HADES: Range-Filtered Private Aggregation on Public Data

Xiaoyuan Liu
UC Berkeley
xiaoyuanliu@berkeley.edu

Ni Trieu
Arizona State University
nitrieu@asu.edu

Trinabh Gupta
University of California Santa Barbara
trinabh@ucsb.edu

Ishtiyaque Ahmad
University of California Santa Cruz
isahmad@ucsc.edu

Dawn Song
UC Berkeley
dawnsong@cs.berkeley.edu

ABSTRACT

In aggregation queries, predicate parameters often reveal user intent. Protecting these parameters is critical for user privacy, regardless of whether the database is public or private. While most existing works focus on private data settings, we address a public data setting where the server has access to the database. Current solutions for this setting either require additional setups (e.g., non-colluding servers, hardware enclaves) or are inefficient for practical workloads. Furthermore, they often do not support range predicates or boolean combinations commonly seen in real-world use cases.

To address these limitations, we built HADES, a fully homomorphic encryption (FHE) based private aggregation system for public data that supports point, range predicates, and boolean combinations. Our one-round HADES protocol efficiently generates predicate indicators by leveraging the plaintext form of public data records. It introduces a novel elementwise-mapping operation and an optimized reduction algorithm, achieving latency efficiency within a limited noise budget. Our highly scalable, multi-threaded implementation improves performance over previous one-round FHE solutions by 204x to 6574x on end-to-end TPC-H queries, reducing aggregation time on 1M records from 15 hours to 38 seconds.

PVLDB Reference Format:
Xiaoyuan Liu, Ni Trieu, Trinabh Gupta, Ishtiyaque Ahmad, and Dawn Song. HADES: Range-Filtered Private Aggregation on Public Data . PVLDB, 18(7): 2159 - 2171, 2025.
doi:10.14778/3734839.3734852

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/camelop/hades-dev.

1 INTRODUCTION

Typically, when a user queries a database, the database is expected to know what the user is asking. Besides, the database may log the query for future analysis. However, privacy-sensitive applications can require stronger protections that allow the server to execute queries without knowing their content. For instance, in the medical field, a doctor might search for patient records based on specific symptoms to determine the best course of treatment. The doctor may want to keep the symptoms hidden from the data service provider to protect patient privacy, especially if a set of symptoms are related to an uncommon disease. Similarly, in the financial sector, an outside investigator may evaluate whether a financial institution's loan approvals are biased toward certain demographic groups. To ensure the integrity of the investigation, specific demographic details and personally identifiable information must not be disclosed to the financial institution.

In existing database solutions, both the infrastructure owner, such as cloud platforms, and the data provider have direct access to user queries. A curious data provider might enable database logging to trace user access, while the infrastructure owner could monitor storage access to database entries. Developing a database that supports privacy-preserving query processing would significantly improve privacy protection in sensitive applications by ensuring that only the user can access the query and its results.

There are existing works approaching this problem with different techniques. Homomorphic Encryption (HE) allows arbitrary computation under encryption. HE-based solutions, such as HEDA [21], HE3DB [4], use ciphertext and homomorphic operations for the complete computation procedure. While providing strong protection under cryptographic assumptions, existing solutions raise performance concerns in practice due to slow homomorphic operations. Multi-Party Computation(MPC)-based solutions [25] are in general faster, but they usually assume non-colluding servers in deployment. There have also been works utilizing secure hardware [26], but those solutions require trust in hardware manufacturers and need extra considerations about memory access pattern leakage.

In this work, we focus on an HE-based solution for simplicity of setup. Specifically, we consider supporting aggregation queries in a public data setting, as aggregation is a common query type that reveals data statistics to support decision-making. In a public data setting, the database content is visible and typically owned by the server, which we refer to as the data provider. Our problem definition can be seen as an extension to the well-known private information retrieval (PIR) [7] problem in two dimensions. First, instead of selecting a single record, we compute the aggregated value from multiple relevant records. Second, instead of accessing values using index or keys (Keyword-PIR [6]), we support various query predicates for data filtering to satisfy various application needs. A similar setting has also been discussed by Hafiz et al.[12]

To enable such private aggregations, we built HADES, a database for efficient private aggregation queries on public data with rich predicate support. The HE-based HADES protocol is single-round, and supports both point and range query predicates, as well as their boolean combinations. During the protocol execution, the

Proceedings of the VLDB Endowment, Vol. 18, No. 7 ISSN 2150-8097.
doi:10.14778/3734839.3734852

query parameters, such as equality-checking targets or ranges to be matched, are completely hidden from the data provider. HADES mainly faces two challenges: the query latency optimization under limited HE noise budget, and parallelization. Existing HE-based solutions [4, 21] process queries completely under ciphertext, and there hasn't been a discussion on how to design and optimize the query protocol in a public data setting. Additionally, existing protocols haven't sufficiently considered design on scalability and optimizations for queries on larger databases.

HADES protocol proposes two major optimizations in algorithm design. First, for point and range queries, to fully utilize the public data setting, we combine PIR-style retrieval and logical circuit to build a novel and flexible building block. The building block efficiently conducts elementwise mapping for public values in the database records while consuming a limited noise budget, accelerating up to 786x compared with a baseline approach. Second, we design a ciphertext aggregation algorithm specifically optimized for larger databases. By optimizing the execution order , the new algorithm reduces computation and communication costs simultaneously and outperforms its baseline by up to 4.5x. Besides the protocol design, the implementation of HADES particularly focuses on parallelization. We design a separate intermediate representation as the execution plan. It sorts out the dependency relations between different stages of parallel tasks before executing the homomorphic operations. As a result, HADES achieves a high thread utilization rate and reduces 761s of single-thread workload to 17s with multi-threading.

Combining the novel protocol designs and efficient implementations, with a clear single-round SQL query interface, HADES achieves up to 6574x latency reduction compared with the best of existing HE-based solutions. Prior to this work, there was no discussion on how to efficiently support private aggregations in public data setting with filtering. HADES is the first work that shows the feasibility of running private aggregations on a million-record database in seconds, rather than hours. Besides, HADES advances the state-of-the-art on extending PIR functionality. It proposes novel building blocks that enable more complicated queries as well as efficient value aggregations.

2 PROBLEM OVERVIEW

2.1 Scenario

To explain the problem, let us follow an example scenario from the standard database benchmark TPC-H [24] (Q6) for supporting business decisions. A **data provider** manages a database that contains detailed information on orders, items, and the supply chain which models real-world business operations. An **analyst** plans to forecast the revenue change after eliminating certain existing product discounts. To explore the change, the analyst writes a SQL query as shown in Figure 1. The query calculates the sum of revenue lost due to discounts for items shipped in 1994, where the discount is between 5% and 7%, and the quantity is less than 24 units.

Private aggregation query protects the privacy of the analyst by hiding the query parameters from the data provider. As shown in the bottom half of Figure 1, the data provider processes the data filtering and aggregation without knowing either the selection of data records or the final aggregation result.

2.2 Query Functionality

For simplicity, we use a SQL interface to express the functionality supported in a private aggregation query. A common private aggregation query involves three clauses: $SELECT$, $FROM$, and $WHERE$. The $SELECT$ clause includes the names of columns wrapped by aggregation operators such as $COUNT$, SUM, and AVG. The $FROM$ clause specifies the table in the database to be accessed.

The key in the private aggregation problem is the support for $WHERE$ clause with encrypted query parameters. The WHERE statement specifies *predicates* which match a subset of the records in the table. Given a data record, the WHERE statement can be seen as a boolean function that outputs true for matched records and false for unmatched ones. Some predicates contain values that affect the outcome of the boolean function. We refer to these values as *predicate parameters*. For example, the ship date attribute "1994-01-01" and the numbers 0.05, 0.07, 24 in the query are the parameters that need to be hidden.

Based on the filtering operation type, predicates can be grouped into point predicates, range predicates, and combined predicates. A point predicate checks whether a specific column in a table matches a particular value, facilitating an equality check on a single point. A range predicate, on the other hand, tests whether the column in a table falls within a specific continuous range of values. The combined predicates apply logical operators, such as AND, OR, NOT, to filter the records based on the result of other predicates. The example query in Figure 1 can be seen as a combined predicate applied to multiple range predicates.

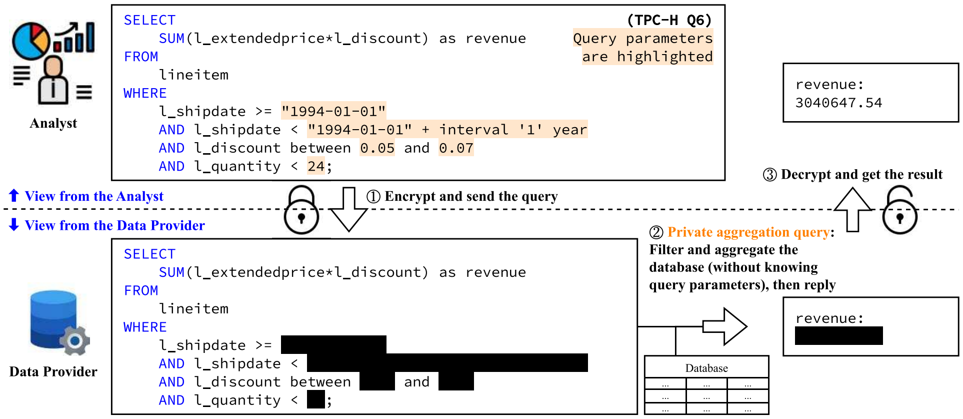
There are also other commonly seen query keywords such as $GROUP\ BY$ and $ORDER\ BY$ that also extend the functionality of the private aggregation query. We discuss these operators in Section 6.

2.3 System Design Goals

Considering functionality and real-world deployment requirements, here we summarize our system design goals.

Privacy. The system should protect the privacy of the analyst by hiding all the information related to the query parameters. Formally, consider a data provider who chooses two queries with differences only in their query parameters. When provided with the original hidden version of one of these two queries, the data provider should not be able to determine which hidden query corresponds to which original query, with a negligible probability. This guarantee ensures no information leakage on the query parameters. In terms of the **threat model**, we assume a strong malicious adversary who may arbitrarily compromise the data provider or the network. The adversary may have access to network packets, or the requests received and the responses sent by the data provider. We assume that the adversary cannot compromise standard cryptographic primitives, and the adversary does not compromise the device of the analyst and cannot access its storage content, such as secret keys.

SQL Support and Interoperability. Our goal is to build a private query system that is simple to use and understand. A key idea to improve interoperability and reduce the learning effort is to make it similar to existing relational database management systems (RDBMS). Supporting commonly used SQL interfaces and data formats facilitates standardized use and testing.


(TPC-H Q6)
Query parameters are highlighted
Analyst
revenue: 3040647.54
⬆ View from the Analyst
① Encrypt and send the query
③ Decrypt and get the result
⬇ View from the Data Provider
② Private aggregation query: Filter and aggregate the database (without knowing query parameters), then reply
Data Provider
Database
revenue: [redacted]

```
SELECT
    SUM(l_extendedprice*l_discount) as revenue
FROM
    lineitem
WHERE
    l_shipdate >= "1994-01-01"
    AND l_shipdate < "1994-01-01" + interval '1' year
    AND l_discount between 0.05 and 0.07
    AND l_quantity < 24;
```

```
SELECT
    SUM(l_extendedprice*l_discount) as revenue
FROM
    lineitem
WHERE
    l_shipdate >= [redacted]
    AND l_shipdate < [redacted]
    AND l_discount between [redacted] and [redacted]
    AND l_quantity < [redacted];
```

Figure 1: Example of private aggregation query. The top half displays the view from the analyst, while the bottom half displays the view from the data provider. Query parameters, intermediate values, and final results are all hidden from the data provider.

One-round protocol with two parties. We want to preserve a similar query pattern as a regular non-private database, where in a single round an analyst sends a query to the database, and the database sends the results of the query after processing it. That is, we do not want more than one round of interaction. We also do not want to introduce additional parties on the database side, for example, multiple databases that are assumed not to collude.

Latency efficiency. We want the system to provide a user experience similar to a non-private DBMS for interactive data analysis with a query latency within a minute, ideally seconds. As the privacy requirement indicates that all data records need to be accessed equally to guarantee indistinguishability, the protocol operations should be highly data-parallelized to be efficient.

Scalability. Real-world aggregation often involves a large number of records. The default-scale TPC-H benchmark contains millions of records in its "lineitem" table. The protocol should support scalable design, that by providing more computation resources, the system should increase its processing speed and handle a larger workload.

2.4 Challenges

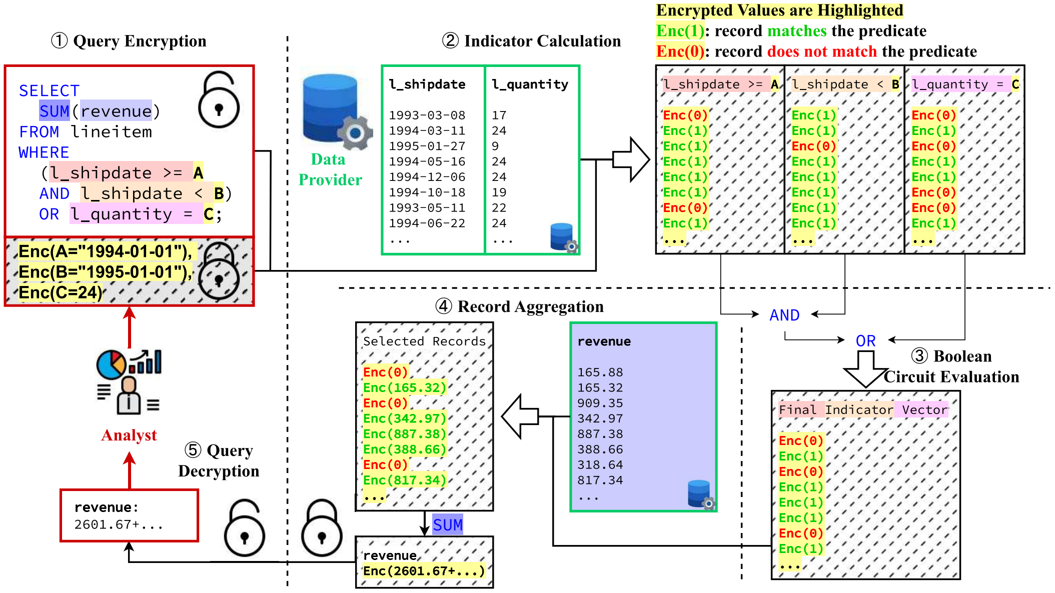
With the design goals, we revisit why existing works failed to meet our requirements and list out the potential challenges. First, a straightforward method is to run the complete query under HE. Although HEDA [21] and HE3DB [4] explored efficient FHE protocol designs for query processing, they assume both encrypted query and database and do not discuss optimizations when the database is non-private. As a result, their solutions require more than 15 hours to process one million records and cannot match practical efficiency needs in this setting. Second, existing MPC-based solutions [16, 25] require more than two parties. Last, previous works such as keyword-PIR [6] and recent works in VLDB such as Pantheon [1] focus on improving PIR with specific types of queries, but still have limited functionalities (e.g. only equality check) and does not support richer predicates. We provide more comparison details in Section 8.

To meet the design goals in Section 2.3, HADES employs HE to provide strong privacy protection through a new two-party one-round aggregation protocol. At its core, HADES borrows an indexing design from PIR and builds efficient homomorphic operators between encrypted query and public dataset for SQL operations. In this way, HADES avoids the high-depth homomorphic circuits which harm efficiency and scalability, and achieves practical performance on a larger workload.

3 HADES WORKFLOW

HADES leverages HE to enable a simple one-round private query processing. Specifically, the analyst handles the encryption of the query and the decryption of the result. The data provider takes two inputs: its own database in cleartext and the encrypted analyst query, and outputs the encrypted aggregation result. Figure 2 illustrates the HADES workflow that executes a five-stage protocol. We provide more details on how each operation is done under encryption in Section 4.

(1) **Query encryption.** After deciding on a specific query to run, the analyst first encrypts all query parameters into a ciphertext, then sends that ciphertext along with the query template (query without parameters) to the data provider.
(2) **Indicator calculation.** The data provider starts by computing encrypted values that indicate which records in the database match the specific point or range predicates. Specifically, it generates encryption of 1's for selected records and encryption of 0's for unselected records for each point or range predicate. The resulting encrypted indicator vectors are inputs for the next stage.
(3) **Boolean circuit evaluation.** Following the boolean operations specified in the query template, which forms a tree-structured circuit, the data provider merges the encrypted indicator vectors with logical operations and generates a final indicator vector with 1's and 0's that represents the final predicate matching result.
(4) **Record aggregation.** In the aggregation stage, the protocol conducts multiplications between the indicator vectors and the target table columns. While the unselected records are masked out by the encryption of 0's, the selected records are


① Query Encryption
② Indicator Calculation
Encrypted Values are Highlighted
Enc(1): record matches the predicate
Enc(0): record does not match the predicate
Data Provider
AND
OR
③ Boolean Circuit Evaluation
④ Record Aggregation
SUM
Analyst
⑤ Query Decryption

```
SELECT
  SUM(revenue)
FROM lineitem
WHERE
  (l_shipdate >= A
  AND l_shipdate < B)
  OR l_quantity = C;
```

Enc(A="1994-01-01"),
Enc(B="1995-01-01"),
Enc(C=24)

l_shipdate	l_quantity
1993-03-08	17
1994-03-11	24
1995-01-27	9
1994-05-16	24
1994-12-06	24
1994-10-18	19
1993-05-11	22
1994-06-22	24
...	...

l_shipdate >= A	l_shipdate < B	l_quantity = C
Enc(0)	Enc(1)	Enc(0)
Enc(1)	Enc(1)	Enc(1)
Enc(1)	Enc(0)	Enc(0)
Enc(1)	Enc(1)	Enc(1)
Enc(1)	Enc(1)	Enc(1)
Enc(1)	Enc(1)	Enc(0)
Enc(0)	Enc(1)	Enc(0)
Enc(1)	Enc(1)	Enc(1)
...	...	...

Final Indicator Vector
Enc(0)
Enc(1)
Enc(0)
Enc(1)
Enc(1)
Enc(1)
Enc(0)
Enc(1)
...

revenue
165.88
165.32
909.35
342.97
887.38
388.66
318.64
817.34
...

Selected Records
Enc(0)
Enc(165.32)
Enc(0)
Enc(342.97)
Enc(887.38)
Enc(388.66)
Enc(0)
Enc(817.34)
...

revenue
Enc(2601.67+...)

revenue:
2601.67+...

Figure 2: HADES workflow (details simplified). Text on a black background represents encrypted values hidden from the data provider. During indicator calculations, the protocol outputs encrypted 1 for selected records and 0 for unselected.

preserved and carried into the ciphertext. Then the protocol aggregates all ciphertexts to compute the final result.

(5) **Query decryption.** The analyst decrypts the response and gets the final result. Because the data provider does not have the corresponding encryption key, it cannot learn any information related to the query parameter.

The use of encrypted indicator vectors in HADES resembles that of existing PIR solutions, such as XPIR [19], that select one record from a set of records. Additionally, to support the combined predicates, HADES introduces HE operations between predicates to compute the logical combinations. The key techniques of HADES are in its protocol design to (1) efficiently generate encrypted indicator vectors for point and range predicates during the indicator calculation stage (stage 2), and (2) efficiently aggregate encrypted records in the record aggregation stage (stage 4). We discuss details about these two protocols in Section 4.

4 HADES BASE PROTOCOL

In this section, we describe the basic HADES protocol to support private aggregation query, which serves as a strawman to provide required functionalities but may not be performant. We explain how it can be further optimized for practical efficiency in Section 5.

We start with the necessary background in HE and the use of indicators. Then we explain the basic protocol of indicator generations for point and range predicates. Last, we cover the aggregation.

4.1 Background

4.1.1 BFV Homomorphic Encryption. Homomorphic encryption (HE) is the major cryptographic tool used to process private queries in HADES. The complicated query functionalities call for Fully Homomorphic Encryption (FHE) that supports both addition and multiplication operations under encryption. Among existing FHE schemes, the BFV scheme [3, 10] provides the required functionality based on the hardness of Ring Learning with Errors (RLWE), while being more efficient than most traditional number-theoretic schemes. Specifically, we use the SEAL variant [15] of BFV for implementation. Additionally, we leverage the CRT batching technique [22] that packs a matrix of integers into a single ciphertext. This approach enables efficient processing of large databases in a SIMD manner, significantly improving per-record processing speed compared to non-batched methods.

The basic BFV scheme takes polynomials of limited degrees as its plaintext space. Since all the polynomials in our protocol use the encoding of CRT Batching, we assume all data takes the form of vectors during the homomorphic operations to simplify the discussion. Under this view, we organize the definition of the BFV scheme as follows.

Let λ be the security parameter and let n be the batching size of the vector. The BFV scheme includes the following algorithms:

(1) **KeyGen**$(\lambda) \to (pk(pk + ek + rk), sk)$: takes the security parameter λ and generates a public key pk, a secret key sk, evaluation keys ek for relinearizations, and Galois rotation keys rk. For simplicity, we use pk to refer to the combination of (pk, ek, rk) from now on.
(2) **Encrypt**$(pk, m) \to ct$: takes the security parameter λ and a plaintext m representing a vector of size n, computes the ciphertext ct. For simplicity, from now on we denote "**Encrypt**$(pk, [...])$" as "**Enc**$([...])$", where "$[...]$" is the plaintext vector.
(3) **Decrypt**$(sk, ct) \to m$: takes the secret key sk and the ciphertext ct, and recovers the plaintext m.
(4) **Negate**$(pk, ct_0) \to ct_1$: takes a ciphertext ct_0 and returns ct_1 that satisfies $\text{Decrypt}(sk, ct_1) = -\text{Decrypt}(sk, ct_0)$.
(5) **AddPlain**$(pk, ct_0, m_1) \to ct_2$: takes a ciphertext ct_0 and a vector m_1 of size n. $\text{Decrypt}(sk, ct_2) = \text{Decrypt}(sk, ct_0) + m_1$.

(6) **Add**$(pk, ct_0, ct_1) \rightarrow ct_2$: takes two ciphertexts and calculates the sum ct_2. Decrypt(sk, ct_2) = Decrypt(sk, ct_0) + Decrypt(sk, ct_1).

(7) **MultiplyPlain**$(pk, ct_0, m_1) \rightarrow ct_2$: takes a ciphertext ct_0 and a vector m_1 of size n, calculates the elementwise product ct_2 that Decrypt(sk, ct_2) = Decrypt$(sk, ct_0) \odot m_1$.

(8) **Multiply**$(pk(+ek), ct_0, ct_1) \rightarrow ct_2$: takes two ciphertexts and calculates the elementwise product ct_2 with Decrypt(sk, ct_2) = Decrypt$(sk, ct_0) \odot$ Decrypt(sk, ct_1). Evaluation keys are used to relinearize the output and maintain the same format as the input ciphertexts.

(9) **Rotate**$(pk(+rk), ct_0, s) \rightarrow ct_1$ uses Galois keys to cyclically rotate the vector encrypted in the input by an offset s, i.e., the $((k+s) \bmod n)$'th element in the vector encrypted in input ciphertext ct_0 becomes the k'th element in the new vector encrypted by ct_1. For simplicity, from now on we denote "**Rotate**$(pk, ...)$" as "**Rot**$(...)$".

In BFV, *noise* is deliberately added during encryption to secure the scheme. This initial noise ensures that the ciphertext masks the plaintext effectively. However, most HE operations—such as addition or multiplication—on the ciphertext increase the noise. Over time, the accumulated noise can exceed a critical threshold, making decryption impossible. The noise budget represents the remaining capacity before this threshold is reached, indicating how many more computations can be safely performed on the encrypted data without harming correctness. Designing efficient HE protocols with a limited noise budget poses additional challenges.

4.1.2 Encrypted Indicator. HADES takes an approach similar to PIR which generates indicators representing whether the record is selected. Here we provide an example to explain the basic idea of PIR. Assume a simple case that the data provider has 8 records $v_0, v_1, ..., v_7$ and assume a batching vector size $n = 1$ to discuss a non-batched version. To retrieve v_5, the analyst sends 8 ciphertexts $Q = [\textbf{Enc}([0]), \textbf{Enc}([0]), \textbf{Enc}([0]), \textbf{Enc}([0]), \textbf{Enc}([0]), \textbf{Enc}([1]), \textbf{Enc}([0]), \textbf{Enc}([0])]$. To get the encryption of the target record, the data provider calculates the sum $\sum_{i=0}^{7} Q[i] * v_i$ which equals to $\textbf{Enc}([v_5])$ and returns it to the analyst. After decryption, the analyst learns the value it requested.

To see how batching works in a SIMD manner, assume a batching vector of size $n = 4$, by chunking the database into vectors, we can reorganize the above procedure. The analyst sends $Q = [\textbf{Enc}([0, 0, 0, 0]), \textbf{Enc}([0, 1, 0, 0])]$ and the data providers calculates $Q[0]*[v_0, v_1, v_2, v_3]+Q[1]*[v_4, v_5, v_6, v_7]$ with **MultiplyPlain**. The vectors "[0, 0, 0, 0]", "[0, 1, 0, 0]" represent the selection of relevant records, which we address as **indicator vectors**. The encryption of such vectors are further denoted as **encrypted indicator vectors**.

To extend PIR to aggregation query processing, we need to (1) extend indexing to filtering, and (2) add an aggregation step. However, such extension brings new challenges. Different from indexing, filtering requires designs to process point and range predicates, and boolean combinations between multiple predicates. Unlike PIR, aggregation requires addition between records under encryption, which may result in potential value overflow. The small field size used by HE makes overflow prevention even more challenging. The rest of the section explains our design to tackle these challenges.

4.1.3 Logical Operations. As mentioned in Figure 2, in stage three we process boolean combinations on encrypted indicator vectors. In practice, we use arithmetic operations to express boolean operations, specifically, for ciphertext a and b:

(1) **NOT**$(a) = 1 - a$

(2) **AND**$(a, b) = a * b$ where "$*$" uses ciphertext **Multiply**.

(3) **OR**(a, b) = **NOT**(**AND**(**NOT**(a), **NOT**(b))) $= a + b - a * b$, where "+", "−", and "$*$" are all homomorphic operations.

Note that, in the "OR" operation, if we know in advance that both terms will not be true simultaneously, we can omit the term “a * b” ($OR(a, b) = a + b$). This normally happens when processing predicates like “x = 42 OR x > 42”, “x < 16 OR x > 32”. In practice, this helps improve query efficiency and save noise budget.

4.2 Supporting Point Predicates

Point predicate checks the equality between records and a certain value. In general, it takes the form $WHERE\ col_x = v$. For a database column col_x, depending on its data type, the point query difficulty varies. Here we first discuss the simple case, where col_x is an 8-bit unsigned integer. Specifically, we discuss two general types of methods as baselines. As we will see later, to efficiently process more bits, we need to combine the core idea of these two methods.

Protocol 1: Batched 8-bit Point Query

Assume a batching size $n = 2^8 = 256$ to be the same as the size of the range of the value, and a database with $N = n = 256$ records. For query $WHERE\ col_x = v$:

- The analyst sends $mapping$ = **Enc**(m), where m is a vector of size 256 with $m[v] = 1$, $m[i] = 0 \forall i \in [0, 256) \setminus \{v\}$.
- The data provider initiates a full-zero encrypted vector of size 256 $result = [0, 0, \dots, 0]$ and applies the following procedure to **each** record in the database, for l'th record with value r:
  (1) Rotate the required indicator to the first slot in the vector according to the record value r:
  $$current \leftarrow \textbf{Rot}(mapping, r)$$
  (2) Use multiplication to mask out irrelevant indicators and only keep the first slot:
  $$current \leftarrow current * [1, 0, 0, \dots]$$
  (3) Rotate the encrypted indicator of the record back to its original location l in the database
  $$current \leftarrow \textbf{Rot}(current, -l)$$
  (4) Add the indicator to the result ciphertext
  $$result \leftarrow result + current$$

The result satisfies $result = \textbf{Enc}(ind)$, where $ind[l] = m[r]$ for all records, i.e., $ind[l] = 1$ iff $r = v$, o/w $ind[l] = 0$.

Method A: PIR-style retrieval. The basic version of PIR can be directly used when processing equality checks in a small range (e.g. 8-bit unsigned integers). The key idea is to consider values in the database as indexes on encrypted 1s' and 0s'. For simplicity, we first consider a non-batched version (i.e. $n = 1$). Specifically, instead of directly encrypting the query parameter, the analyst prepares a “mapping” vector of size equal to the range. For 8-bit unsigned integers, the analyst prepares a vector of size 256. All values in the mapping are zero except for the slot with an offset equals to the query parameter, which is set to one. The analyst encrypts this mapping and sends it to the data provider (i.e. for a point query $col_x = v$, the analyst prepares $mapping[v]$ = **Enc**$([1])$, $mapping[i]$ = **Enc**$([0]) \forall i \in [0, 2^8 = 256) \setminus \{v\}$). The data provider, upon receiving this mapping, picks the corresponding ciphertext according to the values in its database. Specifically, for a database with a list of values $d_0, d_1, d_2, ...$, the data provider generates $mapping[d_0], mapping[d_1], mapping[d_2], ...$, which is the required encrypted indicator vector.

Extending this simple method to a batched version requires homomorphic rotation operations. Protocol 1 provides an overview of the procedure. Compared with the non-batched version, because the indicators are batched together, it takes the data provider more steps to index them. The data provider needs to first rotate the input ciphertext, then apply masking via multiplications to remove irrelevant indicators, rotate the ciphertext back so that the indicator appears in the correct location[1], and add all collected indicators up to form the result indicator also in the batched form.

Method B: Bitwise circuit. Instead of checking equality by "indexing" or "mapping", an alternative method is to check each of the 8 bits then merge the result. Specifically, for checking where $col_x = v$ without batching, we can write v in the binary representation as $v = v_0 * 2^0 + v_1 * 2^1 + ... + v_7 * 2^7$, where $v_0..v_7$ are binary values. Then for each bit, we conduct bitwise checking $EQ(a, b) = 1+2*a*b-a-b$, then combine the result with the boolean "AND" operation.

Compared with method A, method B is easier to batch and requires fewer encrypted values for the input. However, it requires more multiplication operations which are linear to the bit width of the column, resulting in huge HE noise growth.

Combined: Point query on more bits. Real-world queries often involve point queries on data types with more bits. However, both methods mentioned above become impractical when there are more bits to be checked. For example, for a 64-bit equality check, For method A, the client needs to send 2^{64} encrypted values to represent the mapping, which is communication inefficient. For method B, the server needs to process 6 layers of multiplications, which leaves few bits of noise budget for other predicates and other query stages.

HADES protocol combines two methods to form a practical solution. The idea is to conduct multiple 8-bit equality checks in method A and use "AND" operations to combine them similar to method B. For example, for a 32-bit value $|ABCD|$ representing the value $A * 2^{8*3} + B * 2^{8*2} + C * 2^{8*1} + D * 2^{8*0}$ where A, B, C, D are all 8-bit values, and similarly another value $|EFGH|$. To verify $|ABCD| = |EFGH|$, it is equivalent to check $(A = E)$ **AND** $(B = F)$ **AND** $(C = G)$ **AND** $(D = H)$. In this way, we break one 32-bit equality check into four 8-bit equality checks and three AND operations. We solve both problems: (1) As only four mappings of length 2^8 are required, the communication is still efficient. (2) As only three AND operations are involved (two layers of multiplications), the noise budget is still sufficient. Similarly, we can support equality checks for 16 bits, 64 bits, and more. Note that we select the 8-bit equality check as it works well in most cases under our default HE parameters. In practice, one might select a bit width to achieve the best trade-offs between latency and noise growth.

4.3 Supporting Range Predicates

Range predicate checks if the record is within a given range, which in general takes the form of $WHERE\ col_x > v_l$, $WHERE\ col_x >= v_l$, $WHERE\ col_x < v_r$, or $WHERE\ col_x <= v_r$. The basic protocol for an 8-bit range query is similar to the one for a point query. The only difference is in the number of encrypted 1s' used in the query ciphertext. Again, we first explain how to support 8-bit range queries, and then we discuss its extension to wider ranges.

8-bit range query. To support 8-bit range query, we can still use the indexing-based protocol. To cover a range instead of a single point, we set all values that match the predicate in the mapping vector to be 1s'. Consider $WHERE\ col_x > v_l$ as an example, where $v_l \in [0, 255)$. The only change we need to make to Protocol 1 is for analyst to define m as $m[i] = 1 \forall i \in (v_l, 256)$, o/w $m[i] = 0$. Then with the same protocol, the data provider can calculate the indicator for the specified range predicate. For l'th record of value r in the database, the protocol returns $result = \textbf{Enc}(ind)$, where $ind[l] = m[r]$. $ind[l] = 1$ iff $r \in (v_l, 256)$, as desired.

Range query on more bits. Supporting range queries on more bits is similar to supporting point queries for more bits. The key idea is to combine 8-bit building blocks using boolean operations to construct equivalence. However, different from more-bit point queries which only uses 8-bit point queries, more-bit range queries not only use 8-bit range queries but also 8-bit point queries. Concretely, consider 16-bit comparison between the value $|AB|$ where both A and B are 8-bit values and $|AB| = A * 2^{8*1} + B * 2^{8*0}$, and similarly value $|CD|$, we have $|AB| < |CD| \Leftrightarrow A < C$ **OR** $(A = C$ **AND** $B < D)$. It would require two 8-bit range queries, one 8-bit point query, one "OR" operation, and one "AND" operation. Similarly, one can construct range queries on more bits. The conversion takes multiple steps and in each step, an 8-bit range query is made on more significant bits, then a point query to cover the other case.

Because all comparisons are made under encryption, boolean short-circuiting cannot be applied here to save operations. However, as discussed in Section 4.1.3, we can apply the optimized *OR* operation as we know in advance $A < C$ and $A = C$ cannot hold simultaneously, even if we don't know the query parameter value.

4.4 Supporting Aggregation

After indicator calculation with methods mentioned in Section 4.2 & 4.3, and boolean circuit evaluation using boolean operators described in Section 4.1.3, in the record aggregation stage, the data provider takes the input of the database and the final encrypted indicator vector that represent which database records are selected, and outputs the aggregated values according to the query. Here we discuss two types of queries: "COUNT" and "SUM". While HADES focuses on aggregation queries, the "SUM" aggregation can also be used for rich-predicate retrievals when selecting a single record.

Note that the aggregation under encryption presents a challenge due to the field used for homomorphic operations. With CRT-batching, each slot in the encrypted vector can only preserve a limited number of bits. In practice, for an encrypted message $\textbf{Enc}(m)$ used in HADES with our default parameter, for $n = 2^{14}$, we have $0 \leq m[i] < 16384$ where $i \in [0, 2^{14})$. This suggests each slot can hold 17 bits of information. While in the earlier stages it is fine to have a smaller field as indicators are only 0s' and 1s', in the aggregation stage, we introduce new protocol designs to overcome the limitations in the aggregated data type.

COUNT query. In a COUNT query, the data provider adds up the encrypted indicators to see how many values are selected. For

[1] A simple optimization is to construct different masks to select slots and only rotate once by location offset. We discuss an optimization covering this idea in Section 5.

batched indicators, the goal is to merge the indicators into a single value in one slot. Such merging is important when the database is large and multiple encrypted indicator vectors for different chunks need to be merged before being sent back. To support this operation, we use rotation operations along with multiplications for masking. Consider an example with batch size $n = 8$, for an encrypted indicator vector $I = \mathbf{Enc}(ind)$ with value $ind = [1\ 0\ 0\ 1\ 0\ 0\ 1\ 0]$, the data provider calculates $result = (I + \mathbf{Rot}(I, 1) + \mathbf{Rot}(I, 2) + \mathbf{Rot}(I, 3) + \mathbf{Rot}(I, 4) + \mathbf{Rot}(I, 5) + \mathbf{Rot}(I, 6) + \mathbf{Rot}(I, 7)) * [1\ 0\ 0\ 0\ 0\ 0\ 0\ 0]$ and gets the desired merged result with value $[3\ 0\ 0\ 0\ 0\ 0\ 0\ 0]$.

Then, for multiple indicator ciphertexts computed from different chunks of the database, the data provider merges them into a single ciphertext before returning to the analyst. A naive approach is to directly add up the ciphertext. However, this causes potential overflow when there are many chunks. Concretely: (1) It is fine to add up values in the same ciphertext in one slot because we have 2^{14} binary values in one ciphertext, and the size of the field is larger than 2^{17}, $2^{14} < 2^{17}$. (2) It is fine to add up 2^3 aggregated ciphertexts as $2^{14} * 2^3 \leq 2^{17}$, and the field size is larger than 2^{17}. However, when there are more than 2^3 chunks, we need overflow prevention. To solve this, in HADES, instead of directly adding up the chunk results in the same slot, we use rotations and additions to put them in different slots to make use of all 2^{14} available slots. Then the analyst can decrypt the returned ciphertext and add it up in cleartext to get the final sum. In this way, we can support COUNT for up to 2^{31} records within one ciphertext.

SUM query. To sum the selected values for the given list of records in the database, the simplest solution is to run plaintext multiplications between the plaintext database and the ciphertext encrypted indicator vector, the same as in PIR, then add up inside each ciphertext. For example, for the database [11 22 33 44 55 66 77 88] and the encrypted indicator [1 0 0 1 0 0 1 0], we first run plaintext multiplication, then similar to the COUNT query, we use rotations to merge the results into a single slot to get [132 0 0 0 0 0 0 0]. However, different from the COUNT which adds up binary indicator values, the SUM query needs to consider the database column being processed. For example, the decimal in TPC-H takes 42 bits while our HE scheme can only handle 17 bits in one ciphertext slot. If we use the COUNT algorithm, there will be overflows that make the final result unrecoverable.

To address this challenge, HADES decomposes the 42-bit database values into its octal representation, handling 3 bits each time. Since $2^3 * 2^{14} \leq 2^{17}$, we can guarantee no overflow when aggregating within a single ciphertext. To recover the original sum, HADES runs multiple octal SUM queries simultaneously and marks the order of the results in the returned ciphertext. The analyst can recover the original sum by treating the results as the octal decomposition. To explain with a concrete example summing up three 8-bit values $42_{10} = 052_8$, $151_{10} = 227_8$, $19_{10} = 023_8$, HADES breaks the record into 3 pieces, then run 3 SUM queries separately and mark the result order. The data provider calculates $0 + 2 + 0 = 2, 5 + 2 + 2 = 9, 2 + 7 + 3 = 12$ under encryption, then the analyst recovers the sum $2 * 8^2 + 9 * 8^1 + 12 * 8^0 \longrightarrow 212$, which equals $42 + 151 + 19$. The protocol guarantees no overflow and thus the aggregation results are always recoverable.

5 HADES OPTIMIZATIONS

While Section 4 explains the baseline protocol that supports private aggregation queries, latency efficiency still requires careful algorithm design. In this section, we explain two major optimized algorithms to accelerate the two most time-consuming stages in HADES: indicator calculation and record aggregation.

5.1 Batched Elementwise Mapping

In Section 4, we explained how we can use PIR-style retrieval for 8-bit predicates. Here we extend this building block to have a more general functionality. Specifically, assuming batched ciphertext with $n = 2^8$, consider a list of 8-bit record values from the data provider: $db = [r_0, r_1, ...]$, and consider a mapping of size 256 sent by the analyst with 8-bit keys: $mapping = Enc(v)$, where $v = [v_0, v_1, ...]$. Following the same procedure in Protocol 1, the data provider can compute $result = Enc([v[r_0], v[r_1], ...])$, where $v[r_i]$ is the r_i'th value in the mapping being encrypted. The functionality of the building block can be formalized as below.

- **MappingGen**$(pk, f) \rightarrow ct_f$: takes a mapping f from domain $[0, 2^d)$ to any integers, generates a ciphertext ct_f that stores the mapping. We need $M = 2^d$ slots in the ciphertext to store the mapping. If $2^d \leq n$, where n is the size of the vector, the mapping only occupies 2^d slots in all n slots inside ct_f, otherwise, the operation generates a list of ciphertexts to store a single mapping.
- **ApplyElementwiseMapping**$(pk, m, ct_f, d) \rightarrow ct$: takes the domain size bit-width d, an encrypted mapping ct_f, and a plaintext vector m with all elements in domain $[0, 2^d)$, applies elementwise mapping for all elements in m to compute an encrypted ct satisfying that $\forall k : \text{Decrypt}(sk, ct)[k] = f(m[k])$. For simplicity, we omit the pk argument and denote the function as "$\mathbf{Emap}([...])$".

Following Protocol 1, for a database with $N = 2^{14}$ records, the required amount of multiplications and rotations needed for **Emap** is huge, resulting in slow computation. Specifically, it requires 2^{14} multiplications, 2^{14} rotations, and $2^{14} - 1$ additions after applying the location offset-based rotation. Because HE operations are usually time-consuming, making the solution practical requires avoiding an operation number that is linear to the database size.

Rotation caching. Observe that all the rotations are applied to the masked mapping, because there are only M slots used in the mapping, there are only M different possible rotations rather than the batch size n. If we can preprocess to cache these different rotations and reuse them for different database records, we can reduce the required rotations from n to M (in practice, from $2^{14} = 16384$ to $2^8 = 256$). Also observe that, it is possible to apply masking for multiple records together in a batched way. Then, the number of multiplication operations is also reduced.

Following the above idea, we describe the algorithm for efficient batched elementwise mapping in the HADES protocol. Algorithm 1 shows the optimized algorithm and Figure 3 illustrates the procedure. By expanding the mapping to occupy the full ciphertext (assuming n can be divided by M) and calculate all M possible rotations, we cache all possible rotation forms. Then, we construct proper masks to select slots from the rotations that reflect the database value, multiply to apply the masks, and sum the masked rotations to get the final results.

Algorithm 1 Our efficient elementwise mapping algorithm **ApplyElementwiseMapping**$(pk, m, ct_f, d) \rightarrow ct$

```
[step 1]: Repeat the mapping to fill up the ciphertext
while $i \leftarrow [d, \log_2(n))$ do
    $ct_f \leftarrow$ Add$(pk, ct_f,$ Rotate$(pk, gk, ct_f, 2^i))$
end while
[step 2]: Prepare all possible rotations
$r\_ct_f \leftarrow$ list of $2^d$ vectors of ciphertexts        ▷ "r_" for "rotated"
$r\_ct_f[0] \leftarrow ct_f$
while $i \leftarrow [1, 2^d)$ do
    $r\_ct_f[i] \leftarrow$ Rotate$(pk, gk, r\_ct_f[i-1], 1)$
end while
[step 3]: Construct mask vectors in batch
$b\_ind \leftarrow$ list of $2^d$ vectors of plaintext zeros        ▷ "b_" for "batched"
while $i \leftarrow [0, n)$ do
    $b\_ind[(m[i] - i) \mod 2^d][i] \leftarrow 1$
end while
[step 4]: Index values with correct locations
$b\_rot \leftarrow$ list of $2^d$ vectors of ciphertexts        ▷ "b_" for "batched"
while $i \leftarrow [0, 2^d)$ do
    $b\_rot[i] \leftarrow$ Multiply$(pk, ek, r\_ct_f[i], b\_ind[i])$
end while
return $\sum_{i=0}^{2^d-1} b\_rot[i]$
```

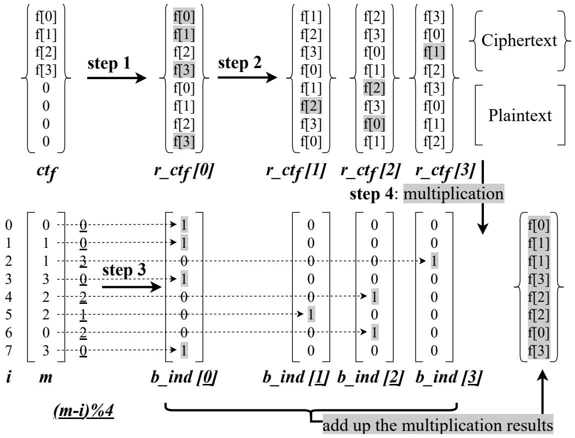


Figure 3: Illustration of the elementwise mapping algorithm. The steps are explained in Algorithm 1. The example uses $d = 2$, $n = 8$ for illustration. The ciphertext output applies elementwise mapping to values in message m as required.

Performance analysis. For the unoptimized protocol, we discussed earlier that for 8-bit elementwise mapping with $n = 2^{14}$, $M = 2^8$, the data provider needs to process 2^{14} multiplications, 2^{14} rotations, and $2^{14} - 1$ additions. For the optimized protocol, consider each step: (1) Repeating the mapping requires $\log_2(n/M)$ additions. (2) Preparing rotations requires $M - 1$ rotations. (3) Construct plaintext masks only require plaintext operations. (4) Applying masks requires M plaintext multiplications. (5) Sum across masked rotations requires $M - 1$ additions. Table 1 summarizes both the theoretical and concrete savings in the number of operations.

Saving across database chunks. The rotation caching not only saves the number of operations required for processing a single database chunk, but also improve performance when the database size is larger than the batching size. Consider a database size $N = k * n$, as discussed in earlier sections, to apply elementwise mapping to such a database, we need to first chunk the database then run protocol on each chunk. While for the basic protocol, the cost grows linearly with k, with rotation caching, the first two steps in the algorithm can be reused across different chunks as long as the mapping is the same, resulting in huge computation savings for large databases. To demonstrate the saving, in Table 2 we provide theoretical cost analysis along with a concrete example of a database with 2^{20}, around 1 million records. Consider in practice the rotation operation often takes more time than the plaintext multiplication, the optimized protocol results in huge savings.

Algorithm	Addition	Rotation	Multiplication
Unoptimized	$n - 1$	n	n
with $n = 2^{14}$, $M = 2^8$	16,383	16,384	16,384
Optimized	$\log_2(n/M) + M - 1$	$M - 1$	M
with $n = 2^{14}$, $M = 2^8$	261 (1.59%)	255 (1.56%)	256 (1.56%)

Table 1: Single-chunk elementwise mapping analysis

Algorithm	Addition	Rotation	Multiplication
Unoptimized	$k * n - k$	$k * n$	$k * n$
with $N = 2^{20}$, $M = 2^8$	1,048,512	1,048,576	1,048,576
Optimized	$\log_2(n/M) + k * M - k$	$M - 1$	$k * M$
with $N = 2^{20}$, $M = 2^8$	16,326 (1.56%)	255 (0.02%)	16,384 (1.56%)

Table 2: Multi-chunk elementwise mapping analysis

5.2 Hybrid Multi-Cipher Reduction

Simple aggregation via looping. While in Section 4.4, we explain how to ensure the correctness of aggregation by preventing overflow, here we describe a new merging algorithm that improves efficiency. When using HE to process aggregations of ciphertexts, a common practice involves two steps: (1) summing up the values in one ciphertext using rotations and additions, and (2) reduce the aggregated multiple ciphertexts into a single ciphertext for return while keeping the sum value for each original ciphertext separate. While sequential executing these two steps is direct and natural, we show by carefully designing an algorithm that merges two steps together, we can save around 80% total computation time.

Concretely, assume for input we have $k = 2^s$ ciphertexts, each encrypts a vector V_i that contains $n = 2^t$ slots, with $s <= t$. This usually represents a input database with $N = 2^s * 2^t$ entries in 2^s chunks. The goal is to get a single ciphertext encrypting vector V_{ans} with slots containing the sum of V_i, such that $V_{ans}[i] = \Sigma V_i$. Using a simple looping algorithm, we need k plaintext-ciphertext multiplications, $k * t$ rotations, and $k * (t + 1)$ additions.

Combining self-reduction and multi-cipher reduction. However, while it is natural to separate two steps, there is a significant waste of computation during the self-addition of the single ciphertext in the first step. Particularly, during the $j'th$ iteration of the second-level loop, the addition between $2^{(t-j-1)}$ values inside the ciphertext are actually repeated 2^j times. To avoid such waste, we propose a new algorithm to reduce the overall computation cost. The new algorithm combines two steps by merging ciphertexts and values inside the ciphertext simultaneously, avoiding the repetition mentioned earlier. As a result, it requires around $2 * k$ plaintext-ciphertext multiplications, $2 * k$ rotations, and $4 * k$ additions. Compared with the simple solution, this avoids a factor of t in both the rotations and additions.

While the complexity for rotations and additions has been significantly reduced, the number of multiplications required is doubled compared with the simple looping algorithm. In practice, the rotation is often as time-consuming as plaintext multiplication, thus the overall saving is still huge. In practice, for $n = 2^{14}, t = 14$, the time consumption ratio between plaintext multiplication, rotation, and addition is around $13.8 : 32.8 : 1$. Omitting the common factor k, the optimization leads to about 80% saving.

Noise budget tradeoff. While the new algorithm overall improves the efficiency, it is worth noting that the consecutive multiplications consume significantly more noise budget than the naive approach. In practice, one may combine the naive algorithm and the optimized algorithm by running the optimized algorithm for the first few steps, and then the naive algorithm to make sure the noise budget is not used up in the end. In such a hybrid solution, because the first few steps can exponentially reduced the number of ciphertexts, the overall saving is still significant.

6 IMPLEMENTATION

Real-world database applications often demand high performance and rich functionalities to support complex query semantics. The HE algorithms introduced in Sections 4 and 5 present complex data dependencies, challenging the implementation of highly parallelized systems. Moreover, the basic protocol discussed thus far only covers limited query keywords and preliminary data types, such as unsigned integers.

To address these challenges and achieve the goals of SQL support, latency efficiency, and scalability outlined in Section 2.3, we developed HADES in 3.6k lines of C++ and 1.2k lines of Python using multi-thread programming. We utilized Microsoft SEAL [22] for BFV homomorphic operations, selecting a polynomial-size of 2^{14} and using CRT to encode 2^{17} bits in each slot. The parameters are selected to achieve the best trade-off between per-slot operation speed and adequately sufficient noise budget. Next, we detail our approach. First, we discuss our staged multi-thread design, which accommodates the data dependency topology. Then, we explore protocol extensions that broaden the coverage of SQL keywords and data types. Our protocol implementation is open-sourced and available at https://github.com/camelop/hades-dev .

Multi-thread scheduling. In Section 3, we introduce the five stages of the HADES protocol, with the middle three stages executed on the data provider side. While parallelization is critical for efficiency and scalability, the complicated data dependencies in HADES present significant implementation challenges. Specifically, each stage involves unique dependency structures and thus requires synchronization and task regrouping. In the indicator calculation stage, the rotation preparation for each encrypted value is only processed once and then used for multiple database chunks. In the boolean circuit evaluation stage, indicators generated in the earlier stage for the same chunk of the database are grouped together following the logical operations specified in the query template. In the record aggregation stage, indicators from different chunks across different subqueries are merged.

To properly organize the execution order, in HADES we explicitly construct a CSV-formatted intermediate representation (IR) to describe task dependencies across different stages. Each line of the CSV contains a parallelizable task with a unique task ID, the task parameters (e.g. which database chunk to access, what boolean operation to compute), and describes its dependent IDs for cross-referencing during execution. To reduce communication between tasks and efficiently share dependent task objects, we choose thread-level parallelization and maintain read-only access for dependent objects. Our separation of stages guarantees that all cross-task dependencies are crossing stages, significantly reducing unnecessary synchronizations and avoiding all dependent task status checks.

To implement the above mechanism, in practice we built a query compiler in Python that pre-fetches table statistics and generates execution IR based on the input query template. We built a multi-thread C++ execution engine that interprets the generated IR and accesses the database to complete the query processing. The C++ execution engine maintains a thread pool to execute tasks from the IR in the same stage utilizing SEAL, goes through all three stages sequentially, and finally generates the result ciphertexts.

Support more keywords and data types. HADES supports operators like AVG, GROUP BY, ORDER BY, and formulas in SELECT statements through multi-subquery merging and preprocessing. It optimizes execution by computing a shared encrypted indicator vector for subqueries and merging aggregation results to minimize communication. AVG is supported by combining SUM and COUNT operations, allowing the analyst to compute averages locally. For GROUP BY, the server preprocesses data into groups and applies queries to each, sending separate results back. Formulas in SELECT statements are precomputed by creating new columns, and ORDER BY is processed locally by the analyst. HADES also supports signed values, fixed-point numbers, and strings by applying value conversions and hashing.

7 EVALUATION

We evaluate HADES, focusing on its query processing latency, scalability, and comparison with state-of-the-art baseline systems. Furthermore, we explore the reasons for the system performance improvement by analyzing the effectiveness of our algorithm optimizations in Section 5. Here we summarize our main results.

- For three TPC-H SQL queries on one million records, HADES effectively reduces the query latency compared with the best previous results from HE3DB and HEDA. Specifically, with the public data setting, it reduces the latency from 14h (Q1), 27h (Q4), and 5h (Q6), to 17.2s, 14.9s, and 80.5s, achieving 2981x, 6574x, and 204x speedup respectively.
- Despite the complicated internal data dependencies, the query processing performance of HADES sufficiently scales when the thread number increases. With 128 threads, the latency to run Q1 on one million records significantly reduces from 760.8s, the single-thread results, to 17.2s.
- Both optimized algorithms demonstrate significant advantages in time consumption and noise budget saving, contributing to the speedup of HADES query protocol.

Experiment setups. TPC-H [24] is a decision support benchmark for SQL queries on large databases, which provides query specifications and table generation utilities. For comparison with previous work, we mainly focus on three queries from TPC-H, Q1, Q4, and Q6 in the experiment. Q1 aggregates eight columns of values in four groups with a predicate on a date column. Q4 aggregates a single column in five groups with multiple predicates that contain a sub-query, which could be pre-processed. Q6 also aggregates a single column without any grouping but has more complicated predicates. While Q1 and Q6 target the main table "lineitem" from the benchmark, which has a directly adjustable size, Q4 targets a different table "orders". To align the table size for Q4, we tune the generation factor from the TPC-H tool so that the generated table size is just over the required size (12K for 10K experiment, 102K for 100K experiment, and 1.002M for 1M experiment). We run all end-to-end experiments with 128 threads on an AMD EPYC 9654 platform. For micro-benchmarks on system components, we report the single-thread performance for clearer comparison.

Baselines. For end-to-end experiments, we mainly compare our results with two previous works, HEDA [21] (192 threads) and HE3DB [4] (96 cores[2]), as our baselines. For HEDA Q1, we cite the performance number for their simplified query (GROUP BY, ORDER BY, AVG removed). For HE3DB, we cite the estimated used time for processing 1M-record, as the actual time is not provided in the paper. HEDA does not provide evaluations in Q4. HEDA-(Q1,Q6)-1M, HE3DB-(Q1,Q6)-100K, HE3DB-Q4 numbers are obtained by interpolation. For micro-benchmarks on system components, the baseline methods used for comparisons are discussed in Section 4.

7.1 TPC-H Latency Comparison

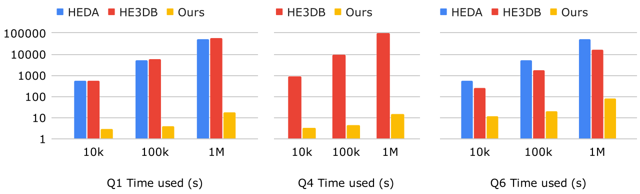


Figure 4: TPC-H latency comparison between HADES, HEDA, and HE3DB. HEDA does not report numbers on Q4.

	Q1 - 10K			Q1 - 1M			Q4 - 1M		
Stages	T	T%	N	T	T%	N	T	T%	N
IC - rotation caching	1.20	42%	27	1.19	7%	28	1.29	9%	27
IC - mask & index	0.47	16%	27	7.31	42%	27	12.28	84%	25
Boolean Circuit Eval	0.17	6%	80	0.23	1%	79	0.50	3%	112
Record Agg	1.03	36%	144	8.52	49%	163	0.56	4%	115

Table 3: HADES performance breakdown in different query processing stages. "IC" on left stands for the indicator calculation stage and two corresponding rows are the two steps in the IC stage. "T" columns provide time used in seconds and "T%" columns show corresponding percentages. "N" columns measures the bits of noise budget consumed by the stage (total budget: around 366 bits).

We first focus on the end-to-end query latency comparison with baseline systems for the TPC-H queries. Figure 4 presents comparison results for various queries with different database sizes. For the smallest 10K workload size, HADES takes 2.9s, 3.3s, and 11.2s respectively to execute each of these three queries. As the first work to report actual latency measured on one million scale workload, HADES reduces the processing time to 17.2s, 14.9s, and 80.5s, achieving 2981x, 6574x, and 204x speedup on three queries compared with the best previous results between HEDA and HE3DB in the public data setting. In terms of the communication cost, for all end-to-end experiments, HADES uses a single ciphertext to encrypt and encode all the query parameters, which takes around 1.74MB. The query results are also encoded in a single ciphertext of the same size.

Comparing different scales of the workload, we observe that as the workload becomes larger, the improvement factor, which is reflected in the bar height difference, also grows. For example in Q1, we improve from HEDA by 187x for 10K records, 1288x for 100K records, and 2981x for 1M records. The reason is that the 10k workload is too small for parallelization, as it cannot even fill one ciphertext, which provides 16k slots. The strength of BFV SIMD operations and multi-threading are not fully exploited compared with baseline solutions that use TFHE.

Comparing different queries, the improvement for Q1 and Q4 is more significant than Q6. This is because Q6 is more predicate-focused and only asks for a single aggregation, while Q1 and Q4 require multiple aggregation columns across different groups of records, resulting in more database chunks reusing the same mapping, and more indicators to merge. Consequently, Q1 and Q4 benefit more from the rotation caching and optimized aggregation algorithms proposed in this paper.

To further analyze the bottleneck of the solution, we profile the protocol execution on TPC-H queries to obtain its performance breakdown in different stages. In addition, we pause the protocol between stages and temporarily reveal the private key to measure the remaining noise budget. In this way, by calculating the difference, we can learn the noise budget consumption in different stages. Table 3 presents a detailed breakdown for the time consumption and the noise growth.

Comparing different queries, the bottleneck stage varies according to the query focus. Q4 asks for the aggregation of fewer columns than Q1 and has more complicated predicates. As a result, the indicator calculation stage for Q4 takes more time.

Comparing the breakdown for the same query (Q1) with different workload scales, we observe major time consumption growth in indexing and aggregation, as both steps are highly parallelizable and are already fully utilizing multi-threading. Contrarily, the boolean circuit evaluation, which is less parallelizable, shows less time growth as a larger workload helps improve the utilization rate of multi-threading. The top region in Figure 5 directly illustrates such a difference. The time needed for rotation caching keeps the same for Q1 with different workload scale. The reason is that the computation required by the rotation caching step is only affected by the predicate, and is irrelevant to the size of the database. Such a design in our optimized algorithm also contributes to the overall savings, especially for larger workloads.

[2] Whether hyper-threading is enabled is not mentioned from the reference.

In terms of noise growth, we observe the latter two stages consume more noise budget in general. When the number of records increases, the noise budget required by aggregation also increases. This is because more steps of merging are required for more records, indicating more layers of ciphertext multiplications for masking.

7.2 Scalability

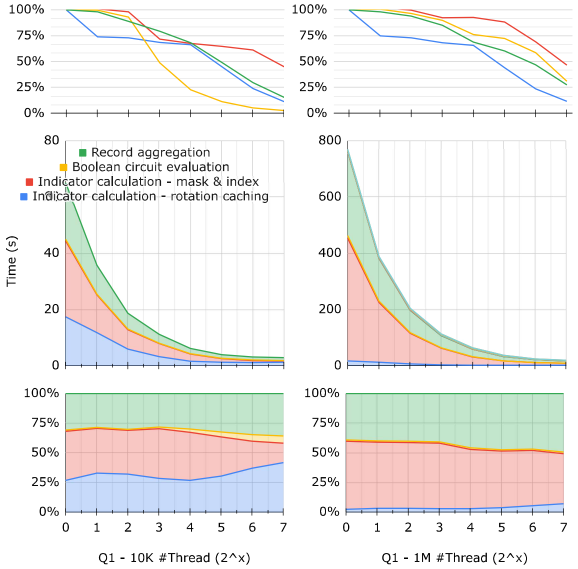


Figure 5: TPC-H Q1 profiling with different number of threads for 10K/1M records. The top region shows the thread utilization rate. The middle region shows for all stages, the accumulative latency decreases as the number of threads increases. The bottom region shows the time consumption percentage for each stage.

Time used (s)	8-bit	16-bit	32-bit	48-bit	64-bit
PIR-style retrieval (method A)	1761.67 (1.74MB)	1767.11 (6.97MB)	* (457 GB)	* (29 PB)	* (2 ZB)
Bitwise circuit (method B)	2780.04	5591.73	11159.98	16705.79	22394.79
8-bit retrieval + **emap** (ours)	8.19	15.08	30.88	45.64	61.47
1-bit (bitwise) + **emap**	3.49	7.24	13.84	20.65	29.11

Table 4: Point query algorithm comparison on 2^{14} records. For PIR-style retrieval experiment skipped and marked $*$, while theoretically the computation time remains the same for different bits, the communication cost is far from practical.

Time used (s)	2^{14} records	2^{17} records	2^{20} records
Naïve emap	1794.04	14352.31*	114818.47*
Opt emap w/o rot caching	8.17	60.84	478.33
Opt emap	8.18	23.25	146.08

Table 5: Ablation study on the elementwise mapping operator for rotation caching in 8-bit point predicate indicator generation. Values marked * are estimated.

To achieve practical performance for large tables, HADES achieves scalability by performing multi-threading in each query processing stage as mentioned in Section 6. Here we investigate the scaling characteristic of HADES and Figure 5 shows how the processing time decreases as more threads are used. We observe that all stages benefit from multi-threading. With an increasing number of used threads, the overall time significantly reduced to seconds.

By comparing with the single-thread performance, we calculate the utilization rate of each thread by its relevant slowdown and illustrate it on the top region of Figure 5. For the most time-consuming "mask & index" step (red) and record aggregation stage (green), the utilization rate keeps above 50% before reaching 32 cores. Also, the utilization rate is generally higher for the larger 1M workload. We also verified that as discussed earlier, the boolean circuit evaluation stage is less parallelizable compared with other stages. We envision that to further reduce query processing time to the sub-second level, a future direction is to further decompose basic HE operations via multi-threading or specific hardware acceleration, as a single ciphertext multiplication is already taking around 70 ms.

7.3 Micro-benchmark

To further investigate the source of acceleration for HADES, we run micro-benchmarks to measure the performance gain from the optimizations mentioned in Section 5.

Indicator generation algorithm comparison. First, we evaluate the indicator generation procedure for different algorithms by their time consumption, communication cost, and introduced noise. The algorithm starts with the encrypted point predicates and 2^{14} records, and outputs the ciphertext indicators for whether the records are selected. We choose four algorithms as our comparison targets. We describe the basic PIR-style retrieval and bitwise circuit methods in Section 4. In addition to our combined method that uses an 8-bit retrieval building block with elementwise mapping, we also add an optimized version of bitwise circuit leveraging the elementwise-mapping operator we designed, which can be treated as a variant that uses a 1-bit retrieval building block. Table 4 summarizes the performance comparison between different algorithms for point predicates with varying bit length.

Comparing with baseline methods, we observe that the elementwise mapping algorithm significantly accelerate point query speed. Additionally, we find that most algorithms scale nearly linearly with the bit-number to be compared, the only exception is PIR-style retrieval but its communication cost becomes impractical when there are more than 16 bits. When selecting the mapping size for the elementwise mapping algorithm, the 8-bit building block avoids 3 levels of ciphertext multiplications at the cost of doubling the computation time. Because ciphertext multiplication is the dominant factor for noise growth, avoiding these 3 levels helps greatly in saving the noise budget. Considering the end-to-end performance, the noise budget saved from here can be used to significantly reduce record aggregation time, by incorporating more optimized merging steps as we discuss next. Similarly, range queries also benefit from elementwise mapping, as the underlying building block is exactly the same as point queries.

Elementwise mapping operation optimization. Here we also take a deeper look into the elementwise mapping step in the point query to verify the theoretical saving discussed in Section 5. Table 5 provides an ablation study for the optimizations used on 8-bit point query indicator generations.

We first observe that the naive protocol is completely impractical even for a slightly larger database. Processing a single 8-bit mapping on 1m records takes 1.3 days. From the "2^{14} records" column that only processes a single ciphertext, compared with the naive protocol, we observe a huge saving from our optimized algorithm, down to 0.46%. We also notice that the actual saving in the experiment is even more significant than our theoretical analysis from operation number estimation in Section 5 (1.56%). This is because in practice, the rotation operation does not have a constant cost. Its cost is influenced by the rotation offset and is minimal when the offset is 1. While our optimized method always uses an offset of 1 to generate all possible rotations, the naive method uses a dynamic offset, resulting in many times of extra cost.

To verify the effectiveness of rotation caching across different database chunks, Table 5 also provides an ablation study without caching the rotation across different database chunks. We observe that when applying the same mapping to more database chunks, rotation caching significantly saves the computation time, which is aligned with our discussion in Section 5, reducing the cost to process each additional 2^{14} records from 7.5s to 2.2s.

Record aggregation optimization. In addition to indicator generation, we also measure the performance of record aggregation. Specifically, we compare the simple aggregation, optimized aggregation and the hybrid solution with different numbers of optimized steps. We use the layer number of multiplication needed to estimate the noise growth. Table 6 provides the comparison details. We observe that the fully optimized aggregation is the fastest among all methods, at the cost of huge noise growth. Besides, all three methods with optimized steps significantly improve from the simple aggregation, providing up to 4.5x speedup, which matches our analysis in Section 5.2. Among them, the 4-step hybrid algorithm achieves performance close to the fully optimized version, while maintaining a controllable noise consumption. This is because in the first four steps, the number of ciphertexts needed to be processed has already been significantly reduced.

To summarize, all proposed optimized algorithms significantly outperform their baseline methods, which leads to the overall latency reduction in the end-to-end HADES query processing.

	Time used (s)				# Mul layer			
# Record	2^{14}	2^{17}	2^{20}	2^{23}	2^{14}	2^{17}	2^{20}	2^{23}
Simple aggregation	0.3	2.4	19.2	153.6	1	1	1	1
Opt aggregation	0.3	0.7	4.3	34.2	1	4	7	10
Hybrid - 2 step	0.3	0.9	7.4	59.0	1	3	3	3
Hybrid - 4 step	0.3	0.7	4.8	39.1	1	4	5	5

Table 6: Record aggregation algorithm comparison.

8 RELATED WORK

FHE-based encrypted database. There are several closest relevant recent works [4, 14, 21, 23] that have explored efficient FHE-based protocol design for query processing with private query and private database. HEDA [21] designs a conversion protocol between two types of FHE ciphertext, allowing the use of both the numerical and binary forms of the data to be used during the query processing. HE3DB [4] proposes new HE operators to further improve the performance and support more query functionalities. Kim et al. [14], compared with the other two works, focuses on the optimization of the equality predicate operators rather than the full query, discussing the construction for both conjunctive and disjunctive queries. Tan et al. [23] further explore range predicates for record retrieval rather than aggregation, with proposed VFE encoding groups the bits to be compared for efficiency. All the above works focus on protocol design for private databases. In comparison, our work explores the PIR-style HE protocol design for public databases, thus achieving order-of-magnitude better performance in more specific use cases as shown in Section 7.

For database with N records	**HADES**	**Splinter (M parties, M>=2)**	**Hafiz (M parties, M>=2)**
Assumptions	Cryptographic (RLWE)	Cryptographic (FSS) + Non-colluding - At least one honest	Information-theoretic + Non-colluding - Above given threshold
Communication	1 round with 1 party	O(logN) rounds with M parties for disjoint conditions	1 round with M parties
Data duplications	1	M	M
Computation	O(N)	O(NlogN)	O(N)

Table 7: Comparison with MPC-based solutions.

MPC-based and Enclave-based private query. There are some other works [8, 12, 16, 25, 26] that target similar functionalities with different assumptions. Splinter [25] also targets private queries on public data. It relies on function secret sharing to hide query parameters in a non-colluding server setting. Similarly, Hafiz et al. [12] relies on an improved IT-PIR protocol to process aggregation queries with multiple non-colluding servers. While these solutions support the setting in HADES, they require non-collusion, more communication rounds, and multiple database copies as summarized in Table 7. In terms of performance, their experiments lack standard TPC-H results and do not include interval-range queries (e.g., $l\_quantity < 24$). For comparable equality-based queries on millions of records, the MPC-based solution demonstrates higher throughput; however, the difference remains within the same order of magnitude (Splinter-custom-Q3 [3] : 173 records/ms vs. HADES-Q4: 67 records/ms). Designed on top of the distributed hardware enclaves, Opaque [26] uses a threat model mainly focusing on root adversaries from the cloud provider, and mitigates access pattern attacks. It relies on the use of specific hardware and assumes that the adversary cannot compromise the trusted hardware.

Extensions for PIR. While the basic PIR [7, 11, 17] solutions mainly support retrieval with indexing, there has been efforts extending this functionality for practical convenience. Keyword PIR [1, 6, 18, 20] extends the use of index to an identifier, functionally introduces an equality check on a single field for retrieving the record. Cristofaro et al.[9] and Boneh et al.[5] extend the predicate support to cover disjunctive and conjunctive clauses. Coeus [2] discusses supporting private document relevance ranking in query. Hayata et al. [13] discusses the range query for IT-PIR. It claims the lack of formal insecurity for existing query privacy preserving schemes and presents an FSS-based multi-round range query protocol with a non-colluding server setting. Also targeting to extend PIR functionality, our solution HADES extends CPIR for data aggregations, efficiently processing comprehensive boolean combinations of point and range predicates.

ACKNOWLEDGMENTS

This work was funded in part by JPMorgan Chase Co. , supported by c3.ai, Lab for Inclusive FinTech , NSF (#2115075, #2126327), and ARPA-H (SP4701-23-C-0074). We also thank Antigoni Polychroniadou, Yujie Lu, and Tynan Sigg for their help with this work.

[3] Hafiz [12] presents a similar query performance without the concrete throughput number for the specific query in a comparable setting.

REFERENCES

[1] Ishtiyaque Ahmad, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta. 2022. Pantheon: Private retrieval from public key-value store. *Proceedings of the VLDB Endowment* 16, 4 (2022), 643–656.

[2] Ishtiyaque Ahmad, Laboni Sarker, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta. 2021. Coeus: A system for oblivious document ranking and retrieval. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 672–690.

[3] Jean-Claude Bajard, Julien Eynard, M Anwar Hasan, and Vincent Zucca. 2016. A full RNS variant of FV like somewhat homomorphic encryption schemes. In *International Conference on Selected Areas in Cryptography*. Springer, 423–442.

[4] Song Bian, Zhou Zhang, Haowen Pan, Ran Mao, Zian Zhao, Yier Jin, and Zhenyu Guan. 2023. HE3DB: An Efficient and Elastic Encrypted Database Via Arithmetic-And-Logic Fully Homomorphic Encryption. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2930–2944.

[5] Dan Boneh, Craig Gentry, Shai Halevi, Frank Wang, and David J Wu. 2013. Private database queries using somewhat homomorphic encryption. In *International Conference on Applied Cryptography and Network Security*. Springer, 102–118.

[6] Benny Chor, Niv Gilboa, and Moni Naor. 1997. *Private information retrieval by keywords*. Citeseer.

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

[8] Emma Dauterman, Mayank Rathee, Raluca Ada Popa, and Ion Stoica. 2022. Waldo: A private time-series database from function secret sharing. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2450–2468.

[9] Emiliano De Cristofaro, Yanbin Lu, and Gene Tsudik. 2011. Efficient techniques for privacy-preserving sharing of sensitive information. In *International Conference on Trust and Trustworthy Computing*. Springer, 239–253.

[10] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch.* 2012 (2012), 144.

[11] William Gasarch. 2004. A survey on private information retrieval. *Bulletin of the EATCS* 82, 72-107 (2004), 113.

[12] Syed Mahbub Hafiz, Chitrabhanu Gupta, Warren Wnuck, Brijesh Vora, and Chen-Nee Chuah. 2024. Private Aggregate Queries to Untrusted Databases. *arXiv preprint arXiv:2403.13296* (2024).

[13] Junichiro Hayata, Jacob CN Schuldt, Goichiro Hanaoka, and Kanta Matsuura. 2024. On private information retrieval supporting range queries. *International Journal of Information Security* 23, 1 (2024), 629–647.

[14] Myungsun Kim, Hyung Tae Lee, San Ling, and Huaxiong Wang. 2016. On the efficiency of FHE-based private queries. *IEEE Transactions on Dependable and Secure Computing* 15, 2 (2016), 357–363.

[15] Kim Laine. 2017. Simple encrypted arithmetic library 2.3. 1. *Microsoft Research https://www. microsoft. com/en-us/research/uploads/prod/2017/11/sealmanual-2-3-1. pdf* (2017).

[16] John Liagouris, Vasiliki Kalavri, Muhammad Faisal, and Mayank Varia. 2023. SECRECY: Secure collaborative analytics in untrusted clouds. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 1031–1056.

[17] Helger Lipmaa. 2010. First CPIR protocol with data-dependent computation. In *Information, Security and Cryptology–ICISC 2009: 12th International Conference, Seoul, Korea, December 2-4, 2009, Revised Selected Papers 12*. Springer, 193–210.

[18] Rasoul Akhavan Mahdavi and Florian Kerschbaum. 2022. Constant-weight PIR: Single-round keyword PIR via constant-weight equality operators. In *31st USENIX Security Symposium (USENIX Security 22)*. 1723–1740.

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

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

[21] Xuanle Ren, Le Su, Zhen Gu, Sheng Wang, Feifei Li, Yuan Xie, Song Bian, Chao Li, and Fan Zhang. 2022. HEDA: multi-attribute unbounded aggregation over homomorphically encrypted database. *Proceedings of the VLDB Endowment* 16, 4 (2022), 601–614.

[22] SEAL 2018. Microsoft SEAL (release 3.0). http://sealcrypto.org. Microsoft Research, Redmond, WA.

[23] Benjamin Hong Meng Tan, Hyung Tae Lee, Huaxiong Wang, Shuqin Ren, and Khin Mi Mi Aung. 2020. Efficient private comparison queries over encrypted databases using fully homomorphic encryption with finite fields. *IEEE Transactions on Dependable and Secure Computing* 18, 6 (2020), 2861–2874.

[24] Transaction Processing Performance Council. 1999. TPC Benchmark H (TPC-H) Standard Specification. http://www.tpc.org/tpch/. Version 2.17.1.

[25] Frank Wang, Catherine Yun, Shafi Goldwasser, Vinod Vaikuntanathan, and Matei Zaharia. 2017. Splinter: Practical private queries on public data. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 299–313.

[26] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. 2017. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 283–298. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/zheng