ArcEDB: An Arbitrary-Precision Encrypted Database via (Amortized) Modular Homomorphic Encryption

Zhou Zhang∗
zhouzhang@buaa.edu.cn
Beihang University
Beijing, China

Song Bian∗
sbian@buaa.edu.cn
Beihang University
Beijing, China

Zian Zhao
zhaozian@buaa.edu.cn
Beihang University
Beijing, China

Ran Mao
maoran_44@buaa.edu.cn
Beihang University
Beijing, China

Haoyi Zhou
haoyi@buaa.edu.cn
Beihang University
Beijing, China

Jiafeng Hua
jiafenghua@163.com
Xidian University
Xian, China

Zhongguancun Laboratory
Beijing, China

Yier Jin
jinyier@gmail.com
University of Science and Technology of China
Hefei, Anhui, China

Zhenyu Guan†
guanzhenyu@buaa.edu.cn
Beihang University
Beijing, China

1 INTRODUCTION

The past quarter century has witnessed the rise of a golden age in building cloud-based outsourcing services [8, 9, 31, 85] to meet the increased needs of low-cost and flexible data management. However, such convenience is often accompanied by data security concerns, as data owners may lose control of their sensitive information and suffer from data breaches. Consequently, we see growing interest from both the data owners and the cloud service providers in developing encrypted database (EDB) infrastructures [5, 16, 24, 33, 56, 80, 83, 86, 87, 99], where data confidentiality is provably secured over the entire outsourcing life cycle.

While a straightforward application of symmetric-key encryption suffices the need of provably secure data storage, the challenge for EDB systems is how to efficiently execute expressive queries over the outsourced data. Hence, although there exists multiple lines of work that study how to securely store [49], retrieve [5, 80, 83, 84, 86, 87], and process [24, 33, 39, 43, 56] encrypted data outsourced to the cloud, such protocols either lack provable security guarantees [36, 47, 65, 84, 103] or only focus on a specific set of data processing functionalities (e.g., data storage only [49] or encrypted search only [38]). For example, oblivious RAM (ORAM) [22, 41, 43, 93, 102] is a well-known primitive in efficiently storing data in an oblivious way, such that fetching data securely from the server achieves poly-logarithmic overheads. Nonetheless, as observed in [39], existing ORAM protocols become less effective when handling multi-dimensional queries (i.e., predicates over multiple attribute columns), where the worst-case querying communication complexity grows to linear, defeating the purposes of adopting ORAM in the first place.

To solve the efficiency-expressiveness dilemma, a line of recent works [16, 99] explore how fully homomorphic encryption (FHE) can be used to construct EDB systems. In particular, [16] proposes a cross-scheme FHE infrastructure that can be used to implement

ABSTRACT

Fully homomorphic encryption (FHE) based database outsourcing is drawing growing research interests. At its current state, there exist two primary obstacles against FHE-based encrypted databases (EDBs): i) low data precision, and ii) high computational latency. To tackle the precision-performance dilemma, we introduce ArcEDB, a novel FHE-based SQL evaluation infrastructure that simultaneously achieves high data precision and fast query evaluation. Based on a set of new plaintext encoding schemes, we are able to execute arbitrary-precision ciphertext-to-ciphertext homomorphic comparison orders of magnitude faster than existing methods. Meanwhile, we propose efficient conversion algorithms between the encoding schemes to support highly composite SQL statements, including advanced filter-aggregation and multi-column synchronized sorting.

We perform comprehensive experiments to study the performance characteristics of ArcEDB. In particular, we show that ArcEDB can be up to 57× faster in homomorphic filtering and up to 20× faster over end-to-end SQL queries when compared to the state-of-the-art FHE-based EDB solutions. Using ArcEDB, a SQL query over a 10K-row time-series EDB with 64-bit timestamps only runs for under one minute.

CCS CONCEPTS

• Security and privacy → Cryptography; Management and querying of encrypted data.

KEYWORDS

Fully Homomorphic Encryption, Secure Database Outsourcing

Both authors contributed equally to this work.

Corresponding author.
A comprehensive list of common SQL query statements, including SELECT, WHERE, GROUP BY, ORDER BY, JOIN, etc. Unfortunately, the query expressiveness of [16] does come at the price of usability issues, especially when compared to the less expressive solution [72]. Hence, in its current state, we observe the following two main obstacles confronting FHE-based EDB frameworks:

- **Limited Data Precision** Most existing FHE-based EDB frameworks [16, 99] attain faster computation speed at the cost of lower data precision. For example, even the most recent work [16] can only perform end-to-end SQL evaluation over encrypted data up to 32-bit precisions. It is obvious that such loss of data precision can be fatal in real-world DB systems, especially in specialized data management systems such as time-series databases. In such cases, complex data types, such as the DATE and TIMESTAMP attributes, are often encoded into large-size integers, where a loss of precision can defeat the purpose of the entire system.

- **Slow Evaluation Speed** Evaluating complex SQL statements over data at scale requires a huge amount of homomorphic operations, where the dominant computations are the (high-precision) homomorphic comparisons between ciphertexts. For instance, as illustrated in Table 1, while HE3DB [16] achieves better performance than HEDA [99], it spends the most amount of computation time in producing filtering results. In particular, when performing 16-bit precision TPC-H Q6 [37] on a 1K-row database, 93% of the overall query evaluation time is consumed by the filtering process. Therefore, completing high-precision comparisons over a large volume of encrypted entries is one of the biggest bottlenecks against efficient EDB designs.

### 1.1 Our Contribution

To address the above challenges, we propose ArcEDB, an FHE-based encrypted database framework that can simultaneously achieve arbitrary-precision data processing and low-latency query evaluation. Specifically, we first build a new FHE infrastructure based on a variant of the modular fully homomorphic encryption (MFHE) scheme equipped with novel encoding techniques and advanced homomorphic operators tailored for SQL processing. Next, building upon the MFHE infrastructure, we formalize the abstract data types to establish a set of standard application program interfaces (APIs) for the efficient evaluation of large-precision SQL queries, where the queries can be composed of a complex combination of filtering, aggregation, and non-polynomial functions (e.g., ORDER BY). The main contributions of ArcEDB are summarized as follows.

- **A New Encrypted SQL Evaluation Infrastructure**: We propose new data encodings, ciphertext types, and operator designs to efficiently evaluate complex SQL over arbitrary-precision encrypted data. To the best of our knowledge, ArcEDB provides the first purely FHE-based infrastructure that supports arbitrary-precision composite SQL with highly expressive clauses, such as unbounded-depth filtering, arithmetic aggregation (e.g., SUM, COUNT) and advanced logic aggregation (e.g., ORDER BY).

- **Arbitrary-Precision Amortized FHE Comparison**: Leveraging our DB-specific modular FHE ciphertexts, we devise a new segment-merging strategy to extend a low-precision comparison to the arbitrary-precision homomorphic comparison operator ArbHCMP. Furthermore, we also show how to batch ArbHCMP to efficiently filter large numbers of data rows.

- **Synchronized Sorting and Aggregation**: To the best of our knowledge, ArcEDB is the first FHE-based EDB framework that supports multi-column synchronized aggregation. More concretely, we can efficiently sort an EDB column to generate a particular order, and synchronize such order across all of the EDB columns for subsequent data retrieval and processing.

- **Performance Improvements**: We show that ArcEDB outperforms the best-performing FHE-based EDB implementations on all SQL microbenchmarks, and is on average 20x faster than the state-of-the-art (SOTA) on end-to-end SQL benchmarks. We demonstrate that using ArcEDB, an encrypted query over time-series databases can finish within one minute. Our code is publicly available\(^1\).

### 1.2 Related Works

#### 1.2.1 Outsourced Encrypted Databases

Over the past decade, a plethora of protocol and system designs are proposed to efficiently execute SQL queries over encrypted data [6, 7, 10–12, 16, 34, 39, 40, 43, 48, 56, 63, 69, 72, 74, 84, 91, 94, 95, 97–99, 105, 107–110, 113]. We focus on existing EDB designs with a special emphasis on outsourced EDB schemes.

Securely processing outsourced data over the cloud is one of the most common applications of EDB [16, 56, 72, 91, 94, 97, 99]. We see two main lanes of research in the area of outsourced EDB: i) storage-centric and ii) query-centric. Protocols such as ORAM [22, 41, 43, 93, 102] are considered as storage-centric protocol, since such protocols perform extremely well (poly-log complexity) at obliviously storing and retrieving encrypted data. Nonetheless, as mentioned above, storage-centric protocols are generally not efficient at handling composite SQL statements over multi-dimensional EDBs. In contrast, dedicated protocols such as [16, 40, 48, 56, 72, 91, 92, 94, 97, 99, 104] are propose to enable fast evaluation of complex queries over encrypted data, and are therefore classified as query-centric EDB protocols. Unfortunately, a number of such constructions, especially schemes based on searchable encryption [21, 38, 56, 91, 94] and order-preserving encryption [1, 75, 97], are challenged by leakage-abuse attacks [20, 40, 52–54, 64, 67, 70, 71, 90, 96, 112]. Recently, FHE-based protocols [16, 56, 99] gain increasing popularity in implementing EDB, attaining both expressive query evaluation and provable security. However, as further elaborated in Section 1.2.2, the

\(^1\)https://github.com/zhouzhangwalker/ArcEDB
We give a comprehensive review of existing homomorphic comparison parameters. Consequently, many FHE-based EDB solutions [16, 72, 99] create arbitrarily deep comparison trees with fixed encryption parameters, ciphertext comparisons. By leveraging its inherent bootstrapping capability, ciphertext comparisons over FHE can be implemented based on two main approaches: i) leveled comparison and ii) unbounded-depth comparison. In what follows, we discuss the benefits and drawbacks of each of the approaches.

**Leveled Homomorphic Comparison:** As detailed in [27, 28, 62], leveled homomorphic comparison methods typically approximate the comparison function using high-degree polynomials. In this way, the computation of homomorphic comparison is transformed into the evaluation of a univariate polynomial over the input ciphertexts, and the overall latency can be amortized using the single instruction multiple data (SIMD) properties of the BFV or CKKS ciphertext. Unfortunately, these methods face two fundamental challenges when applied to encrypted databases. First, the depth (i.e., degree) of the polynomial needs to be known *a priori*, since the depth determines the encryption parameters used to encrypt the database. However, when run-time queries demand the evaluation of a polynomial deeper than the pre-defined maximum depth, the entire database needs to be re-encrypted using a new set of encryption parameters, incurring prohibitive overheads to the protocol participants, especially the client. Second, the polynomial approximation techniques in [27, 28] do not directly support encrypted aggregations over results from ranged equality tests (e.g., \( \geq \)). The main reason here is that, due to the intrinsic continuity of the approximation polynomial, the comparison result of two equal inputs will become 1/2 (instead of 1 for true or 0 for false).

**Unbounded-Depth Homomorphic Comparison:** Different from leveled homomorphic comparisons, unbounded-depth homomorphic comparisons [16, 32, 55, 76, 78] mainly adopt the programmable bootstrap operator (PBS) proposed in [29] to carry out ciphertext comparisons. By leveraging its inherent bootstrapping capability, PBS-based homomorphic comparison schemes can evaluate arbitrarily deep comparison trees with fixed encryption parameters. Consequently, many FHE-based EDB solutions [16, 72, 99] prefer using unbounded-depth comparisons to implement the filtering [16, 99] and sorting [16]. Despite the usability benefits, many existing unbounded-depth FHE comparisons [16, 32, 55, 76, 78] suffer from both low data precision and slow efficiency. In fact, as also elaborated in Table 2, most of the existing homomorphic comparison methods (including many of the leveled comparison schemes) cannot compare ciphertexts that encrypt \( \geq 32\)-bit plaintext values in an efficient manner. To mitigate the deficiency in precision, some works [32, 55] seek a bit-wise encryption approach, where each ciphertext only encrypts one bit of the plaintext value. While bit-wise encryption can be used to achieve arbitrary-precision homomorphic comparison, such approaches often result in even slower performance and large communication costs when applied to EDB applications. Therefore, one of the primary motivations of this work is to design a homomorphic comparison scheme tailored for EDB queries that simultaneously achieves unbounded comparison depth, fast evaluation speed and arbitrary data precision.

**Remark:** We acknowledge the substantial body of work [32, 80, 81] on ciphertext-plaintext comparisons, which are also important in many applications [24, 32, 33, 83]. However, as illustrated in Table 1, comparisons between large numbers of ciphertexts constitute the absolute majority of the computations in evaluating composite SQL statements, and therefore is the main focus of this work.

### Table 2: Qualitative Comparisons Between Ciphertext-Ciphertext Homomorphic Comparison Algorithms

<table>
<thead>
<tr>
<th>Ciphertext type*</th>
<th>Encoding</th>
<th>RLWE Exponent</th>
<th>RLWE Slot</th>
<th>MRLWE Boolean</th>
<th>RLWE Boolean</th>
<th>MLWE Boolean</th>
<th>LWE Coeff</th>
<th>MLWE Coeff</th>
<th>LWE Coeff</th>
<th>MLWE Coeff</th>
<th>LWE Slot</th>
<th>MRLWE Slot</th>
<th>Ours</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit precision</td>
<td></td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>Arbitrary precision</td>
<td></td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td></td>
</tr>
</tbody>
</table>

* Ciphertext type refers to the ciphertext format defined in Section 2.1, including LWE, RLWE, Modular LWE (MLWE), and Modular RLWE (MRLWE).

**Table 2:** Qualitative Comparisons Between Ciphertext-Ciphertext Homomorphic Comparison Algorithms

In this section, we give an overview of the ciphertext types and key homomorphic operators in Section 2.1 and Section 2.2, respectively.

In terms of notations, we use bold lowercase letters (e.g., \( \hat{a} \)) for vectors, tilde lowercase letters (e.g., \( \tilde{a} \)) for polynomials, and bold uppercase letters (e.g., \( A \)) for matrices. Throughout this work, we use \( a \) to denote the security parameter and \( p/P \) for different plaintext moduli (generally \( P > p \)). \( q/Q \) indicate ciphertext moduli with varying sizes (generally \( Q > q \)), and \( n/N/N' \) specify lattice dimensions (generally \( N > n \)). \( \mathbb{Z}_q \) refers to the set of integers modulo \( q \). We define \( R_N \) and \( R_{N,Q} \) denote \( \mathbb{Z}[X]/(X^N + 1) \) and \( \mathbb{Z}[X]/(X^N + 1) \) mod \( Q \). For a comprehensive list of notations and terminologies, please refer to Appendix A.

### 2.1 FHE Ciphertexts Types

Similar to previous encrypted databases [16, 99], we adopt a cross-scheme approach that utilizes all of the BFV [18, 45], TFHE [29],
CKKS [26] and GSW [51] FHE schemes along with the state-of-the-art optimizations techniques [15, 17, 25, 30, 57–59, 61, 73, 78]. In what follows, we review the basics of the fundamental FHE ciphertext types, modular FHE schemes, and FHE operators.

2.1.1 Fundamental Ciphertext Types. We use three types of fundamental FHE ciphertexts: learning-with-errors (LWE) ciphertext LWE, ring-learning-with-errors (RLWE) ciphertext RLWE, and ring-Gentry-Sahai-Waters (GSW) ciphertext GSW.

- LWE\(^{n,q}_{a}(m)\): We define an LWE ciphertext that encrypts only a single message \(m \in \mathbb{Z}_p\) under the secret key \(s \in \mathbb{Z}_q^n\) as follows.

\[
\text{LWE}_a^{n,q}(m) = (b, a) = (-a, s \cdot m + e, a).
\]

where \(a \in \mathbb{Z}_q^n\) and \(e \in \mathbb{Z}_q\). \(\Delta = \left\lfloor \frac{q}{p} \right\rfloor\) is a scaling factor to protect the least significant bits of the message from the noises.

- RLWE\(^{N,Q}_{s}(m)\): An RLWE ciphertext is formulated as

\[
\text{RLWE}^{N,Q}_{s}(m) = (b, \tilde{a}) = (\tilde{t} \cdot s + \Delta \tilde{m} + \tilde{e}, \tilde{a}).
\]

for a polynomial of encoded messages \(\tilde{m} \in \mathbb{R}_{N,P}\) under the secret key \(s \in \mathbb{R}_{N,Q}\) and \(\Delta = \left\lfloor \frac{q}{p} \right\rfloor\).

- RGSW\(^{N,Q}_{s}(m)\): Given a decomposition size \(l\), the RGSW encryption of a message \(m \in \mathbb{Z}_p\) under the secret key \(s \in \mathbb{R}_{N,Q}\) is defined as RGSW\(^{N,Q}_{s}(m) = (b, A) \in \mathbb{Z}_{Q^2}^{2l}\).

The concrete constructions of RGSW can be found in [16, 51, 68, 82]. Here, we can simply consider an RGSW ciphertext as a tuple of two \(2l\)-degree RLWE ciphertexts.

It can be observed that all of the above types of ciphertexts contain intrinsic noises (e.g., \(e\) in LWE and \(\tilde{e}\) in RLWE) that are amplified by the homomorphic operators.

2.1.2 Modular Homomorphic Encryption. As mentioned, while fixed-size FHE ciphertexts can accelerate encrypted computations, such ciphertexts often result in degradation on the plaintext accuracy. Therefore, in this work, we make use of a particular variant of MFHE to solve the accuracy-performance dilemma. Specifically, we define the modular version of the LWE ciphertext as

\[
\text{LWE}_a^{n,q}(\hat{m}) = (\text{LWE}(m_0), \ldots, \text{LWE}(m_{i-1})),
\]

where \(\hat{m} = (m_0, m_1, \ldots, m_{i-1}) = \sum_{j=0}^{i-1} m_j b^j \in \mathbb{Z}_p\) for some large plaintext modulus \(p\) and \(m_j \in \mathbb{Z}_p\) for some small plaintext modulus \(p < P\). Essentially, \(LWE\) is a series of LWE ciphertexts, where each LWE encrypts a radix-\(b\) decomposed chunk of the large integer \(\hat{m}\). Similarly, we define the modular version of the RLWE ciphertext as

\[
\text{RLWE}^{N,Q}_{s}(\hat{m}) = (\text{RLWE}(\hat{m}_0), \ldots, \text{RLWE}(\hat{m}_{i-1})),
\]

where \(\hat{m}_j = \sum_{i=0}^{N-1} m_{i,j} b^j \in \mathbb{R}_{N,P}\) and \(\hat{m} = (\hat{m}_0, \hat{m}_1, \ldots, \hat{m}_{i-1}) = \sum_{i=0}^{N-1} \hat{m}_i b^i \in \mathbb{R}_{N,F}\) for some radix base \(b\). In other words, here, a large plaintext polynomial \(\hat{m} \in \mathbb{R}_P\) is cut into \(\omega\) chunks of \(\hat{m}_i \in \mathbb{R}_P\), where each \(\hat{m}_i\) is encrypted as an RLWE ciphertext RLWE\((\hat{m}_i)\).

Throughout this work, we use RLWE\((i)\) (resp. LWE\((i)\)) to refer to the \(i\)-th ciphertext RLWE\((\hat{m}_i)\) (resp. LWE\((m_i)\)) in RLWE\((\hat{m})\) (resp. LWE\((\hat{m})\)).

Remark: We note that the above definition is slightly different from the general modular FHE ciphertexts formulated in [14].

The main reason here is that the above modular ciphertext definitions are tailored for high-precision EDB operations, such as filter-aggregation, rather than general computations. In Section 4, we show how homomorphic functions can be efficiently applied over such modular ciphertexts.

2.2 Homomorphic Operators

Here, we explain the key homomorphic operators used throughout this work. We abbreviate the ciphertext notations to shorthands such as LWE\((m)\) and RLWE\((m)\) when the parameters are irrelevant from the discussion.

2.2.1 Homomorphic Arithmetic Operators. We primarily use homomorphic arithmetic operators to evaluate linear (i.e., polynomial) operations over RLWE ciphertexts. In particular, all arithmetic operators act over RLWE ciphertexts can be used in a single-instruction-multi-data (SIMD) manner, where one execution of an operator carries effects over a batch (usually all of the \(N\) plaintext elements in \(\hat{m}\) encrypted in RLWE\(^{N,Q}_{s}(\hat{m})\)).

- \(+\), \(-\), and \(\cdot\): We use standard ciphertext addition, subtraction, and multiplication operators over RLWE ciphertexts.

- \(\text{poly}(\text{RLWE}(\hat{m}))\): For any polynomial \(\text{poly}(x)\), \(\text{poly}(\text{RLWE}(\hat{m}))\) represents the homomorphic evaluation of \(\text{poly}\) over the input ciphertext RLWE\((\hat{m})\) encrypting \(\hat{m}\).

- \(\text{Automation}(\text{RLWE}(\hat{m}), b)\): For a given ciphertext RLWE\((\hat{m}|\tilde{X})\), Automation(RLWE\((\hat{m}|\tilde{X}), b)\) outputs a new ciphertext RLWE\(\text{out}(\hat{m}|\tilde{X})\), i.e., the coefficient of the plaintext polynomial is rearranged by the parameter \(b\).

- \(\text{ExternalProduct}(\text{RGSW}(d), \text{RLWE}(\hat{m}))\): ExternalProduct is a special type of homomorphic multiplication, where ExternalProduct\((\text{RGSW}(d), \text{RLWE}(\hat{m})) = \text{RLWE}(d \cdot \hat{m})\). In general, ExternalProduct tends to be faster and generates significantly less noise compared to a straightforward homomorphic multiplication. This is why we will use external products as the basic operator in ArcEDB.

More details on the exact cryptographic properties of the above operators can be found in [16, 18, 23, 29, 50].

2.2.2 Homomorphic Logic Operators. Different from homomorphic arithmetic operators, homomorphic logic operators are better at handling deep chains of non-polynomial functions. In this work, we mainly utilize the following three homomorphic logic operators.

- \(\text{CUX}(\text{RGSW}(t), \text{LWE}(a), \text{LWE}(b))\): For inputs \(\text{LWE}(a)\) and \(\text{LWE}(b)\), given a control signal \(\text{RGSW}(t)\) that encrypts a binary plaintext \(t \in \{0, 1\}\), \(\text{CUX}(\text{RGSW}(t), \text{LWE}(a), \text{LWE}(b))\) homomorphically computes \(t \cdot \text{LWE}(a) + \text{LWE}(b)\), i.e., the function selects \(\text{LWE}(a)\) if \(t = 1\) and \(\text{LWE}(b)\) if \(t = 0\).

- \(\text{BlindRotate}(\text{LWE}^{n,q}_{a}, \text{BK}, TV)\): BlindRotate takes as input an LWE ciphertext LWE\(^{n,q}_{a}\), a bootstrapping key BK, and a polynomial TV (i.e., the test vector in [29]), and generates an RLWE ciphertext RLWE\((X^{-p} TV)\), where \(p = \lceil 2n \cdot (b + \sum_{p=1}^{1} a_i) \rceil / q\).

- \(\text{HomGate}(\text{LWE}(m_0), \text{LWE}(m_1), \text{OP})\): Given two LWE ciphertexts \(\text{LWE}(m_0)\) and \(\text{LWE}(m_1)\) with a two-input logic gate OP, HomGate\((\text{LWE}(m_0), \text{LWE}(m_1), \text{OP})\) produces LWE\((\text{OP}(m_0, m_1))\). Here, OP includes logic operations such as AND, OR, NAND, etc.
To find more details on the SIMD homomorphic logic operators and other operators such as RLWETolWES, LWEtoRLWES and LWEtoRGSW, we refer the readers to the related literature [23, 30, 77].

2.2.3 Ciphertext Type Conversion. Converting between ciphertext types is the key design element in enabling consecutive SQL statements to be executed over encrypted data. Here, we summarize the conventional conversion algorithms proposed in [23, 29, 78].

- RLWETolWES(RLWE(\(\hat{m}\))): RLWETolWES converts RLWE\(_{N,Q}^{\pi,\omega}\) ciphertext to a set of N LWE\(_{a,q}^{\pi,\omega}\) ciphertexts. As defined in [29], RLWETolWES outputs N LWE ciphertexts \(ct = (ct_0, ct_1, ..., ct_{N-1})\) where \(ct_i\) encrypts the \(i\)-th plaintext coefficient of \(\hat{m}\).
- LWEtoRLWES(LWE\(_{N_a,q}^{\pi,\omega}\), ..., LWE\(_{N_{N-1},q}^{\pi,\omega}\)): LWEtoRLWES converts a set of N LWE\(_{a,q}^{\pi,\omega}\) ciphertexts to one RLWE\(_{N,Q}^{\pi,\omega}\) ciphertext, i.e., the inverse of RLWETolWES.
- LWEtoRGSW(LWE, BK): LWEtoRGSW converts an LWE\(_{a,q}^{\pi,\omega}\) ciphertext to an RGSW\(_{N,Q}^{\pi,\omega}\) ciphertext. LWEtoRGSW is generally used to convert an LWE ciphertext to an RGSW switching signal for the CMUX operator.

3 FRAMEWORK OVERVIEW

In this section, we first outline the overall ArcEDB framework in Section 3.1, and then discuss the data types and application-programming interfaces in Section 3.2 and Section 3.3.

3.1 System Workflow

Similar to prior works [16, 99], executing queries over ArcEDB consists of three primary steps: client data encryption, client query encryption, and server query evaluation. In what follows, we outline the main procedures for each of the steps, which are also illustrated in Figure 1.

1. **Client Data Encryption**: During this step, the main task involves homomorphically encrypting all the data tables in the database \(D\) by the client. Here, each encrypted table \(T\) of \(D\) will be encrypted utilizing the TableEncrypt function, producing the encrypted table \([T]\). More specifically, an encrypted Table \([T]\) comprises a number of encrypted data columns that can be categorized into two classes based on the attribute properties, namely, the filter attribute columns \((\text{Attr}^{\text{cmp}})\) and the aggregation attribute columns \((\text{Attr}^{\text{agg}})\).

2. **Client Query Encryption**: When querying the outsourced encrypted table \([T]\), the client initiates the QueryEncrypt function to encrypt the private data in the query. In this work, we consider the query \(Q\) follows a typical SQL SELECT form, where \(Q = \{P_0 \circ P_1 \circ ... \circ P_{|Q|-1}, \text{Attr}^{\text{agg}}\}\). Here, each \(P_i = (\text{Attr}^{\text{cmp}}, \text{cmp}, b_i)\) represents a predicate consisting of the \(i\)-th filtering attribute \(\text{Attr}^{\text{cmp}}\), the comparison operator \(\text{cmp}\), and a predicate value \(b_i\). Different predicates are concatenated with some logic function \(\circ\), which can be AND or OR. \(\text{Attr}^{\text{agg}}\) denotes the aggregation function to be applied on the aggregation attribute \(\text{Attr}^{\text{agg}}\). The QueryEncrypt function homomorphically encrypts the predicate values in each of \(P_i\), yielding the encrypted predicate \([P_i]\), ciphertexts. Subsequently, we obtain the encrypted query \([Q]\) formatted as \(([P_0]_0 \circ ... \circ [P_{|Q|-1}]_0, \text{Attr}^{\text{agg}})\). In the end, \([Q]\) is dispatched to the server.

3. **Server Query Evaluation**: Upon receiving \([Q]\), the server undertakes its evaluation using the Query function on the encrypted table \([T]\). The main computations involved in homomorphic query evaluations are the homomorphic filtering and homomorphic aggregation functions. First, in the filtering stage, each encrypted predicate \([P_i]\) is evaluated over the encrypted table using the FilterPred (more details in Section 3.3) function and produces the filtered result \([F]\), which is a set of ciphertexts encrypting 1 if the predicate \(P_i\) on such item is true and 0 if false. Next, after the predicate evaluation, the filtering results \([F]\) will be homomorphically chained together using the homomorphic logical operators (i.e., AND or OR) defined in Section 2.2 and produce a single filtering result \([F]\). Here, \([F]\) is an array of \([F]\) \(_{\text{row}}\) LWE ciphertexts encrypting either 1 (true) or 0 (false), indicating whether the \(i\)-th row of \(T\) is selected or not. Lastly, the aggregation functions \(\text{Agg}\) are executed over the aggregated \(\text{Attr}^{\text{agg}}\) column to produce the final result \([R]\), which can be either LWE or RLWE ciphertexts, depending on the aggregation function. Finally, the round of query evaluation finishes when the client decrypts \([R]\) that is returned from the server.

3.2 Data Encoding and Structure

In this section, we provide a deeper dive into the exact data types and structures used to encrypt the data tables in ArcEDB. We develop a layered approach to better decouple the high-level EDB data structures and low-level cryptographic primitives (as further illustrated in Figure A1). Below, we detail the concrete constructions for the three proposed layers: the homomorphic ciphertext layer (Section 3.2.1), the encrypted data type layer (Section 3.2.2), and the encrypted table structure layer (Section 3.2.3).

3.2.1 Homomorphic Ciphertext Layer. The core of ArcEDB is the set of modular homomorphic ciphertexts defined to better aid the evaluation of large-precision SQL queries. The main ciphertext types and plaintext encodings adopted in ArcEDB are as follows.

- **LWE**: The modular variant of the LWE ciphertext as defined in Section 2.1.2. LWE is mainly employed for encrypting Boolean values in ArcEDB. Meanwhile, LWE can also encrypt the intermediate results during the query evaluation.
- **Coefficient/Slot/Exponent-based RLWE**: The modular version of LWE ciphertext defined in Section 2.1.2. The plaintext encoding for RLWE is much more complex than that of LWE. In addition to the slot [101] and the coefficient [15, 61] encoding methods that are commonly used in existing FHE-based EDB frameworks, we also introduce a new exponent encoding approach in ArcEDB to handle homomorphic comparisons between high-precision data. Inspired by [32, 79, 81], we employ a specific mapping function \(\pi: \mathbb{Z} \rightarrow R\) with the property \(\pi(a) = X^a\) to embed the plaintext integer \(a \in \mathbb{Z}\) into a polynomial in \(R\). In particular, when encrypting a large-precision value \(\hat{a} = (a_0, a_1, \ldots, a_{|\hat{a}|-1})\), the modular exponent RLWE ciphertext is defined as:

\[
\text{RLWE}(\pi(\hat{a})) = (\text{RLWE}(X^{a_0}), \ldots, \text{RLWE}(X^{a_{|\hat{a}|-1}})).
\]
We point out that, while exponent encoding is useful at comparing large-precision data, it is not previously known how such encoding can be adopted in end-to-end SQL evaluation due to the ciphertext incompatibility issues. In Section 4, we further study the low-level characteristics of exponent encoding, and show designs of efficient ciphertext conversion algorithms to effectively make use of such encoding in FHE-based EDB.

- **Coefficient/Exponent-based \( \text{RGSW} \):** The modular version of the RGSW ciphertext defined in Section 2.1.2. In ArcEDB, RGSW mainly has two types of plaintext encodings: coefficient and exponent. Different from modular RLWE ciphertexts, modular RGSW ciphertexts encrypt plaintexts using negative exponents. In other words, when encrypting \( \hat{b} = \{b_0, b_1, \ldots, b_m\} \), the corresponding ciphertext is formulated as:

\[
\text{RGSW}(\pi(-\hat{b})) = (\text{RGSW}(X^{-b_0}), \ldots, \text{RGSW}(X^{-b_{m-1}}))
\]  

(6)

3.2.2 **Encrypted Data Type Layer.** Building upon the low-level FHE ciphertexts, we define four principal data types that form as the fundamental way of encrypted numerics, encrypted timestamps, encrypted strings, and encrypted Booleans.

- **Encrypted Numerics:** Numeric data types are found extensively throughout SQL queries to store numerical values. This includes both exact numerical types like integers (EINT, EBIGINT) and floating-point numbers (EFLOAT). Under the context of ArcEDB, a column data denoted as the coefficients of a polynomial \( \hat{a} \) is of a numeric type (e.g., EINT) when \( \hat{a} \) is encrypted into any one of the modular RLWE ciphertext variants depending on its encoding method: coefficient (RLWE(\( \hat{a} \))), slot (RLWE(\( E(\hat{a}) \))), or exponent (RLWE(\( \pi(\hat{a}) \))). On the other hand, approximate numbers are first converted to fixed-point integer representations and then encrypted just as exact numerics.

- **Encrypted Timestamps:** Time-related SQL data types, such as ETIMESTAMP and EDATE, are designed to represent time values accurately. These types are translated into high-precision integers and then encrypted as modular RLWE ciphertexts. While existing approaches [16, 99] face precision limitations and do not natively support time data types, we are able to handle time data with arbitrary-precision RLWE ciphertexts, enhancing the functionality of encrypted databases. In general, timestamps are encrypted using exponent encodings, as such attribute columns are mostly used in comparisons instead of aggregations.

- **Encrypted Strings:** Encrypted string data types, such as CHAR and TEXT, are used to store textual values. Similar to time data types, string values in ArcEDB need to be encoded to integers with extremely large precision such that arbitrarily long strings can be correctly decoded after query evaluation. Encrypted string types in ArcEDB are encrypted as modular RLWE ciphertexts.

- **Encrypted Booleans:** Binary data types, including EBOOL, EBIGINT, and EVARBINARY, represent Boolean data values. Since Boolean data can only have up to one-bit precision, we do not need modular ciphertexts, and can directly encrypt the binary values into (R)LWE ciphertexts.

3.2.3 **Encrypted Table Structure Layer.** Based on the above two layers of abstractions, we can define table-level data structures for ArcEDB. As mentioned in Section 3.1, by default, the entire data table \( \mathcal{T} \) is encrypted column-by-column using (modular) RLWE ciphertexts, where each column is divided into sets of \( N \)-sized data chunks. Each chunk is encoded into degree-\( N \) plaintext polynomials \( \hat{m}_i \). However, depending on the actual applications, we can have the following three concrete types of table-level data structures.

- **Filtering Columns:** For attributes that are mainly used in filtering statements (e.g., TIMESTAMP, TEXT), each column \( \hat{m} \) is encrypted into modular RLWE ciphertexts with exponent encoding, i.e., RLWE(\( \pi(\hat{m}) \)).

- **Aggregation Columns:** Since coefficient encoding is more efficient in performance homomorphic aggregations, aggregation attributes (e.g., Salary), are by default encrypted into RLWE(\( \hat{m} \)).

- **Sorting Columns:** We point out that, when we need to synchronize the order of a particular attribute column to other columns, it is much more efficient to encrypt such column using RGSW ciphertexts in a bit-decomposed manner. The formal constructions can be found in Section 5.2 and Appendix C.

**Remark:** It is noted that some attribute columns can be used as both filtering and aggregation columns. The straightforward approach is to encrypt them in both forms. To avoid excessive encryption burdens on the client side, ArcEDB offers ciphertext format...
conversion methods in Section 5.1 to convert exponent modular RLWE ciphertext into coefficient modular RLWE ciphertext.

3.3 ArcEDB API

Utilizing the rich class of data types, we specify the APIs of ArcEDB for both client and server described as follows.

- TableEncrypt(\(T\)) \(\rightarrow\) [\(T\)]: Encrypts the database table \(T\) and yields the encrypted table \([T]\). This process involves encrypting both the filter and the aggregation columns in \(T\).
- QueryEncrypt(\(Q\)) \(\rightarrow\) [\(Q\)]: Encrypts a SQL query \(Q = (P_1 \land P_2 \land \ldots \land P_{|Q|}; \text{Attr}_{\text{agg}}, \text{Agg})\) into an encrypted query \([Q] = (\{P\}_1 \land \{P\}_2 \land \ldots \land \{P\}_{|Q|}; \text{Attr}_{\text{agg}}, \text{Agg})\). In ArcEDB, each encrypted predicate \([P\] is always encrypted in the form of a negative exponent modular RGSW ciphertexts \(\text{RGSW}(\pi(-b_i))\), where \(b_i\) is the corresponding predicate value \(b_i\).
- Query([\(Q\]), [\(T\)]) \(\rightarrow\) [\(R\)]: Evaluates an encrypted query \([Q] = (\{P\}_1 \land \{P\}_2 \land \ldots \land \{P\}_{|Q|}; \text{Attr}_{\text{agg}}, \text{Agg})\) on the encrypted table \([T]\). Essentially, the querying process is to invoke the FilterPred and Aggregation APIs consecutively, detailed as follows.
  - FilterPred([\(P\]), [\(T\)]) \(\rightarrow\) [\(F\)]: Filters an encrypted predicate \([P\] on the encrypted table \([T]\) and outputs \([T]_{\text{row}} = \text{LWE}_{\text{row}} \ldots \text{LWE}_{\text{row}-1}\). The function utilizes advanced homomorphic comparison operators \(\text{HCMP}_{\text{ arb}}, \text{ArbHCMP},\) and \(\text{SIMD}_{\text{ arb}}\), detailed in Section 4, to enhance the precision and efficiency of encrypted predicate evaluation.
  - Aggregation([\(F\]), \text{Attr}_{\text{agg}}, \text{Agg}) \(\rightarrow\) [\(F\)]: Aggregates a specified column \text{Attr}_{\text{agg}} by a function \text{Agg} on the encrypted table \([T]\) and the encrypted filter result \([F]\).

ArcEDB supports both arithmetic aggregation (such as SUM and COUNT) and logic aggregation (such as \(\text{MIN}, \text{MAX}\) and \(\text{Top-k}\)) functions. The detail is constructed in Section 5.

- \(\text{GROUP BY}(\{Q\}, \text{Attr}_{\text{Grpby}}, \{T\}) \rightarrow \{\{R\}\}\): Evaluates the SELECT statement query \([Q]\) with the \text{GROUP BY} attribute \text{Attr}_{\text{Grpby}} on the encrypted table \([T]\). Similar to [16, 99], to implement \text{GROUP BY}, we issues multiple copies of the query \([Q]\), each augmented with an additional equality test for the group attribute.
- \(\text{ORDER BY}(\text{Attr}_{\text{sort}}, \{T\}) \rightarrow \{T^\prime\}\): Evaluates the ORDER BY statement on the encrypted table \([T]\) based on a target attribute \text{Attr}_{\text{sort}}. \text{ORDER BY} is essentially a series of homomorphic comparisons appended by data swapping based on comparison results. Note that existing approaches [16] only focus on sorting the \text{Attr}_{\text{sort}} column alone and do not account for the synchronization of other columns based on the sorted column. ArcEDB introduces a novel homomorphic \text{ORDER BY} technique built upon our exponent-based encoding, enabling efficient synchronized sorting across multiple columns. More details can be seen in Section 5.2.

3.4 Threat Model and Security Guarantees

The security objective of ArcEDB is to protect the outsourced database \(D\) owned by the client \(C\) against a semi-honest server \(S\), aligning with previous works [32, 56, 72, 97, 99]. The concrete public and private data from the scope of the server is summarized as follows.

**Public Data:**
- \(\{T\}_{\text{row}}, \{\text{T}\}_{\text{col}}\): the number of rows as well as the number of columns in the table \(T \in D\).

**Private Data:**
- \(\{Q\}\): the number of filtering predicates in a SQL query \(Q\).
- \text{Attr}, \text{Attr}: The attribute label (e.g., gender, date) and the range of the attribute (e.g., [Gender] = 2).
- \(\odot\): The concatenating logic function (e.g., \&\&, ||) between the filtering predicates in a SQL query.
- \text{Agg}: the aggregation functions (e.g., SUM, MIN) in \(Q\).

**Security of ArcEDB:** Since all private data is encrypted into FHE ciphertext. The security of ArcEDB is deeply rooted in the FHE schemes it employs. Traditional FHE schemes, such as BFV [18], CKKS [26], and TFHE [29], all guarantee security under chosen plaintext attacks, which inherently provides ArcEDB with a semi-honest security on the outsourced database \(D\). Under the premise of circular security of FHE [19], switching between distinct ciphertext formats maintains the the overall semi-honest security of the protocol. It is emphasized that ArcEDB protects not only the data items but also the intermediate computation results, which provides security against access patterns and volume leakage attacks [67].

4 (AMORTIZED) HOMOMORPHIC FILTERING

In this section, we delve into the methodology for efficiently filtering large-size DB columns in ArcEDB. Based on the fact that filtering predicates are essentially a series of comparisons, our focus is on enhancing the efficiency and precision of comparisons of ciphertexts. Our approach contains three pivotal components:

- In Section 4.1, we introduce a fast homomorphic comparison operator, HCMP, utilizing exponent encoding ciphertext. This operator is optimized for swift homomorphic filtering within a constrained precision range.
- In Section 4.2, we extend the precision of HCMP by leveraging modular homomorphic ciphertexts defined in Section 2.1.2. We
observe that comparisons between two such ciphertexts can be efficiently executed by comparing individual chunks and integrating the results using a homomorphic multiplexer (MUX). Consequently, we propose a novel homomorphic MUX operator GateMUX, which is finely tuned to work in conjunction with the outputs of HCMP. Through the synergistic use of HCMP and GateMUX, we devise an innovative homomorphic comparison algorithm ArbHCMP, enabling arbitrary precision filtering in EDB systems.

In Section 4.3, we aim to further accelerate homomorphic filtering speed based on the batch bootstrapping technique proposed in [77]. We introduce an innovative amortized homomorphic MUX operator SIMDCMUX, capable of evaluating multiple MUX operations with a single round of homomorphic computation. Utilizing the HCMP and SIMDCMUX operator, we propose the amortized arbitrary-precision homomorphic comparison operator SIMDArbHCMP to efficiently filter batches of rows to significantly boost the computational efficiency and data precision in large-scale EDB systems.

### 4.1 Limited Precision Filtering

Here, we outline a fast homomorphic comparison algorithm HCMP to filter items with limited precision.

#### 4.1.1 Exponent Encoding based Comparison

Before delving into HCMP, we first discuss the exponent encoding ciphertext format, which is the key inspiration in instantiating HCMP. We first point out that, existing homomorphic filtering algorithms [16, 99] following the PBS procedure [78] proposed in [30] may not be well-suited for EDB systems due to their relatively low evaluation speed. For instance, HE3DB [16] relies on the iterative execution of PBS to perform comparisons between queried attributes and DB items. However, the speed of the PBS algorithm is notably slow, and dominates the computation time (93% as shown in Table 1) in HE3DB [16].

To overcome this issue, we adopt the exponent encoding strategy as introduced in Section 3.2. Leveraging the exponent encoding, ciphertext comparisons can be evaluated through simple multiplications, significantly faster than PBS-based approaches. Specifically, given two log N-bit integers a and b, the comparison between a and b can be expressed as:

$$\tilde{C} = \tilde{TV} \cdot \pi(a) \cdot \pi(-b) \mod (X^N + 1),$$

(7)

where $\tilde{TV} = 1 + X + \ldots + X^{N-1}$ and $\pi(a) = X^a$. The constant term of $\tilde{C}$ depends on $\pi(a) \cdot \pi(-b) = X^{a-b}$. If $a \leq b$, the polynomial $\tilde{TV}$ shifts right, making the zeroth coefficient $\tilde{C}$ equal to 1. Conversely, if $a > b$, $\tilde{TV}$ shifts left, resulting in the zeroth coefficient of $\tilde{C}$ being $-1$. While the above formulation is an illustration using plaintext data, we can see that comparisons using exponent encoding are simply shifting polynomial coefficients. Hence, in the following sections, we show that shift-based comparisons can be implemented extremely fast over FHE ciphertexts, especially when compared to PBS-based approaches [16, 99].

#### 4.1.2 Homomorphic Comparison Operator HCMP

Now, we describe the homomorphic comparison operator HCMP utilizing the exponent encoding. In contrast to the existing approaches [32, 79, 81], we reformulate one of the comparison inputs as an RLWE ciphertext representing a database item and another input as an RGSW ciphertext representing the queried attribute value. The above modification is tailored for the EDB systems where the number of data items (i.e., the size of the entire database) are in general orders of magnitude larger than the number of queried attribute values (i.e., usually less than a dozen).

The detailed algorithm for HCMP is provided in Algorithm 1. For illustrative purposes, we focus on the "less than or equal to" ($\leq$) comparison to walk through Algorithm 1. On Line 1, the initial step involves an ExternalProduct between RLEWE($X^a$) and RGSW($X^{-b}$), yielding $ct = \text{RLEWE}(X^{a-b})$. Subsequently, on Line 2–3, we construct a test vector $TV = \mu + X + \ldots + X^{N-1}$ where $\mu$ denotes the inverse of 2 modulo $p$. In the following phase (Line 4), we evaluate $TV \cdot ct + \mu$ to obtain RLEWE($X^{a-b} \cdot TV + \mu$). It is observed that the constant term of RLEWE($X^{a-b} \cdot TV + \mu$) results in either $\mu + \mu = 1$ or $-\mu + \mu = 0$, depending on whether $a \leq b$. Lastly, on Line 5, we apply the RLEWEi function and extract the zeroth coefficient of RLEWE($X^{a-b} \cdot TV + \mu$) to produce the output LWE ciphertext $ctQ$. The ciphertext $ctQ$ encrypts either 1 or 0, indicating whether the data item $a$ satisfies the predicate condition. For other comparison operators like $>$, $\geq$, $<$, $\leq$, and $\neq$, we only need to slightly modify the test vector $TV$. Detailed explanations for these modifications are provided in Table A3.

### Table 3: Summary of Operation Costs in ArcEDB for Each Homomorphic SQL Statement.

<table>
<thead>
<tr>
<th>SQL statement</th>
<th>#ArbHCMP</th>
<th>#GateGate</th>
<th>#HCMP</th>
<th>#LWEtoRGSW</th>
<th>#LWEtoLWE</th>
<th>#+</th>
<th>-</th>
<th>#Automorphism</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT</td>
<td>$</td>
<td>T</td>
<td>_{low} \cdot</td>
<td>Q</td>
<td>$</td>
<td>$</td>
<td>T</td>
<td>_{low} \cdot (</td>
</tr>
<tr>
<td>SUM</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>COUNT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MIN/MAX</td>
<td>$</td>
<td>T</td>
<td>_{low} - 1$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>GROUP BY</td>
<td>$</td>
<td>T</td>
<td>_{low} \cdot</td>
<td>Attr^{(p)}</td>
<td>$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ORDER BY</td>
<td>$</td>
<td>T</td>
<td>_{low} -</td>
<td>T</td>
<td>_{row}$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>JOIN</td>
<td>$</td>
<td>T</td>
<td>_{low} \cdot</td>
<td>T</td>
<td>_{low}$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

| $|Q|$, $|T|_{low}$, $|Attr^{(p)}|$ are publicly known to the server. |
Algorithm 2: Homomorphic MUX operator GateMUX

**Input:** Three LWE ciphertexts $ct_d = LWE^n(d)$, $ct_a = LWE^n(a)$, $ct_b = LWE^n(b)$, where $d, a, b \in \{0, 1\}$ and plain modulus $p$.

**Input:** A bootstrapping key $BK$.

**Output:** An LWE ciphertext $ct_O = LWE(d \cdot a : b)$.

1. $ct_d = ct_d + 4ct_a + 2ct_b$, scale $\leftarrow \lfloor p/16 \rfloor$, offset $\leftarrow \lfloor q/32 \rfloor$
2. $ct_{linear} \leftarrow scale \cdot ct_d + offset$
3. $T \leftarrow \{0, 1, 0, 0, 1, 1, 1\}$, $TV \leftarrow \sum_{i=0}^{7} T[y_i] \cdot X^{i + y_i \bmod 8}$
4. $ct_{rot} \leftarrow \text{BlindRotate}(ct_{linear}, BK, TV)$
5. $ct_{O} \leftarrow \text{RLWEToLWEs}(ct_{rot})[0]$

**Return:** $ct_O = LWE(d \cdot a : b)$

Algorithm 3: Arbitrary-precision homomorphic comparison operator ArbHCMP

**Input:** A modular RLWE ciphertext $ct_a = RLWE(\pi(\hat{a}))$, where $\hat{a} \equiv \sum_{i=0}^{\omega-1} a_i \cdot N^i$, a modular RGSW ciphertext $ct_b = \text{RGSW}(\pi(-\hat{b}))$ where $\hat{b} \equiv \sum_{i=0}^{\omega-1} b_i \cdot N^i$.

**Input:** A bootstrapping key $BK$, modular ciphertext size $\omega$.

**Output:** An LWE ciphertext $ct_O = LWE(\hat{a} \leq \hat{b})$.

1. $\omega \leftarrow ct_a.size()$
2. if $\omega = 1$ then
   3. $ct_O \leftarrow \text{HCMP}_{\leq}(ct_a[0], ct_b[0])$
else
   5. $ct_d \leftarrow \text{HCMP}_{\leq}(ct_a[0] - 1, ct_a[0] - 1)$
   6. $ct_o \leftarrow \text{ArbHCMP}_{\leq}(ct_d[0] - \omega - 1, ct_b[0] - \omega - 1)$, BK
   7. $ct_{o-1} \leftarrow \text{HCMP}_{\leq}(ct_d[0] - \omega - 1, ct_b[0] - \omega - 1)$
   8. $ct_O \leftarrow \text{GateMUX}(ct_o, ct_o, ct_{o-1}, BK)$

**Return:** $ct_O = LWE(c)$

4.1.3 Limitation of the HCMP. Unfortunately, HCMP suffers from constrained precision limited to log $N$ bits, where $N$ is the encryption parameter for the RLWE ciphertext. While the straightforward way of increasing precision is to enlarge $N$, larger $N$ leads to significantly slower computations.

4.2 Arbitrary Precision Filtering

To enhance the precision of HCMP, we leverage modular homomorphic encryption schemes defined in Section 2.1.2 and devise a new selector operator GateMUX to construct the arbitrary-precision comparison operator ArbHCMP.

We first discuss how to evaluate comparison on modular ciphertexts. As mentioned in Section 3.3, in ArcEDB, we split a large precision $\eta$-bit integer into a sequence of $\omega$ chunks of lower precision log $N$-bit integers, and encryps the integer chunk by chunk. The key insight is that, the comparison between two ciphertext chunks can be conducted individually for each pair of chunks. The results of these individual comparisons can be linked by a multiplexer (MUX) operation to derive the final result. For instance, for two 2 log $N$ bit integer $a = a_1 \cdot N + a_0$ and $b = b_1 \cdot N + b_0$ where $a_1, a_0, b_1, b_0 \in \mathbb{Z}_N$. The expression $a \leq b$ is equivalent to

$$a_1 = b_1 \iff a_0 \leq b_0 : a_1 \leq b_1$$

(8)

Since we can evaluate $a_1 = b_1, a_0 \leq b_0$, and $a_1 \leq b_1$ separately using the HCMP operator, the only piece left is to perform a homomorphic MUX operation to combine individual comparison results.

While there exists prior works [79] for evaluating homomorphic MUX operation on individual comparison results, such techniques work poorly when directly applied to evaluating arbitrary-precision homomorphic comparison due to the low evaluation speed. The fundamental issue lies in the incompatibility of the output from individual comparison results with subsequent combinations. For instance, the method [79] produce an RLWE ciphertext as the individual comparison result and combine these results using leveled homomorphic ciphertext addition and multiplication. However, their output RLWE ciphertext format does not inherently support ciphertext multiplication, incurring additional costs to convert the ciphertext to be compatible with multiplication. Similarly, the comparison result produced by [32] remains incompatible with the CMUX operator and necessitates the use of LWEtoRGSW to bridge the gap, which induces a significant amount of performance overheads.

To avoid the above incompatibility issues, we propose a homomorphic MUX operator GateMUX that takes as input exactly the output of our proposed HCMP. Our key insight is to generate the LWE ciphertext as the individual comparison result (as illustrated in Line 5 in Algorithm 1) and conducting the MUX operation as a three-input-one-output homomorphic gate. By leveraging the inherent capability of LWE ciphertext in performing homomorphic ciphertext evaluations, we can complete the MUX operation with a single programmable bootstrapping.

The main procedure of GateMUX basically follows the HomGate defined in Section 2.2 but with some crucial modiﬁcations. As presented in Algorithm 2, given three LWE ciphertexts $ct_d = LWE^n(d)$, $ct_a = LWE^n(a)$, $ct_b = LWE^n(b)$, where $d, a, b \in \{0, 1\}$. On Line 1–3, we follow the general procedure of HomGate which performs a linear combination of three input ciphertexts and results in $ct_{linear}$. Next on Line 4–5, we design a test polynomial speciﬁcally for the MUX function $\hat{d} \cdot a : b$. On Line 6–7, we apply BlindRotate and LWEtoLWEs to output the LWE ciphertext $ct_O$ encrypting $\hat{d} \cdot a : b$.

Based on the homomorphic comparison operator HCMP and homomorphic MUX algorithm GateMUX derived above, we can finally carry out arbitrary-precision comparisons between the queried attribute and DB items. For $\hat{a}$ and $\hat{b}$ two $\eta$-bit integers, let $\omega = \lfloor \eta/N \rfloor$, we have that $\hat{b} = \{b_0, ..., b_{\omega-1}\} = \sum_{i=0}^{\omega-1} b_i \cdot N^i$ is the queried predicate value and $\hat{a} = \{a_0, ..., a_{\omega-1}\} = \sum_{i=0}^{\omega-1} a_i \cdot N^i$ is one of the item in the Attr rop mp column. As shown in Algorithm 3, let $ct_a$ be the modular RLWE ciphertext that encrypts the $\omega \log N$-bit input $\hat{a}$ with $RLWE(\pi(\hat{a})) = RLWE(X^{a_0}), ..., RLWE(X^{a_{\omega-1}})$ and $ct_b$ be the modular RGSW ciphertext that encrypts the $\omega \log N$-bit queried attribute value $\hat{b}$ with $RGSW(\pi(-\hat{b})) = (\text{RGSW}(X^{-b_0}), ..., \text{RGSW}(X^{-b_{\omega-1}}))$. Suppose that cmp is the type of comparison to be performed (where cmp can be one of the $\leq, \leq, \geq, \neq, \#$ operators). The ArbHCMP outputs an LWE ciphertext encrypting 1 if the predicate $\hat{a}$ cmp $\hat{b}$ is true and 0 if the predicate is false. We take $\leq$ as an example to go through...
While ArbHCMP we introduce a new amortized homomorphic comparison operator \( \hat{\omega} \) we transform the expression

\[
\text{Algorithm 4: Amortized arbitrary-precision homomorphic comparison operator SIMD ArbHCMP}_\leq
\]

**Input**: N modular RLWE ciphertexts \( c_{A} = (RLWE(\pi(a_0)), RLWE(\pi(a_1)), ..., RLWE(\pi(a_{N-1})) \)

for \( \hat{\omega} \) a modular RGSW ciphertext \( ct_{p} = \text{RGSW}(\pi(-\hat{b})) \).

**Output**: N LWE ciphertexts \( \text{ct}_{\omega} = (\text{LWE}(\hat{a_0} \leq \hat{b}), \text{LWE}(\hat{a_1} \leq \hat{b}), ..., \text{LWE}(\hat{a}_{N-1} \leq \hat{b})) \).

1. \( \omega \leftarrow \text{ct}_{A}[0].\text{size()} \)
2. If \( \omega = 1 \) then
   3. For \( i = 0 \) to \( N - 1 \) do
      4. \( \text{ct}_{A}[i] \leftarrow \text{HCM}_\leq (\text{ct}_{A}[i][0], \text{ct}_{p}[0]) \)
   5. Else
      6. For \( i = 0 \) to \( N - 1 \) do
         7. \( \text{ct}_{A}[i] \leftarrow \text{HCM}_\leq (\text{ct}_{A}[i][0 - 1], \text{ct}_{p}[0 - 1]) \)
         8. \( \text{ct}_{p} = \text{SIMD ArbHCMP}_\leq (\text{ct}_{A}[0 : N][0 : 0 - 1], \text{ct}_{p}[0 : 0 - 1], \text{BTK}, \text{KSK}) \)
      9. \( \text{ct}_{\omega} = \text{SIMD CMUX}(\text{ct}_{A}, \text{ct}_{p}, \text{ct}_{\omega - 1}, \text{BTK}, \text{KSK}) \)

Return: \( \text{ct}_{\omega} \)

the detailed procedures for ArbHCMP \( _\leq \) in Algorithm 3. First, when \( \omega = 1 \), we can directly apply HCM \( _\leq \) on \( c_{A}[0] \) and \( c_{p}[0] \) and get the comparison result (Line 2). Otherwise when \( \omega \geq 2 \), supposeing ArbHCMP \( _\leq \) can perform \( (\omega - 1) \cdot \log N \)-bit precision comparison, we transform the expression \( \tilde{a} \leq \tilde{b} \) to the following equation

\[ a_{0 - 1} = b_{0 - 1} \land (a_{0}, ..., a_{\omega - 2}) \leq (b_{0}, ..., b_{\omega - 2}) : a_{\omega - 1} \leq b_{\omega - 1} (9) \]

Thus, on Line 5 – 7, we individually evaluate the comparisons in Equation (9) utilizing HCM \( _\leq \) operator and \( (\omega - 1) \cdot \log N \)-bit precision ArbHCMP \( _\leq \) operator to obtain three LWE ciphertexts \( ct_{A}, ct_{p}, ct_{\omega - 1} \) encrypting \( a_{0 - 1} = b_{0 - 1}, (a_{0}, ..., a_{\omega - 2}) \leq (b_{0}, ..., b_{\omega - 2}), \text{and } a_{\omega - 1} \leq b_{\omega - 1} \). Finally, on Line 8, we perform the MXU operation on \( ct_{A}, ct_{p}, ct_{\omega - 1} \) based on the GateMUX to get the final LWE ciphertext result encrypting \( \hat{a} \leq \hat{b} \). By recursively evaluating \( a_{1} \leq b_{1} \) using HCM and combining the comparison of the result by GateMUX without extra conversion, we can construct ArbHCMP \( _\leq \) for efficient arbitrary precision homomorphic filtering.

### 4.3 Amortized Arbitrary Precision Filtering

While ArbHCMP provides the capability for arbitrary precision homomorphic filtering, the operator can only filter one single DB item per comparison, which can still be too slow when dealing with large databases. Since EDB filtering often involves comparing one queried attribute against all items in some DB columns, a promising strategy to ease the computation burdens is to amortize costs by simultaneously filtering a batch of DB items. In this section, we introduce a new amortized homomorphic comparison operator SIMD ArbHCMP customized for batched filtering.

As indicated in Equation (9), arbitrary-precision comparison is composed of the evaluation of individual lower-precision comparisons and a number of MUX operations between the comparison results. Since we can use the same lightweight HCM building block to construct SIMD ArbHCMP, we only need to design a new homomorphic MUX algorithm in a SIMD manner, i.e., the SIMD CMUX operator, to accelerate homomorphic comparisons over large-size DB columns. The main procedure of SIMD CMUX follows the amortized bootstrapping technique proposed in [77], which homomorphically decrypts multiple LWE ciphertexts into a single LWE ciphertext and applies the specific MUX polynomial to all of the plaintext slots in the LWE ciphertext. In contrast to the GateMUX operator, which conducts a single MUX operation on three-input LWE ciphertexts, SIMD CMUX is able to perform N MUX operations simultaneously on 3N LWE ciphertexts. Due to the space limitation, the concrete constructions for SIMD CMUX is depicted in Algorithm 6.

Based on the amortized homomorphic MUX operator SIMD CMUX devised above, we can easily construct amortized arbitrary-precision homomorphic comparisons between the queried attributes and a lot of DB items. As shown in Algorithm 4, let \( \text{ct}_{A} = (RLWE(\pi(a_0)), RLWE(\pi(a_1)), ..., RLWE(\pi(a_{N-1})) \) be the L modular RLWE ciphertexts encrypt \( \omega \cdot \log N \)-bit integer item \( a_0, a_1, ..., a_{N-1} \) and \( ct_{p} \) be the modular RGSW ciphertext encrypts \( \omega \cdot \log N \)-bit query attribute \( b \) with \( ct_{p} = \text{RGSW}(\pi(-b)) \). Suppose that \( cmp \) is the type of comparison to be performed. The SIMD ArbHCMP outputs N LWE ciphertexts encrypting 1 if the predicate \( \hat{a}_{i} cmp \hat{b} \) is true and 0 if \( \hat{a}_{i} cmp \hat{b} \) is false. We take \( \leq \) as an example and summarize the exact arithmetic procedure for SIMD ArbHCMP \( _\leq \) in Algorithm 4. The main process is similar to Algorithm 3 but change the Line 6 in Algorithm 3 with SIMD ArbHCMP operator to combine the comparison result from all N DB items simultaneously.

### 5 COMPLEX HOMOMORPHIC AGGREGATION

After the homomorphic filtering, ArcEDB obtains the encrypted filtered result \( |F| \), which encompass \( |F|_{\text{row}} \) LWE ciphertexts encrypt either 1 or 0, indicating the selection status of these rows. The subsequent phase is to execute various aggregation functions over the filtered rows. In this section, we explain the mechanisms of both arithmetic and logic aggregations based on the results of homomorphic filtering.

#### 5.1 Homomorphic Arithmetic Aggregation

In this section, we present how to perform arithmetic aggregation such as SUM, COUNT on the filtered rows. Before arithmetic aggregation, \( |F|_{\text{row}} \) LWE ciphertexts encrypting the filtered results must first be packed into an LWE ciphertext \( |F|_{\text{row}} \) through the LWE to RLWE operator. This step is crucial for enabling rapid arithmetic aggregations on the RLWE ciphertexts. After that, the COUNT function is computed as an inner product between \( |F|_{\text{row}} \) and a vector \( I = (1, 1, \ldots, 1) \). Similarly, the SUM function involves an inner product between \( |F|_{\text{row}} \) and the to-be-aggregated column ciphertext Attr\(^{\text{row}} \). The homomorphic inner product is feasible with ciphertexts in either slot \([66, 78]\) or coefficient \([61, 99]\) formats.

However, as discussed Section 4, ArcEDB utilizes the exponential encoding method for achieving low-latency homomorphic filtering. Unfortunately, exponent encoding ciphertext is not inherently compatible with the latter homomorphic inner product for arithmetic aggregation. A straightforward solution is
Algorithm 5: Exponent-to-coefficient ciphertext conversion \texttt{ExpToBase}

\begin{algorithm}
\KwInput{L modular RLWE ciphertexts $ct_t = (RLWE(\pi(a_0)), RLWE(\pi(a_1)), \ldots, RLWE(\pi(a_{L-1}))$ where $a_i = \sum_{j=0}^{\omega-1} a_{ij} \beta^j$.}
\KwInput{Modular RLWE ciphertext dimension $N$.}
\KwOutput{An RLWE ciphertext $ct_O = RLWE(\hat{a})$, where $\hat{a} = \sum_{i=0}^{\omega-1} \beta^i \sum_{j=0}^{L-1} a_{ij} X^j$.}

1. Initialize $TV = 0 + X + 2X^2 + \ldots + (N-1)X^{N-1}$
2. \For{$i = 0$ \KwTo $L-1$}{
3. \For{$j = 0$ \KwTo $\omega - 1$}{
4. $ct_{ij} \leftarrow$ Automorphism($ct_t[i][j], 2N - 1$)
5. $ct_{ij} \leftarrow TV \cdot ct_{ij}$
6. $LWE_{ij} \leftarrow RLWE(CT(ct_{ij}[0])$}
7. \For{$j = 0$ \KwTo $\omega - 1$}{
8. $ct_j \leftarrow RLWE(LWE_{0,j}, LWE_{1,j}, \ldots, LWE_{L-1,j})$
9. $ct_O \leftarrow (ct_0, ct_1, \ldots, ct_{\omega-1})$
\Return{$ct_O = RLWE(\hat{a})$}
\end{algorithm}


to involve the client transmitting ciphertexts in both the exponent and coefficient encoding formats, but this will increase the client’s computational workload. Therefore, we provide a choice to transform the exponent encoding ciphertexts into the coefficient encoding ciphertexts on the server side. The algorithm is detailed in Algorithm 5. Given $L$ modular RLWE ciphertexts $ct_t = (RLWE(\pi(a_0)), RLWE(\pi(a_1)), \ldots, RLWE(\pi(a_{L-1}))$ where $a_i = \sum_{j=0}^{\omega-1} a_{ij} \beta^j$. The conversion process begins by performing the automorphism (Line 4) $X \mapsto X^{2N-1}$ on the exponent encoding ciphertext to obtain $RLWE(\pi(\hat{a})) = (RLWE(X^{-a_0}), \ldots, RLWE(X^{-a_{\omega-1}}))$. Subsequently, a plaintext multiplication (Line 5) is conducted between the test polynomial $TV = 0 + X + 2X^2 + \ldots + (N-1)X^{N-1}$ and $RLWE(\pi(\hat{a}))$, resulting in $(RLWE(X^{-a_0} \cdot TV), \ldots, RLWE(X^{-a_{\omega-1}} \cdot TV))$. After extracting the zero coefficient of each RLWE ciphertext (Line 6) in $RLWE(\pi(\hat{a}))$, we will get $LWE_{ij}$ encrypting $\hat{a}_i$. The final step (Line 8) is to pack the $L$ modular $LWE_{0,0}, LWE_{1,0}, \ldots, LWE_{L-1,0}$ to a modular RLWE ciphertext $ct_O$ encrypting $\hat{a} = \sum_{i=0}^{\omega-1} \beta^i \sum_{j=0}^{L-1} a_{ij} X^j$. This process can be carried out in the offline stage on the server side, and its cost is relatively minor compared to the substantial efficiency gains from utilizing the exponent encoding format in the filtering stage.

### 5.2 Homomorphic Logic Aggregation

While existing FHE-based EDBs, such as [16], can implement logic aggregations like MIN, MAX and ORDER BY, a critical challenge persists in the synchronization of the order in each column across other columns. For example, in practical SQL usage, the ORDER BY statement requires not only sorting the specified column but also ensuring the synchronization of this order across all other table columns. To overcome this challenge, we propose new HomSort and SortSynchronize algorithms for fast sorting and synchronization.

Here, we provide a toy example of the HomSort algorithm evaluating $L = 4$ rows columns in Figure 3, and defer further details in Appendix C.3. Our HomSort algorithm works in two-step process: i) computing a position index array for the to-be-sorted column, and ii) utilizing this index to guide the synchronization of other columns. In Figure 3, we use a plaintext example to demonstrate how the homomorphic sorting and synchronizing algorithm works for a to-be-sorted column $w = [5,3,7,6]$. The conversion process begins by performing the automorphism (Line 4) $X \mapsto X^{2N-1}$ on the exponent encoding ciphertext to obtain $RLWE(\pi(\hat{a})) = (RLWE(X^{-a_0}), \ldots, RLWE(X^{-a_{\omega-1}}))$. Subsequently, a plaintext multiplication (Line 5) is conducted between the test polynomial $TV = 0 + X + 2X^2 + \ldots + (N-1)X^{N-1}$ and $RLWE(\pi(\hat{a}))$, resulting in $(RLWE(X^{-a_0} \cdot TV), \ldots, RLWE(X^{-a_{\omega-1}} \cdot TV))$. After extracting the zero coefficient of each RLWE ciphertext (Line 6) in $RLWE(\pi(\hat{a}))$, we will get $LWE_{ij}$ encrypting $\hat{a}_i$. The final step (Line 8) is to pack the $L$ modular $LWE_{0,0}, LWE_{1,0}, \ldots, LWE_{L-1,0}$ to a modular RLWE ciphertext $ct_O$ encrypting $\hat{a} = \sum_{i=0}^{\omega-1} \beta^i \sum_{j=0}^{L-1} a_{ij} X^j$. This process can be carried out in the offline stage on the server side, and its cost is relatively minor compared to the substantial efficiency gains from utilizing the exponent encoding format in the filtering stage.

### 6 EVALUATION

Throughout the experiments, we wish to answer the following two main research questions (RQs).

- **RQ1**: How do the individual cryptographic components of ArcEDB perform in an encrypted database, and how efficient are they compared to SOTA methods?
- **RQ2**: How does the efficiency and expressiveness of ArcEDB in SQL queries compare to other methods?

#### 6.1 Implementation

We implemented ArcEDB using C++17 and complied with it using GCC 11.4.0. Our implementation is based on Microsoft SEAL [100], TFHEpp [106] and OpenFHE [89]. The experiments were carried out on two Intel(R) Xeon(R) Gold 5318Y processors with 512 GB of RAM. We configured the parameters of ArcEDB to provide at least 128-bit of security level according to [3] and [2], and the detailed parameters are laid out in Table A5.
6.2 Microbenchmarks

To answer the RQ1, we conducted comprehensive benchmarks to evaluate the efficiency of each cryptographic component in constructing encrypted SQL query evaluations, including filtering, filter-aggregation, and ORDER BY.

6.2.1 Filtering. The primary focus of our filtering benchmark is to compare the performance of our proposed ArbHCMP and SIMDarHCMP operators against the existing solutions [16, 32, 55, 75, 78, 111] that facilitate unbounded-depth predicate evaluation. We re-implement these solutions based on their open-source implementations [4, 35, 46, 60, 89, 111]. Our benchmarking categorizes the predicate comparison operators into two groups: relational predicates (such as >, ≤, <, ≤) and equality predicates (including == and ! =). We set different precision levels (Precision = 4, 8, 16, 32, 64 bits) and measure the latency for processing a single predicate. As observed in Figure 4, while other solutions may perform differently on relational and equality comparison operator, ArbHCMP and SIMDarHCMP perform consistently better on both comparison tasks. Specifically, ArbHCMP is 6×−56× faster than the SOTA method and SIMDarHCMP is 20×−112× faster than the SOTA method. We provide more clarifications for Figure 4 and include a complexity comparison between these methods in Appendix D.3.

6.2.2 Filter-Aggregation. In the evaluation of filter-aggregation performance, we conduct a thorough comparison between ArcEDB and the SOTA FHE-based EDB solution [16] on a sum query with conjunctions of 2, 4, and 8 predicates applied on 1K and 32K records. For smaller datasets (1K records), we utilize ArbHCMP as the primary comparison operator and for larger datasets (32K records), SIMDarHCMP performs faster for its effective batch processing capability. Figure 5 illustrates the breakdown in query execution time of ArcEDB and HE^3DB [16]. We note that while ArcEDB introduces extra ExpToBase conversions due to encoding format changes, its impact remains negligible compared to the efficiency enhancements brought about by using exponent encoding in the filtering phase. As illustrated in Figure 5, ArcEDB achieves a speedup of 4× to 7× over HE^3DB [16] for datasets with 10K records with ArbHCMP and

36× to 102× acceleration for datasets with 32K records owing to the use of SIMDarHCMP.

6.2.3 ORDER BY. Lastly, we benchmark the performance of the ORDER BY statement. We explore two common scenarios encountered in SQL ORDER BY evaluations. The first scenario assesses the efficiency of sorting a single column. In this evaluation, we compare the latency of ArcEDB to existing works HE^3DB [16] and TFHE-rs [111]. The results, as shown in Figure 6a, demonstrate that ArcEDB outperforms these methods with an average speedup of 2× to 3×. The second scenario focuses on synchronizing other columns based on a particular order. We compare our approach with the Onion Ring ORAM [22] technique, which implements the Waksman permutation network [13] based on the CMUX operator to permute a series of inputs. As depicted in Figure 6b, our performance can be as much as 3×−8× faster than [22].

6.3 SQL Benchmarks

To answer the RQ2, we test the performance of ArcEDB using the TPC-H benchmarks [37] and real-world time-series DB queries [39, 44, 88]. We compare our results against the best-performing FHE-based frameworks HEDA [99] and HE^3DB [16] (details for the SQL are listed in the Figure A2).

For memory usage, as illustrated in Figure 7, we can achieve on average 1.6× less memory than HE^3DB [16] over 16-bit-precision
As illustrated in Figure 8b, ArcEDB is on average 19× faster than HE\textsuperscript{3}DB [16] and 75× faster than HEDA [99] over 16-bit-precision 32K database query due to reduced aggregation precision.

For latency performance, as observed in Figure 8a, we achieve on average 16× faster than HE\textsuperscript{3}DB and 72× than HEDA [99] over 16-bit-precision 32K database query\textsuperscript{2}. Moreover, since HEDA [99] and HE\textsuperscript{3}DB [16] do not support 64-bit timestamp, we extend their precision based on by constructing comparison circuits and compare them with ArcEDB on real-world time-series database queries. As illustrated in Figure 8b, ArcEDB is on average 19× faster than HE\textsuperscript{3}DB and 75× faster than HEDA over 32K time-series database query. To the best of our knowledge, ArcEDB is the first FHE-based EDB framework that can evaluate time-series database queries due to its arbitrary precision capability. Overall, when utilizing 48 cores, we can evaluate an end-to-end SQL query over 10K-row time-series database with 64-bit timestamps under one minute, nearly 20× faster than HE\textsuperscript{3}DB [16] and 75× faster than HEDA [99].

7 CONCLUSIONS

In this work, we introduced ArcEDB, an FHE-based encrypted database system that enables arbitrary-precision and low-latency query evaluations. By leveraging a variant of the modular fully homomorphic encryption scheme and novel encoding methods, we build a new EDB-orient FHE infrastructure with advanced homomorphic comparison, aggregation, and conversion operators.

\textsuperscript{2}We benchmarked the latency and memory usage using the results specified in [99], as the authors have not made their source code available publicly.

We show that ArcEDB can outperform the best-known FHE algorithms on all DB-related task benchmarks, and is able to evaluate a complete SQL query over a 10K-row time-series DB with 64-bit timestamps within one minute. Although ArcEDB has made notable contributions, there is a need for latency improvement to adapt to real-world scenarios. One important future research would be to improve efficiency through more advanced cryptographic primitives or hardware-based accelerations.

REFERENCES

A FULL NOTATIONS AND OPERATORS
We summarize the notations and operators used in this work in Table A1 and Table A2.

B LAYERED DATA STRUCTURE
As shown in Figure A1, the overall data types in ArcEDB are split into three main layers. From top to bottom, we have the encrypted table structure layer, the encrypted data type layer, and the homomorphic ciphertext layer. While most FHE algorithms operate on the bottom layer, we believe that decoupling data structures with low-level FHE ciphertexts are beneficial in building more advanced EDB systems, especially when large-precision plaintext values are stored and processed.

C DETAIL ALGORITHMS
C.1 Homomorphic Filtering
We listed the test vector of HCMP on different comparison operators \(\leq, <, \geq, >, ==, <>\).

\[
\begin{array}{ccc}
\text{Comparison} & \text{Test Vector} & \mu \\
\text{\(a \leq b\)} & \text{TV} = \mu + 2X + \cdots + \mu \frac{X}{N} = 1/2 & \\
\text{\(a < b\)} & \text{TV} = -\mu + 2X + \cdots + \mu \frac{X}{N} = 1/2 & \\
\text{\(a \geq b\)} & \text{TV} = \mu - X + \cdots - \mu \frac{X}{N} = 1/2 & \\
\text{\(a > b\)} & \text{TV} = -\mu + 2X + \cdots + \mu \frac{X}{N} = 1/2 & \\
\text{\(a == b\)} & \text{TV} = \mu & 1
\end{array}
\]

Appendix Figure A1: An overview of the data structure in ArcEDB.

Appendix Table A3: Test vector of HCMP on different comparison operator \(\leq, <, \geq, >, ==, <>\).
Appendix Table A4: Constructing high-precision comparison based on low-precision comparisons. Taking \( a = \{a_0a_1\} \) and \( b = \{b_0b_1\} \) as an example.

<table>
<thead>
<tr>
<th>Comparison</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a \leq b )</td>
<td>( a_1 = b_1 \land a_0 \leq b_0 \land a_1 \leq b_1 )</td>
</tr>
<tr>
<td>( a &lt; b )</td>
<td>( a_0 = b_0 \land a_1 &lt; b_1 )</td>
</tr>
<tr>
<td>( a \geq b )</td>
<td>( a_1 = b_1 \land a_0 \geq b_0 \land a_1 \geq b_1 )</td>
</tr>
<tr>
<td>( a &gt; b )</td>
<td>( a_1 = b_1 \land a_0 &gt; b_0 \land a_1 &gt; b_1 )</td>
</tr>
<tr>
<td>( a = b )</td>
<td>( a_1 = b_1 \land a_0 = b_0 )</td>
</tr>
<tr>
<td>( a \not&lt; b )</td>
<td>( a_1 &gt; b_1 \lor a_0 \not&lt; b_0 )</td>
</tr>
</tbody>
</table>

Appendix Table A5: The Proposed Parameter Sets

<table>
<thead>
<tr>
<th>Ciphertext Format</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LWE &amp; LWE</strong></td>
<td>( n = 1024, [\log_2 q] = 32 )</td>
</tr>
<tr>
<td><strong>RLWE &amp; RLWE</strong></td>
<td>( n = 1024, [\log_2 q] = 32 ) ( n = 4096, [\log_2 q] = 109 ) ( n = 32768, [\log_2 q] = 720 )</td>
</tr>
<tr>
<td><strong>RGSW &amp; RGSW</strong></td>
<td>( N' = 1024, [\log_2 Q] = 32 ) ( N' = 2048, [\log_2 Q] = 64 ) ( N' = 4096, [\log_2 Q] = 109 )</td>
</tr>
</tbody>
</table>

C.2 SIMD Homomorphic MUX

Our core concept in designing SIMD\textsubscript{MUX} follows the amortized bootstrapping technique proposed in [77], which homomorphically decrypts multiple LWE ciphertexts into a single RLWE ciphertext and applies the specific MUX polynomial to all of the plaintext slots in the RLWE ciphertext simultaneously. The detailed algorithm SIMD\textsubscript{MUX} presented in Algorithm 6 involves the following steps. Given three sets of \( N \)-sized LWE ciphertexts \( \{\text{LWE}_{\text{LWE}}(t_i)\} \), \( \{\text{LWE}_{\text{LWE}}(a_i)\} \), \( \{\text{LWE}_{\text{LWE}}(b_i)\} \), where \( i \in \mathbb{Z}_N \), the first step (Line 1–3 in Algorithm 6) is the linear combination of the LWE ciphertexts. After the linear combination, we can see that \( b_i + a_i \cdot s_i \) is equal to \( (4t_i + 2a_i + b_i) \cdot [q/8] + [q/16] \) with some small error. The next step (Line 4–6) is evaluating the homomorphic decryption circuit. On line 4, we rearrange the ciphertexts to construct a ciphertext vector \( b = [b_0, b_1, ..., b_{N-1}] \) and a ciphertext matrix \( A = [a_0, a_1, ..., a_{N-1}] \in \mathbb{Z}_{\text{LWE}}^{N \times N} \). Next on line 5–6, we apply the homomorphic multiplication operations [58, 66, 78] to evaluate \( A \circ \text{BTK} \) and get \( ct = \text{RLWE}_{\text{LWE}}^{N \times Q}(E(As)) \). After homomorphically adding \( b \) to \( ct \), we obtain \( ct_{c} = \text{RLWE}_{\text{LWE}}^{N \times Q}(E(As + b)) \). Here \( ct_{c} \) encrypts \( A \cdot s + b = m + tq \), and the \( tq \) term is automatically removed as \( ct_{c} \) is an RLWE ciphertext with plain modulus \( q \). Thus, \( ct_{c} \) is encrypting \( m \) where \( m[i] = b_i + a_i \cdot s_i \). The next step (line 7–8) is the evaluation of a specific polynomial \( p_{\text{mux}}(x) : \mathbb{Z}_q \rightarrow \mathbb{Z}_q \). We define the polynomial \( p_{\text{mux}}(x) \) as \( p_{\text{mux}}(x) = \{ [q/p] \cdot [x/[q/8]] \} \in \{1, 2\} \cup \{3, 4\} \cup \{6, 8\} \) \( \{0 \cdot [x/[q/8]] \} \in \{0\} \cup \{2, 3\} \cup \{4, 6\} \).

Simplifying, the value \( m[i] = b_i + a_i \cdot s_i \) will fall into the interval \( \{(4t_i + 2a_i + b_i) \cdot [q/8], (4t_i + 2a_i + b_i + 1) \cdot [q/8]\} \). If \( t_i a_i : b_i = 1 \), we define the value of \( p_{\text{mux}}(x) \) on the the interval \( \{(4t_i + 2a_i + b_i) \cdot [q/8], (4t_i + 2a_i + b_i + 1) \cdot [q/8]\} \) as \( [q/p] \). Otherwise, \( 0 \). As explained in Section 2.2, \( p_{\text{mux}}(x) \) can be directly applied to the RLWE ciphertext \( ct_{c} \). Thus, after line 8, we get the ciphertext \( ct_p = \text{RLWE}_{\text{LWE}}^{N \times Q}(E(As + b)) \) and \( c_i = (t_i a_i : b_i) \cdot [q/p] \). Next, we follow [17, 25, 73] to apply \( \text{SlotToCoeff} \) on \( ct_{p} \) and gets \( ct_{c} = \text{RLWE}_{\text{LWE}}^{N \times Q}(c) \). Here \( c = (t_i a_i : b_i) \cdot [q/p] \). After we perform the modulus switching (line 9), we change back the ciphertext with plain modulus \( p \) and ciphertext modulus \( q \) and gets \( c_m = \text{RLWE}_{\text{LWE}}^{N \times Q}(c) \). Now, the value \( c_i \) is equal to \( (t_i a_i : b_i) \). Finally, we extract the \( N \) LWE ciphertexts from \( c_m \) and key switch the LWE ciphertext to the original parameters (line 12–14) and gets \( \text{LWE}_{\text{LWE}}^{n, q}(c_i), \text{LWE}_{\text{LWE}}^{n, d}(c_i), ..., \text{LWE}_{\text{LWE}}^{n, d}(c_{N-1}) \) where \( c_i = (t_i a_i : b_i) \) and the noise of the output is independent of the input ciphertext.

C.3 Homomorphic Sorting and Synchronization

We present the detail algorithm for Sort\textsubscript{Synchronize} in Algorithm 7 and Hom\textsubscript{Sort} in Algorithm 8.

The Algorithm 7 includes the following steps. Given \( L \) modular RGSW ciphertext \( ct = \text{RGSW}(\pi(a_0)), \text{RGSW}(\pi(a_1)), ..., \text{RGSW}(\pi(a_{L-1})) \) where \( a_i = \sum_{j=0}^{L-1} a_{ij} 2^j \) and \( L \) another modular RLWE ciphertext.
First, in Line 3, for each modular RLWE ciphertext $RLWE(\tilde{m}_i)$, we multiply the $RLWE(\tilde{m}_i)$ with $X^0, X^1, \ldots, X^{L-1}$ and result $ct^{exp} = (X^0 RLWE(\tilde{m}_1), \ldots, X^{L-1} RLWE(\tilde{m}_i))$. Next in Line 4-6, we utilize the modular RGSW ciphertext as the control signal and evaluate a CMUX tree on $ct^{exp}$ to get $RLWE(\tilde{m}_i X^{a_i})$. After that, in Line 7, we summarize the L modular ciphertext and obtain $ct_{res} = \sum_{i=0}^{L-1} RLWE(\tilde{m}_i X^{a_i})$. Since the swap will automatically performed due to the exponent indices, after extracting the coefficients of the $ct_{res}$, we get L synchronized modular LWE ciphertexts $(LWE(\tilde{m}_{s_0}), \ldots, LWE(\tilde{m}_{s_{L-1}}))$. The sequence $s_0, \ldots, s_{L-1}$ satisfy $\tilde{a}_{s_0} \leq \tilde{a}_{s_1} \leq \ldots \tilde{a}_{s_{L-1}}$.

The Algorithm 8 is based on Algorithm 7. Given $L$ modular RLWE ciphertexts $ct_\alpha = (RLWE(\pi(a_{d_0})), \ldots, RLWE(\pi(a_{d_{L-1}})))$ First, in Line 1-6, we compare each two ciphertext in $ct_\alpha$ and construct a comparison matrix $A$ where $A[i][j] = LWE(\tilde{a}_i < \tilde{a}_j)$. Next, in Line 7-10, we summarize each row of $A$ and obtain LWE ciphertext vector $\hat{Id}$ where $\hat{Id}[i]$ encrypts the sorted position of $\tilde{a}_i$. On Line 11-14, we decompose the LWE ciphertext $Id[i]$ into a set of LWE ciphertexts encrypting Boolean values and result $BitId[0][i], \ldots, BitId[L-1][i]$ where $BitId[i][j]$ encrypts the j-bit of $Id[i]$. Finally, on Line 15, we apply the Algorithm 7 and swap $L$ modular RLWE ciphertexts $(RLWE(\tilde{m}_0), \ldots, RLWE(\tilde{m}_{L-1}))$ into $L$ modular LWE ciphertexts $(LWE(\tilde{m}_{s_0}), \ldots, LWE(\tilde{m}_{s_{L-1}}))$. The sequence $s_0, \ldots, s_{L-1}$ satisfy $\tilde{a}_{s_0} \leq \tilde{a}_{s_1} \leq \ldots \tilde{a}_{s_{L-1}}$.

## D EXPERIMENT DETAILS

### D.1 Encryption Parameters

The instantiated parameters are outlined in Table A5, which provide at least 128-bit of security level according to [3].

### D.2 SQL Queries Illustration

We provide the time-series benchmark SQL queries in Figure A2. For TPC-H benchmark [37] queries, we remove the JOIN conditions to be consistent with HEDA [99] and HE^3DB [16].
Algorithm 8: The homomorphic sorting operator HomSort

Input : $\mathcal{L}$ modular RLWE ciphertexts
\[
\mathbf{ct}_a = (\mathrm{RLWE}(\pi(\hat{a}_0)), \ldots, \mathrm{RLWE}(\pi(\hat{a}_{L-1}))).
\]
Input : $\mathcal{L}$ another modular RLWE ciphertexts
\[
\mathbf{ct}_m = (\mathrm{RLWE}(\hat{m}_0), \ldots, \mathrm{RLWE}(\hat{m}_{L-1})) \text{ where } 
\hat{m}_i = \sum_{j=0}^{i-1} m_{i-j} \beta^j.
\]
Output : $\mathcal{L}$ synchronized modular LWE ciphertexts
\[
(\mathrm{LWE}(\hat{m}_s_0), \ldots, \mathrm{LWE}(\hat{m}_{s_{L-1}})).
\]
The sequence $s_0, \ldots, s_{L-1}$ satisfy $a_{s_0} \leq a_{s_1} \leq \ldots a_{s_{L-1}}$

1. for $i = 0$ to $L - 1$ do
2.     for $j = 0$ to $i - 1$ do
3.         $A[i][j] \leftarrow 1 - A[j][i]$
4.     $A[i][i] \leftarrow \mathrm{RLWE}(0)$
5.     for $j = i + 1$ to $L - 1$ do
6.         $A[i][j] \leftarrow \text{ArbHCMP} < (\mathbf{ct}_a[i], \mathbf{ct}_a[j])$
7.     Initialize Id = $(\mathrm{LWE}(0), \ldots, \mathrm{LWE}_{L-1}(0))$
8. for $i = 0$ to $L - 1$ do
9.     for $j = 0$ to $L - 1$ do
10.    $\text{Id}[i] \leftarrow \text{Id}[i] + A[i][j]$
11. for $i = 0$ to $L - 1$ do
12.    $\text{BitId}[i] \leftarrow \text{Decompose}(\text{Id}[i])$
13. for $j = 0$ to log $L - 1$ do
14.    $\text{BitId}[i][j] \leftarrow \text{LWEtoRGSW}(\text{BitId}[i][j])$
15. $\mathbf{ct}_O \leftarrow \text{SortSynchronize(BitId, \mathbf{ct}_m)}$
Return : $\mathbf{ct}_O$

D.3 Clarifications for Figure 4

We conclude the concrete complexity comparisons between Liu et al. [76], HE3DB [16] and ArcEDB in Table A6. Moreover, we provide more clarifications for experimental differences between HE3DB [16] and ArcEDB. The main reduction here comes from bootstrapping: ArbHCMP in ArcEDB requires $\lceil k/10 \rceil - 1$ bootstrapping for the comparison of $k$-bit encrypted integers, while HE3DB requires $2 \cdot \lceil k/5 \rceil - 1$. For instance, when $k$ is 16, ArbHCMP requires 1 bootstrapping operation per $k$-bit comparison, while HE3DB requires 7 ($7 \times$ gain). Besides, the ciphertext dimension of ArbHCMP in bootstrapping remains unvarying at 1024, while that of HE3DB changes with data precision. At $k = 16$, the dimension of HE3DB is 2048 (roughly $2 \times$ gain). Therefore, for 16-bit comparisons, ArcEDB is around $14 \times$ faster than HE3DB as shown in Figure 4 in the main manuscript. Additionally, HE3DB requires twice PBS in the case of $==$, which doubles the runtime, while the performance of ArbHCMP is unchanged. Overall, the difference between ArbHCMP and HE3DB ranges from $14 \times$ to $28 \times$. 