-Gibbs and Kogan [CK20], where a one-time preprocessing phase is performed between the server and a client before its first query. The client stores the preprocessing result locally, and the client then makes an unbounded number of private queries to the server's database.

Motivation. Our Piano work (see Chapter 2) and a subsequent work by Ren et al. [RMS24] showed that public-key cryptography is not necessary for achieving a preprocessing PIR scheme with $\widetilde{O}_\lambda(\sqrt{n})$ *online* computation and communication cost per query, while consuming $\widetilde{O}_\lambda(\sqrt{n})$ client storage. These schemes already achieve the optimal tradeoff in terms of client storage cost and the online computation due to the lower bounds by Corrigan-Gibbs, Henzinger, and Kogan [CHK22] and Yeo [PY22]. However, there is a significant gap in terms of their communication cost and the best theoretical PIR results that rely on public-key cryptography. In particular, the best known sublinear preprocessing PIR schemes with public-key cryptography (including our own work [ZLTS23] and the work of Lazzaretti and Papamanthou [LP23a]) can achieve $\widetilde{O}_\lambda(1)$ *online* communication per query, while the best known schemes without public-key cryptography (including Piano [ZPSZ24] and the work of Mughees et al. [RMS24]) can only achieve $\widetilde{O}_\lambda(\sqrt{n})$ online communication per query. Closing this gap is not only of theoretical interest, but also of practical importance, as all known concretely efficient sublinear PIR schemes [LP23b, ZPSZ24, RMS24] fall into this category.

Overview of results. We show new results that improve the state of our understanding regarding preprocessing PIR. In all of our constructions, the server only needs to store the original database and need not store any per-client state.

Main result 1. First, we construct a two-server preprocessing PIR scheme with asymptotically better communication than prior work, relying only on the existence of PRFs (which is equivalent

Table 3.1: **Comparison of single-server and two-server preprocessing PIR schemes (for unbounded queries)**. Any single-server scheme immediately implies a two-server scheme with the same performance bounds. n is the size of the database and m is the number of clients. The computation overhead counts both the client and the server's computation, and here we report the expected asymptotic costs. The server space counts only the extra storage needed on top of storing the original database.

Scheme	Assumpt.	Compute	Comm.	Space		# servers	Concrete eff.
				client	server		
With public-key cryptography							
[CHK22]	LWE	$\widetilde{O}_\lambda(\sqrt{n})$	$\widetilde{O}_\lambda(\sqrt{n})$	$\widetilde{O}_\lambda(\sqrt{n})$	$\widetilde{O}_\lambda(m \cdot n)^*$	1	✗
[ZLTS23, LP23a]	LWE	$\widetilde{O}_\lambda(\sqrt{n})$	$\widetilde{O}_\lambda(1)$	$\widetilde{O}_\lambda(\sqrt{n})$	$\widetilde{O}_\lambda(m \cdot n)^*$	1	✗
[LMW23]	Ring-LWE	$\mathsf{poly}((\log n)^{1/\epsilon})$	$\mathsf{poly}((\log n)^{1/\epsilon})$	0	$n^{1+\epsilon}$	1	✗
[SACM21]	LWE	$\widetilde{O}_\lambda(\sqrt{n})$	$\widetilde{O}_\lambda(1)$	$\widetilde{O}_\lambda(\sqrt{n})$	0	2	✗
[LP23b]	Various	$\widetilde{O}_\lambda(\sqrt{n})$	$\widetilde{O}_\lambda(1)$	$\widetilde{O}_\lambda(\sqrt{n})$	0	2	✓
Our work	Various	$\widetilde{O}_\lambda(\sqrt{n})$	$\widetilde{O}(\sqrt{n})$ offline $\widetilde{O}_\lambda(1)$ online	$\widetilde{O}_\lambda(\sqrt{n})$	0	1	✓
Without public-key cryptography							
[BIM00]	None	$O(n/\log^2 n)$	$O(n^{1/3})$	0	$O(n^2)$	2	✗
[BIM00]	None	$O(n^{1/2+\epsilon})$	$O(n^{1/2+\epsilon})$	0	$O(n^{1+\epsilon'})^{**}$	2	✗
[CK20]	OWF	$\widetilde{O}_\lambda(\sqrt{n})$	$\widetilde{O}(\sqrt{n})$	$\widetilde{O}_\lambda(\sqrt{n})$	0	2	✓
[KCG21]	OWF	$O(n)$	$\widetilde{O}_\lambda(1)$	$\widetilde{O}_\lambda(\sqrt{n})$	0	2	✓
[ZPSZ24, RMS24]	OWF	$\widetilde{O}_\lambda(\sqrt{n})$	$O(\sqrt{n})$	$\widetilde{O}_\lambda(\sqrt{n})$	0	1	✓
Our work	OWF	$O_\lambda(\sqrt{n})$	$O_\lambda(n^{1/4})$	$\widetilde{O}_\lambda(\sqrt{n})$	0	2	✓
Our work	OWF	$O_\lambda(\sqrt{n})$	$O(\sqrt{n})$ offline $O_\lambda(n^{1/4})$ online	$\widetilde{O}_\lambda(\sqrt{n})$	0	1	✓

$*$: In the unbounded query setting, some earlier works [ZLTS23, LP23a, CHK22] require that the next preprocessing is persistently piggybacked on the current window of $O(\sqrt{n})$ operations, and the preprocessing consumes $O_\lambda(n)$ server space per client to evaluate an $\widetilde{O}(n)$-sized circuit containing a sorting network with FHE.

$**$: $\epsilon' > 0$ depends on ϵ.

to the existence of one-way functions). Our result is stated in the following theorem.

Theorem 3.1.1 (Two-server preprocessing PIR with improved communication)**.** *Assume the existence of one-way functions. There exists a two-server preprocessing PIR scheme with $O_\lambda(n^{1/4})$ communication and $O_\lambda(n^{1/2})$ computation per query, while incurring $\widetilde{O}_\lambda(n^{1/2})$ client storage.*

In comparison with the prior work of Beimel et al. [BIM00], our Theorem 3.1.1 achieves significant asymptotic improvements in communication, computation, and server-side storage. On the other hand, we need to assume one-way functions whereas Beimel et al. [BIM00]'s schemes are information theoretic; further, we additionally require $\widetilde{O}(\sqrt{n})$ space on each client. However, our construction that gives Theorem 3.1.1 is simple and concretely efficient, which is another advantage over Beimel et al. [BIM00].

Main result 2. Second, we construct a new preprocessing PIR scheme in the single-server setting that improves the *online* communication in comparison with the state-of-the-art. In this theorem, we differentiate between online communication and offline communication. The online communication is the communication necessary for the client to obtain an answer to its query, so

it matters to the response time of the client. The offline communication is the cost of background maintenance work amortized to each query, and is not on the critical path of the client's response time.

Theorem 3.1.2 (Single-server preprocessing PIR with improved online communication)**.** *Assume the existence of one-way functions. There exists a single-server preprocessing PIR scheme with $O_\lambda(n^{1/4})$ online communication, $O(n^{1/2})$ offline communication, $O_\lambda(n^{1/2})$ server computation and $\widetilde{O}_\lambda(n^{1/2})$ client computation per query, while incurring $\widetilde{O}(n^{1/2})$ client storage.*

In comparison with our prior scheme PIANO, Theorem 3.1.2 improves the online communication cost from $\widetilde{O}(\sqrt{n})$ to $\widetilde{O}_\lambda(n^{1/4})$, while keeping all other costs the same. Moreover, recall that earlier works [CK20, CHK22], proved the time-space product lower bound, showing that the product of the client space and the online server time has to be at least linear in n. In this sense, Theorem 3.1.2 is tight (up to polylogarithmic factors) in terms of this time-space product.

Similar to Piano [ZPSZ24] and Mughees et al. [RMS24], our 1-server result uses the same model where the client is allowed to make a streaming pass over the database during preprocessing (while consuming small client space). Otherwise, we would encounter the well-known OT barrier [DCMO00].

We evaluate the concrete performance of our 1-server scheme in Section 3.6.

Additional results. While our main results focus on constructions in Minicrypt (i.e., without public-key cryptography), if we are willing to assume classical PIR with $\widetilde{O}_\lambda(1)$ communication (which is known from various assumptions such as LWE, Φ-hiding, Damgård-Jurik, DDH, QR) [CMS99, HHCG+22, MW22, DGI+19], our techniques would then give rise to a concretely efficient single-server PIR scheme with $\widetilde{O}_\lambda(1)$ *online* communication, $\widetilde{O}(\sqrt{n})$ offline communication and computation per query, consuming $\widetilde{O}_\lambda(\sqrt{n})$ client storage. In comparison, although earlier works [ZLTS23] and Lazzaretti and Papamanthou [LP23a] claim to achieve polylogarithmic (online and offline) communication, their schemes suffer from a significant drawback, that is, the server would have to persistently store at least n amount of state per client! Specifically, our work [ZLTS23] and Lazzaretti and Papamanthou [LP23a] require the preprocessing phase of the next epoch be piggybacked on the queries of the current epoch; however, their preprocessing phase requires that the server allocate at least n amount of space per client, to perform homomorphic evaluation of a circuit which is super-linear in size. So far, in the unbounded query setting, it is not known how to get polylogarithmic overall communication (including offline and online) per query *under any assumption*, when the server stores only the original database. We state this additional result in the following theorem.

Theorem 3.1.3. *Assume the existence of a classical single-server PIR scheme (i.e., without preprocessing) that enjoys $\widetilde{O}_\lambda(1)$ communication per query. Then, there exists a single-server preprocessing PIR scheme with $\widetilde{O}_\lambda(1)$ online communication, $\widetilde{O}_\lambda(\sqrt{n})$ computation, $\widetilde{O}(\sqrt{n})$ offline communication, requiring $\widetilde{O}_\lambda(\sqrt{n})$ client storage.*

3.1.1 Technical Highlights

The earlier work of Shi et al. [SACM21] showed that assuming the existence of a privately puncturable PRF [BLW17, BKM17, CC17, BTVW17], one can construct an efficient 2-server preprocessing PIR scheme with $\widetilde{O}_\lambda(\sqrt{n})$ computation per query and requiring $\widetilde{O}_\lambda(\sqrt{n})$ client

storage. Further, the communication per query is only polylogarithmically larger than the size of a punctured key, which can be as small as $\widetilde{O}_\lambda(1)$ using known constructions [BLW17, BKM17, CC17, BTVW17]. Unfortunately, the only known techniques for constructing a privately puncturable PRF [BLW17, BKM17, CC17, BTVW17] require two layers of fully homomorphic encryption, and it is not known whether privately puncturable PRFs can be built from only one-way functions. The elegant TreePIR work of Lazzaretti and Papamanthou [LP23b] showed how to replace the privately puncturable PRF with a weaker primitive called a "weakly privately puncturable PRF". Unfortunately, their approach relies on recursing on a classical PIR scheme for a $\sqrt{n}$-sized database, and because this database is *dynamically constructed* during the scheme, it is not possible to preprocess it. Therefore, Lazzaretti and Papamanthou [LP23b]'s techniques also require public-key cryptography.

Privately programmable pseudorandom set with list decoding. Our main contribution is a new primitive called *Privately Programmable Pseudorandom Set with List Decoding* (PPPS). Given a PPPS key sk, we can expand the key sk to a pseudorandom set denoted $\mathsf{Set}(\mathsf{sk})$ of size $\sqrt{n}$. Further, deciding whether any element in $\{0, 1, \ldots, n-1\}$ is in the set takes only constant time. Importantly, we can call a $\mathsf{Program}$ algorithm to program sk such that the new set is almost the same as the original $\mathsf{Set}(\mathsf{sk})$, except that only one element in the set is now changed to another specified element. The programmed key does not leak information about which element is programmed.

Our notion of PPPS is otherwise very similar to our previous work [ZLTS23], except that we make a relaxation on the correctness when decoding a programmed key — this relaxation is the crucial reason why we can construct it from only one-way functions, whereas our previous work's construction [ZLTS23] relies on LWE. More specifically, we do not require that one can correctly recover the programmed set given a programmed key sk'. Instead, we allow *list-decoding*, that is, given a programmed key sk', the decoding process outputs a list of candidate sets, among which one must be the true programmed set. Moreover, the list-decoding of our PPPS construction is structured, allowing succinct representation and efficient computation.

Using only one-way functions, we construct a PPPS scheme with list decoding for a pseudorandom set of size $\sqrt{n}$, where the programmed key has size $O_\lambda(n^{1/4})$.

Using such a PPPS scheme, we show how to get a two-server scheme with $O_\lambda(n^{1/4})$ communication and $\widetilde{O}_\lambda(n^{1/2})$ computation per query, using only $\widetilde{O}_\lambda(n^{1/2})$ client space (Theorem 3.1.1). Unlike TreePIR [LP23b], our scheme need not recurse on a classical PIR scheme, and thus we do not need public-key operations.

A new broken hint technique. To get our single-server scheme (Theorem 3.1.2), we encounter some further challenges. In particular, it would have been easy to make the scheme work if our PPPS scheme supported programming a key twice at two points. Specifically, in our construction, when the client consumes a pseudorandom set (represented by sk) in the hint table containing the current query x, it needs to replace the consumed entry with another randomly sampled PPPS key sk subject to containing the query x. One way to achieve this is to fetch an unconsumed key from a backup table, and program the key to contain x. However, later, when the client consumes this already-programmed key in another query y, it needs to program the point y to some other random point in order not to leak the query y.

Unfortunately, our PPPS construction does not support programming twice. Interestingly,

earlier works [ZLTS23, LP23a] also encountered a similar challenge of needing to program a key twice, but there it was resolved using different techniques that relied on the LWE assumption, which would not work in our setting.

The way we resolve the problem is to introduce a new technique of allowing *broken* entries in the hint table. Basically, if the client consumes some PPPS key sk in the hint table, it simply replaces the consumed sk with a new entry sampled according to the desired distribution (required for privacy). However, since the client did not perform any preparation work during the preprocessing phase for this new entry, consuming this entry later in a new query would result in an incorrect answer, i.e., the replaced entry is *broken*. Fortunately, we can amplify correctness through repetition. We defer the details to the subsequent technical sections.

Other applications of the broken hint technique. The broken hint technique can also lead to other interesting applications. For example, recall that TreePIR is a 2-server preprocessing scheme [LP23b]. With our new broken hint technique, we can convert TreePIR to a single-server scheme which enjoys the efficiency stated in Theorem 3.1.3.

Further improvements. The approach of using broken entries introduces a super-logarithmic blowup in the communication and computation costs, due to the repetition needed for correctness amplification. In Section 3.5, we suggest an improved scheme that avoids this super-logarithmic blowup and gets us the tighter bounds stated in Theorem 3.1.2, but the resulting scheme is somewhat more complex to describe.

3.2 Privately Programmable Pseudorandom Set with List Decoding (PPPS)

3.2.1 Definition

Distribution of set $\mathcal{D}_n$. We want to construct a pseudorandom set whose distribution emulates a set $S \subset \{0, 1, \ldots, n-1\}$ of size $\sqrt{n}$ sampled from the following distribution denoted $\mathcal{D}_n$ — we assume that n is a perfect forth power ($n^{1/4}$ is an integer):

- Divide the n elements into $\sqrt{n}$ chunks indexed with $0, 1, \ldots, \sqrt{n}-1$, where chunk i contains the elements $[\ell \cdot \sqrt{n},\ (\ell+1) \cdot \sqrt{n} - 1]$
- For each chunk $\ell \in \{0, 1, \ldots, \sqrt{n}-1\}$, sample a random offset $\delta_\ell \xleftarrow{\$} \{0, 1, \ldots, \sqrt{n}-1\}$.
- Output the following set $S := \{\ell \cdot \sqrt{n} + \delta_\ell\}_{\ell \in \{0,1,\ldots,\sqrt{n}-1\}}$.

Offset representation of a set. For convenience, in the rest of the section, we will always use an offset representation of a set, i.e., we will represent a set as

$$S := \{\delta_0, \ldots, \delta_{\sqrt{n}-1}\}$$

where each $\delta_i \in \{0, \ldots, \sqrt{n}-1\}$ represents the relative offset of the i-th element inside the i-th chunk.

Privately programmable pseudorandom set with list decoding. We introduce a new abstraction called a privately programmable pseudorandom set with list decoding that we utilize in our PIR

constructions that follow. Intuitively, this primitive provides an algorithm to generate a secret key that represents pseudorandom subset of $\{0, \ldots, n-1\}$ with a specific distribution. Further, the primitive allows, given a key for a pseudorandom set, to produce a key for pseudorandom set that is the same as the starting set except for being programmed at a particular location with a specified value – with the guarantee that the new key does not reveal the programmed location. Moreover, this primitive has a list decoding algorithm, that given a programmed key outputs a list of sets such that one of them is the one is the correct set that the key represents.

Formally, define a privately programmable pseudorandom set (PPPS) with list decoding which emulates the distribution $\mathcal{D}_n$:

- $\mathsf{sk} \leftarrow \mathsf{Gen}(1^\lambda, n)$: takes in the security parameter 1^λ, the size of the set n, and outputs a secret key sk.
- $S \leftarrow \mathsf{Set}(\mathsf{sk})$: takes in a secret key sk, and expands it to a random set S of size $\sqrt{n}$. We sometimes write $\mathsf{Set}(\mathsf{sk})[i]$ to denote the element in the i-th chunk for this set.
- $\mathsf{sk}', i \leftarrow \mathsf{Program}(\mathsf{sk}, \ell, \delta_\ell)$: takes in a secret key sk, a chunk identifier $\ell \in \{0, 1, \ldots, \sqrt{n}-1\}$, a desired offset δ_ℓ within the specified chunk ℓ, and outputs a programmed key sk', and some auxiliary information i that indicates which of the decoded sets will be correct.
- $\{S_0, \ldots, S_{L-1}\} \leftarrow \mathsf{ListDecode}(\mathsf{sk}')$: takes in a programmed key sk' and outputs a list of sets $S_0, S_1, \ldots, S_{L-1}$, such that one of them is the correctly programmed set corresponding to the key sk'.

Correctness. Correctness requires that for any $\lambda, n \in \mathbb{N}$, for any $\ell, \delta_\ell \in \{0, 1, \ldots, \sqrt{n}-1\}$, the following holds with probability 1: let $\mathsf{sk} \leftarrow \mathsf{Gen}(1^\lambda, n)$, $\mathsf{sk}', i \leftarrow \mathsf{Program}(\mathsf{sk}, \ell, \delta_\ell)$, $S_0, \ldots, S_{L-1} \leftarrow \mathsf{ListDecode}(\mathsf{sk}')$, it must be that S_i is equal to the $\mathsf{Set}(\mathsf{sk})$ but replacing the ℓ-th element with δ_ℓ instead.

Pseudorandomness. We say that a PPPS scheme emulates $\mathcal{D}_n$ iff the following two distributions are computationally indistinguishable:

- Sample $S \xleftarrow{\$} \mathcal{D}_n$ and output S;
- Sample $\mathsf{sk} \leftarrow \mathsf{Gen}(1^\lambda, n)$, output $\mathsf{Set}(\mathsf{sk})$.

Private programmability. We require that there exists a probabilistic polynomial time simulator Sim such that for any $n \in \mathbb{N}$ that is a perfect forth power and polynomially bounded in λ, any $\ell \in \{0, 1, \ldots, \sqrt{n}-1\}$, any index x that belongs to the ℓ-th chunk, the outputs of the following experiments be computationally indistinguishable:

- Real. Sample $\mathsf{sk} \leftarrow \mathsf{Gen}(1^\lambda, n)$ subject to $x \in \mathsf{Set}(\mathsf{sk})$, let $\delta_\ell \xleftarrow{\$} \{0, 1, \ldots, \sqrt{n}-1\}$, and let $\mathsf{sk}', \_ \leftarrow \mathsf{Program}(\mathsf{sk}, \ell, \delta_\ell)$, output sk'.
- Ideal. Output $\mathsf{Sim}(1^\lambda, n)$.

Efficiency. In our PIR scheme later, we need a programmed key sk' to have size at most $O_\lambda(n^{1/4})$. Further, the size of the decoded list $L = n^{1/4}$. Naïvely, since each set has size $\sqrt{n}$, it would take $n^{3/4}$ space to represent the L decoded sets. However, we want our scheme to satisfy a non-trivial notion of efficiency, that is, it takes only $O(\sqrt{n})$ space to represent all L decoded sets. Specifically, the compression is possible because the L decoded sets are correlated.

3.2.2 Construction

Intuition. In our construction, we will divide the $\sqrt{n}$ *chunks* into $n^{1/4}$ *superblocks* where the i-th superblock contains the i-th group of $n^{1/4}$ consecutive chunks.

To program a PPPS key in some chunk ℓ with the specified offset $\widetilde{\delta}$, we first expand the PPPS key to $n^{1/4}$ superblock keys denoted $k_0, \ldots, k_{n^{1/4}-1}$. Let i be the superblock corresponding to chunk ℓ. We then replace k_i with a randomly sampled superblock key $\widetilde{k}_i$. We then expand k_i into $n^{1/4}$ offsets denoted $\delta_0, \ldots, \delta_{n^{1/4}-1}$, one corresponding to each chunk contained in the i-th superblock. Suppose chunk ℓ corresponds to the j-th chunk within the i-th superblock. We then replace δ_j with the desired $\tilde{\delta}$. The programmed key is the combination of $k_0, \ldots, k_{i-1}, \widetilde{k}_i, k_{i+1}, \ldots, k_{n^{1/4}-1}$, and $\delta_0, \ldots, \delta_{j-1}, \widetilde{\delta}, \delta_{j+1}, \ldots, \delta_{n^{1/4}-1}$. Given this programmed key, we do not know which superblock should contain the expanded offsets $\delta_0, \ldots, \delta_{j-1}, \widetilde{\delta}, \delta_{j+1}, \ldots, \delta_{n^{1/4}-1}$. However, we can generate a list of $n^{1/4}$ candidate sets by plugging in the offsets $\delta_0, \ldots, \delta_{j-1}, \widetilde{\delta}, \delta_{j+1}, \ldots, \delta_{n^{1/4}-1}$ into each of the $n^{1/4}$ superblocks. One of them must be the true programmed set.

Detailed PPPS construction. Henceforth, let $\mathsf{PRF}_1 : \{0,1\}^\lambda \times \{0,1\}^{\frac{\log n}{4}} \to \{0,1\}^\lambda$, and $\mathsf{PRF}_2 : \{0,1\}^\lambda \times \{0,1\}^{\frac{\log n}{4}} \to \{0,1\}^{\frac{\log n}{2}}$ be two pseudorandom functions.

- $\mathsf{Gen}(1^\lambda, n)$: Sample a PRF_1 key sk and output sk.
- $\mathsf{Set}(\mathsf{sk})$:
  1. First, expand sk to $n^{1/4}$ superblock keys:
  $$\forall i \in \{0, \ldots, n^{1/4} - 1\} : \; k_i = \mathsf{PRF}_1(\mathsf{sk}, i) \tag{3.1}$$
  2. Next, for each superblock $i \in \{0, \ldots, n^{1/4} - 1\}$, compute the pseudorandom offset for each of its $n^{1/4}$ chunks, that is:
  $$\forall i, j \in \{0, \ldots, n^{1/4} - 1\} : \; \delta_{i,j} = \mathsf{PRF}_2(k_i, j) \tag{3.2}$$
  3. Define the alias $\delta_{i \cdot n^{1/4}+j} := \delta_{i,j}$, and output $S := \{\delta_\ell\}_{\ell \in \{0,\ldots,\sqrt{n}-1\}}$.
- $\mathsf{Program}(\mathsf{sk}, \ell, \widetilde{\delta})$:
  1. Expand sk to $n^{1/4}$ superblock keys denoted as $k_0, \ldots, k_{n^{1/4}-1}$ in Equation (3.1).
  2. Let $i := \lfloor \ell / n^{1/4} \rfloor$ be the superblock containing the ℓ-th chunk, let $j := \ell \mod n^{1/4}$ be the index of chunk ℓ within superblock i.
  3. Sample a fresh PRF key $\widetilde{k}_i$ to replace k_i with.
  4. For $j' \in \{0, 1, \ldots, n^{1/4} - 1\}$, compute $\delta_{j'} = \mathsf{PRF}_2(k_i, j')$.
  5. Output the following terms:
  $$\mathsf{sk}' := \left(\begin{array}{l} (k_0, \ldots, k_{i-1}, \widetilde{k}_i, k_{i+1}, \ldots, k_{n^{1/4}-1}), \\ (\delta_0, \ldots, \delta_{j-1}, \widetilde{\delta}, \delta_{j+1}, \ldots, \delta_{n^{1/4}-1}), \end{array} \right), \quad i$$
- $\mathsf{ListDecode}(\mathsf{sk}')$:

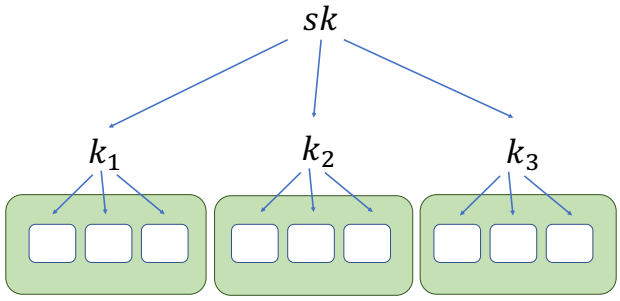


Figure 3.1: Two-layer set representation. The first layer key expands to $n^{1/4}$ superblock keys. Each superblock key further expands to $n^{1/4}$ offsets, one for each chunk in the superblock.

1. Parse $\mathsf{sk}' = (\{k_i\}_{i\in\{0,\ldots,n^{1/4}-1\}}, \{\delta^*_j\}_{j\in\{0,\ldots,n^{1/4}-1\}})$.
2. $\forall i, j \in \{0, \ldots, n^{1/4} - 1\}$, compute $\delta_{i,j}$ like in Equation (3.2), let S be the matrix $S := \{\delta_{i,j}\}_{i,j\in\{0,1,\ldots,n^{1/4}-1\}}$.
3. For $i \in \{0, \ldots, n^{1/4} - 1\}$, let S_i be the same as S except for substituting the i-th row with $\{\delta^*_j\}_{j\in\{0,\ldots,n^{1/4}-1\}}$. In other words,

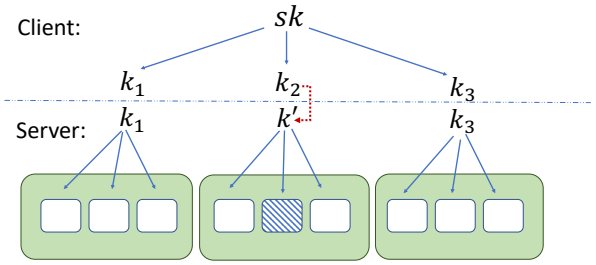
$$S_i := \begin{pmatrix} \delta_{0,0}, & \ldots, & \delta_{0,n^{1/4}-1}, \\ \ldots, & \ldots, & \ldots, \\ \delta_{i-1,0}, & \ldots, & \delta_{i-1,n^{1/4}-1}, \\ \delta^*_0, & \ldots, & \delta^*_{n^{1/4}-1}, \\ \delta_{i+1,0}, & \ldots, & \delta_{i+1,n^{1/4}-1}, \\ \ldots, & \ldots, & \ldots, \\ \delta_{n^{1/4}-1,0}, & \ldots, & \delta_{n^{1/4}-1,n^{1/4}-1}, \end{pmatrix}$$

4. Output $(\mathsf{Flatten}(S_0), \ldots, \mathsf{Flatten}(S_{n^{1/4}-1}))$ where $\mathsf{Flatten}$ outputs the vector obtained from concatenating all rows of the matrix. These are the offset representations of the $n^{1/4}$ candidate sets, each of size $\sqrt{n}$.

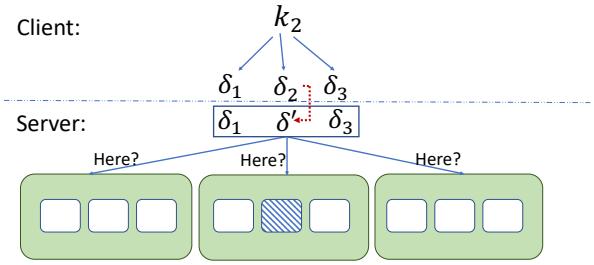
Size of programmed key and efficiency of $\mathsf{ListDecode}$**.** Clearly, the programmed key sk' output by $\mathsf{Program}$ has size $O_\lambda(n^{1/4})$. It is also easy to have a succinct representation of size $O(\sqrt{n})$ of all $n^{1/4}$ candidate sets output by $\mathsf{ListDecode}$. Specifically, one can first compute the common set S of size $\sqrt{n}$ (we abuse the notation that this set is derived from flattening the matrix S in $\mathsf{ListDecode}$). Then, the symmetric difference between the i-th candidate set and the common set S is just $2n^{1/4}$ elements (those elements in the i-th superblocks). So the succinct representation (and hence the efficient algorithm) of $\mathsf{ListDecode}$ takes $O_\lambda(\sqrt{n})$ space and time.

Efficient set membership. The above construction also supports $O_\lambda(1)$-time set membership query. Given a secret key sk that has not been programmed, to check if some element $x \in \mathsf{Set}(\mathsf{sk})$ or not, one simply has to check

$$\mathsf{PRF}_2(\mathsf{PRF}_1(\mathsf{sk}, \lfloor \ell/n^{1/4} \rfloor),\ \ell \bmod n^{1/4}) \stackrel{?}{=} x \bmod n^{1/2} \text{ where } \ell = \lfloor x/n^{1/2} \rfloor$$



Step 1: The client expands the PPPS key to $n^{1/4}$ superblock keys and replaces the key corresponding to x's superblock with a random key.



Step 2: The client expands the replaced superblock key to $n^{1/4}$ offsets and replaces x's offset to a random one. The server constructs the candidate sets by plugging these offsets into every superblock.

Figure 3.2: Illustration about how PPPS is used in our PIR schemes. The client programs the key and the server will decode the list of candidate sets.

3.2.3 Proof of Correctness

To see the correctness, let $\mathsf{sk} \leftarrow \mathsf{Gen}(1^\lambda, n)$, let $\mathsf{sk}', i^* \leftarrow \mathsf{Program}(\mathsf{sk}, \ell, \delta_\ell)$. Recall that sk' can be parsed as $\mathsf{sk}' = (\{k_i\}_{i\in\{0,\ldots,n^{1/4}-1\}}, \{\delta_i^*\}_{i\in\{0,\ldots,n^{1/4}-1\}})$, and by construction, we know that $i^* = \lfloor \ell/n^{1/4} \rfloor$ is the index of the superblock that contains the chunk ℓ. Let $j^* := \ell \mod n^{1/4}$. Let $S_\emptyset := \mathsf{Set}(\mathsf{sk})$, and we can view $S_\emptyset$ as a $n^{1/4} \times n^{1/4}$ matrix. The correct programmed set S^* is $S_\emptyset$ but replacing the element at index (i^*, j^*) with δ_ℓ.

Below, we show that the set S_i output by $\mathsf{ListDecode}$ is the same as S^*. By construction, in the $\mathsf{ListDecode}(\mathsf{sk}')$ algorithm, the intermediate set S is the same as $S_\emptyset$ except for the i^*-th row. Further, the i^*-th row of $S_\emptyset$ is the same as $\{\delta_j^*\}_{j\in\{0,\ldots,n^{1/4}-1\}}$ but replacing the j^*-th element with δ_ℓ. Additionally, the S_i output by $\mathsf{ListDecode}$ is obtained by replacing the i^*-th row of S with $\{\delta_i^*\}_{i\in\{0,\ldots,n^{1/4}-1\}}$.

3.2.4 Proof of Security

We now prove pseudorandomness and private programmability assuming the security of the underlying PRF_1 and PRF_2.

Pseudorandomness. Pseudorandomness follows directly from the pseudorandomness of the underlying PRFs.

Private programmability. We can consider the following sequence of hybrid experiments. Fix an arbitrary chunk identifier ℓ and an index x that belongs to the ℓ-th chunk. Throughout, let $i^* = \lfloor \ell/n^{1/4} \rfloor$, let $j^* = \ell \mod n^{1/4}$.

Experiment Real**.** Recall the definition of the real experiment. Sample a PRF key sk such that $\mathsf{PRF}_2(\mathsf{PRF}_1(\mathsf{sk}, i^*), j^*) = x \mod \sqrt{n}$. Let $\delta_\ell \xleftarrow{\$} \{0, 1, \ldots, \sqrt{n}-1\}$, and let $\mathsf{sk}', \_ \leftarrow \mathsf{Program}(\mathsf{sk}, \ell, \delta_\ell)$, output sk'.

Experiment Hyb**.** Same as Real except with the following modification: when executing the $\mathsf{Program}(\mathsf{sk}, \ell, \delta_\ell)$ algorithm, instead of using the $k_0, \ldots, k_{n^{1/4}-1}$ keys that are expanded using $\mathsf{PRF}_1(\mathsf{sk}, \cdot)$, sample $k_0, \ldots, k_{n^{1/4}-1}$ at random subject to $\mathsf{PRF}_2(k_{i^*}, j^*) = x \mod \sqrt{n}$.

Lemma 3.2.1. *Given that* PRF_1 *is secure,* Hyb *is computationally indistinguishable from* Real.

Proof. Suppose there is an efficient adversary $\mathcal{A}$ that can distinguish Real and Hyb with non-negligible probability. We can construct the following efficient reduction $\mathcal{B}$ which can distinguish a PRF (the family of PRF_1) from a random function with non-negligible probability. Basically, $\mathcal{B}$ is interacting with its own challenger who either answers queries using a PRF or using a truly random function. $\mathcal{B}$ will query its own challenger on the inputs $0, 1, \ldots, n^{1/4}-1$, and it will obtain $k_0, \ldots, k_{n^{1/4}-1}$ from its challenger. It will check if k_{i^*} satisfies the relation $\mathsf{PRF}_2(k_{i^*}, j^*) = x \mod \sqrt{n}$. If not, $\mathcal{B}$ aborts and outputs 0. Otherwise, it runs the Program algorithm where it plugs in the terms $k_0, \ldots, k_{n^{1/4}-1}$ as the superblock keys. It gives the resulting sk' to $\mathcal{A}$. Henceforth, we use $b = 0$ to denote the world in which $\mathcal{B}$'s challenger uses a truly random function, and we use $b = 1$ to denote the world in which $\mathcal{B}$'s challenger uses a randomly sampled PRF function. We use the notation $\Pr_b[\cdot]$ to denote the probability of events in world $b \in \{0, 1\}$. Let G be the good event that the relation $\mathsf{PRF}_2(k_{i^*}, j^*) = x \mod \sqrt{n}$ is satisfied.

$$\Pr_b[\mathcal{B} \text{ outputs } 1] = 0 \cdot \Pr_b[\overline{G}] + \Pr_b[\mathcal{A} \text{ outputs } 1 | G] \cdot \Pr_b[G]$$

We know that $\Pr_0[G] = 1/\sqrt{n}$ which is non-negligible. If the PRF is secure, then it must be that $|\Pr_1[G] - \Pr_0[G]| \leq \mathsf{negl}(\lambda)$ due to a straightforward reduction to PRF security. Therefore, we have that

$$\begin{aligned}
&\left|\Pr_1[\mathcal{B} \text{ outputs } 1] - \Pr_0[\mathcal{B} \text{ outputs } 1]\right| \\
&= \left|\Pr_1[\mathcal{A} \text{ outputs } 1 | G] \cdot \Pr_1[G] - \Pr_0[\mathcal{A} \text{ outputs } 1 | G] \cdot \Pr_0[G]\right| \\
&\geq \left|\Pr_1[\mathcal{A} \text{ outputs } 1 | G] - \Pr_0[\mathcal{A} \text{ outputs } 1 | G]\right| \cdot \frac{1}{\sqrt{n}} - \mathsf{negl}(\lambda)
\end{aligned}$$

Observe also that in world 0, conditioned on G, $\mathcal{A}$'s view in the experiment is identically distributed as Hyb. In world 1, conditioned on G, $\mathcal{A}$'s view in the experiment is identically distributed as Real. Therefore, the term

$$\left|\Pr_1[\mathcal{A} \text{ outputs } 1 | G] - \Pr_0[\mathcal{A} \text{ outputs } 1 | G]\right|$$

represents $\mathcal{A}$'s advantage in distinguishing Real and Hyb. We can now conclude that if $\mathcal{A}$ can distinguish Real and Hyb with non-negligible probability, then $\mathcal{B}$ can break PRF security with non-negligible probability. ∎

Experiment Ideal**.** The Ideal experiment is almost the same as Hyb except with the following modification: when outputting the sk', instead of using the $\delta_0, \ldots, \delta_{j^*-1}, \delta_{j^*+1}, \ldots, \delta_{n^{1/4}-1}$ terms derived from evaluating $\mathsf{PRF}_2(k_{i^*}, \cdot)$ at the points $0, 1, \ldots, j^* - 1, j^* + 1, \ldots, n^{1/4} - 1$, we now sample $\delta_0, \ldots, \delta_{j^*-1}, \delta_{j^*+1}, \delta_{n^{1/4}-1}$ at random from $\{0, \ldots, n^{1/2} - 1\}$ instead.

Observe that in the Ideal experiment, we no longer make use of knowledge of the query x. Therefore, the description of the Ideal experiment also uniquely specifies the simulator Sim we want to construct.

Lemma 3.2.2. *Given that* PRF_2 *is secure,* Ideal *is computationally indistinguishable from* Hyb.

Proof. It suffices to show that the following two probability ensembles are computationally indistinguishable for any fixed $\delta^* \in \{0, 1, \ldots, \sqrt{n} - 1\}$, and $j^* \in \{0, 1, \ldots, n^{1/4} - 1\}$.

1. Distr_0: Output a randomly sampled vector $\delta_0, \ldots, \delta_{n^{1/4}-1} \in \{0, 1, \ldots, \sqrt{n} - 1\}^{n^{1/4}}$.
2. Distr_1: Sample a PRF key k subject to $\mathsf{PRF}_2(k, j^*) = \delta^*$. Sample $\delta' \in \{0, 1, \ldots, \sqrt{n} - 1\}$ at random. For $j \in \{0, 1, \ldots, n^{1/4} - 1\}$, compute $\delta_j = \mathsf{PRF}_2(k, j)$. Output

$$\delta_0, \ldots, \delta_{j^*-1}, \delta', \delta_{j^*+1}, \ldots, \delta_{n^{1/4}-1}.$$

If there is an efficient adversary $\mathcal{A}$ that can distinguish between the above Distr_0 and Distr_1 with non-negligible probability, we can construct an efficient reduction $\mathcal{B}$ that can distinguish whether it is interacting with a random oracle or a randomly chosen PRF function. Basically, $\mathcal{B}$ sends the inputs $0, \ldots, n^{1/4} - 1$ to the oracle it is interacting with, and gets back $\delta_0, \ldots, \delta_{n^{1/4}-1}$. If $\delta_{j^*} \neq \delta^*$, then $\mathcal{B}$ aborts and outputs 0. Otherwise, it replaces δ_{j^*} with a random value from $\{0, \ldots, n^{1/2} - 1\}$ and gives the resulting vector to $\mathcal{A}$. Suppose $\mathcal{B}$ is interacting with a random oracle, then conditioned on the good event $\delta_{j^*} = \delta^*$, $\mathcal{A}$'s view is identically distributed as Distr_0. On the other hand, suppose $\mathcal{B}$ is interacting with a PRF, then conditioned on the good event $\delta_{j^*} = \delta^*$, $\mathcal{A}$'s view is identically distributed as Distr_1. The rest of the proof can be completed due to a similar probability calculation as Theorem 3.2.1. ∎

3.3 Our Two-Server PIR Scheme

3.3.1 Construction

Intuition. The scheme has three major components.

- *Preprocessing.* The client randomly samples $\tilde{O}(\sqrt{n})$ privately programmable pseudorandom sets, each of size $\sqrt{n}$. The client queries the right server for the sets' parities, storing them along with the keys. Moreover, the client queries the right server for the values of logarithmic numbers of randomly sampled indices for each $\sqrt{n}$-size chunk. Those entries are stored as the "replacement" entries.
- *Online Query.* Given a query x, the client finds a set S such that $x \in S$. The client then finds a replacement entry r that resides in the same chunk as x. The client privately programs the set, intending to change it from S to $(S/\{x\}) \cup \{r\}$. Once the client knows the parity for this new set, it can compute $\mathsf{DB}[x]$ because it already knows $\mathsf{DB}[r]$ and the parity for S. The client uses the PPPS programming function to program the set, and sends the programmed key sk' to the left server. The left server runs the list decoding algorithm, then computes and returns all $n^{1/4}$ candidate sets' parities. The client knows that there is one candidate parity corresponding to the correct set $(S/\{x\}) \cup \{r\}$, which is enough to compute the answer.
- *Refresh.* Each query consumes a set. After each query, the client just samples a new set conditioned on it containing the query x, and queries the right server for its parity with the same query technique mentioned above. The new set will replace the consumed set.

The detailed algorithm for bounded, random queries. We describe the detailed construction for $Q = \sqrt{n} \log \kappa \cdot \alpha$ random, distinct queries in Figure 3.3. We can easily extend such a scheme

Two-server scheme for $Q = \sqrt{n}\log\kappa \cdot \alpha$ queries

Offline preprocessing.

- *Hint table.* Let $M_1 = \sqrt{n}\log\kappa \cdot \alpha(\kappa)$. For each $i \in [M_1]$, sample a fresh PPPS key sk_i, send sk_i to the right server and receive a parity $p_i := \oplus_{j \in \mathsf{Set}(\mathsf{sk}_i)} \mathsf{DB}[j]$ back. Let $T := \{(\mathsf{sk}_i, p_i)\}_{i \in [M_1]}$ denote the client's *hint table*.
- *Replacement entries.* For each chunk $\ell \in \{0, \ldots, \sqrt{n} - 1\}$, repeat the following $M_2 = 3\log\kappa \cdot \alpha(\kappa)$ times: sample a random index $r_1 \in \{\ell \cdot \sqrt{n}, \ldots, (\ell+1) \cdot \sqrt{n} - 1\}$ in chunk ℓ, send r_1 to the left server, and receive $\mathsf{DB}[r_1]$. Store the tuple $(r_1, \mathsf{DB}[r_1])$.

  Similarly, for each chunk $\ell \in \{0, \ldots, \sqrt{n} - 1\}$, repeat the following M_2 times: sample a random index r_2 in chunk ℓ, send r_2 to the right server, and receive $\mathsf{DB}[r_2]$. Store the tuple $(r_2, \mathsf{DB}[r_2])$.

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

1. **Step 1: (Client Querying)**
   - Find the first entry (sk, p) in the hint table such that $x \in \mathsf{Set}(\mathsf{sk})$. [a]
   - Find the first unconsumed replacement entry $(r_1, \mathsf{DB}[r_1])$ retrieved from right server, such that r_1 is in $\mathsf{chunk}(x)$. [b]
   - $(\mathsf{sk}_1', j_1) \leftarrow \mathsf{Program}(\mathsf{sk}, \mathsf{chunk}(x), r_1 \bmod \sqrt{n})$.
   - Send sk_1' to the left server.
2. **Step 2: (Client Reconstructing)**
   - Receive $(\beta_{0,1}, \ldots, \beta_{n^{1/4}-1,1})$ from the left server.
   - Save the answer as $y = p \oplus \mathsf{DB}[r_1] \oplus \beta_{j_1,1}$.
3. **Step 3: (Client Refreshing)**
   - Sample sk_2 such that $x \in \mathsf{Set}(\mathsf{sk}_2)$.
   - Find the first unconsumed replacement entry $(r_2, \mathsf{DB}[r_2])$ retrieved from left server, such that r_2 is in $\mathsf{chunk}(x)$.
   - $(\mathsf{sk}_2', j_2) \leftarrow \mathsf{Program}(\mathsf{sk}_2, \mathsf{chunk}(x), r_2 \bmod \sqrt{n})$.
   - Send sk_2' to right server.
   - Receive $(\beta_{0,2}, \ldots, \beta_{n^{1/4}-1,2})$ from the right server.
   - Replace the hint (sk, p) with $(\mathsf{sk}_2, \mathsf{DB}[r_2] \oplus \beta_{j_2,2} \oplus y)$ in the table.
4. **Server Responding: (Same for Left and Right Server)**
   - Upon receiving sk', compute $(S_0, S_1, \ldots, S_{n^{1/4}-1}) \leftarrow \mathsf{ListDecode}(\mathsf{sk}')$.
   - Return $(\beta_0, \ldots, \beta_{n^{1/4}-1})$ to the client where $\beta_i = \oplus_{i \in S_b} \mathsf{DB}[i]$.

[a] In a rare case, if not found, let sk be a freshly sampled PPPS key subject to $x \in \mathsf{Set}(\mathsf{sk})$, and let $p = 0$.

[b] In a rare case, if such an r_1 is not found, let it be a random index in $\mathsf{chunk}(x)$, and use 0 whenever $\mathsf{DB}[r_1]$ is needed later.

Figure 3.3: Two-server preprocessing PIR with $O_\lambda(n^{1/4})$ communication, $O_\lambda(n^{1/2})$ computation based on PRFs.

to support unbounded, arbitrary queries using known techniques [ZLTS23]. For completeness, we explain how the extension works shortly after.

Efficiency. Observe that the list decoding produces $n^{1/4}$ candidate sets, each of size $\sqrt{n}$. Naively, expanding all sets and computing their corresponding parities takes $O_\lambda(n^{3/4})$ time. We are still going to rely on the fact that $\mathsf{ListDecode}$ has an $O(\sqrt{n})$-size succinct representation to optimize computation. Recall that we can first compute the common set of size $\sqrt{n}$. The symmetrical difference between each possible decoding set and the common set will only contain $2n^{1/4}$ elements. Therefore, to compute the parities for all $n^{1/4}$ possible sets, we first compute the parity for the common set S, which takes $O_\lambda(\sqrt{n})$ time. Then, it takes $O_\lambda(n^{1/4})$ time to enumerate the symmetrical difference between the i-th set and the common set, which suffices to compute the parity for the i-th set. So the total computation time will be $O_\lambda(\sqrt{n})$.

Supporting unbounded, arbitrary queries. For completeness, we review the techniques described in previous works about upgrading the PIR scheme from supporting Q random, distinct queries to supporting unbounded, arbitrary queries. We can easily get rid of the distinct query assumption in the following way: we require the client to store a local cache of size Q for caching the most recent Q queries. If the client wants a repeated query, it can lookup in the cache and make a distinct fake query.

Further, we can assume that the queries are random without loss of generality as follows: we can have the client and the servers agree on a small-domain pseudorandom permutation (PRP) [RY13] (which is implied by one-way functions [HMR12]) upfront and the servers can permute the database according to the PRP. Another option is to have one of the servers build the database as a key-value storage and use a cuckoo hash table [PR04, Yeo23] directly based on a PRF to locate the queries, and share it with the other server. Notice that in both implementations, the client can still make queries adaptively depending on the real query sequence and the responses, which is sufficient for practical usage. Then, as long as the client makes the queries independent of the randomness of the PRP/PRF, those queries can be considered as uniformly random. This assumption is only needed for the correctness.

Lastly, we can remove the bounded Q query assumption as follows: we use a pipelining trick suggested in earlier works [ZLTS23, ZPSZ24]. Essentially, we can spread the preprocessing for the next window of Q queries over the current window of Q queries.

Theorem 3.3.1. *Let $\alpha(\kappa)$ be any superconstant function. Suppose that $\mathsf{PRF}_1, \mathsf{PRF}_2$ are secure pseudorandom functions, and n is bounded by $\mathsf{poly}(\lambda)$ and $\mathsf{poly}(\kappa)$. The two-server scheme in Figure 3.3 that supports $Q = \sqrt{n}\log\kappa \cdot \alpha$ random and distinct queries is private, and correct with probability $1 - \mathsf{negl}(\lambda) - \mathsf{negl}(\kappa)$ for some negligible function $\mathsf{negl}(.)$. Further, it achieves the following performance bounds:*

- *$O_\lambda(\sqrt{n}\log\kappa\alpha(\kappa))$ client storage and no additional server storage;*
- ***Preprocessing Phase**:*
  - *$O_\lambda(n\log\kappa \cdot \alpha(\kappa))$ server time and $O_\lambda(\sqrt{n}\log\kappa \cdot \alpha(\kappa))$ client time;*
  - *$O_\lambda(\sqrt{n}\log\kappa \cdot \alpha(\kappa))$ communication;*
- ***Query Phase**:*
  - *$O_\lambda(\sqrt{n})$ expected client time and $O_\lambda(\sqrt{n})$ server time per query;*
  - *$O_\lambda(n^{1/4})$ communication per query.*

Therefore, the amortized communication per query is $O_\lambda(n^{1/4})$, and the amortized server computation and expected client computation per query is $O_\lambda(\sqrt{n})$.

Proof. We defer the privacy and correctness proofs to Section 3.3.2 and Section 3.3.3 respectively. Here, we focus on proving the efficiency claims.

In the proof, we abuse the notation and write $\alpha(\kappa)$ as α. The client stores $M_1 = \sqrt{n}\log\kappa \cdot \alpha$ number of PPPS keys, and $M_2 = 3\log\kappa \cdot \alpha$ number of replacement entries per chunk. Therefore, the space required is $O_\lambda(\sqrt{n}\log\kappa \cdot \alpha)$.

During the offline phase, the client sends M_1 PPPS keys to the right server, and sends M_2 indices per chunk to either server for constructing replacement entries. Therefore, the offline communication is bounded by $O_\lambda(\sqrt{n}\log\kappa \cdot \alpha)$. The right server needs to expand the sets for each PPPS key received and evaluate the xor-sums. Both servers need to return the bits for the replacement entries. Therefore, the total server computation is bounded by $O_\lambda(n\log\kappa \cdot \alpha)$.

During the query phase, the client sends one programmed PPPS key to each server, and the size of a programmed key is at most $O_\lambda(n^{1/4})$. Each server sends back the xor-sums of $n^{1/4}$ candidate sets. Each candidate set has size $n^{1/2}$, however, all $n^{1/4}$ candidate sets have a succinct representation of size only $n^{1/2}$, and servers can compute this succinct representation in time $O_\lambda(n^{1/2})$. Further, it is not hard to see that due to the structure of the candidate sets, the server can compute all $n^{1/4}$ xor-sums in time only $O(n^{1/2})$. Therefore, the servers' running time is bounded by $O_\lambda(n^{1/2})$ during each query. The client needs to find a matched hint, and compute $O(1)$ xor operations during each query. Its running time is dominated by the cost of finding a matched hint, which can be done by invoking the set membership operation for each of the M_1 hints. Using Theorem 3.3.2 and the pseudorandomness property of the PPPS, the expected number of hints checked until a key sk such that $\mathsf{Set}(\mathsf{sk})$ contains the current query is found is $O(\sqrt{n})$. Also, the expected number of tries in the rejection sampling during the refresh phase is also $\sqrt{n}$. Therefore, the client's expected running time per query is upper bounded by $O_\lambda(\sqrt{n})$. ∎

3.3.2 Privacy Proof

Suppose that the underlying PPPS scheme satisfies private programmability. Below, we prove the privacy of our two-server PIR scheme.

In the preprocessing phase, the server sends the sets $\mathsf{Set}(\mathsf{sk}_i)$ only to the right server, thereby no information about these sets is leaked to the left server. Similarly, no information about the indices r_1 is leaked to the right server and no information about the indices r_2 is leaked to the left server. We will first prove the lemma about the distribution of client's hint table, when the adversary controls either of the left or right server.

Lemma 3.3.2. *Recall that in each time step t, the adversary $\mathcal{A}$ adaptively chooses a query $x_t \in \{0, 1, \ldots, n-1\}$ for the client. At the end of each time step t, the client's hint table is distributed as a table of size M_1, where each entry is a freshly sampled PPPS key, even when conditioned on $\mathcal{A}$'s view so far.*

Proof. Suppose the above statement holds at the end of time step $t-1$. We prove that it still holds at the end of time step t. Since the hint table is distributed as a fresh randomly sampled table even when conditioned on $\mathcal{A}$'s view at the end of $t-1$, we may henceforth assume an arbitrary fixed query x_t. The distribution of the hint table before the t-th query can be equivalently rewritten as:

- First, determine the decision whether any of the M_1 entries contains the current query x_t, and if so, which is the first entry (denoted i^*) that contains x_t. If not found, we assume $i^* = M_1 + 1$.
- For each $i < i^*$, sample a random PPPS key subject to not containing x_t.
- For $i = i^*$, sample a random PPPS key subject to containing x_t.
- For each $i > i^*$, sample a random PPPS key.

Using the above interpretation, it is easy to see that the distribution of the hint table after the t-th query is unaltered, no matter which of the two servers $\mathcal{A}$ controls. ■

Left Server Privacy

We first construct the following simulator for proving left server privacy.

Simulator construction.

- During the preprocessing phase, for each chunk ℓ, sample M_2 random indices belonging to ℓ, send them to $\mathcal{A}$.
- During each query, call the simulator of the PPPS scheme which outputs sk', send sk' to $\mathcal{A}$.

Indistinguishability of Real **and** Ideal**.** We now prove the indistinguishability of the Real and Ideal assuming the private programmability of the underlying PPPS scheme.

First, due to Theorem 3.3.2, we can equivalently rewrite the Real experiment for the right server as follows: at the end of each time step, resample the entire hint table freshly at random before continuing to answer more queries. As a result, the view of $\mathcal{A}$ who controls the right server is distributed as:

- *preprocessing phase*. For each chunk ℓ, send M_2 random indices in chunk ℓ to $\mathcal{A}$.
- *Each time step* t.
  - sample a PPPS key sk at random subject to containing the query x_t; sample δ at random from $\{0, \ldots, \sqrt{n} - 1\}$.
  - call $\mathsf{sk}', \_ \leftarrow \mathsf{Program}(\mathsf{sk}, \mathsf{chunk}(x_t), \delta)$;
  - send sk' to $\mathcal{A}$.

One way to see this is to think of the distribution of the table as the equivalent distribution in the proof of Theorem 3.4.2. Further, observe that each r_2 mod $\sqrt{n}$ in the scheme is distributed randomly from the perspective of the left server, since these indices were only sent to the right server during the preprocessing phase.

Therefore, the rest of the proof follows due to a straightforward hybrid argument where we replace the programmed keys (denoted sk' earlier) sent to the right server in all time steps one by one with a simulated key, relying on the private programmability of the underlying PPPS.

Right Server Privacy

We first construct the following simulator for proving right server privacy.

Simulator construction.

- During the preprocessing phase, send M_1 randomly sampled PPPS keys to $\mathcal{A}$. Further, for each chunk ℓ, sample M_2 random indices in ℓ, send them to $\mathcal{A}$.

- During each query, call the simulator of the PPPS scheme which outputs sk', send sk' to $\mathcal{A}$.

Indistinguishability of Real **and** Ideal**.** We now prove the indistinguishability of the Real and Ideal assuming the private programmability of the underlying PPPS scheme.

The view of $\mathcal{A}$ who controls the right server is distributed as:

- *preprocessing phase*. Sample M_1 random PPPS keys, and send them to $\mathcal{A}$. Further, for each chunk ℓ, send M_2 random indices in chunk ℓ to $\mathcal{A}$.
- *Each time step* t.
  - sample a PPPS key sk at random subject to containing the query x_t; sample δ at random from $\{0, \ldots, \sqrt{n} - 1\}$.
  - call $\mathsf{sk}', \_ \leftarrow \mathsf{Program}(\mathsf{sk}, \mathsf{chunk}(x_t), \delta)$;
  - send sk' to $\mathcal{A}$.

To see the above, observe that each $r_1 \bmod \sqrt{n}$ in the scheme is distributed randomly from the perspective of the right server, since they were only sent to the left server during the preprocessing phase.

Therefore, the rest of the proof follows due to a straightforward hybrid argument where we replace the programmed keys (denoted sk' earlier) sent to the right server in all time steps one by one with a simulated key, relying on the private programmability of the underlying PPPS.

3.3.3 Correctness Proof

We show that with $Q = \sqrt{n} \log \kappa \cdot \alpha$ random, distinct queries, the probability of ever having correctness error is negligibly small. An error can happen if one of the following bad events takes place:

- *No matched hint*. During some query for x, no hint is found that contains the query x.
- *Depleting replacement entries*. During some query for x, there is no more replacement entry of the form $(r_1, \mathsf{DB}[r_1])$ or $(r_2, \mathsf{DB}[r_2])$ corresponding to $\mathsf{chunk}(x)$.

Below, we show that the probability of each bad event during a window of Q random, distinct queries is negligibly small.

Probability of no matched hint. Due to Theorem 3.3.2, for any fixed time step t, we can assume the client's hint table contains freshly sampled PPPS keys and is independent of the current query x_t. Due to the pseudorandomness property of the PPPS, the sets generated by the keys in the hint table are computationally indistinguishable from M_1 sets independently sampled from the distribution $\mathcal{D}_n$. Below we calculate the probability that a fixed element x_t is not in any of the M_1 sets sampled independently from $\mathcal{D}_n$ – the probability that x_t is not contained in any entry in the client's hint table can only be negligibly different.

The probability that one set sampled from $\mathcal{D}_n$ contains x_t is $1/\sqrt{n}$. Therefore, the probability that none of the M_1 sets contains x_t is $(1 - 1/\sqrt{n})^{M_1}$, and given the choice of $M_1 = \sqrt{n} \log \kappa \alpha(\kappa)$ where $\alpha(\kappa)$ is a super-constant function, this probability is negligibly small in κ.

Finally, taking a union bound over all polynomially many time steps the probability of ever not having a matched hint is negligibly small in κ.

Probability of depleting replacement entries. One can only deplete the replacement entries of some chunk ℓ if the chunk ℓ is encountered more than M_2 times. With Q random distinct queries, each query will hit a random chunk. The expected number of hits per chunk is therefore $Q/\sqrt{n} = \log \kappa \cdot \alpha$. By the Chernoff bound, the probability that the number of visits to some fixed chunk ℓ exceeds $M_2 = 3 \log \kappa \cdot \alpha$ is negligibly small in κ, as long as $\alpha(\kappa)$ is a super-constant function.

Finally, taking a union bound over all chunks and all polynomially many time steps, the probability of ever depleting replacement entries of any chunk is negligibly small in κ.

3.4 Our Single-Server PIR Scheme

3.4.1 Construction

Notation. For $x \in \{0, 1, \ldots, n-1\}$, we define $\mathsf{chunk}(x) := \lfloor x/n^{1/2} \rfloor$ and $\mathsf{superblock}(x) := \lfloor \mathsf{chunk}(x)/n^{1/4} \rfloor$. We assume $(\mathsf{Gen}, \mathsf{Set}, \mathsf{Program}, \mathsf{Decode})$ is a PPPS scheme over the distribution $\mathcal{D}_n$ as described in Section 3.2.

Intuition. The major differences between our single-server scheme and the two-server scheme are summarized below.

- *Preprocessing.* The two server scheme allows the client to do preprocessing with one server and do online queries with another server. Our single-server scheme uses the technique from Piano [ZPSZ24] such that the client makes a streaming pass over the whole database (retrieving from the only server) and runs the preprocessing locally.
- *Query and Refresh.* The two-server scheme allows the client to replace a consumed set with a new set on-the-fly, because the client can query another server for the new parity. Instead, in the single-server scheme, we use a new broken hint idea. The client still generates a new set after the query, but it only marks the new set as "broken hint" since the parity is unknown. To ensure correctness, the client now *uses all the matched sets* for a given query, and as long as there is one non-broken hint, the answer can be computed correctly.

Detailed algorithm for bounded, random queries. In Figure 3.4, we describe our algorithm that supports $Q = \sqrt{n}/2$ random and distinct queries. It is well known how to upgrade such an algorithm to support an *unbounded* number of *arbitrary* queries [ZLTS23]. For completeness, we briefly describe the upgrade afterward.

Efficiency. Observe that although the list decoding produces $n^{1/4}$ candidate sets each of size $\sqrt{n}$, all $n^{1/4}$ sets can actually be represented using only $O(\sqrt{n})$ space. Furthermore, computing the parities of all sets takes only $O(\sqrt{n})$ time. This is because all $n^{1/4}$ sets are derived from some common set S by replacing the offsets within each of the $n^{1/4}$ superblocks with another random vector $\delta_0, \ldots, \delta_{n^{1/4}-1}$. We give a full efficiency analysis in the proof of Theorem 3.4.1.

Supporting unbounded, arbitrary queries. We can easily get rid of the distinct query assumption in the following way: we can require the client to store a local cache of size Q to store the most recent Q queries. If the client wants a repeated query, it can lookup in the cache and make a dummy query.

Single-Server Scheme for $Q = \sqrt{n}/2$ Queries [a]

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

Preprocessing.

- Client samples $M_1 = 2\sqrt{n}\log\kappa \cdot \alpha(\kappa)$ master PPPS keys denoted $\mathsf{sk}_1, \ldots, \mathsf{sk}_{M_1} \in \{0,1\}^\lambda$. Initialize the parities $p_1, \ldots, p_{M_1}$ to zeros.
- Client downloads the whole DB from the server in a streaming way: when the client has the j-th chunk $\mathsf{DB}[j\sqrt{n} : (j+1)\sqrt{n}]$:
  - *Update the primary table*: for $i \in [M_1]$, let $p_i \leftarrow p_i \oplus \mathsf{DB}[\mathsf{Set}(\mathsf{sk}_i)[j]]$.
  - *Store replacement entries*: sample and store $M_2 = 3\log\kappa \cdot \alpha(\kappa)$ tuples of the form $(r, \mathsf{DB}[r])$ where r is a random index from the j-th chunk.
  - Delete $\mathsf{DB}[j\sqrt{n} : (j+1)\sqrt{n}]$ from the local storage.
- At this moment, let $T := \{(\mathsf{sk}_i, p_i)\}_{i\in[M_1]}$ denote the client's *hint table*. Mark all the hints as "good".

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

1. Client: For each matched entry (sk_i, p_i) such that $x \in \mathsf{Set}(\mathsf{sk}_i)$ in the hint table, do the following unless there are already $M_3 = 3\log\kappa \cdot \alpha$ matched entries:
   - For the first good (i.e., non-broken) matched entry, find the first unconsumed replacement entry $(r, \mathsf{DB}[r])$ for $\mathsf{chunk}(x)$. [b]
   - Otherwise, sample a random index r in $\mathsf{chunk}(x)$.
   - $(\mathsf{sk}', i^*) \leftarrow \mathsf{Program}(\mathsf{sk}_i, \mathsf{chunk}(x), r \bmod \sqrt{n})$.
   - Send sk' to the server, and receive $\{\beta_i\}_{i\in\{0,\ldots,n^{1/4}-1\}}$ from the server.
   - For the first good matched entry, save the answer $p_i \oplus \beta_{i^*} \oplus \mathsf{DB}[r]$.
   - Sample a fresh PPPS key sk_{new} subject to $x \in \mathsf{Set}(\mathsf{sk})$, and replace the consumed entry (sk_i, p_i) with $(\mathsf{sk}_{\text{new}}, 0)$ and mark the entry as *broken*.
2. Client: If fewer than M_3 keys are sent in the previous step, send more dummy programmed keys to the server until there are M_3 keys sent [c].
3. Client: Output the saved answer. If no answer was saved, output 0.
4. Server: For each sk' received, let $S_0, \ldots, S_{n^{1/4}-1} \leftarrow \mathsf{ListDecode}(\mathsf{sk}')$. For each $i \in \{0, \ldots, n^{1/4}-1\}$, send the xor-sum $\oplus_{j\in S_i}\mathsf{DB}[j]$ to the client[d].

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

[b] If not found, treat it as the otherwise case.

[c] The dummy key is constructed as sampling a random PPPS key sk subject to $x \in \mathsf{Set}(\mathsf{sk})$ and call $\mathsf{sk}', \_ \leftarrow \mathsf{Program}(\mathsf{sk}, \mathsf{chunk}(x), \delta')$.

[d] We use the normal representation of the set S_i and not the offset representation.

Figure 3.4: Our single-server preprocessing PIR scheme.

Furthermore, we can assume that the queries are random without loss of generality as follows: we can let the client and the server agree on a pseudorandom permutation (PRP) [RY13, HMR12] upfront and the server can permute the database according to the PRP. Another option is let the server build the database as a key-value storage and use a cuckoo hash table [PR04, Yeo23] directly based on a PRF to locate the queries. Notice that, in both implementations, the client can still make queries adaptively depending on the real query sequence and the responses, which is sufficient for practical usage. Then, as long as the client makes the queries independent of the randomness of the PRP/PRF, those queries can be considered as uniformly random. This assumption is only needed for correctness.

Lastly, we can remove the bounded Q query assumption as follows: the straightforward way is that once the client finishes a window of Q queries, the client and the server rerun the preprocessing phase again, using fresh randomness. The drawback is that the client has to wait a long time before starting the next window. As previous work pointed out [ZPSZ24, ZLTS23], we can easily avoid this drawback through a simple pipelining trick, by spreading the preprocessing work of the next Q window over the current Q window of queries.

Theorem 3.4.1. *Let $\alpha(\kappa)$ be any super-constant function. Suppose that $\mathsf{PRF}_1, \mathsf{PRF}_2$ are secure pseudorandom functions, and n is bounded by $\mathsf{poly}(\lambda)$ and $\mathsf{poly}(\kappa)$. The single-server scheme in Figure 3.4, which supports $\sqrt{n}/2$ random, distinct queries, is private, and correct with probability $1 - \mathsf{negl}(\lambda) - \mathsf{negl}(\kappa)$ for some negligible function $\mathsf{negl}(\cdot)$. Furthermore, it achieves the following performance bounds:*

- *$O_\lambda(\sqrt{n} \log \kappa \cdot \alpha)$ client storage and no additional server storage;*
- ***Preprocessing Phase**:*
  - *$O(n)$ server time and $O_\lambda(n \log \kappa \cdot \alpha)$ client time;*
  - *$O(n)$ communication;*
- ***Query Phase**:*
  - *$O_\lambda(\sqrt{n} \log \kappa \cdot \alpha)$ expected client time and $O_\lambda(\sqrt{n} \log \kappa \cdot \alpha)$ server time per query;*
  - *$O_\lambda(n^{1/4} \log \kappa \cdot \alpha)$ communication per query.*

Therefore, the amortized online communication per query is $O_\lambda(n^{1/4} \log \kappa \cdot \alpha)$, the amortized offline communication per query is $O(\sqrt{n})$, and the amortized client and server computation per query is $O_\lambda(\sqrt{n} \log \kappa \cdot \alpha)$.

Proof. We defer the privacy and the correctness proofs to Section 3.4.2 and Section 3.4.3 respectively. The client only stores $M_1 = 2\sqrt{n} \log \kappa \alpha$ λ-bit keys and stores in total $\sqrt{n} \cdot M_2 = 3\sqrt{n} \log \kappa \cdot \alpha$ index-value pairs. So the storage is $O_\lambda(\sqrt{n} \log \kappa \cdot \alpha)$. The preprocessing phase's performance bounds follow straightforwardly by the algorithm descriptions.

For the query phase, the client first enumerates all M_1 hints to find x, which takes $O_\lambda(\sqrt{n} \log \kappa \cdot \alpha)$ time. For all the $M_3 = \Theta(\log \kappa \cdot \alpha)$ found hints, the $\mathsf{Program}$ algorithm takes $O_\lambda(n^{1/4})$ client computation. For the server, during the query phase, the client sends M_3 programmed PPPS keys to the server, the size of which is $O_\lambda(n^{1/4})$. The server sends back the xor-sum of $n^{1/4}$ candidate sets for each key after running $\mathsf{ListDecode}$. Even though each candidate set has size $n^{1/2}$, all the candidate sets have a succinct representation of size $n^{1/2}$ and server can compute this representation in time $O_\lambda(n^{1/2})$. Furthermore, as observed in Theorem 3.3.1, due to the structure of the candidate sets, the server can compute all the $n^{1/4}$ xor-sums in time only $O(n^{1/2})$. Hence, the server's running time $O_\lambda(\sqrt{n})$.

The client needs to find M_3 matched hints, compute $O(\log \kappa \cdot \alpha)$ xor operations during each query and for the matched hints, it needs to sample fresh PPPS key $\mathsf{sk}_{\mathsf{new}}$ subject to $x \in \mathsf{Set}(\mathsf{sk}_{\mathsf{new}})$. The client's computation time is dominated by this sampling step. We consider the expected computation time: each key in the hint table will have $1/\sqrt{n}$ probability to be replaced in this,and each sampling takes $O_\lambda(\sqrt{n})$ expected time to finish using Lemma 3.4.2 and pseudorandomness of PPPS, the expected number of keys checked until a key sk such that $\mathsf{Set}(\mathsf{sk})$ contains the current query is found is $O(\sqrt{n})$. So the total expected time for the query phase is $O_\lambda(\sqrt{n} \log \kappa\alpha)$ per query. The server time is $O_\lambda(\sqrt{n}(\log \kappa \cdot \alpha))$ per query. The online communication per query is $O_\lambda(n^{1/4}(\log \kappa \cdot \alpha))$. ∎

3.4.2 Privacy Proof

Suppose that the underlying PPPS scheme satisfies private programmability. Below, we prove the privacy of our single-server PIR scheme.

In the preprocessing phase, the server observes a single scan over the database, and thus no information is leaked. The rest of the proof will therefore focus on the query phase.

Lemma 3.4.2. *Recall that in each time step t, the adversary $\mathcal{A}$ adaptively chooses a query $x_t \in \{0, 1, \ldots, n-1\}$ for the client.*

At the end of each time step t, the client's hint table is distributed as a table of size M_1 where each entry is a freshly sampled PPPS key, even when conditioned on $\mathcal{A}$'s view so far.

Proof. Suppose the above statement holds at the end of time step $t-1$. We prove that it still holds at the end of time step t. Since the hint table is distributed as a fresh randomly sampled table even when conditioned on $\mathcal{A}$'s view at the end of $t-1$, we may henceforth assume an arbitrary fixed query x_t. The distribution of the hint table before the t-th query can be equivalently rewritten as:

- First, sample the indices of the entries (henceforth denoted I) that contain the query x_t. Specifically, each $i \in [M_1]$ is chosen into the set I independently with probability $1/\sqrt{n}$.
- For each $i \notin I$, sample a random PPPS key subject to not containing x_t.
- For each $i \in I$, sample a random PPPS key subject to containing x_t.

Using the above interpretation, it is easy to see that the distribution of the hint table after the t-th query is unaltered. ∎

Simulator construction and the Ideal **experiment.** Consider the following simulator construction which does not make use of the queries: in every time step t, call the simulator Sim of the PPPS scheme, and let the output be sk'. Send sk' to the server.

Indistinguishability of Real **and** Ideal**.** We now prove the indistinguishability of the Real and Ideal assuming the private programmability of the underlying PPPS scheme.

First, due to Theorem 3.4.2, we can equivalently rewrite the Real experiment as follows: at the end of each time step, resample the entire hint table freshly at random before continuing to answer more queries. As a result, the messages sent to $\mathcal{A}$ in each time step t are distributed as

Repeat M_3 times:

- sample a PPPS key sk at random subject to containing the query x_t; sample δ at random from $\{0, \ldots, \sqrt{n}-1\}$.

- call $\mathsf{sk}', \_ \leftarrow \mathsf{Program}(\mathsf{sk}, \mathsf{chunk}(x_t), \delta)$;
- send sk' to $\mathcal{A}$.

One way to see this is to think of the distribution of the table as having the equivalent distribution in the proof of Theorem 3.4.2.

Therefore, the rest of the proof follows from a straightforward hybrid argument where we replace the programmed keys (denoted sk' earlier) sent to the server in all time steps one by one with a simulated key, relying on the private programmability of the underlying PPPS.

3.4.3 Correctness Proof

For the correctness analysis, we may assume that every set is sampled independently from $\mathcal{D}_n$. Due to the pseudorandomness property of the PPPS, this will only affect the correctness probability by a negligible amount.

Recall that we have a window of $Q = \sqrt{n}/2$ random, distinct queries. There are only two bad events that can cause correctness failure: 1) the client cannot find a good hint that contains the query index; 2) the client runs out of replacements in a chunk.

We first analyze the second bad event, i.e., depleting replacement entries. For every query, at most one replacement entry is consumed. Therefore, the second bad event only happens when the client makes more than M_2 queries in one chunk. The analysis is the same as the analysis of depleting replacement entries in the proof of correctness for our 2-server scheme (see Section 3.3.3).

The first bad event, i.e., no matched good hint, can only arise from the following events: 1) there are no good hints left that match the query; 2) there are more than M_3 matched hints but the first M_3 matched hints are all broken.

Fix a sequence of queries $x_1, \ldots, x_m$ and consider the error probability for x_m. Consider the initial hint table in which each entry represents a random set sampled from $\mathcal{D}_n$.

If a hint in the initial hint table contains x_m and does not contain $x_1, \ldots, x_{m-1}$, this hint will remain good until the query for x_m. We have that, for any hint,

$$\Pr[\text{ this hint contains } x_m] = 1/\sqrt{n} \ .$$

Furthermore, conditioned on a hint containing x_m, if x_i and x_m are not in the same chunk, then the hint contains x_i with probability $1/\sqrt{n}$ because each chunk has its own randomness. Moreover, if x_i and x_m are in the same chunk, this hint definitely will not contain x_i. Hence, we have that

$$\begin{aligned}
&\Pr\left[\text{ this hint is broken} \,\middle|\, \text{ this hint contains } x_m\right] \\
\leq &\Pr\left[\exists i \in \{1, \ldots, m-1\} \text{ this hint contains } x_i \,\middle|\, \text{ this hint contains } x_m\right] \\
\leq &(m-1)/\sqrt{n} \leq Q/\sqrt{n} \leq 1/2 \ .
\end{aligned}$$

Therefore,

$$\begin{aligned}
&\Pr[\text{ this hint is good and contains } x_m] \\
\geq &\Pr\left[\text{ this hint is good} \,\middle|\, \text{ this hint contains } x_m\right] \cdot \Pr[\text{ this hint contains } x_m] \geq 1/(2\sqrt{n}) \ .
\end{aligned}$$

Then, the probability that there is no good hint matching x_m in the table is at most

$$\left(1 - \frac{1}{2\sqrt{n}}\right)^{M_1} = \left(1 - \frac{1}{2\sqrt{n}}\right)^{2\sqrt{n}\ln\kappa\cdot\alpha(\kappa)}$$
$$\leq (1/e)^{\ln\kappa\alpha(\kappa)} = \kappa^{-\alpha(\kappa)}.$$

Now let us argue that for query x_m, the probability of more than M_3 hints being matched is small. Due to the Lemma 3.4.2, we may assume that at the beginning of each query, the hint table contains freshly and independently chosen sets. Each set sampled from $\mathcal{D}_n$ contains the query with probability $1/\sqrt{n}$. The expected number of hints that match the query is therefore $M_1/\sqrt{n} = 2\log\kappa\cdot\alpha$. Using the Chernoff bound, we have that the probability of more than $M_3 = 3\log\kappa\cdot\alpha$ hints being matched is bounded by a negligibly small function in κ.

Finally, we can apply a union bound over all $\sqrt{n}/2$ queries and conclude that the probability of the first bad event (i.e., no matched good hint) ever happening is negligibly small in κ.

3.5 Our Optimized Single-Server Scheme with Non-Black-Box Use of PPPS

Since our earlier single-server scheme in Section 3.4 employs the broken hints technique, there is a superlogarithmic multiplicative overhead due to the repetition needed for correctness compared to the two-server scheme in Section 3.3. In this section, we describe an improved single-server preprocessing PIR scheme that avoids this super-logarithmic repetition overhead. For this optimized version, we cannot use the PPPS scheme in a completely blackbox manner.

3.5.1 Construction

Why previous schemes do not worry about broken hints. First, we need to understand why in the previous single-server schemes like PIANO [ZPSZ24], the client does not have to worry about broken hints. Notice that in the two-server scheme, the client will replace the consumed hint with a resampled hint that contains the current query, and interact with another server to fetch the correct parity for this new hint. This ensures that all hints are useful. In the single-server setting, the client cannot rely on the additional server to dynamically fetch the parity for the resampled hint. To fix this, PIANO requires the client to prepare polylogarithmic *backup* hints for each of the $\sqrt{n}$ chunks. The backup hints for the i-th chunk record the parity of the set and also the value of the item in the i-th chunk. After querying for x, the client will replace the consumed hint with a backup hint from x's chunk and lazily mark that i-th element of the hint to be the current query x, which maintains the distribution of the hint table. Since the values for the original i-th element and x are known to the client, the correct parity for this edited hint can be derived by the client locally, avoiding the issue of broken hints. Then, the client only needs to find the first hint that contains x, instead of finding all matched hints and making multiple queries based on those hints,

hoping that one of them will be good. [1] Now, after the client expands the key to a whole set, the client can easily enforce the marking (if necessary) by just changing the corresponding element.

We will try to follow the backup hint and the lazy-marking idea. There is one challenge remaining: in PIANO, the client can expand the whole set and *possibly change two elements obliviously* before sending the set to the server. The client not only needs to change the offset in the current query's chunk to ensure the privacy, it also needs to change the offset in the lazy marked element's chunk to reflect the correct history. Our main construction in Section 3.2 only allows us to program one location. How should we modify the scheme to support programming at two points?

One way to achieve that is by using a privately programmable pseudorandom function (PPPRF). A PPPRF allows us to directly program on the key. This is nearly the same idea in two previous single-server PIR schemes [ZLTS23, LP23a]. Unfortunately, this primitive is only known under strong cryptographic assumptions like LWE, and still only exists in the theoretical literature – it is completely impractical at the current stage.

Our solution: rejection sampling. We observe that all we need to do is just to "maintain" the right distribution of the hint table and the keys sent to the server. If we can monitor the change in the distribution, we can rejection-sample the PRF keys according to the right distribution and avoid the difficulty of doing private programming on the key level.

Maintaining the distribution for the local hint table is relatively simple and is already achieved by the lazy-marking technique. For the local hint table, what we actually care about is the distribution of the actual sets, regardless of how we represent them. Assume that the client already queries for y and knows $\mathsf{DB}[y]$. Now the client consumes a hint that contains y, and replaces it with a hint from those backup hints prepared for $\mathsf{chunk}(y)$. Suppose the key for the backup hint is sk. $\mathsf{Set}(\mathsf{sk})$ may not contain y, but we can mark y alongside the key, such that whenever we need to do membership testing on this key, we will consider that y is already programmed into the set. This maintains the local table's distribution.

The trickier case is how we handle the keys that the client sends to the server. Suppose the client is now querying for x, and it finds the first hint that contains x. Also, it notices that the hint is marked with y. Recall that in the PPPS programming algorithm, the client will expand the master key into $n^{1/4}$ keys and expand the keys corresponding to the superblock of x to $n^{1/4}$ offsets. Now there are two cases:

1. *x and y are in the same superblock.* This is a relatively simple case. As the normal programming step in the PPPS, we will replace the sub-key corresponding to the superblock of x (the same as y) with a uniformly random sub-key. However, when we expand the original sub-key to $n^{1/4}$ offsets, we need to replace x's offset with a random one, and also manually set the offset corresponding to y's chunk to y's offset, reflecting that we enforce y to be included. No rejection sampling is required in this case.

   For the answer, since we are just replacing the original element in y's chunk to y, the influence of such change should be corrected. We can require the client to store the database

[1] A natural question is why the client cannot simply pick the first good hint. Observe that this would be equivalent to deleting a consumed hint from the table because a broken hint will never be used again. Then, it would skew the distribution of the hint table because the remaining hints are less likely to contain the past queries, which leads to privacy leakage.

Figure 3.5: The augmented query protocol based on a single hint. The augmentation is mainly about returning the parities of those superblocks corresponding to the $n^{1/4}$ sub-keys and will be used in Figure 3.6.

Augmented Query Protocol Based on a Single Hint[a]

Client's input:

- A master PPPS key sk assuming $x \in \mathsf{Set}(\mathsf{sk})$;
- The parity of $\mathsf{Set}(\mathsf{sk})$ is assumed to be p_{sk};
- A random index r in x's chunk such that the $\mathsf{DB}[r]$ is assumed to be known.

1. **Step 1: (Client)**
   (a) Let $(\mathsf{sk}', i) \leftarrow \mathsf{Program}(\mathsf{sk}, \mathsf{chunk}(x), r \bmod \sqrt{n})$ and send sk' to the server;
2. **Query Step 2: (Server)** Upon receiving sk':
   (a) Parse sk' as
   $$(k_0, \ldots, k_{n^{1/4}-1}), (\delta_0, \ldots, \delta_{n^{1/4}-1}).$$
   (b) Let $S_0, \ldots, S_{n^{1/4}-1} \leftarrow \mathsf{ListDecode}(\mathsf{sk}')$;
   (c) For $i \in \{0, \ldots, n^{1/4} - 1\}$:
      i. Let $\beta_i = \oplus_{x \in S_i} \mathsf{DB}[x]$;
      ii. Expand k_i to $n^{1/4}$ offsets and consider them being the corresponding index offsets in chunk $\{i \cdot n^{1/4}, \ldots, (i+1) \cdot n^{1/4} - 1\}$. Compute the xor-sum of the database values for those indices as
      $$\alpha_i = \oplus_{j \in \{0, \ldots, n^{1/4}-1\}} \mathsf{DB}[(j + in^{1/4})\sqrt{n} + \mathsf{PRF}_2(k_i, j)].$$
   (d) Return $(\alpha_0, \ldots, \alpha_{n^{1/4}-1})$, $(\beta_0, \ldots, \beta_{n^{1/4}-1})$ to the client.
3. **Step 3: (Client)**
   Upon receiving $(\alpha_0, \ldots, \alpha_{n^{1/4}-1})$, $(\beta_0, \ldots, \beta_{n^{1/4}-1})$:
   - Save $(\alpha_0, \ldots, \alpha_{n^{1/4}} - 1)$ if the answer needs to be corrected later.
   - Return $p_{\mathsf{sk}} \oplus \beta_i \oplus \mathsf{DB}[r]$ as the answer.

[a] The augmented part is highlighted in red.

Figure 3.6: The optimized sublinear single server preprocessing PIR protocol introduced in Section 3.5. The protocol uses a subroutine defined in Fig. 3.5 as a subroutine.

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

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

Preprocessing.

- In addition to the previous algorithm:
  - Client samples $M_2 = 3\log\kappa\alpha(\kappa)$ backup keys $\mathsf{sk}_{i,1}, \ldots, \mathsf{sk}_{i,M_2}$ for chunk i;
  - For each backup hint, Client stores $\mathsf{sk}_{i,j}$ and the following information:
    - The parity for the whole set: $p_{i,j} = \oplus_{x\in\mathsf{Set}(\mathsf{sk}_{i,j})}\mathsf{DB}[x]$;
    - The parity for those items within the superblock that contains the i-th chunk: $b_{i,j} = \oplus_{x\in\mathsf{Set}(\mathsf{sk}_{i,j})}\mathbf{1}\left[\mathsf{superblock}(x) = \left\lfloor\frac{i}{n^{1/4}}\right\rfloor\right]\mathsf{DB}[x]$ [b]
    - The DB-value of the i-th item: $\mathsf{DB}[\mathsf{Set}(\mathsf{sk}_{i,j})[i]]$.

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

1. **Query:**
   (a) Client finds the first matched hint (sk_i, p_i) such that $x \in \mathsf{Set}(\mathsf{sk}_i)$ and the hint does not have a positive constraint y that $\mathsf{chunk}(y) = \mathsf{chunk}(x)$.
   (b) If there is no positive constraint on this hint: proceed the subroutine Figure 3.5 as usual.
   (c) If there is a positive constraint $+y$ on this hint:
      i. If $\mathsf{superblock}(y) = \mathsf{superblock}(x)$, execute the subroutine, except that
         - After the client programs the key and gets $n^{1/4}$ sub-keys and offsets, replaces the $(\mathsf{chunk}(y) \mod n^{1/4})$-th offset with y's offset;
         - Let the return value be v, mark the answer as $v \oplus \mathsf{DB}[y] \oplus \mathsf{DB}[\mathsf{Set}(\mathsf{sk}_i)[\mathsf{chunk}(y)]]$.
      ii. If $\mathsf{superblock}(y) \neq \mathsf{superblock}(x)$, execute the subroutine, except that:
         - After the client has the programmed key of $n^{1/4}$ sub-keys and offsets, replace the $\mathsf{superblock}(y)$-th key with a rejection-sampled key k'. The rejection sampling key k' should satisfy all constraints related to the matched hint $(+y, -z_1, \ldots, -z_t)$ if k' controls $\mathsf{superblock}(y)$ (as specified in Section 3.5).
         - Let the return value be v, mark the answer as $v \oplus \alpha_{\mathsf{superblock}(y)} \oplus b$, where $\alpha_{\mathsf{superblock}(y)}$ is the corresponding parity of superblock controlled by the rejection-sampled sub-key and b is the original parity for that superblock, known by preprocessing.
2. **Refresh:**
   - Client adds the constraint $-x$ for the first $i-1$ entries.
   - Client replaces the matched hint with an unconsumed hint from the $\mathsf{chunk}(x)$'s backup hint group. Clean all other constraints and only mark $+x$ for the i-th entry.

[a] For clarity, we present the scheme supporting distinct and random queries.
[b] $\mathbf{1}[A]$ is 1 when A is true and 0 otherwise.

value for the i-th element for all the backup hints dedicated for the i-th chunk. Then, if this promoted hint sk is used, we can correct the answer by xoring the influence value, $\mathsf{DB}[\mathsf{Set}(\mathsf{sk})[\mathsf{chunk}(y)]] \oplus \mathsf{DB}[y]$.

2. *x and y are in the different superblocks.* This is the more involved case. In the programming, we will again replace the sub-keys corresponding to x's superblock with a uniformly random sub-key and replace the offsets in x's chunk with a random one. However, we need to ensure that the key corresponding to y's superblock will expand to the correct offset for y's chunk, to ensure that y is included. Now, we will rejection-sample a sub-key k for y's superblock, such that it expands to the correct offset of y. This is what we refer to as a **positive** constraint, and we write this constraint as $+y$. Additionally, to maintain the right distribution, there are also **negative** sampling constraints. Observe that if the i-th hint entry is the first one to contains x, it means that from the client's perspective, the first $i-1$ entries do not contain x. Suppose between the query y and query x, there are some other queries. After making those queries, the client also knows that the current hint entry does not contain some queries $z_1, \ldots, z_t$. We write down the negative constraints as $-z_1, \ldots, -z_t$. Then, when we do the rejection sampling for the sub-key for y's superblock, we will consider all the constraints: $+y, -z_1, \ldots, -z_t$, until we find a sub-key k that satisfies all the constraints. This ensures that the resampled sub-key for y's superblock has the right distribution.

   We also need to argue that the rejection-sampling is efficient. The positive constraint is satisfied with probability $1/\sqrt{n}$. Conditioned on the positive constraint being satisfied, for each negative constraint, it is satisfied with probability at least $1 - 1/\sqrt{n}$. We only need to focus on the negative constraints related to the $n^{1/4}$ chunks in that particular superblock. Due to a simple balls-into-bins argument, the maximum queries in each chunk is bounded by $3 \log \kappa \cdot (\alpha(\kappa))$ with high probability, and the same bound holds for the maximum number of negative constraints in each chunk. Therefore, the rejection sampling can satisfy all the negative constraints in those $n^{1/4}$ chunks with probability at least $1 - n^{1/4} \cdot \frac{3 \log \kappa \cdot (\alpha(\kappa))}{\sqrt{n}}$, which is at least $1/2$ given a sufficiently large n. Therefore, we conclude that a freshly sampled sub-key satisfies all the constraints with probability at least $1/2\sqrt{n}$, and the expected sample time is $O(\sqrt{n})$.

   A notable detail is that when we draw a new fresh key, we first check the positive constraint. Only when the positive constraint is satisfied, we check the negative constraints. The positive constraint is satisfied with probability $1/\sqrt{n}$ but only takes $O(1)$ time to check. Conditioned on the positive constraint being satisfied, the negative constraints take $O(\sqrt{n})$ to check, but they are satisfied with probability at least $1/2$. So even including the constraint checking time, the expected sampling time is still $O(\sqrt{n})$.

   For the answer, since we are essentially replacing the original superblock with a rejection-sampled one. To remove the influence of this step, we can require the client to store the parity for the corresponding superblock controlling the i-th chunk for all the backup hints dedicated for the i-th chunk. Then, if this promoted hint sk is used, we can correct the answer by xor-ing the parity for the original superblock (known by preprocessing) and the parity for the rejection-sampled superblock. This is why we need an augmented version of the response protocol for the server Figure 3.5 – we need the server to tell the client about the parity of the rejection-sampled superblock.

We provide the pseudocode of this construction in Figure 3.6.

Theorem 3.5.1. *Let $\alpha(\kappa)$ be any super-constant function. Assume n is bounded by $\mathsf{poly}(\lambda)$ and $\mathsf{poly}(\kappa)$. The PIR scheme in Figure 3.6 that supports $\sqrt{n}\log\kappa\alpha(\kappa)$ queries is private, correct with probability $1-\mathsf{negl}(\lambda)-\mathsf{negl}(\kappa)$ and achieves the following performance bounds (the optimized parts are underlined):*

- *$O_\lambda(\sqrt{n}\log\kappa\alpha(\kappa))$ client storage and no additional server storage;*
- ***Preprocessing Phase**:*
  - *$O_\lambda(n\log\kappa\alpha(\kappa))$ client time and $O(n)$ server time;*
  - *$O(n)$ communication;*
- ***Query Phase**:*
  - *$\underline{O_\lambda(\sqrt{n})\text{ expected client time}}$ and $\underline{O_\lambda(\sqrt{n})\text{ server time}}$ per query;*
  - *$\underline{O_\lambda(n^{1/4})\text{ communication}}$ per query.*

Therefore, the amortized online communication per query is $O_\lambda(n^{1/4})$, the amortized offline communication per query is $O(\sqrt{n})$, the amortized client computation is $O_\lambda(\sqrt{n}\log\kappa\alpha)$, and the server computation is $O_\lambda(\sqrt{n})$.

Proof. The correctness proof will be similar to the correctness proof for the main scheme described in Section 3.3.3.

We defer the privacy proof to later.

The additional storage for the client will be those M_2 backup hints per chunk, which results in total of $O(\lambda\sqrt{n}\log\kappa\alpha(\kappa))$ space. Notice that the client does not directly store the negative constraints, otherwise the storage will be blown up. The client can directly store the history of those matched hint entry index (which takes $O(\sqrt{n}\log n)$ space) and also the generation time label of each hint. Whenever the client finds the first matched hint, say entry i, it finds all the previous queries that are older than the current hint's generation time and whose matched hint entry indices are larger than i. Those history queries will be the negative constraints. By doing this, maintaining all the constraints only takes $O(\sqrt{n}\log n)$ space.

The preprocessing phase's performance bounds are obvious by the algorithm description.

For the query phase, the client first finds the first hint that contains x. Each hint contain x with probability $1/\sqrt{n}$, so the expected time is $O(\lambda\sqrt{n})$. The subroutine in Figure 3.5 takes $O(\lambda n^{1/4})$ client computation and $O(\lambda\sqrt{n})$ server computation, while consuming $O(\lambda n^{1/4})$ communication cost. The refresh phase takes only $O(1)$ time (if the client maintains the negative constraint using the technique mentioned above). So the expected client time is $O(\lambda\sqrt{n})$ per query. The server time is $O(\lambda\sqrt{n})$ per query. The online communication per query is $O(\lambda n^{1/4})$ per query. ∎

3.5.2 Privacy Proof

Proof. We will prove this theorem via a sequence of hybrids. The first hybrid in the sequence will capture the real experiment in the privacy definition of single server PIR, and the last hybrid will capture the ideal experiment. We first define the hybrid Real^*:

- *Preprocessing phase.* $\mathcal{A}$ receives the streaming signal. The client samples $\mathsf{sk}_1,\ldots,\mathsf{sk}_{M_1}\xleftarrow{\$}\{0,1\}^\lambda$.
- *Query phase.* For each round t, $\mathcal{A}$ chooses a query x_t:

- The client finds the first matched key sk_i in the hint table where $x \in \mathsf{Set}(\mathsf{sk}_i)$ and if there's a positive constraint $+y$ on this entry, x is not in the same chunk as y.
- If the entry i has no positive constraint: execute the subroutine.
- If the entry i has a positive constraint $+y$:
  - If y and x are in the same superblock: execute the subroutine, with the difference that when the client is expanding the sub-key corresponding to the superblock of x to $n^{1/4}$ offsets, replace the offset of y's location with y's offset.
  - If y and x are not in the same superblock: execute the subroutine with the difference that when the client is expanding the master key to $n^{1/4}$ sub-keys, replace the sub-key corresponding with y's superblock to a rejection sampled key k', such that when k' is considered as the key for that superblock, the expanded set satisfies all constraints related to entry i;
- Then, the client replaces the entry sk_i with a freshly-sampled key sk', but deletes other constraints and marks constraint $+x$ for entry i;
- The client also marks constraints $-x$ for entry 1 to entry $i-1$.

This hybrid is identically distributed as the actual experiment where we just remove all parts unrelated to the privacy proof.

We define Hyb_1 by removing PRF_1 from Real^*: for each sampled master key sk, the client will directly sample $n^{1/4}$ random sub-keys and store them in the local storage. When the client executes the programming step in the subroutine, it no longer has to expand the master key to $n^{1/4}$ sub-keys.

Hyb_1 is computationally indistinguishable from Real^* due to a straightforward reduction to the pseudorandomness of PRF_1.

Since we already replaced all master keys with $n^{1/4}$ sub-keys, the later hybrids no longer need to expand a master key to $n^{1/4}$ sub-keys in the subroutine.

We define Hyb_2 as follows: Hyb_2 is the same as Hyb_1, except that whenever the client finds the matched hint for query x, it will resample all those $n^{1/4}$ sub-keys **according to all the marked negative constraints and also the new constraint $+x$.** Notice that the history-dependent positive constraint (usually written as $+y$) is not considered in this new resample step and only enforced just before the client sending messages to the server.

Now we argue that Hyb_2 is identically distributed as Hyb_1.

Claim 3.5.2. *The view of the adversary in* Hyb_2 *is identically distributed as in* Hyb_1.

Proof. For all Q queries, let's consider the matched hint index vector $\mathbf{I} = (i_1, \ldots, i_Q)$ where i_t denotes the client finds i_t as the matched hint in the t-th round. Notice that the recorded constraints at the t-th round are completely determined by $i_1, \ldots, i_{t-1}$.

Now we add the matched index vector to the view of the adversarys, and we will argue that even with the augmented view, the views of the adversary are still identically distributed in the two experiments.

First, observe that the two experiments have exactly the same way to generate the matched index vector. We only need to argue that, the selected entries in each round have the same distribution in both experiments, and then the messages observed by the adversary should have the same distribution.

We claim that conditioned on the same $i_1, \ldots, i_{t-1}$, the selected entry in round t have the same distributions in the two experiments.

In Hyb_1, we can equivalently consider the client first uniformly sampling the M_1 primary hints and extra $t-1$ replaced hints. Then, given the queries $x_1, \ldots, x_{t-1}$, the client finds the matched hint and hence derives $i_1, \ldots, i_{t-1}$ and those selected entry.

In Hyb_2, we can equivalently consider the client first observing the queries $x_1, \ldots, x_{t-1}$ and sampling $i_1, \ldots, i_{t-1}$ from the marginal distribution (conditioned on $x_1, \ldots, x_{t-1}$ and all the initial M_1 hints and the $t-1$ replacement hints are random). Then, the client samples the selected entry for each round by deriving the constraints from $i_1, \ldots, i_{t-1}$ and does rejection sampling.

A key observation is that in Hyb_1, the adversary cannot observe what the hint table is before round t. Then, from the adversary's perspective, the selected entry's distribution is the *posterior* distribution of them after observing $i_1, \ldots, i_{t-1}$. Moreover, that *posterior* distribution is exactly the same as the distribution the client uses to generate the selected entry (and even the whole table) in round t (i.e., containing the query x_t and some negative constraints $-z_1, -z_2, \ldots$) in Hyb_2. Therefore, from the adversary's perspective, the selected hints in the two experiments share the same distribution, and thus the views in the two experiments are identically distributed.

■

We define Hyb_3 as follows: Hyb_3 is the same as Hyb_2, except that whenever the client finds the matched hint for query x, it will resample that entry also considering the history-dependent positive constraint. Hyb_2 does not consider the history-dependent positive constraint because the history-dependent constraint is only marked locally and is only enforced before sending the message. That is, it will include the positive constraints of including the current query x and a history query y (if necessary) and all other negative constraints. We will prove that Hyb_3 is computationally indistinguishable from Hyb_2.

Claim 3.5.3. Hyb_3 *is computationally indistinguishable from* Hyb_2.

Proof. For each query x_t, there are three cases for the selected hint entry: 1) it does not have a history-dependent positive constraint; 2) it has a history-dependent positive constraint $+y$, but y and x are not in the same superblock; 3) it has a history-dependent positive constraint $+y$ such that y and x are in the same superblock.

The first case is trivially identical in the two experiments. For the second case, since the client will always resample the sub-key for y's superblock before sending messages to the server, it does not matter whether the original hint contains y or not (which only affects the sub-key for y's superblock). The final case is that the current query x and the positive constraint y are in the same superblock. We can view Hyb_2 and Hyb_3 as the case where we sample the sub-key in that particular superblock according to the same constraints, except that Hyb_3 will have one additional constraint that the offset in y's chunk should be y's offset. Then the client will expand the sampled sub-key to $n^{1/4}$ offsets and change the offset in y's chunk to y's offset. Due to the pseudorandomness of PRF_2, the two distributions are computationally indistinguishable.

■

Now, define Hyb_4 the same as Hyb_3, except that in the query phase, after the client resamples the selected hint entry, the client will not do the processing steps related to the history-dependent positive constraint.

For clarity, we write the full definition of Hyb_4:

- *Preprocessing phase.* $\mathcal{A}$ receives the streaming signal. The client samples $\mathsf{sk}_1, \ldots, \mathsf{sk}_{M_1} \xleftarrow{\$} \{0,1\}^\lambda$.
- *Query phase.* For each round t, $\mathcal{A}$ chooses a query x_t:
  - The client finds the first matched key sk_i in the hint table, that is, $x \in \mathsf{Set}(\mathsf{sk}_i)$ and if there's a positive constraint $+y$ on this entry, x is not in the same chunk as y.
  - Resample a key sk subject to all the constraints recorded for this entry (e.g. including the current query x and the history query y, while satisfying some negative constraints $-z_1, \ldots$).
  - Execute the subroutine based on this resampled key.
  - Then, the client replaces the entry sk_i with a freshly-sampled key sk', but deletes other constraints and marks constraint $+x$ for entry i;
  - The client also marks constraints $-x$ for entry 1 to entry $i-1$.

Hyb_4 is identically distributed as Hyb_3. There are only two different cases in Hyb_4 and Hyb_3:

- The history-dependent positive constraint y is in the same superblock as the current query x. Notice that the client already resamples the sub-key subject to y is included, so the offset in $\mathsf{chunk}(y)$ is already y's offset;
- The history-dependent positive constraint y is not in the same superblock as the current query x. In Hyb_3, we already have another step to resample the sub-key corresponding to y's superblock. However, notice that the two resampling steps are independent and subject to exactly the same constraints, so removing the latter one preserves the distribution.

Now we define Hyb_5 as follows:

- *Preprocessing phase.* $\mathcal{A}$ receives the streaming signal. The client samples $\mathsf{sk}_1, \ldots, \mathsf{sk}_{M_1} \xleftarrow{\$} \{0,1\}^\lambda$.
- *Query phase.* For each round t, $\mathcal{A}$ chooses a query x_t:
  - The client finds the first matched key sk_i in the hint table, that is, $x \in \mathsf{Set}(\mathsf{sk}_i)$.
  - Execute the subroutine with this matched key.
  - Then, the client replaces the entry sk_i with a freshly-sampled key sk' subject to $x \in \mathsf{Set}(\mathsf{sk}')$.

Notice that in Hyb_5, the client does not record any constraint and does not resample the matched hint.

Claim 3.5.4. *The adversary's views in* Hyb_4 *and* Hyb_5 *are computatonally indistinguishable.*

Proof. This argument is similar to the argument for Claim 3.5.2. We can still consider adding the matched index vector $\mathbf{I}$ to the adversary's views and prove the augmented views are indistinguishable.

We first argue that the distributions of $\mathbf{I}$ are computationally indistinguishable in the two experiments. Observe that at given any time, if we expand all the local hints to sets, the distributions of those sets are computationally indistinguishable. The only difference between the two experiments is that in Hyb_4, the client replaces the matched hint with a uniformly random new hint and lazily marks the current query x to the entry in Hyb_4, while in Hyb_5, the client directly does rejection-sampling to sample a new hint containing x. These two sets are computationally indistinguishable due to the pseudorandomness of the PRF_2.

Conditioned on the same matched indices $i_1, \ldots, i_t$, we only need to argue that the selected entry in round t has the same distribution in both experiments, and then the messages observed by the adversary should have the same distribution.

In Hyb_4, we can equivalently consider the client first observes the queries $x_1, \ldots, x_{t-1}$ and samples $i_1, \ldots, i_t$ from the marginal distribution (conditioned on $x_1, \ldots, x_t$, those initial M_1 hints are uniformly random, and the extra $t-1$ replacement hints are sampled conditioned on containing the corresponding queries). Then, the client samples the selected entry for this round by deriving the constraints from $i_1, \ldots, i_t$ and does rejection sampling.

In Hyb_5, we can equivalently consider the client first generates M_1 primary hints and also the $t-1$ replacement hints given the queries $x_1, \ldots, x_{t-1}$. Then, the client finds the matched hint in each round and hence derives $i_1, \ldots, i_t$ and chooses those selected entries.

Again, the key observation is that in Hyb_5, the adversary cannot observe what the hint table is before round t. Then, from the adversary's perspective, the whole table's distribution is the *posterior* distribution after observing $i_1, \ldots, i_t$. Moreover, that *posterior* distribution is exactly the same as the distribution the client recorded as the form of constraints.

Therefore, from the adversary's perspective, the selected entries in the two experiments share the same distribution conditioned on the same matched indices $i_1, \ldots, i_t$. So we can conclude that the adversary's views in the two experiments are computationally indistinguishable. ∎

Finally, we define the hybrid Ideal:

- *Preprocessing phase.* $\mathcal{A}$ receives the streaming signal.
- *Query phase.* For each round t, $\mathcal{A}$ chooses a query x_t:
  - The client will:
    - Independently sample $k_0, \ldots, k_{n^{1/4}-1} \xleftarrow{\$} \{0,1\}^\lambda$;
    - Independently sample $r_0, \ldots, r_{n^{1/4}-1} \xleftarrow{\$} \{0, \ldots, \sqrt{n}-1\}$.
    - Send $(k_0, \ldots, k_{n^{1/4}-1}), (r_0, \ldots, r_{n^{1/4}-1})$ to the server.

The argument that Ideal and Hyb_5 are computationally indistinguishable is nearly the same as the proof in Section 3.4.2. ∎

3.6 Evaluation

We implement our optimized single-server PIR scheme (Section 3.5) and compare it against a state-of-the-art preprocessing single server PIR scheme (Piano [ZPSZ24]). We show that the performance of our scheme is reasonably efficient and practical, while having a huge advantage in the online communication.

Implementation and Parameters. We implement our scheme with Go, based on the open-sourced code base of Piano [ZPSZ24]. Our code base is open-sourced[2]. We set $Q = \sqrt{n} \ln n$, set the correctness failure parameter κ to 40 and set the computational security parameter λ to 128. We adjust the chunk size and also superblock size by constant factors to optimize the overall performance. The parameter combination ensures the failure probability is bounded by $2^{-\kappa} = 2^{-40}$ for all queries. We use 128-bit keys and use AES to instantiate the PRF.

[2] `https://github.com/wuwuz/QuarterPIR`

Evaluation Setup. We evaluate our scheme and the baseline scheme on a single AWS m5.8xlarge instance with 128GB of RAM and run the experiments on a local network. In this case, the network will not be the bottleneck. However, we do expect that our scheme can perform relatively better compared to Piano in a network-constrained environment.

3.6.1 Experimental Results

We evaluate the schemes under two scenarios: 1) a 64GB database with 4.2 billions of 16-byte entries; 2) a 100GB database with 1.6 billions of 64-byte entries. The same as Piano, we use 8-thread parallelization during the preprocessing phase, and only use a single thread during the online phase.

	64GB		**100GB**	
	Piano	Ours	Piano	Ours
	Preprocessing			
Client time	81min	114min	32min	46min
Communication	64GB	64GB	100GB	100GB
	Per query			
Online Time	14.0ms	42.7ms	11.9ms	46.3ms
Online Communication	256KB	5KB	100KB	8KB
Am. Offline Time	3.3ms	4.7ms	2.2ms	3.2ms
Am. Offline Communication	46KB	46KB	120.5KB	120.5KB
Client Storage	419MB	684MB	839MB	1.8GB

Table 3.2: Performance of our scheme and Piano on 64GB and 100GB sized databases. The 64GB database has 16-byte entries and the 100GB database has 64-byte entries. "Am." denotes "Amortized". We report both the online costs and the offline costs amortized over $Q = \sqrt{n} \ln n$ queries.

In Table 3.2, we show the cost for the one-time preprocessing, the online query cost, and also the amortized offline cost. Notice that the amortized offline cost can also be considered as the background maintenance cost that is not on the critical path of the query.

Computation Costs. As seen in Table 3.2, our scheme is worse compared to Piano in terms of computation cost. The offline time is worse by around $1.4\times$ (the same for the maintenance time), and the per query online time is worse by around $3.0\times$. Although the two schemes have the same asymptotic computation costs, our scheme is more complicated, resulting in a larger constant factor. The most significant factor is that our scheme needs to make at least two PRF evaluations per hint, while Piano only makes one PRF evaluation. Also, Piano further optimizes the PRF evaluations. This explains the $3.0\times$ gap.

Communication Costs. Our scheme has the same offline communication cost (same streaming preprocessing) and a much better online communication compared to Piano. The asymptotic online communication is $O_\lambda(n^{1/4})$ for our scheme and is $O(\sqrt{n})$ for Piano. Thus, given a bigger

n, the gap will be larger. The concrete performance also depends on many other factors, including the size of the entries, the size of the chunk and the size of the superblock. We see a $12\times$ gap when n is 1.6 billion in the 100GB database case (with a larger entry size) and a $51\times$ gap when n is 4.2 billion in the 64GB database case (with a smaller entry size).

Storage Costs. Our scheme has worse storage cost compared to Piano. In our experiment, the gap is $1.6 \times -2.1\times$. Our optimized scheme needs to additionally store the superblock parity for each hint, and also stores the constraints generated during the query phase, compared to Piano.

3.7 Additional Result: Sublinear Preprocessing PIR with $\widetilde{O}(1)$ Online Communication from Stronger Assumptions

Previously, we focused on designing preprocessing PIR schemes that rely only on one way functions (OWF). In this section, we show that the "broken hint" technique used in our one-server construction Section 3.4 can also be applied to the state-of-the-art two-server scheme, TreePIR [LP23b]. This gives us a single-server scheme with $\widetilde{O}_\lambda(1)$ online communication cost, $O(\sqrt{n})$ offline communication, and $\widetilde{O}_\lambda(\sqrt{n})$ computation per query, assuming the existence of a classical single-server PIR scheme with polylogarithmic bandwidth.

3.7.1 Privately Puncturable Pseudorandom Set with List Decoding

The elegant TreePIR work [LP23b] constructs a Privately Puncturable Pseudorandom Set with List Decoding (henceforth denoted PPPS^-). Specifically, their implied PPPS^- also emulates the same set distribution $\mathcal{D}_n$ which we inherit in our paper (see Section 3.2.1), and it supports the following operations:

- $\mathsf{sk} \leftarrow \mathsf{Gen}(1^\lambda, n)$: takes in the security parameter 1^λ, the size of the universe n, and outputs a secret key sk representing a set.
- $S \leftarrow \mathsf{Set}(\mathsf{sk})$: takes in a secret key sk and outputs a set S of size $\sqrt{n}$.
- $\mathsf{sk}', i \leftarrow \mathsf{Puncture}(\mathsf{sk}, \ell)$: takes in a secret key sk, a chunk index ℓ, and outputs a punctured key sk' which removes the element in the ℓ-th chunk from the set, as well as auxiliary information i that indicates which of the list decoded answers later is the correct answer.
- $S_0, \ldots, S_{L-1} \leftarrow \mathsf{ListDecode}(\mathsf{sk}')$: takes in a punctured key sk', and outputs a set of candidate sets $S_0, \ldots, S_{L-1}$. It is guaranteed that one of them is the correctly punctured set.

A PPPS^- scheme needs to satisfy the following properties:

1. **Correctness.** For any λ, n, any $\ell \in \{0, \ldots, \sqrt{n} - 1\}$, the following must hold with probability 1: let $\mathsf{sk} \leftarrow \mathsf{Gen}(1^\lambda, n)$, $\mathsf{sk}', i \leftarrow \mathsf{Puncture}(\mathsf{sk}, \ell)$, $S_0, \ldots, S_{L-1} \leftarrow \mathsf{ListDecode}(\mathsf{sk}')$, it must be that S_i is equal to $\mathsf{Set}(\mathsf{sk})$ but with the ℓ-th element removed.
2. **Pseudorandomness.** Defined in the same way as in Section 3.2.1.
3. **Private puncturability.** There exists a probabilistic polynomial-time simulator Sim such that for any n that is polynomially bounded in λ, for any $x \in \{0, \ldots, n-1\}$, the following two distributions are computationally indistinguishable:

- Real: Sample $\mathsf{sk} \leftarrow \mathsf{Gen}(1^\lambda, n)$ subject to $x \in \mathsf{Set}(\mathsf{sk})$, let $\mathsf{sk}', \_ \leftarrow \mathsf{Puncture}(\mathsf{sk}, \mathsf{chunk}(x))$, output sk'.
- Ideal: Let $\mathsf{sk}' \leftarrow \mathsf{Sim}(1^\lambda, n)$, output sk'.

TreePIR [LP23b] implies a PPPS^- scheme constructed from PRFs, satisfying not only the above properties but also the following efficiency requirements:

- *Fast membership*: testing whether an element $x \in \{0, \ldots, n-1\}$ is in the set or not takes $O_\lambda(1)$ time.
- *Small punctured key*: a punctured key has size $\widetilde{O}_\lambda(1)$.
- *Efficient list decoding*: Although ListDecode outputs $\sqrt{n}$ candidate sets, all $\sqrt{n}$ candidate sets has a succinct representation of size $O(\sqrt{n})$; moreover, the ListDecode algorithm runs in $O_\lambda(\sqrt{n})$ time.

3.7.2 A Single-Server PIR Scheme with Polylogarithmic Online Communication

Given a PPPS^- scheme with the aforementioned security and efficiency requirements, we can construct a single-server preprocessing PIR scheme as in Figure 3.7.

We can prove the following theorem about this construction.

Theorem 3.7.1. *Assume the existence of a classical PIR scheme with linear computation, storage and polylogarithmic communication. We can construct a PIR scheme supporting $\sqrt{n}/2$ queries that is private and achieves the following performance bounds:*

- *$O_\lambda(\sqrt{n} \log \kappa\alpha(\kappa))$ client storage and no additional server storage;*
- ***Preprocessing Phase**:*
  - *$O_\lambda(n \log \kappa \cdot \alpha)$ client time and $O_\lambda(n \log \kappa \cdot \alpha)$ server time;*
  - *$O(n)$ communication;*
- ***Query Phase**:*
  - *$\widetilde{O}_\lambda(\sqrt{n})$ expected client time and server time per query;*
  - *$\widetilde{O}_\lambda(1)$ communication per query.*

Further, assuming that n is bounded by $\mathsf{poly}(\lambda)$ and $\mathsf{poly}(\kappa)$, all the $Q = \sqrt{n}/2$ queries in the scheme in Section 3.7 will be answered correctly with probability at least $1 - \mathsf{negl}(\lambda) - \mathsf{negl}(\kappa)$ for some negligible function $\mathsf{negl}(\cdot)$.

Proof. The proof about efficiency is nearly the same as the proof of Theorem 3.4.1 – These are the main differences: the PPPS^- replaces the PPPS here and that the classical PIR scheme that is run during the query phase needs to be taken into account for the storage, computation and communication. Since the storage and computation of the classical scheme is linear, and the communication polylogarithmic, the overall complexities do not change asymptotically. Also, we note in the context of the PPPS^- scheme that the parities of the $\sqrt{n}$ sets can be computed by the server in $O(\sqrt{n})$ time as shown in [LP23b].

The proof of correctness is the same as the analysis in Section 3.4.3, assuming correctness of the underlying classical PIR scheme. The privacy proof is also nearly the same as the privacy proof in Section 3.4.2 except for the following couple of changes: 1) the PIR's simulator additionally

Single-Server Scheme for $Q = \sqrt{n}/2$ Queries

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

Preprocessing. Same as in Figure 3.4 but with no more need for replacement entries.

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

1. For each entry (sk_i, p_i) in the client's hint table, if $x \in \mathsf{Set}(\mathsf{sk}_i)$ (henceforth called a matched entry), do the following unless there are already $M_3 = 3 \log \kappa \cdot \alpha$ matched entries:
   - Let $\mathsf{sk}', i^* \leftarrow \mathsf{Puncture}(\mathsf{sk}_i, \mathsf{chunk}(x))$. Send sk' to the server.
   - Server calls $S_0, \ldots, S_{\sqrt{n}-1} \leftarrow \mathsf{ListDecode}(\mathsf{sk}')$, and for each $i \in \{0, \ldots, \sqrt{n}-1\}$, it computes $\beta_i := \oplus_{j \in S_i} \mathsf{DB}[j]$.
   - Client and server run a classical PIR scheme on the database $(\beta_0, \ldots, \beta_{\sqrt{n}-1})$, and the client retrieves β_{i^*}.
   - If the matched entry is good, client saves the answer $p_i \oplus \beta_{i^*}$.
   - Client samples a fresh PPPS^- key sk_{new} subject to $x \in \mathsf{Set}(\mathsf{sk})$, and replaces the consumed entry (sk_i, p_i) with $(\mathsf{sk}_{\text{new}}, 0)$ and marks the entry as *broken*.
2. Let cnt be the number of matched entries in the previous step. If $\mathsf{cnt} < M_3$, then the client repeats the following $M_3 - \mathsf{cnt}$ times: sample a random PPPS^- key sk subject to $x \in \mathsf{Set}(\mathsf{sk})$, call $\mathsf{sk}', \_ \leftarrow \mathsf{Puncture}(\mathsf{sk}, \mathsf{chunk}(x))$, and send sk' to the server. Invoke a classical PIR scheme with the server to retrieve any index in $\{0, \ldots, \sqrt{n}-1\}$, and ignore the answer received.
3. Client outputs any saved answer. If no answer was saved, output 0.

Figure 3.7: Our single-server preprocessing PIR scheme with $\widetilde{O}(1)$ online bandwidth from a classical PIR scheme.

calls the underlying classical PIR's simulator; and 2) instead of calling the simulator of the PPPS scheme, the PIR's simulator now calls the simulator of the PPPS^- scheme. ■

Part II

Application: Private Information Searching

Chapter 4

Pacmann: Efficient Private Approximate Nearest Neighbor Search

4.1 Overview

So far, most existing Private Information Retrieval (PIR) schemes, including our own work of Piano [ZPSZ24] and QuarterPIR [GZS24], aim to achieve the PIR functionality defined in the 1995 work of Chor et al. [CGKS95]. The original definition is indeed clean and simple in the academic sense: for each query, the client wants to retrieve only a single record from a database without revealing the record's index to the server. Yet this definition is too simplistic for real-world applications. For example, search engines like Google and Bing can search for documents according to the semantic meaning of the client's query, rather than the exact keywords. They also provide similarity search services, such as finding similar images or documents based on files uploaded by the user. These advanced search features are not supported by the original PIR definition, and such a gap between the theoretical definition and practical applications has been a long-standing problem in the field of privacy-preserving information retrieval.

In this work, we aim to bridge this gap by proposing a new private search algorithm that enables *private approximate nearest neighbor search* (ANN). ANN search is a fundamental building block for many advanced retrieval systems, including multimedia information search [LKLJ18, LXY+19], recommendation systems [HSS+20, LLJ+21], and generative AI systems [LPP+20]. In the ANN context, we assume a server is holding a database of n vectors, and a client wants to find the (approximately) most similar vector in the database to a given query vector where the similarity is defined by a distance metric (e.g., Euclidean distance). Specifically, we focus on private ANN search algorithms that allow the client to find the most similar vector in the database without revealing the query vector to the server.

To date, there have been several proposed private nearest neighbor search algorithms, which provide cryptographic privacy guarantees over the user's query [HDCG+23, ABG+24, SSLD22].[1] However, **existing algorithms suffer from poor tradeoffs between: (1) search quality and/or (2) efficiency.** For example, consider Tiptoe [HDCG+23], the state-of-the-art private search algorithm with cryptographically strong privacy guarantees. Tiptoe incurs a linear computation

[1] Cryptographic privacy means that the server learns cryptographically negligible information about the query.

cost for the server per query; that is, the server needs to at least scan through the entire database for each query. Although Tiptoe is parallelizable, the linear computation cost may be a bottleneck for large-scale applications. At the same time, Tiptoe's search accuracy is limited. For example, it only achieves around 40% of the non-private search accuracy in the MS-MARCO dataset.

Private ANN search algorithms can be categorized by two main design choices: the search algorithm and the privacy primitive. These two aspects are closely intertwined: the search algorithm must be chosen to be compatible with the privacy primitive, while also ensuring that information is not leaked to the server and the search quality is not overly degraded. For example, Tiptoe makes two design choices that limit its performance: (1) It uses a clustering-based approximate nearest neighbor algorithm that significantly limits the search quality. (2) Its privacy guarantees are based on a preprocessed somewhat homomorphic encryption (SHE) scheme, which requires significant computation linear in the dataset size. This is a common theme in existing private ANN search algorithms: they either use a simple search algorithm that is not competitive with the state-of-the-art non-private ANN search algorithms, or they rely on expensive cryptographic primitives that incur high computation costs. This leads to the natural question:

> *Can we design a private ANN search algorithm that utilizes advanced search algorithms while achieving strong privacy guarantees with highly efficient privacy mechanisms?*

4.1.1 Our Contribution

In this work, we present PACMANN (Privately ACcess More Approximate Nearest Neighbors), a fully private nearest neighbor search algorithm that addresses the search quality-efficiency tradeoff. We make the observation that the existing private search solutions fail to leverage the graph-based ANN search algorithms [MY18, JSDS+19, IM18] that have shown better quality-efficiency tradeoffs compared to other families of ANN search algorithms (including Local-Sensitive Hashing (LSH) [IM98, SSLD22] and clustering-based algorithms [ABG+24, HDCG+23]). The core challenge lies in the fact that graph-based ANN search algorithms are inherently *iterative* algorithms that require multiple adaptive steps to find the results (See Figure 4.3). Implementing such algorithms with generic privacy-preserving techniques like fully homomorphic encryption (FHE) [Gen09] or secure multi-party computation (SMPC) [CD+15] could incur high overheads.

Our key insight is that we can implement a graph-based ANN search algorithm by running the iterative search locally on the client side, so that the computation can be done in plaintext and with no privacy concerns. To achieve this without the client storing the whole graph structure, we let the server store the graph structure and let the client dynamically fetch necessary information from the server. We now see that this approach reduces the problem to a relatively standard PIR problem: for each step, the client will access the graph information stored on the server to find the next vertex to traverse, without revealing the access pattern to the server. However, if we apply standard PIR schemes [CGKS95] here, the computation per graph access will be at least linear to the size of the whole graph[2], making the whole system inefficient. We utilize Piano [ZPSZ24], our new sublinear PIR construction from the recently popularized client-preprocessing PIR model [CK20] to address the efficiency issue. The Piano PIR scheme allows the client to privately query the

[2]The lower bound is proved in [BIM00].

graph information with sublinear computation and communication per-query costs after a one-time preprocessing phase with linear cost. Moreover, the preprocessing phase can be shared for multiple ANN queries (each ANN query makes multiple accesses to the graph), so the preprocessing cost can be amortized. Finally, we have to make further customizations and optimizations to both the graph-based ANN search algorithm and the Piano PIR to make the whole scheme efficient.

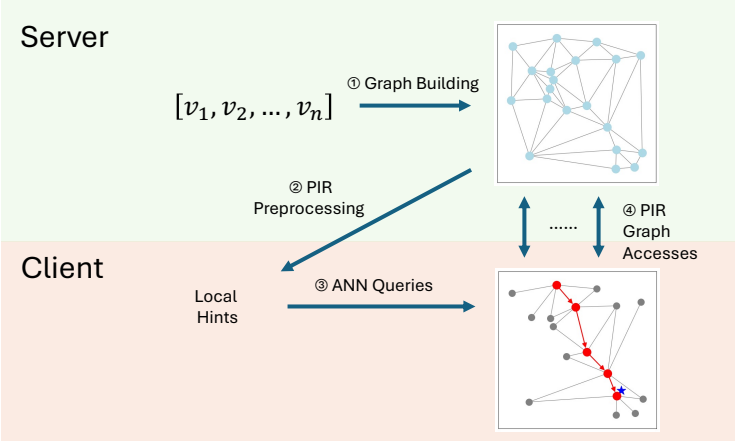
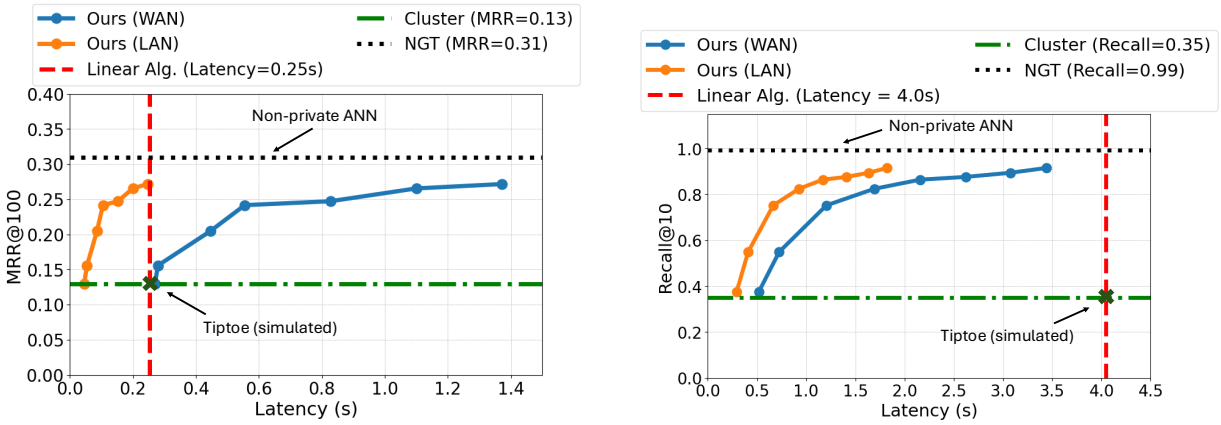


Figure 4.1: The high-level overview of PACMANN. 1) The server builds a graph structure on the database vectors. 2) The client and the server run the preprocessing protocol for the PIR scheme, storing the local hint in the client's storage. 3) The client makes ANN queries. 4) The client runs the graph traversal algorithm locally, but uses the PIR scheme to access the graph information remotely.

Contributions. We summarize our contributions as follows:

1. **Algorithm Design.** We present the design of PACMANN, which builds on a customized graph-based ANN search algorithm and Piano PIR [ZPSZ24]. In graph-based ANN search algorithms [MY18, JSDS+19, IM18], each vertex in the graph represents a database vector and is connected to multiple other vertices based on proximity in a vector (embedding) space. Given a query vector, the algorithms usually start from a given vertex and traverse the graph to find ANNs. In each step of the traversal, the algorithms examine all connected vertices to the current vertex and move to the next vertex that is closer to the query, until a stopping criterion is met. To implement the graph traversal algorithm privately, PACMANN requires the client to *locally* run graph traversal over carefully-selected subgraphs. To efficiently retrieve the appropriate subgraph, we modify the Piano PIR scheme [ZPSZ24] to handle batched queries. Figure 4.1 gives a high-level system overview of PACMANN.
2. **Empirical results.** We empirically evaluate the performance of PACMANN in terms of search quality, query latency, communication, and storage costs. Our results show that PACMANN achieves significantly better search quality than the clustering-based private ANN search algorithms used by state-of-the-art Tiptoe [HDCG+23] and Wally [ABG+24]. For example, our implementation finds the most relevant result in around 63% of the queries on the MSMARCO dataset, compared to 29% for clustering-based algorithms—a **2.1x improvement** in search success rate. In the 100M SIFT dataset, our evaluation shows a **2.5x better** recall@10



(a) Results on MS-MARCO dataset.

(b) Results on SIFT-100M dataset.

Figure 4.2: Tradeoff between search quality and latency. "Linear Alg." is an SIMD-optimized linear algorithm that provides a lower bound on the latency of Tiptoe [HDCG+23], the state-of-the-art private search algorithm. As a safe lower bound, we do not include network latency for "Linear Alg.". "Cluster" is a clustering-based algorithm that provides an upper bound on the quality of Tiptoe. The intersection of the two can be viewed as our (simulated) result for Tiptoe. We use NGT, a state-of-the-art non-private ANN search algorithm [IM18], to establish an upper bound on the search quality. We plot PACMANN's results in both the LAN (5ms RTT) and WAN (50ms RTT) settings.

compared to Tiptoe. Figure 4.2 shows that PACMANN achieves 90% of the search quality of the leading non-private ANN algorithm (NGT [IM18]), measured in mean reciprocal ranking (MRR) and recall. PACMANN also has lower latency than linear computation cost algorithms, including Tiptoe [HDCG+23] and Preco [SSLD22] when the database is larger than 5M records in the LAN setting (i.e., when the search engine and the database are co-located, so network round-trip latency is 5 ms), and 50M records in the WAN setting, respectively. For example, for a database with 100M records, Tiptoe requires at least 4s of search latency, whereas PACMANN achieves 1.6s in the LAN setting **(a 60% reduction)** and 3.1s in the WAN setting **(a 22% reduction)**. Experimental details can be found in Section 4.5.

4.1.2 Related Work

Most Private Information Retrieval (PIR) schemes are designed for the basic array-type access where the client knows exactly the location of the record to query. There are some other works that consider more advanced access patterns like key-value queries [CGN97, PSY23, CD24]. However, ANN search usually requires a more sophisticated search algorithm and thus requires a more complex access pattern.

Several existing works directly consider the private search problem. In addition to Tiptoe [HDCG+23], Wally [ABG+24] improves the efficiency of Tiptoe by relaxing the privacy guarantee to differential privacy, and relying on a large batch of anonymous queries to hide the privacy of an individual query. Nonetheless, Tiptoe and Wally are based on a simple clustering-based

algorithm that partitions the vectors into roughly $\sqrt{n}$ clusters, and only performs exhaustive search in one particular cluster during the search phase. This simple algorithm is not competitive with the state-of-the-art non-private ANN search algorithms. For example, when we evaluate this algorithm on the MSMARCO dataset, its success rate to find the most relevant result is less than 30%, while a non-private ANN search algorithm can have 75% of success rate or even higher. Preco [SSLD22] is another cryptographically private ANN search algorithm that utilizes a Locality Sensitive Hashing (LSH) based search algorithm. Unfortunately, Preco makes a strong assumption that there must be two non-colluding servers to store the database, which is not applicable in many practical scenarios. Preco also requires a linear computation cost on both servers per query.

In an independent and concurrent work, Zhu et al.[ZPZP24] also propose a privacy-preserving ANN search algorithm under a different setting in that the database is provided by the client and outsourced to the server while being encrypted. In that case, the server not only stores the per-client encrypted database, but can also store per-client state to facilitate the search. Their solution is based on the HNSW graph-based ANN indexing algorithm [MY18] and Path Oblivious RAM (ORAM) [SvDS+18]. Our setting is different in that we assume the database is public and shared among all clients, while the server does not have any per-client storage.

4.2 Formal Definitions

K-Approximate Nearest Neighbor Search (K-ANN). Assume a d-dimensional metric space with a distance function $\Delta(\cdot, \cdot)$. Given a database containing n vectors, denoted as $\mathsf{DB} = \{v_1, v_2, \ldots, v_n\} \in \mathbb{R}^{d \times n}$, and a query vector $q \in \mathbb{R}^d$ such that a K-approximate nearest neighbor search algorithm takes the database DB and the query vector q as input, and outputs an index set $I = \{i_1, \ldots, i_K\}$ such that the distances between q and $v_{i_1}, \ldots, v_{i_K}$ are minimized, i.e., $I = \{i_1, \ldots, i_K\}$ is an approximation to the true K-nearest neighbors of q in DB. Specifically, we use the recall or the mean reciprocal rank (MRR) to evaluate the quality of the approximation, depending on the context.

Single Server (Preprocessing) Private K-ANN. A single-server preprocessing private ANN protocol is run between a stateful client and a server. The protocol consists of two phases: preprocessing phase and query phase.

1. **Preprocessing:** The preprocessing phase is run before the query phase and could involve the communication between the client and the server. The server will receive the vector database $\mathsf{DB} \in \mathbb{R}^{d \times n}$ as input.
2. **Queries:** The query phase can include multiple (adaptive) queries from the client. Each query is a vector $q \in \mathbb{R}^d$. The client and the server can have multiple rounds of communication for each query. At the end, the client will output K indices where the corresponding vectors are the K-approximate nearest neighbors of q in DB.

Intuitively, the privacy of a private ANN protocol requires that the server learns negligible information about the query vector q. We will define the privacy of a private ANN protocol with a simulation-based definition. A single-server private ANN protocol satisfies privacy if there exists

a simulator Sim such that for any probabilistic polynomial-time adversary $\mathcal{A}$ acting as the server, the views of $\mathcal{A}$ in the following two experiments are computationally indistinguishable:

- Real: the client interacts with $\mathcal{A}(1^\lambda, \mathsf{DB})$ who acts as the server. In each query step, $\mathcal{A}$ may adaptively choose the next query q for the client. The client is invoked with q as input.
- Ideal: the simulated client $\mathsf{Sim}(1^\lambda, n)$ interacts with $\mathcal{A}(1^\lambda, \mathsf{DB})$ who acts as the server and $\mathcal{A}$ still may adaptively choose the next query q for the client. However, the simulator is invoked with only the knowledge of the size of the database, and without the information of the chosen query q.

Throughout this paper, we assume the adversary is semi-honest. That is, the adversary follows the server's protocol specification but may try to learn additional information from the server's view. We leave the extension to malicious adversaries as future work[3].

4.3 Our Graph-based ANN Construction

As a starting point, we describe our ANN search algorithm construction without privacy considerations. Our construction relies on a customized version of the graph-based ANN search algorithm.

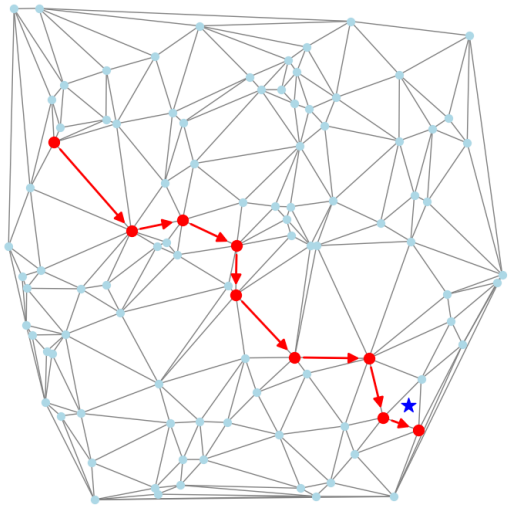


Figure 4.3: An illustration of the graph-based ANN search algorithm on a 2D space. The starting vertex is located around the upper left part of the graph, and the query vector is shown as a blue star. We highlight the path that the algorithm takes to reach the approximate nearest neighbor.

[3]We could potentially use the verifiable PIR techniques [BDKP22] to upgrade the security to malicious adversaries.

4.3.1 Preliminary: Generic Graph-based ANN Search Blueprint

Many popular graph-based ANN search algorithms such as NSW [MPLK12], HNSW [MY18], DiskANN [JSDS+19], NGT [IM18], and FINGER [CCJ+23] follow the same technical blueprint: the algorithm builds a graph based on the input vectors during the preprocessing phase, and then performs a graph traversal algorithm to find the approximate nearest neighbors for a given query vector. We provide a graphical illustration of this process in Figure 4.3 and a pseudocode description in Figure 4.4.

Preprocessing. The preprocessing phase takes in n vectors $\mathsf{DB} = \{v_1, v_2, \ldots, v_n\} \in \mathbb{R}^{d \times n}$ from a metric space and builds an indexing graph G where the n vertices represent the vectors. Different algorithms may have different ways of selecting graph edges. One common approach (e.g., seen in [JSDS+19]) is to connect each vertex v_i to several nearest neighbors, i.e. vectors that are close to v_i in the metric space, as well as multiple distant neighbors to ensure the diversity of the neighbor list [WXYW21]. Most recent ANN algorithms use more advanced structures on top of the indexing graph (e.g. hierarchical structure in HNSW [MY18] or auxiliary tree-structure in NGT [IM18]) to improve the search efficiency.

Query. Given a query vector q, the query algorithm performs a graph traversal algorithm on the index graph G and outputs a vertex u^* as the approximate nearest neighbor of q. The search algorithm starts from an entry point u_{start}, often picked as the closest vertex to the centroid of DB. We denote the set of neighbors of a node v as $N(v)$. In each hop, the search algorithm greedily moves from the current vertex v to its neighbor in $N(v)$ that is the closest to the query vector q. The algorithm terminates when the number of hops reaches a certain threshold or a local minimum is met, i.e. none of the neighbors is closer to q than the current iterate. This can be easily extended to outputting K approximate nearest neighbors by keeping track of the visited vertices and outputting the top K nearest vertices in the visited set.

4.3.2 Our Customized Graph Building Algorithm

To make graph-based ANN search private, PACMANN imposes two constraints: (1) We retain the single graph structure, as opposed to adding auxiliary structures, as is done in the non-private setting. This is done to facilitate the upgrade to the privacy-preserving version. (2) We enforce regularity on the graph's out-degree distribution. This prevents information leakage from the number of neighbors accessed. These properties are not met by existing ANN algorithms, and we customize the graph-building algorithm to meet them. We now describe the high-level idea to build a regular C-out directed indexing graph.

Start from an unbalanced graph. Our first observation is that we can take advantage of existing ANN libraries to find nearby neighbors for all vectors in the database and treat them as the candidate neighbors in the graph. Specifically, we find $2C$ approximate nearest neighbors as neighbor candidates for each vertex using an existing ANN library (e.g., NGT [IM18]), then trim the candidate list to C candidates with the sparse-neighborhood-graph (SNG) heuristic [AM93]. Roughly, SNG sorts the candidates by their distance to the vertex and adds candidates to the neighbor list one by one. It only adds a candidate if it is not too close to a neighbor already in the list; this is done to ensure diversity (see Figure 4.5 for the details).

Figure 4.4: Description of the graph based ANN search algorithm including the optional optimizations. The pseudocode can be read in the following two ways: 1) the non-highlighted part describes the generic graph-based ANN search algorithm; 2) the full description, including the optimizations we introduced in Section 4.4.2, describes our customized version of the algorithm. In the non-private setting, the algorithm is run on the server. In the private setting, the client will run the query algorithm locally, and use Batched Piano PIR to perform the information retrieval dynamically and privately.

Graph-based ANN Search Algorithm

Preprocessing.

- Input: a set of n vectors $\mathsf{DB} = \{v_1, v_2, \ldots, v_n\} \in \mathbb{R}^{d \times n}$.
- Output: a directed graph G with n vertices corresponding to the n vectors in DB, and entry point(s) u_{start}, $u^1_{\text{start}}, \ldots, u^{\sqrt{n}}_{\text{start}}$.
  1. $G \leftarrow \mathsf{ANN.BuildGraph}(\mathsf{DB})$; *// Description in Section 4.3.2*
  2. Let the closest vector to the centroid be the starting vertex $u_{\text{start}} = \arg\min_{u \in [n]} \Delta(v_u, \frac{1}{n}\sum_{j \in [n]} v_j)$.
  3. Let the extra starting vertices be $\sqrt{n}$ random vertices $u^1_{\text{start}}, \ldots, u^{\sqrt{n}}_{\text{start}}$ sampled from $[n]$.

Query.

- Parameters: max hop number H, beam search width m denoting the number of parallel paths to explore.
- Input: a query vector $q \in \mathbb{R}^d$.
- Output: an index u^* such that v_{u^*} is recognized as an approximate nearest neighbor of q.

1. Let $u = u_{\text{start}}$.
2. Let $u_1, \ldots, u_m$ be the top m vertices in $u^1_{\text{start}}, \ldots, u^{\sqrt{n}}_{\text{start}}$ that are closest to q.
3. For $t = 1, 2, \ldots, H$:
   - Update u as follows:
     - Read the neighbor list $N(u)$ from G and all vectors v_j for $j \in N(u)$ from DB.
     - Let $u' = \arg\min_{j \in N(u)} \Delta(v_j, q)$.
     - If $\Delta(u', q) \leq \Delta(u, q)$, update $u \leftarrow u'$.
   - Similarly, update $u_1, \ldots, u_m$.
4. Let $u = \arg\min_{u' \in \{u, u_1, \ldots, u_{\sqrt{n}}\}} \Delta(u', q)$.
5. Output $u^* = u$.

We temporarily add directed edges between each vertex and its C neighbor candidates *in both directions* to ensure the graph is well-connected, i.e. so that each vertex has at least C inbound and C outbound directed edges.

Balancing the graph. We now have a graph that could be highly unbalanced in terms of the degrees. We use a sampling technique to balance the graph. That is, for each directed edge $(x \to y)$, we keep it with a probability of $C/\mathsf{InboundDegree}(y)$. This ensures that a vertex will have C inbound edges in expectation after the sampling process. Then, we ensure that each vertex has exactly C outbound edges. For those vertices with more than C outbound edges, we again use the SNG heuristic to trim the outbounds to C. For those vertices with fewer than C outbound edges, we connect them to vertices selected uniformly at random.

A detailed description of our customized graph building algorithm is provided in Figure 4.5.

4.4 PACMANN: Private Graph-based ANN

We next describe how PACMANN protects query privacy over our customized graph-based ANN search algorithm in Section 4.3. We start with a basic, inefficient construction, then show how PACMANN optimizes it. Due to space constraints, we present the full construction of PACMANN in Section 4.4.3.

4.4.1 Private Graph-based ANN Search with PIR

As a strawman scheme, one could try to protect query privacy by implementing the search algorithm with generic cryptographic primitives, such as fully homomorphic encryption [Gen09]. and/or multi-party computation [CD+15]. However, the search algorithm of graph-based ANN is inherently iterative and adaptive in that during the graph traversal process, each hop's vertex is dynamically determined by both the previous hop's vertex and also the query vector. Handling such (adaptive) iteration is a common challenge in designing cryptographic protocols [4] as observed by previous works [BFK+09, DdSGOTV22, CGG+24, GHAHJ22, HKP21]

Localized Searching. Our main idea is to *perform the iterative graph traversal algorithm on the client side*, completely removing the privacy concern of performing iterative computation on the server. Naively, this would require the client to store the whole indexing graph, which is impractical in terms of storage cost. Instead, we let the client dynamically and privately retrieve subgraph information from the server. Specifically, during search, the client can retrieve the neighbor list $N(v_i)$ of the current vertex v_i and also all vectors in the neighbor list from the server for each hop. The client then computes the distances between the query vector q and all the neighbor vectors in $N(v_i)$ locally, and chooses the closest neighbor within that set. This process is repeated until the search terminates.

Protecting Retrieval Privacy with PIR. Even without directly observing the query vector, the server can infer non-trivial information from the vertex indices retrieved by the client. For example, if the server learns that the client retrieves many vertices whose underlying records represent

[4]The standard FHE or MPC techniques are designed for the so-called circuit model, where the computation flow is fixed and known in advance. Although the circuit model is indeed Turing-complete, it does not naturally support a random-access memory and control flow operations like if-else conditions. In the context of graph-based ANN search, we do need to (adaptively) read the graph from a random access memory, so the standard techniques are not directly applicable.

Our Graph Building Algorithm: ANN.BuildGraph.

Input: a set of n vectors $\mathsf{DB} = \{v_1, v_2, \ldots, v_n\} \in \mathbb{R}^{d \times n}$ and a target degree C.

Output: a directed regular C-out graph G with n vertices corresponding to the n vectors in DB.

Subroutine: ANN.TrimNeighbors(u, L, C). *// Given a vertex u and a list of neighbors L, return the picked C neighbors of u in L with the SNG heuristic.*

1. Let w be the number of vertices in L and let $(i_1, \ldots, i_w)$ be the sorted indices in L according to the distance to the vector v_u (ascendingly).
2. Let Kept $\leftarrow \emptyset$, Dscd $\leftarrow \emptyset$.
3. For $j = i_1, \ldots, i_w$:
   (a) If $\exists j' \in$ Kept such that $\Delta(v_j, v_{j'}) < \Delta(v_j, v_u)$, label j as discarded. *// SNG heuristic*
   (b) If j is not discarded, insert j into Kept. Otherwise, insert j into Dscd.
   (c) Terminate if $|\mathsf{Kept}| = C$.
4. If $|\mathsf{Kept}| < C$, insert the first $C - |\mathsf{Kept}|$ vertices in Dscd to Kept.
5. Return Kept.

Graph Building.

1. For each $u \in [n]$:
   (a) Let N_u be the set of $2C$ approximate neighbor indices to v_u obtained by an underlying ANN algorithm.
   (b) Trim N_u to C neighbors: let $N_u \leftarrow$ ANN.TrimNeighbors(u, N_u, C).
   (c) For all $u' \in N(u)$, add directed edges $(u \to u')$ and edge $(u' \to u)$ to the graph G.
2. Fix InDegree(u) to be the in-degree of vertex u in the current graph G.
3. For each edge $(i, j) \in G$: keep the edge in G with probability $\frac{C}{\mathsf{InDegree}(j)}$.
4. For each $u \in [n]$:
   (a) Denote $N_G(u)$ as the set of outbounds of u in G.
   (b) If $|N_G(u)| < C$: add edges from u to random vertices in $[n]$ until $|N_G(u)| = C$.
   (c) If $|N_G(u)| > C$: let $N_G(u) \leftarrow$ ANN.TrimNeighbor$(u, N_G(u), C)$.
5. Output the graph G.

Figure 4.5: Description of our graph building algorithm ANN.BuildGraph.

documents about food topics, the server can infer that the client is searching for information about foods. Thus, the final challenge is how the client can retrieve the graph information (including the neighbor lists and the vectors) without revealing which vertex it is retrieving.

We use a Private Information Retrieval (PIR) scheme for this purpose. Standard PIR [CGKS95] allows a client to retrieve one or multiple entries from a server-stored database of n entries without revealing the indices of the entries to the server.

To integrate a PIR scheme into our private graph-based ANN search algorithm, we can run a standard PIR protocol for a database storing all the vertex information in the graph, where each entry i stores the vector v_i and also the neighbor list $N(i)$. Whenever the client needs to access the graph, it issues a PIR query to retrieve the information privately. Notice that by merging the vector information and also the neighbor information into a single database, we can finish each hop in one round of communication.

Specifically, we choose to use our Piano PIR scheme (Chapter 2) as the underlying PIR scheme. Piano achieves practical efficiency for the single-server setting, matching our requirements. Given a database with n entries, Piano achieves $\tilde{O}(\sqrt{n})$ computation and communication per query with $\tilde{O}(\sqrt{n})$ client storage after a one-time preprocessing phase that incurs linear computation and communication costs. Other alternatives of single-server PIRs [GZS24, RMS24, WLZ+23] could also be used in our construction for similar efficiency guarantees.

In the algorithm description, we use the following syntax of to capture the Piano PIR functionality with the batch-mode interface:

- $\mathsf{hint} \leftarrow \mathsf{PIR.Prep}(1^\lambda, \mathsf{DB})$: given the security parameter 1^λ and the database DB, generate the preprocessed PIR hint hint.
- $\mathsf{msg} \leftarrow \mathsf{PIR.BatchQuery}(\mathsf{hint}, \mathsf{batch})$: given the PIR hint and a batch of query indices $\mathsf{batch} \subseteq [n]$, the client generates the PIR query message msg.
- $\mathsf{ans} \leftarrow \mathsf{PIR.BatchAnswer}(\mathsf{DB}, \mathsf{msg})$: upon receiving the batched PIR query message msg, the server generates the batched answer ans.
- $(\beta, \mathsf{hint}') \leftarrow \mathsf{PIR.BatchRecover}(\mathsf{hint}, \mathsf{ans}, \mathsf{batch})$: given the batched answer ans and the PIR hint hint, recover the answers β and update the hint to hint'. Here, β is a set that includes all the successful query indices and their corresponding answers. That is, β includes the tuples $(i, \mathsf{DB}[i])$ for all $i \in \mathsf{batch}$ that are successfully queried.

By applying the Piano PIR scheme to our private ANN construction, after a linear cost preprocessing phase, each PIR query in the graph traversal algorithm will incur $\widetilde{O}(\sqrt{n})$ computation and communication cost. Again, if we assume the search algorithm takes H hops and each hop requires the client to retrieve the information of C neighbors, the overall running time of the scheme will be $\widetilde{O}(HC\sqrt{n})$. As long as $H \cdot C$ is $o(\sqrt{n})$, the resulting scheme will have sublinear computation cost, being more efficient than the prior private ANN schemes [HDCG+23, SSLD22].

Tradeoff and Practicality of Preprocessing. Despite significant online efficiency improvements, the Piano PIR client must download the whole database (i.e., the indexing graph) in a streaming fashion during the preprocessing phase. Here, streaming means that the client only stores a small portion of the database to preprocess at a time, but the total amount of data the client has to download is still the whole database. Therefore, our scheme is more suitable for a client with a good network connection. In our evaluation where we assume a good network connection (1 Gbps), the preprocessing phase takes around 5 minutes and 3 GB of client storage space for a database with 100 million vectors, and the total download communication is around 60GB. An alternative approach to reduce the preprocessing cost is to have a second server to preprocess

the database and directly provide the client with the preprocessed results, as suggested in many existing PIR schemes [ZPSZ24, LP23b, GZS24, KCG21].

4.4.2 Necessary Optimizations

In practice, we observe that the total number of visited vertices in the graph-based search algorithm (that is, the product between max hop number H and the graph out-degree C) tends to be around a few hundreds or thousands. We describe the necessary optimizations here and conduct an ablation study in the evaluation section to show the effectiveness of these optimizations in Section 4.5. Empirically, we observe that our optimizations reduce the concrete computation cost by 76%, and the overall latency by nearly 70% (including the communication time).

Beam search. The basic description of the graph-based ANN search algorithm Figure 4.4 traverses the graph in a single path. In practice, we can explore m multiple paths in parallel. This approach is used in other graph-based ANN algorithms [JSDS$^+$19]. Although doing so increases the query cost per hop, we observe that beam search significantly reduces the total hops to reach a given search quality, and is thus essential to reducing the overall latency.

Fast Starting. Intuitively, if the starting vertex is already very close to the query vector q, we can reach the approximate nearest neighbors with fewer hops. With fast starting, we let the client store around $O(\sqrt{n})$ vertices' information locally, and scan all these vertices to find those that are already close to q to start the searching algorithm. We get this benefit for free, because Piano PIR already requires the client to store $\tilde{O}(\sqrt{n})$ vectors locally, which are selected uniformly at random.

Batched PIR Query. We observe that the PIR queries in each hop are parallel and can be handled in a single batch. We can use a batching technique called partial batch retrieval (PBR) [SSLD22] to further improve the efficiency. In PBR, we can use a pseudorandom permutation to permute the database upfront, and then partition the permuted database into B sub-databases where each sub-database has n/B entries. On average, each sub-database will have Q/B queries given a batch of Q queries (if we are using beam search, $Q = mC$). By doing this, each PIR query will be issued to a sub-database of size n/B instead of the full database, saving the computation and communication cost. To ensure privacy, the client has to issue a fixed number of queries for each sub-database, say choosing the number $T = 1.5Q/B$. If fewer than T queries are in the same sub-database, the client submits dummy queries. If more than T queries are in the same sub-database, the client only submits the first T queries and discards the rest. For example, if we have a batch size $Q = 32$ and set partition number $B = 16$ and $T = 3$, around 90% of the queries in the batch will be successfully submitted in expectation. [5] Intuitively, if each vertex has 32 neighbors, our retrieval process can retrieve nearly 30 neighbors' information with a single

[5]We can approximate this process with randomly throwing Q balls into B bins, each with a maximum load of T, and calculate how many balls are discarded due to overflow in expectation. Given some fixed order of throwing the balls, the probability that the i-th thrown ball is thrown into a fully-loaded bin is always $\Pr[\mathsf{Bin}(i-1, 1/B) \geq T]$. Here, $\mathsf{Bin}(t, p)$ denotes the Binomial distribution – that is, the number of successful trials in t repeated, independent trials where each trial has success probability of p. Then, the expected number of discarded balls (i.e., failed queries) in total is $\sum_{i=1}^{Q} \Pr[\mathsf{Bin}(i-1, 1/B) \geq T]$.

batched query. Since the graph tends to be highly-connected, we observe that even with a mild query failure probability, the algorithm still finds good results with more hops.

Assume we are using the Piano PIR scheme that has $O(\sqrt{n})$ computation/communication cost per query. If we split the database into B sub-databases, the total cost for a Q-size batch will be $O\left(\left(\frac{Q}{B}+B\right)\cdot\sqrt{\frac{n}{B}}\right)$. The optimal selection will be $B=\Theta(Q)$ and the cost will be $O(\sqrt{Qn})$. This gives us a $O(\sqrt{Q})$ improvement in the efficiency compared to naively issuing Q independent queries that has $O(\sqrt{n}Q)$ cost.

4.4.3 Putting Everything Together

We include the full description of our algorithm in Figure 4.6. The algorithm further involves the local caching optimization that the client will keep track of all queried vertices and avoid repetitive queries.

Preprocessing. We use the same graph building algorithm as in Figure 4.5 to build a graph G from the vectors DB. We then combine the neighbor list of each vertex with its vector to build a new database $\mathsf{DB}'=\{(v_i,N(i))\}_{i\in[n]}$. Then, we randomly sample $\sqrt{n}$ starting vertices $u^1_{\text{start}},\ldots,u^{\sqrt{n}}_{\text{start}}$ from $[n]$. We then run the client-side preprocessing algorithm of Piano and let the client store the hint hint. Further, we let the client download those starting vertices and their vectors and neighbor lists from the server.

Query for K-approximate nearest neighbors of vector $q\in\mathbb{R}^d$. We generalize the search algorithm to the following exploration process. The client can maintain two sets of vertices locally: 1) $\mathsf{visited}$ including all vertices that the client has already visited and stored their vectors and neighbor lists, and 2) $\mathsf{developed}$ including all vertices that the client has already visited and retrieved all their neighbors' information. The client can maintain $\mathsf{visited}$ as a priority queue where the closer vertices to q are at the front. Each time the client can pick multiple vertices from $\mathsf{visited}$ and fetch their neighbors' information in parallel. The client will move those picked vertices from $\mathsf{visited}$ to $\mathsf{developed}$, and add their neighbors to $\mathsf{visited}$. We collect all those neighbors' indices as batch and issue a batched PIR query to the server. After receiving the batched answer, the client will recover all the successful queries and push them to $\mathsf{visited}$. The client also needs to update the local hint hint after each query. We can repeat this process until the max hop number is reached. Finally, the client can output the top K nearest vertices in $\mathsf{visited}$ and $\mathsf{developed}$ to the query vector q.

We summarize the main result in the following theorem.

Theorem 4.4.1. *Assume there exists one-way functions. Consider a database with n vectors in $\mathbb{R}^d$. Assuming a non-private graph-based ANN algorithm that preprocesses the database into a C-degree graph, and searches for the approximate nearest neighbors with H hops in the graph. Then, by applying the Piano PIR scheme, we can build a private ANN scheme with the following efficiency, while achieving the same search quality as the non-private graph ANN:*

- *Preprocessing: $\tilde{O}(n(d+C))$ client and server time and $O(n(d+C))$ communication;*
- *Query: $\tilde{O}(H\cdot\sqrt{Cn}\cdot(d+C))$ client and server time and $O(H\cdot\sqrt{Cn}\cdot(d+C))$ communication.*
- *Storage: $\tilde{O}(\sqrt{n}\cdot(d+C))$ client storage and no additional server storage except the database.*

Proof. The proof follows directly from the efficiency of the Piano PIR scheme when we consider a database with n entries and each entry stores a vector in $\mathbb{R}^d$ and a neighbor list of size C. Moreover, by applying the batched optimization we described in Section 4.4.2, we further reduce the computation and communication cost of each PIR query by $\sqrt{C}$. The privacy proof is fairly straightforward, as the server only sees multiple Piano PIR queries from the client. We can construct a simulator for the Ideal experiment that simulates the server's view by invoking the simulator of the Piano PIR scheme for each query issued from the client. It follows from the privacy of the Piano PIR scheme that the two views (the Real and the Ideal experiments) are computationally indistinguishable. ■

4.5 Evaluation

We evaluate PACMANN's performance in terms of search quality and latency and compare it against two baselines: a state-of-the-art private ANN search algorithm, Tiptoe [HDCG+23], and a non-private ANN search algorithm, NGT [IM18]. Our evaluation results show that PACMANN achieves better search quality with lower online query latency than Tiptoe in two real datasets, SIFT and MS-MARCO. We provide more details in Section 4.5.3, including a detailed breakdown of the preprocessing, storage, and communication cost, and a discussion about quantization and the alternative implementations.

4.5.1 Evaluation Setup

Our open-source implementation[6] uses Python for data preprocessing and Golang for the core algorithm, including the graph ANN algorithm and the PIR scheme. Details of the implementation are provided in Section 4.5.3. The experiments are run on a single server with a 2.4GHz Intel Xeon E5-2680 CPU and 256 GB of RAM. We evaluate the latency numbers on two simulated network settings, the local area network (LAN) setting with 5ms round-trip-time (RTT) and the wide area network (WAN) setting with 50ms RTT. Although our implementation supports multi-threading optimization, we only use a single thread for all the experiments for a fair comparison.

Quality Metrics

Recall: Recall is the standard metric used in evaluating the quality of ANN algorithms [ABF20]. For each query q, if the algorithm outputs a K-index set I, and the top K ground truth indices are I^*, the recall@K is defined as: $\mathsf{Recall@}K = \frac{|I^* \cap I|}{K}$.

Mean Reciprocal Rank (MRR): MRR is a standard quality metric for information retrieval systems [NRS+16]. Given a query q, assume there is a ground truth index i^* such that $\mathsf{DB}[i^*]$ is the most relevant entry. If the client outputs a list of indices that actually contains i^* at the j-th rank where $j \leq K$, the reciprocal rank (denoted as RR@K) score is $1/j$. Otherwise, RR@K is 0. MRR is defined as the average of RR@K over multiple queries.

Datasets

[6] https://github.com/privsearch/private-search-temp

PACMANN: Private Graph ANN with Preprocessing

Preprocessing.

- Server Side Preprocessing: *Input: a set of n vectors* $\mathsf{DB} = \{v_1, v_2, \ldots, v_n\} \in \mathbb{R}^{d \times n}$.
  - Runs $G \leftarrow \mathsf{BuildGraph}(\mathsf{DB})$.
  - Randomly sample $\sqrt{n}$ starting vertices $u^1_{\text{start}}, \ldots, u^{\sqrt{n}}_{\text{start}}$ from $[n]$.
  - Let $\mathsf{DB}' = \{(v_i, N(i))\}_{i \in [n]}$ where $N(i)$ is the neighbor list of i in G.
    - Client Side Preprocessing:
      - Stores hint $\leftarrow \mathsf{PIR.Prep}(1^\lambda, \mathsf{DB}')$. *// Involving communication with the server.*
      - Downloads the representative indices $u^1_{\text{start}}, \ldots, u^{\sqrt{n}}_{\text{start}}$ from the server.
      - For all $u \in \{u^i_{\text{start}}\}_{i \in [\sqrt{n}]}$, downloads $\mathsf{DB}'[u] = (v_u, N(u))$ from the server.

Query for K-approximate nearest neighbors of vector $q \in \mathbb{R}^d$.

- Parameters: max hop number H and beam search width m.
- Client Side Algorithm.
  - Let visited $= \{u^i_{\text{start}}\}_{i \in [\sqrt{n}]}$ be a priority queue where the closest vertices to q are at the front.
  - Let developed $= \emptyset$.
  - For $t = 1, 2, \ldots, H$:
    - Let $u_1, \ldots, u_m$ be the top m indices in visited and move them from visited to developed.
    - Let batch $= \cup_{i \in [m]} N(u_i)$ be the union of the neighbor lists of $u_1, \ldots, u_m$.
    - Remove those indices in batch that are already in visited $\cup$ developed.
    - Send msg $\leftarrow \mathsf{PIR.BatchQuery}(\mathsf{hint}, \mathsf{batch})$ to the server.
    - Upon receiving ans from the server, run $(\beta, \mathsf{hint}') \leftarrow \mathsf{PIR.Recover}(\mathsf{ans}, \mathsf{hint})$. where β is a set that includes all the successful query indices and their corresponding answers.
    - For each successful query tuple $(u, (v_u, N(u))) \in \beta$, add u to visited and store the vector v_u and neighbor list $N(u)$ locally.
    - Update the local PIR hint hint $\leftarrow$ hint$'$.
  - Finally, for all $u \in$ visited $\cup$ developed, compute the distance $\Delta(v_u, q)$. Output the k indices with the smallest distances.
- Server Side Algorithm.
  - Upon receiving a PIR batch query message msg from the client, run ans $\leftarrow \mathsf{PIR.BatchAnswer}(\mathsf{DB}', \mathsf{msg})$ and return ans.

Figure 4.6: Our private ANN scheme with preprocessing.

SIFT Dataset [JTDA11] We use the first 100 million 128-dimensional vectors from the SIFT dataset for our evaluation. We vary the database size by picking the first 2 million to 100 million vectors. We test 1,000 top-10 queries in the SIFT query set for each configuration, and measure the average recall@10 (the top-10 ground truth nearest neighbors for each query are provided by the dataset).
MS-MARCO Dataset. [NRS+16] The dataset contains 3.2 million text documents with more than 5000 test queries such that the top 1 relevant document is provided. We follow the same procedure as in the Tiptoe paper [HDCG+23] to process the dataset: 1) embed the documents and the queries into a 768-dimensional vector space with sentence-BERT [RG19]; 2) reduce the dimensions to 192 with PCA. We follow the same quality evaluation metric as in the Tiptoe paper that we compute the average MRR@100 for the first 1000 queries.

Baselines

- *Tiptoe (Simulated).* We aim to compare our scheme against the state-of-the-art private ANN search algorithm, Tiptoe [HDCG+23]. Due to resource constraints, we were unable to run the full Tiptoe system, in which the dominating cost comes from the clustering step that requires dozens of servers running hundreds of core-hours as reported in the original paper. Hence, we simulated Tiptoe by using two baselines as follows.
  - Latency lower bound: *Linear Algorithm.* The Tiptoe algorithm computes n inner products per query with preprocessed homomorphic encryption. As a latency lower bound, [7] we implement a linear time algorithm that only computes the inner product between the query and each database vector in plaintext. We optimized the implementation using AVX-512 instructions. The throughput of this algorithm is approximately 11.9 GB/s, matching the memory bandwidth of the machine. Finally, we do not include network latency for this baseline.
  - Quality upper bound: *Cluster.* We replicate the clustering-based ranking algorithm used in Tiptoe with full precision (32-bit)[8]. We cluster the vectors into $\sqrt{n}$ clusters offline. For each query, we find the closest cluster centroid, and search for the nearest neighbor within the cluster.
- *NGT.* NGT is one of the state-of-the-art non-private ANN search algorithms. We use the implementation from the GoNGT library [IM18]. We compare our scheme against NGT in terms of search quality.

4.5.2 Implementation Details

We describe our implementation of the two components below.

Graph-based ANN. The graph-based ANN algorithm can be divided into two parts: the graph building part and the query part. The graph building part follows our description in Figure 4.5 and sets the outbound number $C = 32$. The query part strictly follows our description in Figure 4.6.

[7] We compared the simulated online latency with the actual numbers reported in the original paper [HDCG+23], and the simulated latency is 8% less than the reported latency. See the detailed results in Section 4.5.3.

[8] Our reimplementation of the clustering-based algorithm reached 0.15 MRR@100 on the MS-MARCO dataset, which is better than the 0.13 MRR@100 reported in the Tiptoe paper [HDCG+23].

Batched PIR. We implement a batch-mode version of the Piano PIR scheme [ZPSZ24] by first implementing the single-query version of the Piano PIR scheme with 128-bit security parameter, then wrapping it with a batch-mode interface. The interface simply splits the whole database into B partitions, and then runs a single-query Piano PIR scheme on each partition parallelly. Given a batch of Q queries, we simply identify which partition each query belongs to, and then make Q/B queries at each partition to ensure privacy. We choose $B = 16$ and $Q = 32$ in our evaluation. Moreover, we implement the automatic maintenance for the client state. We keep track of the consumed hints in the client's space and automatically run another preprocessing phase when the hints are exhausted.

4.5.3 Evaluation Results

PACMANN provides a favorable tradeoff between privacy, search quality, and latency. We first measure the tradeoff between search quality and query latency. Specifically, by increasing the number of rounds and the number of parallel queries in each round, our algorithm can achieve higher search qualities at the cost of higher latency. The results are shown in Figure 4.2. We consider a wide range of search quality requirements, where the lower bound is set by the cluster search algorithm (replicating the Tiptoe search algorithm), and the upper bound is set by the non-private ANN algorithm, NGT. We observe that our scheme provides a wide range of tradeoffs between quality and latency, and can indeed achieve approximately 90% of NGT's search quality. In the SIFT-100M dataset, the advantage of our scheme is more significant, and even with 91% recall@10, our scheme still has a lower latency than the linear algorithm baseline.

Scalability: PACMANN outperforms Tiptoe in latency and accuracy on datasets of at least 2M-50M records. We next evaluate the scalability of our scheme by scaling the database size up to 100 million vectors. We tune the parameters of our scheme to achieve 0.90 recall@10 for each data point; that is, 9 out of 10 of the search results are indeed ground truth top-10 nearest neighbors on average. The results are shown in Figure 4.7. We observe that in the LAN setting where the computation time is dominant, our scheme beats the linear algorithm baseline when the database size is larger than 5M. In the WAN setting where the communication time is more dominant, our scheme beats the linear baseline when the database size is larger than 50M. We observe that the number of rounds to achieve a certain recall@10 increases roughly logarithmically with the database size, and we know from the theoretical analysis that PIR cost scales with $\sqrt{n}$. This suggests that the total latency of our scheme increases sublinearly with the database size.

Ablation study: Our optimizations give a 70% latency reduction. Finally, we conduct an ablation study on the 10M subset of the SIFT dataset to evaluate the optimizations in Section 4.4.2: 1) *Beam Search:* The number of parallel paths in the beam search algorithm is increased from 1 to 3. 2) *Fast Starting:* The client starts the search by choosing the starting vertices from $\widetilde{O}(\sqrt{n})$ preprocessed vertices. 3) *Batched PIR:* We enable the batch-mode PIR for each round of the search.

We compare four configurations in Figure 4.8, where we increase the number of rounds until the quality reaches 0.90 recall@10. The beam search optimization significantly reduces the maximum number of rounds needed to achieve the same quality by 3x. Adding the fast-start strategy furtuher reduces the required rounds by 20%. When we enable the batch-mode PIR,

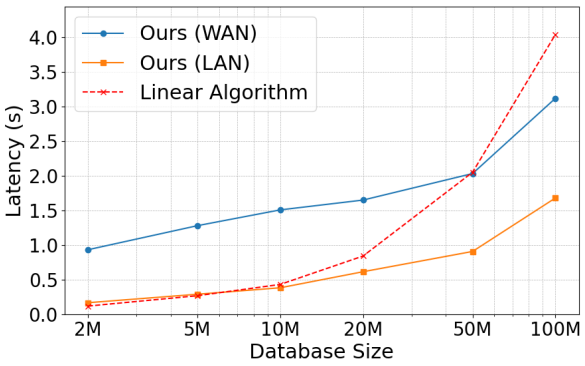


Figure 4.7: Latency results on different database sizes, sampled from the SIFT dataset. We tune the parameters of our scheme to achieve 0.90 recall@10 for each data point; For each data point, the total latency is the sum of the actual computation time and the round-trip time of the communication, multiplied by the number of rounds. We plot both the LAN setting (5ms rtt) and the WAN setting results (50ms rtt) in this figure.

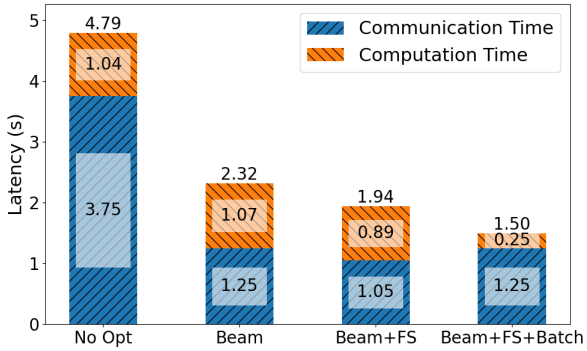


Figure 4.8: Ablation study on the 10M subset of SIFT (WAN setting). Given a fixed configuration, we increase the max number of rounds until the quality reaches 0.90 recall@10. "No Opt" means no optimizations are enabled. "Beam" means the beam search optimization. "FS" means the fast starting strategy. "Batch" means we enable the batch-mode Piano PIR. Our full implementation enables all the optimizations.

we observe that the computation time in each round is reduced by 4x, but, as we mentioned in Section 4.4.3, we do introduce some query failures, which we need to balance with the slight increase in the number of rounds.

4.5.4 Detailed Breakdown

Finally, we provide a detailed breakdown of the costs of our scheme in Table 4.1. We pick two representative parameter settings for the MS-MARCO dataset and the SIFT dataset, all achieving 90% of the quality of the state-of-the-art non-private ANN search algorithm. The graph building time is significant for both datasets, taking 8.5 minutes for the MS-MARCO dataset and 343.5 minutes for the SIFT dataset. However, the graph building only happens once on the server side. The PIR preprocessing is per-client, but it is relatively cheap, taking 9.1 seconds for the MS-MARCO dataset and 271.6 seconds for the SIFT dataset. The most expensive part is that during the preprocessing, the client has to scan over the whole index structure in a streaming fashion, incurring large communication cost. For each online query, the latency and computation time match our analysis in Section 4.5.3. Notably, the communication cost on the critical path is relatively low, taking 1.5MB for the MS-MARCO dataset and 14.4MB for the SIFT dataset. Notice that the client has to update its local state after each query. We see that the maintenance time per query is relatively cheap, but the communication cost is high, taking 60.1MB for the MS-MARCO dataset and 399.4MB for the SIFT dataset. Theoretically, the client stores around $\widetilde{O}(n)$ amount of local state. Empirically, we see that the client stores 0.6GB of data for the MS-MARCO dataset and 2.9GB of data for the SIFT dataset, and the scaling factor matches our

	MS-MARCO (3M)	SIFT (100M)
Preprocessing		
Graph Buliding Time	8.5 min	343.5 min
PIR Preprocessing	9.1 s	271.6 s
Communication	2.7 GB	59.6 GB
Online per query		
Latency	1.1 s	3.0 s
Computation Time	0.10 s	1.48 s
Online Communication	1.5 MB	14.4 MB
Rounds	20	32
Maintenance per query		
Time	0.19 s	1.99 s
Communication	60.1 MB	399.4 MB
Client Storage	0.6 GB	2.9 GB

Table 4.1: Detailed breakdown of our results on different datasets in the WAN setting. We list the preprocessing, online query, and maintenance costs. The graph-building cost happens only once. The PIR preprocessing cost is incurred when each client joins the system. The online query cost is the cost on the critical path of the search queries. Following each online query, there is a necessary one-round maintenance to update the client state. We use 16 threads for the graph building and one thread for other parts. The corresponding quality for the experiments is 0.266 MRR@100 for MS-MARCO and 0.90 recall@10 for SIFT.

theoretical analysis.

Discussion on quantization and comparing against Tiptoe. We noticed that the original Tiptoe implementation uses 4-bit quantization for the vectors to further trade off the search quality for the latency. In our evaluation, we use the full 32-bit precision for all the algorithms. Thus, the quality of the clustering-based algorithm should be considered an upper bound for the actual Tiptoe algorithm. For the latency, the "Linear Algorithm" baseline should be a good approximation of the actual Tiptoe algorithm. As an example, if we extrapolate the reported numbers of Tiptoe's algorithm based on the Table 6 and Table 7 in the original paper [HDCG$^+$23] to the MS-MARCO dataset, the latency will be around 0.27 seconds. [9]

Alternative Implementations. As alternatives, one can indeed use other PIR schemes to implement our graph-based ANN algorithm. As an example, using another recent single-server PIR scheme, SimplePIR [HHCG$^+$22] (used also in Tiptoe [HDCG$^+$23]), comes with a tradeoff. In

[9]The authors of Tiptoe reported the total core-second is 145 seconds for their experiment on a dataset with 360 million 192 dimensional vectors. The full Tiptoe system contains three parts: "preprocessing", "ranking" and "URL" where the "ranking" part corresponds to our ANN search experiment. According to Table 7 in the paper, it should take about $(1.9/(6.5 + 1.9 + 0.6)) = 21\%$ of the cost. Thus, an estimation of the time on a single-core for the 3.2M size MS-MARCO dataset will be $145 \times 21\% \times \frac{3.2\times10^6}{360\times10^6} \approx 0.27$s, which is close to our simulated "Linear Algorithm" latency (0.25 seconds).

the 10M SIFT dataset experiment, we can save the offline communication cost from roughly 6GB to around 300MB, but this would increase the online latency from 1.5s to roughly 90s according to our estimation as follows. We tested the throughput of the SimplePIR scheme on our server, which is around 10 GB/s. Then, we are making in total 25 rounds of graph explorations where each round having 96 parallel queries. The batching technique allows each individual query to be made in a sub-database of 16 times smaller than the whole database (around 6GB). Therefore, the total estimated time would be $25 \times 96 \times \frac{\text{6GB}}{16 \times \text{10GB/s}} \approx 90\text{s}$.

4.6 Theoretical Implications

We focused on the practical aspects of our scheme in the main body. Although graph-based ANN search has been empirically shown to be successful in practice, the theoretical understanding of the graph-based ANN search is still very limited. Here, we provide some theoretical insights into the graph-based private ANN search given the existing theoretical results.

Definition 4.6.1 ((c, r)-Approximate Nearest Neighbor)**.** Given n vectors $\mathsf{DB} = \{v_1, v_2, \ldots, v_n\} \in \mathbb{R}^{d \times n}$ and a distance function $\Delta(\cdot, \cdot)$, we say that an algorithm is a (c, r)-approximate nearest neighbor search algorithm if for any vector $q \in \mathbb{R}^d$ such that the true minimal distance between q and any vector in DB is at most r, the algorithm outputs an index i such that $\Delta(v_i, q) \leq c \cdot r$ with at least constant probability.

For the low-dimension case where $d = \Theta(\log n)$, we refer the reader to Prokhorenkova and Shekhovtsov [PS20] for the detailed discussion. Here, we will focus on the high-dimensional regime (also known as the sparse data case) such that the dimension $d = \omega(\log n)$, which is the most common setting for ANN search (recall that most embedding spaces are at least 100-dimensional).

Two notable theoretical results can be leveraged to analyze the graph-based ANN search. Laarhoven [Laa18] proves the following theorem:

Theorem 4.6.2 (Laarhoven [Laa18])**.** *Consider a database contains n independently random vectors in the unit sphere in $\mathbb{S}^{d-1}$ and the distance is measured by Euclidean distance. For $c > 1$, there exists a (c, r)-graph-based ANN search algorithm with the $O(n^{1+\rho+o(1)})$ space complexity and $O(n^{\rho+o(1)})$ query time complexity while taking only $O(1)$ hops in the graph, where*

$$\rho \geq \frac{c^4}{2c^4 - 2c^2 + 1}.$$

We call this setting the *average case setting*. Intuitively, we can think about n^ρ in the above theorem roughly denoting the average degree of the graph. For example, if we aim to have a 2-approximation ANN, then $\rho \approx 0.64$. We see that with the approximation factor c getting worse, $\lim_{c \to \infty} \rho(c) = \frac{1}{2}$.

On the other hand, Diwan et al. [DGM+24] studied the problem of building a navigable graph. On a high level, a navigable graph provides the guarantee for "in-distribution" queries for ANN search, that is, when the query vector will be exactly some vector in the database. They showed the following theorem:

Theorem 4.6.3 ([DGM+24]). *For any n vectors in $\mathbb{R}^d$, it is possible to build a navigable graph with average degree of at most $2\sqrt{n \ln n}$. Moreover, the "greedy-routing" strategy always succeeds in finding the correct vector in the database with at most 2 hops.*

The above two theorems provide the following theoretical insights: for the high-dimensional regime, to achieve a strong guarantee on the ANN search, the average degree of the graph will be roughly $n^{\frac{1}{2}+o(1)}$, and the query hop number will only be a constant. Therefore, if we use the same PIR technique to make the graph-based ANN search private, the underlying graph-information database is of size $n^{\frac{3}{2}+o(1)}$, where the total number of entries is n and each entry is of size $n^{1/2+o(1)}$. It is interesting to see that this is not a typical setting of the PIR literature, because the entry size is much larger than a constant. If we plug in the best parameters of the client-preprocessing PIR [ZPSZ24, NGH24] into the Laarhoven's algorithm, we will get the following result:

Theorem 4.6.4. *Assume the existence of one-way functions. For $c > 1$, there exists a (c, r)-graph-based private ANN search algorithm for Euclidean distance in the average case setting (random vectors on the unit sphere $\mathbb{S}^{d-1}$) with the following properties:*

- *Preprocessing cost: $\tilde{O}(n^{1+\rho+o(1)})$ communication and computation cost;*
- *Query cost:*
  - *$\tilde{O}(n^{1/2+\rho+o(1)})$ computation cost;*
  - *$\tilde{O}(n^{\rho+o(1)})$ communication cost;*
- *Storage cost:*
  - *Client: $\tilde{O}(n^{1/2})$;*
  - *Server: $\tilde{O}(n^{1+\rho+o(1)})$.*

Here,

$$\rho \geq \frac{c^4}{2c^4 - 2c^2 + 1}.$$

Specifically, we need to use the new result in Nguyen et al. [NGH24] for optimizing the client storage. Notice that based on the existing client-specific preprocessing PIR lower bounds [CK20, PY22, LMWY20], to privately access a data structure of size N, the product between the client storage S and the online time T satisfies $S \times T = \Omega(N)$. In this sense, the above theoretical result is nearly tight in terms of the client storage and online query time if we follow the graph-based ANN paradigm: the client storage is $\tilde{O}(n^{1/2})$ and the online query time is $\tilde{O}(n^{1/2+\rho+o(1)})$, while the product matches the data structure size of $n^{1+\rho+o(1)}$.

Gap between Theory and Practice. Notably, the above theoretical results on graph-based ANN algorithms are based on the analysis of "high-degree" graphs, where the average degree is $\Omega(\sqrt{n})$, and the query hop number is a small constant. In practice, we see a different combination: popular graph-based ANN algorithm implementations usually pick a much smaller average degree, e.g., 32 or 64, while making the query hop number larger (e.g. scaling with the logarithm of the database size). It remains an interesting open question whether there exists a strong theoretical result for small-degree graphs with a search process invoking a large number of hops.

4.7 Conclusion

We present PACMANN, a new private ANN search scheme that allows clients to perform nearest neighbor search queries over hundreds of millions of vectors while preserving the queries' privacy. PACMANN achieves significantly better search quality compared to the state-of-the-art private ANN search schemes and has lower latency in large-scale datasets. PACMANN could potentially be applied to a wide range of information retrieval applications including conventional search and retrieval-augmented generation.

Limitations. PACMANN has several limitations, which may present opportunities for future work. First, it inherits the drawbacks of the single-server preprocessing PIR schemes and requires the client to download the whole indexing structure offline in a streaming manner. Therefore, PACMANN will not be suitable for network-constrained scenarios. PACMANN also does not naturally support dynamic updates to the database, and handling dynamic updates has been a challenging open problem in the preprocessing PIR literature. Finally, PACMANN is designed under the assumption that the database is public, so we do not consider server-side privacy. We leave these questions as interesting future directions.

Part III

Conclusion

Conclusion

Summary of the Thesis. In this thesis, we introduce two novel PIR constructions, Piano and QuarterPIR, which achieve sublinear computation and communication costs following preprocessing. These constructions transform the practical PIR landscape by delivering near real-time response times for databases containing billions of entries, while maintaining modest communication and storage requirements. We demonstrate the practical utility of our PIR constructions through a key application—private information searching—where we develop Pacmann, a new private approximate nearest neighbor search algorithm. Pacmann emerges as the first private search algorithm to simultaneously deliver high search quality and fast response times for databases containing hundreds of millions of entries. The fundamental concept underlying Pacmann – the separation of client computation from server storage using sublinear PIR schemes – establishes a new paradigm for designing practical privacy-preserving algorithms.

On-going Work. As of writing this thesis, the author is working on the following directions:

- *Reducing the Online Communication Cost of Preprocessing PIR with OWF.* Our QuarterPIR work shows that the online communication cost of preprocessing PIR with OWF can be reduced to $\tilde{O}(n^{1/4})$. On the other hand, we also show that the online communication cost can be reduced to polylogarithmic with stronger assumptions such as LWE. The author and his collaborators have identified a potential new direction that may help close this gap with the help of coding theory and a recursion technique for privately programmable pseudorandom function.
- *Optimal Sublinear PIR with Information-Theoretic Security.* Piano and QuarterPIR assume a minimum cryptographic assumption that one-way functions exist. Is it possible to achieve similar results without any cryptographic assumption? Two recent results [ISW24, SWZ25] show new information-theoretic PIR constructions that achieve sublinear query computation complexity, but with sub-optimal storage overhead. The author and his collaborators are working on a new practical PIR construction that achieves the optimal storage-computation tradeoff with information-theoretic security.

Future Directions. We present a brief, non-exhaustive list of interesting future directions in the context of PIR and private search:

- *Handling Dynamic Databases.* Although preprocessing PIR constructions achieve a significant advantage in efficient query processing, a key limitation is that they are usually designed for static databases. For many real-world applications, the database is dynamic and frequently updated, requiring the PIR system to support efficient updates. The existing techniques for supporting updates in preprocessing PIR [ZPSZ24, HPPY24] remain inefficient in practice, and there is a strong need for more efficient solutions.
- *Hybrid Preprocessing PIR.* Throughout the thesis, we have focused on the client-specific preprocessing PIR paradigm, where the client ultimately stores the preprocessing result. On the other hand, researchers have shown significant progress in the server-specific preprocessing PIR paradigm [BIM00, LMW23, LLFP24] that allows the server to store an encoded version of the database. It remains an open question whether we can combine the advantages of both paradigms to achieve a more efficient PIR system.
- *Access Control in PIR.* The existing PIR constructions focus on hiding the query from the

server, but do not provide any access control mechanism. In practice, access control is nearly as important as privacy, as it is vital to the security and integrity of the information retrieval service. It is an interesting direction to explore how to integrate access control mechanisms into PIR systems without compromising the clients' privacy and the servers' efficiency.

- *Integrating with Secure Hardware Technology.* Several secure hardware technologies, such as Intel SGX [CD16], ARM TrustZone [PS19] and Nvidia Confidential Computing [DGK+23], have been developed to provide secure execution environments for privacy-preserving applications. Given their practicality and efficiency, it would be interesting to explore the integration between PIR and secure hardware technologies to further improve the performance and provide more advanced features for PIR systems.

Final Remarks

As we conclude this research journey, I find myself reflecting on what privacy research truly means today. Our generation has lived through the digital revolution and has already lost much of its privacy. The digital trails we've left – on social media, shopping sites, location apps, and countless other services – have permanently exposed our personal information. This raises an important question: Why study privacy technologies when the damage has already been done?

The answer is simple yet powerful: while we cannot take back the privacy we've lost, we can build better protections for the future. Throughout the Internet's development, society consistently chose convenience and growth over privacy protection. Companies built digital empires by exploiting our personal data, while consumers eagerly adopted these services without understanding the privacy implications. I believe that the fundamental drive behind this repetitive pattern is the widespread assumption that privacy and efficiency are always at odds.

Our research on Private Information Retrieval (PIR) and privacy-preserving information search shows this trade-off isn't necessary: even complex information systems can have strong privacy protection without becoming significantly slower or less useful, by incorporating carefully designed privacy-preserving technologies.

Beyond PIR, we see similar progress in other privacy technologies like Zero-Knowledge Proofs (ZKP), Homomorphic Encryption (HE), and Secure Multi-Party Computation (SMPC). Once thought too impractical for real use, these approaches are becoming increasingly efficient. Together, these advances show that privacy, efficiency, and even business goals can work together in well-designed systems.

The incoming era of artificial intelligence poses a bigger challenge to privacy: generative models have created an even greater hunger for nearly all available information. We must not repeat our past mistakes from the Internet era. AI development needs strong ethical guidelines that prioritize privacy and security, ensuring the technology serves people rather than exploits them.

Building a privacy-native information system requires a long-term commitment across researchers, developers, and policy makers. We hope our research inspires both academics and industry to keep developing privacy technologies – not just for ourselves, but for everyone who will inherit the digital world we're shaping today.

Bibliography

[A+48] UN General Assembly et al. Universal declaration of human rights. *UN General Assembly*, 302(2):14–25, 1948. 1

[ABF20] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems*, 87:101374, 2020. 4.5.1

[ABG+24] Hilal Asi, Fabian Boemer, Nicholas Genise, Muhammad Haris Mughees, Tabitha Ogilvie, Rehan Rishi, Guy N Rothblum, Kunal Talwar, Karl Tarbe, Ruiyu Zhu, and Marco Zuliani. Scalable Private Search with Wally. *arXiv preprint arXiv:2406.06761*, 2024. 2, 1, 1.1, 4.1, 4.1.1, 2, 4.1.2

[ACLS18] Sebastian Angel, Hao Chen, Kim Laine, and Srinath T. V. Setty. PIR with Compressed Queries and Amortized Query Processing. In *S&P*, 2018. 1, 1, 2.1, 2.6

[AM93] Sunil Arya and David M Mount. Approximate nearest neighbor queries in fixed dimensions. In *SODA*, volume 93, pages 271–280. Citeseer, 1993. 4.3.2

[aol06] AOL search data release reveals a great deal. `http://www.usatoday.com/tech/columnist/andrewkantor/2006-08-17-aol-data_x.htm`, 2006. 1

[AS16] Sebastian Angel and Srinath TV Setty. Unobservable Communication over Fully Untrusted Infrastructure. In *OSDI*, volume 16, pages 551–569, 2016. 2.6

[AYA+21] Ishtiyaque Ahmad, Yuntian Yang, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta. Addra: Metadata-private voice communication over fully untrusted infrastructure. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI2021, July 14-16, 2021*, 2021. 2.3, 2.6

[BDKP22] Shany Ben-David, Yael Tauman Kalai, and Omer Paneth. Verifiable private information retrieval. In *Theory of Cryptography Conference*, pages 3–32. Springer, 2022. 3

[BFG03] Richard Beigel, Lance Fortnow, and William I. Gasarch. A Nearly Tight Bound for Private Information Retrieval Protocols. *Electronic Colloquium on Computational Complexity (ECCC)*, 2003. 1

[BFK+09] Mauro Barni, Pierluigi Failla, Vladimir Kolesnikov, Riccardo Lazzeretti, Ahmad-Reza Sadeghi, and Thomas Schneider. Secure evaluation of private linear

branching programs with medical applications. In *Computer Security–ESORICS 2009: 14th European Symposium on Research in Computer Security, Saint-Malo, France, September 21-23, 2009. Proceedings 14*, pages 424–439. Springer, 2009. 4.4.1

[BGI16] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function Secret Sharing: Improvements and Extensions. In *CCS*, 2016. 2.6

[BIM00] Amos Beimel, Yuval Ishai, and Tal Malkin. Reducing the Servers Computation in Private Information Retrieval: PIR with Preprocessing. In *CRYPTO*, pages 55–73, 2000. 1, 1.1, 2.1, 2.6, 3.1, 3.1, 2, III

[bin20] Data leak: Unsecured server exposed Bing Mobile App Data. `https://www.wizcase.com/blog/bing-leak-research/`, 2020. 1

[BIPW17] Elette Boyle, Yuval Ishai, Rafael Pass, and Mary Wootters. Can We Access a Database Both Locally and Privately? In *TCC*, 2017. 2.6, 2.3

[BKM17] Dan Boneh, Sam Kim, and Hart William Montgomery. Private Puncturable PRFs from Standard Lattice Assumptions. In *EUROCRYPT*, pages 415–445, 2017. 1, 1.1, 2.1.1, 3.1.1

[BLW17] Dan Boneh, Kevin Lewi, and David J. Wu. Constraining Pseudorandom Functions Privately. In *PKC*, 2017. 2.1, 3.1.1

[BMW24] Alexander Burton, Samir Jordan Menon, and David J. Wu. RESPIRE: High-Rate PIR for Databases with Small Records. In *ACM CCS*, 2024. 1, 1.1

[BS80] Jon Louis Bentley and James B. Saxe. Decomposable Searching Problems I: Static-to-Dynamic Transformation. *J. Algorithms*, 1(4):301–358, 1980. 2.5.3

[BTVW17] Zvika Brakerski, Rotem Tsabary, Vinod Vaikuntanathan, and Hoeteck Wee. Private Constrained PRFs (and More) from LWE. In *TCC*, 2017. 1, 1.1, 2.1.1, 3.1.1

[Byg14] Lee Andrew Bygrave. *Data privacy law: an international perspective*. Oxford University Press, 2014. 1

[CC17] Ran Canetti and Yilei Chen. Constraint-Hiding Constrained PRFs for NC^1 from LWE. In *EUROCRYPT*, pages 446–476, 2017. 1, 1.1, 2.1.1, 3.1.1

[CCJ+23] Patrick Chen, Wei-Cheng Chang, Jyun-Yu Jiang, Hsiang-Fu Yu, Inderjit Dhillon, and Cho-Jui Hsieh. Finger: Fast inference for graph-based approximate nearest neighbor search. In *Proceedings of the ACM Web Conference 2023*, pages 3225–3235, 2023. 4.3.1

[CD+15] Ronald Cramer, Ivan Bjerre Damgård, et al. *Secure multiparty computation*. Cambridge University Press, 2015. 4.1.1, 4.4.1

[CD16] Victor Costan and Srinivas Devadas. Intel SGX explained. *Cryptology ePrint Archive*, 2016. III

[CD24] Sofía Celi and Alex Davidson. Call me by my name: Simple, practical private information retrieval for keyword queries. In *Proceedings of the 2024 on ACM*

SIGSAC Conference on Computer and Communications Security, pages 4107–4121, 2024. 2, 4.1.2

[CG97] Benny Chor and Niv Gilboa. Computationally Private Information Retrieval. In *STOC*, 1997. 1

[CGG+24] Arka Rai Choudhuri, Sanjam Garg, Aarushi Goel, Sruthi Sekar, and Rohit Sinha. Sublonk: Sublinear prover plonk. *Proceedings on Privacy Enhancing Technologies*, 2024. 4.4.1

[CGKS95] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private Information Retrieval. In *FOCS*, 1995. 1, 2.6, 4.1, 4.1.1, 4.4.1

[CGN97] Benny Chor, Niv Gilboa, and Moni Naor. Private information retrieval by keywords. 1997. 2, 4.1.2

[Cha04] Yan-Cheng Chang. Single Database Private Information Retrieval with Logarithmic Communication. In *ACISP*, 2004. 1, 2.6, 2.3

[CHK22] Henry Corrigan-Gibbs, Alexandra Henzinger, and Dmitry Kogan. Single-Server Private Information Retrieval with Sublinear Amortized Time. In *Eurocrypt*, 2022. 1, 1.1, 1.1, 2.1, 2.1.1, 2.3, 2.6, 3.1, 3.1, 3.1

[CHR17] Ran Canetti, Justin Holmgren, and Silas Richelson. Towards Doubly Efficient Private Information Retrieval. In *TCC*, 2017. 1, 2.6, 2.3

[CK20] Henry Corrigan-Gibbs and Dmitry Kogan. Private Information Retrieval with Sublinear Online Time. In *EUROCRYPT*, 2020. 1, 1.1, 1.1, 2.1, 2.1.1, 2.3, 2.6, 3.1, 3.1, 3.1, 4.1.1, 4.6

[CKGS98] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private Information Retrieval. *J. ACM*, 45(6):965–981, November 1998. 2.6

[CMS99] Christian Cachin, Silvio Micali, and Markus Stadler. Computationally private information retrieval with polylogarithmic communication. In *EUROCRYPT*, pages 402–414, 1999. (document), 1, 1.1, 2.6, 2.3, 3.1

[Con22] Graeme Connell. Technology Deep Dive: Building a Faster ORAM Layer for Enclaves. `https://signal.org/blog/building-faster-oram/`, 2022. 1

[DCMO00] Giovanni Di Crescenzo, Tal Malkin, and Rafail Ostrovsky. Single Database Private Information Retrieval Implies Oblivious Transfer. In *EUROCRYPT*, 2000. 3.1

[DdSGOTV22] Cyprien Delpech de Saint Guilhem, Emmanuela Orsini, Titouan Tanguy, and Michiel Verbauwhede. Efficient proof of RAM programs from any public-coin zero-knowledge system. In *International Conference on Security and Cryptography for Networks*, pages 615–638. Springer, 2022. 4.4.1

[DGI+19] Nico Döttling, Sanjam Garg, Yuval Ishai, Giulio Malavolta, Tamer Mour, and Rafail Ostrovsky. Trapdoor Hash Functions and Their Applications. In *Advances in Cryptology – CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part III*, 2019.

1, 1.1, 3.1

[DGK+23] Gobikrishna Dhanuskodi, Sudeshna Guha, Vidhya Krishnan, Aruna Manjunatha, Michael O'Connor, Rob Nertney, and Phil Rogers. Creating the First Confidential GPUs: The team at NVIDIA brings confidentiality and integrity to user code and data for accelerated computing. *Queue*, 21(4):68–93, 2023. III

[DGM+24] Haya Diwan, Jinrui Gou, Cameron Musco, Christopher Musco, and Torsten Suel. Navigable Graphs for High-Dimensional Nearest Neighbor Search: Constructions and Limits. *arXiv preprint arXiv:2405.18680*, 2024. 4.6, 4.6.3

[DPC22] Alex Davidson, Gonçalo Pestana, and Sofía Celi. FrodoPIR: Simple, Scalable, Single-Server Private Information Retrieval. Cryptology ePrint Archive, Paper 2022/981, 2022. `https://eprint.iacr.org/2022/981`. 1, 1, 2.1, 2.3, 2.6

[DRRT18] Daniel Demmler, Peter Rindal, Mike Rosulek, and Ni Trieu. PIR-PSI: Scaling Private Contact Discovery. *Proc. Priv. Enhancing Technol.*, 2018(4):159–178, 2018. 1

[Fea] Nick Feamster. Oblivious DNS Deployed by Cloudflare and Apple. `https://medium.com/noise-lab/oblivious-dns-deployed-by-cloudflare-and-apple-1522ccf53cab`. 1

[Gas04] William I. Gasarch. A Survey on Private Information Retrieval. *Bulletin of the EATCS*, 82:72–107, 2004. 1

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 169–178, 2009. 1, 4.1.1, 4.4.1

[GHAHJ22] Aarushi Goel, Mathias Hall-Andersen, Aditya Hegde, and Abhishek Jain. Secure multiparty computation with free branching. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 397–426. Springer, 2022. 4.4.1

[GI14] Niv Gilboa and Yuval Ishai. Distributed Point Functions and Their Applications. In *Advances in Cryptology – EUROCRYPT 2014*, 2014. 2.6

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 1996. 2.5.3

[Gol87] O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *STOC*, 1987. 2.5.3

[GR05] Craig Gentry and Zulfikar Ramzan. Single-Database Private Information Retrieval with Constant Communication Rate. In *ICALP*, 2005. 1, 2.6, 2.3

[GZS24] Ashrujit Ghoshal, Mingxun Zhou, and Elaine Shi. Efficient pre-processing pir without public-key cryptography. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 210–240. Springer, 2024. (document), 1, 1.1, 1.1, 1.1, 1.1, 1.2, 4.1, 4.4.1

[HDCG+23] Alexandra Henzinger, Emma Dauterman, Henry Corrigan-Gibbs, , and Nickolai Zeldovich. Private Web Search with Tiptoe. In *29th ACM Symposium on Operating Systems Principles (SOSP)*, Koblenz, Germany, October 2023. (document), 2, 1, 1.1, 4.1, 4.1.1, 2, 4.2, 4.1.2, 4.4.1, 4.5, 4.5.1, 7, 8, 4.5.4

[HFM+24] Matthew M Hong, David Froelicher, Ricky Magner, Victoria Popic, Bonnie Berger, and Hyunghoon Cho. Secure discovery of genetic relatives across large-scale and distributed genomic data sets. *Genome Research*, 34(9):1312–1323, 2024. 1

[HHCG+22] Alexandra Henzinger, Matthew M. Hong, Henry Corrigan-Gibbs, Sarah Meiklejohn, and Vinod Vaikuntanathan. One Server for the Price of Two: Simple and Fast Single-Server Private Information Retrieval. Cryptology ePrint Archive, Paper 2022/949, 2022. `https://eprint.iacr.org/2022/949`. 1, 1, 1, 1.1, 2.1, 2.1.1, 1, 2.4, 2.4.1, 2.4.2, 2.3, 2.6, 3.1, 4.5.4

[HKP21] David Heath, Vladimir Kolesnikov, and Stanislav Peceny. Masked Triples: Amortizing Multiplication Triples Across Conditionals. In *IACR International Conference on Public-Key Cryptography*, pages 319–348. Springer, 2021. 4.4.1

[HMR12] Viet Tung Hoang, Ben Morris, and Phillip Rogaway. An enciphering scheme based on a card shuffle. In *Advances in Cryptology–CRYPTO 2012: 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, pages 1–13. Springer, 2012. 3.3.1, 3.4.1

[HPPY24] Alexander Hoover, Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. Plinko: Single-Server PIR with Efficient Updates via Invertible PRFs. *Eurocrypt*, 2024. 1, III

[HSS+20] Jui-Ting Huang, Ashish Sharma, Shuying Sun, Li Xia, David Zhang, Philip Pronin, Janani Padmanabhan, Giuseppe Ottaviano, and Linjun Yang. Embedding-based retrieval in facebook search. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2553–2561, 2020. 1.1, 4.1

[IKOS04] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Batch Codes and Their Applications. In *STOC*, 2004. 2.6

[IM98] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. STOC '98. Association for Computing Machinery, 1998. 4.1.1

[IM18] Masajiro Iwasaki and Daisuke Miyazaki. Optimization of indexing based on k-nearest neighbor graph for proximity search in high-dimensional data. *arXiv preprint arXiv:1810.07355*, 2018. (document), 4.1.1, 1, 4.2, 2, 4.3.1, 4.3.2, 4.5, 4.5.1

[ISW24] Yuval Ishai, Elaine Shi, and Daniel Wichs. PIR with client-side preprocessing: information-theoretic constructions and lower bounds. In *Annual International Cryptology Conference*, pages 148–182. Springer, 2024. 1.1, III

[JSDS+19] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. Diskann: Fast accurate billion-point nearest neighbor search on a single node. *Advances in Neural Information Processing Systems*, 32, 2019. 4.1.1, 1, 4.3.1, 4.4.2

[JTDA11] Hervé Jégou, Romain Tavenard, Matthijs Douze, and Laurent Amsaleg. Searching in one billion vectors: re-rank with source coding. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 861–864. IEEE, 2011. 4.5.1

[JTJS14] Rob Jansen, Florian Tschorsch, Aaron Johnson, and Björn Scheuermann. The Sniper Attack: Anonymously Deanonymizing and Disabling the Tor Network. In *NDSS*, 2014. 1

[KCG21] Dmitry Kogan and Henry Corrigan-Gibbs. Private Blocklist Lookups with Checklist. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 875–892. USENIX Association, August 2021. 1, 1.1, 2.1, 2.5.3, 2.6, 3.1, 4.4.1

[KK04] Shashank Khanvilkar and Ashfaq Khokhar. Virtual private networks: an overview with performance evaluation. *IEEE Communications Magazine*, 42(10):146–154, 2004. 1

[KO97] E. Kushilevitz and R. Ostrovsky. Replication is not needed: single database, computationally-private information retrieval. In *FOCS*, 1997. 1

[KU11] Kiran S Kedlaya and Christopher Umans. Fast polynomial factorization and modular composition. *SIAM Journal on Computing*, 40(6):1767–1802, 2011. 2.1

[KW21] Sam Kim and David J. Wu. Watermarking Cryptographic Functionalities from Standard Lattice Assumptions. *J. Cryptol.*, 34(3), jul 2021. 2.1

[Laa18] Thijs Laarhoven. Graph-Based Time-Space Trade-Offs for Approximate Near Neighbors. In *34th International Symposium on Computational Geometry (SoCG 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018. 4.6, 4.6.2

[Lip09] Helger Lipmaa. First CPIR protocol with data-dependent computation. In *ICISC*, 2009. 1

[LKLJ18] Malte Ludewig, Iman Kamehkhosh, Nick Landia, and Dietmar Jannach. Effective nearest-neighbor music recommendations. In *Proceedings of the ACM Recommender Systems Challenge 2018*, pages 1–6. 2018. 1.1, 4.1

[LLFP24] Arthur Lazzaretti, Zeyu Liu, Ben Fisch, and Charalampos Papamanthou. Multi-server doubly efficient PIR. *Cryptology ePrint Archive*, 2024. 1, III

[LLJ+21] Sen Li, Fuyu Lv, Taiwei Jin, Guli Lin, Keping Yang, Xiaoyi Zeng, Xiao-Ming Wu, and Qianli Ma. Embedding-based product retrieval in taobao search. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pages 3181–3189, 2021. 1.1, 4.1

[LLWR22] Jian Liu, Jingyu Li, Di Wu, and Kui Ren. PIRANA: Faster (Multi-query) PIR via Constant-weight Codes. Cryptology ePrint Archive, Paper 2022/1401, 2022. `https://eprint.iacr.org/2022/1401`. 2.6

[LMW23] Wei-Kai Lin, Ethan Mook, and Daniel Wichs. Doubly Efficient Private Information Retrieval and Fully Homomorphic RAM Computation from Ring LWE. In *STOC*, 2023. (document), 1, 1.1, 2.1, 2.6, 2.3, 3.1, III

[LMWY20] Kasper Green Larsen, Tal Malkin, Omri Weinstein, and Kevin Yeo. Lower bounds for oblivious near-neighbor search. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1116–1134. SIAM, 2020. 4.6

[LP23a] Arthur Lazzaretti and Charalampos Papamanthou. Near-optimal private information retrieval with preprocessing. In *Theory of Cryptography Conference*, pages 406–435. Springer, 2023. 1, 1, 1.1, 1.1, 2.1, 2.1.1, 2.3, 2.6, 2.7, 3.1, 3.1, 3.1, 3.1.1, 3.5.1

[LP23b] Arthur Lazzaretti and Charalampos Papamanthou. TreePIR: Sublinear-Time and Polylog-Bandwidth Private Information Retrieval from DDH. In *CRYPTO*, 2023. 1, 1.1, 2.1, 2.6, 3.1, 3.1, 3.1.1, 3.7, 3.7.1, 3.7.1, 3.7.2, 4.4.1

[LPP+20] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020. 4.1

[LXY+19] Xirong Li, Chaoxi Xu, Gang Yang, Zhineng Chen, and Jianfeng Dong. W2VV++: Fully Deep Learning for Ad-hoc Video Search. In *Proceedings of the 27th ACM International Conference on Multimedia*, MM '19. Association for Computing Machinery, 2019. 4.1

[MBFK16] Carlos Aguilar Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. XPIR: Private information retrieval for everyone. *Proceedings on Privacy Enhancing Technologies*, pages 155–174, 2016. 2.3, 2.6

[MBG+08] Damon McCoy, Kevin Bauer, Dirk Grunwald, Tadayoshi Kohno, and Douglas Sicker. Shining light in dark places: Understanding the Tor network. In *Privacy Enhancing Technologies: 8th International Symposium, PETS 2008 Leuven, Belgium, July 23-25, 2008 Proceedings 8*, pages 63–76. Springer, 2008. 1

[MCG+08] Carlos Aguilar Melchor, Benoit Crespin, Philippe Gaborit, Vincent Jolivet, and Pierre Rousseau. High-Speed Private Information Retrieval Computation on GPU. In *Proceedings of the 2008 Second International Conference on Emerging Security Information, Systems and Technologies*, SECURWARE '08, pages 263–272, Washington, DC, USA, 2008. IEEE Computer Society. 1

[MCR21] Muhammad Haris Mughees, Hao Chen, and Ling Ren. OnionPIR: Response Efficient Single-Server PIR. In *CCS*. Association for Computing Machinery, 2021. 2.3, 2.6

[MG07] Carlos Aguilar Melchor and Philippe Gaborit. A Lattice-Based Computationally-Efficient Private Information Retrieval Protocol. *IACR Cryptology ePrint Archive*, 2007:446, 2007. 1

[MPLK12] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. Scalable distributed algorithm for approximate nearest neighbor search problem in high dimensional general metric spaces. In *Similarity Search and Applications: 5th International Conference, SISAP 2012, Toronto, ON, Canada, August 9-10, 2012. Proceedings 5*, pages 132–147. Springer, 2012. 4.3.1

[MR22] Muhammad Haris Mughees and Ling Ren. Vectorized Batch Private Information Retrieval. Cryptology ePrint Archive, Paper 2022/1262, 2022. `https://eprint.iacr.org/2022/1262`. 1, 1, 2.1, 2.6

[MW22] Samir Jordan Menon and David J. Wu. SPIRAL: Fast, High-Rate Single-Server PIR via FHE Composition. In *IEEE S&P*, 2022. 1, 1, 1, 1.1, 2.1, 2.1.1, 2.3, 2.6, 3.1

[MY18] Yu A Malkov and Dmitry A Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 42(4):824–836, 2018. 4.1.1, 1, 4.1.2, 4.3.1

[MZRA22] Yiping Ma, Ke Zhong, Tal Rabin, and Sebastian Angel. Incremental {Offline/Online}{PIR}. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1741–1758, 2022. 2.6

[NGH24] Hoang-Dung Nguyen, Jorge Guajardo, and Thang Hoang. Client-Efficient Online-Offline Private Information Retrieval. Cryptology ePrint Archive, Paper 2024/719, 2024. 1, 4.6, 4.6

[NRS+16] Tri Nguyen, Mir Rosenberg, Xia Song, Jianfeng Gao, Saurabh Tiwary, Rangan Majumder, and Li Deng. Ms marco: A human-generated machine reading comprehension dataset. 2016. 4.5.1

[OG11] Femi G. Olumofin and Ian Goldberg. Revisiting the Computational Practicality of Private Information Retrieval. In *Financial Cryptography*, pages 158–172, 2011. 1

[OPPW23] Hiroki Okada, Rachel Player, Simon Pohmann, and Christian Weinert. Towards practical doubly-efficient private information retrieval. *Cryptology ePrint Archive*, 2023. 1

[OS07] Rafail Ostrovsky and William E. Skeith, III. A survey of single-database private information retrieval: techniques and applications. In *PKC*, pages 393–411, 2007. 1

[Per24] Carlos Perez. Scale Semaphore Project Proposal. `https://hackmd.io/@brech1/scale-semaphore-proposal`, 2024. 1

[PPY18] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. Private Stateful Information Retrieval. In *CCS*, 2018. 2.3, 2.6

[PR04] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004. 3.3.1, 3.4.1

[PS18] Chris Peikert and Sina Shiehian. Privately Constraining and Programming PRFs,

the LWE Way. In *Public Key Cryptography (2)*, volume 10770 of *Lecture Notes in Computer Science*, pages 675–701. Springer, 2018. 2.1

[PS19] Sandro Pinto and Nuno Santos. Demystifying ARM trustzone: A comprehensive survey. *ACM computing surveys (CSUR)*, 51(6):1–36, 2019. III

[PS20] Liudmila Prokhorenkova and Aleksandr Shekhovtsov. Graph-based nearest neighbor search: From practice to theory. In *International Conference on Machine Learning*, pages 7803–7813. PMLR, 2020. 4.6

[PSY23] Sarvar Patel, Joon Young Seo, and Kevin Yeo. {Don't} be Dense: Efficient Keyword {PIR} for Sparse Databases. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3853–3870, 2023. 2, 4.1.2

[PY22] Giuseppe Persiano and Kevin Yeo. Limits of Preprocessing for Single-Server PIR. In *SODA*, pages 2522–2548. SIAM, 2022. 3.1, 4.6

[RG19] Nils Reimers and Iryna Gurevych. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, 2019. 1.1, 4.5.1

[RKH+21] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In *International conference on machine learning*, pages 8748–8763. PMLR, 2021. 1.1

[RMS24] Ling Ren, Muhammad Haris Mughees, and I Sun. Simple and Practical Amortized Sublinear Private Information Retrieval using Dummy Subsets. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024. 1, 1.1, 2.3.4, 3.1, 3.1, 3.1, 4.4.1

[RY13] Thomas Ristenpart and Scott Yilek. The mix-and-cut shuffle: small-domain encryption secure against N queries. In *Advances in Cryptology–CRYPTO 2013: 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, pages 392–409. Springer, 2013. 3.3.1, 3.4.1

[SACM21] Elaine Shi, Waqar Aqeel, Balakrishnan Chandrasekaran, and Bruce Maggs. Puncturable Pseudorandom Sets and Private Information Retrieval with Near-Optimal Online Bandwidth and Time. In *CRYPTO*, 2021. 1, 1.1, 2.1, 2.3.4, 2.6, 3.1, 3.1.1

[SC07] Radu Sion and Bogdan Carbunar. On the Computational Practicality of Private Information Retrieval. In *Network and Distributed Systems Security Symposium (NDSS)*, 2007. 1

[SPS14] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. Practical Dynamic Searchable Symmetric Encryption with Small Leakage. In *Network and Distributed System Security Symposium (NDSS)*, 2014. 2.5.3

[SSGN17] Jessica Su, Ansh Shukla, Sharad Goel, and Arvind Narayanan. De-anonymizing web browsing data with social networks. In *Proceedings of the 26th international conference on world wide web*, pages 1261–1269, 2017. 1

[SSLD22] Sacha Servan-Schreiber, Simon Langowski, and Srinivas Devadas. Private approximate nearest neighbor search with sublinear communication. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 911–929. IEEE, 2022. 2, 4.1, 4.1.1, 2, 4.1.2, 4.4.1, 4.4.2

[SSP13] Elaine Shi, Emil Stefanov, and Charalampos Papamanthou. Practical Dynamic Proofs of Retrievability. In *ACM Conference on Computer and Communications Security (CCS)*, 2013. 2.5.3

[SvDS+18] Emil Stefanov, Marten van Dijk, Elaine Shi, T.-H. Hubert Chan, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An Extremely Simple Oblivious RAM Protocol. *J. ACM*, 65(4):18:1–18:26, 2018. 4.1.2

[SWZ25] Jaspal Singh, Yu Wei, and Vassilis Zikas. Information-Theoretic Multi-server Private Information Retrieval with Client Preprocessing. In *Theory of Cryptography Conference*, pages 423–450. Springer, 2025. III

[WLZ+23] Yinghao Wang, Xuanming Liu, Jiawen Zhang, Jian Liu, and Xiaohu Yang. Crust: Verifiable and Efficient Private Information Retrieval With Sublinear Online Time. *Cryptology ePrint Archive*, 2023. 1, 4.4.1

[WXYW21] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search. *arXiv preprint arXiv:2101.12631*, 2021. 4.3.1

[Yeo23] Kevin Yeo. Cuckoo hashing in cryptography: Optimal parameters, robustness and applications. *arXiv preprint arXiv:2306.11220*, 2023. 3.3.1, 3.4.1

[ZLTS23] Mingxun Zhou, Wei-Kai Lin, Yiannis Tselekounis, and Elaine Shi. Optimal Single-Server Private Information Retrieval. In *EUROCRYPT*, 2023. 1, 1, 1.1, 1.1, 2.1, 2.1.1, 2.3, 2.6, 2.7, 3.1, 3.1, 3.1, 3.1.1, 3.3.1, 3.4.1, 3.5.1

[ZPSZ24] Mingxun Zhou, Andrew Park, Elaine Shi, and Wenting Zheng. Piano: Extremely Simple, Single-Server PIR with Sublinear Server Computation. In *IEEE S& P*, 2024. (document), 1, 1.1, 1.1, 1.1, 1.2, 3.1, 3.1, 3.1, 3.3.1, 3.4.1, 3.5.1, 3.6, 4.1, 4.1.1, 1, 4.4.1, 4.5.2, 4.6, III

[ZPZP24] Jinhao Zhu, Liana Patel, Matei Zaharia, and Raluca Ada Popa. Compass: Encrypted Semantic Search with High Accuracy. Cryptology ePrint Archive, Paper 2024/1255, 2024. `https://eprint.iacr.org/2024/1255`. 4.1.2

[ZSF25] Mingxun Zhou, Elaine Shi, and Giulia Fanti. Pacmann: Efficient Private Approximate Nearest Neighbor Search. *ICLR*, 2025. 1, 1.1, 1.1, 1.2