

Conflux: A High-Performance Keyword Private Retrieval System for Dynamic Datasets

Zehao Chen^{*†‡}, Zhaoyan Shen[§], Qian Wei[§], Hang Lu^{¶||}, Lei Ju^{†*}

^{*}School of Cyber Science and Technology, Shandong University, Qingdao, China

[†]Quan Chen Laboratory, Jinan, China

[‡]State Key Laboratory of Cryptography and Digital Economy Security, Shandong University, Qingdao, China

[§]School of Computer Science and Technology, Shandong University, Qingdao, China

[¶]SKLP, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China

^{||}Zhongguancun Laboratory, Beijing, China

{czh-rot,wei.qian}@mail.sdu.edu.cn, luhang@ict.ac.cn, {shenzhaoyan,julei}@sdu.edu.cn

Abstract—Homomorphic Encryption (HE)–based Private Information Retrieval (PIR) allows clients to retrieve plaintext records from untrusted servers without revealing query content. While promising in theory, existing solutions fall short in practice due to two fundamental limitations: (1) poor support for dynamic datasets, which limits applicability in real-world, evolving workloads; and (2) excessive I/O overhead from full-database scans. These bottlenecks prevent current designs from bridging the gap between cryptographic privacy and system efficiency.

In this paper, we introduce Conflux, an efficient keyword PIR system for dynamic data environments through a protocol-architecture co-design approach. At the protocol level, Conflux employs a novel two-phase retrieval mechanism, consisting of an oblivious filtering phase followed by a precise retrieval phase. This design natively supports efficient online insertions, deletions, and updates, while maintaining near-optimal computational complexity. At the system level, Conflux adopts a heterogeneous accelerator architecture that tightly couples computational storage devices and incorporates software–hardware co-optimization techniques to mitigate I/O bottlenecks. Experimental results show that Conflux reduces query processing time by up to 2.64× compared to the state-of-the-art methods, while retaining full support for dynamic datasets.

I. INTRODUCTION

The proliferation of cloud computing and outsourced data services has intensified the demand for query privacy, particularly in domains such as healthcare, finance, and cybersecurity [1]–[4], where revealing the query itself may inadvertently expose sensitive or proprietary information. As clients increasingly interact with public, remote databases, safeguarding the confidentiality of query content becomes just as critical as protecting the data itself. Private Information Retrieval (PIR) [5] is a promising solution that enables clients to issue encrypted queries to public cloud databases and retrieve the corresponding plaintext entries, without revealing which records are being accessed.

Fundamentally, PIR adheres to an “all-for-one” model [6], wherein each client query uniformly accesses all entries in the cloud databases to conceal access patterns. The earliest PIR schemes, referred to as Index PIR [7]–[13], are designed to support private queries over structured datasets. However,

they assume that the client knows the exact index of the target entry in advance, which is ill-suited in modern cloud-based environments where queries are typically keyword-driven and clients rarely possess corresponding indices. To bridge this gap, numerous studies have extended Index PIR into the domain of Keyword PIR [14]. Systems such as Pantheon [15] and CwPIR [16] propose homomorphic equality operators to obliviously compute the index associated with the queried keyword. Once the index is determined, traditional Index PIR is used to retrieve the desired data. More recent efforts, including MulPIR [17] and SparsePIR [18], further optimize this process by precomputing a keyword-to-index mapping. These optimizations bypass the costly equality operators, allowing the server to invoke Index PIR for data retrieval directly.

Despite recent advances, existing Keyword PIR solutions continue to face two fundamental bottlenecks that impede practical deployment: (1) **A significant trade-off persists between performance and support for dynamic data.** Systems like Pantheon [15] and CwPIR [16], which support dynamic updates via homomorphic equality operations, suffer from computational overheads up to 100× higher than traditional Index PIR. Conversely, mapping-based approaches [17], [18] offer high query performance but lack efficient mechanisms for supporting dynamic data. Even a single update or insertion necessitates a full mapping reconstruction. In clinical [19] or financial [20] environments with frequent updates, this overhead quickly becomes prohibitive, rendering such schemes impractical (see § III-A). (2) **I/O cost becomes a major bottleneck at scale.** Because PIR fundamentally requires each query to access the entire database, I/O throughput limits system efficiency. Preliminary experiments indicate that in MulPIR and SparsePIR, I/O operations account for over 60% of the total online query latency, severely restricting response speed and scalability (in § III-B).

In this paper, we introduce Conflux, a novel Keyword PIR system that reconsiders both protocol design and system architecture to enable efficient, scalable, and update-resilient private retrieval. At the protocol level, Conflux discards rigid keyword-to-index mappings and instead adopts a two-phase retrieval mechanism: In Phase 1, a public hash function maps

Lei Ju is the corresponding author.

all keywords to corresponding buckets (i.e., an approximate location), from which a candidate set is privately retrieved using Index PIR; In Phase 2, equality operators are applied with the candidate set to precisely retrieve the target entry. This approach substantially reduces the computational burden associated with equality operators while naturally supporting dynamic operations, including updates, insertions, and deletions, without requiring costly global remapping. Furthermore, Conflux introduces a non-blocking adaptive tuning mechanism that continuously optimizes the keyword-to-bucket mapping. This allows the system to adapt to changes in dataset scale while maintaining uninterrupted availability.

At the system architecture level, Conflux diverges from traditional PIR systems that rely solely on GPU acceleration by embracing a heterogeneous hardware design. It integrates computational storage devices (CSDs) [21] and GPUs, each optimized for distinct stages of the protocol. Specifically, Conflux offloads Phase 1 to CSDs, enabling initial candidate selection to occur near the storage layer. This near-data processing ensures that only the candidate set with a small scale is transferred to the host, significantly reducing data movement overhead. The subsequent Phase 2, which involves computationally intensive equality checks, is accelerated on the GPU, leveraging its massive parallelism for high-throughput computation. To fully exploit this division of labor, Conflux incorporates additional architectural optimizations, including inter-device pipelining and load-balanced data distribution across different CSDs. These enhancements maximize throughput and minimize redundant transfers, ensuring high efficiency even at scale.

We implement the open-source Conflux system and evaluate it on six Facebook datasets [22], systematically comparing its performance and scalability against a range of existing Keyword PIR baselines [15], [17], [18]. Experimental results demonstrate that Conflux delivers up to a $62.7\times$ performance improvement over dynamic-supporting Keyword PIR systems. Furthermore, compared to those schemes that lack update support, Conflux also reduces the online query latency by $2.64\times$ while fully retaining compatibility with dynamic datasets.

The contributions of this paper are summarized as follows:

- To the best of our knowledge, this is the first Keyword PIR system that natively supports dynamic datasets while maintaining near-Index PIR performance.
- We propose the Conflux protocol, which eliminates static mappings through a two-phase retrieval mechanism, substantially improving both efficiency and flexibility in dynamic database environments.
- We design a protocol-aware heterogeneous accelerator that leverages high internal bandwidth and pipelined execution across CSDs and GPUs to optimize the performance during retrieval.
- We demonstrate Conflux’s superior performance over the baselines via a thorough evaluation.

In the rest of this paper, we first present the background of PIR systems in § II, and then outline our motivation in § III. Next, we proceed with detailed descriptions of the protocol

TABLE I: Execution time (ms) of BFV operations on an NVIDIA A100 GPU ($N = 2^{15}$).

Operation	Input	Output	Execution Time
Add/Sub	$\llbracket x \rrbracket, \llbracket x \rrbracket$	$\llbracket x \pm x \rrbracket$	0.005
PMult	$x, \llbracket x \rrbracket$	$\llbracket x \cdot x \rrbracket$	0.144
Rotate	$\llbracket x \rrbracket, k$	$\phi_k(\llbracket x \rrbracket)$	0.78
Mult	$\llbracket x \rrbracket, \llbracket x \rrbracket$	$\llbracket x \cdot x \rrbracket$	1.01
Bootstrapping	$\llbracket x \rrbracket$	$\llbracket x \rrbracket_{BS}$	> 200

and system architecture in § IV and § V, respectively, and conclude with evaluation results in § VI.

II. BACKGROUND

A. Fully Homomorphic Encryption

Fully Homomorphic Encryption (FHE) [23]–[25] is a cryptographic primitive that enables computation over encrypted data, and is commonly used as a core building block in private information retrieval (PIR) systems. In this paper, we focus on the BFV [25] scheme among various FHE schemes, due to its widespread adoption in existing PIR solutions.

Data Structure. In the BFV scheme, a plaintext x is represented as a vector of degree N , where $N \in \{2^{10}, 2^{11}, \dots, 2^{15}\}$. Each element within this vector is an integer modulo a smaller plaintext modulus T , i.e., each element $< T$. Using a public key pk , the function `BFV.encrypt` transforms a plaintext vector x into a ciphertext $\llbracket x \rrbracket$, which internally consists of two polynomial vectors modulo a larger ciphertext modulus Q . This gap between T and Q ensures both correctness and security. Using a private key sk and `BFV.decrypt` function, the ciphertext $\llbracket x \rrbracket$ can be decrypt back to x . Assume $N = 4$, x is $[1, 2, 3, 4]$, Table I shows different BFV operations and their computation costs.

• **Add (+) / Sub (-):** Operates on two ciphertexts and returns their sum or difference. $\llbracket x \rrbracket + \llbracket x \rrbracket = \llbracket x + x \rrbracket$, yields encryptions of $[2, 4, 6, 8]$.

• **PMult (·):** Multiplies a ciphertext with a plaintext and returns a ciphertext. $\llbracket x \rrbracket \cdot x = \llbracket x \cdot x \rrbracket$, encrypts $[1, 4, 9, 16]$.

• **Mult (·):** Multiplies two ciphertexts, producing a ciphertext of their component-wise product. $\llbracket x \rrbracket \cdot \llbracket x \rrbracket = \llbracket x \cdot x \rrbracket$ also encrypts $[1, 4, 9, 16]$.

• **Rotate (ϕ):** Applies a cyclic rotation to the encrypted vector. Given a ciphertext $\llbracket x \rrbracket$ and rotation index $k = 1$, the operation $\phi_1(\llbracket x \rrbracket)$ yields an encryption of $[4, 1, 2, 3]$ for a right-rotation by 1.

• **Bootstrapping ($_{BS}$):** In the BFV scheme, each ciphertext carries an inherent noise term that grows with the number of sequential `PMult` and `Mult` operations, i.e., with increasing multiplicative depth. Once the accumulated noise exceeds a certain threshold, the ciphertext becomes undecryptable. Therefore, `Bootstrapping`, as the most expensive operation, is used to reduce the noise level inside ciphertexts, thereby restoring decryptability and enabling continued computation. Notably, it is a commonly used component in existing

Keyword PIR schemes, working as a subroutine within the homomorphic equality operators [15], [16].

B. Private Information Retrieval

Typically, a Private Information Retrieval (PIR) system comprises three primary components: the client, the server, and a public database DB hosted on the server. The PIR server allows the client to retrieve any records from the database without revealing any information about the query. Depending on the nature of the client's query, existing PIR schemes are commonly classified into two main categories: *Index PIR* [5] and *Keyword PIR* [14].

1) **Index PIR**: Index PIR operates under the assumption that the client knows the exact index of the target entry in the database. The client encodes the index as a one-hot vector, encrypts it using Fully Homomorphic Encryption (FHE), and transmits it to the server. The server then performs a privacy-preserving inner product between the encrypted query and the public database, returning an encrypted result that the client decrypts to retrieve the target item. Numerous practical PIR schemes, e.g., FastPIR [7], SealPIR [8], OnionPIR [9], et al., all adopt this paradigm. Due to its reliance on lightweight BFV operations (e.g., PMult and Add), Index PIR offers low computational overhead and is well-suited for scenarios where the query index is known a priori.

2) **Keyword PIR**: However, in real-world scenarios [22], [26], [27], clients usually do not possess the exact index of the desired entry. To address this limitation, Keyword PIR enables queries based on semantic identifiers (e.g., keywords) and is typically built on top of Index PIR schemes. Initially, Keyword PIR is typically achieved by leveraging FHE-based **equality operators** (EQ) [15], [16], [28]. In such schemes, the client encrypts the target keyword k_t and transmits it to the server. The server then performs homomorphic equality checks against each keyword k_i in the dataset. As in Formula 1, this yields encrypted indicators that serve as inputs to a subsequent Index PIR query used to retrieve the corresponding value v_t .

$$EQ(\llbracket k_t \rrbracket, k_i) = \begin{cases} \llbracket 1 \rrbracket, & \text{if } k_t = k_i \\ \llbracket 0 \rrbracket, & \text{otherwise} \end{cases} \quad (1)$$

Note that Equality operators involve costly BFV operations like Mult and Bootstrapping , making Keyword PIR 1~2 orders of magnitude slower than Index PIR [15], [16].

Recently, some schemes have bypassed costly homomorphic equality checks by constructing precomputed keyword-to-index mappings. For example, MulPIR [17] uses cuckoo hashing [29] to map each keyword to multiple candidate positions, which are retrieved and aggregated via Index PIR. SparsePIR [18] encodes sparse keyword-to-value mappings as structured linear combinations, achieving performance comparable to Index PIR while reducing communication cost. Notably, while these approaches significantly improve Keyword PIR performance, they introduce a key limitation: they lack native support for dynamic databases, due to the injective nature of the mapping (details in § III).

C. Computational Storage Device

Computational storage devices (CSDs) have been utilized to alleviate I/O bottlenecks in modern computing systems [30]–[33]. By embedding computational engines directly within storage devices, CSDs significantly mitigate data movement overheads, thereby enhancing overall system efficiency. In this work, we focus on a class of CSDs that closely resemble Samsung's commercially available SmartSSD [21], a representative example of practical in-storage computing.

Compared to conventional NVMe SSDs, SmartSSD provides two major architectural enhancements. First, it incorporates an internal PCIe switch that enables peer-to-peer (P2P) communication between NVMe flash modules and the onboard accelerator, bypassing the host's PCIe bus entirely. Second, SmartSSD is equipped with a lightweight FPGA-based accelerator, which supports in-storage computation via a lightweight FPGA, enabling in-situ processing that offloads work from the host and improves throughput by minimizing data movement. Importantly, both compute and memory bandwidth scale linearly with the number of deployed SmartSSDs, supporting efficient parallelization. Prior studies [34]–[38] have demonstrated SmartSSD's effectiveness for I/O-bound and data-intensive workloads across diverse domains.

D. Threat Model

Consistent with prior PIR systems [7]–[9], [15]–[18], [39], Conflux operates under the semi-honest model, where the server follows the protocol specification but may attempt to infer client queries. Under this model, clients issue private queries to the untrusted server that hosts a publicly available database in plaintext. The adversary is allowed to observe all client queries, monitor the server's responses, and inspect all internal server-side operations, including memory access patterns and computation traces. Conflux ensures that no information about the client's query, e.g., the identifier or semantic meaning of the target item, is leaked.

III. MOTIVATION

In this section, we conducted an in-depth investigation of existing solutions from both the protocol design and system deployment perspectives, and identified two major bottlenecks that hinder their practical applications: (1) the lack of native support for dynamic updates, and (2) I/O overhead emerging as the dominant performance bottleneck in large-scale scenarios.

A. Rigid Update Model

In high-stakes domains such as stock trading and clinical data systems, datasets evolve continuously, making efficient update support critical for correctness and availability. For instance, in clinical systems such as MIMIC-III [19], each patient record can receive hundreds to thousands of updates per day. In stock markets, LOBSTER [20] log millions of order updates per stock per day. However, these state-of-the-art keyword PIR schemes, e.g., MulPIR [17] and SparsePIR [18], are fundamentally tailored for static datasets. They rely on precomputed keyword-to-index mappings that tightly bind the

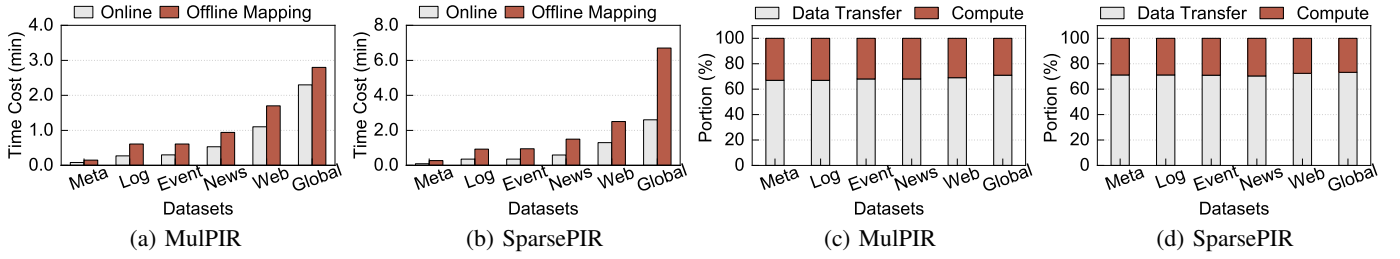


Fig. 1: (a) and (b) shows the online and offline execution time in MulPIR and SparsePIR. (c) and (d) further illustrates the time breakdown analysis for the online execution time. Experimental setup details are provided in § VI-B.

query logic to the underlying data layout. As a result, even a single update or insert renders the entire mapping invalid. More importantly, such keyword-to-index mappings cannot be incrementally updated—there is no mechanism to patch or extend them. Instead, the entire structure must be rebuilt offline, during which no queries can be served.

To quantify this, we evaluated both MulPIR and SparsePIR in a batch private query setting on an NVIDIA A100 GPU platform. We measured two key metrics: (1) the amortized online execution time per private query and (2) the offline execution time to rebuild the keyword-to-index mapping after dataset updates. Experimental setup details refer to § VI-B. Figure 1(a) and (b) present the experimental results. As shown, Mapping-based schemes require reconstructing the keyword-to-index structure upon every dataset update, and the cost can even surpass that of executing a single private retrieval. For instance, under the Meta dataset, MulPIR serves a single private query with an amortized latency of 0.08 minutes, whereas rebuilding the mapping requires up to 0.15 minutes. SparsePIR exhibits a similar trend, with an online time of 0.09 minutes and an offline time reaching 0.27 minutes.

These findings highlight a practical need for more sophisticated PIR designs capable of handling real-world dataset dynamics without frequent or costly remapping (§ IV).

B. I/O-Dominated Performance Bottlenecks

In addition to their static design, these systems perform full-database scans for each query—a pattern that induces heavy I/O overhead and limits overall system performance. To better characterize the sources of the online time cost, we profile MulPIR and SparsePIR servers in detail, and the breakdown results are reported in Figures 1(c) and (d). As shown, on average across all datasets, homomorphic computations (e.g., PMult, Add) contribute only 32% of SparsePIR’s online time cost, with the remaining 68% dominated by I/O operations such as disk access and host-GPU memory transfers. This trend is exacerbated in MulPIR, where I/O overhead constitutes up to 72% of total time cost. These results underscore a fundamental shift: as homomorphic computation becomes lightweight, I/O latency has emerged as the dominant performance bottleneck in PIR systems. Addressing storage subsystem limitations is therefore essential to achieving further performance gains in private retrieval systems.

Algorithm 1: Oblivious Filtering in Conflux

1 **Client:**
2 1: Compute bucket index $b_{id} \leftarrow H(k)$;
3 2: Create queries $\delta[i]$, $i \in [0, m)$ such that:
4 $\delta[i] \leftarrow [1]$ if $i = b_{id}$, else $[0]$;

5 **Server:**
6 1: Encode keys and values per bucket:
7 $K_i, V_i \leftarrow \text{BFV.Encode}(\text{bucket}_i)$, $i \in [0, m)$;
8 2: Select a bucket using homomorphic operations:
9 $K_{sum} \leftarrow \sum_{i=0}^{m-1} \delta[i] \cdot K_i$, $V_{sum} \leftarrow \sum_{i=0}^{m-1} \delta[i] \cdot V_i$;

These results highlight the urgent need for a PIR architecture that holistically integrates computation and data movement. As I/O latency increasingly dominates the runtime, novel architecture designs become critical for unlocking further performance gains (§ V).

IV. CONFLUX PROTOCOL DESIGN

A. Conflux Protocol Workflow

Design Approach. The core design insight of Conflux is to relax the rigid requirement for an exact keyword-to-index mapping. Instead of constructing a precise, update-sensitive index, Conflux employs a fuzzy mapping to filter the search space obliviously. It then performs precise retrieval using equality operators [15], [16] within the reduced search space to retrieve the target item. This two-phase design offers two major advantages. First, the mapping becomes lightweight and resilient to updates, eliminating the need for full reconstruction upon data changes. Second, by carefully bounding the scope of the oblivious filtering, the costly computational overhead of the equality operator can be tightly controlled, ensuring that the overall query complexity remains comparable to Index PIR.

We now describe the step-by-step execution of Conflux, which proceeds in three distinct routines. For clarity, we assume a database containing n key-value pairs (k_i, v_i) , where each key k_i and value are a and b bits.

1) **Setup: Hash-Based Bucketing:** To enable efficient and update-tolerant keyword retrieval, the Conflux server partitions the key-value database into m disjoint buckets using a public hash function $H : \{0, 1\}^a \rightarrow [0, m)$. Each key-value pair is assigned to bucket $H(k_i)$, resulting in roughly $s = n/m$

Algorithm 2: Precise Retrieval in Conflux

1 **Client:**
2 1: Parse k into segments based on modulus T :
3 $k^0, k^1, \dots, k^{w_k-1} \leftarrow k$;
4 2: Create encrypted query vector:
5 $q[i] \leftarrow \llbracket k^i \rrbracket, i \in [0, w_k)$;
6 **Server:**
7 1: Perform homomorphic equality check:
8 $I[i] \leftarrow EQ(q[\cdot], K_{sum}[i][\cdot]), i \in [0, h_k)$;
9 2: Select matching value using indicators:
10 $Ans[j] \leftarrow \sum_{i=0}^{h_k-1} I[i] \cdot V_{sum}[i][j], j \in [0, w_v)$;

entries per bucket. These mappings are stored in plaintext, and both H and m are public parameters shared with Conflux clients. Consequently, it allows clients to locally determine the candidate bucket for any query key via $H(k)$.

2) **Phase 1: Oblivious Filtering via Index PIR:** Algorithm 1 illustrates the process in this phase.

Client side: Given a query key k , the client first computes the corresponding bucket index $b_{id} = H(k)$. It then constructs m encrypted indicator vector $\{\delta\} = \{\delta_0, \delta_1, \dots, \delta_{m-1}\}$ for m buckets, where $\delta_i = \llbracket 1 \rrbracket$ if $i = b_{id}$, and $\delta_i = \llbracket 0 \rrbracket$ otherwise (line 2~5). These ciphertexts serve to privately indicate which bucket contains the desired item. The client sends $\{\delta\}$ to the server as input to the PIR computation.

Server side: Upon receiving the $\{\delta\}$, the server encodes the content of each bucket into plaintext vector matrices to facilitate efficient homomorphic operations. As discussed in § II-A, the BFV plaintext vector supports batching with degree N , where each element must be smaller than the plaintext modulus T . Therefore, for each bucket i , the server first splits each of its s_i keys into w_k segments, and then encodes all segments into a 2D plaintext matrix $K_i \in \mathbb{Z}_T^{h_k^i \times w_k}$, where:

$$h_k^i = \lceil s_i/N \rceil, \quad w_k = \lceil a/\log_2(T-1) \rceil$$

Similarly, values are encoded into a matrix $V_i \in \mathbb{Z}_T^{h_v^i \times w_v}$:

$$h_v^i = \lceil s_i/N \rceil, \quad w_v = \lceil b/\log_2(T-1) \rceil$$

Then, the server performs a homomorphic selection over all buckets. Specifically, it multiplies each encrypted indicator with the corresponding buckets using `PMult` to mask out all non-targeted buckets (line 8). The resulting intermediate matrices are aggregated across all buckets using `Add`, yielding the final accumulated results (line 9): $K_{sum} \in \mathbb{Z}_T^{h_k \times w_k}$, $V_{sum} \in \mathbb{Z}_T^{h_v \times w_v}$, where $h_k = \max(h_k^i)$ and $h_v = \max(h_v^i)$, representing the dimensions of the largest bucket.

3) **Phase 2: Precise Retrieval via Equality Operator:** Algorithm 2 illustrates the process in this phase.

Client side. In this phase, the client encrypts the query key k and sends it to the server. To align with the encoding format used in Phase 1, the client splits the query key k into $w_k = \lceil a/\log_2(T-1) \rceil$ segments under the plaintext modulus

T , and encrypts each segment separately (line 1~5). This yields w_k ciphertexts $\{q[0], q[1], \dots, q[w_k-1]\}$.

Server side. Upon receiving the ciphertexts, the server performs the homomorphic equality check between the query $q[w_k]$ and the output of Phase 1, i.e., K_{sum} . Specifically, the Conflux server calculates the encrypted index I via the equality operator (EQ) as follows:

$$\begin{cases} t[i] &= \sum_{j=0}^{w_k-1} 1 - (q[j] - K_{sum}[i][j])^{T-1} \\ I[i] &= 1 - (w_k - t[i]_{BS})^{T-1}, \quad i \in [0, h_k) \end{cases} \quad (2)$$

After this, the server can use the encrypted indicators $I[h_k]$ to extract the corresponding value from V_{sum} (line 10). The result, denoted as Ans , comprises w_v ciphertexts, each encoding one segment of the matched value. These ciphertexts form the server's response and are returned to the client, who decrypts them to recover the target value v .

Protocol Optimizations and Clarifications. While the above description adopts a two-phase view for clarity, all client-side messages—including the $\{\delta\}$ and $q[w_k]$ —can be constructed and transmitted in a single round. Thus, Conflux maintains a one-round interaction model. Additionally, the stated communication costs (i.e., m ciphertexts in Phase 1 and w_k ciphertexts in Phase 2, with w_v in response) represent a conservative upper bound. In practice, these costs can be amortized via plaintext batching. Leveraging the N -way SIMD capabilities of the BFV scheme, each ciphertext can encode up to N elements, yielding effective communication of $\lceil m/N \rceil$, $\lceil w_k/N \rceil$, and $\lceil w_v/N \rceil$ ciphertexts, respectively, as demonstrated in prior works [8], [9], [15]–[17].

B. Supporting Data Dynamics

Relying on the two-phase retrieval paradigm, Conflux natively supports dynamic datasets, overcoming a key limitation of prior Keyword PIR systems that rely on static keyword-to-index mappings and incur high update costs. Next, we describe how Conflux efficiently handles insertions, deletions, and real-time updates without disrupting query availability.

1) **Update, Insert, and Delete Supports:** At the core of Conflux's dynamic capability is the hash-based oblivious filtering mechanism in Phase 1. Since the public hash function H is non-injective and operates independently per keyword, dataset modifications do not require global reindexing. Specifically, to insert or update a key k_i , the server simply computes $H(k_i)$ and places or replaces the associated entry in the corresponding bucket. Deletion is even simpler, requiring only the removal of the target entry from its bucket.

2) **Bucket Adaptive Tuning:** To maintain retrieval efficiency under significant dataset size fluctuations (e.g., massive insertions or deletions), Conflux supports dynamic adjustment of the number of buckets m , aiming to balance the computational load between the two retrieval stages. Intuitively, If m is too small relative to the dataset size n , buckets become congested, which degrades Phase 1 filtering precision and increases the workload passed to Phase 2. Conversely, if m is too large, buckets become excessively sparse. While this

improves selectivity in Phase 1, it induces substantial padding during BFV packing, thereby reducing packing efficiency and inflating the computational cost of Phase 1. The packing efficiency can be computed by $n/(N * \sum_{i=0}^{m-1} \lceil s_i/(n/m) \rceil)$, where N is the BFV degree and s_i denotes the number of entries in bucket i . To strike a controllable balance, we set the average bucket capacity to $4N$ to control the number of buckets, $m = n/4N$. This setting maintains aggressive filtering while keeping the high packing efficiency ($\approx 90\%$) to bounding Phase 1 cost. We will evaluate the effectiveness of this configuration in § VI-D.

Specifically, Conflux maintains a dedicated background thread that continuously monitors system load by computing the average bucket occupancy $o = n'/m$, where n' denotes the current number of key-value pairs. When o deviates significantly from the target (i.e., exceeds $4N$), the system recalculates the optimal number of buckets as $m' = n'/4N$ and triggers a full rehash of the dataset. Crucially, Conflux can perform this adjustment asynchronously and non-blockingly to ensure uninterrupted service. During rehashing, clients can continue issuing private queries using the previous configuration m ; although the server may process these queries with slightly reduced efficiency, the results remain correct and consistent with those under the new configuration m' . Once rehashing completes, the server broadcasts the updated configuration m' , and the old version m is safely retired.

C. Security Analysis

Conflux provides strong query privacy guarantees under the threat model described in § II-D by leveraging the semantic security (IND-CPA [40]) of the BFV scheme. Since each query in Conflux is executed over a fixed and content-independent access structure determined solely by public system parameters, the system ensures that an adversary—who may observe the complete server-side execution trace—cannot effectively distinguish between any two distinct client queries.

In more detail, for any two query keys k_1 and k_2 , the following artifacts remain indistinguishable to any polynomial-time adversary: (1) The encrypted queries sent over the network; (2) The resulting memory access patterns during filtering and equality checking; (3) The intermediate computation traces within both Phase 1 and Phase 2; (4) The final response ciphertext returned to the client. These properties collectively ensure that no information about the client’s retrieval intent is leaked through communication volume, control flow, or data access behavior. Instead, the only information available to the adversary is publicly known system metadata: the hash H , the bucket count m , and the plaintext dataset.

V. CONFLUX ACCELERATOR DESIGN

Although the Conflux protocol achieves near-Index PIR performance with full support for dynamic data, it still suffers from a pronounced I/O bottleneck at the system level. To mitigate this issue, we design an I/O-efficient accelerator that tightly integrates computational storage devices (CSDs) into a heterogeneous architecture. We deploy Conflux on this

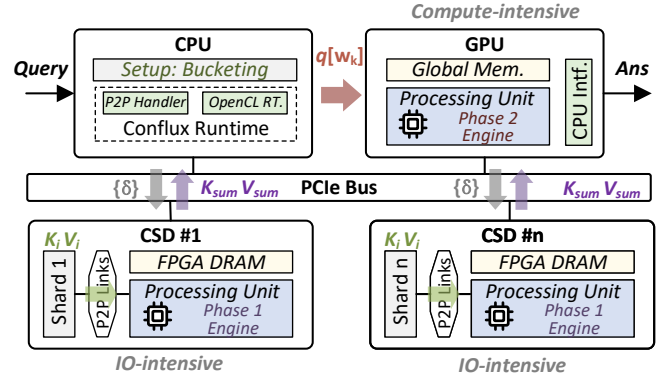


Fig. 2: The overview of the Conflux architecture.

platform to reduce data transfer overhead and enable scalable performance at scale.

A. Overall Architecture

Conflux’s two-phase retrieval exhibits distinct and asymmetric performance characteristics. Phase 1 performs an Index PIR-style oblivious scan over the entire dataset using a public, non-secret hash function to identify candidate entries. Since this phase applies only lightweight BFV operations (i.e., PMult, Add), it is I/O-bound, dominated by the cost of data traversal rather than computation. In contrast, Phase 2 operates in the Keyword PIR regime, performing equality checks over the filtered results returned from Phase 1. This involves computationally expensive homomorphic operations (i.e., Mult and Bootstrapping), making it compute-intensive. To efficiently serve these heterogeneous demands, we design a specialized accelerator integrating a CPU, CSDs, and a GPU, as shown in Figure 2. Each component is assigned workloads that align with its strengths, enabling efficient and scalable execution. Specifically:

- The CPU orchestrates the overall query workflow. It handles client interaction, hash-based bucketing, metadata management, and runtime updates. It also coordinates task dispatch to the CSDs and GPU via OpenCL [41].
- Phase 1 is offloaded to the CSDs, which execute oblivious filtering directly within the storage layer. By leveraging embedded FPGAs and P2P access to flash, all CSDs locally perform lightweight homomorphic operations and output K_{sum} and V_{sum} , which are then forwarded to the GPU for further processing. Note that, as discussed in § II-C, the CSD array provides higher aggregate bandwidth and near-linear scalability in compute throughput, enabled by intra-CSD P2P links. Building upon these properties, offloading Phase 1 to the CSDs allows Conflux to exploit higher data-movement efficiency and data parallelism, thereby accelerating the filtering phase.
- Phase 2 is offloaded to the GPU, which efficiently handles the EQ between K_{sum} and encrypted queries via its massive parallelism. The matched key then guides the final filtering of V_{sum} to produce the response.

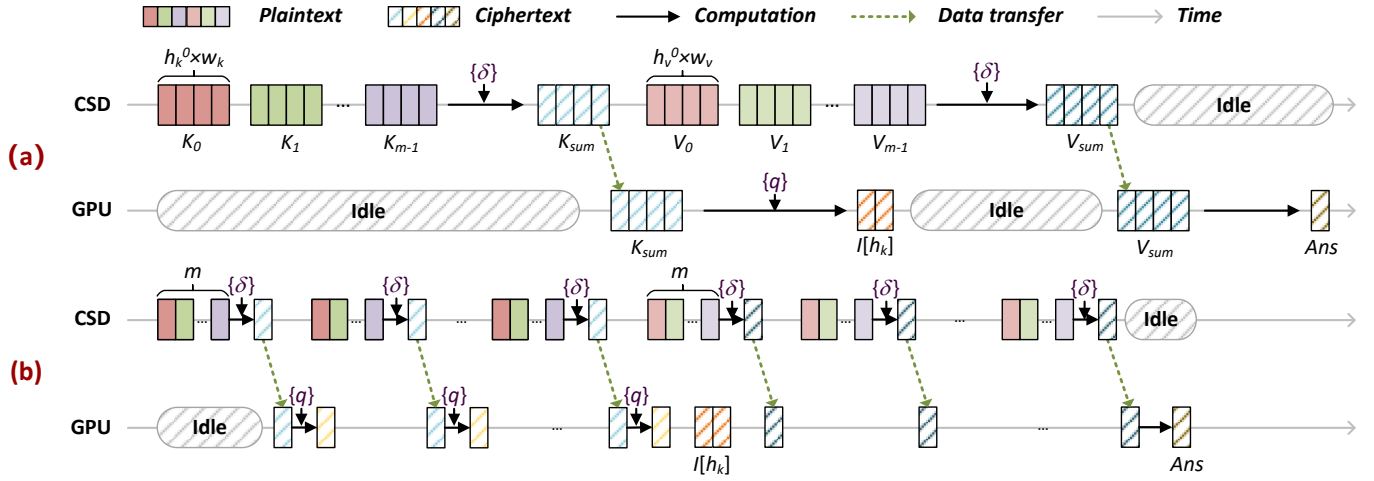


Fig. 3: Pipelined execution inside the Conflux architecture: (a) coarse-grained, (b) fine-grained.

This division of labor minimizes I/O overhead, maximizes heterogeneous hardware utilization, and ensures high-throughput private retrieval even under large-scale workloads. In the next sections, we introduce two further optimizations that further enhance system performance: pipelined execution and balanced workload distribution.

B. Inter-device Pipelined Execution

In the baseline Conflux execution model, Phase 1 and Phase 2 are executed sequentially: the CSD first performs filtering via Phase 1 to generate K_{sum} and V_{sum} , which are then transferred to the GPU for precise retrieval via Phase 2. While intuitive, the serialized design underutilizes the system's heterogeneous resources and limits overall throughput. To maximize resource utilization, we design a dependency-aware pipelining strategy that coordinates CSD and GPU execution through fine-grained analysis of parallelism opportunities.

Coarse-grained Pipeline. The workflow in Phase 2 exhibits only a partial dependency on Phase 1 outputs: specifically, the GPU requires K_{sum} for executing equality operator but does not rely on V_{sum} . This partial dependency enables early GPU activation: once the CSDs complete the key aggregation, K_{sum} can be streamed directly to the GPU to begin processing the equality check with encrypted query $q[w_k]$. Concurrently, CSDs continue computing V_{sum} . As shown in Figure 3(a), this coarse-grained pipelining effectively overlaps the two phases and reduces GPU idle time.

Fine-grained Pipeline. Building upon the coarse-grained pipeline strategy, we can further decompose the pipeline into finer-grained units. The key insight is that the equality operators on $K_{sum}[:, w_k]$ are independent across columns. Therefore, instead of waiting for full matrix aggregation, each CSD can compute individual columns of K_i across buckets, form partial K_{sum} columns on the fly, and immediately stream them to the GPU for partial equality checks. This streaming-style execution enables concurrent processing across devices over a 2D tile space of size $h_k \times w_k$, allowing pipelined

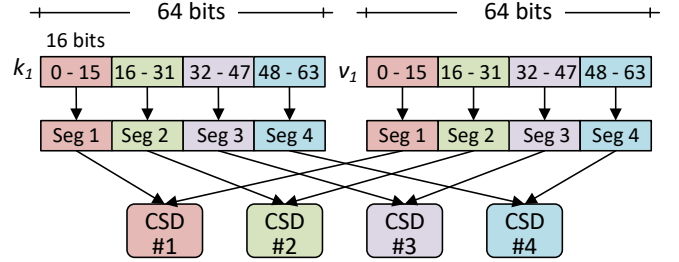
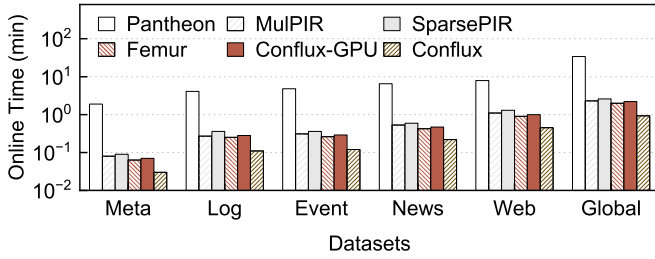


Fig. 4: Flattening the database across CSDs via bit slicing. Here, $T = 2^{16} + 1$, and the key and value are 64 bits.

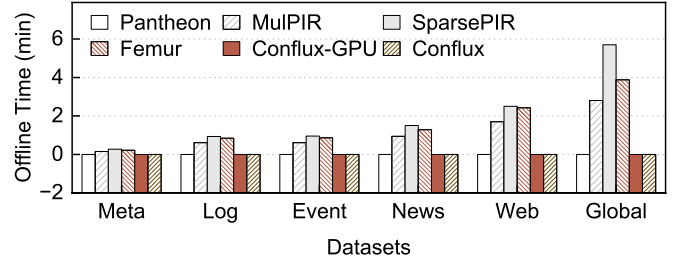
execution of independent equality checks. A similar mechanism applies to V_{sum} : once the GPU determines the encrypted index I using K_{sum} and $q[w_k]$, corresponding columns of V_i are merged and streamed to retrieve values from V_{sum} . Final aggregation into Ans occurs entirely on the GPU. Figure 3(b) illustrates this tightly interleaved execution model, which overlaps computation with data transfer to effectively hide Phase 2 latency within the execution window of Phase 1, thereby improving overall system efficiency.

C. Workload Distribution to Multiple CSDs

To effectively exploit the parallelism of multiple CSDs, Conflux requires a carefully designed strategy for distributing both data and computation. However, conventional partitioning methods, such as round-robin [42] or bucket-based schemes, incur substantial inter-device communication overhead in Conflux, which significantly degrades overall system performance. This inefficiency stems from a misalignment between the physical data layout and computational pattern of Conflux, particularly in Phase 1, where bucket encoding and cross-bucket aggregation can trigger frequent and costly data transfers across CSDs via the host PCIe bus. To mitigate this problem, we propose a data flattening strategy that: (1) evenly balances computational workloads across all CSDs, and (2) minimizes inter-device data transfers.



(a) Online Execution Time



(b) Offline Execution Time

Fig. 5: Comparison of per-query online and offline execution times for different schemes across multiple datasets.

Our key idea is to flatten and distribute the entire dataset across CSDs in such a way that the data layout aligns with the computational structure. Concretely, following the plaintext modulus T defined in the BFV scheme and the encoding pattern, each key and value in the database is sliced along the bit dimension into w_k and w_v segments, respectively. These segments are then uniformly distributed across all available CSDs. Figure 4 illustrates this process. This bit-level slicing yields two primary benefits. (1) Uniform flattening ensures that each CSD holds a balanced amount of data, thereby ensuring load balancing across CSDs and preventing skew; (2) By decoupling the physical layout from the hash bucket boundaries, it eliminates the need for inter-CSD data exchange during both bucket encoding and cross-bucket aggregation. As a result, all Phase 1 operations, including bucket encoding and cross-bucket aggregation, can be executed using local data alone, achieving full computational locality. This approach eliminates PCIe communication overhead and enables highly parallel, load-balanced execution. Consequently, Conflux scales efficiently across multiple CSDs, fully leveraging their aggregate bandwidth and compute capabilities.

D. System Implementation

We implement our Conflux system in a heterogeneous architecture composed of a host CPU, an NVIDIA A100 GPU, and the commercial off-the-shelf SmartSSD [21]. The CPU manages system orchestration using a two-threaded controller written in C++17. One thread handles client request reception and triggers query logic, while the other performs hash computations and manages the database state. Our profiling indicates that the CPU maintains an average utilization of less than 20% during normal operation. The SmartSSD hosts the entire database, managed by RocksDB [43], and includes an FPGA-based search engine synthesized with Xilinx Vitis 2021.2 (frequency: 300MHz). The search engine follows the circuit design outlined in prior works [44]–[47], implementing dedicated pipelines for the PMult and Add modules. Table II details the FPGA resource utilization for each SmartSSD. For the GPU side, we integrate the HEonGPU library [48] to implement Phase 2 of the query processing logic. The host system coordinates the various components through Xilinx Runtime (XRT) for FPGA management and CUDA 12.3 for GPU operations.

TABLE II: Resource Utilization

	LUT	FF	BRAM	URAM	DSP
Count	357,869	390,185	840	80	625
Percent	34.46%	11.32%	65.63%	62.50%	31.93%

TABLE III: Datasets

	Meta	Log	Event	News	Web	Global
Key	16 B	32 B	32 B	64 B	96 B	256 B
Value	64 B	256 B	288 B	512 B	1 KB	2 KB

VI. EVALUATION

A. Experimental Setup

Our heterogeneous evaluation platform includes an Intel Xeon(R) Gold 6342 CPU, an NVIDIA A100 GPU, and 8 Samsung SmartSSDs. Each SmartSSD integrates a 4 TB NVMe drive directly connected to a Kintex UltraScale+ KU15P FPGA via a PCIe Gen3.0 $\times 4$ interface.

Datasets. To capture a broad range of real-world workloads, we construct six representative dataset configurations inspired by production-scale Facebook traces [49], [50]. Each dataset comprises key-value pairs with varying key and value sizes, spanning from small-scale to enterprise-scale regimes. Table III details the dataset parameters. All datasets used in our experiments contain 2^{25} entries.

Metrics. Our evaluation includes three key metrics: server-side online execution time (**OT**), offline reconstruction time (**OFT**), and client-server communication volume (**COM**). OT measures the time (in minutes) for the server to process a single private query. OFT denotes the offline reconstruction time (in minutes) due to any modifications of datasets. COMM refers to the total amount of data exchanged between the server and the client for one query, measured in megabytes (MB).

Conflux and Baselines. We compare Conflux against four representative keyword PIR systems. (1) **Pantheon** [15] is the fastest known scheme that supports dynamic datasets by relying on equality operators. In contrast, (2) **MulPIR** [17], (3) **SparsePIR** [18], and (4) **Femur** [51] represent state-of-the-art keyword PIR schemes designed for static datasets. Specifically, MulPIR and SparsePIR employ hint-free keyword-to-index mappings, while Femur leverages a learned index with a

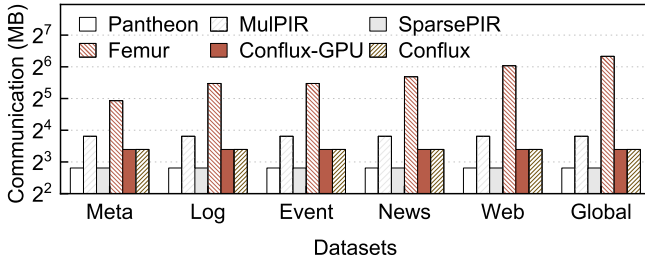


Fig. 6: Average communication overhead per query.

compact hint to accelerate lookup. These schemes both offer low online latency but require rebuilding the keyword-to-index mapping upon dataset updates. All baselines are implemented atop HEONGPU [48] and run on an NVIDIA A100 GPU with 8 NVMe SSDs. For Conflux, we evaluate two variants: (5) **Conflux-GPU**, which runs entirely on the A100, and (6) **Conflux**, deployed on our heterogeneous accelerator platform. In all experiments, clients issue queries under the most recent configuration determined by the bucket adaptive tuning mechanism. To isolate system-level differences and ensure a fair comparison, all schemes are evaluated under the same full security level, and BFV parameters are unified across all schemes as $(N, T, \log_2 Q) = (2^{15}, 65,537, 881)$. Furthermore, all implementations incorporate a batch query mechanism [8], allowing the server to process 16 queries per batch, simulating realistic multi-query workloads.

B. Overall Performance

We evaluate Conflux against several baselines across six real-world datasets. To simulate realistic workloads, updates (including both keys and values) are issued periodically after every 16 queries. Figure 5 and 6 report the results.

For retrieval latency, Conflux sharply reduces online time versus Pantheon by restricting expensive equality checks to a small candidate set. On the Meta dataset, Pantheon requires 1.9 minutes per query, while Conflux-GPU completes in 0.07 minutes (27 $\times$); with the full Conflux design, latency drops to 0.03 minutes (63 $\times$). Both Pantheon and Conflux support dynamic updates, so neither incurs offline rebuild cost. In contrast, static-mapping systems (i.e., MulPIR, SparsePIR, Femur) lack dynamic updates and must rebuild their keyword-index mappings after dataset changes, incurring substantial offline time. For instance, on the Event dataset, their per-query latency and separate index-build time are 0.31 / 0.61 minutes, 0.36 / 0.95 minutes, and 0.26 / 0.86 minutes, respectively. By relaxing the mapping constraint, Conflux achieves both high performance and dynamic update capability. Specifically, on the Event dataset, Conflux-GPU completes a query in 0.29 minutes, faster than MulPIR and SparsePIR, though slightly slower than Femur. With the full accelerator stack, Conflux can further reduce runtime to 0.12 minutes.

For communication, all systems use identical BFV parameters, so the ciphertext size is fixed (i.e., 3.5 MB). The main differences arise solely from the number of ciphertexts exchanged. Across datasets, Pantheon and SparsePIR each send

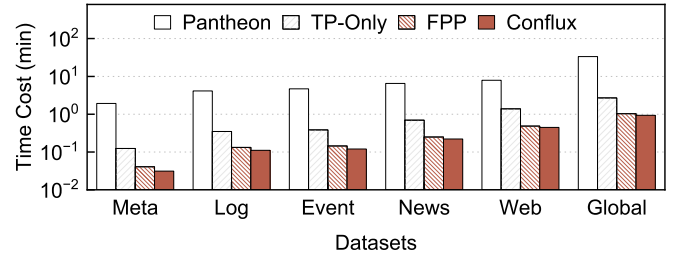


Fig. 7: Performance improvement due to different technologies in the Conflux system.

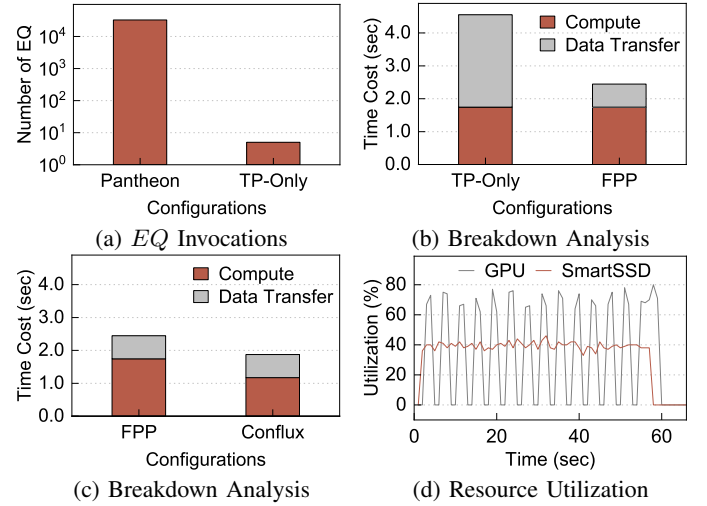


Fig. 8: In the Meta, (a) reports the number of EQ invocations under Pantheon and TP-Only; (b) and (c) present the time breakdown across TP-Only, FPP, and Conflux, highlighting differences in execution stages; (d) shows the hardware utilization of the GPU and SmartSSD during Conflux execution.

and receive one ciphertext, incurring 7 MB per query. Conflux performs two retrieval mechanism, adding one extra ciphertext for a total of 10.5 MB. MulPIR, due to its cuckoo-hash structure, sends three ciphertexts and receives one, totaling 14 MB. Femur exhibits larger variability, i.e., 30 MB~80 MB, depending on key-value size and its transmitted learned-index hint. Although Conflux's communication cost is slightly higher than the minimum, the additional few megabytes are negligible compared with overall retrieval latency.

C. Effectiveness of Individual Technologies

In this subsection, we assess the individual contributions of Conflux's key technologies. Our analysis covers both protocol-level innovations, i.e., the two-phase query design, as well as system-level enhancements, including the heterogeneous architecture and pipelined execution. To isolate the impact of each component, we perform an incremental ablation study using Pantheon [15] as the baseline. We evaluate the following configurations:

- **Two-Phase Retrieval Only (TP-Only):** Deploys the two-phase Conflux protocol on the GPU.

- **Full Protocol without Pipelining (FPP):** Deploys the Conflux protocol on the heterogeneous system but without pipelined execution.
- **Full Conflux System (Conflux):** Incorporates all proposed techniques, including pipelined scheduling, to represent the complete design.

We report the online execution time for each configuration across various datasets, with results summarized in Figure 7. Note that all configurations maintain the same communication volume for the same dataset.

1) *Protocol-level Effectiveness:* As shown in Figure 7, the TP-Only configuration yields significant performance gains over Pantheon by narrowing the effective search space, resulting in a $7.8\times$ to $25.2\times$ speedup. This improvement primarily stems from TP-Only’s ability to restrict equality checks to a single candidate bucket through data masking. For example, as shown in Figure 8(a), on the Meta dataset, Pantheon performs equality checks over all keys, totaling 1,024 EQ , whereas TP-Only executes EQ operations only within a single candidate bucket, requiring just 5. Note that the computational cost of EQ operations is identical whether performed between two ciphertexts or between a ciphertext and a plaintext.

2) *Hardware-level Effectiveness:* For the TP-only configuration, Phase 1 becomes the dominant performance bottleneck, as the entire dataset must be transferred from storage to the GPU for processing, incurring substantial I/O overhead. To this end, the FPP design addresses this limitation by offloading Phase 1 computation to SmartSSDs equipped with lightweight FPGAs. These devices enable early-stage filtering directly within the storage layer, thereby fully exploiting the CSD array’s large aggregate bandwidth and reducing expensive CSDs-GPU data transfer. As shown in Figure 8(b), this architectural shift significantly shortens data transfer time on the Meta dataset. Meanwhile, although a CSD provides limited compute capability, the aggregated parallelism of 8 CSDs delivers performance comparable to a single A100 GPU. As a result, FPP achieves a speedup of $1.86\times$ to $2.09\times$ across different datasets, demonstrating the effectiveness of in-storage computation in mitigating I/O latency.

Building on this, we further evaluate the impact of pipelined execution by comparing FPP with the full Conflux system. From Figure 8(c) to (d), Conflux enables concurrent execution between SmartSSDs and the GPU via a fine-grained execution pipeline. This coordination ensures that both compute units are fully utilized throughout query processing. As a result, Conflux achieves an additional $1.10\times$ to $1.31\times$ performance improvement compared to FPP, confirming the effectiveness of pipelined execution in maximizing system throughput.

D. Adaptive Bucket Tuning Analysis

In this subsection, we evaluate the benefits of the bucket adaptive tuning strategy during dynamic dataset expansion. We begin by initializing each of the six datasets with 2^{18} key-value pairs and configuring the number of buckets to $m = 8$. To simulate real-world usage patterns, we perform continuous SET operations, incrementally expanding the dataset size

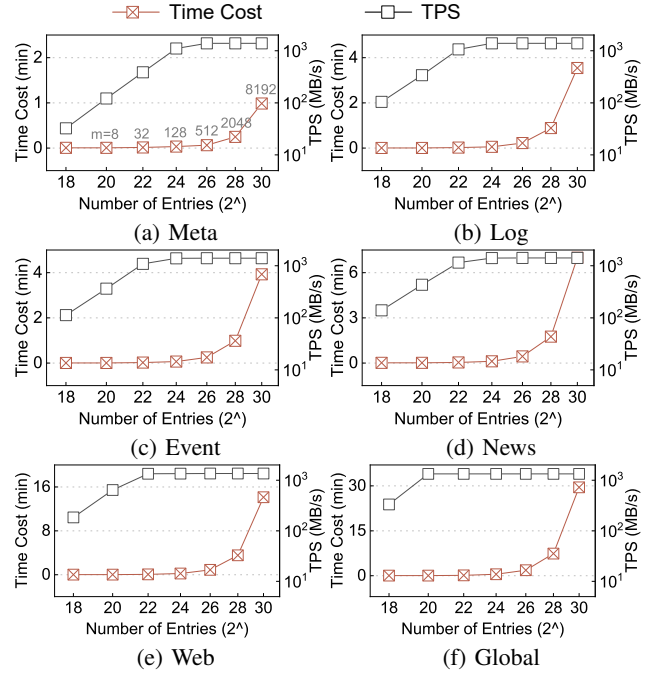


Fig. 9: Benefits from the bucket adaptive tuning.

up to 2^{30} . At each expansion stage, we measure two key performance metrics: (i) query latency, defined as the time cost per private query, and (ii) system throughput, computed as the total data volume processed divided by the time cost. The results are presented in Figure 9.

As expected, query latency increases with dataset size, reflecting the inherent overhead of PIR schemes that require access to the entire dataset. However, a more nuanced trend emerges in system throughput: it increases steadily and eventually plateaus. This improvement is primarily attributed to two synergistic effects of the adaptive tuning strategy: (1) Progressive bucket refinement. As more data is inserted, Conflux dynamically increases the bucket count m (from 8 up to 8192), which enhances the filtering power of Phase 1. Since Phase 2 is relatively expensive in the early stages, reducing the number of candidates forwarded to it significantly boosts throughput. (2) Phase dominance transition. Once m exceeds a certain threshold ($m \approx 128$), the cost contribution of Phase 2 diminishes, and Phase 1 becomes the dominant factor in query latency. At this point, Conflux reaches its optimal operating regime: throughput saturates, and query latency grows roughly linearly with the dataset size.

We further investigate the impact of bucket capacity (i.e., $2N$, $4N$, $8N$) on the performance of Conflux. As shown in Figure 10(a), the results reveal that bucket capacity exerts opposing effects on Phases 1 and 2. Specifically, smaller bucket capacity, due to the higher total count, enhances the filtering capacity of Phase 1, thereby significantly reducing the candidate set size passed to Phase 2. However, because of hash dispersion, smaller bucket capacity are more susceptible to holes during Phase 1 encoding, e.g., the encoding

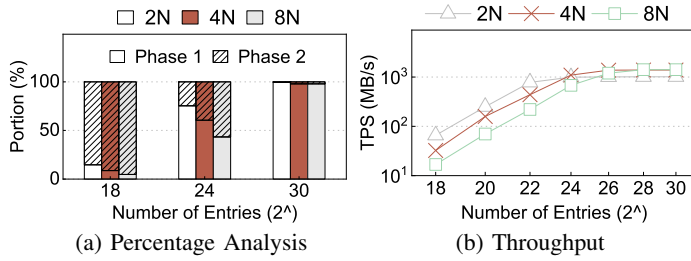


Fig. 10: Impact of bucket capacity on Conflux performance.

efficiency of $2N$ is only 67%. This, in turn, increases the computational overhead in Phase 1. Conversely, larger bucket capacity improves the encoding density, e.g., 89% for $4N$, but at the cost of weaker filtering in Phase 1, which shifts more workload to downstream processing in Phase 2. This trade-off between filtering effectiveness and encoding efficiency directly influences overall system throughput. As shown in Figure 10(b), the $2N$ suffers from lower peak TPS due to inefficient encoding in Phase 1, while the $8N$ exhibits significantly slower convergence due to the limited filtering effectiveness. In contrast, under our experimental settings, medium-sized buckets (i.e., $4N$) generally provide a more favorable cost balance between Phase 1 and Phase 2.

E. Scalability Analysis

In this subsection, we evaluate the effectiveness of Conflux’s workload distribution strategy when scaling across multiple SmartSSDs. Experiments are conducted on six datasets with configurations ranging from 1 to 8 SmartSSDs. Since the performance bottleneck of query processing primarily lies in Phase 1, which is executed on SmartSSDs, increasing the number of GPUs has a limited impact on overall performance. Therefore, we set the GPU count to 1 for all experiments. For comparison, we adopt the traditional bucket-based database partitioning strategy [52] as the baseline. Figure 11 shows the normalized online query execution time across different SmartSSD counts, with all results normalized to the performance achieved using a single SmartSSD.

Under the bucket-based partitioning approach, although data is evenly distributed across SmartSSDs, Phase 1 triggers substantial inter-device communication due to cross-bucket aggregation. This data movement significantly increases inter-device I/O overhead, ultimately reducing the performance gains of aggregated bandwidth and computation provided by more SmartSSDs. As shown in Figure 11, the overall performance improvement under this strategy is limited and falls short of ideal linear scalability.

In contrast, Conflux adopts a flattened execution model in which all I/O access and computation remain fully localized within each SmartSSD, thereby eliminating inter-device communication. This design ensures efficient resource utilization, reduces latency, and improves throughput. As evidenced in Figure 11, Conflux achieves near-linear performance scaling as the number of SmartSSDs increases, demonstrating the effectiveness of our design in addressing scalability demands.

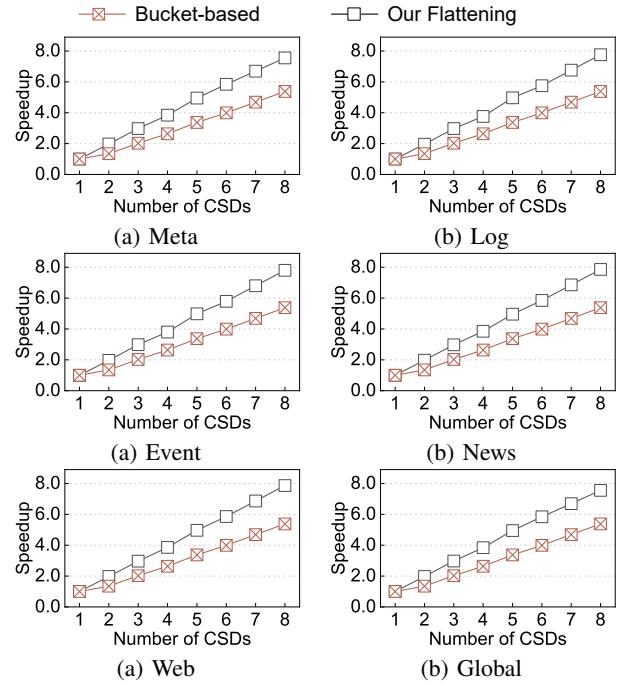


Fig. 11: Scalability with CSDs of the Conflux system.

F. Sensibility Analysis

In this subsection, we further explore the sensitivity and stability of Conflux from two perspectives. First, we examine how the ratio of queries to updates affects overall system performance. Second, we evaluate the impact of bucket adaptive tuning on foreground retrieval performance.

For the first analysis, following the Facebook trace [22], we evaluate three commonly used query:update ratios: 1:1, 5:1, and 10:1. To simulate a near-static workload and reflect the worst-case scenario, we also include a 50:1 ratio. We compare Conflux-GPU and Conflux MulPIR and Femur, omitting other baselines as MulPIR and Femur consistently demonstrate the best performance. All experiments are conducted on the Meta dataset. Figure 12(a) presents the results, with the performance of Femur, Conflux-GPU, and Conflux normalized to that of MulPIR. Overall, as the proportion of queries increases (i.e., fewer updates), the average retrieval latency gap among the four systems gradually narrows. This aligns with expectations: in static systems, the amortized reconstruction cost diminishes as updates become infrequent, causing their average latency to approach online latency. In contrast, the performance of Conflux-GPU and Conflux remains stable. When the ratio reaches 50:1, Femur slightly outperforms Conflux-GPU in terms of average latency. However, since Conflux achieves substantially lower online retrieval latency through CSDs, it still delivers the best overall performance.

For the second analysis, we evaluate the effect of bucket resizing on foreground retrieval throughput. To remove performance variations caused by different dataset scales, we report normalized TPS results, as shown in Figure 12(b). The results show that the impact of resizing on online retrieval

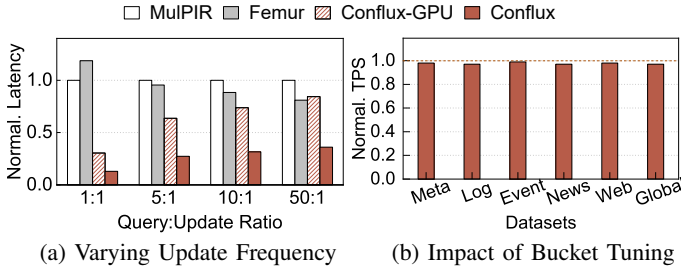


Fig. 12: (a) and (b) illustrate the effects of update frequency and bucket tuning on foreground performance.

performance is negligible. This finding primarily stems from two architectural design features: (1) **Task decoupling**: bucket tuning performs a lightweight hash mapping over all keys, which mainly consumes CPU resources, while foreground retrieval relies on SmartSSD and GPU acceleration. (2) **Path isolation**: the I/O traffic of bucket tuning is carried over the system PCIe bus, which operates independently of the CSD-internal P2P channels used during Phase 1 retrieval, effectively preventing traffic contention. Therefore, Conflux can perform background adaptive tuning while maintaining stable foreground TPS performance, achieving efficient dynamic scalability.

G. Cost and Energy Efficiency

In this section, we select Femur, which achieves the best average query latency among all baselines, as the reference system to evaluate the cost-effectiveness and energy efficiency of Conflux. To ensure a fair and representative comparison under typical workloads, the query:update ratio is set to 10:1. Unlike Femur’s GPU-only architecture, Conflux introduces additional hardware components, i.e., SmartSSDs, to alleviate I/O bottlenecks and accelerate query execution. While these additions improve performance, they also incur extra cost, making it essential to assess their practical efficiency from a system-level perspective. We report two key metrics: cost efficiency (*performance-per-dollar*) and energy efficiency (*queries-per-Joule*). The system cost accounts for all hardware components, including the server (\$45,000), storage (\$400 per SSD), SmartSSDs (\$2,000 each), and GPUs (\$8,000). Power consumption is measured using a calibrated power analyzer.

As shown in Figure 13(a), Conflux consistently outperforms Femur and Conflux-GPU in cost efficiency, achieving $2.04\times\sim 2.48\times$ and $1.74\times\sim 2.07\times$ improvements across six representative datasets. Although Conflux incurs higher hardware investment due to its heterogeneous deployment, the inclusion of SmartSSDs effectively mitigates I/O overhead, thereby increasing end-to-end throughput and improving cost amortization over time. A similar trend holds for energy efficiency. As shown in Figure 13(b), Conflux achieves $2.81\times\sim 3.41\times$ and $2.13\times\sim 2.54\times$ improvements in energy efficiency compared to Femur and Conflux-GPU. This advantage is primarily attributed to reduced query latency and the inherently low power footprint of SmartSSDs. Notably, the

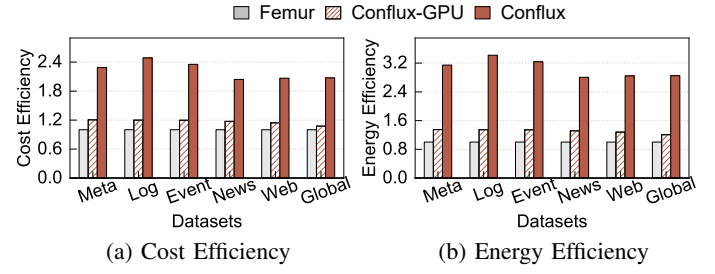


Fig. 13: Cost/energy efficiency among different schemes.

embedded FPGA within each SmartSSD contributes minimal energy overhead. Overall, Conflux offers a compelling trade-off between performance, cost, and power, making it a practical solution for privacy-preserving query processing.

VII. RELATED WORK

Privacy-Enhanced Database Systems. A large body of work has focused on secure query processing over private data, leveraging paradigms such as order-preserving encryption (OPE) [53], fully homomorphic encryption (FHE) [23]–[25], searchable encryption (SE) [54], [55], trusted execution environments (TEE) [56]–[58], and private set intersection (PSI) [59], [60]. These works primarily address data confidentiality in outsourced settings, where users encrypt and upload their private datasets to untrusted servers/partners. In contrast, Conflux targets a complementary threat model where the database is public, and the objective is to protect the client’s query intent from the server, e.g., Apple’s Wally [61] and Web Search [3], rather than the data itself.

Private Information Retrieval on Public Data. Private Information Retrieval (PIR) addresses the challenge of retrieving records from a public database without revealing the query intent to the server. Several PIR systems, such as Pantheon [15], CwPIR [16], MulPIR [17], and SparsePIR [18], Femur [51], have explored keyword PIR via homomorphic equality operators or index-based mappings. While effective for static workloads, these approaches incur significant offline reconstruction overhead when datasets are updated. More recent works employ Regev encryption schemes [62] to further improve PIR performance. Systems such as ChameleonPIR [63], FrodoPIR [64], and SimplePIR [11] exploit public-key reusability to shift computation offline and reduce query latency. However, due to the inherent limitations of LWE-based encryption, these designs involve substantial data transfer and interaction, making them impractical for dynamic datasets. In contrast, Conflux adopts a two-phase retrieval paradigm specifically designed for dynamic environments, achieving the performance of rigid keyword-to-index mappings while maintaining compatibility with dynamic cloud environments.

Hardware Acceleration. GPU-based acceleration has become a common approach for improving the efficiency of PIR systems, given the substantial computational overhead associated with homomorphic encryption. Prior works have demon-

strated significant performance gains using various forms of GPU parallelism [15], [17], [65], [66]. However, these efforts primarily focus on computational acceleration and largely overlook the performance bottlenecks associated with data movement and I/O throughput. In contrast, Conflux adopts a hardware–software co-design that integrates GPU acceleration with SmartSSD-based I/O offloading. To the best of our knowledge, Conflux is the first PIR system to combine GPUs and computational storage devices in a pipelined architecture, enabling holistic, end-to-end performance improvements that go beyond compute-only optimizations.

VIII. CONCLUSION

In this paper, we present Conflux, a high-performance, update-resilient Keyword PIR system tailored for dynamic datasets. Conflux introduces a two-phase retrieval protocol that eliminates the need for static index mappings, enabling seamless support for data updates. To further optimize performance, Conflux incorporates a protocol-aware hardware architecture that leverages both SmartSSDs and a GPU, effectively mitigating I/O bottlenecks and maximizing parallelism. Our evaluation shows that Conflux reduces query latency by up to $2.64\times$ compared to state-of-the-art systems, while maintaining robust performance under dynamic and write-intensive workloads.

ACKNOWLEDGMENT

This work was supported in part by the National Natural Science Foundation of China under Grant 62432005, 62372272, 62272271, and 624B2091; in part by the Shandong Provincial Natural Science Foundation under Grants ZR2024MF099 and ZR2023LZH014; in part by the Department of Science and Technology of Shandong Province under Grant SYS202201; in part by the Research Project of Quancheng Laboratory (Grant No. QCL20250301, QCL20250101); in part by the Taishan Scholars Program under Grant TSQN202211281; in part by the Open Research Program of the State Key Laboratory of Processors under Grant CLQ202405; and in part by the Qingmiao Program of Shandong university under Grant SDUQM2510.

REFERENCES

- [1] S. Angel and S. Setty, “Unobservable communication over fully untrusted infrastructure,” in *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016, pp. 551–569.
- [2] P. Mittal, F. Olumofin, C. Troncoso, N. Borisov, and I. Goldberg, “Pir-tor: Scalable anonymous communication using private information retrieval,” in *Proceedings of USENIX security symposium (Security)*, 2011.
- [3] A. Henzinger, E. Dauterman, H. Corrigan-Gibbs, and N. Zeldovich, “Private web search with tiptoe,” in *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, 2023, pp. 396–416.
- [4] K. Zhong, Y. Ma, and S. Angel, “Ibex: Privacy-preserving ad conversion tracking and bidding,” in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022, pp. 3223–3237.
- [5] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan, “Private information retrieval,” *Journal of the ACM (JACM)*, vol. 45, no. 6, pp. 965–981, 1998.
- [6] H. Sun and S. A. Jafar, “The capacity of private information retrieval,” *IEEE Transactions on Information Theory*, vol. 63, no. 7, pp. 4075–4088, 2017.
- [7] I. Ahmad, Y. Yang, D. Agrawal, A. El Abbadi, and T. Gupta, “Addra: Metadata-private voice communication over fully untrusted infrastructure,” in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.
- [8] S. Angel, H. Chen, K. Laine, and S. Setty, “Pir with compressed queries and amortized query processing,” in *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 962–979.
- [9] M. H. Mughees, H. Chen, and L. Ren, “Onionpir: Response efficient single-server pir,” in *Proceedings of the ACM SIGSAC Conference on Computer and Communications security (CCS)*, 2021, pp. 2292–2306.
- [10] B. Li, D. Micciancio, M. Raykova, and M. Schultz-Wu, “Hintless single-server private information retrieval,” in *Annual International Cryptology Conference (CRYPTO)*, 2024, pp. 183–217.
- [11] A. Henzinger, M. M. Hong, H. Corrigan-Gibbs, S. Meiklejohn, and V. Vaikuntanathan, “One server for the price of two: Simple and fast Single-Server private information retrieval,” in *32nd USENIX Security Symposium (USENIX Security)*, 2023, pp. 3889–3905.
- [12] C. A. Melchor, J. Barrier, L. Fousse, and M.-O. Killijian, “Xpir: Private information retrieval for everyone,” *Proceedings on Privacy Enhancing Technologies (PoPETs)*, pp. 155–174, 2016.
- [13] Z. Chen, H. You, Q. Wei, H. Lu, L. Ju, and Z. Shen, “Smartpir: A private information retrieval system using computational storage devices,” in *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2025, p. 1749–1762.
- [14] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan, “Private information retrieval,” *Journal of the ACM (JACM)*, vol. 45, no. 6, pp. 965–981, 1998.
- [15] I. Ahmad, D. Agrawal, A. E. Abbadi, and T. Gupta, “Pantheon: Private retrieval from public key-value store,” *Proceedings of the VLDB Endowment (VLDB)*, vol. 16, no. 4, pp. 643–656, 2022.
- [16] R. A. Mahdavi and F. Kerschbaum, “Constant-weight pir: Single-round keyword pir via constant-weight equality operators,” in *Proceedings of the USENIX Security Symposium (Security)*, 2022, pp. 1723–1740.
- [17] A. Ali, T. Lepoint, S. Patel, M. Raykova, P. Schoppmann, K. Seth, and K. Yeo, “Communication-computation trade-offs in PIR,” in *30th USENIX Security Symposium (Security)*, 2021, pp. 1811–1828.
- [18] S. Patel, J. Y. Seo, and K. Yeo, “Don’t be dense: Efficient keyword PIR for sparse databases,” in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 3853–3870.
- [19] A. E. Johnson, T. J. Pollard, L. Shen *et al.*, “Mimic-iii, a freely accessible critical care database,” *Scientific data*, 2016.
- [20] “Lobster: Limit order book system - real-time event-based market data,” accessed: 2025-07-30. [Online]. Available: <https://lobsterdata.com/info/>
- [21] J. H. Lee, H. Zhang, V. Lagrange, P. Krishnamoorthy, X. Zhao, and Y. S. Ki, “Smartssd: Fpga accelerated near-storage data analytics on ssd,” *IEEE Computer architecture letters*, vol. 19, no. 2, pp. 110–113, 2020.
- [22] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab *et al.*, “Scaling memcache at facebook,” in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013, pp. 385–398.
- [23] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, “Tfhe: fast fully homomorphic encryption over the torus,” *Journal of Cryptology*, vol. 33, no. 1, pp. 34–91, 2020.
- [24] J. H. Cheon, A. Kim, M. Kim, and *et al*, “Homomorphic encryption for arithmetic of approximate numbers,” in *Springer International Conference on the Theory and Applications of Cryptology and Information Security (ASIACRYPT)*, 2017, pp. 409–437.
- [25] S. Halevi, Y. Polyakov, and V. Shoup, “An improved rns variant of the bfv homomorphic encryption scheme,” in *Proceedings of the Cryptographers’ Track at the RSA Conference (CT-RSA)*. Springer, 2019, pp. 83–105.
- [26] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Voshall, and W. Vogels, “Dynamo: amazon’s highly available key-value store,” in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2007, pp. 205–220.
- [27] G. Li, W. Tian, J. Zhang, R. Grosman, Z. Liu, and S. Li, “Gaussdb: A cloud-native multi-primary database with compute-memory-storage disaggregation,” *Proceedings of the VLDB Endowment*, vol. 17, no. 12, pp. 3786–3798, 2024.

- [28] J. Liu, J. Li, D. Wu, and K. Ren, "Pirana: Faster multi-query pir via constant-weight codes," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2023, pp. 39–39.
- [29] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.
- [30] J. Lin, L. Liang, Z. Qu, I. Ahmad, L. Liu, F. Tu, T. Gupta, Y. Ding, and Y. Xie, "Inspire: in-storage private information retrieval via protocol and architecture co-design," in *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA)*, 2022, pp. 102–115.
- [31] Y. Wang, X. Pan, Y. An, J. Zhang, and G. Reinman, "Beacongnn: Large-scale gnn acceleration with out-of-order streaming in-storage computing," in *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2024, pp. 330–344.
- [32] Z. Yang, Y. Lu, X. Liao, Y. Chen, J. Li, S. He, and J. Shu, "λ-IO: A unified IO stack for computational storage," in *21st USENIX Conference on File and Storage Technologies (FAST)*, 2023, pp. 347–362.
- [33] S. Wang, Z. Lin, S. Wu, H. Jiang, J. Zhang, and B. Mao, "Learnedftl: A learning-based page-level ftl for reducing double reads in flash-based ssds," in *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2024, pp. 616–629.
- [34] B. Tian, H. Liu, Z. Duan, X. Liao, H. Jin, and Y. Zhang, "Scalable billion-point approximate nearest neighbor search using smartssds," in *2024 USENIX Annual Technical Conference (USENIX ATC)*, 2024, pp. 1135–1150.
- [35] H. Jang, J. Song, J. Jung, J. Park, Y. Kim, and J. Lee, "Smart-infinity: Fast large language model training using near-storage processing on a real system," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2024, pp. 345–360.
- [36] Z. Duan, H. Feng, H. Liu, X. Liao, H. Jin, and B. Li, "AegonKV: A high bandwidth, low tail latency, and low storage cost KV-SeparatedLSM store with SmartSSD-basedGC offloading," in *23rd USENIX Conference on File and Storage Technologies (FAST)*, 2025, pp. 321–335.
- [37] S. Kang and S.-W. Jun, "Sting: Near-storage accelerator framework for scalable triangle counting and beyond," in *Proceedings of the 61st ACM/IEEE Design Automation Conference (DAC)*, 2024, pp. 1–6.
- [38] B. Tian, H. Liu, Y. Tang, S. Xiao, Z. Duan, X. Liao, H. Jin, X. Zhang, J. Zhu, and Y. Zhang, "Towards high-throughput and low-latency billion-scale vector search via CPU/GPU collaborative filtering and re-ranking," in *23rd USENIX Conference on File and Storage Technologies (FAST)*, 2025, pp. 171–185.
- [39] J. Lin, L. Liang, Z. Qu, I. Ahmad, L. Liu, F. Tu, T. Gupta, Y. Ding, and Y. Xie, "Inspire: in-storage private information retrieval via protocol and architecture co-design," in *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA)*, 2022, pp. 102–115.
- [40] J. H. Cheon, H. Choe, A. Passelègue, D. Stehlé, and E. Suvanto, "Attacks against the ind-cpad security of exact fhe schemes," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 2505–2519.
- [41] A. Munshi, "The openssl specification," in *2009 IEEE Hot Chips 21 Symposium (HCS)*, 2009, pp. 1–314.
- [42] Y. Zheng, Y. Tian, J. V. D'silva, and B. Kemme, "Dbmlsched: Scheduling in-database machine learning jobs," in *VLDB Workshops*, 2023.
- [43] Facebook, "Rocksdb," 2024, <https://github.com/facebook/rocksdb>.
- [44] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "Heax: An architecture for computing on encrypted data," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020, pp. 1295–1309.
- [45] Y. Yang, H. Zhang, S. Fan, H. Lu, M. Zhang, and X. Li, "Poseidon: Practical homomorphic encryption accelerator," in *Proceedings of IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 870–881.
- [46] Y. Zhu, X. Wang, L. Ju, and S. Guo, "Fxnenn: Fpga-based acceleration framework for homomorphic encrypted cnn inference," in *Proceedings of IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 896–907.
- [47] M. Han, Y. Zhu, Q. Lou, Z. Zhou, S. Guo, and L. Ju, "coxhe: A software-hardware co-design framework for fpga acceleration of homomorphic computation," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2022, pp. 1353–1358.
- [48] A. Şah Özcan and E. Savaş, "HEonGPU: a GPU-based fully homomorphic encryption library 1.0," *Cryptology ePrint Archive, Paper 2024/1543*, 2024.
- [49] Z. Cao, S. Dong, S. Vemuri, and D. H. Du, "Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook," in *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2020, pp. 209–223.
- [50] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proceedings of the ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, 2012, pp. 53–64.
- [51] J. Zhang, L. Peng, M. Sha, W. Liu, X. Li, S. Wang, F. Li, M. Gao, and H. Zhang, "Femur: A flexible framework for fast and secure querying from public key-value store," *Proceedings of the ACM on Management of Data (SIGMOD)*, vol. 3, no. 3, pp. 1–29, 2025.
- [52] C. H. Costa, P. Maia, F. Carlos *et al.*, "Sharding by hash partitioning," in *International Conference on Enterprise Information Systems (ICEIS)*, vol. 1, 2015, pp. 313–320.
- [53] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu, "Order preserving encryption for numeric data," in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data (SIGMOD)*, 2004, pp. 563–574.
- [54] D. Cash, J. Jaeger, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Roşu, and M. Steiner, "Dynamic searchable encryption in very-large databases: Data structures and implementation," *Cryptology ePrint Archive*, 2014.
- [55] Z. Gui, K. G. Paterson, S. Patrabis, and B. Warinschi, "Swissse: System-wide security for searchable symmetric encryption," *Proceedings on Privacy Enhancing Technologies (PET)*, 2024.
- [56] M. Bailleu, J. Thalheim, P. Bhatotia, C. Fetzer, M. Honda, and K. Vaswani, "SPEICHER: Securing LSM-based Key-Value stores using shielded execution," in *17th USENIX Conference on File and Storage Technologies (FAST)*, 2019, pp. 173–190.
- [57] I. Battiston, L. Feliuss, S. Ansmink, L. Kuiper, and P. Boncz, "Duckdb-sgx2: The good, the bad and the ugly within confidential analytical query processing," in *Proceedings of the 20th International Workshop on Data Management on New Hardware (DaMoN)*, 2024, pp. 1–5.
- [58] Y. Zou, Y. Li, S. Wang, L. Su, Z. Gu, Y. Lu, Y. Guan, D. Niu, M. Gao, Y. Xie, and L. Feifei, "Salus: A practical trusted execution environment for cpu-fpga heterogeneous cloud platforms," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024, pp. 252–266.
- [59] F. Falzon and E. A. Markatou, "Re-visiting authorized private set intersection: A new privacy-preserving variant and two protocols," *Proceedings on Privacy Enhancing Technologies (PET)*, 2025.
- [60] Y. Sun, J. Katz, M. Raykova, P. Schoppmann, and X. Wang, "Actively secure private set intersection in the client-server setting," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2024, p. 1478–1492.
- [61] H. Asi, F. Boemer, N. Genise, M. H. Mughees, T. Ogilvie, R. Rishi, K. Talwar, K. Tarbe, A. Wadia, R. Zhu *et al.*, "Scalable private search with wally," *arXiv preprint arXiv:2406.06761*, 2024.
- [62] O. Regev, "New lattice-based cryptographic constructions," *Journal of the ACM (JACM)*, vol. 51, no. 6, pp. 899–942, 2004.
- [63] S. Celi and A. Davidson, "Call me by my name: Simple, practical private information retrieval for keyword queries," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2024, pp. 4107–4121.
- [64] A. Davidson, G. Pestana, and S. Celi, "Frodopir: Simple, scalable, single-server private information retrieval," *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2023.
- [65] M. Maruseac, G. Ghinita, M. Ouyang, and R. Rughinis, "Hardware acceleration of private information retrieval protocols using gpus," in *IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2015, pp. 120–127.
- [66] M. Lam, J. Johnson, W. Xiong, K. Maeng, U. Gupta, Y. Li, L. Lai, I. Leontiadis, M. Rhu, H.-H. S. Lee *et al.*, "Gpu-based private information retrieval for on-device machine learning inference," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024, pp. 197–214.