Hermes: High-Performance Homomorphically Encrypted Vector Databases

Dongfang Zhao
High-Performance Data-Intelligent Computing (HPDC) Lab
dzhao@uw.edu

Abstract

Fully Homomorphic Encryption (FHE) has long promised the ability to compute over encrypted data without revealing sensitive contents—a foundational goal for secure cloud analytics. Yet despite decades of cryptographic advances, practical integration of FHE into real-world relational databases remains elusive. This paper presents **Hermes**, the first system to enable FHE-native vector query processing inside a standard SQL engine. By leveraging the multi-slot capabilities of modern schemes, Hermes introduces a novel data model that packs multiple records per ciphertext and embeds encrypted auxiliary statistics (e.g., local sums) to support in-place updates and aggregation. To reconcile ciphertext immutability with record-level mutability, we develop new homomorphic algorithms based on slot masking, shifting, and rewriting. Hermes is implemented as native C++ loadable functions in MySQL using OpenFHE v1.2.4, comprising over 3,500 lines of code. Experiments on real-world datasets show up to 1,600× throughput gain in encryption and over 30× speedup in insertion compared to per-tuple baselines. Hermes brings FHE from cryptographic promise to practical reality—realizing a long-standing vision at the intersection of databases and secure computation.

CCS Concepts

• **Security and privacy → Cryptography**; • **Information systems → Database management systems**.

Keywords

homomorphic encryption, encrypted databases, query processing, SIMD packing

1 Introduction

1.1 Background and Motivation

For over four decades, database systems have formed the backbone of modern data infrastructure, enabling structured storage, efficient retrieval, and scalable analytics. As the industry has transitioned to cloud-first architectures, the confidentiality of outsourced data has become a central concern. Increasingly, users demand end-to-end cryptographic protection—not merely during transmission or at rest, but throughout the entire computation pipeline.

Among all cryptographic primitives, Fully Homomorphic Encryption (FHE) stands out for its conceptual power: it allows arbitrary computation on encrypted data, thereby eliminating the need to trust the compute server. Since Gentry's seminal construction [?], subsequent schemes such as BFV and CKKS [? ?] have made dramatic advances in efficiency, expressiveness, and implementation maturity [?]. Today, FHE is no longer a theoretical curiosity—it underpins practical tools in secure machine learning and privacy-preserving analytics [? ?].

And yet, one foundational system remains conspicuously absent from this evolution: the relational database. Despite decades of research on encrypted databases [? ?], most practical systems either leak access patterns through auxiliary indexing or offload all computation to the client. The longstanding vision of a relational engine that operates directly on semantically secure ciphertexts remains, as of today, unrealized.

This disconnect is not for lack of demand or cryptographic capability, but arises from a deeper structural asymmetry: the absence of a principled interface between database semantics and homomorphic computation. Traditional DBMSs operate over mutable records, evolving schemas, and runtime query plans. FHE schemes, by contrast, operate over static circuits and immutable ciphertexts. The two abstractions—relational and algebraic—have matured in parallel, but seldom intersected in implementation or semantics.

Bridging this divide requires more than encrypting inputs or wrapping FHE inside SQL. It demands a new class of system architecture—one that reconciles the structural invariants of encrypted computation with the operational dynamics of relational execution. Without such an alignment, even the most powerful cryptographic schemes remain impractical for secure query processing.

As we stand at the convergence of deployable FHE toolchains and rising expectations for secure cloud analytics, the foundational question is no longer *whether* relational systems can support native homomorphic execution—but *how*, and with what interface. Our motivation is to take a principled step toward answering that question.

1.2 Proposed Work

We present **Hermes**, a database system that enables secure and efficient vectorized query execution over homomorphically encrypted data. Built atop the multi-slot capabilities of modern FHE schemes, Hermes is implemented as a suite of native MySQL loadable functions interfacing with OpenFHE. It addresses both algorithmic and systems-level challenges that arise when adapting FHE to relational query processing. The system contributes the following core components:

(1) Vectorized Ciphertext Encoding. Hermes adopts a packed representation in which each ciphertext encodes a fixed-length array of attribute values, corresponding to multiple database tuples. This SIMD-style layout amortizes encryption overhead and enables batch homomorphic operations. Unlike conventional per-tuple encryption, whose cost grows linearly with table size, Hermes aligns computational complexity with the number of ciphertexts, improving both throughput and storage efficiency.

(2) Auxiliary Sum Embedding for Aggregation. To accelerate homomorphic aggregation, Hermes reserves the final slot of each ciphertext to store a *local sum*—a precomputed additive statistic

over the remaining slots. This value is computed at packing time and incrementally updated upon insertion and deletion. As a result, encrypted aggregation queries (e.g., `SUM`, `GROUP BY`) can be answered without invoking Galois rotations or slot-wise scan circuits. Rotations are used only in rare structural updates (e.g., middle-slot deletion).

(3) Slot-Level Updates via Encrypted Shift Operations. Hermes supports dynamic record-level updates directly on packed ciphertexts. Insertions shift a tail segment rightward to insert a new value; deletions shift leftward and clear the trailing slot. Both operations are implemented using homomorphic masking and slot-wise addition, allowing structural updates without re-encryption or plaintext recovery. These operations preserve semantic security and enable efficient encrypted mutability.

(4) Non-Intrusive Integration with MySQL.. Hermes is realized as a set of C++ plugins compiled against OpenFHE v1.2.4 and registered through the MySQL UDF mechanism. The system requires no changes to the MySQL engine or query planner. Ciphertexts are stored as string-encoded blobs (e.g., base64), and all encryption contexts and key material are globally initialized. The codebase comprises over 2,000 lines of cryptographic logic, supported by shell and Python scripts for deployment and automation.

(5) Encrypted Query Execution Pipeline. Hermes supports full-stack encrypted query execution for common aggregate patterns. During ingestion, records are packed into ciphertexts with embedded auxiliary sums. At query time, ciphertexts are aggregated via homomorphic addition, and the result is decrypted in a single step. This design minimizes ciphertext transformations and avoids client-side orchestration, enabling secure and performant query execution entirely within the database engine.

1.3 Contributions

This paper makes the following contributions:

- **Packed Vector Representation for Encrypted Tuples.** We propose a vectorized ciphertext encoding in which each homomorphic ciphertext stores multiple database tuples in parallel slots. This layout exploits the SIMD capabilities of modern FHE schemes to amortize encryption overhead and reduce per-tuple processing cost.
- **Auxiliary Local Sum Embedding.** To support efficient encrypted aggregation, we embed a slot-resident auxiliary sum into each ciphertext at packing time. This value is incrementally maintained under updates and enables encrypted `SUM` queries without requiring Galois rotations in the common case.
- **In-Place Encrypted Slot Manipulation.** We introduce homomorphic primitives for slot-level insertion and deletion using encrypted shift-and-mask operations. These allow structural updates to packed ciphertexts without decryption or full re-encryption, supporting secure and dynamic record-level mutability.
- **Drop-In Integration with MySQL via UDFs.** We implement Hermes as a collection of C++ loadable functions interfacing OpenFHE with the MySQL UDF mechanism. The system introduces no changes to the MySQL engine and executes all encryption logic natively within SQL workflows.
- **Secure End-to-End Query Execution within SQL.** Hermes supports full homomorphic execution pipelines for aggregate SQL queries, including `SELECT SUM` and `GROUP BY`. These are executed securely and efficiently inside the DBMS without external orchestration or client-side coordination.

2 Preliminaries

2.1 Homomorphic Encryption Background

Hermes is built atop the BFV scheme [**?**], a lattice-based fully homomorphic encryption (FHE) construction supporting exact arithmetic over integers modulo a plaintext modulus t. Let $R = \mathbb{Z}[X]/(X^N+1)$ be a cyclotomic ring of degree $N = 2^k$ for some $k \in \mathbb{N}$. A plaintext message is represented as an element of $R_t := R/tR$, while ciphertexts reside in R_q^2, where $q \gg t$ is a large ciphertext modulus.

A key feature of BFV is its support for *plaintext packing* via the Chinese Remainder Theorem (CRT). When $t \equiv 1 \mod 2N$, the ring R_t splits as a direct product of $n = N/2$ slots:

$$R_t \cong \mathbb{Z}_t^n,$$

enabling SIMD-style evaluation of vectorized inputs. Each plaintext $\mathbf{m} = (m_0, \dots, m_{n-1}) \in \mathbb{Z}_t^n$ is encoded into a single polynomial $m(X) \in R_t$, encrypted as $\mathsf{Enc}(m) = \mathbf{c} \in R_q^2$, and evaluated homomorphically across all slots.

Homomorphic operations preserve slot-wise semantics:

$$\mathsf{EvalAdd}(\mathsf{Enc}(\mathbf{m}), \mathsf{Enc}(\mathbf{m}')) = \mathsf{Enc}(\mathbf{m} + \mathbf{m}'),$$
$$\mathsf{EvalMult}(\mathsf{Enc}(\mathbf{m}), \mathsf{Enc}(\mathbf{m}')) = \mathsf{Enc}(\mathbf{m} \cdot \mathbf{m}'),$$

where the arithmetic is component-wise modulo t. In addition, the BFV scheme supports automorphism-based rotation operations that cyclically shift slot positions:

$$\mathsf{EvalRotate}(\mathsf{Enc}(\mathbf{m}), r) = \mathsf{Enc}((m_{(i+r) \bmod n})_{i=0}^{n-1}).$$

Hermes leverages this packing structure to encode multiple database tuples into a single ciphertext. This design enables parallel evaluation, slot-wise updates, and sublinear aggregation. Importantly, all packing and update logic remains entirely within the encrypted domain, without leaking intermediate values or structural metadata.

In addition, the BFV scheme supports automorphism-based slot rotation using Galois group actions. For any $r \in \mathbb{Z}$, a cyclic slot rotation is realized via the Galois automorphism $\sigma_r : X \mapsto X^r \mod (X^N+1)$, which acts on ciphertexts by permuting their internal slot layout. To evaluate this transformation homomorphically, the client must precompute and upload the corresponding Galois keys:

$$\mathsf{EvalAtIndex}(\mathsf{Enc}(\mathbf{m}), r) = \mathsf{Enc}((m_{(i+r) \bmod n})_{i=0}^{n-1}),$$

where the rotation index r corresponds to a specific automorphism in the group $\mathrm{Gal}(R_q/\mathbb{Q})$. These keys enable Hermes to perform selective slot access, masking, and reorganization within encrypted vectors while preserving semantic security.

2.2 Provable Security

Semantic security for public-key encryption is captured by the notion of indistinguishability under chosen-plaintext attack (IND-CPA). Let $\Pi = (\mathsf{KeyGen}, \mathsf{Encrypt}, \mathsf{Decrypt}, \mathsf{EvalAdd}, \mathsf{EvalMult})$ be a probabilistic encryption scheme defined over a message space $\mathcal{M}$ and a ciphertext space C. The scheme Π is IND-CPA secure if no probabilistic polynomial-time (PPT) adversary can, with non-negligible advantage, distinguish between the encryptions of any two chosen messages in $\mathcal{M}$.

Formally, consider the following security game between a challenger and an adversary $\mathcal{A}$:

(1) The challenger generates a key pair $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(\lambda)$ for security parameter λ, and sends pk to $\mathcal{A}$.
(2) The adversary chooses two equal-length messages $m_0, m_1 \in \mathcal{M}$ and submits them to the challenger.
(3) The challenger samples a uniform bit $b \in \{0, 1\}$, computes $c^* \leftarrow \mathsf{Encrypt}_{\mathsf{pk}}(m_b)$, and returns c^* to $\mathcal{A}$.
(4) The adversary outputs a guess $b' \in \{0, 1\}$.

The scheme is IND-CPA secure if, for all PPT adversaries $\mathcal{A}$,

$$\left| \Pr\left[\mathcal{A}(\mathsf{pk}, c^*) = b\right] - \frac{1}{2} \right| < \varepsilon(\lambda),$$

where ε is a negligible function in λ.

Modern fully homomorphic encryption schemes such as BFV, BGV, and CKKS satisfy IND-CPA under the hardness assumption of the Ring Learning With Errors (RLWE) problem. For these schemes, the encryption process is randomized: each plaintext message is masked with independently sampled noise and embedded in a structured polynomial ring, ensuring ciphertext distributions are computationally indistinguishable from random.

The multi-slot encoding structure introduced in BFV and related schemes does not weaken IND-CPA security. A vector $\mathbf{m} = (m_0, \ldots, m_{n-1}) \in \mathcal{M}^n$ is encrypted into a single ciphertext $c \in C$, but as long as Encrypt remains probabilistic and the adversary lacks access to decryption oracles, semantic security extends to the full plaintext vector.

Subsequent homomorphic operations (e.g., addition, multiplication, rotation) are performed over ciphertexts without leaking intermediate plaintexts. Since the adversary cannot observe any result of decryption or affect the noise distribution, the composed evaluation remains semantically secure under standard composition theorems.

This security model provides the foundation upon which encrypted database systems can be constructed: so long as no decryption is performed server-side, and all query logic is implemented using semantically secure operations, the confidentiality of underlying plaintexts is preserved.

2.3 MySQL User-Defined Functions (UDFs)

MySQL supports native extensibility through *User-Defined Functions* (UDFs), which allow developers to register custom functions written in C/C++ as shared objects. Once registered, a UDF can be invoked in SQL queries as a first-class function, operating over tuples during query execution.

A typical scalar UDF consists of the following exported routines:

- `foo_init` (`foo_init`): initializes internal memory and checks argument types.
- `foo` (`foo`): performs the main per-tuple computation.
- `foo_deinit` (`foo_deinit`): releases allocated memory and closes resources.

Aggregate UDFs additionally implement:

- `foo_add` (`foo_add`): accumulates intermediate results across tuples.
- `foo_clear`, `foo_reset` (`foo_clear`, `foo_reset`): manage state lifecycle during grouping.

Each UDF receives input arguments via the `UDF_ARGS` structure, which exposes parameter values and types through typed pointers. Output is returned through a caller-allocated buffer and must specify an explicit output length. All memory is managed manually, and no standard C++ types (e.g., `std::string`) may be used within the interface boundary.

This plugin model maps naturally to encrypted computation. The per-tuple stateless execution model aligns with ciphertext-level operations, while aggregate UDF hooks enable accumulation over encrypted records. Importantly, UDFs operate entirely inside the database engine, avoiding the need for external processes or client-managed cryptographic operations.

Hermes implements each homomorphic operation as a standalone UDF module, ensuring modularity and compatibility with SQL queries. Operations such as encryption, slot-wise insertion, and encrypted summation are exposed as standard SQL functions, compiled into dynamically loadable `.so` libraries. This design permits seamless integration with the MySQL query planner and execution engine, bridging encrypted computation with traditional relational workflows.

2.4 Terminology and Notation

We summarize key terms and notation used throughout this paper. Let n denote the number of slots supported by the plaintext modulus and ring parameters of the BFV encryption scheme. All vectors are implicitly assumed to be over $\mathbb{Z}_t^n$, where t is the plaintext modulus.

- **Slot:** Each ciphertext encodes a vector of n plaintext elements, known as *slots*. Slots are independently addressable under SIMD operations supported by BFV.
- **Ciphertext:** A ciphertext $c \in C$ is a BFV-encoded object that homomorphically encrypts $[m_0, \ldots, m_{n-1}]$ for some plaintext values $m_i \in \mathbb{Z}_t$. Ciphertexts may undergo operations such as addition, multiplication, and automorphic permutation.
- **Group ID:** In Hermes, input tuples are partitioned by a logical *group ID*, which assigns each record to a packed vector instance. All records within the same group are encoded into the same ciphertext. The mapping from record to group is deterministic and index-driven.
- **Auxiliary Sum Slot:** Hermes reserves the final slot (slot $n-1$) in each ciphertext to store the *local sum*, i.e., the plaintext sum $\sum_{j=0}^{n-2} m_j$ of all payload slots. This value is computed at packing time and used to accelerate encrypted aggregation without rotation.
- **Packed Representation:** A ciphertext that stores $n-1$ application values and one auxiliary statistic is referred to as a

packed representation. This format enables efficient updates and aggregation via SIMD-aware primitives.

- **Logical Length:** Some plaintext vectors do not occupy all n slots. The number of meaningful entries is called the *logical length*, and is used to control serialization, summation, and slot masking operations.

3 Hermes: A High-Performance FHE Layer for Vectorized Databases

Hermes is a MySQL-integrated system that introduces a new data abstraction for storing, updating, and querying encrypted relational data via fully homomorphic encryption (FHE). It is built upon the multi-slot property of lattice-based FHE schemes (e.g., BFV), where a single ciphertext can encode a vector of plaintexts. We identify three core technical contributions that together enable a performant and provably secure FHE-based query engine.

3.1 Packed Ciphertext Data Model

We propose a new packing-based layout where each ciphertext holds a fixed-size group of database records. This model amortizes encryption cost and supports efficient SIMD-style processing. To enable insert and delete operations over individual slots, we design a homomorphic slot manipulation interface that combines masking, shifting, and slot-zeroing operations, all within the ciphertext algebra. Unlike conventional systems that require decryption and re-encryption for each update, our design ensures that updates can be applied entirely in the encrypted domain.

3.1.1 Slot-Aware Packing Strategy. In Hermes, we introduce a structured packing layout that enables efficient storage and manipulation of encrypted data within the SIMD-style slot architecture of the BFV homomorphic encryption scheme. Rather than encrypting each tuple into a separate ciphertext—which incurs significant memory and performance overhead—we adopt a batched layout where each ciphertext encodes a fixed-size vector of plaintext values, referred to as *slots*.

Let N denote the ring dimension of the BFV scheme. When batching is enabled (via the Chinese Remainder Theorem on cyclotomic rings), the scheme yields $n = N/2$ usable slots. For example, with $N = 8192$, Hermes can store $n = 4096$ individual scalar values per ciphertext. However, instead of utilizing all n slots for raw data, we reserve one designated slot (typically the last) to store an auxiliary value, such as the *local sum* of the vector entries. This yields an effective capacity of $n - 1$ logical records per ciphertext.

Each slot in a packed ciphertext has a fixed positional meaning: slot i corresponds to the i-th logical record in the group. This deterministic alignment enables precise slot-wise reasoning and efficient update semantics. When a ciphertext is decrypted, the unpacked vector reveals values at well-defined offsets, allowing Hermes to reconstruct the plaintext table fragment without additional metadata. This position-indexed design is critical for supporting secure in-place modifications—such as insertions and deletions—without leaking tuple identity or requiring explicit key-column mapping.

Formally, a packed ciphertext c in Hermes can be viewed as a vector:

$$c = \text{Enc}([v_0, v_1, \ldots, v_{n-2}, \sigma]),$$

where v_i is the plaintext value in logical slot i, and σ is the slot reserved for auxiliary information, such as $\sigma = \sum_{i=0}^{n-2} v_i$ in the case of sum queries.

Compared to traditional row-wise encryption, this layout significantly improves storage density and amortizes encryption cost across multiple tuples. It also enables efficient homomorphic operations such as component-wise masking, conditional updates, and encrypted aggregation over slots. In essence, we reinterpret ciphertexts as *relational vectors* whose entries correspond to logical rows in a database group, thereby enabling vectorized query execution over encrypted domains.

3.1.2 Homomorphic Insertion. Hermes supports in-place insertion of plaintext values into packed ciphertexts without requiring decryption or client-side re-encryption. The goal is to insert a new value v into logical slot i of an existing ciphertext c, where slots i through $n - 2$ may already contain data. Since ciphertexts are encoded as vectors of fixed size, insertion requires shifting all subsequent entries rightward by one position to make room for the new value. This is analogous to an in-place insert into an array but performed entirely under encryption.

The insertion procedure consists of three homomorphic operations: (1) Galois rotation, (2) plaintext masking, and (3) ciphertext addition. We first perform a Galois right-rotation on c, yielding a new ciphertext c_{rot} in which all slots are shifted by one position. However, this operation naively shifts all slots, including the local sum at slot $n - 1$, which must be preserved. Therefore, Hermes applies a plaintext mask m_{preserve} to c before rotation to ensure that only slots i through $n - 2$ are moved. This yields a masked-and-rotated ciphertext c_{shifted} in which slot i is now vacant.

Next, the new plaintext value v is embedded into a one-hot plaintext vector m_v where $m_v[i] = v$ and $m_v[j] = 0$ for $j \neq i$. This mask is encrypted into c_v, a ciphertext containing v only at the desired slot. Finally, we compute:

$$c' = \text{EvalAdd}(c_{\text{shifted}}, \text{EvalMult}(c_v, m_i)),$$

where m_i is a binary plaintext mask that ensures v is added only at slot i. This avoids interference with other slots.

In addition to updating the main data slots, Hermes updates the local sum σ stored at slot $n - 1$. This is achieved via scalar plaintext addition:

$$\sigma' = \text{EvalAddPlain}(\sigma, v).$$

Since v is known in plaintext, the update to the auxiliary slot incurs negligible overhead and does not require re-encryption.

Importantly, this entire procedure is performed within the MySQL server process using OpenFHE's C++ API, without leaking which slot was modified or what value was inserted. The result is a new ciphertext that logically contains all previous values, v inserted at slot i, and an updated sum at slot $n - 1$. This mechanism allows Hermes to support encrypted ingestion of streaming data with minimal cryptographic cost.

When a ciphertext pack reaches the maximum number of supported slots—i.e., the FHE ring dimension divided by 2—no further insertions can occur without creating a new ciphertext. In Hermes, we adopt a conservative policy: each group is associated with a single ciphertext, and insertion beyond capacity is disallowed. Future extensions may support *chained packing*, where overflow records spill into a second ciphertext. However, doing so requires maintaining slot alignment metadata and reintroduces some of the per-ciphertext management overhead that Hermes aims to avoid.

3.1.3 Homomorphic Deletion. Hermes enables encrypted deletion over packed ciphertexts by simulating a logical left-shift over the underlying slots while preserving the ciphertext algebra. The goal is to remove the value at slot i of ciphertext c, and to maintain a compacted layout where all subsequent values are shifted leftward and the last slot is cleared. Importantly, this is done entirely within the encrypted domain, without re-encoding the ciphertext or exposing the deletion position to the server.

The deletion procedure consists of three conceptual steps: (1) rotation, (2) masking, and (3) auxiliary slot update. First, Hermes isolates the subvector of ciphertext c corresponding to slots $i+1$ through $n-2$ and rotates it leftward by one position. This is accomplished via a Galois left rotation:

$$c_{\text{rot}} = \text{EvalRotate}(c, -1),$$

followed by a masking step to ensure that only the intended region is affected. Specifically, a plaintext mask m_{shift} is applied before rotation to preserve slot integrity and avoid accidental overwriting of unaffected entries.

Next, Hermes applies a zeroing mask z to clear the final slot (slot $n-2$), which now contains a duplicate or stale value as a result of the shift. The mask z is a binary plaintext vector where $z[j] = 1$ for $j < n-2$ and $z[n-2] = 0$, ensuring:

$$c' = \text{EvalMult}(c_{\text{rot}}, z).$$

The slot at index i—which originally contained the value to delete—has now been overwritten by the shifted entry from $i+1$, and the final slot is set to zero, restoring the invariant that only $n-1$ data entries are active.

Finally, the local sum maintained in slot $n-1$ is updated by subtracting the deleted plaintext value v. Since v is assumed to be known at deletion time (e.g., from a query input), this update is executed via plaintext subtraction:

$$\sigma' = \text{EvalSubPlain}(\sigma, v).$$

All steps—rotation, masking, and sum adjustment—are performed homomorphically within the database server using OpenFHE APIs. This design avoids decryption-reencryption cycles, ensures slot occupancy consistency, and maintains statistical privacy regarding which slot was deleted. As with insertion, the deletion logic treats ciphertexts as logical arrays and preserves relational semantics under homomorphic constraints.

When all payload slots in a packed ciphertext have been deleted, the ciphertext becomes logically empty, although its auxiliary sum slot may still contain residual values. In Hermes, such ciphertexts are treated as semantically inert: they contribute nothing to aggregate queries, and further deletions are treated as no-ops. However, the ciphertext itself is retained in the database to preserve structural consistency unless explicitly garbage-collected. To maintain correctness, Hermes ensures that the auxiliary sum is updated to zero whenever the payload becomes empty. This guarantees that global aggregation remains accurate even in the presence of entirely deleted groups.

3.2 Local Sum Maintenance for Aggregate Queries

To support efficient encrypted aggregation, Hermes maintains a running *local sum* for each packed ciphertext. This design allows queries such as `SELECT SUM(attr)` or `GROUP BY key` to be evaluated via lightweight ciphertext addition without slot-wise traversal or Galois rotations. The following subcomponents describe how the local sum is constructed, maintained, and queried.

3.2.1 Local Sum Encoding at Packing Time. Hermes adopts a novel hybrid encoding strategy in which each packed ciphertext stores not only a fixed-length array of plaintext values but also a precomputed auxiliary statistic: the sum of those values. This local sum is embedded into the last slot of the ciphertext at the time of packing, enabling efficient aggregation queries without the need for slot-wise traversal or ciphertext rotations.

Given a batch of $n-1$ plaintext values $(v_0, v_1, \ldots, v_{n-2})$ to be packed into a ciphertext, Hermes computes their sum $s = \sum_{i=0}^{n-2} v_i$ in plaintext and constructs an extended vector of the form:

$$\mathbf{m} = (v_0, v_1, \ldots, v_{n-2}, s) \in \mathbb{Z}_t^n,$$

where t is the plaintext modulus and n is the total number of BFV batching slots (typically $n = N/2$ for a ring dimension N). The extended vector $\mathbf{m}$ is then passed to the OpenFHE API to generate a packed plaintext object, which is encrypted into a ciphertext $c = \text{Encrypt}(pk, \text{MakePackedPlaintext}(\mathbf{m}))$.

This packing layout effectively reserves the final slot (slot $n-1$) as a local accumulator, denoted as $\text{sum}_{\text{local}}$. Because BFV ciphertexts support SIMD-style processing, all data and the auxiliary sum coexist in a single ciphertext with minimal overhead. Moreover, since the sum is computed before encryption, it incurs no additional homomorphic cost.

The benefit of this encoding is twofold. First, it amortizes the cost of sum computation over all packed values, reducing the number of ciphertexts needed for downstream aggregation. Second, it enables highly optimized aggregate queries—such as `SELECT SUM(attr)`—by reducing global summation to a single ciphertext-level addition over the final slot of each ciphertext. Instead of rotating and adding up all slots, the query planner only needs to sum the values at slot $n-1$ across all ciphertexts:

$$\text{SUM(attr)} = \text{EvalAdd}(c_1[n-1], c_2[n-1], \ldots, c_k[n-1]).$$

This approach dramatically reduces computational complexity and latency, particularly when processing large encrypted datasets. By pre-allocating and embedding local statistics into the ciphertext layout, Hermes transforms aggregation from a slot-wise operation into a slot-constant one, leveraging the structure of the BFV scheme for practical query acceleration.

3.2.2 Incremental Sum Update on Insertion and Deletion. While Hermes encodes a local sum at packing time, database workloads often require dynamic updates. To ensure the correctness of aggregation queries over mutable encrypted data, we design a lightweight incremental strategy to maintain the local sum as records are inserted or deleted. This strategy preserves consistency between the payload slots and the auxiliary sum slot, while avoiding costly recomputation or re-encryption.

Consider a ciphertext c whose first $n-1$ slots contain encrypted values $(v_0, \dots, v_{n-2})$, and whose final slot (slot $n-1$) stores the corresponding local sum $s = \sum_{i=0}^{n-2} v_i$. When a new value v_{new} is inserted into the payload (e.g., at slot i), Hermes also adds v_{new} into the local sum via scalar plaintext addition. This is performed by constructing a plaintext vector $\mathbf{m}_{\text{aux}}$ such that:

$$\mathbf{m}_{\text{aux}} = (0, 0, \dots, 0, v_{\text{new}}),$$

where only the final slot is nonzero. Then the updated ciphertext c' is computed homomorphically as:

$$c' = \text{EvalAdd}(c, \mathbf{m}_{\text{aux}}).$$

This operation increases the local sum while preserving the rest of the payload unchanged. Note that no re-encryption is required, and the update remains entirely within the homomorphic domain.

A similar process applies to deletions. Suppose a value v_{del} is removed from a known slot. Hermes constructs a plaintext subtraction mask $\mathbf{m}'_{\text{aux}}$ where:

$$\mathbf{m}'_{\text{aux}} = (0, 0, \dots, 0, -v_{\text{del}}),$$

and applies:

$$c' = \text{EvalAdd}(c, \mathbf{m}'_{\text{aux}}).$$

The net effect is that the auxiliary slot now holds the correct sum of the modified payload. Since inserted and deleted values are supplied in cleartext as part of the query expression, this approach avoids the overhead of decrypting and summing the full ciphertext.

This fine-grained sum adjustment strategy is a key enabler of real-time encrypted aggregation. It ensures that the auxiliary slot always remains synchronized with the payload, supporting fast 'GROUP BY' and 'SUM' queries without compromising correctness or requiring client interaction. All updates are executed inside MySQL via homomorphic functions and do not reveal which slot was updated.

3.2.3 Aggregation Without Rotation. One of the most computationally expensive operations in homomorphic encryption is ciphertext rotation. In traditional packing-based FHE systems, computing an aggregate such as `SELECT SUM(attr)` requires repeated Galois rotations and additions to bring all slot values into alignment before summing them. For a ciphertext with n slots, this requires up to $\log_2(n)$ rotation keys and roughly $O(\log n)$ homomorphic operations per ciphertext—an unacceptable overhead for practical encrypted query processing.

Hermes avoids this performance bottleneck through its local sum embedding strategy. As described earlier, each packed ciphertext in Hermes stores the total sum of its payload values in a dedicated auxiliary slot (typically slot $n-1$). Because this slot location is consistent and known across all ciphertexts, global aggregation reduces to a single ciphertext-wise vector addition, where only the auxiliary slot participates semantically in the computation.

Formally, given ciphertexts $c_1, c_2, \dots, c_k$ representing k disjoint data groups, each containing a local sum in slot $n-1$, Hermes computes the global sum as:

$$c_{\text{global}} = \text{EvalAdd}(c_1, c_2, \dots, c_k),$$

followed by extracting slot $n-1$ of c_{global} during decryption. This process eliminates all ciphertext rotations, slot traversals, and per-slot masking. As the number of ciphertexts increases, the amortized cost of aggregation remains constant per ciphertext.

Moreover, since the local sums are already encrypted, no additional encryption overhead is incurred at query time. The aggregation logic requires only one homomorphic addition per ciphertext and a single decryption at the end. This makes Hermes especially suitable for large-scale encrypted OLAP workloads with heavy use of 'SUM', 'GROUP BY', and related operations.

In effect, Hermes shifts the computational burden from query execution to data preparation (packing), optimizing for the read-mostly query patterns common in analytical database systems. This trade-off yields substantial performance improvements in practice: we observe up to 10× reduction in aggregation latency compared to baseline packing schemes that rely on slot-wise summation with rotations.

By aligning the data model with ciphertext algebra, Hermes achieves near-constant-time aggregation over encrypted data, demonstrating that homomorphic query execution can be made practical for common SQL aggregates with the right structural design.

3.3 Security Analysis and Proof of Correctness

We provide a formal security analysis of Hermes under the standard IND-CPA model. Our design avoids client-server interaction after encryption and ensures that all slot-level operations are data-independent, thereby preventing leakage from slot position, access pattern, or update frequency. We further prove that the packed representation with embedded statistics does not weaken the semantic security of the base FHE scheme, and our masking/shift update primitives are indistinguishable under standard lattice assumptions.

3.3.1 Security Model and Threat Assumptions. We analyze the security of Hermes under the standard indistinguishability under chosen-plaintext attack (IND-CPA) model, instantiated using the BFV fully homomorphic encryption (FHE) scheme. Our system assumes a passive, honest-but-curious adversary with full access to encrypted database contents, including ciphertexts before and after updates, as well as query results. The adversary may observe ciphertext structure, slot layout, and numerical patterns in decrypted outputs but is assumed to follow protocol without actively deviating from the query logic.

We assume the following attacker capabilities:

- **Ciphertext Access:** The server adversary has full access to the database ciphertexts and can observe any changes over time, including ciphertext insertions, deletions, and group-level rewrites.
- **Query Transcript Observation:** The adversary sees the encrypted query inputs, results, and any auxiliary ciphertexts generated during evaluation (e.g., temporary ciphertexts for masking or rotation).
- **Packing Metadata Awareness:** The attacker knows the ciphertext packing format, including how many slots are reserved, and which slot is used for the auxiliary local sum.

- **No Side Channels:** The attacker does not observe low-level timing, memory access, or power traces. Preventing side-channel leakage is beyond the scope of this work.

Our primary goal is to ensure that Hermes does not introduce any new leakage channels beyond what is inherent in the underlying FHE scheme. In particular, we require the following invariants to hold:

(1) **Indistinguishability of Ciphertexts:** For any two payload vectors v and v' of the same length, their packed ciphertexts $c = \texttt{Encrypt}(v)$ and $c' = \texttt{Encrypt}(v')$ are computationally indistinguishable under the IND-CPA security of BFV.
(2) **Update Pattern Privacy:** Insertion and deletion operations do not reveal the affected slot position or the identity of the modified value to the server. That is, for any two slots i and j, the adversary cannot distinguish an insertion at slot i from one at j by examining ciphertext differences alone.
(3) **Aggregation Consistency:** Global aggregation over auxiliary slots (e.g., encrypted 'SUM') reveals no more information than the final aggregate result; the per-ciphertext local sum values are protected under the encryption scheme.
(4) **No Adaptive Leakage:** Hermes does not reveal any intermediate plaintexts or randomness that could be used to link ciphertexts over time. All operations are performed homomorphically and reuse no client secrets post-encryption.

Importantly, the Hermes data model and update interface preserve these properties without relying on secure multi-party computation (MPC), secure enclaves, or trusted client-side computation. All operations—packing, insert/delete, aggregation—are fully executed on encrypted data within the untrusted MySQL server.

This model captures realistic cloud deployment scenarios where users outsource encrypted databases to an untrusted cloud provider but wish to retain strong confidentiality guarantees under standard cryptographic assumptions (e.g., Ring-LWE hardness). We note that this threat model is stronger than many property-preserving encryption (PPE) systems, which often leak order, frequency, or access pattern by design.

3.3.2 Preservation of IND-CPA Security. The semantic security of Hermes relies on the assumption that the underlying BFV scheme remains IND-CPA secure under all ciphertext manipulations performed in the system. This includes ciphertext packing, masked updates, homomorphic insertion and deletion, and the embedding of auxiliary local statistics such as the group sum. We now formally argue that Hermes does not introduce any structural leakage or functional dependencies that could be exploited to distinguish ciphertexts.

First, although Hermes embeds deterministic metadata (e.g., local sums) into specific slots, the entire ciphertext is encrypted using randomized BFV encryption over the full plaintext vector. IND-CPA security guarantees that the ciphertext remains indistinguishable even if some slots are derived from others, so long as the message vector is encrypted as a whole with fresh randomness. In particular, the inclusion of auxiliary statistics does not compromise the semantic security of any individual slot.

Second, Hermes performs all slot-level updates—such as insertion and deletion—purely through homomorphic operations. These updates use masking, shifting, and slot-wise additions, but never invoke decryption, re-encryption, or key switching. No new randomness is injected, and no plaintext values are revealed or reconstructed during runtime. As a result, each modified ciphertext maintains the same IND-CPA security level as its original version, with no additional leakage beyond the allowed ciphertext interface.

Definition (IND-CPA Security). Let $\mathsf{Enc}_{\mathsf{pk}}(m; r)$ denote the encryption of a plaintext m under public key pk and randomness r. A scheme is IND-CPA secure if no probabilistic polynomial-time (PPT) adversary $\mathcal{A}$ can distinguish between encryptions of m_0 and m_1 for any chosen pair of equal-length plaintexts, even given arbitrary access to the encryption oracle. Formally, for any PPT adversary $\mathcal{A}$, the advantage

$$\mathrm{Adv}_{\mathcal{A}}^{\text{IND-CPA}} = |\Pr[\mathcal{A}(\mathsf{Enc}(m_0)) = 1] - \Pr[\mathcal{A}(\mathsf{Enc}(m_1)) = 1]|$$

is negligible in the security parameter λ.

Lemma 1. Let C be the set of ciphertexts generated and manipulated by Hermes. Then under the assumption that the base FHE scheme (BFV) is IND-CPA secure, the ciphertexts $c \in C$ remain IND-CPA secure throughout all operations in Hermes.

Proof Sketch. The proof proceeds by reduction. Suppose there exists an adversary $\mathcal{A}_{\text{Hermes}}$ that can distinguish two ciphertexts $c_0, c_1 \in C$ produced under Hermes' operations with non-negligible advantage. Then we construct an adversary $\mathcal{A}_{\text{BFV}}$ that breaks IND-CPA security of the underlying BFV scheme.

We consider the following components:

- **Packing**: When packing multiple values into a single ciphertext, we construct a plaintext vector $v = (v_0, \dots, v_{n-2}, s)$ where $s = \sum_{i=0}^{n-2} v_i$. The encryption $\mathsf{Enc}(v)$ is indistinguishable from $\mathsf{Enc}(v')$ for any v' of same dimension, as this reduces to IND-CPA security over vector plaintexts in BFV.
- **Auxiliary Slot**: Including a local sum s in the last slot of the plaintext vector does not affect security, since s is deterministically computed from values already encrypted. Its inclusion does not leak new information beyond what is already present in the vector.
- **Insertion**: To insert a plaintext v into a packed ciphertext c, we encrypt v into a ciphertext c_v, construct a plaintext mask m with $m[i] = 1$, and compute $c' = c + \mathsf{EvalMult}(c_v, m)$. The resulting c' remains IND-CPA secure because it is the sum of IND-CPA secure ciphertexts and homomorphic evaluations thereof. The slot mask m is public and fixed, so no leakage is introduced.
- **Deletion**: Similar reasoning applies. The rotated ciphertext and masked-out tail are both outputs of deterministic homomorphic operations on encrypted data, preserving indistinguishability under the semantic security of BFV.
- **Aggregation**: Aggregation over auxiliary slots only involves ciphertext addition. Since BFV supports additive homomorphism, and all ciphertexts involved are IND-CPA secure, the sum remains indistinguishable.

Thus, any distinguishing advantage of $\mathcal{A}_{\text{Hermes}}$ implies an advantage in distinguishing $\mathsf{Enc}(v)$ vs $\mathsf{Enc}(v')$, contradicting the IND-CPA security of BFV.

■

Discussion. An important subtlety in Hermes is the deterministic embedding of auxiliary data—such as local sum—into a fixed slot. While this creates a structural invariant, it does not violate IND-CPA since the data being embedded is functionally dependent on other encrypted slots. From a cryptographic perspective, this corresponds to publishing a function $f(v_0, \ldots, v_{n-2}) = s$ along with the encryption of the vector, which is a standard practice in FHE-based secure computation.

Further, all masking and slot operations are purely homomorphic and do not involve conditional branching or secret-dependent control flow. No part of the computation leaks information via side channels, timing, or slot access patterns. Hermes never reveals plaintexts or intermediate states to the server.

3.3.3 Update Privacy via Slot-Level Obfuscation. To preserve data-obliviousness within packed ciphertexts, Hermes ensures that individual insertions and deletions are indistinguishable with respect to their slot index. Since encrypted updates are performed via uniform homomorphic operations—namely Galois rotations, plaintext masking, and slot-wise additions—the adversary cannot determine which slot was modified given a before-and-after pair of ciphertexts. We formalize this guarantee using a slot-level indistinguishability game.

Definition (Slot Update Indistinguishability Game). Let c be an encrypted packed ciphertext over n slots under BFV. Consider two update positions $i_0, i_1 \in \{0, \ldots, n-2\}$. The goal of the adversary $\mathcal{A}$ is to distinguish whether an insertion was applied at position i_0 or i_1.

The game proceeds as follows:

(1) Challenger generates keypair $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$.
(2) Challenger chooses a plaintext vector $\vec{v} = (v_0, \ldots, v_{n-2})$ and computes $c \leftarrow \mathsf{Enc}_{\mathsf{pk}}(\vec{v})$.
(3) Challenger chooses a random bit $b \in \{0, 1\}$.
(4) Challenger inserts a known value δ at slot i_b using Hermes' homomorphic insert logic, resulting in $c' = \mathsf{Insert}(c, i_b, \delta)$.
(5) Adversary is given (c, c') and outputs a guess $b' \in \{0, 1\}$.

We define the adversary's advantage as:

$$\mathrm{Adv}_{\mathcal{A}}^{\text{slot-ins}} = \left| \Pr[b' = b] - \frac{1}{2} \right|.$$

Hermes guarantees slot-level update privacy if $\mathrm{Adv}_{\mathcal{A}}^{\text{slot-ins}}$ is negligible in the security parameter λ.

Theorem. If the underlying BFV encryption scheme is IND-CPA secure, then Hermes satisfies slot update indistinguishability for any polynomial-time adversary $\mathcal{A}$.

Proof Sketch. We reduce the problem to the IND-CPA security of BFV and the semantic uniformity of the masking and shift primitives. Let m_i be a plaintext mask such that $m_i[j] = \delta$ if $j = i$, and 0 otherwise. The insertion operation computes:

$$c' = \mathsf{EvalAdd}(c_{\text{shifted}}, \mathsf{EvalMult}(c_\delta, m_i)),$$

where c_{shifted} is the result of a homomorphic rotation and masking on c that preserves all but slot i.

Because $c_\delta = \mathsf{Enc}(\delta)$ is fresh and semantically secure under BFV, and m_i is public and independent of δ, the term $\mathsf{EvalMult}(c_\delta, m_i)$ remains indistinguishable regardless of the value of i. The subsequent addition with c_{shifted} is also homomorphic and does not reveal slot index, since the masked slots are zeroed out beforehand. As a result, the output ciphertext c' is statistically indistinguishable whether insertion occurred at i_0 or i_1.

If $\mathcal{A}$ had non-negligible advantage in distinguishing i_0 from i_1, we could construct a distinguisher $\mathcal{B}$ that breaks IND-CPA security by embedding i_b into the encryption pattern. This contradicts our assumption.

■

Discussion. It is important to note that Hermes maintains *position-hiding homomorphism* not by permuting slot values, but by maintaining the slot structure while masking all auxiliary effects of an update. The Galois shift operator is position-invariant modulo a known offset; the zeroing mask ensures no unintentional data leaks from adjacent slots. Since BFV has no noise growth observable in ciphertext structure, and all operations are free of conditional branches, Hermes presents no side-channel or structural leakage to the server.

Furthermore, the reserved auxiliary slot (e.g., local sum) is updated using scalar plaintext operations, and its growth reflects only the net value change, not the specific position of insertion or deletion. This makes Hermes suitable for applications requiring structural privacy in addition to semantic security.

3.3.4 Correctness of Encrypted Updates and Aggregates. We now formally prove that Hermes correctly maintains the semantics of encrypted vector updates and aggregate computations in the presence of insertions and deletions. Our objective is to show that:

(1) The logical payload vector remains consistent with intended insertions and deletions.
(2) The auxiliary sum slot reflects the current sum of all valid payload slots.
(3) The result of encrypted aggregation over multiple ciphertexts equals the true sum of the underlying plaintexts.

We assume a ciphertext $c \leftarrow \mathsf{Encrypt}([v_0, \ldots, v_{n-2}, \sigma])$, where $v_i \in \mathbb{Z}_t$ is the plaintext payload at slot i and $\sigma = \sum_{i=0}^{n-2} v_i$ is the precomputed local sum stored in the auxiliary slot $n-1$.

(1) Correctness of Homomorphic Insertion. Let $v_{\text{new}} \in \mathbb{Z}_t$ be a value inserted into position i (with $0 \le i \le n-2$). Define the updated plaintext vector as:

$$[v'_0, \ldots, v'_{n-2}] = [v_0, \ldots, v_{i-1}, v_{\text{new}}, v_i, \ldots, v_{n-3}]$$

The homomorphic insertion process rotates the suffix $\{v_i, \ldots, v_{n-2}\}$ one position to the right using a Galois automorphism, masks the insertion point i with a one-hot vector $m^{(i)} \in \mathbb{Z}_t^n$ such that $m^{(i)}[i] = 1$ and 0 elsewhere, and adds $v_{\text{new}} \cdot m^{(i)}$ via plaintext multiplication.

Let $c_{\text{ins}} = \mathsf{EvalAdd}(c_{\text{shifted}}, \mathsf{EvalMult}(\mathsf{Encrypt}(v_{\text{new}}), m^{(i)}))$. Then, under correct decryption:

$$\mathsf{Decrypt}(c_{\text{ins}})[j] = \begin{cases} v_j & \text{if } j < i \\ v_{\text{new}} & \text{if } j = i \\ v_{j-1} & \text{if } i < j \le n-2 \\ \sigma + v_{\text{new}} & \text{if } j = n-1 \end{cases}$$

which matches the intended semantics. Note the local sum is homomorphically updated via:

$$\sigma' = \sigma + v_{\text{new}} \in \mathbb{Z}_t$$

via scalar plaintext addition on the last slot.

(2) Correctness of Homomorphic Deletion. Let v_i be deleted from position i. The deletion logic performs a Galois rotation that left-shifts all slots $j > i$, then applies a mask $z \in \mathbb{Z}_t^n$ such that $z[n-1] = 0$ and $z[j] = 1$ for $j < n-1$, to zero out the stale tail. The resulting ciphertext c_{del} decrypts to:

$$\mathsf{Decrypt}(c_{\text{del}})[j] = \begin{cases} v_j & \text{if } j < i \\ v_{j+1} & \text{if } i \le j \le n-3 \\ 0 & \text{if } j = n-2 \\ \sigma - v_i & \text{if } j = n-1 \end{cases}$$

where σ is again updated via scalar subtraction. The vector semantics remain preserved, and auxiliary statistics remain valid.

(3) Correctness of Global Aggregation. Let $c^{(1)}, \dots, c^{(k)}$ be a collection of packed ciphertexts, each of form:

$$c^{(\ell)} = \mathsf{Encrypt}\left(\left[v_0^{(\ell)}, v_1^{(\ell)}, \dots, v_{n-2}^{(\ell)}, \sigma^{(\ell)}\right]\right),$$
$$\sigma^{(\ell)} = \sum_{j=0}^{n-2} v_j^{(\ell)}$$

Then, the encrypted global sum is computed via:

$$c_{\text{agg}} = \mathsf{EvalAdd}(c^{(1)}, \dots, c^{(k)})$$

followed by slot extraction $\mathsf{Extract}(c_{\text{agg}}, n-1)$ to isolate the final aggregated value. The decrypted result satisfies:

$$\mathsf{Decrypt}(c_{\text{agg}})[n-1] = \sum_{\ell=1}^{k} \sigma^{(\ell)} = \sum_{\ell=1}^{k} \sum_{j=0}^{n-2} v_j^{(\ell)}$$

which matches the true plaintext sum across all tuples.

3.3.5 *Leakage Considerations and Mitigation.*

Although Hermes retains the semantic security of the underlying BFV encryption scheme, its use of structured packing and homomorphic updates introduces higher-level leakage channels that must be formally considered. These leakages do not compromise ciphertext contents, but they may reveal information about slot usage, update patterns, or statistical aggregates.

The first class of leakage stems from slot determinism. Each logical record is always stored in a fixed position within the packed vector, so repeated insertions or deletions at the same logical index will affect the same ciphertext slot. An observer with access to the ciphertext stream may correlate operations with slot-specific behavior. This violates the ideal of position-hiding computation and opens the door to side-channel inferences.

A second leakage vector arises from update frequency. If certain ciphertexts are modified more often than others, their noise budget will decay faster, and their ciphertexts may appear structurally different. Even if all contents remain IND-CPA secure, the pattern of modification frequency can reveal nontrivial workload characteristics—e.g., temporal locality or key distribution skews.

A third concern is the auxiliary slot that stores the local sum. In sparse or low-entropy domains, the local sum itself can be highly revealing. For example, a sum of zero in a 4-slot vector may suggest all-zero plaintexts, or, in highly skewed settings, may allow inferences about individual values.

We mitigate these leakages through several strategies. First, randomized slot assignment can be used during insertion to obscure logical-slot correspondence. Rather than inserting always at index i, the system selects a slot $j \in \{0, \dots, n-2\}$ uniformly at random and adjusts bookkeeping accordingly. Second, dummy updates—i.e., adding and removing zero values—can be issued at controlled intervals to equalize apparent activity across ciphertexts. Third, noise can be added to the auxiliary slot at packing time. If each local sum σ_i is perturbed by a small, centered noise r_i, then the final aggregated sum remains correct after canceling or decrypting the total noise $r = \sum_i r_i$.

Importantly, all mitigation strategies are implemented within the homomorphic domain and require no client-side secret state or round-trip interaction. Moreover, they do not reduce correctness or throughput under the vectorized model. These countermeasures demonstrate that Hermes can maintain high performance without sacrificing practical privacy guarantees beyond the IND-CPA baseline.

4 System Implementation

We implement Hermes as a suite of MySQL loadable functions written in C++ using OpenFHE v1.2.4. The system consists of modular components for slot-wise packing, homomorphic insertion and deletion, local-sum-aware aggregation, and slot-selective decryption—all executed natively within the MySQL server runtime.

The codebase comprises 3,518 lines, including 2,045 lines of C++ core logic, 967 lines of shell scripts, and supporting code in Python and CMake. All components are implemented in C++ and compiled into dynamic libraries registered as MySQL UDFs. Our minimalist integration strategy avoids any modification to the MySQL source code or query engine internals.

The full source code is publicly available at:

https://github.com/hpdic/hermes

Experiments and development were partially supported by the NSF-funded Chameleon Cloud infrastructure [**?**].

4.1 Plugin Architecture and Integration with MySQL

Hermes is implemented as a native plugin using the MySQL User-Defined Function (UDF) interface. Each cryptographic operation is compiled as a C++ function into a shared object `.so` file, which is dynamically linked into the MySQL server process at runtime. This mechanism preserves full compatibility with unmodified MySQL

8.x deployments, requiring no server-side modifications or query proxies.

Each plugin function adheres to the MySQL UDF lifecycle. The `_init` function validates argument count and type, and initializes internal state if necessary. The core logic resides in `_func`, which executes per-row during SQL evaluation. The optional `_deinit` function releases any memory or handles. Hermes uses this interface to expose cryptographic primitives including ciphertext packing, secure insertion and deletion, encrypted aggregation, and packed decryption.

To ensure compatibility between SQL types and OpenFHE plaintexts, all inputs are explicitly parsed within the plugin. MySQL UDF arguments arrive as raw C pointers and type tags (e.g., `INT_RESULT`, `STRING_RESULT`). Hermes implements strict parsing logic to convert these inputs into 64-bit signed integers, using `std::istringstream` when necessary. Each scalar value $v \in \mathbb{Z}$ is then embedded into a plaintext vector $\mathbf{m} \in \mathbb{Z}_t^n$, using OpenFHE's `MakePackedPlaintext` API with a fixed slot layout.

Hermes adopts a ring dimension N, yielding $n = N/2$ usable slots in the BFV batching scheme. One slot (typically slot $n-1$) is reserved for auxiliary metadata such as the local sum. All remaining $n-1$ slots are used to store plaintext values. This layout is encoded during packing and preserved during ciphertext mutation. Plugin-side logic ensures that no slot is overwritten inadvertently, and runtime bounds checking is enforced.

The encryption and decryption routines are invoked entirely within the MySQL server process. Ciphertext objects are stored in memory for the duration of the query and are not serialized unless explicitly requested. For diagnostic purposes, Hermes returns string-encoded ciphertext metadata of the form:

```
"0xADDR (v, size=s)"
```

where `ADDR` is the memory address of the ciphertext, $v \in \mathbb{Z}$ is the decrypted payload, and s is the object size in bytes. This format is used for interactive debugging and was critical during early plugin validation.

To integrate OpenFHE v1.2.4, Hermes statically compiles the BFV module and its dependencies into a relocatable shared object. At runtime, shared libraries such as `libOPENFHEcore.so` are copied into MySQL's plugin directory and made visible to `mysqld` via an injected `LD_LIBRARY_PATH` set in a `systemd` override file. This guarantees correct dynamic linking on Ubuntu 24.04 LTS and avoids linker errors from unresolved symbols.

All plugin code is written in C++17, built via CMake with full support for debug symbols and module-wise compilation. To comply with MySQL plugin constraints, all external symbols are wrapped in `extern "C"` linkage and use plain pointer-based memory semantics. No exceptions are allowed across the FFI boundary. Hermes writes all diagnostic logs to `stderr`, which are forwarded into the MySQL error log at `/var/log/mysql/error.log` for offline inspection.

4.2 Context and Key Management in OpenFHE

Hermes relies on the BFV scheme as implemented in OpenFHE v1.2.4. To ensure correct and consistent behavior across all plugin functions, we construct a single shared encryption context that is initialized at plugin load time and reused throughout the query lifecycle. This context is created using OpenFHE's `GenCryptoContext` API, parameterized with a plaintext modulus t, ring dimension N, and multiplicative depth d. In our implementation, we set $t = 65537$, $N = 2^{14}$, and $d = 2$, which balances noise growth and packing efficiency.

The encryption context C defines the underlying polynomial ring $R = \mathbb{Z}_q[x]/(x^N + 1)$, as well as the modulus chain $q = q_1 \cdot q_2 \cdots q_L$ used for leveled homomorphic operations. Once C is created, we enable the following scheme features:

- Public key encryption (PKE) for basic encryption and decryption;
- Leveled SHE for both plaintext-ciphertext and ciphertext-ciphertext arithmetic;
- Advanced SHE for evaluation keys including Galois and relinearization keys.

Key generation is performed once at startup using `KeyGen()`, yielding a public key pk and a secret key sk. We also generate auxiliary keys:

- Relinearization keys `EvalMultKeyGen`(sk) for ciphertext size reduction;
- Rotation keys `EvalSumKeyGen`(sk) and `EvalAtIndex`(sk, i) for slot manipulation.

To prevent accidental key regeneration or state corruption across plugin invocations, all keys and contexts are encapsulated in static C++ global variables guarded by initialization flags. This ensures that the context is instantiated only once, and reused by all subsequent invocations of Hermes UDFs. All keys are stored in memory and are not serialized to disk, thereby avoiding potential I/O or deserialization bugs during runtime.

In addition to functional keys, Hermes supports a default plaintext packing policy where a vector $\mathbf{v} = (v_0, v_1, \ldots, v_{n-2})$ is padded with an auxiliary local sum $v_{n-1} = \sum_{i=0}^{n-2} v_i$. This logic is executed at plaintext construction time and preserved across all homomorphic operations.

By centralizing context and key management, Hermes simplifies plugin logic and ensures cross-function compatibility. All encryption and decryption are performed using the same parameter set, eliminating the risk of parameter mismatch or ciphertext incompatibility. This design is particularly important in the MySQL UDF context, where each query invokes the plugin in isolation and lacks stateful session memory.

4.3 Packed Ciphertext Format and Memory Layout

Hermes adopts a packing model based on the batching technique available in BFV, which maps plaintext vectors into the ring $R_t = \mathbb{Z}_t[x]/(x^N + 1)$ via the Chinese Remainder Theorem (CRT). For a ring dimension N, this results in $n = N/2$ usable plaintext slots, each of which can store an individual integer mod t.

Each ciphertext in Hermes is treated as a structured container with fixed semantic fields:

- Slots $[0, \ldots, n-2]$: store plaintext database records (e.g., salary values).
- Slot $n-1$: reserved for auxiliary metadata (e.g., local sum).

This layout enforces slot-position invariants critical for query semantics. For example, slot i always corresponds to the i-th tuple in a logical group. When a slot is modified, its relative position remains unchanged; thus, operations such as `INSERT` and `DELETE` preserve the integrity of the ciphertext as a vector block.

To ensure predictable memory usage and interoperation with MySQL, each ciphertext object is allocated on the heap and remains opaque to MySQL. Rather than serializing ciphertexts into strings or blobs, Hermes returns a debug string containing:

- The memory address of the ciphertext (as a hexadecimal pointer).
- A plaintext verification value (via local decryption).
- The ciphertext object size (from C++ `sizeof` operator).

This enables validation and tracing without performing I/O. For example, an encrypted salary vector might yield the result:

```
0x7ffee31b2ac0 (sum = 13200, size = 128)
```

Internally, all ciphertexts are represented using OpenFHE's polymorphic type system. The type `Ciphertext<DCRTPoly>` encapsulates a leveled ciphertext vector over RNS-encoded polynomial rings. While OpenFHE supports deep serialization, we deliberately avoid this in Hermes to minimize dependency on fragile runtime I/O code.

Slot-level indexing is consistent across all plugin operations. For example:

- `HERMES_PACK_CONVERT()` maps a plaintext array into slots $[0, \ldots, n{-}2]$ and computes a local sum into slot $n{-}1$.
- `HERMES_INSERT()` and `HERMES_REMOVE()` manipulate only targeted slots while preserving untouched regions.
- `HERMES_SUM()` reads only slot $n{-}1$ across ciphertexts and aggregates via `EvalAdd`.

By adopting this aligned memory and slot layout, Hermes enables relational semantics over encrypted vectors with minimal runtime overhead. Slot position serves as both a logical index and a security-invariant boundary, allowing homomorphic updates to be computed without decoding or leaking positional access.

4.4 Encrypted Aggregate Query Support

Hermes supports encrypted evaluation of SQL-style aggregate queries by leveraging the auxiliary local sum slot embedded in each packed ciphertext. This slot, typically at index $n - 1$, stores the sum of all payload values within the ciphertext. As described in earlier sections, this local sum is precomputed during packing and maintained incrementally across insertions and deletions. The aggregation engine relies on this invariant to compute full-table or group-wise sums without rotating or scanning individual slots.

To compute a global sum over a table, Hermes executes ciphertext-level additions using OpenFHE's `EvalAdd` API. Given a collection of ciphertexts $\{c^{(1)}, c^{(2)}, \ldots, c^{(k)}\}$, each with its local sum $\sigma^{(\ell)}$ stored in slot $n - 1$, the global sum is computed as:

$$\sigma_{\text{global}} = \text{Decrypt}\left(\left[\texttt{EvalAdd}(c^{(1)}, \ldots, c^{(k)})\right][n-1]\right)$$

This implementation is exposed to MySQL via the UDF of global sum `HERMES_PACK_GLOBAL_SUM()`. The function accepts a sequence of ciphertext pointers corresponding to groups (or full-table partitions), adds them homomorphically, and returns the resulting ciphertext encoding the total sum in the last slot. The decryption step is performed separately using `HERMES_DEC_VECTOR_SUMMABLE()`, which extracts the $n - 1$-th slot and converts it into a SQL-visible scalar.

For group-wise aggregation (e.g., `GROUP BY department`), Hermes relies on a grouping UDF `HERMES_PACK_GROUP_SUM()`, which performs group-level ciphertext addition using MySQL's internal grouping operator. In this mode, each group maintains its own packed ciphertexts with preserved slot structure. The UDF dispatch logic ensures that ciphertexts with the same group key are aggregated homomorphically, producing per-group sum ciphertexts with slot $n - 1$ holding the encrypted result.

All aggregation logic is rotation-free. Unlike standard FHE aggregation strategies that require $\log n$ Galois rotations and relinearizations to sum all slots within each ciphertext, Hermes uses only one ciphertext addition per group and one decryption at the end. This leads to significant runtime savings, particularly when $k \gg n$ and group sizes are unbalanced.

Internally, Hermes maintains a fixed slot indexing contract: only slot $n - 1$ may be accessed for aggregation purposes. This enforces consistent semantics and allows plugin logic to treat ciphertexts as opaque blobs outside the final slot. As a result, Hermes's aggregation functions are modular, efficient, and fully SQL-compatible under encrypted execution.

4.5 Homomorphic Update Engine

Hermes supports in-place updates to packed ciphertexts by implementing homomorphic analogues of insertion and deletion. These operations allow the system to modify encrypted vectors directly within the database engine, without requiring decryption or client-side recomputation. The core principle is to simulate array-like slot operations—shifting, overwriting, and nulling—entirely within the ciphertext algebra provided by the BFV scheme.

Insertion is modeled as a three-step procedure. First, the tail of the vector (from the insertion index i to slot $n - 2$) is rotated one position to the right using a Galois automorphism. Second, a new value is encrypted into a single-slot vector and masked into position i using a one-hot plaintext mask. Third, the auxiliary slot is incremented homomorphically by the inserted value, using scalar plaintext addition on slot $n - 1$. All components are assembled using OpenFHE's native operators: `EvalRotate`, `EvalMult` with plaintext masks, and `EvalAddPlain`.

Deletion follows the inverse pattern. The vector suffix $[i{+}1, \ldots, n{-}2]$ is rotated left by one slot, overwriting the deleted position. A plaintext mask then zeros out the final payload slot to maintain the invariant that only $n - 1$ slots are active. The local sum slot is adjusted via plaintext subtraction. Crucially, the deleted value must be known in plaintext at deletion time to update the sum correctly, which is compatible with SQL's explicit deletion semantics (e.g., `WHERE attr = v`).

Both insertion and deletion operations are stateless with respect to the plugin logic. The update functions accept a ciphertext pointer, a plaintext value, and a slot index as input. All intermediate ciphertexts—such as rotated and masked versions—are constructed on the fly and discarded after use. This minimizes memory footprint and avoids cross-query state retention.

The update engine enforces strict bounds checking and slot alignment. No insertion is allowed at slot $n{-}1$, which is reserved for auxiliary metadata. All index operations are validated against the configured slot count during context initialization. This prevents buffer overflows or malformed ciphertexts due to user error or adversarial input.

Unlike vectorized plaintext arrays, encrypted slot operations must preserve both algebraic structure and semantic security. Hermes ensures this by reusing known-safe operations from the BFV scheme and by avoiding conditional branching based on encrypted values. All update paths are data-oblivious at the slot level: the pattern of rotations and additions depends only on public slot indices and user-provided plaintexts, never on encrypted contents.

The homomorphic update engine is implemented entirely within the server-side plugin runtime and does not rely on external key material or interactive round trips. This enables Hermes to support encrypted ingestion, correction, and deletion of data under full automation, paving the way for secure database updates in real-world deployments.

4.6 Decryption Interface and Result Extraction

Hermes performs decryption entirely within the MySQL server process, using the BFV secret key generated at plugin initialization. Since all ciphertexts follow a structured packing layout, the decryption interface focuses on slot-specific result recovery, rather than full plaintext vector traversal. The most common operation is extracting the auxiliary slot $n-1$, which holds the local or aggregated sum over encrypted database values.

The decryption workflow begins with the invocation of a dedicated decryption function, which receives a ciphertext handle as input. The handle is passed as a hexadecimal string from SQL, identifying an in-memory ciphertext stored in the plugin's internal heap. After verifying the validity of the pointer and its alignment with the expected OpenFHE type system, the function decrypts the ciphertext using the server's secret key.

Decryption is implemented using OpenFHE's `Decrypt` API, followed by conversion into a plaintext vector using `SetLength` and `GetPackedValue`. Once the plaintext vector is recovered, only the desired slot is extracted—usually slot $n-1$ for aggregate queries or slot i for targeted inspection. The result is then formatted as a scalar value and returned to MySQL via the UDF return buffer.

All decryption paths are read-only and do not mutate the ciphertext or context state. This ensures that repeated invocations yield identical results and remain free of side effects. To enforce security hygiene, Hermes never exposes the full plaintext vector to the SQL interface; only a single scalar per call is returned. This protects against mass leakage through oversized responses or implicit slot traversal.

The plugin logic also includes bounds checking on the slot index. If a user attempts to extract from an invalid slot (e.g., outside $[0, n-1]$), the decryption function returns `NULL` and emits a warning in the MySQL error log. This design ensures fail-safe behavior in case of malformed input or version mismatch between packing and unpacking logic.

Hermes deliberately avoids automatic decryption of entire ciphertext arrays. For use cases requiring multiple slot values, the system recommends issuing repeated scalar decryptions rather than exposing full vector access. This restriction enforces minimal data release and is consistent with the threat model that permits aggregate leakage but not tuple-level reconstruction.

All decryption functions are implemented as MySQL loadable functions with stateless behavior. They accept no server-side secrets or session data beyond the plugin-managed key context and do not cache intermediate plaintexts. This makes the decryption interface robust against server restarts, concurrent access, and cross-session attacks.

4.7 Implementation Challenges and Leaerned Lessons

MySQL's user-defined function (UDF) interface, while ostensibly C-compatible, imposes a range of low-level constraints that complicate integration with modern C++ libraries like OpenFHE. One prominent example is the UDF return buffer: although MySQL allows `char*` return types, it silently allocates a default buffer of 256 bytes and expects the result to be manually null-terminated and sized via the `length` pointer. If the developer fails to carefully track the length or overflows the buffer, the output is truncated or causes undefined behavior, with no warning from MySQL. To ensure safe operation, Hermes explicitly caps debug string lengths and uses `memcpy` with zero padding.

Another challenge arises from MySQL's strict type expectations during UDF registration and invocation. The `arg_type[]` array must precisely match the declared input behavior, and any mismatch—e.g., expecting `INT_RESULT` but receiving a string-cast input—can result in either incorrect parsing or segmentation faults. This sensitivity is further exacerbated when dealing with numeric values passed as `STRING_RESULT`, requiring explicit string-to-integer conversions inside the plugin. Hermes handles this by explicitly checking and parsing each argument type and issuing warnings for unsupported patterns.

Despite being written in C++, all Hermes UDFs must be registered using `extern "C"` linkage, preventing name mangling and ensuring compatibility with MySQL's dynamic symbol resolution. However, this also restricts the use of modern C++ idioms such as RAII, smart pointers, or class encapsulation for memory management. Instead, plugin state (e.g., ciphertext handles, encryption context) must be tracked manually via global heap allocations or pointer-passing, with careful attention to lifecycle management to avoid memory leaks or dangling references across UDF calls.

Linking OpenFHE into the MySQL runtime poses its own set of challenges. Since MySQL does not inherit environment variables like `LD_LIBRARY_PATH` when started as a systemd service, shared libraries such as `libOPENFHEpke.so` are not found unless explicitly injected. Hermes addresses this by generating a systemd override file to add the plugin directory to the runtime path and restarting the MySQL daemon after deployment. This step is essential but poorly documented, and its omission results in non-descriptive runtime errors about unresolved symbols in the shared object.

A particularly subtle issue involves the generation and use of Galois keys for slot rotation operations. OpenFHE requires that Galois keys be generated explicitly after context and secret key initialization, and failure to do so results in cryptic runtime errors or silent failures during rotation-based operations. Moreover, if multiple plugins generate conflicting Galois keys or overwrite global state, the system may enter a corrupted state, especially if MySQL reuses shared library handles across multiple function invocations. Hermes mitigates this by generating all required keys exactly once at system initialization and disallowing repeated or partial regeneration during runtime.

5 Evaluation

5.1 Experimental Setup

We evaluate Hermes on three public datasets covering health, genomic, and financial records. All experiments are conducted on a standard Linux server with the following configuration:

- **CPU:** Intel Xeon Silver 4310 @ 2.10GHz, 64 cores
- **Memory:** 256GB DDR4
- **OS:** Ubuntu 24.04 LTS
- **MySQL:** v8.0.42
- **OpenFHE:** v1.2.4 with BFV scheme

Hermes uses a default ring dimension of $N = 2^{14}$ and plaintext modulus $t = 2^{16}$, resulting in $n = 8192$ plaintext slots per ciphertext.

We evaluate Hermes on three real-world datasets that represent diverse application domains and data distributions, summarized in Table 1. The **COVID-19** dataset contains 341 daily records of pandemic-related statistics in the U.S., such as hospitalizations and case counts. The **hg38** dataset corresponds to genomic annotations in the human reference genome, consisting of over 34,000 entries such as transcription start sites and exon counts. The **Bitcoin** dataset records cryptocurrency trade volumes on a 3-day interval from 2013 to 2022. Since raw Bitcoin trade values are extremely large, we apply a rescaling factor of 1/24 to obtain minute-level approximations before encryption.

All datasets are stored as single-attribute tables in a MySQL database. Hermes operates over these tables using only SQL-based UDFs, without modifying schema definitions or client-side logic. We use a packing size of 8192 slots throughout all experiments, and all queries are executed on a single-node MySQL server with OpenFHE v1.2.4.

Table 1: Summary of evaluation datasets

Dataset	Domain	#Tuples	Value Range
COVID-19 [?]	Healthcare	341	~[0, 100k]
hg38 [?]	Genomics	34,424	~[0, 10k]
Bitcoin [?]	Finance	1,086	~[0, 2^{31}]

5.2 Encryption Throughput

We begin by evaluating the cost of converting plaintext tuples into homomorphically encrypted representations. This loading step is typically the most expensive preprocessing phase and dominates the storage pipeline in encrypted database systems.

Table 2 compares the runtime of two encryption approaches: (i) *singular encryption*, which encrypts each tuple individually into a one-slot ciphertext, and (ii) *packed encryption*, which converts a batch of plaintexts into a vector ciphertext using our SIMD-style packing API. All evaluations are performed within MySQL using Hermes UDFs over three datasets (cf. Section 5.1). We measure both the total time (in milliseconds) and the per-tuple latency (in microseconds).

Table 2: Encryption throughput comparison across datasets.

Metric	COVID-19	Bitcoin	hg38
Total Tuples	341	1086	34424
Packed Encrypt (ms)	33	28	315
Packed Avg (μs)	96	25	9
Singular Encrypt (ms)	5270	16239	508963
Singular Avg (μs)	15454	14953	14785
Speedup (×)	161×	598×	1643×

Across all datasets, Hermes achieves substantial speedups with packed encryption: **161×** for `COVID-19`, a remarkable **598×** on the lightweight `bitcoin` workload, and an impressive **1643×** speedup on the full-scale `hg38` dataset. These gains arise from amortizing expensive encryption operations across slots, reducing the number of encryption calls, and avoiding repeated memory allocation and ciphertext construction.

We emphasize that this experiment isolates encryption cost from I/O overhead. The 'INSERT INTO' latency is dominated by MySQL's internal transaction logic and is orthogonal to the observed compute-time performance of encryption itself. As such, these results represent the upper bound of Hermes' cryptographic efficiency under practical SQL workflows.

5.3 Insertion Performance

To evaluate insertions, we initialize both singular and packed tables using group 1 of each dataset. We then generate 100 random integers and insert them either individually (singular) or into a specific slot (packed). We measure both the total latency and the per-insert average.

Table 3: Insertion latency comparison for 100 encrypted inserts.

Metric	COVID-19	Bitcoin	hg38
Packed Insert (ms)	644	632	3497
Singular Insert (ms)	2067	2026	2025
Speedup (×)	3.2×	3.2×	0.6×

As shown in Table 3, packed insertion consistently outperforms singular insertion on the `COVID-19` and `bitcoin` datasets, reducing per-insert latency by over 3×. This performance gain is attributed to Hermes' in-place ciphertext editing using `HERMES_PACK_ADD`, which enables localized modification of encrypted vectors without reconstructing new ciphertexts for each tuple.

However, the `hg38` dataset exhibits an inverse pattern where packed insertion is notably slower. This slowdown likely stems from large-slot shifts triggering costly Galois automorphism operations and increased memory pressure from repeated ciphertext evaluations. While correctness is preserved, this highlights a corner case where packed operations may be less efficient than singular encryption, particularly under high-index updates.

5.4 Deletion Performance

We benchmark 100 deletions on both singular and packed tables. Packed deletions use `HERMES_PACK_RMV` to clear a specified slot and shift the last slot forward, while singular deletions remove ciphertexts one by one. Table 4 reports total and per-deletion latency for both approaches across all datasets.

Table 4: Packed vs. Singular deletion latency.

Metric	covid19	bitcoin	hg38
Packed total (ms)	18	29	1590
Singular total (ms)	367	946	602
Speedup (×)	20.4×	32.6×	0.4×

We observe that packed deletion significantly outperforms singular deletion on `covid19` and `bitcoin`, achieving speedups of 20.4× and 32.6×, respectively. These datasets have relatively small ciphertext sizes, so the overhead of packing amortizes well. However, on the `hg38` dataset, the packed deletion is slower than the singular counterpart. This inversion is due to the large slot width and high noise budget of the packed ciphertext, which amplifies the cost of masking and Galois rotation. This result confirms that packed deletion scales less favorably on heavy ciphertexts and motivates future work in adaptive slot reuse and lazy compaction.

5.5 Scalability with Respect to Packing Granularity

We evaluate how the packed group size affects the performance of encryption, insertion, and deletion in Hermes, using the full `hg38` dataset. Table 5 reports the total runtime (in milliseconds) for each operation under six group sizes, ranging from 128 to 4,096 slots.

Table 5: Total latency (ms) for 100 operations on hg38 under varying group sizes.

Group Size	Encrypt	Insert	Delete
128	8,321	3,437	1,386
256	4,365	3,396	1,360
512	3,252	3,452	1,456
1,024	2,289	3,437	1,512
2,048	1,014	3,413	1,509
4,096	625	3,394	1,535

Encryption shows a clear inverse relationship with group size: larger groups yield lower total encryption time due to better amortization across slots. Moving from 128 to 4,096 slots reduces encryption time by over 13×, from 8,321 ms down to 625 ms. This confirms the expected scaling benefit of SIMD-style batching in BFV.

Insertion and deletion times, however, remain relatively flat across group sizes, varying only within a narrow band (approximately 3,400–3,450 ms for inserts and 1,360–1,535 ms for deletes). The lack of strong scaling in update operations is due to the fixed cost of rotation and masking within large ciphertexts. For instance, while packing reduces the number of ciphertexts to manage, larger slot vectors incur higher shift and noise overhead per update.

6 Related Work

6.1 Fully Homomorphic Encryption

Homomorphic encryption (HE) enables computation over encrypted data without decryption. The seminal work by Gentry [?] introduced the first fully homomorphic encryption (FHE) scheme, laying the foundation for later constructions such as BFV [? ?], BGV [?], and CKKS [?]. These schemes have been implemented in libraries including SEAL [?], HElib [?], and OpenFHE [?]. TFHE [?] supports Boolean circuits and has seen recent improvements in bootstrapping performance [? ?].

Among these, CKKS is favored for approximate computation in machine learning [? ? ?], whereas BFV and BGV are suited for exact arithmetic and strong security guarantees [?]. System-level optimizations, such as SIMD-style packing [?] and backend accelerators [? ?], have significantly improved HE performance in practice.

Our work aligns with this direction by leveraging OpenFHE to support slot-based encryption inside a relational database engine. Hermes focuses not on designing new HE schemes, but on integrating homomorphic encoding directly into SQL execution pipelines with minimal system intrusion.

6.2 Encrypted Database Systems

One major class of encrypted databases relies on client-side processing, where queries are executed after downloading encrypted data [?]. While this model minimizes trust assumptions, it sacrifices the benefits of server-side scalability and often incurs high bandwidth costs.

An alternative approach is to split queries between client and server, as seen in information-hiding systems [?], where server-side filters retrieve encrypted tuples guided by secure indices. Systems like CryptDB [?], Arx [?], and Symmetria [?] explore different trade-offs between functionality and leakage, using layered encryption, partitioned encodings, or specialized circuits.

Homomorphic encryption has also been used directly in encrypted database systems, including Symmetria [?], which uses multiplicative HE with additive extensions, and Rache [?], which leverages layout-aware ciphertext placement and SIMD-optimized key switching. These systems often operate on scalar records and leave the ingestion process external to the system.

Hermes differs from prior systems by integrating homomorphic encryption directly into the ingestion path. Unlike approaches that rely on client-side ciphertext generation or immutable payload storage, Hermes exposes SQL-level UDFs for packing, encrypting, and mutating ciphertexts at runtime. Our system introduces a slot-aware data model that supports in-place ciphertext updates and maintains per-group local aggregates as part of the packing process. These capabilities enable Hermes to achieve high ingestion throughput while preserving compatibility with relational operators and encrypted vector semantics.

6.3 Vector Databases and Similarity Search

Recent years have seen rapid development of both native and extended vector database systems. Native systems like Faiss [?], Pinecone, and Chroma [**?**] optimize for pure vector workloads using compact index structures and approximate similarity operators. Mixed workload engines like Weaviate [**?**], Qdrant [**?**], and Vald [**?**] support textual, categorical, and vector fields in tandem.

Some full-stack systems extend relational engines with vector support. Pase [**?**] integrates quantization-aware indexing into PostgreSQL. AnalyticDB-V [**?**] extends Alibaba's analytic engine [**?**] with embedding support. Apache Lucene [**?**], originally a keyword search engine, now includes hierarchical graph indexes for fast k-NN search.

All these systems assume plaintext access to vectors during indexing and query execution. In contrast, Hermes supports encrypted ingestion and runtime packing, enabling similarity-preserving computation without decrypting vectors. Our UDF-based design allows MySQL users to benefit from homomorphic encryption with no query rewriting or external proxy.

7 Conclusion and Future Work

Hermes demonstrates that Fully Homomorphic Encryption (FHE), long regarded as a theoretical milestone, can be practically harnessed for high-throughput, SQL-compatible query processing. By redesigning how encrypted records are packed, updated, and aggregated, we show that multi-slot ciphertexts—augmented with slot-aware mutation and auxiliary metadata—enable expressive relational workloads under provable security guarantees. The system integrates with unmodified MySQL via standard UDFs and achieves performance suitable for real-world ingestion pipelines, without relying on hardware acceleration or cryptographic shortcuts.

This work opens several compelling avenues for future development. At the systems level, integrating encrypted indexing mechanisms—such as order-preserving encodings, hash-based range filters, or Galois-key-aware layout strategies—would unlock efficient filtering and join operations over ciphertexts. At the data model level, generalizing the encoder to support multi-attribute layouts or nested packing schemes could enable vectorized updates across structured relational records. Finally, applying Hermes to non-relational contexts—such as embedded or streaming systems—offers an opportunity to deploy encrypted query support in latency-sensitive, resource-constrained environments.

We view Hermes not as an endpoint, but as a starting point for reconciling two historically separate disciplines: secure computation and relational data management. As both encryption schemes and database engines mature, we believe the design principles behind Hermes—modular encryption, slot-aware semantics, and in-place mutability—will serve as building blocks for a broader class of secure data infrastructures. Making encrypted computation a routine part of database execution is no longer a theoretical aspiration—it is an engineering trajectory now underway.

Acknowledgments

Results presented in this paper were obtained using the Chameleon testbed supported by the National Science Foundation.