The Thesis Committee for Sidaarth Sabhnani
certifies that this is the approved version of the following thesis:

From Lattices to Tensor Cores: Accelerating Communication-Efficient Private Information Retrieval

SUPERVISING COMMITTEE:

David Wu, Supervisor

Christopher Rossbach

From Lattices to Tensor Cores: Accelerating Communication-Efficient Private Information Retrieval

by

Sidaarth Sabhnani

Thesis

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science

The University of Texas at Austin

May 2026

Dedication

To Amrin, Sher, and Appa—and our life together.

Epigraph

Arise, awake and stop not till the goal is reached.

—S.V.

Acknowledgments

I would like to thank my advisor, David Wu, who introduced me to cryptography through an exceptional course and welcomed me to his research. His clarity of thought shaped both the theoretical foundations and the direction of this thesis.

I am grateful to Christopher Rossbach for serving on my committee, and for his influence across my time at UT—as my undergraduate honors program director, instructor, and mentor.

I would also like to thank Ahmed Gheith, for whom I had the privilege of serving as a teaching assistant and whose lectures instilled a genuine passion for computing. His guidance throughout my educational journey has been invaluable.

Finally, I am thankful to my family, especially my father Kailash for his unconditional support, and my friends for their encouragement throughout this process.

Abstract

From Lattices to Tensor Cores: Accelerating Communication-Efficient Private Information Retrieval

Sidaarth Sabhnani, MS
The University of Texas at Austin, 2026

SUPERVISOR: David Wu

Private information retrieval (PIR) lets a client fetch a record from a server-held database without revealing which record was accessed. Modern lattice-based PIR (Henzinger et al., 2023b) reduces each query to a database matrix–vector product under linearly homomorphic encryption with preprocessing, producing a large query-independent hint that the client uses for decryption. At deployment-scale workloads, cross-client batching (Menon and Wu, 2024) turns the database scan into a general matrix multiplication (GEMM)—a natural target for GPU acceleration. Recent schemes (Li et al., 2024; Menon and Wu, 2024; Mahdavi et al., 2025) compress the hint into the response via additional homomorphic computation at a throughput cost. The only prior GPU-accelerated lattice-PIR (Lehmkuhl et al., 2025) targeted memory bandwidth; however, a roofline analysis shows that batched PIR is compute-bound, scaling linearly to maximum throughput at the ridge point with only sub-linear growth in latency. Tensor cores, the specialized matrix-multiply units that deliver peak throughput in modern GPUs, were not used due to a mismatch between their precision and PIR moduli. That work was also unable to map the compression arithmetic to high-throughput GPU kernels, offloading it to a separate CPU cluster. Our goal is to accelerate communication-efficient PIR end-to-end—high online throughput, low online latency, fast offline preprocessing.

We present SandwichPIR: a cryptographic construction that enables hardware-aware algorithmic design for accelerated, communication-efficient lattice-PIR. SandwichPIR leverages a sequence of modulus-switching operations—a *sandwich* between a polynomial ring and the integer ring—so that each stage of PIR processing uses the required form of modular arithmetic for minimizing client latency and maximizing server throughput. Three algorithmic moves realize this: (i) client encryption, decryption, and GPU offline preprocessing run in the polynomial ring rather than the integer ring, an asymptotic speedup; (ii) the database scan in the integer ring becomes a byte-decomposed INT8 tensor-core GEMM; and (iii) response compression in the polynomial ring is redesigned as a second byte-decomposed INT8 tensor-core GEMM, eliminating the CPU cluster required by prior work.

The result is the first single-GPU PIR pipeline to achieve high throughput, low latency, and small communication. On an NVIDIA L4 GPU ($0.80/hr on AWS), SandwichPIR's online throughput is 4× higher than prior GPU lattice-PIR implementations at 15× better performance per dollar, and 10× higher than a CPU-cluster baseline at 30× better performance per dollar; our numbers include response compression on the GPU. Prior work excludes compression from its throughput, offloading it to a CPU cluster that accounts for over 99% of compute time and 95% of per-query cost when measured; without compression, clients fall back to downloading the full hint. On the offline path, our GPU hint kernel delivers a 20× throughput improvement and 76× better performance per dollar over prior work's GPU matmul baseline; column-sharded across multiple GPUs, we observe near-linear speedup for both the online and offline paths. We deploy SandwichPIR on the full English Wikipedia (6.4 M articles, 8 GiB) with a browser WebAssembly client; a single L4 serves >900 queries per second at under half a second end-to-end latency with a 320 KiB query, 160 KiB response, and a 288 KiB reusable hint replacing the 1 GiB hint download.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Private Information Retrieval

While TLS protects a client's traffic from network-level eavesdroppers, the server still sees every query in plaintext. For many applications this information is sensitive: a single search query can reveal where someone is, how they are doing, or what they care about (Henzinger et al., 2023a).

Private information retrieval (PIR) (Chor et al., 1995; Kushilevitz and Ostrovsky, 1997) closes this gap: it allows a client to retrieve the i-th record from a database of N records held by a server without revealing the index i to the server. The trivial solution—downloading the entire database—achieves perfect privacy but requires $O(N)$ communication; the goal is sublinear communication at manageable server cost.

PIR is a foundational primitive for privacy-preserving systems, with applications across several domains. In *Certificate Transparency auditing* (Henzinger et al., 2023b), a browser verifies certificate logs without disclosing which sites the user visits. In *password breach checking* (Ali et al., 2021), a client tests its credentials against a database of known leaks without exposing their password. In *private web search* (Henzinger et al., 2023a), a client issues search queries against a web-scale corpus without revealing the query terms or the documents retrieved. In *private media consumption* (Gupta et al., 2016), a client streams movies or articles without leaking their viewing history. Real-world deployments now exist in production: Apple's Live Caller ID Lookup (Apple, 2024) and Google's private device enrollment (Yeo and Patel, 2021) both build on PIR.

Chor et al. (1995) introduced PIR in the multi-server model, whose non-collusion assumption is brittle compared to cryptographic hardness; Kushilevitz and Ostrovsky (1997) gave the first single-server PIR from linearly homomorphic encryption. We focus exclusively on the single-server setting; today, the most performant constructions are lattice-based instantiations of this template (Henzinger et al., 2023b; Menon and Wu, 2024; Li et al., 2024; Mahdavi et al., 2025), built on the Learning-With-Errors (LWE) problem over the integers and its ring variant (RLWE) over polynomials.

Efficiency metrics. A PIR server must touch every byte of the database on each query—otherwise it would learn which records the client is *not* interested in—so *throughput* (database size divided by server time) is the natural performance metric, with an $\Omega(N)$ work floor. A new line of research on doubly-efficient PIR (Canetti et al., 2017) avoids this floor, but at concretely prohibitive cost, so we do not consider it here. *Communication* is the total bytes exchanged per query, plus any one-time offline cost. Every PIR scheme navigates a tradeoff between the two, and recent work has pushed both—though rarely at the same time.

1.2 The State of the Art

Concretely-efficient single-server PIR schemes fall into three design points based on the offline preprocessing they require, plus an orthogonal axis of hardware acceleration.

Hint-download schemes. SimplePIR (Henzinger et al., 2023b) achieves the highest throughputs using linearly homomorphic encryption with preprocessing: in an offline phase the client downloads a *hint* $H = A \cdot \mathrm{DB}$, the projection of the database under a common public matrix A, and every online query then reduces to the server computing $\mathrm{DB} \cdot q$ over $\mathbb{Z}_q$ for the client's encrypted query q—a dense matrix-vector

product bounded only by memory bandwidth on a single CPU core, which the client decrypts using its cached hint. The hint is large, however: even on the modest 8 GiB Wikipedia database we study, SimplePIR's hint ranges from 512 MiB to 1 GiB depending on the lattice dimension, and in some instantiations exceeds the database itself. Additionally, this penalizes dynamic databases, since every update forces a re-download, and clients that query multiple databases, since each requires its own hint.

Public-key schemes. Spiral and OnionPIR (Menon and Wu, 2022; Mughees et al., 2021) use offline client public keys and fully homomorphic encryption to compress queries and responses, achieving the best per-query communication for large records. Throughput is 10–100× lower than hint-download schemes, however, and the server must store and manage per-client public-key state—we do not study them in this thesis.

Silent preprocessing. Recent work eliminates both offline communication and per-client server state by compressing the hint itself into the per-query response. Tiptoe (Henzinger et al., 2023a) and HintlessPIR (Li et al., 2024) add a second layer of linearly homomorphic encryption so the server returns a small encrypted product in place of the full hint matrix; throughput drops to ∼7× below SimplePIR for Tiptoe and ∼62% of SimplePIR for HintlessPIR. YPIR (Menon and Wu, 2024) and InsPIRe (Mahdavi et al., 2025) instead apply LWE-to-RLWE *ring packing* (Chen et al., 2021)—a technique that compresses the many per-row scalar ciphertexts of a SimplePIR-style response into a single packed polynomial RLWE ciphertext, an asymptotic factor-d reduction in download—achieving the lowest per-query communication at similar throughput to HintlessPIR.

Hardware acceleration. The CPU PIRs above all lean on SIMD (AVX2/AVX-512) to saturate memory bandwidth (Longa and Naehrig, 2016). GPUs are the nat-

ural accelerator: under cross-client batching, the server's per-query matrix-vector product becomes a dense general matrix multiplication (GEMM) that amortizes the database read across all queries in the batch. The only prior GPU-accelerated lattice-PIR (Lehmkuhl et al., 2025) offloaded the database scan to a GPU for its higher memory bandwidth via black-box matrix-multiplication library calls, reporting ∼2.15 TiB/s on an NVIDIA V100 (16 GiB VRAM, $3.06/hr on AWS `p3.2xlarge`) at batch size 256 using the INT32 SIMT path and leaving tensor cores unused due to a mismatch between their precision and PIR moduli. That throughput is the database scan alone: the authors conclude that response compression "is not efficient on GPUs" and offload it to a separate CPU cluster, which when measured accounts for over 99% of total compute time and 95% of per-query dollar cost; without compression, clients fall back to downloading the full hint.

1.3 Contributions

This thesis presents SandwichPIR, a cryptographic construction that enables hardware-aware algorithmic design for accelerated, communication-efficient lattice-PIR. Our starting point is the roofline insight that at deployment-scale batch sizes, the database is read once and reused across all clients, so arithmetic intensity grows with the batch size; PIR is compute-bound rather than memory-bandwidth-bound, scaling linearly to maximum throughput at the ridge point with only sub-linear growth in latency. INT8 tensor cores deliver peak throughput per dollar on modern GPUs, and exploiting them for lattice-PIR requires non-trivial algorithmic work. SandwichPIR leverages a sequence of modulus-switching operations—a *sandwich* between a polynomial ring R_Q and the integer ring $\mathbb{Z}_{2^{32}}$—so that each stage of PIR processing uses the required form of modular arithmetic, minimizing client latency and maximizing server throughput. Our contributions are as follows.

Contribution 1: Tensor-core-aware design for lattice-PIR. Two algorithmic moves recast the two dominant server computations of a SimplePIR-family PIR as dense INT8 tensor-core GEMMs. The database scan runs in $\mathbb{Z}_{2^{32}}$—the native accumulator of an INT8 tensor core—via byte decomposition, which splits each 32-bit query operand into four 8-bit digits and stacks them so a single INT8 GEMM produces all cross-products; this lifts the scan $\sim$4$\times$ above the INT32 SIMT plateau on the same die. Response compression in R_Q becomes a second such GEMM: we absorb InspiRING packing's (Mahdavi et al., 2025) client-independent automorphism permutations offline into a single matrix M, turning a sparse permuted gather into the dense GEMM $Y \cdot M$ at the same arithmetic intensity as the scan. Composing scan in $\mathbb{Z}_{2^{32}}$ with encryption and packing in R_Q requires the ciphertext *body* to round-trip $R_Q \to \mathbb{Z}_{2^{32}} \to R_Q$ while the *mask* stays in R_Q throughout; we prove this asymmetry is necessary, since the symmetric alternative introduces noise that grows linearly in the database dimension. The result is the first single-GPU PIR pipeline to achieve high throughput, low latency, and small communication. On an NVIDIA L4 (24 GiB VRAM, \$0.80/hr on AWS `g6.xlarge`), SandwichPIR's online throughput reaches 8.35 TiB/s—4$\times$ higher than DistPIR (Lehmkuhl et al., 2025) on a V100 at 15$\times$ better performance per dollar, and 10$\times$ higher than its 8-instance CPU-cluster baseline at 30$\times$ better performance per dollar. On an NVIDIA A100 (40 GiB VRAM, $\sim$\$3/hr per GPU on AWS `p4d.24xlarge`), throughput reaches 20.28 TiB/s. Our measurements include response compression on the GPU. Prior work excludes compression from its throughput, offloading it to a CPU cluster (Section 1.2).

Contribution 2: GPU-accelerated offline phase and multi-GPU sharding. Every SimplePIR-family scheme must compute a hint $H = A \cdot \mathrm{DB}$; prior implementations (Lehmkuhl et al., 2025) achieve 0.51 GiB/s on GPU via matrix multiplication and 0.25 GiB/s on CPU via polynomial-ring arithmetic on a 4 GiB database, declining the polynomial-ring path on GPU as impractical. Our polynomial-based hint kernel runs on GPU at 10 GiB/s on L4 and 19 GiB/s on A100—at 76$\times$ better per-

formance per dollar than the GPU matmul baseline. We GPU-accelerate the InspiRING packing-matrix M assembly as well; end-to-end offline preprocessing completes an 8 GiB database in 5.9 s on L4 and 3.1 s on A100. Column-wise database sharding across multiple GPUs achieves near-linear speedup: 2.71× online and 2.59× offline on 3×A100 (TACC Lonestar6).

Contribution 3: End-to-end system and Private Wikipedia. Client encryption and decryption also run in the polynomial ring rather than the integer ring, an asymptotic speedup over scalar operations that reduces per-query client time from seconds to milliseconds. We implement SandwichPIR as an application-agnostic stack—a stateless GPU-accelerated PIR server and a Rust client library compiled to both native binaries and browser WebAssembly—with ∼2,700 lines of CUDA code. The server and client communicate over a stateless HTTP API with *lazy hint delivery*: on the first query the server ships a client-key-independent polynomial in-band with the response, and the client caches it so subsequent queries receive only the query-dependent body. As a reference application, we deploy Private Wikipedia: all 6.4 million English Wikipedia articles bin-packed into an 8 GiB database, served to a browser UI that searches by title and fetches articles privately via the WASM client. A single L4 serves >900 queries per second with a 288 KiB cached hint and 160 KiB per-query response; end-to-end latency—including client encryption, network transfer, server compute, and client decryption—is under half a second: 360 ms on mobile and 184 ms on fixed broadband for first queries, 349 ms and 176 ms for subsequent. The full implementation is open-source at `https://github.com/sidsabh/sandwichpir`.

1.4 Thesis Organization

Chapter 2 develops the cryptographic background in three parts: PIR with preprocessing and its linearly homomorphic encryption abstraction; the lattice problems and lattice-based encryption that instantiate it; and the lattice-PIR schemes

(SimplePIR, Tiptoe, HintlessPIR, YPIR, InsPIRe, DistPIR) that motivate this thesis. Chapter 3 examines the design space through a GPU lens: cross-client batching, a roofline analysis that puts PIR in the compute-bound regime, byte-decomposed tensor-core arithmetic, the GEMM reformulation of InsPIRe ring packing, GPU-side offline preprocessing, client-side polynomial-ring arithmetic, and an audit of the prior schemes against the resulting criteria. Chapter 4 presents the SandwichPIR protocol, its asymmetric modulus-switching design, security reduction to RLWE, concrete bit-security estimate, concrete correctness, and per-query communication costs. Chapter 5 covers implementation, performance, communication, and multi-GPU scaling, with comparisons to prior work (Lehmkuhl et al., 2025) on both its V100 and CPU-cluster baselines. Chapter 6 deploys Private Wikipedia via SandwichPIR: system architecture, the browser WebAssembly client with lazy hint delivery, batch-size selection under Wikipedia's request-rate load, fleet sizing and GPU selection, and end-to-end latency analysis on mobile and broadband. Chapter 7 summarizes the contributions, discusses limitations, and outlines future directions.

Chapter 2

Cryptographic Background

This chapter develops the cryptographic background for lattice-based PIR. It is organized into three parts. Section 2.1 defines the preliminaries used throughout.

Cryptographic core (Sections 2.2–2.3). We define *private information retrieval* (PIR) with preprocessing—the central primitive of this thesis—and then introduce *linearly homomorphic encryption with preprocessing* (LHEwP) (Henzinger et al., 2023b), the cipher machinery that SimplePIR introduced as a building block. Every modern lattice-PIR scheme we examine is a PIR-with-preprocessing construction; the schemes differ in which LHEwP instance and function they use internally and where the resulting hint material resides. As a first example, we present Kushilevitz and Ostrovsky's $O(\sqrt{N})$-communication PIR from any linearly homomorphic cipher (using LHEwP with trivial preprocessing).

Lattice core (Sections 2.4–2.7). Lattice-based ciphers are believed secure against quantum adversaries and admit the most efficient linearly homomorphic operations of any cipher family. We introduce the lattice problems whose assumed hardness underpins modern cryptography, define the two learning-with-errors assumptions we use throughout (*LWE* over $\mathbb{Z}_q$ and its ring variant *RLWE* over a cyclotomic polynomial ring), instantiate the corresponding linearly homomorphic symmetric ciphers, and collect the shared properties that the lattice-PIR schemes below rely on.

Lattice-based PIR (Sections 2.8–2.15). We review the concretely-efficient lattice-PIR schemes that motivate this thesis. SimplePIR sets the template; the boot-

strapping family (Tiptoe, HintlessPIR) and the ring-packing family (YPIR, InsPIRe) each eliminate SimplePIR's offline hint download by different mechanisms. DistPIR (Lehmkuhl et al., 2025) is the first GPU-accelerated PIR. We focus throughout on *large-record* PIR, the natural setting of the SimplePIR family, instead of small-record PIR (the DoublePIR family).

2.1 Preliminaries

We write λ for the security parameter. For $n \in \mathbb{N}$ we write $[n]$ for $\{1, \ldots, n\}$. We write $\mathbb{Z}_q$ for the integers modulo q, and $R_q = R/qR$ for the cyclotomic ring modulo q (Definition 2.5). Bold uppercase letters denote matrices $(\mathbf{A}, \mathbf{B})$; bold lowercase letters denote vectors $(\mathbf{u}, \mathbf{v})$; $\mathbf{A}^\mathsf{T}$ is the transpose. We write $\|\cdot\|_\infty$ for the ℓ_∞ norm throughout—of vectors in $\mathbb{Z}^n$ and of the coefficient vector of ring elements $g \in R$. For mixed-modulus products $\mathbf{AB}$ where $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$ and $\mathbf{B} \in \mathbb{Z}^{m \times k}$ (or vice versa), each entry of $\mathbf{B}$ is identified with its mod-q representative in $(-q/2, q/2]$ before multiplication.

We write $\mathrm{poly}(\lambda)$ for any function $O(\lambda^c)$, $c \in \mathbb{N}$, and $\mathrm{negl}(\lambda)$ for any function $o(\lambda^{-c})$ for every $c \in \mathbb{N}$. An algorithm is *efficient* if it runs in probabilistic polynomial time in its input length; two distribution families are *computationally indistinguishable* if no efficient algorithm distinguishes them with non-negligible advantage. We write $x \xleftarrow{R} S$ for sampling x uniformly at random from a set S, and $x \leftarrow \chi$ for sampling from a named distribution χ.

Rounding and modulus switching. For $x \in \mathbb{R}$, $\lfloor x \rceil$ denotes the nearest integer to x (rounding up on ties), and $\lfloor x \rfloor$ the largest integer not exceeding x. For integers q and q', the *modulus-switching operator* $\lfloor \cdot \rceil_{q,q'} : \mathbb{Z}_q \to \mathbb{Z}_{q'}$ lifts its input $x \in \mathbb{Z}_q$ to the representative $x' \in (-q/2, q/2]$ and outputs $\lfloor (q'/q) \cdot x' \rceil$ in $\mathbb{Z}_{q'}$ (division and rounding over the rationals). All of these extend component-wise to vectors, matrices, and polynomial coefficients.

Discrete Gaussians and sub-Gaussian tail bounds. The *discrete Gaussian* $D_{\mathbb{Z},\sigma}$ over $\mathbb{Z}$ with mean 0 and standard deviation σ has PMF

$$\Pr[X = x : X \leftarrow D_{\mathbb{Z},\sigma}] \;\propto\; \exp\bigl(-x^2/(2\sigma^2)\bigr).$$

We write $D(\sigma) \coloneqq D_{\mathbb{Z},\sigma}$ when the domain is clear from context. A real-valued random variable X is *sub-Gaussian* with width parameter σ (in the convention used throughout lattice cryptography (Menon and Wu, 2024)) if for every $t \geq 0$,

$$\Pr[|X| > t] \;\leq\; 2\exp(-\pi t^2/\sigma^2).$$

The notation σ is overloaded across these two definitions by convention, and the noise analysis of Chapter 4 and Appendix A works in the sub-Gaussian width; the conversion is:

- $X \leftarrow D_{\mathbb{Z},\sigma_{\text{disc}}}$: sub-Gaussian with width $\sigma = \sigma_{\text{disc}}\sqrt{2\pi}$.
- X mean-zero and bounded, $\mathbb{E}[X] = 0$ and $|X| \leq B$: sub-Gaussian with width $\sigma = B\sqrt{2\pi}$ (Hoeffding's lemma).

In particular, at our parameters $X_s = X_e = D(0.5)$ gives width $\sigma_s = \sigma_e = 0.5\sqrt{2\pi} \approx 1.253$, and a $[-1/2, 1/2]$-bounded rounding error has width $\sqrt{2\pi}/2$. Sub-Gaussian random variables compose under two rules:

- *Scaling*: X sub-Gaussian with parameter $\sigma \Rightarrow cX$ sub-Gaussian with parameter $|c|\sigma$, for any $c \in \mathbb{R}$.
- *Independent sum*: independent sub-Gaussians $X_1, \ldots, X_k$ with parameters σ_i sum to a sub-Gaussian with parameter $\bigl(\sum_i \sigma_i^2\bigr)^{1/2}$.

2.2 Private Information Retrieval

Private information retrieval (PIR) (Chor et al., 1995) lets a client retrieve a record from a public database held by a server, without revealing which record

was queried. We adopt the PIR-with-preprocessing formulation of Henzinger et al. (2023b), which exposes hint material on *both* the server and client sides—enough to describe every scheme in the lattice-PIR family by which hints it makes non-trivial.

Definition 2.1 (PIR with preprocessing, adapted from Henzinger et al. (2023b))**.** Over plaintext space $\mathcal{D}$ and database size $N \in \mathbb{N}$, a PIR-with-preprocessing scheme is a tuple of efficient algorithms $(\mathsf{Setup}, \mathsf{Query}, \mathsf{Answer}, \mathsf{Recover})$:

- $\mathsf{Setup}(\mathrm{db}) \to (\mathrm{hint}_s, \mathrm{hint}_c)$: on input a database $\mathrm{db} \in \mathcal{D}^N$, output preprocessed hints for the server and the client.
- $\mathsf{Query}(i) \to (\mathrm{st}, \mathrm{qu})$: on input an index $i \in [N]$, output a secret client state st and a database query qu.
- $\mathsf{Answer}(\mathrm{db}, \mathrm{hint}_s, \mathrm{qu}) \to \mathrm{ans}$: output a server answer.
- $\mathsf{Recover}(\mathrm{st}, \mathrm{hint}_c, \mathrm{ans}) \to r$: output a record $r \in \mathcal{D}$.

The scheme must satisfy:

1. **Correctness.** The scheme has correctness error δ if, for all databases $\mathrm{db} = (d_1, \ldots, d_N) \in \mathcal{D}^N$ and all indices $i \in [N]$,
$$\Pr\left[\hat{d}_i = d_i \;:\; \begin{array}{r}(\mathrm{hint}_s, \mathrm{hint}_c) \leftarrow \mathsf{Setup}(\mathrm{db}) \\ (\mathrm{st}, \mathrm{qu}) \leftarrow \mathsf{Query}(i) \\ \mathrm{ans} \leftarrow \mathsf{Answer}(\mathrm{db}, \mathrm{hint}_s, \mathrm{qu}) \\ \hat{d}_i \leftarrow \mathsf{Recover}(\mathrm{st}, \mathrm{hint}_c, \mathrm{ans})\end{array}\right] \geq 1 - \delta.$$
2. **Query privacy.** For a bit $b \in \{0, 1\}$, the query-privacy game between an adversary $\mathcal{A}$ and a challenger proceeds as follows:
   - On input 1^λ, $\mathcal{A}$ outputs a database db.
   - The challenger computes $(\mathrm{hint}_s, \mathrm{hint}_c) \leftarrow \mathsf{Setup}(\mathrm{db})$ and sends both to $\mathcal{A}$.
   - $\mathcal{A}$ outputs a pair of indices $i_0, i_1 \in [N]$. The challenger computes $(\mathrm{st}, \mathrm{qu}) \leftarrow \mathsf{Query}(i_b)$ and returns qu.

- $\mathcal{A}$ outputs a bit $b' \in \{0,1\}$.

The scheme satisfies query privacy if for every efficient $\mathcal{A}$,

$$|\Pr[b' = 1 \ : \ b = 0] \ - \ \Pr[b' = 1 \ : \ b = 1]| \ = \ \mathrm{negl}(\lambda).$$

For the PIR scheme to be non-trivial, the total communication must be smaller than the database: $|\mathrm{hint}_c| + |\mathrm{qu}| + |\mathrm{ans}| \ll |\mathrm{db}|$.

2.3 Linearly Homomorphic Encryption with Preprocessing

The building block we use to construct PIR is *linearly homomorphic encryption with preprocessing*, introduced by Henzinger et al. (2023b). A linear function $f\colon \mathbb{Z}_p^m \to \mathbb{Z}_p$ is one of the form $f(\mathbf{v}) = \langle \mathbf{w}, \mathbf{v} \rangle$ for some $\mathbf{w} \in \mathbb{Z}_p^m$; matrix–vector products $\mathbf{v} \mapsto \mathbf{W}\mathbf{v}$ are obtained by evaluating one such f per row of $\mathbf{W}$. A function-specific *hint* is computed from f once, up-front, and enables fast online evaluation of f on any ciphertext.

Definition 2.2 (Linearly homomorphic encryption with preprocessing, generalized from Henzinger et al. (2023b))**.** A scheme $\mathcal{E} = (\mathsf{Setup}, \mathsf{Enc}, \mathsf{Preproc}, \mathsf{Apply}, \mathsf{Dec})$ over key space $\mathcal{K}$, plaintext modulus p, and dimension m is a tuple of efficient algorithms:

- $\mathsf{Setup}(1^\lambda) \to \mathrm{pp}$: output public parameters.
- $\mathsf{Enc}(\mathrm{pp}, \mathrm{sk}, \mathbf{v}) \to \mathrm{ct}$: encrypt $\mathbf{v} \in \mathbb{Z}_p^m$ under $\mathrm{sk} \in \mathcal{K}$.
- $\mathsf{Preproc}(\mathrm{pp}, f) \to (\mathrm{hint}_s, \mathrm{hint}_c)$: on input a linear function $f\colon \mathbb{Z}_p^m \to \mathbb{Z}_p$, output server- and client-side function hints (either may be $\perp$).
- $\mathsf{Apply}(\mathrm{pp}, f, \mathrm{hint}_s, \mathrm{ct}) \to \mathrm{ct}_f$: produce a ciphertext encrypting $f(\mathbf{v})$.
- $\mathsf{Dec}(\mathrm{sk}, \mathrm{hint}_c, \mathrm{ct}_f) \to y$: output the decrypted value $y \in \mathbb{Z}_p$.

The scheme must satisfy:

1. **Correctness.** The scheme has correctness error δ if, for all $\mathrm{sk} \in \mathcal{K}$, $\mathbf{v} \in \mathbb{Z}_p^m$, and linear f,

$$\Pr\left[\, y = f(\mathbf{v}) \;:\; \begin{array}{r} \mathrm{pp} \leftarrow \mathsf{Setup}(1^\lambda) \\ \mathrm{ct} \leftarrow \mathsf{Enc}(\mathrm{pp}, \mathrm{sk}, \mathbf{v}) \\ (\mathrm{hint}_s, \mathrm{hint}_c) \leftarrow \mathsf{Preproc}(\mathrm{pp}, f) \\ \mathrm{ct}_f \leftarrow \mathsf{Apply}(\mathrm{pp}, f, \mathrm{hint}_s, \mathrm{ct}) \\ y \leftarrow \mathsf{Dec}(\mathrm{sk}, \mathrm{hint}_c, \mathrm{ct}_f) \end{array} \,\right] \;\geq\; 1 - \delta.$$

2. **Semantic security.** For a bit $b \in \{0,1\}$, the semantic-security game between an adversary $\mathcal{A}$ and a challenger proceeds as follows:

   - On input 1^λ, $\mathcal{A}$ outputs two vectors $\mathbf{v}_0, \mathbf{v}_1 \in \mathbb{Z}_p^m$.
   - The challenger computes $\mathrm{pp} \leftarrow \mathsf{Setup}(1^\lambda)$, samples $\mathrm{sk} \xleftarrow{R} \mathcal{K}$, and sends $(\mathrm{pp}, \mathrm{ct})$ with $\mathrm{ct} \leftarrow \mathsf{Enc}(\mathrm{pp}, \mathrm{sk}, \mathbf{v}_b)$.
   - $\mathcal{A}$ outputs a bit $b' \in \{0,1\}$.

   The scheme is *semantically secure* if for every efficient $\mathcal{A}$,

$$|\Pr[b' = 1 \;:\; b = 0] \;-\; \Pr[b' = 1 \;:\; b = 1]| \;=\; \mathrm{negl}(\lambda).$$

The semantic-security game is single-challenge and mirrors the query-privacy game of Definition 2.1: the adversary chooses two inputs, gets one encryption, and tries to tell them apart. This matches the single-server PIR setting, in which the server sees only ciphertexts encrypted under one freshly sampled client secret.

2.3.1 Square-Root PIR from Linearly Homomorphic Encryption

Kushilevitz and Ostrovsky (1997) gave the first single-server PIR from any linearly homomorphic cipher, achieving $O(\sqrt{N})$ total communication. Fix an LHEwP scheme $\mathcal{E} = (\mathsf{Setup}_\mathcal{E}, \mathsf{Enc}, \mathsf{Preproc}, \mathsf{Apply}, \mathsf{Dec})$ over $\mathbb{Z}_p$ and dimension $\sqrt{N}$ (Definition 2.2); the KO construction uses $\mathcal{E}$'s linear homomorphism but *not* its $\mathsf{Preproc}$, so both hint_s and hint_c are $\perp$ throughout. Reshape the database as $\mathrm{db} \in \mathbb{Z}_p^{\sqrt{N} \times \sqrt{N}}$ and index records by $(r, c) \in [\sqrt{N}]^2$.

- $\mathsf{Setup}(\mathrm{db})$: output $(\mathrm{hint}_s, \mathrm{hint}_c) = (\perp, \perp)$.
- $\mathsf{Query}((r, c))$: sample $\mathrm{sk} \xleftarrow{R} \mathcal{K}$, $\mathrm{pp} \leftarrow \mathsf{Setup}_{\mathcal{E}}(1^\lambda)$, and output $\mathrm{st} = (\mathrm{sk}, \mathrm{pp}, r)$ and $\mathrm{qu} = \big(\mathrm{pp}, \mathsf{Enc}(\mathrm{pp}, \mathrm{sk}, \mathbf{u}_c)\big)$, where $\mathbf{u}_c \in \mathbb{Z}_p^{\sqrt{N}}$ is the indicator vector for column c.
- $\mathsf{Answer}(\mathrm{db}, \perp, \mathrm{qu})$: parse $\mathrm{qu} = (\mathrm{pp}, \mathrm{ct})$; for each row $r' \in [\sqrt{N}]$, compute $\mathrm{ans}_{r'} = \mathsf{Apply}(\mathrm{pp}, f_{r'}, \perp, \mathrm{ct})$ with $f_{r'}(\mathbf{v}) = \langle \mathrm{db}_{r'}, \mathbf{v} \rangle$, and output $\mathrm{ans} = (\mathrm{ans}_1, \ldots, \mathrm{ans}_{\sqrt{N}})$.
- $\mathsf{Recover}((\mathrm{sk}, \mathrm{pp}, r), \perp, \mathrm{ans})$: output $\mathsf{Dec}(\mathrm{sk}, \perp, \mathrm{ans}_r)$.

Correctness. By correctness of $\mathcal{E}$, $\mathsf{Dec}(\mathrm{sk}, \perp, \mathrm{ans}_r) = f_r(\mathbf{u}_c) = \mathrm{db}[r, c]$.

Query privacy. Reduces to semantic security of $\mathcal{E}$: a distinguisher between queries for (r_0, c_0) and (r_1, c_1) would distinguish encryptions of $\mathbf{u}_{c_0}$ from $\mathbf{u}_{c_1}$.

Communication and computation. $|\mathrm{qu}| = |\mathrm{ans}| = \sqrt{N}$ ciphertexts. Server work is $\sqrt{N}$ invocations of Apply, one per database row, each evaluating a $\sqrt{N}$-dimensional linear function; the per-Apply cost depends on the underlying cipher (Section 2.5.1). Server computation scales with every database entry, the standard $\Omega(N)$ floor for single-server PIR.

The practical communication and throughput of any such KO-style scheme is therefore determined by the size of the ciphertext and per-Apply cost of $\mathcal{E}$, respectively. We now develop the lattice machinery needed to instantiate $\mathcal{E}$ with the most efficient linearly homomorphic ciphers known; Section 2.11 later generalizes LHEwP to the composable-preprocessing setting that every silent-preprocessing scheme in this chapter uses.

2.4 Lattices and their Hard Problems

We review the lattice theory to the extent needed to define the underlying assumption; see Peikert (2016) for a complete background.

Definition 2.3 (Lattice)**.** Given linearly independent $\mathbf{b}_1, \ldots, \mathbf{b}_n \in \mathbb{R}^n$, the *(full-rank) lattice* generated by these vectors is

$$\mathcal{L}(\mathbf{B}) \;=\; \Big\{ \textstyle\sum_{i=1}^{n} z_i \mathbf{b}_i \;:\; z_i \in \mathbb{Z} \Big\}.$$

The ordered tuple $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$ is a *basis*; many bases generate the same lattice.

The canonical hard problem is finding short vectors. Exact SVP—given a basis, return a shortest nonzero lattice vector—is **NP**-hard, so no polynomial-time algorithm is expected. Cryptography uses approximate and decisional relaxations:

Definition 2.4 (Approximate and decisional SVP)**.** Let $\lambda_1(\mathcal{L})$ denote the length of the shortest nonzero vector in $\mathcal{L}$.

- SVP_γ: given $\mathbf{B}$, find $\mathbf{v} \in \mathcal{L}(\mathbf{B}) \setminus \{\mathbf{0}\}$ with $\|\mathbf{v}\|_\infty \le \gamma \cdot \lambda_1(\mathcal{L})$.
- GapSVP_γ: the decisional version of SVP_γ.
- SIVP_γ: given $\mathbf{B}$, find n linearly independent lattice vectors each of length at most γ times the n-th successive minimum of $\mathcal{L}$.

No polynomial-time algorithm for GapSVP_γ or SIVP_γ is known, classically or quantumly, at the polynomial approximation factors $\gamma = \mathrm{poly}(n)$ used to build lattice cryptography ($\gamma = n$ is the smallest known factor to suffice). Polynomial-time algorithms only achieve slightly subexponential approximation factors via the time–approximation trade-off $\gamma = 2^k$ in $2^{\widetilde{\Theta}(n/k)}$ time, infeasible at the concrete parameters used in practice. Quantum algorithms achieve the same asymptotics, with only smaller hidden constants—lattice problems are therefore believed post-quantum.

2.5 Learning with Errors

The Learning with Errors (LWE) problem (Regev, 2005) asks to solve a noisy linear system over $\mathbb{Z}_q$.

Learning with Errors (LWE). The LWE problem is defined with respect to lattice parameters n, m, q, χ, where χ is an *error distribution* over $\mathbb{Z}_q$ (oftentimes, this is a discrete Gaussian distribution over $\mathbb{Z}_q$). The $\mathsf{LWE}_{n,m,q,\chi}$ assumption states that for a random choice $\mathbf{A} \xleftarrow{R} \mathbb{Z}_q^{n\times m}$, $\mathbf{s} \xleftarrow{R} \mathbb{Z}_q^n$, $\mathbf{e} \leftarrow \chi^m$, the following two distributions are computationally indistinguishable:

$$(\mathbf{A},\, \mathbf{s}^\top \mathbf{A} + \mathbf{e}^\top) \;\overset{c}{\approx}\; (\mathbf{A},\, \mathbf{r}),$$

where $\mathbf{r} \xleftarrow{R} \mathbb{Z}_q^m$.

Theorem 2.1 (LWE hardness (Regev, 2005))**.** *Let $\chi = D_{\mathbb{Z},\alpha q}$ be a discrete Gaussian with parameter $\alpha q \geq 2\sqrt{n}$ and $\alpha \in (0,1)$. For $m = \mathrm{poly}(n)$ and $q \leq 2^{\mathrm{poly}(n)}$, solving decision-$\mathrm{LWE}_{n,m,q,\chi}$ is at least as hard as* quantumly *solving* GapSVP_γ *and* SIVP_γ *on arbitrary n-dimensional lattices, for $\gamma = \widetilde{O}(n/\alpha)$; search-LWE is at least as hard as decision.*

Classical reductions hold with larger moduli. This is a *worst-case to average-case* reduction: an efficient algorithm for average-case LWE would solve the lattice problems on every input lattice.

2.5.1 Symmetric Encryption from LWE

LWE yields a simple symmetric-key encryption scheme: an LWE sample, masked by a scaled plaintext, is already a ciphertext (Regev, 2005). For plaintext modulus $p < q$, the scheme $\mathcal{E}_{\mathrm{LWE}} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ over message space $\mathbb{Z}_p$ is:

- $\mathsf{Gen}(1^\lambda)$: sample $\mathbf{s} \xleftarrow{R} \mathbb{Z}_q^n$ (uniform secret; small/normal-form variants are equivalent up to a polynomial change in samples, see Section 2.7).

- $\mathsf{Enc}(\mathbf{s}, \mu)$: sample $\mathbf{a} \xleftarrow{R} \mathbb{Z}_q^n$ and $e \leftarrow \chi$, and output

$$\text{ct} \;=\; \big(\mathbf{a},\; b = \langle \mathbf{a}, \mathbf{s} \rangle + e + \lfloor q/p \rfloor \cdot \mu\big) \;\in\; \mathbb{Z}_q^n \times \mathbb{Z}_q. \tag{2.1}$$

- $\mathsf{Dec}(\mathbf{s}, (\mathbf{a}, b))$: compute $b - \langle \mathbf{a}, \mathbf{s} \rangle \;=\; \lfloor q/p \rfloor \cdot \mu + e \pmod q$ and output $\lfloor \cdot \rceil_{q,p}$ applied to the result.

Correctness. Decryption succeeds whenever $|e| < \frac{q}{2p} - (q \bmod p)$, which holds with overwhelming probability for χ concentrated below $q/(2p)$.

Semantic security. Under the decision-LWE assumption and triangle inequality, $\mathcal{E}_{\text{LWE}}$ is semantically secure (Definition 2.2): the pair $(\mathbf{a}, \langle \mathbf{a}, \mathbf{s} \rangle + e)$ is indistinguishable from uniform $(\mathbf{a}, u)$, and adding $\lfloor q/p \rfloor \mu_b$ to a uniform element preserves uniformity. An efficient adversary with non-negligible advantage would therefore break decision-LWE.

2.6 Ring Learning with Errors

Ring LWE (RLWE) (Lyubashevsky et al., 2010) replaces the vector space $\mathbb{Z}_q^n$ with a polynomial ring.

Definition 2.5 (Cyclotomic ring)**.** For a power-of-two d, define $R = \mathbb{Z}[x]/(x^d + 1)$ and $R_q = R/qR$. Elements are polynomials of degree $< d$ with coefficients in $\mathbb{Z}_q$; multiplication is negacyclic convolution. For any modulus q', we write $R_{q'} \coloneqq \mathbb{Z}_{q'}[x]/(x^d + 1)$; the underlying polynomial ring R is fixed by d.

Polynomial-ring operations. Let $g = \sum_{i=0}^{d-1} \alpha_i x^i \in R$.

- Coeffs: $R \to \mathbb{Z}^d$ maps $g \mapsto (\alpha_0, \ldots, \alpha_{d-1})^T$. We write $\|g\|_\infty = \max_i |\alpha_i|$ for the ℓ_∞ norm of the coefficient vector.

- $\mathsf{NCyclicMat}\colon R \to \mathbb{Z}^{d\times d}$ is the linear map representing multiplication by g: for every $f \in R$,

$$\mathsf{Coeffs}(f)^T \cdot \mathsf{NCyclicMat}(g) \;=\; \mathsf{Coeffs}(fg)^T,$$

concretely

$$\mathsf{NCyclicMat}(g) \;=\; \begin{bmatrix} \alpha_0 & \alpha_1 & \alpha_2 & \cdots & \alpha_{d-1} \\ -\alpha_{d-1} & \alpha_0 & \alpha_1 & \cdots & \alpha_{d-2} \\ -\alpha_{d-2} & -\alpha_{d-1} & \alpha_0 & \cdots & \alpha_{d-3} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -\alpha_1 & -\alpha_2 & -\alpha_3 & \cdots & \alpha_0 \end{bmatrix}.$$

We extend $\mathsf{NCyclicMat}$ to vectors $\mathbf{g} = (g_1, \ldots, g_m) \in R^m$ component-wise: for every $f \in R$,

$$\begin{aligned} \mathsf{Coeffs}(f \cdot \mathbf{g})^T \;&=\; \left[\mathsf{Coeffs}(fg_1)^T \mid \cdots \mid \mathsf{Coeffs}(fg_m)^T\right] \\ &=\; \mathsf{Coeffs}(f)^T \cdot \mathsf{NCyclicMat}(\mathbf{g}^T). \end{aligned} \tag{2.2}$$

Definition 2.6 (Ring LWE)**.** Fix a cyclotomic ring $R = \mathbb{Z}[x]/(x^d + 1)$ of degree d, modulus q, and error distribution χ over R. The $\mathrm{RLWE}_{d,m,q,\chi}$ assumption is that for $\mathbf{a} \xleftarrow{R} R_q^m$, $s \xleftarrow{R} R_q$, and $\mathbf{e} \leftarrow \chi^m$, the distribution $(\mathbf{a}, s \cdot \mathbf{a} + \mathbf{e})$ is computationally indistinguishable from $(\mathbf{a}, \mathbf{v})$ with $\mathbf{v} \xleftarrow{R} R_q^m$.

Lyubashevsky et al. (2010) give a quantum worst-case to average-case reduction from approximate SVP_γ on *ideal lattices* in R (for appropriate q, χ, and γ) to RLWE—a narrower class than the arbitrary lattices LWE reduces from, and so a somewhat stronger assumption. Every lattice-PIR scheme in this thesis except SimplePIR relies on RLWE—for client-side query encryption and server-side hint computation, ring packing, and/or homomorphic hint compression.

2.6.1 Symmetric Encryption from RLWE

$\mathcal{E}_{\mathrm{RLWE}}$ mirrors $\mathcal{E}_{\mathrm{LWE}}$ over R_q in place of $\mathbb{Z}_q^n$, with message space $R_p = \mathbb{Z}_p[x]/(x^d + 1)$:

$$\mathsf{Enc}(s, \mu) \;=\; \big(a,\; b = a \cdot s + e + \lfloor q/p \rfloor \cdot \mu\big) \;\in\; R_q^2, \qquad a \xleftarrow{R} R_q,\; e \leftarrow \chi. \tag{2.3}$$

Decryption computes $b - a \cdot s$ and applies $\lfloor \cdot \rceil_{q,p}$ coefficient-wise; a single ciphertext carries d plaintext coefficients of R_p. Correctness (per-coefficient $|e_j| < q/(2p) - (q \bmod p)$) and semantic security (decision-RLWE) transfer verbatim from $\mathcal{E}_{\mathrm{LWE}}$.

2.6.2 Practical Benefits of RLWE

Two concrete properties make RLWE—rather than plain LWE—the workhorse of every PIR scheme in this thesis except SimplePIR.

Compact ciphertexts. An LWE ciphertext encodes a single $\mu \in \mathbb{Z}_p$ as $n{+}1$ elements of $\mathbb{Z}_q$, for expansion $(n{+}1)\log q/\log p$. An RLWE ciphertext encodes a ring element $\mu \in R_p$ as two elements of R_q, for expansion $2\log q/\log p$—a factor of $(n{+}1)/2$ smaller, roughly $1000\times$ at $n \approx 2^{11}$.

Fast multiplication via the NTT. When $q \equiv 1 \pmod{2d}$, we call q *NTT-friendly*: polynomial multiplication in R_q factors through the Number Theoretic Transform (NTT) and its inverse (INTT) (Satriawan et al., 2023), turning each $O(d^2)$ negacyclic convolution into: NTT ($O(d \log d)$), coefficient-wise multiply ($O(d)$), INTT ($O(d \log d)$)—for $O(d \log d)$ $\mathbb{Z}_q$ operations overall. This round trip is what makes RLWE-based homomorphic operations feasible at deployment-scale parameters. However, for a power-of-two d, the congruence $q \equiv 1 \pmod{2d}$ forces q to be odd, so q cannot be a power of two: NTT-friendly arithmetic does not overlap with the free wrap-around of word ($2^{32}/2^{64}$) moduli.

2.7 Working with LWE and RLWE Encryption

$\mathcal{E}_{\mathrm{LWE}}$ and $\mathcal{E}_{\mathrm{RLWE}}$ share a small set of mechanics that every lattice-PIR scheme in this thesis builds on: linear homomorphism, a normal-form convention, RLWE-to-LWE coefficient extraction, common-reference-string randomness, concrete-security accounting, and an additive noise budget. We collect them here.

Linear homomorphism. For LWE ciphertexts $\mathrm{ct}_i = (\mathbf{a}_i, b_i)$ encrypting μ_i under $\mathbf{s}$ and weights $w_i \in \mathbb{Z}_q$,

$$\sum_i w_i \cdot \mathrm{ct}_i \;=\; \Big(\sum_i w_i \mathbf{a}_i,\; \sum_i w_i b_i\Big),$$

which decrypts to $\sum_i w_i \mu_i$ with error $\sum_i w_i e_i$. RLWE ciphertexts $(a_i, b_i) \in R_q^2$ are linearly homomorphic over R_q under the same rules, with the additional capability that the scalar w may be a *ring* element rather than only an integer—this is the ingredient that makes server-side ring automorphisms (and thus LWE-to-RLWE packing) possible. In both cases parameters must be chosen so the aggregated noise stays below the decryption threshold.

Normal form. For both LWE and RLWE, the *normal-form* variant—secret drawn from the error distribution rather than uniformly—is at least as hard as the uniform-secret variant up to a small change in the number of samples (Applebaum et al., 2009). The schemes that apply homomorphic automorphisms to RLWE ciphertexts (HintlessPIR, YPIR, and InsPIRe; Section 2.10) sample secrets from χ in normal form, since a small secret tightens the automorphism noise bound. SimplePIR and Tiptoe sample uniformly from $\mathbb{Z}_q^n$, the form in which Regev originally stated LWE.

Coefficient extraction (RLWE $\to$ LWE). A single RLWE ciphertext $(a, b) \in R_q^2$ encrypting $\mu = \sum_j \mu_j x^j \in R_p$ under s is simultaneously d scalar LWE ciphertexts encrypting the μ_j under $\mathbf{s} = \mathsf{Coeffs}(s) \in \mathbb{Z}_q^d$. The mask vectors come straight from $\mathsf{NCyclicMat}(a)$ (Section 2.6): row j is the LWE mask $\mathbf{a}^{(j)} \in \mathbb{Z}_q^d$ such that

$$b_j \;=\; \langle \mathbf{a}^{(j)}, \mathbf{s} \rangle \;+\; e_j \;+\; \lfloor q/p \rfloor \cdot \mu_j.$$

Extraction is exact—it introduces no additional noise—and is the bridge between RLWE's polynomials and LWE's scalars.

Common Reference String (CRS) model. Every PIR scheme in this thesis operates in the *Common Reference String* (CRS) model, which rests on two ideas: a hardness justification and a communication-saving construction. The hardness justification: the LWE problem remains hard when the matrix $\mathbf{A}$ is reused across many trials, each using an independently sampled secret (Peikert et al., 2008, Lemma 7.3); analogous statements hold for the a-polynomials of RLWE. This licenses the construction: $\mathbf{A}$ (or the RLWE a-polynomials) is derived as the output of a public hash function applied in counter mode to a fixed short seed (e.g., the first 256 bits of π run through `ChaCha20`), modeled as a random oracle, so client and server agree on and store the seed rather than the full $\mathbf{A}$. Per-query communication of every scheme that follows is reduced by the size of this message-independent randomness.

Concrete security. Concrete bit-security of LWE is assessed via the Lattice Estimator (Albrecht et al., 2015), which enumerates the known attack families and reports the bits of security they achieve against the given parameters. The Estimator treats RLWE instances as LWE instances per coefficient extraction and applies the same cryptanalysis.

The independence heuristic. Like all lattice-based PIR constructions of interest to this thesis, we adopt the *independence heuristic*: noise terms arising in intermediate homomorphic computations are modeled as mutually independent sub-Gaussians (Section 2.1), even when they are formally correlated. Variance is then additive, so k noise terms of common bound B aggregate to $\sqrt{k}\,B$ rather than kB.

Failure probability and the noise budget. Each homomorphic operation accumulates error. Given total noise $\mathcal{N}$ with sub-Gaussian parameter $\sigma_{\mathcal{N}}$ and decryption threshold $T = \lfloor q/p \rfloor/2$, decryption fails with probability at most $2\exp(-\pi T^2/\sigma_{\mathcal{N}}^2)$. Our target throughout this thesis is $\log_2(\delta) < -40$, achieved by choosing parameters

so that $T/\sigma_{\mathcal{N}} > \sqrt{40 \ln 2/\pi} \approx 2.97$; this *noise budget* is the central design constraint of every lattice-based scheme in this thesis.

Modulus switching. The modulus-switching operator $\lfloor \cdot \rceil_{q,q'}$ of Section 2.1, applied coefficient-wise to an RLWE ciphertext, rescales R_q^2 to $R_{q'}^2$ without re-encrypting and satisfies $\lfloor a \rceil_{q,q'} = (q'/q) \cdot a + \epsilon_a \pmod{q'}$ with $\|\epsilon_a\|_\infty \leq 1/2$.

Lemma 2.2 (Single-modulus switch)**.** *Let $(a, b) \in R_q^2$ be an RLWE ciphertext satisfying $b - a \cdot s = \lfloor q/p \rfloor \cdot \mu + e \pmod{q}$ for some $s \in R_q$, error $e \in R$, and $\mu \in R_p$. Let $(a', b') = (\lfloor a \rceil_{q,q'}, \lfloor b \rceil_{q,q'}) \in R_{q'}^2$. Then*

$$b' - a' \cdot s \;=\; \lfloor q'/p \rfloor \cdot \mu + e'_1 + e'_2 \pmod{q'},$$

where e'_1 is a bounded deterministic message-rescaling error and e'_2 absorbs the rescaled ciphertext noise plus a fresh rounding term. Decryption succeeds whenever $\|e'_1 + e'_2\|_\infty < \lfloor q'/p \rfloor/2$; we defer concrete bounds on both to Chapter 4.

Taking $q' \ll q$ shrinks the download by $\log_2(q/q')$ bits per ciphertext coefficient.

Why these are the right ciphers for LHE. $\mathcal{E}_{\mathrm{LWE}}$ and $\mathcal{E}_{\mathrm{RLWE}}$ together give the most concretely-efficient linearly homomorphic ciphers known. Each ciphertext element is a $\mathbb{Z}_q$ scalar of $\log q$ bits; with preprocessing (Section 2.3), online evaluation of a linear f is essentially as fast as plaintext evaluation—just f computed over $\mathbb{Z}_q$ instead of $\mathbb{Z}_p$, with cost independent of the security parameter. Paillier and ElGamal, the older linearly homomorphic schemes, pay a factor proportional to their security parameter, i.e., λ or λ^2 on every operation, which dominates the $\Omega(N)$ scan of a multi-gigabyte database.

2.8 SimplePIR

Lattice-PIR template. Every lattice-PIR scheme below extends the Square-Root PIR template (Section 2.3.1), varying along three axes: which LHEwP instance ($\mathcal{E}_{\mathrm{LWE}}$ vs. $\mathcal{E}_{\mathrm{RLWE}}$); where the function hint $\mathbf{H} = \mathbf{A} \cdot \mathrm{db}$ lives (hint_c in SimplePIR, hint_s in Tiptoe, absorbed into composable preprocessing in HintlessPIR/YPIR/InsPIRe); and whether Apply is composed with extra ciphertext operations (mask bootstrapping, ring packing, modulus switching) before Recover.

SimplePIR (Henzinger et al., 2023b) was the first single-server PIR to push throughput to the memory bandwidth of the machine. As a PIR-with-preprocessing scheme, it instantiates $\mathcal{E}_{\mathrm{LWE}}$ with a non-trivial Preproc: the function hint $\mathbf{H} = \mathbf{A} \cdot \mathrm{db}$ is handed to the client as hint_c once, offline. Query, Answer, and Recover then run online per query without the random portion.

Setup. The database of N records (each $\mathcal{I}$ bits) is reshaped as $\mathrm{db} \in \mathbb{Z}_p^{\ell_1 \times \ell_2}$ with $\ell_1 \ell_2 \log_2 p = N\mathcal{I}$. The matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times \ell_1}$ is generated from the public CRS seed (Section 2.7), so only the seed—not $\mathbf{A}$—is shared between client and server. The server computes the hint once, offline:

$$\mathbf{H} \;=\; \mathbf{A} \cdot \mathrm{db} \;\in\; \mathbb{Z}_q^{n \times \ell_2}, \qquad \mathrm{hint}_s = \bot, \quad \mathrm{hint}_c = \mathbf{H}.$$

The hint is downloaded once per client and reused for every subsequent query.

Query, Answer, Recover. To retrieve row $i^\star$, the client forms the indicator vector $\mathbf{u}_{i^\star} \in \mathbb{Z}_p^{\ell_1}$ and sends $\mathrm{qu} = \mathsf{Enc}(\mathrm{sk}, \mathbf{u}_{i^\star}) = \mathbf{A}^T\mathbf{s} + \mathbf{e} + \lfloor q/p \rfloor \, \mathbf{u}_{i^\star} \in \mathbb{Z}_q^{\ell_1}$. The server computes

$$\mathbf{r} \;=\; \mathrm{db}^T \cdot \mathrm{qu} \;\in\; \mathbb{Z}_q^{\ell_2}.$$

For each column j,

$$r_j \;=\; \langle \mathbf{H}_j, \mathbf{s} \rangle \;+\; \textstyle\sum_i \mathrm{db}[i,j] \cdot e_i \;+\; \lfloor q/p \rfloor \cdot \mathrm{db}[i^\star, j] \pmod q,$$

so $(\mathbf{H}_j, r_j)$ is an LWE ciphertext encrypting $\mathrm{db}[i^\star, j]$ under $\mathbf{s}$. The client uses the cached hint $\mathbf{H}$ to $\mathsf{Recover}$ each row $\mathrm{db}[i^\star, j]$ via LWE decryption.

Noise accumulation. The output ciphertext carries the original query error $\mathbf{e}$ weighted by the $i^\star$-th database column: the stochastic part of r_j is $\sum_i \mathrm{db}[i, j] \cdot e_i$, which under the independence heuristic (Section 2.7) is sub-Gaussian with parameter $\sigma_e \cdot \sqrt{\sum_i \mathrm{db}[i, j]^2}$. This is the first instance of a pattern central to every lattice-PIR scheme: *each homomorphic operation inflates the noise*, parameters must be sized so the accumulated noise stays below the decryption threshold from Section 2.5.1, and any scheme that does more homomorphic computation on the client's query than SimplePIR further inflates the budget. We suppress noise analysis for the remaining scheme sections and defer the concrete accounting for our construction to Chapter 4.

Communication and throughput. Per-query upload is $\ell_1 \log q$ bits; per-query response is $\ell_2 \log q$ bits. The one-time hint is $n \cdot \ell_2 \cdot \log q$ bits: $\sim$1 GiB at $n = 2^{11}$, $q = 2^{32}$, $\ell_2 = 2^{17}$ (for the 8 GiB Wikipedia database). Server work is a single $\mathbb{Z}_q$ matrix–vector product that scans every byte of db; with tight parameters ($q \approx 2^{32}$, $p = 256$), each plaintext byte costs one 32-bit multiply-add, and throughput approaches the memory bandwidth (10–14 GiB/s/core on commodity CPUs).

The hint is the catch. SimplePIR's large hint is onerous in three settings: dynamic databases (every update forces a re-download), multi-database clients (each database needs its own hint), and short-lived clients (the hint may exceed total query traffic). Additionally, the hint may be intractably large, even approaching or exceeding the size of the database for some instantiations. In cloud deployments the hint is also a distinct egress charge: at AWS's standard \$0.09/GB outbound rate, a $\sim$1 GiB hint costs $\sim$\$0.09 per client, which for 10^6 unique clients is $\sim$\$90,000—over three orders of magnitude above the compute cost of serving them (Chapter 5, Ta-

ble 5.4). Every lattice-PIR scheme that follows is an attempt to keep SimplePIR's throughput while shrinking or eliminating hint_c.

2.9 Tiptoe: Bootstrapping the Hint

Tiptoe (Henzinger et al., 2023a) was the first scheme to eliminate the offline hint download of SimplePIR. As a PIR-with-preprocessing scheme, it keeps the LHEwP hint server-side ($\text{hint}_s = \mathbf{H}$) and absorbs the per-query mask computation $\mathbf{s}^T\mathbf{H}$ into Apply using client-uploaded *bootstrapping keys*—outer-scheme encryptions of the inner LWE secret that let the server homomorphically evaluate the SimplePIR decryption mask, one RLWE ciphertext per coordinate of $\mathbf{s} \in \mathbb{Z}_q^n$. Concretely, the client samples a fresh RLWE secret $\tilde{s} \in R_q$ and uploads n ciphertexts $(\bar{a}_k, \bar{b}_k) \in R_q^2$, where $(\bar{a}_k, \bar{b}_k)$ encrypts the constant polynomial $\mathbf{s}[k]$ (equivalently, the length-d slot vector with $\mathbf{s}[k]$ in every slot):

$$\bar{b}_k \;=\; \bar{a}_k \cdot \tilde{s} + \bar{e}_k + \lfloor q/p \rfloor \cdot \mathbf{s}[k] \qquad (k \in [n]).$$

After the standard scan $\mathbf{r} = \mathrm{db}^T\mathrm{qu} \in \mathbb{Z}_q^{\ell_2}$, the server processes $\mathbf{H} \in \mathbb{Z}_q^{n \times \ell_2}$ in column chunks of width d (the outer scheme's slot count). For each chunk J, it encodes the row chunks $\mathbf{H}[k,\, Jd{:}(J{+}1)d]$ as length-d plaintext slot vectors, multiplies each against $(\bar{a}_k, \bar{b}_k)$ via the outer scheme's plaintext–ciphertext product, and sums across k to obtain one outer RLWE ciphertext encrypting d consecutive entries of $\mathbf{s}^T\mathbf{H}$ in its slots. The server returns $\mathbf{r}$ along with the $\lceil \ell_2/d \rceil$ outer ciphertexts. The client decrypts under $\tilde{s}$ to recover $\mathbf{s}^T\mathbf{H}$, subtracts from $\mathbf{r}$, and rounds.

Cost. Tiptoe's scan matches SimplePIR, but the $\mathbf{s}^T\mathbf{H}$ homomorphic evaluation is structurally heavy: the outer encryption of $\mathbf{s}$ must support homomorphic operations over the LWE ciphertext modulus $\mathbb{Z}_q$, forcing concretely larger RLWE parameters. Tiptoe achieves only $\sim$14% of SimplePIR's throughput at 35$\times$ larger upload.

2.10 Gadgets and Homomorphic Automorphisms

The remaining schemes—HintlessPIR, YPIR, and InsPIRe—all rely on homomorphic *ring automorphisms*: applying a structured permutation to the encrypted polynomial without decrypting. The mechanic underneath is *gadget decomposition*, which keeps noise growth manageable when an RLWE ciphertext is multiplied by a large polynomial.

Gadget matrices. For a modulus $q \in \mathbb{N}$ and a decomposition base $z \in \mathbb{N}$ with $z \ll q$, the gadget vector is $\mathbf{g}_z = [1, z, z^2, \ldots, z^{t-1}] \in \mathbb{Z}_q^t$ (row vector) where $t = \lceil \log q / \log z \rceil$, and the gadget matrix is $\mathbf{G}_{n,z} = \mathbf{I}_n \otimes \mathbf{g}_z^T \in \mathbb{Z}_q^{nt \times n}$. The base-$z$ digit-decomposition operator $\mathbf{G}_{n,z}^{-1} : \mathbb{Z}_q^n \to \mathbb{Z}^{nt}$ expands each component of its input into its base-z representation, with digits in $(-z/2, z/2]$, and satisfies $\mathbf{G}_{n,z}^T \cdot \mathbf{G}_{n,z}^{-1}(\mathbf{x}) = \mathbf{x}$ for all $\mathbf{x} \in \mathbb{Z}_q^n$; the 1-dimensional version is $g_z^{-1} = \mathbf{G}_{1,z}^{-1} : \mathbb{Z}_q \to \mathbb{Z}^t$. Both extend to matrices column-wise, and to R_q identically. Gadget decomposition replaces a single multiplication by a large $a \in \mathbb{Z}_q$ with t multiplications by small digits in $(-z/2, z/2]$—the same technique used in a modern fully-homomorphic encryption construction (Gentry et al., 2013).

Cyclotomic ring automorphisms. For every $g \in (\mathbb{Z}/2d)^*$, the map $\tau_g : R \to R$ defined by $\tau_g\big(p(x)\big) = p(x^g)$ is a ring automorphism; we use the same name τ_g on R_q and extend it to vectors and matrices component-wise.

Homomorphic evaluation of automorphisms. An RLWE ciphertext encrypting μ under secret s can be re-encrypted as $\tau(\mu)$ under the same s, for any target automorphism $\tau = \tau_g$, via a precomputed automorphism key (Menon and Wu, 2024).

Construction 2.1 (Automorphisms on RLWE encodings, adapted from Menon and Wu (2024))**.** Let λ be a security parameter, d a power-of-two ring degree, q a ciphertext modulus, and χ an error distribution over $R = \mathbb{Z}[x]/(x^d + 1)$.

- $\mathsf{AutomorphSetup}(1^\lambda, s, \tau, z)$: On input a secret $s \in R_q$, a target automorphism $\tau\colon R_q \to R_q$, and a decomposition base $z \in \mathbb{N}$ (with gadget depth $t = \lceil \log q / \log z \rceil$), sample $\mathbf{a} \xleftarrow{R} R_q^t$ and $\mathbf{e} \leftarrow \chi^t$, and output

$$\mathbf{W}_\tau \;=\; \begin{bmatrix} \mathbf{a}^T \\ s\mathbf{a}^T + \mathbf{e}^T - \tau(s) \cdot \mathbf{g}_z \end{bmatrix} \;\in\; R_q^{2\times t}.$$

- $\mathsf{Automorph}(\mathbf{W}_\tau, \mathbf{c}, \tau, z)$: On input an automorphism key $\mathbf{W}_\tau$, an RLWE ciphertext $\mathbf{c} = (c_0, c_1) \in R_q^2$ encrypting μ, the automorphism τ, and base z, output

$$\mathbf{c}' \;=\; \mathbf{W}_\tau \cdot g_z^{-1}\big(\tau(c_0)\big) + \begin{bmatrix} 0 \\ \tau(c_1) \end{bmatrix} \;\in\; R_q^2. \tag{2.4}$$

  The output satisfies $[-s \mid 1] \cdot \mathbf{c}' = \tau(\mu) + e'$, with e' bounded in terms of the ambient noise and gadget parameters (deferred to Chapter 4).

Each step can run in the NTT domain: τ itself is just a permutation of slot indices there, and the $\mathbf{W}_\tau$-times-gadget-decomposition is a coefficient-wise multiply that the NTT representation makes cheap. This is the core mechanic of CDKS ring packing (Section 2.12.2), InspiRING (Section 2.14), and HintlessPIR's slot-algebra improvement over Tiptoe (Section 2.11).

2.11 HintlessPIR: Cheaper Bootstrapping

HintlessPIR (Li et al., 2024) also keeps the LHEwP hint server-side ($\text{hint}_s = \mathbf{H}$), but generalizes LHEwP to *homomorphic encryption with composable preprocessing*: hint_s now captures a chain of operations on ciphertexts—linear functions and non-linear ones, namely automorphism evaluation and the gadget-based key-switching that supports it—rather than a single linear f. HintlessPIR uses this to compute the SimplePIR mask $\mathbf{s}^T\mathbf{H}$ at much lower upload cost than Tiptoe's two-layer construction.

Composable preprocessing. HintlessPIR extends LHEwP's hint_s to capture not just a single function f but *chains* of ciphertext operations. The construction rests

on a structural property of RLWE: every RLWE-based ciphertext (a, b) splits into a public-randomness half a (derived from the CRS seed, known to the server before any query) and a message-dependent half b. For the operations relevant to PIR—linear matvecs, automorphism evaluation—the public-randomness half of an output ciphertext is a deterministic function of the public-randomness halves of its inputs, not of the message. This means hint_s extends to chains: for $f = f_2 \circ f_1$, the offline phase walks f_1's deterministic public output through $\mathsf{Preproc}_{f_2}$, caching a single hint_s for the whole chain. The online $\mathsf{Apply}(\text{pp}, f, \text{hint}_s, \text{ct})$ then operates only on the query's message-dependent half b.

The silent-preprocessing schemes that follow (YPIR, InsPIRe) inherit this idea to absorb their entire post-scan chain—ring packing, key-switching, modulus switching—into a single offline state hint_s depending only on the CRS and the database, leaving $\text{hint}_c = \perp$.

HintlessPIR's protocol. HintlessPIR computes the SimplePIR mask $\mathbf{s}^T\mathbf{H}$ homomorphically using *slot algebra*. Under the NTT, an RLWE ciphertext $(a, b) \in R_q^2$ behaves as d independent slot ciphertexts: a length-d plaintext vector $\mathbf{m} \in \mathbb{Z}_p^d$ encoded as a polynomial admits component-wise multiplications and additions directly on the ciphertext (one polynomial multiplication, no key switching), and a cyclic shift of $\mathbf{m}$ within its slots lifts to the Galois automorphism $X \mapsto X^{5^k}$ on the underlying polynomial—realized homomorphically by one key-switching against a rotation key from $\tau_5(\tilde{s})$ back to $\tilde{s}$. The mask matvec $\mathbf{s}^T\mathbf{H}$ then uses the diagonally-dominant algorithm: an $n \times d$ block of $\mathbf{H}$ decomposes into n generalized diagonals $\mathbf{H}^{(i)}_{\text{diag}}[j] = \mathbf{H}[i, (i + j) \bmod d]$, and the slot-wise sum

$$(\mathbf{s}^T\mathbf{H})[j] \;=\; \sum_{i=0}^{n-1} \tau_5^i(\mathbf{s})[j] \cdot \mathbf{H}^{(i)}_{\text{diag}}[j]$$

evaluates the matvec using just the encryption of $\mathbf{s}$ and one rotation key, with the n rotations applied iteratively (τ_5^i from τ_5^{i-1}).

The client uploads, alongside the SimplePIR query, a single RLWE ciphertext encrypting all n coordinates of $\mathbf{s}$ in slots under a fresh RLWE secret $\tilde{s}$, plus the rotation key. This requires $d \geq 2n$, since τ_5 cycles each half of the d slots independently and the LWE secret must fit in one half. After the SimplePIR scan $\mathbf{r} = \mathrm{db}^T \mathrm{qu} \in \mathbb{Z}_q^{\ell_2}$, the server processes $\mathbf{H}$ in $n \times d$ column chunks; within each chunk it runs the diagonally-dominant matvec above, producing one packed RLWE ciphertext per chunk encrypting that slice of $\mathbf{s}^T\mathbf{H}$, for $\lceil \ell_2/d \rceil$ ciphertexts total. The server returns these alongside $\mathbf{r}$; the client decrypts under $\tilde{s}$, subtracts the recovered mask values from $\mathbf{r}$, and rounds.

The mask computation runs n rotation steps in sequence. Without preprocessing each step costs $O(t\, d \log d)$ (gadget-decomposing the input's public-randomness half and NTT-ing each digit). Composable preprocessing computes the NTT-domain gadget decompositions of all intermediate public halves once offline (they are deterministic given the CRS and rotation key), reducing each online rotation step to a $(t{+}2)d$-operation combine against the cached NTTs.

Cost. HintlessPIR is faster and requires less communication per its single bootstrapping key than Tiptoe, but still reaches at most 62% of SimplePIR's throughput at $4\times$ its online communication, inheriting Tiptoe's structural issue (the outer RLWE scheme must encrypt $\mathbb{Z}_q$-sized messages—the inner LWE's ciphertext modulus—forcing concretely larger RLWE parameters) and paying for its communication savings via sequential key-switching.

2.12 Ring Packing and Modulus Switching

YPIR (Menon and Wu, 2024) added two primitives for the lattice-PIR setting that together compress the response of any SimplePIR-family scheme: *LWE-to-RLWE ring packing*, which compresses many scalar LWE ciphertexts into a single RLWE ciphertext, and a *two-target modulus switch*, which trims the resulting RLWE.

We develop both here, then describe the YPIR scheme that uses them in Section 2.13.

2.12.1 LWE-to-RLWE Ring Packing

After a SimplePIR-family scan, the server holds ℓ_2 scalar LWE ciphertexts $\{(\mathbf{a}_j, b_j) \in \mathbb{Z}_q^{d+1}\}_{j\in[\ell_2]}$ encrypting $\mu_j \in \mathbb{Z}_p$ under a common secret $\mathbf{s} = \mathsf{Coeffs}(s) \in \mathbb{Z}_q^d$ that is the coefficient vector of an RLWE secret $s \in R_q$; this alignment between LWE and RLWE dimensions is what enables packing. Packing groups the ciphertexts into chunks of d and compresses each chunk into a single RLWE ciphertext in R_q^2, shrinking the response from $\ell_2(d{+}1)\log q$ bits to $2\ell_2 \log q$ bits—a factor of $(d{+}1)/2 \approx 1000\times$ smaller at $d = 2^{11}$.

Definition 2.7 (LWE-to-RLWE packing)**.** Given d scalar LWE ciphertexts $(\mathbf{a}_j, b_j)$ encrypting μ_j under a common secret that equals $\mathsf{Coeffs}(s)$ for some $s \in R_q$, produce a single RLWE ciphertext encrypting $\sum_{j=0}^{d-1} \mu_j x^j$ under s.

Remark 2.1 (Why not run SimplePIR over RLWE end-to-end?)*.* If our goal is RLWE-rate communication, a natural question is why we pack at all—why not encrypt the SimplePIR query as an RLWE ciphertext directly and run the scan as a slot-algebra matrix-multiply? Three obstacles.

Communication blow-up. In any linear PIR scheme over a database $D \in \mathbb{Z}_p^{\ell_1\times\ell_2}$, upload $\times$ download $\geq \ell_1\ell_2$ scalars. There are exactly two ways to lay D out over R_p for an RLWE-only protocol, and each pays a factor of d on one axis: *group d rows into one polynomial* (upload ℓ_1 scalars, download $d\cdot\ell_2$ scalars—d items returned when only one was asked for), or *group d columns into one polynomial* (upload $d\cdot\ell_1$ scalars, download ℓ_2 scalars—query is $d\times$ larger). Either way the product inflates from $\ell_1\ell_2$ to $d\cdot\ell_1\ell_2$.

NTT-friendly modulus required. For fast online evaluation both query and database must live in NTT representation, requiring an NTT-friendly modulus q (a prime $\equiv 1 \bmod 2d$) or an FFT over $\mathbb{C}$—both slower than the power-of-two-modulus arithmetic LWE-based scans use natively (Menon and Wu, 2024).

Database storage blow-up. Storing the database in NTT form expands its in-memory size by $\log q / \log p$: each $\log p$-bit plaintext entry becomes a $\log q$-bit NTT slot. Since SimplePIR-family schemes are memory-bandwidth-bound, this storage inflation maps directly to throughput. Menon and Wu (2024) measure a 3.6× slowdown at $\log q / \log p = 3.5$.

Ring packing sidesteps obstacles 1 and 3 by running the scan in scalar LWE—producing ℓ_2 ciphertexts of the right size in the power-of-two integer ring, with no NTT-storage tax on the database—and only then *compressing d at a time* into one RLWE ciphertext for transmission. Obstacle 2 (NTT-friendly modulus) is unavoidable in the packing step itself, but packing operates on the post-scan ciphertexts rather than on the full database, so its cost amortizes away from the per-byte work.

2.12.2 CDKS Packing

CDKS packing (Chen et al., 2021) implements Definition 2.7 by homomorphically evaluating the *field trace* of the cyclotomic ring (defined below). Lifting an LWE ciphertext $(\mathbf{a}, b) \in \mathbb{Z}_q^{d+1}$ to an RLWE ciphertext $(\iota(\mathbf{a}), b) \in R_q^2$—where $\iota(\mathbf{a}) = \sum_i a_i x^i$ embeds the mask vector as a polynomial—yields a ciphertext whose phase decrypts to a polynomial whose constant term is the LWE message μ but whose other $d - 1$ coefficients are uncontrolled junk. Packing is the problem of *annihilating* those junk coefficients while preserving the constant term, then re-using the now-cleaned ciphertext as one of the d slots of an output RLWE ciphertext.

Galois group and trace. The automorphisms $\{\tau_g : g \in (\mathbb{Z}/2d)^*\}$ form a group of size d under composition—the Galois group of R. The *field trace* $\mathrm{Tr}(p) = \sum_g \tau_g(p)$ sums over every group element; on the power basis it acts as

$$\mathrm{Tr}(x^i) \;=\; \begin{cases} d & i = 0, \\ 0 & 0 < i < d, \end{cases} \tag{2.5}$$

so applying Tr to the lifted RLWE ciphertext kills every coefficient but the constant (up to the factor of d, easily absorbed into the INTT scaling).

Log-depth trace evaluation (CDKS). Direct evaluation sums over all d automorphisms, costing d key-switchings. CDKS exploits the recursive structure of the group to evaluate the trace via a divide-and-conquer tree of $\log_2 d$ levels. Packing layers this across d inputs at once: each of $\log_2 d$ levels pairs adjacent ciphertexts and applies $\mathrm{ct} \mapsto \mathrm{ct} + \tau(\mathrm{ct})$ via Eq. 2.4 for an appropriate τ, annihilating junk coefficients while the constants coalesce into the d slots of a single output RLWE encrypting $\sum_j \mu_j x^j$. Total cost: $\log_2 d$ key-switchings, with the client uploading $\log_2 d$ key-switching matrices. This is the only idea YPIR inherits from CDKS; InspiRING (Section 2.14) replaces it with a structurally different approach.

2.12.3 Two-Target Modulus Switching

Recall the single-modulus switch (Lemma 2.2, Section 2.7); we extend it here to the two-target setting that ring-packed responses admit. A packed RLWE has two output components—a CRS-derived *mask* polynomial a and a query-dependent *body* polynomial b—and we can switch each to its own target modulus, with the mask switched more aggressively because the client only needs it at the body's precision after rescaling.

Lemma 2.3 (Two-target modulus switch)**.** *Same setup as Lemma 2.2, with target moduli $q_1 \geq q_2$. Round the mask to q_1 and the body to q_2:*

$$a' = \lfloor a \rceil_{q,q_1} \in R_{q_1}, \qquad b' = \lfloor b \rceil_{q,q_2} \in R_{q_2}.$$

The client first rescales the mask from q_1 to q_2 as $a'' = \lfloor a' \rceil_{q_1,q_2} \in R_{q_2}$ and then computes

$$b' - a'' \cdot s \;=\; \lfloor q_2/p \rceil \cdot \mu + e'_1 + e'_2 \pmod{q_2},$$

where e'_1 and e'_2 play the same deterministic/stochastic roles as in Lemma 2.2; concrete bounds are deferred to Chapter 4.

The two-target switch lets the mask amortize over many queries (it is CRS-derived and identical across clients) while the body ships at a smaller modulus per

query. For our Wikipedia-size parameters ($d = 2048$, $q \approx 2^{32}$, $\ell_2/d = 64$ packed RLWEs), the post-packing response is already $64 \cdot 2 \cdot d \cdot \log q/8 \approx 1\,\text{MiB}$—ring packing did the bulk of the work, shrinking the response by roughly $(d{+}1)/2 \approx 1000\times$ from the raw $\ell_2(d{+}1)\log q$ bits of unpacked LWE ciphertexts. The two-target switch then shaves a further $\sim 2\times$, taking the mask to $q_1 = 2^{18}$ (288 KiB) and the body to $q_2 = 2^{10}$ (160 KiB).

2.13 YPIR

YPIR (Menon and Wu, 2024) is a silent-preprocessing scheme whose online Apply is the composition $h \circ g \circ f$ of three stages: the SimplePIR-style scan $f\colon \text{qu} \mapsto \text{db}^T\text{qu}$ producing ℓ_2 scalar LWE ciphertexts; CDKS ring packing g (Section 2.12.2) compressing them into $\lceil \ell_2/d \rceil$ RLWE ciphertexts; and the two-target modulus switch h (Section 2.12.3) trimming each RLWE for download. Recall composable preprocessing (Section 2.11) extends hint_s to chains of ciphertext operations whose public-randomness halves are deterministic given the CRS. YPIR applies this to the public side of both g and h: the LWE ciphertexts' public halves are deterministic functions of the CRS-derived $\mathbf{A}$ and the database (via $\mathbf{H} = \mathbf{A} \cdot \text{db}$), so the public-randomness halves of every intermediate ciphertext that g and h would emit can be precomputed into hint_s. The message side still walks the depth-$\log_2 d$ CDKS trace tree serially online—each level consumes the previous level's body—so composable preprocessing here trades expensive online NTTs for cheap online combines against cached material, but does not flatten the chain.

LHEwP instance. YPIR upgrades both Enc and $\mathsf{Preproc}$ from LWE to RLWE (the same encryption-side optimization DistPIR later exploits, Section 2.15). Enc: the client generates ℓ_1 scalar LWE ciphertexts by encrypting under RLWE and extracting coefficients (Section 2.7), then ships only the scalar LWEs; the server sees the same scalar-LWE query interface as SimplePIR. $\mathsf{Preproc}$: the hint $\mathbf{H} = \mathbf{A} \cdot \text{db}$ is computed

as an RLWE convolution rather than an LWE matmul, then walked through g and h to produce the message-independent halves of every RLWE the online server will emit. $\mathbf{H}$ itself is consumed in this offline walk and discarded for fixed databases; only the preprocessed material persists as hint_s. Menon and Wu (2024) report a 9× online speedup from this preprocessing step (Section 4.2). Apply runs the scan in the RLWE-prime modulus q (forgoing word arithmetic), then g and h against the precomputed material; Dec decrypts directly under the LWE secret via RLWE with no client-side hint. The original YPIR construction layers these onto DoublePIR (small-record retrieval); the variant we care about, YPIR+SP, applies the same packing to SimplePIR itself for large-record retrieval. We treat YPIR below as shorthand for YPIR+SP.

Performance. YPIR needs no offline communication and reduces per-query communication by 2.2× over HintlessPIR on databases with 32–64 KiB records at roughly the same throughput: the CDKS trace tree's $\log_2 d$ levels run in sequence (each level consumes the previous), so the chain of automorphisms and key-switchings runs in addition to the scan and cannot amortize into the memory-bandwidth-bound scan pass.

2.14 InsPIRe

InsPIRe (Mahdavi et al., 2025) is another silent-preprocessing scheme, designed for composable preprocessing (Section 2.11) from the ground up rather than retrofit. The full InsPIRe protocol composes SimplePIR with a new ring-packing algorithm called InspiRING and a homomorphic polynomial-interpolation step that compresses InspiRING's ℓ_2/d packed RLWEs into a single one—suited to retrieving one polynomial-sized record at a time. The InspiRING packing itself uploads only 2 key-switching matrices (vs. CDKS's $\log_2 d$), yielding smaller queries at throughput comparable to CDKS-based packing. For the large-record setting that this thesis

targets (one record spans many columns), we want *all* ℓ_2/d packed responses, so we adopt InspiRING but not the polynomial-interpolation step—a configuration not presented in the InsPIRe paper, and part of this thesis's contribution. The rest of this section develops the packing algorithm.

InspiRING packing. InspiRING, like CDKS, implements Definition 2.7 by homomorphically evaluating the field trace (Eq. 2.5); the two differ in how they factor the trace sum over the Galois group. InspiRING is structurally a different algorithm than CDKS, designed for the CRS model from the start. Where CDKS was originally formulated without preprocessing in mind (YPIR retrofits composable preprocessing on top), InspiRING explicitly arranges its three stages so that the bulk of every random-component computation can be moved offline. Client upload drops from $\log_2 d$ key-switching matrices (CDKS) to just 2: the Galois group $(\mathbb{Z}/2d)^*$ is generated by two automorphisms τ_5 and τ_{2d-1} (it is isomorphic to $\mathbb{Z}_{d/2} \times \mathbb{Z}_2$), and the trace factors as $\mathrm{Tr}(p) = \sum_{j=0}^{d/2-1} \tau_5^j(p) + \tau_{2d-1}\tau_5^j(p)$, so every rotation needed by the iterative collapse is an automorphic image of one of two base key-switching matrices (Mahdavi et al., 2025, Appendix, Part 4).

Construction 2.2 (InspiRING packing (Mahdavi et al., 2025))**.** Let λ be a security parameter, d a power-of-two ring degree, q an NTT-friendly modulus, $z \in \mathbb{N}$ a gadget base with depth $t = \lceil \log q / \log z \rceil$, and τ_5, τ_{2d-1} the two generators of the Galois group of $R_q = \mathbb{Z}_q[x]/(x^d + 1)$. Let $(\mathsf{AutomorphSetup}, \mathsf{Automorph})$ be the primitives from Construction 2.1.

- $\mathsf{InspiRING.Setup}(1^\lambda, \tilde{s}, z)$: On input the security parameter, the client's RLWE secret $\tilde{s} \in R_q$, and the gadget base z, output the *packing key* $\mathrm{pk} = (\mathbf{K}_g, \mathbf{K}_h)$ with

$$\mathbf{K}_g \leftarrow \mathsf{AutomorphSetup}(1^\lambda, \tilde{s}, \tau_5, z), \qquad \mathbf{K}_h \leftarrow \mathsf{AutomorphSetup}(1^\lambda, \tilde{s}, \tau_{2d-1}, z).$$

- $\mathsf{InspiRING.Pack}(\mathrm{pk}, \{(\mathbf{a}_k, b_k)\}_{k=0}^{d-1})$: On input pk and d scalar LWE ciphertexts encrypting messages $m_0, \ldots, m_{d-1}$ under $\mathbf{s} = \mathsf{Coeffs}(\tilde{s})$, output a single RLWE ciphertext $(a_{\mathrm{fin}}, b_{\mathrm{fin}}) \in R_q^2$ encrypting $\sum_{k=0}^{d-1} m_k x^k$ under $\tilde{s}$. The computation proceeds in three stages:

  1. *Transform.* For each k, lift $(\mathbf{a}_k, b_k)$ to an intermediate ciphertext $(\hat{\mathbf{a}}_k, \tilde{b}_k) \in R_q^d \times R_q$, where $\hat{\mathbf{a}}_k[j] = d^{-1}\tau_5^j(\tilde{a}_k)$ and $\hat{\mathbf{a}}_k[j + d/2] = d^{-1}\tau_{2d-1}\tau_5^j(\tilde{a}_k)$ for $j < d/2$, with $\tilde{a}_k = \sum_i \mathbf{a}_k[i]x^{-i}$, and $\tilde{b}_k = b_k$ encrypts the constant polynomial m_k under the module secret $\hat{\mathbf{s}} \in R_q^d$ (built from $\tilde{s}$ by the same automorphisms that produce $\hat{\mathbf{a}}_k$).
  2. *Aggregate.* $(\hat{\mathbf{a}}_{\mathrm{agg}}, \tilde{b}_{\mathrm{agg}}) \leftarrow \sum_{k=0}^{d-1}(\hat{\mathbf{a}}_k, \tilde{b}_k) \cdot x^k$, an intermediate ciphertext encrypting $\sum_k m_k x^k$.
  3. *Collapse.* Iteratively reduce the d-component random vector $\hat{\mathbf{a}}_{\mathrm{agg}}$ to a single RLWE mask via $d - 2$ applications of automorphic images of $\mathbf{K}_g$ (one per τ_5-orbit step) and a final application of $\mathbf{K}_h$ to fold the τ_{2d-1}-twisted half. Return $(a_{\mathrm{fin}}, b_{\mathrm{fin}}) \in R_q^2$.

Under the CRS model, the random components of all intermediate ciphertexts in $\mathsf{Transform}$, $\mathsf{Aggregate}$, and the bulk of $\mathsf{Collapse}$ depend only on the LWE ciphertexts' $\mathbf{a}$-vectors and on $(\mathbf{K}_g, \mathbf{K}_h)$—all CRS-derived—so they can be precomputed offline, leaving online cost $O(td^2)$ $\mathbb{Z}_q$ operations against the offline-prepared material (vs. $O(d^3 + td^2 \log d)$ offline).

InspiRING packing noise. InspiRING incurs less noise growth from its packing routine than CDKS does: the factored two-generator trace applies each base key-switching matrix sequentially rather than recursively combining independent accumulations, avoiding the tree-style noise doubling of the binary-tree CDKS. The slack can be spent on a larger plaintext modulus p (more bits per record coefficient, lowering ℓ_2) or on more aggressive modulus switching in the response (smaller q_1, q_2 in Lemma 2.3, shrinking the download).

2.15 DistPIR: GPU at a Cost

The DistributionalPIR paper (Lehmkuhl et al., 2025)—whose titular "distributional" scheme is a different contribution—introduces the lattice-PIR optimizations and GPU acceleration, hitting ∼2 TiB/s on an NVIDIA V100. We refer to this PIR construction (and the benchmarks the paper reports for it) as DistPIR. DistPIR shares YPIR's RLWE-side encryption and preprocessing, but adds one step YPIR declined: a modulus switch *down to a word modulus* before the scan, recovering the free wrap-around arithmetic that NTT-friendly primes forbid (Section 2.6.2).

DistPIR's modulus-switch-before-evaluation. SimplePIR's LWE-based Enc and Preproc are concretely heavy: each query encrypts an ℓ_1-vector under an LWE secret of dimension $d \approx 2^{11}$ ($d \cdot \ell_1$ word ops), and the offline hint is a $d \times \ell_1 \cdot \ell_2 = d \cdot N$ matmul. Switching to RLWE cuts both by $\sim d/\log d$ (as YPIR does), but YPIR then runs Apply directly in the RLWE-prime modulus q, forgoing word arithmetic. DistPIR's twist is to have the client encrypt under RLWE with a prime modulus just above 2^{32} and then apply the single-modulus switch (Lemma 2.2) to rescale the ciphertext down to $\mathbb{Z}_{2^{32}}$ *before* homomorphic evaluation (rather than after, as in prior modswitch usages); preprocessing is symmetric, computed in RLWE and modswitched to $\mathbb{Z}_{2^{32}}$ for storage. The resulting Apply is a plain integer GEMM mod 2^{32}, yielding both the $\sim d/\log d$ encryption/preprocessing speedup of RLWE and the cheap word-modulus evaluation of LWE: a $> 100\times$ speedup on Enc and Preproc over SimplePIR with no measurable hit to Apply (Table 2 there).

GPU offload for Apply. The LWE matvec in Apply is an integer GEMM mod 2^{32}, and "the cost of homomorphic operations is the same as the underlying plaintext operations up to a change in word size." DistPIR offloads this to GPU through standard matrix-multiplication libraries along the SIMT CUDA-core path, leaving tensor cores unused. The ∼2 TiB/s headline is the batched scan alone on a V100 ($3.06/hr), with cross-client batching stacking queries into a matmul to saturate

GPU compute; the CPU-cluster variant they report peaks at ∼0.80 TiB/s across 8 instances at $0.29/hr each.

Hint compression on CPU. DistPIR's authors report that "homomorphic operations on RingLWE-based ciphertexts are not efficient on GPUs," so when their scheme is paired with HintlessPIR (Section 2.11) for the stateless-client / mask-compression variant, that pairing runs on a separate *CPU cluster*. Their own measurements show this hybrid is operationally lopsided: on a 38 GiB Twitter-scale database, HintlessPIR-side hint compression takes 0.54 CPU core-seconds while the GPU scan takes only 4 ms—99.3% of compute time, and 95% of per-query dollar cost, is on the CPU side. The gap is that the only prior GPU-accelerated lattice-PIR is GPU-accelerated in name only: either the CPU-side ring-packing step sets the per-query dollar cost (the headline ∼2 TiB/s is the scan alone, with packing not counted), or the client is forced to download the full SimplePIR hint (intractably large, Section 2.8).

Chapter 3

Lattice-PIR on the GPU

Chapter 2 laid out the lattice-PIR design space. Every construction there has two compute-heavy server operations: a database matrix multiplication—the *scan*—and a ring-packing pass that compresses the response into a small number of RLWE ciphertexts. At deployment-scale batch sizes these dominate wall time. This chapter asks what it takes to place both on INT8 tensor cores—the highest-throughput compute path available on modern GPUs—and identifies five design criteria that no prior scheme satisfies simultaneously.

Cross-client batching turns the scan into a dense matmul; a roofline analysis identifies INT8 tensor cores as its cost-optimal datapath (Sections 3.1–3.3). InspiRING ring packing (Mahdavi et al., 2025), as published, is a rotation-dependent permuted gather; we show the rotations are client-independent and fold into a single offline matrix M, reducing online packing to a dense GEMM $Y \cdot M$ (Section 3.4). Section 3.5 moves the offline phase onto the GPU, compounding the $\sim$180$\times$ CPU-side speedup Lehmkuhl et al. (2025) report for RLWE convolution over LWE matmul. Section 3.6 replaces the $O(\ell_1 d)$ per-query scalar-LWE encryption with an $O(\ell_1 \log d)$ RLWE encryption followed by coefficient extraction. Section 3.7 audits the lattice-PIR family against all five criteria (offline hint, client query, scan, packing, communication): every prior scheme fails at least one.

3.1 Cross-Client Batching

A single-query SimplePIR scan performs two 32-bit arithmetic operations per byte of database read, bounding a system's performance at its memory-bandwidth

ceiling (10–14 GiB/s). Menon and Wu (2024) observe that the bottleneck is bandwidth, not arithmetic, so effective throughput improves by doing *more* arithmetic per byte.

Stacking B independent queries into $\mathbf{Q} \in \mathbb{Z}_q^{\ell_1 \times B}$ replaces the matvec $\mathrm{db}^T \cdot \mathrm{qu}$ with a matmul $R = \mathrm{db}^T \cdot \mathbf{Q}$: each byte of db now contributes to B inner products, raising arithmetic intensity from 2 ops/byte to $2B$ ops/byte without increasing the byte count. The transformation is transparent to the client; the server alone decides the batch size. Following Menon and Wu (2024), we write the *effective throughput* of a batch as $B \cdot \ell_1 \ell_2 / T$, where T is the time to answer all B queries on a database $\mathrm{db} \in \mathbb{Z}_p^{\ell_1 \times \ell_2}$. YPIR reports 17 GiB/s (1.4× over single-query SimplePIR) at $B = 4$ on a commodity CPU, saturating that hardware's compute ceiling; larger B yields no further improvement.

Batching is practical so long as the server's queue is $\gtrsim B$ deep. English Wikipedia receives ∼4,500 page views per second (Wikimedia, b); at $B = 64$ (the batch size our Wikipedia deployment uses, Chapter 6), a new batch fills in ∼14 ms, exceeding a single GPU's compute rate, so the deployment replicates across a small fleet of GPUs (Chapter 6 sizes this at 5 L4s). Once bandwidth is no longer the limit, peak integer throughput (TOPS) selects the hardware.

3.2 Roofline Analysis

The batched server operation is a matmul $\mathrm{db}^T \cdot \mathbf{Q}$ with $\mathrm{db} \in \mathbb{Z}^{\ell_1 \times \ell_2}$ and $\mathbf{Q} \in \mathbb{Z}^{\ell_1 \times B}$. The database is read once from HBM and reused across all B clients, so the arithmetic intensity is

$$\mathrm{AI} \;=\; \frac{\ell_1 \ell_2 B}{\ell_1 \ell_2 + \ell_1 B + \ell_2 B} \;\approx\; B \qquad (\ell_1, \ell_2 \gg B). \tag{3.1}$$

(The database does not fit in any cache, so HBM traffic and total memory traffic coincide.) The *Roofline model* (Williams et al., 2009) bounds achievable throughput by $\min(\mathrm{BW} \cdot \mathrm{AI},\ \text{Peak compute})$; the *ridge point* $\mathrm{AI}^* = \text{Peak compute}/\mathrm{BW}$ is the small-

GPU	INT8 TC (TOPS)	BW (GiB/s)	AI*	$/hr	TOPS/$
V100	—	838	—	3.06	∼2.5
T4	130	320	378	0.53	247
L4	242	279	807	0.80	**302**
A100	624	1899	306	∼3.00	∼208

Table 3.1: GPU specifications from NVIDIA datasheets (NVIDIA Corporation). AI* is the ridge point for INT8 tensor-core operations; the V100 has no INT8 tensor cores and falls back to the SIMT CUDA-core path (∼7.8 TOPS INT32 peak: 5120 CUDA cores ×1.53 GHz, giving ∼2.5 TOPS/$). Prices are AWS on-demand rates.

est intensity at which peak compute is reachable. In practice, GEMM tile granularity caps realized intensity below B, but by $B = 64$–128, measured throughput saturates at its compute ceiling on both GPUs (Figure 3.1): the workload is compute-bound, and the relevant deployment metric is TOPS per dollar.

On INT8 TOPS/$, the inference-class L4 dominates: L4 delivers 302 TOPS/$, T4 247, A100 ∼208, V100 ∼2.5 (SIMT fallback). The prior GPU PIR's V100 choice (Lehmkuhl et al., 2025) was therefore suboptimal for a compute-bound workload—a detailed cost comparison appears in Chapter 5. Note that on the INT32 SIMT path DistPIR actually uses, the V100 and L4 have comparable throughput (∼7.8 vs. ∼7.5 INT32 TOPS).

Figure 3.1 realizes this picture for the database matmul on an A100 and an L4. Each panel plots the effective database throughput against batch size B; the diagonal is the HBM roof $\text{BW} \cdot B$, and the two horizontals are the realized INT32 SIMT and INT8 tensor-core ceilings. Both kernels saturate at their compute plateaus beyond the ridge point; despite byte decomposition multiplying the INT8 work by 4×, the tensor-core ceiling still sits roughly 4× above the INT32 SIMT plateau on both GPUs (Section 3.3 breaks this down). The small-batch regime is equally informative: at $B = 1$ the tensor-core kernel already reaches ∼75% of HBM bandwidth on both GPUs, while the SIMT kernel hits only ∼7–20%. The SIMT CUDA-core datapath cannot issue loads fast enough to saturate HBM at low arithmetic intensity; the

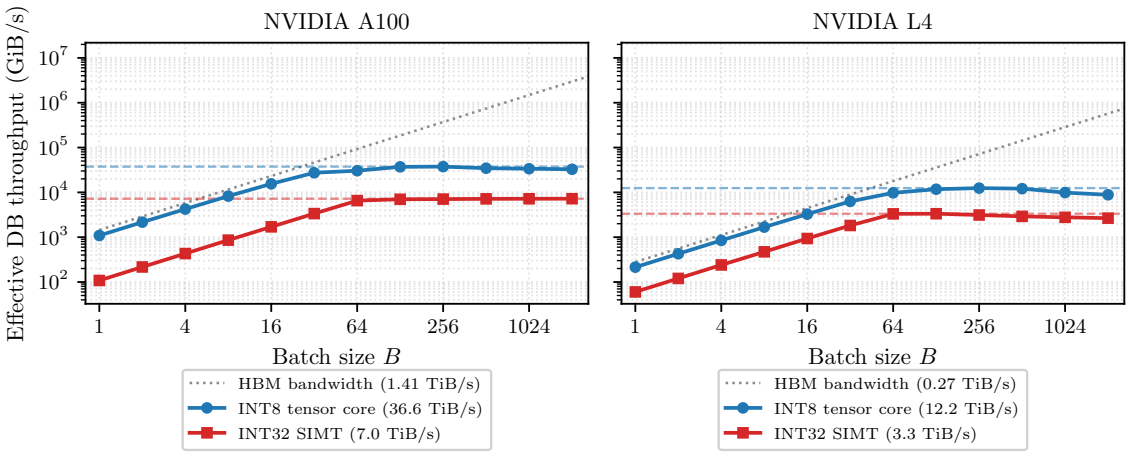

Figure 3.1: Roofline for the database matmul on NVIDIA A100 and NVIDIA L4, 65K × 65K uint8 database (4 GiB). Dashed horizontals mark the realized compute ceiling of each kernel (peak effective throughput across the batch sweep); legend numbers show the same values.

tensor-core datapath can, in the same way CPU-side SIMD kernels saturate DRAM where scalar loops do not. Tensor cores therefore dominate SIMT across the whole batch range, not just at the compute-bound plateau.

3.3 The Tensor-Core Constraint

Tensor cores are functionally equivalent to tiny *systolic arrays*—pipelined MAC grids that stream operands through dedicated hardware without the scalar fetch/issue/writeback overhead of SIMT CUDA cores—and deliver their TOPS rates only along a narrow datapath: INT8 multiplies with INT32 accumulators on small tiles, with no integer path at deployment-grade TOPS for wider operands. The parameter choice $p = 2^8$ (plaintext byte) and $W = 2^{32}$ (scan modulus) is deliberate: database entries fit natively in the INT8 operand and scan accumulators wrap for free on the INT32 accumulator at W, so byte decomposition of the wider scan operand maps cleanly onto the hardware path without requiring INT16 operands or INT64 accumulation. PIR operands are 32-bit (queries modswitched into $\mathbb{Z}_W$) while database

entries are 8-bit ($\mathbb{Z}_p$ with $p = 256$); the 32-bit side does not run natively on tensor cores. Running a PIR matmul on tensor cores therefore requires splitting each multi-byte operand into its bytes and recombining the partial products:

$$x \cdot y = \sum_{i=0}^{a-1} \sum_{j=0}^{b-1} (x_i \cdot y_j) \cdot 2^{8(i+j)}, \qquad x = \sum_i x_i \, 2^{8i}, \; y = \sum_j y_j \, 2^{8j}. \tag{3.2}$$

Each $x_i \cdot y_j$ is a native INT8 tensor-core product, so a wide-operand matmul becomes $a \cdot b$ cross-product GEMMs recombined by positional shifts.

Free modular reduction at $W = 2^{32}$. The INT32 accumulator wraps modulo 2^{32} on overflow. A matmul in $\mathbb{Z}_W$ with $W = 2^{32}$ thus incurs no Barrett reduction—the hardware performs the reduction for free. A matmul modulo an NTT-friendly prime Q cannot exploit this directly: the INT32 accumulator wraps mod 2^{32}, not mod Q, so any overflow during accumulation corrupts the result. When the inner dimension is small enough to keep the accumulated cross-products below 2^{31}, byte decomposition still permits a single end-reduction mod Q—the strategy we use for the packing GEMM (Section 3.4). The scan's inner dimension $\ell_1 \gtrsim 2^{16}$ rules this out, forcing the scan into $\mathbb{Z}_W$ with the free wrap. This asymmetry between word and prime moduli shapes the construction in Chapter 4: its scan runs modulo W to harvest the free wrap, then a $Q \to W$ input and $W \to Q$ output modulus switch flank the matmul so that packing stays in Q.

Figure 3.1 quantifies this gap: at the uint8 / $p = 256$ operating point used in this thesis, the byte-decomposed tensor-core path plateaus at 12.2 TiB/s on L4 and 36.6 TiB/s on A100—roughly 4× the corresponding INT32 SIMT ceilings. The realized speedup falls well short of the raw TOPS ratio (Table 3.1: ∼48× on L4, ∼31× on A100): byte decomposition multiplies the INT8 work by 4, and recombining the partial products adds an accumulator-scan pass.

3.4 InsPIRe Packing as a GEMM

We rewrite the online phase of InspiRING.Pack (Construction 2.2) as a dense tensor-core GEMM. The derivation is due to Mahdavi et al. (2025, Section 5); we recall it here and then fold the rotation-dependent permutations offline.

Data layout. Under composable preprocessing (Section 2.11), the online phase of InspiRING.Pack has three data stores:

- *Server offline* (from the CRS seed, per output $o \in [\rho]$), stored in NTT domain: $\mathbf{T}_o, \bar{\mathbf{T}}_o \in \mathbb{Z}_Q^{Rt \times d}$ (matrix form of the Collapse chain for the $R = d/2-1$ rotations under τ_5^i and the $\tau_{2d-1}\tau_5^i$ conjugates, with t gadget digits; rows indexed by (i,k)), $\hat{\mathbf{T}}_o \in \mathbb{Z}_Q^{t \times d}$ (matrix form of the final $\mathbf{K}_h$ step), and the post-collapse RLWE mask $\hat{a}_o \in \mathbb{Z}_Q^d$ (kept in coefficient form, applied after INTT in post-processing).
- *Client upload* (per client $c \in [B]$), in NTT domain: $y_c, z_c \in \mathbb{Z}_Q^{t \times d}$—the body rows of $\mathbf{K}_g$ and $\mathbf{K}_h$ (Construction 2.2); the a-halves of both are CRS-derived.
- *Online from the scan* (per output o): $\mathbf{b}_o \in \mathbb{Z}_Q^d$—the b-values of the chunk of d scalar LWE outputs being packed into output o.

The slot permutations $\sigma_i, \bar{\sigma}_i \colon [d] \to [d]$ encode the τ_5^i and $\tau_{2d-1}\tau_5^i$ reorderings in the NTT domain; they depend only on the ring's Galois structure, not on any client. Packing compresses the ℓ_2 scalar LWE ciphertexts into $\rho = \lceil \ell_2/d \rceil$ packed RLWE ciphertexts, shrinking the response by $\sim d/2$ (Section 2.12.1).

Per-slot packing equation. For each client c, output o, and NTT slot z, InspiRING's Collapse stage produces the packed body coefficient (in condensed NTT

domain)

$$\tilde{b}_{c,o}[z] \;=\; \underbrace{\sum_{i=0}^{R-1}\sum_{k=0}^{t-1} y_c[k,\sigma_i(z)]\cdot \mathbf{T}_o[it{+}k,z]}_{\text{Term 1: rotations}} \;+\; \underbrace{\sum_{i=0}^{R-1}\sum_{k=0}^{t-1} y_c[k,\bar{\sigma}_i(z)]\cdot \bar{\mathbf{T}}_o[it{+}k,z]}_{\text{Term 2: conjugate rotations}} \;+\; \underbrace{\sum_{k=0}^{t-1} z_c[k,z]\cdot \hat{\mathbf{T}}_o[k,z]}_{\text{Term 3: final}}, \tag{3.3}$$

where Term 1 absorbs the τ_5^i-rotation chain of Collapse, Term 2 the $\tau_{2d-1}\tau_5^i$-conjugate chain, and Term 3 the final $\mathbf{K}_h$ step. Once $\tilde{b}_{c,o}$ is built, post-processing adds the scan's $\mathbf{b}_o$, assembles the RLWE with $\hat{a}_o$, and modswitches for transmission (Section 2.12.1, Lemma 2.3).

The permuted-gather obstacle. Term 3 is a dense pointwise contraction, but Terms 1 and 2 gather y_c at positions $\sigma_i(z), \bar{\sigma}_i(z)$ that vary with the rotation index i, breaking the dense access pattern a GEMM requires. Reading y_c at the rotation-permuted positions on the fly—or expanding into all R rotated copies—falls off the dense tensor-core GEMM datapath: the rotation index couples to the output slot z, leaving d separate matmuls, a sparse GEMM, or a SIMT gather kernel running at the much lower SIMT compute ceiling.

Absorbing permutations into M. The permutations $\sigma_i, \bar{\sigma}_i$ depend only on the Galois structure of the ring—they are entirely client-independent. Apply the inverse permutation to the server's precomputed data offline and fold all R rotations into a single matrix $M \in \mathbb{Z}_Q^{td\times\rho d}$ with

$$M[kd+z',\; od+z] \;=\; \sum_{\{i\,:\,\sigma_i(z)=z'\}} \mathbf{T}_o[it{+}k,\; z] \;+\; \sum_{\{i\,:\,\bar{\sigma}_i(z)=z'\}} \bar{\mathbf{T}}_o[it{+}k,\; z]. \tag{3.4}$$

Stacking each client's $\mathbf{K}_g$ key-switching matrix body y_c as one td-long row of $Y \in \mathbb{Z}_Q^{B\times td}$, Terms 1 and 2 of Equation (3.3) collapse into a single dense GEMM:

$$\boxed{\underbrace{C}_{B\times\rho d} = \underbrace{Y}_{B\times td} \cdot \underbrace{M}_{td\times\rho d}} \tag{3.5}$$

Row $(kd + z')$ of M contributes exactly the Term-1 summand $y_c[k, \sigma_i(z)] \cdot \mathbf{T}_o[it+k, z]$ for every rotation i with $\sigma_i(z) = z'$, and the symmetric Term-2 summand. Term 3 of Equation (3.3)—the $\mathbf{K}_h$ key-switch using z_c and $\hat{\mathbf{T}}_o$—is added per output in post-processing (below), since folding it into M would require Y to carry z_c alongside y_c.

M is read once from HBM and reused across all B clients, so the arithmetic intensity is

$$\text{AI} \;=\; \frac{B \cdot td \cdot \rho d}{B \cdot td + td \cdot \rho d + B \cdot \rho d} \;\approx\; B \qquad (td \cdot \rho d \gg B \cdot td,\; B \cdot \rho d), \tag{3.6}$$

the same as the database scan (Equation 3.1): the packing matrix M plays the role of db, and packing reaches the same theoretical roofline ridge at the same batch sizes. In practice, GEMM tile granularity caps the realized intensity below B, but both GEMMs reach the tensor-core compute ceiling at deployment batch sizes (Figure 3.1).

Both Y and M have 32-bit entries, so the GEMM uses byte decomposition with $a = b = 4$, i.e. 16 INT8 cross-products (Equation 3.2), dispatched as a single batched CUTLASS call—once the permutations are folded into M, the online packing step is a stock INT8 GEMM and nothing more sophisticated is required from the packing kernel itself. Unlike the scan, this GEMM runs modulo the NTT-friendly prime Q, so the INT32 accumulator does not wrap for free. Each uint8 cross-product is at most $255^2 < 2^{16}$, and accumulating over the inner dimension $K = td = 4 \cdot 2048 = 8192$ gives at most $K \cdot 255^2 \approx 5.3 \times 10^8$, comfortably below the INT32 overflow threshold 2^{31}. The 16 byte slices recombine in UINT64, with Barrett reduction mod Q applied once at the end.

Post-processing and concurrency with the scan. The GEMM output $C \in \mathbb{Z}_Q^{B\times\rho d}$ carries the Term-1/Term-2 rotation contributions to each packed RLWE's body polynomial $\tilde{b}_{c,o}$ in condensed NTT domain. Per-output post-processing accumulates Term 3 via a per-slot contraction $\sum_k z_c[k,z]\cdot\hat{\mathbf{T}}_o[k,z]$ (the $\mathbf{K}_h$ key-switch step, $B\cdot\rho\cdot t\cdot d$ $\mathbb{Z}_Q$ multiply-adds total—a factor of $d \approx 2000$ fewer than the main $Y \cdot M$ GEMM's $B \cdot t \cdot \rho \cdot d^2$), runs INTT, adds the scan's b-values $\mathbf{b}_o$ to produce the final body, modswitches the body to $R_{q_{22}}$, and pairs it with the precomputed modswitched mask $a'_o \in R_{q_{21}}$ for transmission. Only the $\mathbf{b}_o$ addition couples scan and pack, so packing is scan-independent and can in principle run concurrently with the scan. In practice both kernels will saturate the GPU's tensor cores (Figure 3.1), so overlapping compute doesn't improve throughput.

3.5 Offline on the GPU

The offline phase of any SimplePIR-family scheme reads the database once and emits precomputed material; it is data-movement-heavy rather than tensor-core-bound. Both pieces of our offline phase—the hint and the InspiRING M matrix—map cleanly to GPU kernels.

NTT-based hint. The SimplePIR hint $\mathbf{H} = \mathbf{A}\cdot\mathrm{db}$ is a bank of independent inner products between CRS-derived a-polynomials and database column blocks. We compute each hint block as a pointwise NTT-domain multiply–accumulate:

$$\mathbf{H}_j \;=\; \mathrm{INTT}\Big(\sum_{b=0}^{n_b-1} \mathrm{NTT}(\mathrm{db}_j^{(b)}) \odot \mathrm{NTT}(\mathbf{a}_b)\Big) \pmod{Q},$$

where $\mathbf{a}_b$ expands from the CRS seed and $\mathrm{db}_j^{(b)}$ is the b-th row block of column j. The asymptotic advantage over a $\mathbb{Z}_Q$-matmul hint (the symmetric alternative of Section 4.3.1) is $d/\log d \approx 186\times$ at $d = 2048$: NTT convolution is $\Theta(\ell_1\ell_2 \log d)$ work per database byte against $\Theta(\ell_1\ell_2 d)$ for the matmul. Lehmkuhl et al. (2025) already measure $\sim$180$\times$ on CPU for exactly this swap (RLWE convolution at $\sim$0.25 GiB/s vs.

SimplePIR-CPU LWE matmul at ∼1.4 MiB/s on a 4 GiB database). Moving the NTT onto the GPU compounds this on two axes: HBM bandwidth is an order of magnitude above DRAM, and the NTT is massively data-parallel both within a single transform and across the many independent hint-column NTTs (Ozcan et al., 2025), so SIMT CUDA cores cover the arithmetic while HBM covers the streaming. Chapter 5 reports the measured throughput; the asymptotic $d/\log d$ gap narrows empirically because the symmetric alternative runs on INT8 tensor cores while the NTT does not, but the asymmetric hint still wins on both L4 and A100.

InspiRING precomputation (the M matrix). Building M (Equation 3.4) runs as an offline GPU pipeline built on the same NTT primitives as the hint kernel—many independent NTTs and pointwise operations that parallelize across outputs and monomials. Chapter 5 details the implementation.

3.6 Client RLWE

The server scan consumes ℓ_1 scalar LWE ciphertexts. Encrypting them one at a time costs $O(\ell_1 d)$ $\mathbb{Z}_q$ operations—at $\ell_1 = 2^{16}$ and $d = 2048$, seconds of single-core time per query. The client can instead encrypt $n_b = \ell_1/d$ RLWE ciphertexts—one per block of d coordinates, each via an $O(d \log d)$ NTT round trip—and extract the b-polynomial coefficients to ship as scalar LWEs (Section 2.7). Total cost drops to $O(\ell_1 \log d)$, a factor of $d/\log_2 d \approx 186$ at $d = 2^{11}$. Lehmkuhl et al. (2025) confirm the gain empirically: on a 4 GiB database their scalar-LWE client takes 657 ms per query while the RLWE hybrid takes 4.1 ms (161×), a drastic reduction in client latency. The server sees the same scalar-LWE query interface as SimplePIR.

3.7 Auditing Lattice-PIR Through the GPU Lens

Sections 3.1–3.6 identify five criteria for a low-latency, high-throughput, communication-efficient lattice-PIR scheme:

1. the offline hint $\mathbf{H} = \mathbf{A} \cdot \mathrm{DB}$ runs on the GPU via NTT convolution in R_Q (Section 3.5);

2. the client query is RLWE-encoded and coefficient-extracted at cost $O(\ell_1 \log d)$ rather than scalar LWE at $O(\ell_1 d)$ (Section 3.6);

3. the scan is a $\mathbb{Z}_W$ dense GEMM with $W = 2^{32}$ that saturates the GPU's INT8 tensor cores (free reduction on the INT32 accumulator; Section 3.3);

4. the packing is a dense GEMM that can run on the GPU's INT8 tensor cores;

5. total communication is minimal: base query/response plus $O(1)$ packing keys in R_q, with no $O(n\ell_2 \log q)$-bit SimplePIR-style hint download.

Table 3.2 evaluates prior schemes against this profile; every one fails at least one criterion.[1]

HintlessPIR. Keeps the SimplePIR scan and *eliminates* the SimplePIR offline hint download by homomorphically evaluating $\langle \mathbf{H}_j, \mathbf{s} \rangle$ in RLWE over the LWE ciphertext modulus $\mathbb{Z}_q$. Packing requires a sequential chain of n slot-algebra rotations and key-switchings against the encryption of $\mathbf{s}$, followed by a component-wise product against the hint, rather than a dense GEMM. Communication inflates $4\times$ over SimplePIR.

CDKS / YPIR. CDKS packing evaluates the field trace via a $\log_2 d$-deep tree of $(\mathsf{ct} + \tau(\mathsf{ct}))$ operations, each level depending on the previous. The workload is a chain of automorphism + gadget-multiply + add, not a GEMM; YPIR inherits the structure. CRS-model preprocessing removes redundant NTTs but cannot flatten the sequential tree into a matmul.

[1] We omit Tiptoe from the audit: its large client upload puts it outside the operating regime of interest here.

Prior GPU PIR (Lehmkuhl et al., 2025). Scans at $\sim$2 TiB/s on a V100 using the SIMT CUDA-core path; the V100 has no INT8 tensor cores, and the paper leaves them unused on newer GPUs. The authors conclude that "homomorphic operations on RingLWE-based ciphertexts are not efficient on GPUs", so packing is explicitly routed to a separate CPU cluster, yielding 99.3% of compute time and 95% of per-query cost on the CPU side—criterion 4 (GPU-side packing) fully conceded.

YPIR+InsPIRe. YPIR's RLWE scan with InspiRING packing in place of the CDKS tree is the structural exception in the audit: InspiRING's two-generator trace factorization and monomial-X^k aggregation admit a matrix-matrix form, which is the only prior packing that maps onto a dense GEMM. We treat InsPIRe throughout as shorthand for InspiRING's core packing algorithm—InspiRING.Pack on the ℓ_2 scalar LWE outputs of a scan—dropping the homomorphic polynomial-interpolation step the full InsPIRe paper (Mahdavi et al., 2025) adds for smaller-record retrieval (Section 2.14). The naive online computation reads the client key at rotation-dependent permuted indices (Equation 3.3)—the obstacle Section 3.4 removes by absorbing those permutations offline into M.

The last row of Table 3.2 is the target shape; Chapter 4 assembles SandwichPIR to meet it.

Table 3.2: Lattice-PIR schemes against the five criteria. $Q = (\ell_1 + \ell_2)\log q$ is the base query-plus-response traffic; HintlessPIR's outer-RLWE terms live in parameters $h \gg q$, $N \gg n$.

Scheme	Hint	Query	Scan	Packing	Comm.
SimplePIR	LWE matmul	scalar LWE	$\mathbb{Z}_q$ matvec	—	$Q + n\ell_2 \log q$ hint
HintlessPIR	LWE matmul	scalar LWE	$\mathbb{Z}_q$ matvec	KS + SIMD $\odot$	$Q + \mathsf{enc}_h(\mathbf{s}) + \mathrm{KSK}_h + \lceil \ell_2/d \rceil$ RLWEs in $R_{h,N}$
YPIR	RLWE conv	RLWE extract	R_Q matvec	CDKS tree	$Q + \log_2 d$ KSKs in $R_{q,n}$
Prior GPU (no pack)	$\mathbb{Z}_W$ GEMM	RLWE + MS	$\mathbb{Z}_W$ GEMM, SIMT	—	$Q + n\ell_2 \log q$ hint
Prior GPU (w/ pack)	$\mathbb{Z}_W$ GEMM	RLWE + MS	$\mathbb{Z}_W$ GEMM, SIMT	KS + SIMD $\odot$ (CPU)	$Q + \mathsf{enc}_h(\mathbf{s}) + \mathrm{KSK}_h + \lceil \ell_2/d \rceil$ RLWEs in $R_{h,N}$
YPIR + InsPIRe	RLWE conv	RLWE extract	R_Q matvec	perm. gather	$Q + 2$ KSKs in $R_{q,n}$
Ideal	**NTT conv, GPU**	**RLWE + MS**	$\mathbb{Z}_W$ **GEMM, TC**	**dense GEMM, TC**	$Q + 2$ **KSKs in** $R_{q,n}$

Chapter 4

SandwichPIR

4.1 Design Motivation

Chapter 3 identified five criteria for a GPU-native lattice-PIR scheme—offline hint, client query, scan, packing, communication—and observed that every prior scheme fails at least one (Table 3.2). SandwichPIR addresses all five:

- *Client query* (criterion 2): RLWE-encoded and coefficient-extracted, adopted directly from Section 3.6.
- *Communication* (criterion 5): query + response + $O(1)$ packing keys in R_Q, no hint download, via InspiRING packing and the two-target modulus switch (Lemma 2.3).
- *Hint, scan, packing* (criteria 1, 3, 4): addressed by three construction-level commitments developed below.

The central obstacle is a three-way modulus conflict. The plaintext modulus $p = 256$ matches the INT8 operand width, so each database entry is a native tensor-core byte. The scan modulus $W = 2^{32}$ matches the INT32 accumulator, which wraps mod W for free—no Barrett reduction (Section 3.3). The NTT modulus Q must satisfy $Q \equiv 1 \pmod{2d}$ for fast polynomial arithmetic and $Q < W$ so that each scalar body value survives the $Q \to W \to Q$ round trip exactly; we pick $Q = 4{,}294{,}955{,}009$, the largest NTT-friendly prime below W, giving the largest ciphertext modulus under both constraints. Fast client RLWE encryption, fast offline hint preprocessing, and ring packing all require this Q, so the query arrives in $\mathbb{Z}_Q$ and the hint $\mathbf{H} = \mathbf{A} \cdot \mathrm{db}$

lives in $\mathbb{Z}_Q$. The scan matmul, however, wants W: since query and packing both anchor at $\mathbb{Z}_Q$, the body must make a round trip $\mathbb{Z}_Q \to \mathbb{Z}_W \to \mathbb{Z}_Q$ to reach the scan's preferred modulus and return. This round trip extends DistPIR's (Lehmkuhl et al., 2025) pre-evaluation modulus switch from $\mathbb{Z}_Q$ to $\mathbb{Z}_W$ with a return switch back to $\mathbb{Z}_Q$ for ring packing. The only remaining design choice is whether the *mask* makes that round trip too. SandwichPIR reconciles all three with the design commitments below.

Enc: **asymmetric modulus switching.** The body of every scalar LWE ciphertext makes the $Q \to W \to Q$ round trip so the scan runs as a $\mathbb{Z}_W$ GEMM; the mask never enters $\mathbb{Z}_W$. Section 4.3 proves this asymmetry is necessary: the symmetric alternative (mask modswitched to $\mathbb{Z}_W$ alongside the body) introduces two secret-dependent noise terms whose variance grows linearly in ℓ_1, pushing $\log_2 \delta$ from -105 to -35 at Wikipedia scale and to -0.4 at 64 GiB. Asymmetry zeroes both terms, leaving the scheme correct at $\delta = 2^{-105}$ for our parameters (Section 4.5).

Preproc: **RLWE/NTT-convolution hint.** With the mask staying in $\mathbb{Z}_Q$, the offline hint $\mathbf{H} = \mathbf{A} \cdot \mathrm{db}$ is computed as an NTT convolution in R_Q—exact, no rounding, $\Theta(\ell_1 \ell_2 \log d)$ work instead of the $\Theta(\ell_1 \ell_2 d)$ of an LWE matmul. The kernel is HBM-streaming and SIMT-parallel (Section 3.5); moving it onto the GPU compounds the $\sim$180$\times$ CPU-side speedup of RLWE convolution over LWE matmul that Lehmkuhl et al. (2025) report.

Apply: **InspiRING packing as a dense GEMM.** Section 3.4 absorbs InspiRING's rotation-dependent permutations offline into a single packing matrix M, reducing online packing to the dense GEMM $Y \cdot M$ over $\mathbb{Z}_Q$. The packing GEMM runs on INT8 tensor cores with byte decomposition, so both heavy online phases—scan and packing—share the same compute unit and arithmetic intensity.

LHEwP instance. The three commitments above are refinements of the LHEwP template (Definition 2.2), chained via composable preprocessing (Section 2.11) as $\mathsf{Apply} = h \circ g \circ f$. Enc encrypts under $\mathcal{E}_{\mathrm{RLWE}}$ over Q, extracts scalar LWE ciphertexts, and modswitches bodies to $\mathbb{Z}_W$. $\mathsf{Preproc}$ computes $\mathbf{H} = \mathbf{A} \cdot \mathrm{db}$ exactly in $\mathbb{Z}_Q$ via NTT convolution, then walks $\mathbf{H}$ through $\mathsf{InspiRING.Setup}$ to absorb the public-randomness halves of every post-scan packing ciphertext into $\mathrm{hint}_s = (M, \hat{\mathbf{T}}_o, a'_o)$; $\mathbf{H}$ is consumed in this walk and discarded, $\mathrm{hint}_c = \perp$. Apply composes the scan f in $\mathbb{Z}_W$ (tensor-core GEMM), the ring-pack g in R_Q as $Y \cdot M$ (tensor-core GEMM), and the two-target modswitch h for transmission. Dec is the two-target decryption of Lemma 2.3.

The sandwich chain, asymmetric on the mask/body split:

$$\begin{aligned} \text{body:}\quad & \mathbb{Z}_Q \xrightarrow{\mathrm{MS}_{Q\to W}} \mathbb{Z}_W \xrightarrow{\mathrm{scan}} \mathbb{Z}_W \xrightarrow{\mathrm{MS}_{W\to Q}} \mathbb{Z}_Q, \\ \text{mask:}\quad & \mathbb{Z}_Q \longrightarrow \mathbb{Z}_Q, \\ \text{combine:}\quad & \xrightarrow{\mathsf{InsPIRe}} R_Q \xrightarrow{\mathrm{MS}_{Q\to(q_{21},q_{22})}} \text{response}. \end{aligned} \tag{4.1}$$

Since $Q < W$, each scalar body value survives the round trip exactly (Section 4.3); only sums of body roundings accumulate bounded sub-Gaussian noise.

4.2 Protocol

Construction 4.1 instantiates the four-algorithm PIR-with-preprocessing template of Definition 2.1 with the sandwich chain (Equation 4.1). Each record is a row of the database, so the plaintext domain of Definition 2.1 is $\mathcal{D} = \mathbb{Z}_p^{\ell_2}$; the database has $N = \ell_1$ records. Table 4.1 fixes the concrete parameters; Appendix A.5 evaluates the noise budget (Lemma 4.3) at these settings and across database sizes.

Construction 4.1 (SandwichPIR Protocol)**.** Let λ be a security parameter. The database is a matrix $\mathrm{DB} \in \mathbb{Z}_p^{\ell_1 \times \ell_2}$ with $\ell_1 = n_b d$ rows and $\ell_2 = \rho d$ columns; we index records by a row $i^* \in [\ell_1]$.

- Let $d = d(\lambda)$ be the ring dimension (a power of two), and $R_Q = \mathbb{Z}_Q[x]/(x^d + 1)$ the cyclotomic ring.

Table 4.1: Parameter settings for SandwichPIR.

Symbol	Description	Value
d	Ring dimension (polynomial degree)	2048
Q	NTT-friendly ciphertext modulus	$4{,}294{,}955{,}009 \approx 2^{32}$
W	Word modulus (tensor core accumulator)	2^{32}
p	Plaintext modulus (bits per DB entry)	256
X_s, X_e	Secret and error distributions	$D(0.5)$
z	Gadget decomposition base	256
t	Gadget decomposition depth	4
q_{21}	Final modswitch target (mask, amortized)	2^{18}
q_{22}	Final modswitch target (body, per query)	2^{10}

- Let $Q = Q(\lambda)$ be the NTT-friendly ciphertext modulus and $W = 2^{32}$ the word modulus; we require $Q < W$.
- Let $q_{21} = q_{21}(\lambda)$ and $q_{22} = q_{22}(\lambda)$ be the mask and body targets of the final two-target modulus switch.
- Let $X_s = X_e = \chi(\lambda)$ be a sub-Gaussian distribution over $\mathbb{Z}$; secret keys are sampled from χ^d.
- Let (z, t) be the gadget base and depth used throughout, with $z^t \geq Q$.
- Let $(\mathsf{InspiRING.Setup}, \mathsf{InspiRING.Pack})$ be the packing algorithms of Section 2.14 instantiated over (R_Q, z, t).

The scheme SandwichPIR $= (\mathsf{Setup}, \mathsf{Query}, \mathsf{Answer}, \mathsf{Recover})$ is defined as follows, with the CRS seed seed_0 accessible to both parties.

- $\mathsf{Setup}(\mathrm{db})$: On input the database db, expand seed_0 to obtain $\mathbf{a}_0, \ldots, \mathbf{a}_{n_b-1} \in R_Q$ via $\mathrm{PRG}(\mathsf{seed}_0, b)$, and set $\mathbf{A} = \mathsf{NCyclicMat}(\mathbf{a}_0, \ldots, \mathbf{a}_{n_b-1}) \in \mathbb{Z}_Q^{\ell_1 \times d}$. Compute the *asymmetric NTT hint* in $\mathbb{Z}_Q$ via pointwise polynomial convolution,

$$\mathbf{H}[j,k] \;=\; \sum_{i=1}^{\ell_1} \mathrm{db}[i,j] \cdot \mathbf{A}[i,k] \pmod{Q}, \qquad (j,k) \in [\ell_2] \times [d], \tag{4.2}$$

and run $\mathsf{precomp} \leftarrow \mathsf{InspiRING.Setup}(\mathbf{H}, \mathsf{seed}_0)$, which uses composable preprocessing (Section 2.11) to absorb the public-randomness halves of every post-scan packing ciphertext into the packing matrix $M \in \mathbb{Z}_Q^{td \times \rho d}$ (Section 3.4), the per-output $\mathbf{K}_h$ matrices $\hat{\mathbf{T}}_1, \ldots, \hat{\mathbf{T}}_\rho$ used in Term-3 post-processing, and the per-output modswitched masks $a'_1, \ldots, a'_\rho \in R_{q_{21}}$ (cached by the client under lazy hint delivery, Remark 4.1). $\mathbf{H}$ is consumed in this offline walk; only the preprocessed material persists. Output $(\mathrm{hint}_s, \mathrm{hint}_c) = (\mathsf{precomp}, \perp)$.

- $\mathsf{Query}(i^*)$: On input a target row $i^* \in [\ell_1]$, parse $i^* = b^* d + j^*$ with $b^* \in [n_b]$ and $j^* \in [d]$. Sample $s \leftarrow \chi^d$. For each block $b \in [n_b]$, sample $\mathbf{e}_b \leftarrow \chi^d$, expand $\mathbf{a}_b$ from seed_0, set $\boldsymbol{\mu}_b = x^{j^*}$ if $b = b^*$ and $\boldsymbol{\mu}_b = 0$ otherwise, and form the RLWE body

$$\mathbf{b}_b \;=\; \mathbf{a}_b \cdot s + \mathbf{e}_b + \lfloor Q/p \rfloor\, \boldsymbol{\mu}_b \;\in\; R_Q.$$

  Extract the ℓ_1-dimensional scalar-LWE query $\mathbf{q} \in \mathbb{Z}_Q^{\ell_1}$ by coefficient extraction: $q_{bd+j} = \mathbf{b}_b[j]$ for $b \in [n_b]$, $j \in [d]$. Modulus-switch the body to $\mathbb{Z}_W$:

$$q'_i \;=\; \lfloor (W/Q)\, q_i \rceil \;\in\; \mathbb{Z}_W, \qquad i \in [\ell_1]. \tag{4.3}$$

  Run $\mathsf{pk} \leftarrow \mathsf{InspiRING.Setup}(s, \mathsf{seed}_0)$ to derive the client-side packing keys (two key-switching matrices of t rows each under s). Output $\mathrm{st} = s$ and $\mathrm{qu} = (\mathsf{pk}, \mathbf{q}')$.

- $\mathsf{Answer}(\mathrm{db}, \mathrm{hint}_s, \mathrm{qu})$: parse $\mathrm{hint}_s = \mathsf{precomp}$ and $\mathrm{qu} = (\mathsf{pk}, \mathbf{q}')$.

  1. *Scan in $\mathbb{Z}_W$.* Compute the database matmul

$$r_j \;=\; \sum_{i=1}^{\ell_1} \mathrm{db}[i, j] \cdot q'_i \pmod W, \qquad j \in [\ell_2]. \tag{4.4}$$

     This is a dense INT8 $\times$ INT8 $\rightarrow$ INT32 tensor-core GEMM; the accumulator wraps modulo $W = 2^{32}$ for free.

  2. *Modswitch $W \rightarrow Q$.* Set $r'_j = \lfloor (Q/W)\, r_j \rceil \in \mathbb{Z}_Q$ for each $j \in [\ell_2]$. Semantically, r'_j is the body of a scalar-LWE ciphertext encrypting $\mathrm{db}[i^*, j]$ under s whose mask $\mathbf{H}[j, \cdot]$ is CRS-derived and absorbed into $\mathsf{precomp}$ (Section 4.3.2).

3. *InspiRING packing.* Group the ℓ_2 scalar-LWE outputs into ρ RLWE ciphertexts in R_Q^2:

$$\mathsf{ct}^{\mathrm{out}} \leftarrow \mathsf{InspiRING.Pack}(\mathsf{pk}, \mathsf{precomp}, \{r'_j\}_{j\in[\ell_2]}) \in R_Q^{2\times\rho}, \tag{4.5}$$

implemented as the dense GEMM $Y \cdot M$ of Section 3.4.

4. *Final two-target modswitch.* For each output $o \in [\rho]$, apply the two-target modulus switch of Lemma 2.3 to $\mathsf{ct}_o^{\mathrm{out}} = (a_o, b_o)$: rescale $a_o \in R_Q$ to $a'_o \in R_{q_{21}}$ and $b_o \in R_Q$ to $b'_o \in R_{q_{22}}$. Output $\mathrm{ans} = ((a'_1, b'_1), \ldots, (a'_\rho, b'_\rho))$.

- $\mathsf{Recover}(\mathrm{st}, \mathrm{hint}_c, \mathrm{ans})$: parse $\mathrm{st} = s$, $\mathrm{hint}_c = \perp$, and $\mathrm{ans} = ((a'_1, b'_1), \ldots, (a'_\rho, b'_\rho))$. Apply the two-target decryption of Lemma 2.3 per output $o \in [\rho]$:

$$\begin{aligned} v'_o &= \lfloor -s \cdot a'_o \rceil_{q_{21}, q_{22}} + b'_o \in R_{q_{22}}, \\ v_o &= \lfloor v'_o \rceil_{q_{22}, p} \in R_p. \end{aligned} \tag{4.6}$$

Output the concatenation $(v_1, \ldots, v_\rho) \in R_p^\rho$ as the recovered row $\mathrm{db}[i^*, \cdot] \in \mathbb{Z}_p^{\ell_2}$.

Remark 4.1 (Lazy hint delivery). HintlessPIR, YPIR, and InsPIRe all pitch their offline preprocessing as *silent*—no hint downloaded, every query returns a self-contained response—but the construction hides a reusable artifact: the *mask* portion of every response (the a-polynomial of the RLWE ciphertext) is a deterministic function of the CRS and the database, identical across all queries from all clients. None of the three papers exploits this. SandwichPIR promotes the observation to a first-class protocol feature. The final-modswitch mask polynomials $a'_1, \ldots, a'_\rho$ are CRS-deterministic; rather than ship them offline at Setup time or embed them in every response, the server transmits them in-band with the client's first response, and the client caches them as its hint_c (Definition 2.1) for the remainder of the session. Subsequent queries are flagged as *body-only*: the server returns only $(b'_1, \ldots, b'_\rho)$, and the client recovers each output by running Recover against the cached hint_c masks. We call this *lazy hint delivery*; it amortizes a single CRS-derived mask over a stateless HTTP API without a separate offline phase (Chapter 6).

4.3 Asymmetric vs Symmetric Modulus Switching

The sandwich routes values through $\mathbb{Z}_Q \to \mathbb{Z}_W \to \mathbb{Z}_Q$. A single body value survives this round trip exactly when $Q < W$: forward and reverse modswitches contribute errors of magnitude at most $1/2$ each, for a total $|v'' - v| \le Q/(2W) + 1/2 < 1$, forcing $v'' = v$ by integrality. For SandwichPIR's $Q = 4{,}294{,}955{,}009$ and $W = 2^{32}$, the gap is 12,287. Accumulated body values pick up only small sub-Gaussian noise: each forward rounding $\epsilon_{b,i}$ is mean-zero and bounded by $1/2$, therefore sub-Gaussian with width $\sqrt{2\pi}/2$ (Hoeffding's lemma), so $\sum_i \mathrm{DB}[i,j]\,\epsilon_{b,i}$ is sub-Gaussian with width $(p/2)(\sqrt{2\pi}/2)\sqrt{\ell_1}$. Wraparound in the $\mathbb{Z}_W$ matmul is invisible modulo Q: every W-wrap contributes kQ to the rescaled sum, which vanishes.

After the online matmul the server holds a body $r_j \in \mathbb{Z}_W$ and a hint vector $\mathbf{H}_j \in \mathbb{Z}_Q^d$ for each output column j; it modswitches r_j to $\mathbb{Z}_Q$ and the packing step homomorphically subtracts $\langle \mathbf{H}_j, s \rangle$ to recover the plaintext indicator. How the hint is computed determines how much noise the subtraction introduces.

Two strategies present themselves. The *symmetric* choice mirrors the body's journey on the mask: modswitch each row's mask $\mathbf{a}_i$ from $\mathbb{Z}_Q$ to $\mathbb{Z}_W$, compute $\mathbf{H} = \mathbf{A}' \cdot \mathrm{DB}$ as a second word-arithmetic GEMM in $\mathbb{Z}_W$, and modswitch the result back to $\mathbb{Z}_Q$. The *asymmetric* choice leaves the mask in $\mathbb{Z}_Q$ entirely and computes $\mathbf{H} = \mathbf{A} \cdot \mathrm{DB}$ as an NTT convolution in $\mathbb{Z}_Q$. Both strategies produce a hint in $\mathbb{Z}_Q^d$ against which $\langle \mathbf{H}_j, s \rangle$ can be subtracted. They differ in noise.

4.3.1 Symmetric Hint (GEMM)

With the mask pre-modswitched to $\mathbf{a}'_i = \lfloor (W/Q)\,\mathbf{a}_i \rceil = (W/Q)\,\mathbf{a}_i + \boldsymbol{\epsilon}_{a,i}$ ($\|\boldsymbol{\epsilon}_{a,i}\|_\infty \le 1/2$), the hint is the $\mathbb{Z}_W$ GEMM $H_{j,k} = \sum_i \mathrm{DB}[i,j]\,a'_{i,k} \pmod W$; the server then modswitches $H_{j,k}$ to $\mathbb{Z}_Q$, introducing a further rounding $\epsilon_{H,j,k}$. Substituting into $\lfloor (Q/W)\,r_j \rceil - \langle \mathbf{H}'_j, s \rangle$ and grouping terms yields five noise sources:

1. RLWE error e_i through the scan, width σ_e per row, summed over ℓ_1.

2. Body rounding $\epsilon_{b,i}$ from the $Q \to W$ modswitch through the scan, bounded width $\sqrt{2\pi}/2$ per row, summed over ℓ_1.

3. *Mask rounding $\boldsymbol{\epsilon}_{a,i}$ through the scan, inner-producted with the secret s.* Each row contributes a ring product of width $\sigma_s \cdot (1/2) \cdot \sqrt{d}$, and the rows sum—so the stochastic variance scales linearly in ℓ_1.

4. Body rounding ϵ_r from the $W \to Q$ reverse modswitch, one term, width $\sqrt{2\pi}/2$.

5. *Hint rounding $\boldsymbol{\epsilon}_H$ inner-producted with the secret s.* Width $\sigma_s \cdot (1/2) \cdot \sqrt{d}$, one copy (not summed over rows).

Terms 3 and 5 involve the secret s, whose coefficients are themselves random; they are the price of having modswitched the mask.

4.3.2 Asymmetric Hint (NTT)

The mask never enters $\mathbb{Z}_W$. The hint is computed directly in $\mathbb{Z}_Q$ by NTT convolution,

$$H^{\mathrm{NTT}}_{j,k} = \sum_i \mathrm{DB}[i,j] \cdot a_{i,k} \pmod{Q},$$

exactly, with no rounding. The decryption equation collapses to three noise sources:

$$\lfloor \tfrac{Q}{W} r_j \rceil - \langle \mathbf{H}^{\mathrm{NTT}}_j, s \rangle = \sum_i \mathrm{DB}[i,j] \left(b_i - \langle \mathbf{a}_i, s \rangle\right) + \tfrac{Q}{W} \sum_i \mathrm{DB}[i,j]\, \epsilon_{b,i} + \epsilon_r \pmod{Q}. \tag{4.7}$$

The first sum recovers $\sum_i \mathrm{DB}[i,j](\lfloor Q/p \rfloor \mu_i + e_i)$ exactly in $\mathbb{Z}_Q$ because $\mathbf{a}_i$ was never modswitched—this is where the symmetric hint's noise terms 3 and 5 originated. Only RLWE error (Term 1 from the symmetric list), body rounding through the scan (Term 2), and the single reverse-modswitch rounding (Term 4) survive. Terms 3 and 5 are identically zero. The remaining two terms in Lemma 4.3—the final-MS mask rounding $\times$ secret and the packing gadget—come from the post-scan InspiRING packing and two-target modswitch; they are independent of the hint strategy and appear in both cases.

4.3.3 Why Asymmetry Wins

Term 3 in the symmetric list scales as $\ell_1 \cdot \sigma_s^2 \cdot (p/2)^2 \cdot d/4$: linear in the database height. At our parameters ($\sigma_s = 0.5\sqrt{2\pi}$, $p = 256$, $d = 2048$) this alone already consumes $\log_2 \delta$ aggressively: at $\ell_1 = 2^{16}$ (Wikipedia, 8 GiB) it drops $\log_2 \delta$ from -105 to -35; at $\ell_1 = 2^{18}$ (64 GiB) to -0.4 (one in two queries fails); at $\ell_1 = 2^{20}$ (1 TiB) the symmetric hint no longer satisfies the sub-Gaussian bound at any target failure probability. Even at 4 GiB ($\ell_1 = 2^{16}$, $\rho = 32$) where both hints technically satisfy $\log_2 \delta < -30$, the symmetric hint leaves only $\log_2 \delta = -36$ of headroom—too narrow for a cryptographic margin—while the asymmetric hint delivers -106 (Appendix A.5, Table A.2). The asymmetric hint is therefore the only viable choice at deployment scale: the extra symmetric noise terms are unrecoverable by any parameter tweak short of reshaping the database.

4.4 Security

Security of SandwichPIR requires that RLWE encodings under a secret $s \in R_Q$ remain pseudorandom even given the InspiRING packing key pk, which consists of RLWE encryptions of scaled automorphisms $\tau_g(s) \cdot z^k$ under the same secret. Pseudorandomness thus relies on a *circular security* assumption—the standard convention for lattice-based (fully) homomorphic encryption and RLWE-based PIR (Gentry et al., 2013; Chen et al., 2021; Li et al., 2024; Mahdavi et al., 2025). Our security argument follows InsPIRe's query-privacy reduction (Mahdavi et al., 2025, Appendix 6), specialized by dropping the RGSW samples InsPIRe uses for its homomorphic polynomial-evaluation step (which SandwichPIR does not perform; Section 2.14). Formally, the reduction assumes the key-dependent RLWE assumption of Mahdavi et al. (2025, Definition 6.1) for the scaled-automorphism family $\mathcal{F}_{\mathsf{auto}} = \{k \cdot \tau_g : k \in \mathbb{Z}_Q, g \in (\mathbb{Z}/2d)^*\}$, at $m = \ell_1/d + 2t$ samples under a common secret s: the $n_b = \ell_1/d$ query-block RLWE ciphertexts and the $2t$ RLWE rows of the packing key.

Theorem 4.1 (Query privacy)**.** *Let $m = \ell_1/d + 2t$. If* $\mathrm{RLWE}_{d,m,Q,\chi}$ *(with circular security for the $2t$ packing-key samples) is (T, ϵ)-hard, then for any pair of target rows $i_0^*, i_1^* \in [\ell_1]$, any algorithm running in time $T - O(\ell_1)$ distinguishes the query distributions $\mathcal{Q}_{i_0^*}$ and $\mathcal{Q}_{i_1^*}$ induced by Construction 4.1 with advantage at most 2ϵ.*

Proof. Fix $i^* \in [\ell_1]$ and parse $i^* = b^* d + j^*$. Let

$$\mathcal{D}_1 = \{(\mathbf{A}, \mathbf{A}s + \mathbf{e} - \mathbf{F}(s)) : s \leftarrow \chi^d,\ \mathbf{e} \leftarrow \chi^m\}, \qquad \mathcal{D}_2 = \{(\mathbf{A}, \mathbf{u}) : \mathbf{u} \xleftarrow{R} R_Q^m\}$$

denote the m-sample key-dependent RLWE and uniform distributions over $R_Q^m \times R_Q^m$, where $\mathbf{F}(s) \in R_Q^m$ encodes the InspiRING packing-key automorphism shifts (Construction 2.1): zero in the n_b query-block slots, $\tau_5(s) \cdot z^k$ in the first t packing-key slots, and $\tau_{2d-1}(s) \cdot z^k$ in the last t. Under the key-dependent RLWE assumption of Mahdavi et al. (2025, Def. 6.1) at $\mathcal{F}_{\mathsf{auto}}$, $\mathcal{D}_1 \approx \mathcal{D}_2$. Consider the simulator $\mathcal{S}$ that, on input $(\mathbf{A}, \mathbf{v})$ and i^*, outputs $(\mathbf{A}, \mathbf{v} + \lfloor Q/p \rfloor \boldsymbol{\mu}(i^*))$, where $\boldsymbol{\mu}(i^*) \in R_Q^m$ is the i^*-dependent public message vector—x^{j^*} in the b^*-th query-block slot, zero in all other query-block slots, and zero in every packing-key slot. Reassembling the first n_b shifted ring elements via coefficient extraction into an ℓ_1-length vector, deterministic-modswitching each coefficient by $q_i' = \lfloor (W/Q)\, q_i \rceil$, and packaging the remaining $2t$ as the rows of pk recovers the exact output shape of $\mathsf{Query}(i^*)$. Then:

- When $(\mathbf{A}, \mathbf{v}) \leftarrow \mathcal{D}_1$, $\mathcal{S}$'s output is distributed identically to $\mathcal{Q}_{i^*}$.
- When $(\mathbf{A}, \mathbf{v}) \leftarrow \mathcal{D}_2$, $\mathcal{S}$'s output is distributed identically to $\mathcal{D}_2$: adding a public shift to a uniform vector preserves uniformity.

$\mathcal{S}$ runs in time $O(\ell_1)$, so any algorithm distinguishing $\mathcal{Q}_{i^*}$ from $\mathcal{D}_2$ in time $T - O(\ell_1)$ has advantage at most ϵ. By the triangle inequality, any algorithm distinguishing $\mathcal{Q}_{i_0^*}$ from $\mathcal{Q}_{i_1^*}$ in time $T - O(\ell_1)$ has advantage at most 2ϵ. □

A standard hybrid extends Theorem 4.1 to multi-query adversaries: let k be the number of queries the adversary observes. If $\mathrm{RLWE}_{d,m,Q,\chi}$ is (T, ϵ)-hard with

the same $m = \ell_1/d + 2t$ samples of Theorem 4.1, then any k-query distinguisher in time $T - O(k \cdot \ell_1 \log d)$ has advantage at most $k\epsilon$, following Henzinger et al. (2023b, Corollary C.3). The $\log d$ factor over SimplePIR's $O(kn\sqrt{N})$ time bound comes from the reduction running Query $O(k)$ times, with each SandwichPIR Query costing $O(\ell_1 \log d)$ (Section 3.6).

Proposition 4.2 (Concrete security)**.** *The RLWE instance with $d = 2048$, $Q = 4{,}294{,}955{,}009 \approx 2^{32}$, secret and error distributions $X_s = X_e = D(0.5)$, and $m = 2^{23}$ LWE-equivalent samples achieves at least 192 bits of security against known lattice attacks, as estimated by the lattice estimator (Albrecht et al., 2015).*

4.5 Correctness

Correctness reduces to a sub-Gaussian noise budget. Under the independence heuristic (Section 2.7), the per-coefficient decryption error decomposes into a deterministic margin τ and a stochastic sub-Gaussian contribution of width σ; a union bound over the $d\rho$ output coefficients then bounds the failure probability δ.

Lemma 4.3 (Correctness of SandwichPIR)**.** *Fix parameters $(d, Q, p, W, q_{21}, q_{22}, z, t)$ and database dimensions (ℓ_1, ρ). With the asymmetric NTT hint, the probability that* Recover *returns an incorrect record coefficient is at most*

$$\log_2 \delta \;\le\; \log_2(2d\rho) \;-\; \frac{\pi\,\tau^2}{\sigma^2_{\mathrm{NTT}}\,\ln 2}, \tag{4.8}$$

where the deterministic margin is

$$\tau \;=\; \tfrac{\lfloor q_{22}/p \rfloor}{2} \;-\; \tfrac{1}{2}\Big(2 + (q_{22} \bmod p) + \tfrac{q_{22}}{Q}(Q \bmod p)\Big), \tag{4.9}$$

and the stochastic variance aggregates five independent sub-Gaussian sources:

$$\begin{aligned}\sigma^2_{\mathrm{NTT}} \;=\; &\underbrace{\left(\tfrac{q_{22}}{q_{21}}\right)^2 \tfrac{d\,\sigma_s^2}{4}}_{\textit{final-MS mask}} + \underbrace{\left(\tfrac{q_{22}}{Q}\right)^2 \left(\tfrac{p}{2}\right)^2 \ell_1\,\sigma_e^2}_{\textit{RLWE error}} + \underbrace{\left(\tfrac{q_{22}}{Q}\right)^2 \tfrac{z^2\,\sigma_s^2\,t\,d\,(d-1)}{4}}_{\textit{packing gadget}} \\ &+ \underbrace{\left(\tfrac{q_{22}}{Q}\right)^2 \left(\tfrac{Q}{W}\right)^2 \left(\tfrac{p}{2}\right)^2 \ell_1 \tfrac{2\pi}{4}}_{\textit{body rounding through scan}} + \underbrace{\left(\tfrac{q_{22}}{Q}\right)^2 \tfrac{2\pi}{4}}_{\textit{reverse-MS body}}\;.\end{aligned} \tag{4.10}$$

The proof—per-term derivations of (4.9) and (4.10), together with the union-bound argument for (4.8)—is deferred to Appendix A. At our parameters (Table 4.1) the margin evaluates to $\tau = 1$ and the last two rounding terms contribute less than 0.3% of σ^2_{NTT}.

4.6 Communication

Each SandwichPIR query round trip consists of an upload (the coefficient-extracted scalar-LWE query plus two InspiRING key-switching matrices) and a download (the packed response, after the two-target modswitch to (q_{21}, q_{22})). Under the CRS model (Section 2.7), both parties share a seed from which the uniform a-halves of every RLWE pair are reconstructible, so only the key-dependent b-halves travel the wire. The response's a-half—the packed mask—depends on the CRS and database alone, not on any particular query; it is shipped once per session under lazy hint delivery and cached by the client, while the query-dependent body ships on every query.

- **Query upload** (every query): ℓ_1 scalar-LWE body coefficients in $\mathbb{Z}_W$ plus $2t$ ring elements in R_Q for the two key-switching matrices' b-halves:

$$\text{upload} \;=\; \frac{\ell_1 \log_2 W + 2\,t\,d\,\log_2 Q}{8} \text{ bytes.}$$

- **First-query download** (mask + body):

$$\text{download}_{\text{first}} \;=\; \frac{\rho\,d\,(\log_2 q_{21} + \log_2 q_{22})}{8} \text{ bytes.}$$

- **Subsequent-query download** (body only, with the first-query mask cached):

$$\text{download}_{\text{sub}} \;=\; \frac{\rho\,d\,\log_2 q_{22}}{8} \text{ bytes.}$$

The body-to-record ratio is fixed by the modswitch at $\log_2 q_{22}/\log_2 p$ bits per bit—a classical PIR rate of $\log_2 p/\log_2 q_{22}$, independent of database size. The interesting scaling is in the *total* wire cost relative to a non-private retrieval of the

Table 4.2: Per-query communication across database sizes at the parameters of Table 4.1. Square layouts use $\ell_1 = \ell_2 = 2^k$; Wikipedia uses $\ell_1 = 2^{16}$, $\ell_2 = 2^{17}$. Upload is the same on every query; mask is shipped in-band with the first query only and cached thereafter. Multipliers are (upload + download)/ℓ_2 relative to a non-private retrieval, for the first query (paying the mask) and subsequent queries (steady state).

DB	ℓ_1	ρ	Upload	Mask (first only)	Body (every)	1st-query	Subsequent
0.25 GiB (square)	2^{14}	8	128 KiB	36 KiB	20 KiB	11.5×	9.3×
1 GiB (square)	2^{15}	16	192 KiB	72 KiB	40 KiB	9.5×	7.3×
4 GiB (square)	2^{16}	32	320 KiB	144 KiB	80 KiB	8.5×	6.3×
8 GiB (Wikipedia)	2^{16}	64	320 KiB	288 KiB	160 KiB	6.0×	3.8×
1 TiB (square)	2^{20}	512	4.06 MiB	2.25 MiB	1.25 MiB	7.6×	5.3×

same record (baseline: ℓ_2 bytes; the $\log_2 \ell_1$-bit index upload is negligible at any deployment, $\log_2 \ell_1 \ll 8\ell_2$). Table 4.2 reports two such multipliers: first-query (upload + $\text{download}_{\text{first}}$)/$\ell_2$ and steady-state subsequent (upload + $\text{download}_{\text{sub}}$)/$\ell_2$; they differ by exactly the amortized mask.

Tall ℓ_2/ℓ_1 layouts amortize the upload better: upload scales with ℓ_1, the record with ℓ_2, so a fixed-ℓ_1 deployment enlarges ℓ_2 to stretch the upload across more payload. Wikipedia's 2:1 shape at $\ell_1 = 2^{16}$ pays 3.8× non-private in steady state and 6.0× on the first query—versus 6.3× steady state for the 4 GiB balanced square at the same ℓ_1.

Chapter 5

Evaluation

This chapter presents performance, packing tradeoffs, batching, and cost results for SandwichPIR.

5.1 Experimental Setup

Hardware. We benchmark on two NVIDIA GPUs: L4 (Ada Lovelace, 242 INT8 TOPS, 24 GiB) and A100-PCIe-40GB (Ampere, 624 INT8 TOPS). L4 runs execute on AWS EC2 `g6.xlarge` instances; all A100 runs (single- and multi-GPU) execute on TACC Lonestar6 nodes. Specifications appear in Table 3.1. We exclude the V100 (no INT8 tensor cores) and the T4 (dominated by the L4 on TOPS/$).

Cost attribution. All dollar costs in this thesis use AWS on-demand pricing (as of April 2026): L4 via `g6.xlarge` at $0.80/hr; V100 via `p3.2xlarge` at $3.06/hr (DistPIR's baseline platform); A100-40GB imputed from `p4d.24xlarge` at $21.958/hr ÷ 8 A100s ≈ $2.75/hr per A100 (conservatively rounded to $3/hr); 8×CPU from DistPIR's `c7i.2xlarge` at $0.29/hr per instance (Lehmkuhl et al., 2025); client CPU via `m6a.xlarge` (AMD EPYC 7R13) at $0.173/hr; outbound egress at $0.09/GB.

Software. Rust codebase with infrastructure adapted from YPIR's implementation, CUDA kernels, CUTLASS (Thakkar et al., 2023) for INT8 GEMMs via FFI.

Databases. **Synthetic ($\leq$ 4 GiB):** square layout $\ell_1 = \ell_2 = 2^k$ ($\rho = 2^{k-\log_2 d}$), randomly initialized. **Wikipedia (8 GiB):** synthetic database with the shape of the

full English Wikipedia deployment of Chapter 6: $\ell_1 = 2^{16}$ rows $\times \ell_2 = 2^{17}$ bytes (128 KiB items, $\rho = 64$).

Methodology. All server timings are end-to-end GPU measurements averaged over 10 trials with an excluded warmup. DistPIR is the appropriate comparison baseline on three counts: it is the only prior GPU-accelerated lattice-PIR (Chapter 3, Section 3.7); it is the only prior work to measure *cross-client batching* at scale—the regime SandwichPIR targets—whereas SimplePIR, YPIR, HintlessPIR, and InsPIRe report single-query or low-batch numbers on commodity CPUs; and at ∼2.15 TiB/s on V100 (scan alone; Section 5.6) it reports the highest throughput of any lattice-PIR in the literature, an order of magnitude above the CPU-saturation ceiling of those schemes (∼10–20 GiB/s/core at memory bandwidth). DistPIR reports two deployment points: a single V100 and an 8-machine CPU cluster (8 × 8-core nodes, data-parallel SimplePIR scan).

5.2 Implementation

We release the full SandwichPIR implementation—server, client, CUDA kernels, WebAssembly build—under an open-source license;[1] every measurement in this chapter is reproducible from that repository. The implementation comprises ∼2,700 lines of handwritten CUDA kernels—NTT hint, InspiRING precomp, M-matrix scatter, byte decomposition, and panel accumulation—around batched CUTLASS INT8 u8×u8→u32 GEMMs for the scan and packing; and ∼7,500 lines of Rust for the protocol, tests, the native and WebAssembly client libraries, and the server. Chapter 3 established the two online GEMMs (scan in $\mathbb{Z}_W$, packing in $\mathbb{Z}_Q$) and the NTT-based hint; we describe the CUDA kernels that realize them, with per-stage measurements on L4 at the 4 GiB throughput-peak operating point in Tables 5.1 and 5.2.

[1] `https://github.com/sidsabh/sandwichpir`

Single-modulus arithmetic. Both SandwichPIR moduli $W = 2^{32}$ and $Q \approx 2^{32}$ admit a single-modulus path: the scan's INT32 tensor-core accumulator wraps at W for free, and the NTT runs in a single pass mod Q (lazy Barrett reduction details in the NTT hint kernel below). We therefore use no CRT decomposition. Prior RLWE-based schemes (Menon and Wu, 2024; Mahdavi et al., 2025) instead CRT-decompose a ∼56-bit ciphertext modulus into two ∼28-bit primes, running two NTT passes plus a CRT recombination per kernel.

NTT hint kernel. One CUDA block per hint column processes all $n_b = \ell_1/d$ row-blocks, streaming u8 database entries from global memory and fusing the forward NTT, pointwise Barrett multiply against precomputed $\mathrm{NTT}(\mathbf{a}_b)$, and multiply-accumulate into one kernel. The butterfly uses a *lazy* Barrett reduction: inputs are kept in $[0, 4Q)$ between rounds, avoiding a full reduction each step (Menon and Wu, 2022). Fusion keeps the NTT output, pointwise multiply, and per-b accumulation in shared memory throughout the thread block, avoiding global-memory round-trips between stages. On a standalone $d = 2048$ benchmark on L4, the butterfly runs at $0.35\,\mu$s/poly at batch 32,768—1.5× GPU-NTT (Ozcan et al., 2025)'s strict-Barrett butterfly ($0.54\,\mu$s/poly).

InspiRING precomp. Phase 1 dominates offline cost (Table 5.1) and parallelizes fully across (rotation, output) pairs: a $(d/2)\times\rho$ CUDA block grid, each block's threads cooperating on the inner loop $j \in [d]$ with a register-resident running product for the monomial table (no per-j global load). Phase 2 parallelizes only across the ρ outputs; inside each block the $R = d - 1$-step backward reduction is strictly sequential, but is dominated by Phase 1 at our deployment scales (Table 5.1).

M-matrix construction. Parallel across all three scatter axes: a $\lceil d/256 \rceil \times \rho \times t$ CUDA grid, one thread per (i, k, z) triple. Output-row collisions use `atomicAdd` on

a u64 staging buffer; a second fully-parallel kernel Barrett-reduces mod Q and byte-decomposes each entry into the four u8 slices CUTLASS expects. M is built once per database and reused across every query batch.

Online: byte-decomposed GEMMs. The scan $r = \mathrm{DB} \cdot q'$ in $\mathbb{Z}_W$ decomposes each 4-byte query operand into four 1-byte slices against the 1-byte database entries, stacked so that a single CUTLASS INT8 GEMM produces all four cross-products; a custom kernel recombines them with positional shifts into the u32 result, where the `int32` accumulator wraps modulo 2^{32} for free, matching W exactly. The packing GEMM $Y \cdot M$ in $\mathbb{Z}_Q$ requires exact accumulation: with $K_{\mathrm{gemm}} = td = 8192$ and byte-level operands ≤ 255, each cell accumulates at most $8192 \cdot 255^2 \approx 5.3 \times 10^8 < 2^{31}$, so no INT32 overflow. All 16 byte-slice products of the 4-byte $\times$ 4-byte decomposition are issued as a single CUTLASS call; a panel-accumulate kernel recombines the 4×4 output grid with 256^{i+j} shifts into the final u32 result mod Q.

PCIe overlap. Query upload and response download cross PCIe every request. All device-side buffers (scratch for the two GEMMs and intermediate tensors) and the pinned host-side query/key/response buffers are pre-allocated at server startup, so the online path issues no allocations. We pin the host-side buffers so the DMA engine copies directly to HBM without an intermediate driver-staging copy, and run two CUDA streams per GPU: a copy stream for H$\rightarrow$D and D$\rightarrow$H, and a compute stream for the two GEMMs and post-processing. The batch's upload on the copy stream is overlapped with the preceding batch's packing GEMM on the compute stream, hiding PCIe latency behind compute. For multi-GPU shards, the pinned buffer is marked portable so every GPU's DMA engine reads the same query from a single host allocation in parallel.

Measured breakdown. Tables 5.1 and 5.2 instrument the offline and online pipelines on L4 at the 4 GiB throughput-peak operating point ($B = 128$, the batch that max-

Table 5.1: Measured offline per-kernel times on L4, 4 GiB ($\ell_1 = \ell_2 = 2^{16}$).

Stage	Kernel	Time (ms)
Hint (390.7 ms)	NTT multiply–accumulate	373.2
	Barrett finalize + INTT	17.5
InspiRING precomp (2,000 ms)	Setup (monomials, rotated mask tables)	370
	Prep pack LWEs	16.4
	Running-product (Phase 1)	1,505.3
	Backward reduction (Phase 2)	107.2
M**-matrix** (441.3 ms)	Atomic scatter + Barrett + byte decomp	441.3
Total offline		**2,832**

imizes online throughput on L4 per Table 5.4). Offline is dominated by InspiRING precomp Phase 1 (1.5 s of the 2.83 s total). Online is split between the packing GEMM (25.5 ms) and the scan GEMM (29.5 ms); the remaining 4.4 ms covers H→D upload, post-processing, and D→H transfer. Amortized per-query compute cost is $59.4/128 = 0.46$ ms (clients still see the full 59.4 ms batch latency).

5.3 Performance Results

On the 4 GiB reference benchmark with batched clients, SandwichPIR reaches **8.35 TiB/s** on L4 and **20.28 TiB/s** on A100—3.9× and 9.4× the throughput of DistPIR (Lehmkuhl et al., 2025) on V100, respectively, and ∼10× the throughput of DistPIR's 8-machine CPU-cluster scan baseline. At \$0.80/hr vs. \$3.06/hr, the L4 delivers 15× better performance per dollar than the V100; against the 8×CPU cluster, 30× better. Multi-GPU sharding extends the per-GPU numbers nearly linearly (Section 5.5); single-query, 8 GiB, and multi-GPU results are consolidated in Table 5.4.

Online throughput (Figure 5.1). Both SandwichPIR curves beat the DistPIR GPU baseline across the entire batch range: A100 plateaus at ∼20 TiB/s by

Table 5.2: Measured online per-kernel times on L4, 4 GiB, $B = 128$.

Stage	Kernel	Time (ms)
Upload (0.9 ms)	H→D packing keys	0.9
Packing GEMM (25.5 ms)	Y byte decomposition	0.1
	Byte-decomposed CUTLASS u8×u8→i32	22.1
	Accumulator scan + Barrett mod Q	2.4
	$\hat{\mathbf{T}} \cdot Z$ key-switch term + finalize	0.9
Scan GEMM (29.5 ms)	Byte-decomposed CUTLASS u8×u8→i32	28.8
	Accumulator scan + $W \to Q$ modswitch	0.7
Post (3.5 ms)	INTT + add body + two-target modswitch + bitpack	2.7
	D→H download	0.8
Total per batch ($B = 128$)		**59.4**

$B \approx 128$, L4 at ∼8 TiB/s by $B \approx 64$, while DistPIR's GPU saturates below ∼2 TiB/s and its 8×CPU cluster at ∼0.8 TiB/s. The shape matches the roofline prediction of Section 3.2: INT8 tensor cores dominate the SIMT path (where DistPIR runs) across the whole batch range, not just at the plateau.

Offline throughput (Figure 5.2). The pure NTT hint—the analog of DistPIR's RLWE-hybrid offline—runs at ∼10.2 GiB/s on L4 and ∼18.5 GiB/s on A100 at 4 GiB. On L4, this is 20× the throughput of DistPIR's GPU SimplePIR variant (0.51 GiB/s) at 76× better performance per dollar and 40× the throughput of their 1-CPU RLWE hybrid (0.25 GiB/s) at 15× better performance per dollar. For comparison, we measured a symmetric TC-GEMM hint on the same L4: 4.55 GiB/s at 4 GiB (2.2× below our NTT hint), confirming that NTT's $d/\log d$ algorithmic advantage beats the tensor-core hardware advantage empirically—and the correctness collapse of Section 4.3.3 rules out the symmetric choice at deployment scale regardless. The full-pipeline curve (hint + InspiRING precomp + M-matrix build) sits nearly an order of magnitude lower than the pure hint because InspiRING precomp's Phase 1 running-product accumulation (dominant offline cost per Table 5.1) adds substantial

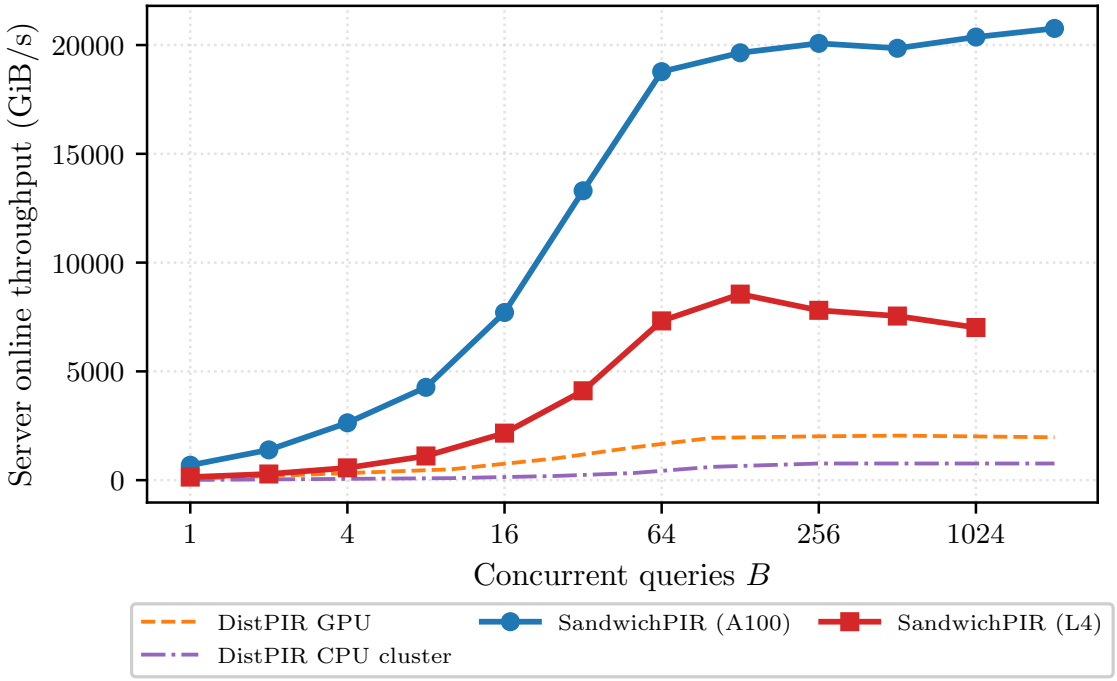

Figure 5.1: Server online throughput on a 4 GiB database vs. batch size, overlaid on DistPIR's Figure 5.

work beyond the hint kernel, but still beats both DistPIR variants: 1.42 GiB/s on L4 (2.8× GPU SimplePIR, 5.7× CPU hybrid) and 2.47 GiB/s on A100 (4.8× and 10×).

Client encryption and decryption (Figures 5.3 and 5.4). Per-query client encryption runs 30–100× faster than DistPIR's scalar-LWE baseline—near the $d/\log d \approx 186$ asymptotic from Section 3.6. A gap remains against DistPIR's own RLWE hybrid: SandwichPIR additionally generates two InspiRING packing keys per query, which the hybrid variant does not. Decryption—with the same $d/\log d$ asymptotic gain—computes the inner product of the RLWE secret with the mask, adds the body, and rounds to plaintext (one polynomial multiply per packed output), and scales linearly with ρ; encryption scales with ℓ_1 and plateaus once ℓ_1 is fixed. These measurements ran on the host CPU of our L4 node (`g6.xlarge`, AMD EPYC 7R13),

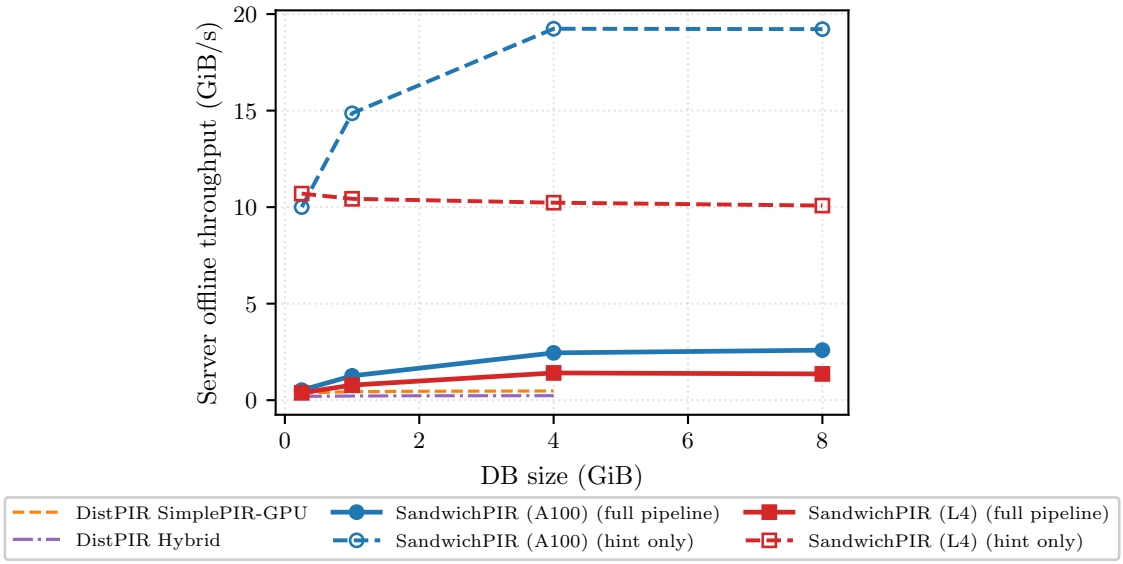

Figure 5.2: Offline preprocessing throughput vs. database size, overlaid on DistPIR's Figure 4. Per GPU config: full pipeline (hint + InspiRING precomp + M-matrix build; solid) and pure NTT hint (dashed, open marker).

matching the commodity `m6a.xlarge` client instance ($0.173/hr).

5.4 Packing Tradeoff

A SandwichPIR session exchanges three components per round trip: a query upload (ℓ_1 scalar-LWE body coefficients in $\mathbb{Z}_W$, packed as 4-byte words), a packing-key upload ($2t \cdot d \cdot 4$ bytes for the two InspiRING key-switching matrices), and a response download. Under lazy hint delivery (Section 4.2) the response decomposes into a one-time packed mask ($\rho \cdot d \cdot \log_2 q_{21}/8$ bytes, cached by the client after the first response) and a per-query body ($\rho \cdot d \cdot \log_2 q_{22}/8$ bytes). Subsequent queries do *not* re-transfer the mask. Per-client response egress is therefore 224 KiB at 4 GiB (mask + body) and 448 KiB at 8 GiB, versus >512 MiB for DistPIR's SimplePIR hint alone (Table 5.4).

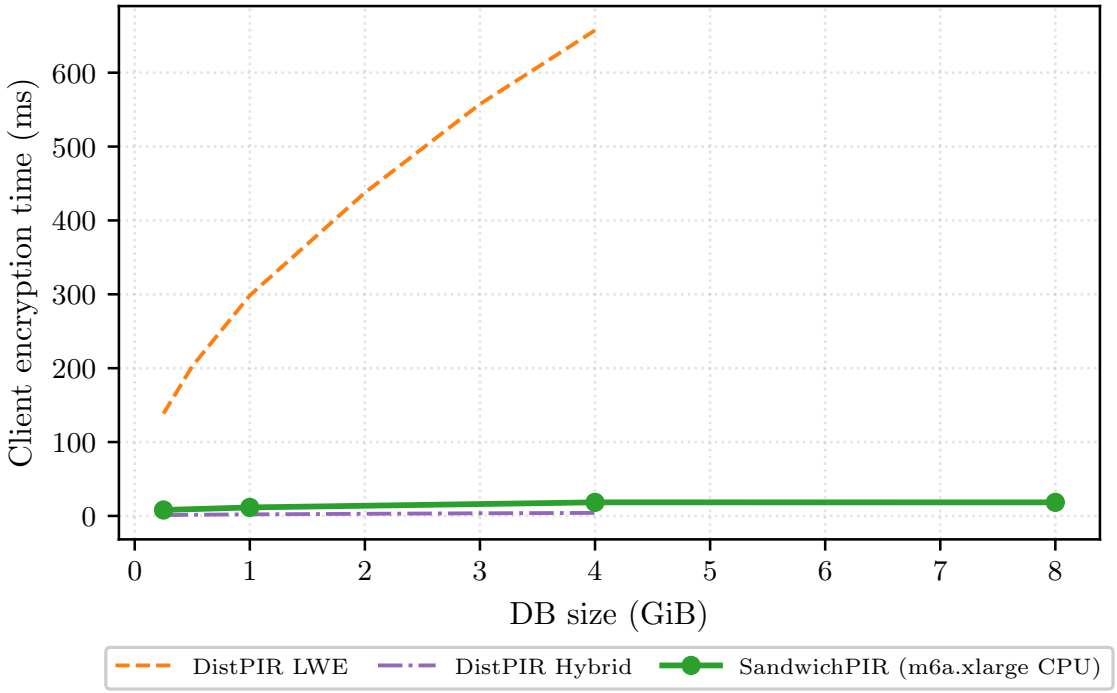

Figure 5.3: Per-query client encryption time vs. database size, overlaid on DistPIR's Figure 3.

Lazy hint delivery vs. the full SimplePIR hint. The alternative, shipping the full uncompressed SimplePIR hint $\mathbf{H} = \mathbf{A} \cdot \mathrm{DB}$ once at setup, saves the per-query packing-key upload but pays one large one-time download. The crossover in total bytes over K queries from a single client is

$$\underbrace{(\text{packed mask})}_{\rho\, d\, \log_2 q_{21}/8} + K \cdot \underbrace{(\text{packing keys})}_{2t\, d \cdot 4} = \underbrace{(\text{full SimplePIR hint})}_{n\, \ell_2\, \log_2 Q/8}. \tag{5.1}$$

On the 4 GiB synthetic benchmark ($\ell_2 = 65{,}536$, $n = d = 2{,}048$, $\rho = 32$, $t = 4$, $\log_2 Q = 32$, $q_{21} = 2^{18}$, $q_{22} = 2^{10}$) Equation (5.1) evaluates to $144\,\mathrm{KiB} + 64\,\mathrm{KiB} \cdot K = 512\,\mathrm{MiB}$, crossover at $K \approx 8{,}190$. On the 8 GiB Wikipedia configuration ($\ell_2 = 131{,}072$, $\rho = 64$) the full hint is 1 GiB, the packed mask 288 KiB, and the crossover sits at $K \approx 16{,}400$ (Chapter 6). Both sit beyond any realistic session and the full-hint alternative also pays a cold-start bandwidth spike (∼11 min on a 12 Mbps mobile link

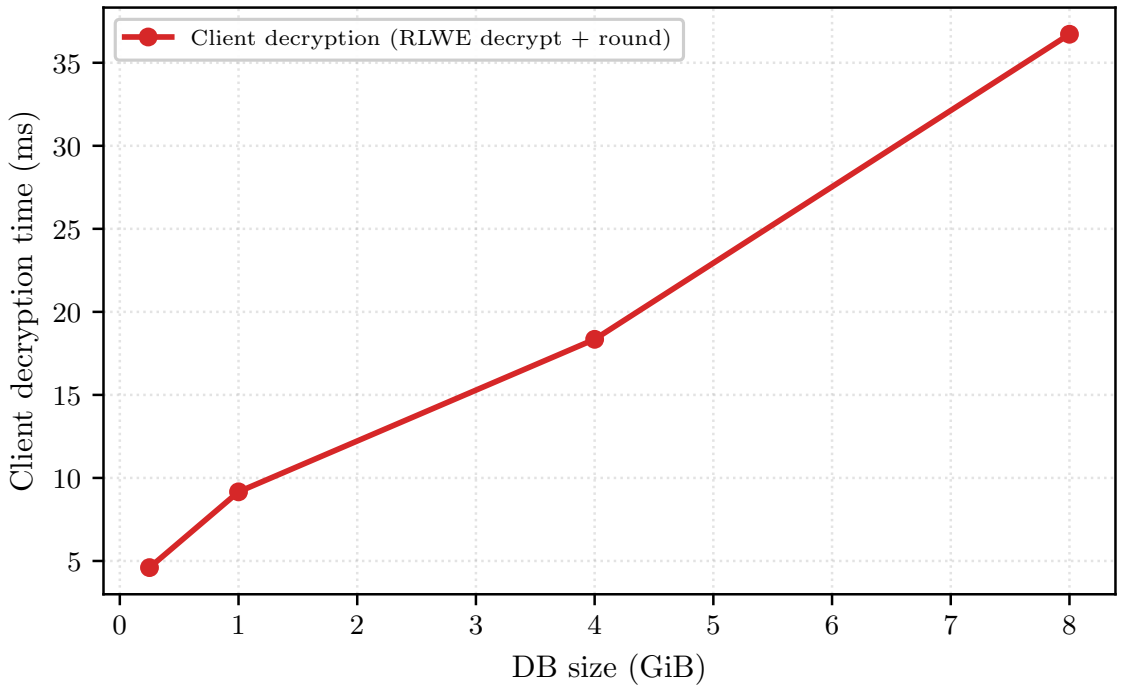

Figure 5.4: Per-query client decryption time vs. database size.

at the 8 GiB scale; Chapter 6) before the client can issue its first query. Lazy hint delivery therefore wins in both total bytes and time-to-first-response across every deployment scenario we consider. However, the full pipeline pays for packing in throughput: the scan GEMM already saturates the L4's tensor cores, and the packing GEMM runs sequentially, giving 8.35 TiB/s end-to-end vs. the scan-only 12.2 TiB/s plateau of Figure 3.1—a nonetheless worthwhile ∼32% tax that buys InspiRING's response compression.

5.5 Multi-GPU Scaling

Table 5.3 reports online and offline scaling on TACC Lonestar6 (A100-PCIe-40GB) for the 8 GiB Wikipedia database. Each GPU receives a column partition of ρ/N RLWE outputs and runs its own tensor-core kernel; no inter-GPU communication is required.

Table 5.3: Multi-GPU scaling on TACC Lonestar6 A100-PCIe-40GB, 8 GiB database, peak batch per N. Each GPU receives a column partition of ρ/N RLWE outputs.

N	B	Online (ms)	Online (TiB/s)	Online speedup	Offline speedup
1	1024	394	20.31	1.00×	1.00×
2	1024	218	36.68	1.81×	1.90×
3	2048	291	**55.01**	**2.71**×	**2.59**×

Peak online throughput scales from 20.31 TiB/s on a single A100 (VRAM-limited to $B = 1024$) to 55.01 TiB/s on 3×A100 at $B = 2048$, a 2.71× speedup. At matched batch ($B = 1024$), offline scaling slightly exceeds online (1.90× vs. 1.81× at $N = 2$) because precomp rebuild is embarrassingly parallel along output columns, whereas the online GEMM pays a per-shard fixed cost per query: each shard requires the full query and packing keys. At peak batch per N, the larger batch amortizes that fixed cost and online overtakes (2.71× vs. 2.59× at $N = 3$). PCIe contention on the Lonestar6 node explains the remaining gap from ideal; NVLink would reduce it.

5.6 Summary

Table 5.4 consolidates SandwichPIR's measurements across all GPU platforms, database sizes, and operating points (single-query, peak-throughput batch, and Wikipedia deployment), alongside the prior-work baselines.

Reading the table. Single-query rows ($B = 1$) sit at the memory-bandwidth floor of the scan (Section 3.2); peak-batch rows hit the compute-bound plateau where per-byte arithmetic intensity is sufficient (Section 5.3). The L4 4 GiB peak (8.35 TiB/s, $0.104/Mq compute) is our primary apples-to-apples comparison with prior work's V100 reference ($1.55/Mq compute): 3.9× throughput at 15× better performance per dollar *on compute alone*. Multi-GPU shards extend throughput linearly while keeping compute $/Mq within 2× of the single-GPU L4 floor.

Egress dominates compute at scale. At AWS's standard \$0.09/GB outbound rate and the 10^6-unique-clients model, SandwichPIR's 224 KiB first-query response costs \$20.64/Mq at 4 GiB (and \$41.29/Mq at 8 GiB)—already hundreds of times the L4 peak compute cost, and even at $B = 1$ (compute \$6.20/Mq) egress exceeds compute by 3×. Per-query total cost is therefore dominated by response-byte egress at every operating point, and the egress floor is fixed by the protocol's response shape (Lemma 2.3's two-target modswitch and InspiRING's ρ packed RLWEs).

Hint compression in DistPIR's cost column. DistPIR's reported throughputs on both V100 and the CPU cluster measure the SimplePIR-style scan alone; hint compression is not included in the throughput axis. DistPIR's own end-to-end measurements with HintlessPIR packing enabled show the CPU cluster accounting for 99.3% of compute time and 95% of per-query dollar cost (Lehmkuhl et al., 2025, Table 12)—packing, not scan, dominates their pipeline when compression is included. For the total-cost column we therefore charge each DistPIR client the full SimplePIR hint egress at AWS's \$0.09/GB rate—the only way to close the protocol end-to-end without hint compression—pushing their total cost to >\$48,000/Mq, over 2,000× our peak L4. This is a valid interpretation for DistPIR: enabling their HintlessPIR-based hint compression would serialize a CPU-side packing step behind the GPU scan and collapse the reported throughput to CPU rates, but they do not report throughput numbers in that configuration, so we model the scan-only throughput against a full-hint egress instead.

Table 5.4: Consolidated SandwichPIR throughput and cost. Single-GPU rows (L4, A100) are the comparison against DistPIR on V100 and an 8×CPU cluster; 3×A100 rows report column-wise multi-GPU sharding (Section 5.5). * peak-throughput batch; † Wikipedia deployment. $/Mq comp = per-query fleet compute; $/Mq total adds AWS egress ($0.09/GB) under 10^6 unique clients each issuing one query. SandwichPIR egress per client: 224/448 KiB at 4/8 GiB. DistPIR per-client egress is >512 MiB from the SimplePIR hint alone (query upload not modeled) (Lehmkuhl et al., 2025, Fig. 5, scan-only). [a]DistPIR's 8×CPU scan is paired with their 1-CPU RLWE-hybrid offline (Lehmkuhl et al., 2025, Fig. 4).

Platform	DB / B	Online (GiB/s)	Offline (GiB/s)	$/Mq comp.	$/Mq total
SandwichPIR					
L4	4 GiB / 1	143	1.42	6.20	26.84
L4	4 GiB / 128*	**8,551**	1.42	**0.104**	**20.74**
L4	8 GiB / 1	143	1.36	12.47	53.76
L4	8 GiB / 64†	7,320	1.36	0.243	41.53
A100	4 GiB / 1	687	2.47	4.86	25.50
A100	4 GiB / 2048*	**20,767**	2.47	**0.164**	**20.80**
A100	8 GiB / 1	728	2.58	9.14	50.43
A100	8 GiB / 1024*	20,797	2.58	0.321	41.61
3×A100	4 GiB / 1	1,720	5.65	5.75	26.39
3×A100	4 GiB / 1024*	**48,937**	5.65	**0.206**	**20.85**
3×A100	8 GiB / 1	1,894	6.69	10.50	51.79
3×A100	8 GiB / 2048*	**56,330**	6.69	0.355	41.65
DistPIR					
V100	4 GiB / 1	62.7	0.51	54.2	>48,372
V100	4 GiB / 512*	2,199	0.51	1.55	>48,320
8×CPU	4 GiB / 1	10.6	0.25[a]	243	>48,561
8×CPU	4 GiB / 256*	822	0.25[a]	3.14	>48,321

Chapter 6

Private Wikipedia

This chapter instantiates SandwichPIR as a private browser-search application on the full ∼6.4M-article English Wikipedia corpus. We motivate cross-client batching at $B = 64$ from a cost–latency tradeoff under Wikipedia's ∼4,500 page-views/sec load, describe the architecture (static-file server hosting the WebAssembly bundle and title index, stateless HTTPS API for GPU-backed PIR queries), and report end-to-end latency budgets on real mobile and fixed-broadband network profiles.

6.1 Motivation

English Wikipedia comprises approximately 6.4 million articles (November 2023 dump); after brotli compression (Alakuijala et al., 2018) and bin-packing (Section 6.3) the corpus fits in 8 GiB—small enough for GPU memory, large enough for a non-trivial PIR workload. The articles a person reads reveal sensitive information about interests, health, and politics. The underlying wiki sees ∼2 edits per second, but PIR requires a static database during online queries—updates must batch into periodic snapshots. Refreshing the 8 GiB snapshot every ∼6 s—the L4 offline compute time (Table 5.4)—bounds staleness to ≤ 12 edits out of 6.4M articles (imperceptible to readers) while leaving the online GEMMs to run at SandwichPIR's full throughput between refreshes.

Spiral (Menon and Wu, 2022) studied Wikipedia as a PIR target on a CPU deployment, reporting 4.3 s end-to-end query latency on an uncompressed Wikipedia variant (larger than our 8 GiB brotli-packed build). SandwichPIR revisits the same target with our SimplePIR-family GPU improvements—tensor-core scan and packing,

lazy hint delivery, and multi-GPU sharding—pushing throughput orders of magnitude higher (Chapter 5) and delivering end-to-end latency under half a second (Section 6.4).

Hardware. We deploy on NVIDIA L4 (Ada Lovelace, 242 INT8 TOPS, 24 GiB VRAM, AWS $0.80/hr). In the compute-bound regime (Section 3.2), L4 tops the TOPS-per-dollar ranking at 302 INT8-TC-TOPS/$ (Table 3.1), ahead of T4 (247), A100 (∼208), and V100 (∼2.5 on INT32 SIMT; V100 has no INT8 tensor cores). A100 and its multi-GPU extension would be latency-optimal but at a cost premium.

6.2 Batch Scaling and Queueing

Batching cross-client queries trades a small queue wait against a large reduction in server fleet size. English Wikipedia receives ∼4,500 page views/sec on average (Wikimedia, b), so $\lambda \approx 4.5$ queries/ms. We pick $B = 64$ for the deployment based on the throughput–queueing tradeoff below; the rest of Chapter 6 uses this batch.

Under uniform arrivals at rate λ, the expected queue wait for a query in a batch of B is $\mathbb{E}[\text{wait}] = (B-1)/(2\lambda)$. Table 6.1 sweeps B on a single L4 at 8 GiB and instantiates the deployment model, serving 10^6 unique clients (one query each). Per-query compute cost is a property of the per-GPU throughput ($B/T(B)$): scaling the fleet to meet λ multiplies the hourly bill and the queries/hour by the same factor s, so $/Mq is invariant in s and depends only on B. The fleet is replicated: each L4 holds the full 8 GiB database and its own hint_s, and a load balancer distributes incoming queries across the s GPUs; sharding the database across replicas would only help if throughput scaled perfectly linearly in s, which replication already achieves.

Why $B = 64$. Batching from $B = 1$ to $B = 64$ collapses the L4 fleet from 253 to 5 (51× smaller) and the per-query compute cost from $12.47 to $0.24 (52× cheaper) at ∼7 ms expected wait—negligible against the 218 ms upload floor on a 12 Mbps mobile

Table 6.1: Batch scaling on a single L4, 8 GiB Wikipedia. Fleet size $s = \lceil \lambda T(B)/B \rceil$ L4s at \$0.80/hr, sized to meet English Wikipedia's $\lambda = 4.5$ queries/ms (Wikimedia, b); $\mathbb{E}[\text{wait}] = (B-1)/(2\lambda)$ under uniform arrivals. Per-query compute cost is \$0.80/hr $\cdot\, T(B)/B$ (independent of s); the "\$/Mq compute" column scales this to 10^6 queries.

B	$T(B)$ (ms)	Tput (TiB/s)	$T(B)/B$ (ms/q)	$\mathbb{E}[\text{wait}]$ (ms)	L4s	\$/Mq compute
1	56.1	0.14	56.10	0	253	12.47
2	56.2	0.28	28.10	0.1	127	6.24
4	56.4	0.55	14.09	0.3	64	3.13
8	57.2	1.09	7.16	0.8	33	1.59
16	58.7	2.13	3.67	1.7	17	0.82
32	61.7	4.05	1.93	3.4	9	0.43
64†	**69.9**	**7.15**	**1.09**	**7.0**	**5**	**0.24**
128	135.3	7.39	1.06	14.1	5	0.24

† deployment batch chosen below.

link (Section 6.4). Compute per query plateaus at \$0.24/Mq by $B = 64$: this is the roofline ridge (Section 3.2) where per-byte arithmetic intensity saturates the tensor cores, so $B = 128$ and beyond cost the same while the wait doubles—$B = 64$ is the natural deployment knee.

Egress dominates compute. At $B = 64$, the \$0.24/Mq compute cost is $\sim$170$\times$ smaller than the \$41.29/Mq AWS egress incurred by the \$0.09/GB outbound rate on the 448 KiB first-query response. Once compute is collapsed at the ridge, further cost savings require shrinking the response.

6.3 Architecture

Database preparation. Article plaintext comes from the HuggingFace release (November 2023 English dump (Wikimedia, a)) as pre-extracted title–text pairs. Each article is individually compressed with brotli-5 (a 4-byte u32 length prefix followed by the compressed payload) and bin-packed into fixed 128 KiB rows: every article in the November 2023 snapshot fits in a single 128 KiB row after compression, so shorter articles share a row but no article spans consecutive rows—required for privacy, since

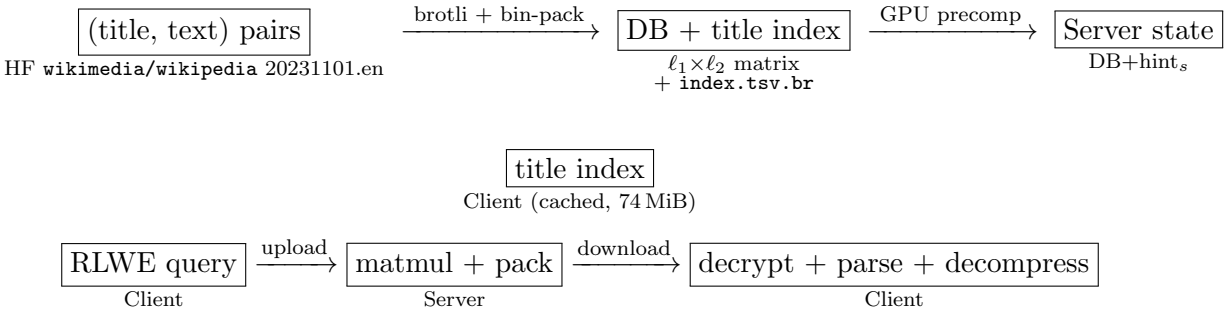

Figure 6.1: Data flow in Private Wikipedia. Top: offline database preparation from the HuggingFace Wikipedia dump. Bottom: online query processing; the title index is a one-time session-scoped download cached on the client, while only the RLWE query is uploaded per request.

multi-row articles would force the client to issue multiple queries and leak the article's length to the server. The result is $\ell_1 = 2^{16} = 65{,}536$ rows of $\ell_2 = 2^{17} = 131{,}072$ bytes, totaling 8 GiB. The database fits in the 24 GiB of an L4's VRAM alongside hint$_s$. Title $\to$ (row, offset) mappings are emitted as a TSV index, brotli-compressed to 74 MiB.

Server. The origin has two responsibilities. *Static files:* the WebAssembly client bundle and the 74 MiB title index are served as ordinary static assets over HTTPS—cacheable, deployable on any CDN, and identical across all clients. *PIR query API:* a stateless HTTPS endpoint receives the 320 KiB query, runs the scan and packing GEMMs on the GPU, and returns the 160 KiB query-dependent body (plus the 288 KiB CRS-derived mask in-band on the first query under lazy hint delivery; cached by the client thereafter). Offline, the server loads the database into GPU memory and runs the NTT-based hint $H = A \cdot \mathrm{DB}$ (Section 3.5) followed by the InsPIRe preprocessing walk (Section 3.4), retaining hint$_s = (M, \hat{\mathbf{T}}, a')$; H is consumed in the walk. The precomputation is rerun on each snapshot refresh (Section 6.1).

Client. The client is compiled to WebAssembly and runs in the browser. At session start, it downloads the 74 MiB title index (`index.tsv.br`) and uses it to resolve the user-typed title to a (row, offset) pair. It then generates an RLWE encryption of an indicator vector for the target row, modulus-switches from $\mathbb{Z}_Q$ to $\mathbb{Z}_{2^{32}}$, and produces two InsPIRe key-switch matrices (64 KiB), uploading 320 KiB in total. On the first query, the client also downloads the 288 KiB CRS-derived mask and caches it as hint_c; subsequent queries are flagged body-only, and the server returns just the 160 KiB query-dependent body. On receiving the response, the client recovers the RLWE ciphertext, decrypts to the target row's plaintext bytes, parses the 4-byte u32 length prefix at the stored offset, and brotli-decompresses the payload into the rendered article. End-to-end client latency is measured in Section 6.4. Appendix B presents screenshots of the implemented client interface in action (index load, autocomplete, articles rendered).

6.4 Results

SandwichPIR's Wikipedia deployment runs as a browser-native application: a stateless GPU server behind HTTPS, a WebAssembly client requiring no native install, and a 74 MiB brotli-compressed title index resolved client-side. Five L4s at \$0.80/hr saturate Wikipedia's $\lambda \approx 4.5$ req/ms load (Table 6.1, $B = 64$), and each client transfers 480 KiB to retrieve an article from the 8 GiB database—a communication ratio of $\sim 6 \times 10^{-5}$.

End-to-end latency. Figure 6.2 decomposes per-query latency on an L4 server (8 GiB database, $B = 64$) across four scenarios: first-query and subsequent-query exchanges under mobile and fixed-broadband network profiles, using the Ookla Speedtest Global Index median U.S. speeds for February 2026 (Ookla, 2026).

Upload bandwidth sets the floor on mobile: the 320 KiB query (body plus packing keys) takes 218 ms on a 12 Mbps mobile link and 47 ms on 55.83 Mbps fixed

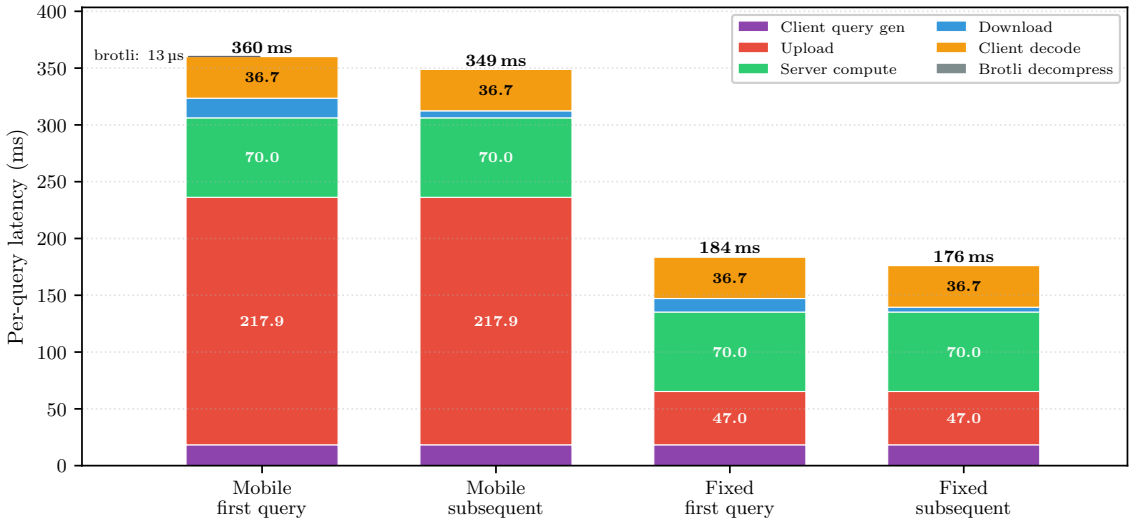

Figure 6.2: End-to-end per-query latency for Private Wikipedia. Server is an L4 at $B = 64$ on the 8 GiB database; client is `m6a.xlarge` (AMD EPYC 7R13). Network uses Ookla median U.S. speeds for February 2026 (Ookla, 2026): mobile 211.84 down / 12.03 up Mbps; fixed broadband 308.11 down / 55.83 up Mbps. Bandwidth-only; client-to-region RTT excluded.

broadband. On fixed broadband, server compute and client-side RLWE decryption (36.7 ms) are the leading components; brotli decompression is below 0.1 ms. End-to-end totals: subsequent queries complete in 349 ms on mobile and 176 ms on fixed broadband; first-query totals are 360 ms and 184 ms (the additional 11/8 ms is the mask download, respectively). The $B = 64$ server batch contributes 70 ms in every scenario.

What the latency model excludes. Figure 6.2 and the cost model of Section 6.2 isolate SandwichPIR's per-query construction-level impact. Application-space overheads—orchestration and frontend machinery common to any title-search deployment—are not included, and the 74 MiB title index (a one-time session-scoped static download) is not counted in per-query egress. The network model is bandwidth-only; round-trip latency between the client and the AWS region hosting the PIR server (typically ∼20–50 ms) is deployment-dependent and excluded.

Chapter 7

Conclusion

7.1 Summary

Prior lattice PIR forced a choice: large hints for maximum throughput, or costly homomorphic compression for small communication—and the only prior GPU-accelerated scheme (Lehmkuhl et al., 2025) left packing to a separate CPU cluster, so no single-GPU pipeline achieved both high throughput and small communication. This thesis presented SandwichPIR, a single-server PIR construction that closes this gap. The server performs a word-arithmetic database scan in $\mathbb{Z}_{2^{32}}$ sandwiched between modulus switches to an NTT-friendly prime $Q \approx 2^{32}$, under which both the client's RLWE ciphertext and InspiRING ring packing are defined. The two heavy server computations—scan and packing—thereby reduce to dense INT8 tensor-core GEMMs on a single GPU; cross-client batching drives them into the compute-bound regime (Section 3.2), so peak integer tensor-core TOPS per dollar is the determining cost metric.

On a 4 GiB database, SandwichPIR sustains 8.35 TiB/s online throughput on an NVIDIA L4 and 20.28 TiB/s on a single A100. Against the fastest prior lattice-PIR baseline (Lehmkuhl et al., 2025) on a V100 at each scheme's peak batch, this is 3.9× the throughput at 15× better performance per dollar on L4 and 9.4× the throughput on a single A100; against its 8-instance CPU-cluster baseline, L4 is ∼10× faster at 30× better performance per dollar.

The GPU-accelerated offline phase—an NTT-based hint kernel at ∼10 GiB/s on L4 and ∼19 GiB/s on A100 (20–76× prior hint implementations) plus InspiRING packing-matrix assembly—completes the 8 GiB database in ∼5.9 s on an L4 and ∼3.1 s

on an A100. Column-wise multi-GPU sharding extends to 47.79 TiB/s on a 3×A100 shard at 4 GiB (rising to 55.01 TiB/s on the 8 GiB Wikipedia database), a 2.71× online and 2.59× offline speedup over single-A100 with no inter-GPU communication.

We demonstrated the system end-to-end on the full English Wikipedia (6.4 million articles bin-packed into 2^{16} rows of 128 KiB) behind a WebAssembly browser client with lazy hint delivery: subsequent queries complete in 349 ms on mobile networks and 176 ms on fixed broadband, of which the $B = 64$ server batch contributes 70 ms.

7.2 Limitations and Future Work

Dynamic databases. InspiRING's online-to-offline trade (folding every rotation and gadget key-switch into a CRS-derived matrix M built from the database) makes SandwichPIR ideal for static or snapshot workloads like Wikipedia. Incremental updates exist for localized changes—a *Modify* of k bytes in a single row admits an $O(kd)$ hint update and rebuilds only $\lceil k/d \rceil$ of the ρ packing outputs—but a *Replace* of a full row invalidates all ρ outputs and costs nearly a full offline rebuild; *Insert* and *Delete* are computationally equivalent to Replace but additionally change ℓ_1 and so require PIR parameter re-selection. Every mutation therefore reruns a proportional fraction of the offline precomp, so under high update rates "offline" is no longer a one-time cost; a natural follow-on is a packing strategy that keeps more work online, trading peak throughput for faster incremental updates.

Multi-node scaling and large databases. We study SandwichPIR only in a single-node setting, where GPU memory bounds capacity: an L4 (24 GiB VRAM) fits roughly 16–20 GiB of database after the packing matrix and working buffers, an A100-40GB fits ∼32 GiB, and an A100-80GB ∼64 GiB. Our multi-GPU experiments use up to 3×A100 on a single TACC Lonestar6 node connected via PCIe; the same column-wise sharding extends to a distributed setting by broadcasting the query/keys

and gathering the per-shard responses across a network fabric. A 1 TiB deployment would require ∼44 L4s or ∼14 A100-80GB GPUs—more than any commodity single node—but memory is only a capacity bound, orthogonal to the compute-bound cost argument that drives TOPS/$. The near-linear throughput scaling of Section 5.5 should carry through to hundreds-of-gigabyte databases across nodes, with the coordination layer—query broadcast and response gather over the network fabric—as the remaining engineering.

Upload cost. The 320 KiB per-query upload dominates mobile end-to-end latency (218 ms of the 349 ms subsequent-query total; Chapter 6), an inherent cost of the SimplePIR family which requires one encrypted scalar per database row. Table 4.2 traces this out: the total wire cost is a 3.8×–11.5× multiplier over a non-private retrieval depending on database shape, and the upload dominates for every entry. Reducing it would require moving to a different PIR family—InsPIRe is state-of-the-art for the SimplePIR communication pattern—so porting the tensor-core pipeline to a more compact PIR construction is an open direction.

Small-record acceleration. SandwichPIR targets large records (Wikipedia articles at 128 KiB), where every coefficient of the RLWE response carries useful payload. Small-record workloads—a single $\mathbb{Z}_p$ element per query, as in private SCT auditing (Henzinger et al., 2023b)—admit different tradeoffs: DoublePIR- and InsPIRe-family schemes (Mahdavi et al., 2025) shrink the *response* to polylog(N) scalars (at the cost of a *larger* upload from the second-round query). Extending the tensor-core pipeline to the small-record regime is a concrete next step, broadening SandwichPIR's applicability from large-record workloads like Wikipedia to SCT-style small-payload deployments.

Client-side index. Our browser deployment ships a 74 MiB brotli-compressed title index once per session so the client can resolve a title string to a database row without

revealing which title (Chapter 6). At the U.S. median mobile downlink (211 Mbps) this is a 2.9 s one-time cost—$\sim$8$\times$ a subsequent-query latency, and the dominant component of first-session setup. One potential avenue is to replace the explicit index with a private search based on neural embeddings, as in Tiptoe (Henzinger et al., 2023a).

Private web search. Tiptoe (Henzinger et al., 2023a) composes PIR with neural retrieval to build private search: the server holds per-document embeddings, the client's query is an embedding, and the PIR backend retrieves the nearest-neighbor documents. Replacing Tiptoe's CPU PIR backend with SandwichPIR would require reworking the byte-decomposition strategy: Tiptoe's embedding arithmetic operates on a larger plaintext modulus than SandwichPIR's $p = 256$, which forces a ciphertext modulus above 2^{32} and breaks the INT8 tensor-core byte-decomposition used for the scan.

Hardware trends and specialized accelerators. The roofline analysis of Chapter 3 shows that SandwichPIR's throughput will scale directly with peak INT8 tensor-core TOPS: the workload is compute-bound at practical batch sizes, and INT8 is already the narrowest operand type that fits our modulus. NVIDIA tensor cores are themselves systolic arrays wrapped in a general-purpose GPU; dedicated systolic-array chips such as Google TPUs strip away the SIMT, graphics, and memory-subsystem overhead a GPU carries, and typically offer higher integer throughput per dollar for pure-GEMM workloads. SandwichPIR's GEMM-centric structure makes it a natural target; porting the two GEMMs could further improve performance per dollar, though the engineering question is whether the byte-decomposition, panel-accumulate, modulus switches, and INTT glue can be scheduled efficiently alongside the systolic matmul.

Additional Acknowledgments

We thank Alexandra Henzinger and Samir Menon for helpful discussions on this work, and Ryan Lehmkuhl for sharing DistPIR's baseline measurements used in Chapter 5. This research was supported in part by the Texas Advanced Computing Center (TACC) at The University of Texas at Austin, which provided computational resources (`http://www.tacc.utexas.edu`).

Appendix A

Correctness of SandwichPIR

We prove Lemma 4.3 for the asymmetric NTT hint of Section 4.3.2 and extend the analysis to the symmetric GEMM alternative of Section 4.3.1. The GEMM derivation produces the failure probabilities in the GEMM column of Table A.2; both hint variants share Stages 2–4 of the proof and differ only in Stage 1.

Fix any database $\mathrm{db} \in \mathbb{Z}_p^{\ell_1 \times \ell_2}$ with $\ell_1 = n_b d$ and any target index $i^* = b^* d + j^* \in [\ell_1]$. Run Construction 4.1 to obtain $(\mathrm{hint}_s, \perp) \leftarrow \mathsf{Setup}(\mathrm{db})$, $(\mathrm{st} = s, \mathrm{qu} = (\mathsf{pk}, \mathbf{q}')) \leftarrow \mathsf{Query}(i^*)$, $\mathrm{ans} = ((a'_o, b'_o))_{o \in [\rho]} \leftarrow \mathsf{Answer}$, and $r \leftarrow \mathsf{Recover}$. Write $\tilde{\mathbf{s}} = \mathsf{Coeffs}(s) \in \mathbb{Z}_Q^d$, $\tilde{\mathbf{e}} \in \mathbb{Z}_Q^{\ell_1}$ for the concatenation of $\mathsf{Coeffs}(\mathbf{e}_0), \ldots, \mathsf{Coeffs}(\mathbf{e}_{n_b-1})$, and $\mathbf{A} = \mathsf{NCyclicMat}(\mathbf{a}_0, \ldots, \mathbf{a}_{n_b-1}) \in \mathbb{Z}_Q^{\ell_1 \times d}$. Throughout, invoke the independence heuristic (Section 2.7): rounding errors, RLWE errors, and gadget digits are treated as independent sub-Gaussians with the indicated parameters, and coordinate-wise bounds on mean-zero terms translate to sub-Gaussian widths via Hoeffding's lemma.

The proof decomposes into four stages: (1) the $\mathbb{Z}_W$ scan and $W \to Q$ reverse modswitch produce a scalar LWE ciphertext with noise parameter $\sigma_{\mathsf{scan},\bullet}$ ($\bullet \in \{\mathrm{NTT}, \mathrm{GEMM}\}$); (2) InspiRING packing produces an RLWE ciphertext in R_Q^2 with noise $\sigma_{\mathsf{pack},\bullet}$; (3) the two-target modswitch (Lemma 2.3) produces noise $\sigma_\bullet$; (4) the decoder plus a union bound over $d\rho$ output coefficients yield the failure probability.

A.1 Stage 1: Scan and $W \to Q$ reverse modswitch

Coefficient-extracted query. Applying identity (2.2) to each RLWE body and concatenating,

$$\mathbf{c}^\top \;=\; \big(\mathsf{Coeffs}(\mathbf{b}_0)^\top, \ldots, \mathsf{Coeffs}(\mathbf{b}_{n_b-1})^\top\big) \;=\; \tilde{\mathbf{s}}^\top \mathbf{A} + \tilde{\mathbf{e}}^\top + \lfloor Q/p \rfloor\, \mathbf{u}_{i^*}^\top \;\in\; \mathbb{Z}_Q^{\ell_1}, \quad \text{(A.1)}$$

where $\mathbf{u}_{i^*}$ is the i^*-th standard basis vector. Thus $\mathbf{q} = \mathbf{c}$.

$Q \to W$ body modswitch. The client rounds $q'_i = \lfloor (W/Q)\, c_i \rceil \in \mathbb{Z}_W$, equivalently $(Q/W)\, q'_i = c_i - \epsilon_{b,i} \pmod Q$ with $|\epsilon_{b,i}| \le Q/(2W)$.

NTT hint: three noise sources. The server computes $r_j = \sum_i \mathrm{db}[i,j]\, q'_i \bmod W$ and reverse-modswitches $r'_j = \lfloor (Q/W)\, r_j \rceil$ with rounding $|\epsilon_{r,j}| \le 1/2$. Every W-wrap in the scan contributes a multiple of Q to $(Q/W)\, r_j$ (since $W > Q$) and vanishes $(\bmod Q)$. Substituting (A.1) and writing $\mathbf{H}[\cdot, j] = \mathbf{A}^\top \mathrm{db}[:, j] \in \mathbb{Z}_Q^d$ for the NTT hint column,

$$r'_j - \tilde{\mathbf{s}}^\top \mathbf{H}[\cdot, j] \;=\; \lfloor Q/p \rfloor\, \mathrm{db}[i^*, j] + e^{\mathrm{NTT}}_{\mathsf{scan},j} \pmod Q, \quad \text{(A.2)}$$

where

$$e^{\mathrm{NTT}}_{\mathsf{scan},j} \;=\; \underbrace{\tilde{\mathbf{e}}^\top \mathrm{db}[:, j]}_{\text{RLWE error}} - \underbrace{\sum_i \mathrm{db}[i,j]\, \epsilon_{b,i}}_{Q \to W \text{ rounding through scan}} + \underbrace{\epsilon_{r,j}}_{W \to Q \text{ rounding}}. \quad \text{(A.3)}$$

Each $\tilde{e}_i$ is sub-Gaussian with parameter σ_e; each $\epsilon_{b,i}$ is mean-zero and bounded by $Q/(2W)$ (Hoeffding width $(Q/W)\sqrt{2\pi}/2$); $\epsilon_{r,j}$ is mean-zero and bounded by $1/2$ (width $\sqrt{2\pi}/2$); and $|\mathrm{db}[i,j]| \le p/2$. Summing independent contributions:

$$\sigma^2_{\mathsf{scan},\mathrm{NTT}} \;\le\; \ell_1 (p/2)^2 \sigma_e^2 \;+\; (Q/W)^2\, \ell_1\, (p/2)^2\, \tfrac{2\pi}{4} \;+\; \tfrac{2\pi}{4}. \quad \text{(A.4)}$$

GEMM hint: two additional secret-dependent noise sources. The symmetric alternative (Section 4.3.1) pre-modswitches each mask coordinate to $\mathbb{Z}_W$: $a'_{i,k} = \lfloor (W/Q)\, a_{i,k} \rceil = (W/Q)\, a_{i,k} + \epsilon_{a,i,k}$ with $|\epsilon_{a,i,k}| \le 1/2$. It then computes the hint

as a $\mathbb{Z}_W$ GEMM $H^{\mathrm{GEMM}}_{j,k} = \sum_i \mathrm{db}[i,j]\, a'_{i,k} \bmod W$ and reverse-modswitches each entry back to $\mathbb{Z}_Q$: $H'_{j,k} = \lfloor (Q/W)\, H^{\mathrm{GEMM}}_{j,k} \rceil = (Q/W)\, H^{\mathrm{GEMM}}_{j,k} + \epsilon_{H,j,k}$ with $|\epsilon_{H,j,k}| \le 1/2$. Substituting $a'_{i,k}$ and rescaling,

$$(Q/W)\, H^{\mathrm{GEMM}}_{j,k} \;=\; \sum_i \mathrm{db}[i,j]\, a_{i,k} \;+\; (Q/W) \sum_i \mathrm{db}[i,j]\, \epsilon_{a,i,k} \pmod{Q},$$

so $\tilde{\mathbf{s}}^\top \mathbf{H}'_j$ picks up two extra noise terms relative to $\tilde{\mathbf{s}}^\top \mathbf{H}[\cdot, j]$:

$$\tilde{\mathbf{s}}^\top \mathbf{H}'_j \;=\; \tilde{\mathbf{s}}^\top \mathbf{H}[\cdot, j] \;+\; (Q/W) \sum_i \mathrm{db}[i,j]\, \langle \tilde{\mathbf{s}}, \boldsymbol{\epsilon}_{a,i} \rangle \;+\; \langle \tilde{\mathbf{s}}, \boldsymbol{\epsilon}_{H,j} \rangle \pmod{Q}, \tag{A.5}$$

and thus

$$e^{\mathrm{GEMM}}_{\mathsf{scan},j} \;=\; e^{\mathrm{NTT}}_{\mathsf{scan},j} \;-\; (Q/W) \sum_i \mathrm{db}[i,j]\, \langle \tilde{\mathbf{s}}, \boldsymbol{\epsilon}_{a,i} \rangle \;-\; \langle \tilde{\mathbf{s}}, \boldsymbol{\epsilon}_{H,j} \rangle. \tag{A.6}$$

The two new terms inner-product a bounded rounding vector with the random secret $\tilde{\mathbf{s}} \sim D(\sigma_s)^d$. Each $\langle \tilde{\mathbf{s}}, \boldsymbol{\epsilon}_{a,i} \rangle$ has width $(1/2)\sigma_s\sqrt{d}$ (coord-wise bound $1/2$, inner product of length d), and the weighted sum over ℓ_1 rows scales it by $\sqrt{\ell_1} \cdot (p/2)$; $\langle \tilde{\mathbf{s}}, \boldsymbol{\epsilon}_{H,j} \rangle$ has width $(1/2)\sigma_s\sqrt{d}$ (one copy, not summed). Adding to (A.4),

$$\sigma^2_{\mathsf{scan},\mathrm{GEMM}} \;\le\; \sigma^2_{\mathsf{scan},\mathrm{NTT}} \;+\; \underbrace{(Q/W)^2\, \ell_1\, (p/2)^2\, \tfrac{d\sigma_s^2}{4}}_{\text{mask rounding} \times s \text{ through scan}} \;+\; \underbrace{\tfrac{d\sigma_s^2}{4}}_{\text{hint rounding} \times s}. \tag{A.7}$$

The mask-rounding term scales as ℓ_1—it is the $\Theta(\ell_1)$ secret-dependent blow-up that drives the GEMM hint's failure probability off a cliff as the database grows (Section 4.3.3, Table A.2). The NTT hint leaves $\mathbf{a}_i$ in $\mathbb{Z}_Q$, so $\boldsymbol{\epsilon}_{a,i} = 0$ and $\boldsymbol{\epsilon}_{H,j} = 0$; both extra terms vanish.

Scalar LWE form. For $\bullet \in \{\mathrm{NTT}, \mathrm{GEMM}\}$, with $\mathbf{H}^\bullet[\cdot, j]$ denoting the corresponding hint column, the pair $(\mathbf{H}^\bullet[\cdot, j], r'_j) \in \mathbb{Z}_Q^d \times \mathbb{Z}_Q$ is a scalar LWE ciphertext encrypting $\mathrm{db}[i^*, j]$ under $\tilde{\mathbf{s}}$ with plaintext scale $\lfloor Q/p \rfloor$ and noise parameter $\sigma_{\mathsf{scan},\bullet}$.

A.2 Stage 2: InspiRING packing

For each output $o \in [\rho]$, the d scalar LWE ciphertexts $\{(\mathbf{H}^{\bullet}[\cdot, od{+}k], r'_{od+k})\}_{k\in[d]}$ encrypt the messages $m_k = \mathrm{db}[i^*, od+k]$ under $\tilde{\mathbf{s}}$ with per-entry noise $\sigma_{\mathsf{scan},\bullet}$. Invoking the InspiRING correctness analysis (Mahdavi et al., 2025, Appendix) applied to Construction 2.2 with (R_Q, z, t), $\mathsf{InspiRING.Pack}$ produces $(a_o, b_o) \in R_Q^2$ such that

$$b_o - s \cdot a_o \;=\; \lfloor Q/p \rfloor\, \mathsf{row}_o + e^{\bullet}_{\mathsf{pack},o} \pmod{Q}, \tag{A.8}$$

where $\mathsf{row}_o = \sum_{k\in[d]} \mathrm{db}[i^*, od+k]\, x^k \in R_p$. Each coefficient of $e^{\bullet}_{\mathsf{pack},o}$ aggregates the d scan-noise inputs through the Aggregate stage and adds a gadget-digit term from the Collapse stage's $d-1$ key-switches (each t independent digits in $(-z/2, z/2]$, inner-producted with $\tilde{\mathbf{s}}$):

$$\sigma^2_{\mathsf{pack},\bullet} \;\le\; \sigma^2_{\mathsf{scan},\bullet} \;+\; \tfrac{z^2\,\sigma_s^2\,t\,d\,(d-1)}{4}. \tag{A.9}$$

A.3 Stage 3: Two-target modswitch

Applying Lemma 2.3 to (a_o, b_o) with targets (q_{21}, q_{22}), the server's response is $(a'_o, b'_o) \in R_{q_{21}} \times R_{q_{22}}$, and the client recovers via Construction 4.1's $\mathsf{Recover}$. Writing out the algebra,

$$v'_o \;=\; \lfloor q_{22}/p \rfloor\, \mathsf{row}_o + e_{1,o} + e^{\bullet}_{2,o} \pmod{q_{22}},$$

where $e_{1,o}$ collects the deterministic offsets and $e^{\bullet}_{2,o}$ the stochastic noise.

Deterministic bound (goes into τ). Two bounded contributions aggregate into $e_{1,o}$:

- the message-rescaling error from $(q_{22}/Q)\lfloor Q/p \rfloor \neq \lfloor q_{22}/p \rfloor$, giving $|\cdot| \le \frac{1}{2}\big((q_{22} \bmod p) + (q_{22}/Q)(Q \bmod p)\big)$ per coefficient;
- the body switch rounding $\epsilon_{b,o}$ introduced by $b'_o = \lfloor b_o \rceil_{Q,q_{22}}$, bounded $|\epsilon_{b,o}| \le 1/2$ per coefficient, absorbed deterministically rather than summed into σ^2.

Bounding worst-case,

$$\|e_{1,o}\|_\infty \;\le\; \tfrac{1}{2}\big(2 + (q_{22} \bmod p) + (q_{22}/Q)(Q \bmod p)\big), \tag{A.10}$$

which subtracts from the codeword half-gap $\lfloor q_{22}/p \rfloor/2$ to give τ (Equation 4.9). Treating $\epsilon_{b,o}$ deterministically rather than via Hoeffding avoids a $\sigma_\epsilon^2 = 2\pi/4 = \pi/2$ per-coefficient contribution that would dominate σ^2_{NTT} and collapse the bound; the deterministic route trades a $1/2$ subtraction from τ for a $\pi/2$ addition to σ^2, a favorable exchange at our parameters (Section 2.7).

Stochastic variance. Each coefficient of $e^{\bullet}_{2,o}$ is sub-Gaussian with parameter

$$\sigma^2_\bullet \;\le\; \underbrace{(q_{22}/q_{21})^2\, \tfrac{d\,\sigma_s^2}{4}}_{\text{final-MS mask} \times s} + (q_{22}/Q)^2\, \sigma^2_{\mathsf{pack},\bullet}. \tag{A.11}$$

The first term is the $Q \to q_{21}$ mask rounding $\boldsymbol{\epsilon}_a \in [-1/2, 1/2]^d$ (in the q_{21} domain) inner-producted with $\tilde{\mathbf{s}}$ and rescaled to the q_{22} decoding domain by the (q_{22}/q_{21}) factor; the second rescales the pre-modswitch packing noise $\sigma^2_{\mathsf{pack},\bullet}$ by $(q_{22}/Q)^2$.

Expanding $\sigma^2_{\mathsf{pack,NTT}}$ via (A.9) and $\sigma^2_{\mathsf{scan,NTT}}$ via (A.4) recovers the five-term closed form of Lemma 4.3 (4.10). For the GEMM hint, further expanding the two extra scan terms from (A.7) gives

$$\sigma^2_{\mathrm{GEMM}} \;=\; \sigma^2_{\mathrm{NTT}} \;+\; \underbrace{(q_{22}/W)^2\, \ell_1\, (p/2)^2\, \tfrac{d\,\sigma_s^2}{4}}_{\text{mask rounding} \times s \text{ through scan}} \;+\; \underbrace{(q_{22}/Q)^2\, \tfrac{d\,\sigma_s^2}{4}}_{\text{hint rounding} \times s}, \tag{A.12}$$

using $(q_{22}/Q)^2(Q/W)^2 = (q_{22}/W)^2$ for the mask-through-scan term. The GEMM hint inflates the NTT-hint variance by two secret-dependent terms; the ℓ_1-linear mask-through-scan term dominates at all deployment scales and determines the GEMM entries of Table A.2.

A.4 Stage 4: Recovery and tail bound

$\mathsf{Recover}$ computes $v_o = \lfloor v'_o \rceil_{q_{22},p} \in R_p$, applying the decoder $\lfloor (p/q_{22}) \cdot \rceil$ coefficient-wise. For any coefficient $z \in [d]$, $v_o[z] = \mathsf{row}_o[z] = \mathrm{db}[i^*, od + z]$ whenever $|(e_{1,o} +$

$e^{\bullet}_{2,o})[z]| \le \tau$ (Equation 4.9): $\lfloor q_{22}/p \rfloor/2$ is half the distance between adjacent codewords in $\mathbb{Z}_{q_{22}}$, and subtracting the deterministic $\|e_{1,o}\|_\infty$ term yields the margin. At the parameters of Table 4.1 ($q_{22} = 2^{10}$, $p = 256$), $\lfloor q_{22}/p \rfloor/2 = 2$, $q_{22} \bmod p = 0$, and $(q_{22}/Q)(Q \bmod p) < 1$, so $\tau = 1$.

Applying a sub-Gaussian tail bound to $e^{\bullet}_{2,o}$ and a union bound over the $d\rho$ output coefficients,

$$\Pr\big[\, v_o[z] = \mathrm{db}[i^*, od + z] \text{ for all } o \in [\rho],\, z \in [d] \,\big] \;\ge\; 1 - 2\,d\,\rho\,\exp\!\big(-\pi\,\tau^2/\sigma^2_{\bullet}\big). \quad \text{(A.13)}$$

Taking $\log_2$ recovers Equation (4.8) with $\sigma^2_{\bullet}$ in place of σ^2_{NTT}: Lemma 4.3 is the $\bullet = \mathrm{NTT}$ case. For the symmetric hint, substitute σ^2_{GEMM} from (A.12). □

A.5 Noise Budget at Concrete Parameters

Table A.1 evaluates each term of (4.10) at Wikipedia scale ($\ell_1 = 2^{16}$, $\rho = 64$, $\approx 8\,\mathrm{GiB}$) under the parameters of Table 4.1.

Table A.1: Per-term noise contributions at Wikipedia scale ($\ell_1 = 2^{16}$, $\rho = 64$).

#	Term	Value	% of σ^2_{NTT}
1	Final MS: $s\times$ mask rounding	1.2×10^{-2}	33.2
2	RLWE error e_i through scan	9.6×10^{-5}	0.3
3	Packing: $s\times$ gadget rounding	2.5×10^{-2}	66.3
4	Body ϵ_b ($Q \to W$) through scan	9.6×10^{-5}	0.3
5	Body ϵ_r ($W \to Q$)	8.9×10^{-14}	≈ 0
	σ^2_{NTT}	0.037	
	$\log_2 \delta$	$\mathbf{-105}$	

Packing (Term 3, 66.3%) and the final modswitch (Term 1, 33.2%) account for essentially all of σ^2_{NTT}; sandwich body roundings are negligible (Terms 4–5, $<$ 0.3%). The two terms the symmetric GEMM hint would introduce—mask rounding through the scan and hint rounding inner-producted with the secret—are absent here by construction (Section 4.3.2).

Table A.2 extends the evaluation to larger databases, comparing the asymmetric (NTT) hint of Section 4.3.2 to the symmetric (GEMM) alternative of Section 4.3.1.

Table A.2: Failure probability $\log_2 \delta$ across database sizes, NTT hint vs GEMM hint.

DB size	ℓ_1	ρ	NTT	GEMM
4 GiB	2^{16}	32	-106	-36
8 GiB (Wikipedia)	2^{16}	64	-105	-35
16 GiB	2^{17}	64	-104	-16
64 GiB	2^{18}	128	-103	-0.4
1 TiB	2^{20}	512	-97	fail

With the NTT hint, SandwichPIR maintains $\log_2 \delta < -97$ even at 1 TiB; the GEMM hint's headroom collapses as ℓ_1 grows, reaching -0.4 at 64 GiB (one in two queries fails) and exceeding the sub-Gaussian bound beyond.

Appendix B

Private Wikipedia User Interface

This appendix presents screenshots of the Private Wikipedia web client described in Chapter 6. The UI is a vanilla-JavaScript page that loads the brotli-compressed article index and the WebAssembly-compiled PIR client on first load (Figure B.1); users type a title prefix to autocomplete against the 6.4M-entry index (Figure B.2); selecting an article triggers a PIR query whose decrypted result—including the first-query 288 KiB mask download—renders in place (Figure B.3); and a subsequent query from the same session returns body-only under lazy hint delivery (Figure B.4). All screenshots are from a deployment running on a TACC Lonestar6 A100 node serving the 8 GiB Wikipedia database of Chapter 6.

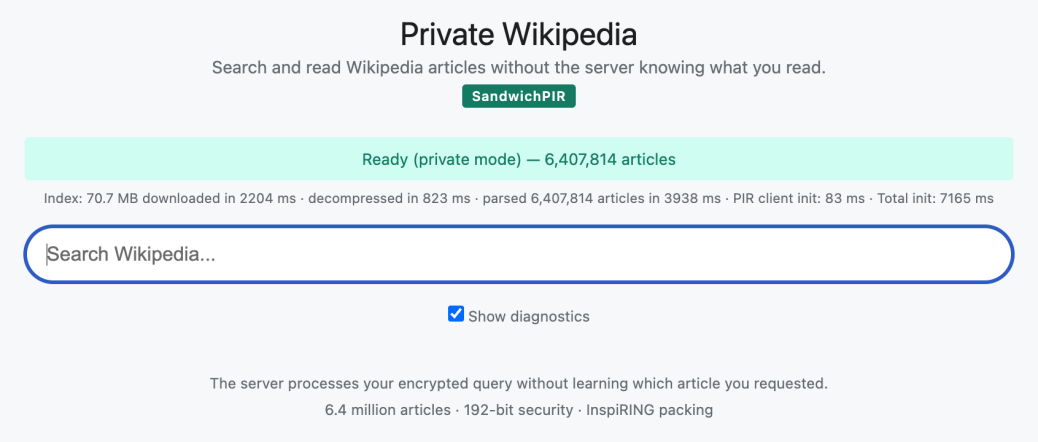

Figure B.1: Private Wikipedia on page load. The WebAssembly client has initialized, the 6.4M-entry title index has been fetched and brotli-decompressed, and the PIR client has derived its CRS-expanded state from the shared seed. The search box is now ready to accept a query.

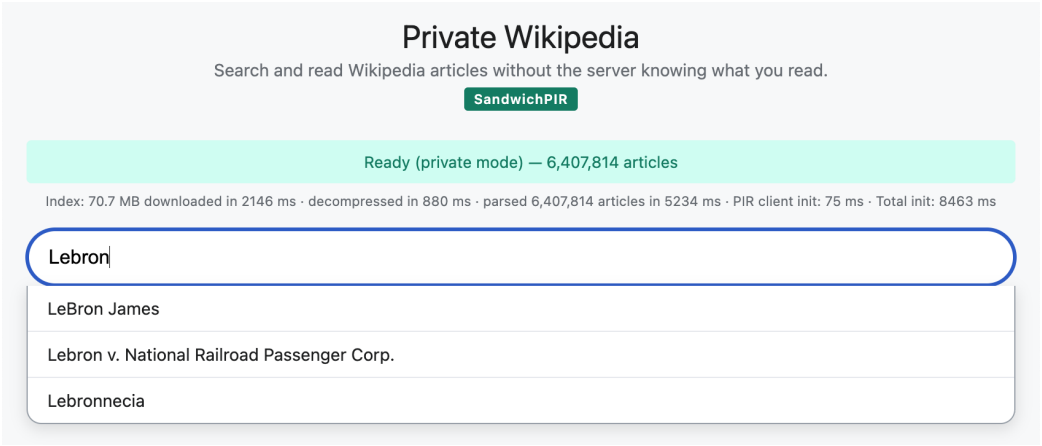

Figure B.2: Title-prefix autocomplete. The browser binary-searches the client-side index—no network traffic—and displays matching article titles as the user types.

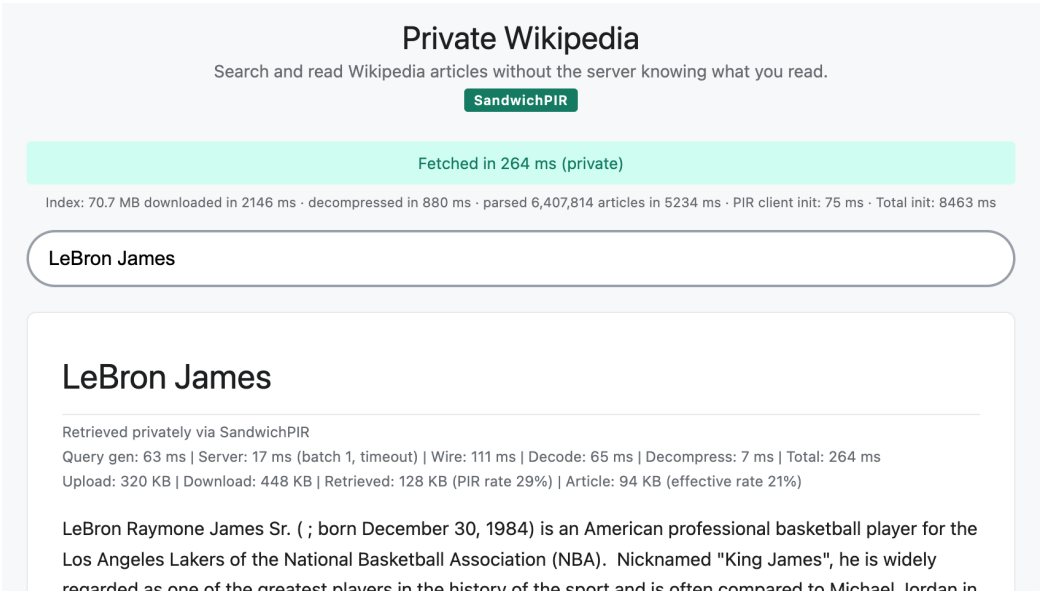

Figure B.3: First-query result after the user selects a title. The client encoded the target row as an RLWE query, generated the InspiRING packing keys, uploaded the query (320 KiB), downloaded the 448 KiB response (CRS-derived mask + body), decrypted, and rendered the article in place. The diagnostics panel shows per-component timings (query generation, upload, server compute, download, decryption).

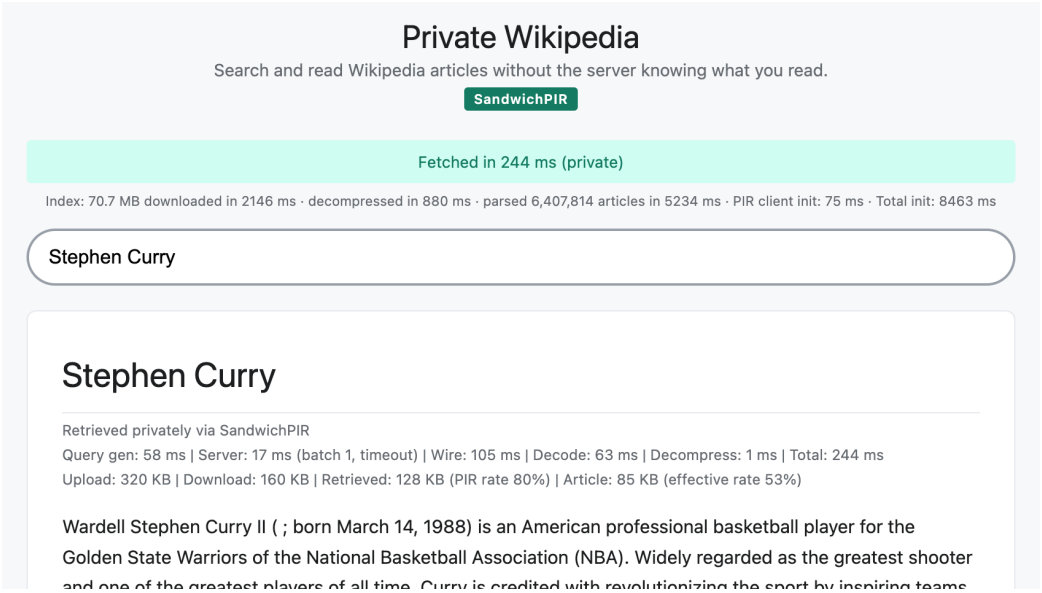

Figure B.4: Subsequent-query result (same session). The mask from the first query has been cached locally, so the response is body-only (160 KiB)—the lazy hint delivery regime (Section 4.2). The diagnostics panel reflects the smaller download and correspondingly lower end-to-end latency, matching the subsequent-query budget of Figure 6.2.

References

Jyrki Alakuijala, Andrea Farruggia, Paolo Ferragina, Eugene Kliuchnikov, Robert Obryk, Zoltan Szabadka, and Lode Vandevenne. Brotli: A general-purpose data compressor. *ACM Trans. Inf. Syst.*, 37(1), December 2018. ISSN 1046-8188. doi: 10.1145/3231935. URL https://doi.org/10.1145/3231935.

Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015. doi: doi:10.1515/jmc-2015-0016. URL https://doi.org/10.1515/jmc-2015-0016.

A. Ali, T. Lepoint, S. Patel, M. Raykova, P. Schoppmann, K. Seth, and K. Yeo. Communication–computation trade-offs in pir. In *USENIX Security Symposium*, pages 1811–1828, 2021. URL https://eprint.iacr.org/2019/1483.pdf.

Apple. Understanding how Live Caller ID Lookup preserves privacy. https://developer.apple.com/documentation/identitylookup/understanding-how-live-caller-id-lookup-preserves-privacy, 2024.

Benny Applebaum, David Cash, Chris Peikert, and Amit Sahai. Fast cryptographic primitives and circular-secure encryption based on hard learning problems. In *Proceedings of the 29th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '09, page 595–618, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 9783642033551. doi: 10.1007/978-3-642-03356-8_35. URL https://doi.org/10.1007/978-3-642-03356-8_35.

Ran Canetti, Justin Holmgren, and Silas Richelson. Towards doubly efficient private information retrieval. In Yael Kalai and Leonid Reyzin, editors, *Theory of Cryptography*, pages 694–726, Cham, 2017. Springer International Publishing. ISBN 978-3-319-70503-3.

Hao Chen, Wei Dai, Miran Kim, and Yongsoo Song. Efficient homomorphic conversion between (ring) LWE ciphertexts. In *ACNS*, pages 460–479, 2021. doi: 10.1007/978-3-030-78372-3_18.

B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*, pages 41–50, 1995. doi: 10.1109/SFCS.1995.492461.

Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013*, pages 75–92, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-40041-4.

Trinabh Gupta, Natacha Crooks, Whitney Mulhern, Srinath Setty, Lorenzo Alvisi, and Michael Walfish. Scalable and private media consumption with popcorn. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 91–107, Santa Clara, CA, March 2016. USENIX Association. ISBN 978-1-931971-29-4. URL `https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/gupta-trinabh`.

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 2023a.

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. In *32nd USENIX Security Symposium (USENIX Security 23)*, Anaheim, CA, August 2023b. USENIX Association. URL `https://www.usenix.org/conference/usenixsecurity23/presentation/henzinger`.

Eyal Kushilevitz and Rafail M. Ostrovsky. Replication is not needed: single database, computationally-private information retrieval. *Proceedings 38th Annual Symposium on Foundations of Computer Science*, pages 364–373, 1997. URL https://api.semanticscholar.org/CorpusID:11000506.

Ryan Lehmkuhl, Alexandra Henzinger, and Henry Corrigan-Gibbs. Distributional private information retrieval. In *34th USENIX Security Symposium (USENIX Security 25)*, Seattle, WA, 2025. USENIX Association.

Baiyu Li, Daniele Micciancio, Mariana Raykova, and Mark Schultz-Wu. Hintless single-server private information retrieval. In *CRYPTO*, pages 183–217. Springer-Verlag, 2024. doi: 10.1007/978-3-031-68400-5_6.

Patrick Longa and Michael Naehrig. Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In *Cryptology and Network Security: 15th International Conference, CANS 2016, Milan, Italy, November 14-16, 2016, Proceedings*, page 124–139, Berlin, Heidelberg, 2016. Springer-Verlag. ISBN 978-3-319-48964-3. doi: 10.1007/978-3-319-48965-0_8. URL https://doi.org/10.1007/978-3-319-48965-0_8.

Vadim Lyubashevsky, Chris Peikert, and Oded Regev. *On Ideal Lattices and Learning with Errors over Rings*, volume 60, pages 1–23. 05 2010. ISBN 978-3-642-13189-9. doi: 10.1007/978-3-642-13190-5_1.

Rasoul Akhavan Mahdavi, Sarvar Patel, Joon Young Seo, and Kevin Yeo. InsPIRe: Communication-efficient PIR with server-side preprocessing. Cryptology ePrint Archive, Paper 2025/1352, 2025. URL https://eprint.iacr.org/2025/1352.

Samir Jordan Menon and David J Wu. Spiral: Fast, high-rate single-server pir via fhe composition. In *2022 IEEE symposium on security and privacy (SP)*, pages 930–947. IEEE, 2022.

Samir Jordan Menon and David J. Wu. YPIR: High-Throughput Single-Server PIR with silent preprocessing. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 5985–6002, Philadelphia, PA, August 2024. USENIX Association. ISBN 978-1-939133-44-1. URL https://www.usenix.org/conference/usenixsecurity24/presentation/menon.

Muhammad Haris Mughees, Hao Chen, and Ling Ren. Onionpir: Response efficient single-server pir. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 2292–2306, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384544. doi: 10.1145/3460120.3485381. URL https://doi.org/10.1145/3460120.3485381.

NVIDIA Corporation. NVIDIA gpu datasheets and whitepapers. https://resources.nvidia.com. V100, T4, L4, and A100 product datasheets.

Ookla. Speedtest Global Index: United States median country speeds. https://www.speedtest.net/global-index/united-states, February 2026. Median mobile: 211.84 Mbps down, 12.03 Mbps up, 26 ms. Median fixed broadband: 308.11 Mbps down, 55.83 Mbps up, 13 ms.

Alisah Ozcan, Arsalan Javeed, and Erkay Savas. High-performance number theoretic transform on gpu through radix2-ct and 4-step algorithms. *IEEE Access*, 13:87862–87883, 2025. doi: 10.1109/ACCESS.2025.3570024.

Chris Peikert. A decade of lattice cryptography. *Found. Trends Theor. Comput. Sci.*, 10(4):283–424, March 2016. ISSN 1551-305X. doi: 10.1561/0400000074. URL https://doi.org/10.1561/0400000074.

Chris Peikert, Vinod Vaikuntanathan, and Brent Waters. A framework for efficient and composable oblivious transfer. In *Proceedings of the 28th Annual Conference on Cryptology: Advances in Cryptology*, CRYPTO 2008, page

554–571, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 9783540851738. doi: 10.1007/978-3-540-85174-5_31. URL https://doi.org/10.1007/978-3-540-85174-5_31.

Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '05, page 84–93, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1581139608. doi: 10.1145/1060590.1060603. URL https://doi.org/10.1145/1060590.1060603.

Ardianto Satriawan, Infall Syafalni, Rella Mareta, Isa Anshori, Wervyan Shalannanda, and Aleams Barra. Conceptual review on number theoretic transform and comprehensive review on its implementations. *IEEE Access*, 11:70288–70316, 2023. doi: 10.1109/ACCESS.2023.3294446.

Vijay Thakkar, Pradeep Ramani, Cris Cecka, Aniket Shivam, Honghao Lu, Ethan Yan, Jack Kosaian, Mark Hoemmen, Haicheng Wu, Andrew Kerr, Matt Nicely, Duane Merrill, Dustyn Blasig, Aditya Atluri, Fengqi Qiao, Piotr Majcher, Paul Springer, Markus Hohnerbach, Jin Wang, and Manish Gupta. CUTLASS, January 2023. URL https://github.com/NVIDIA/cutlass.

Wikimedia. Wikimedia downloads, a. URL https://dumps.wikimedia.org.

Wikimedia. Wikipedia: Statistics, b. URL https://en.wikipedia.org/wiki/Wikipedia:Statistics. Average >4,500 page views/sec; >2 edits/sec on the English Wikipedia.

Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009. ISSN 0001-0782. doi: 10.1145/1498765.1498785. URL https://doi.org/10.1145/1498765.1498785.

Kevin Yeo and Sarvar Patel. Protecting your device information with private set membership. https://security.googleblog.com/2021/10/protecting-your-device-information-with.html, 2021.