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
Zhongguancun Laboratory
Beijing, China

Jiafeng Hua
jiafenghhua@163.com
Xidian University
Xian, 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

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.

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

Table 1: Runtime Analysis of 16-bit Precision TPC-H Q6 [37] on HEDA [99],HE^{3}DB [16] and ArcEDB.

Scheme	Filter (s)		Aggregation (s)		Total (s)	
HEDA [99]	11758	99.7%	32	0.3%	11790	100%
HE^{3}DB [16]	1363	93.4%	97	6.6%	1460	100%
ArcEDB	91	48.9%	95	51.1%	186	100%

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 HE^{3}DB [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 20× 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

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

	Lu *et al.* [79]	Cheon *et al.* [28]	Kortekaas [72]	Iliashenko *et al.* [62]	Antonio *et al.* [55]	PEGASUS [78]	Liu *et al.* [76]	SortingHat [32]	TFHE-rs [111]	HE^{3}DB [16]	Ours
Ciphertext type$^+$	RLWE	RLWE	MRLWE	RLWE	MLWE	LWE	LWE	MLWE	MLWE	LWE	MRLWE
Encoding	Exponent	Slot	Boolean	Boolean	Boolean	Coeff	Coeff	Boolean	Coeff	Coeff	Exponent
32-bit precision	✗	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓
Arbitrary precision	✗	▲*	✓	▲*	✓	✗	▲*	✓	✓	✗	✓
Equality	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓
Unbounded depth	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓
Batching	✗	✓	✓	✓	✗	✗	✗	✗	✗	✗	✓
Aggregation^{++}	✗	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓

$^+$Ciphertext type refers to the ciphertext format defined in Section 2.1, including LWE, RLWE, Modular LWE (MLWE), and Modular RLWE (MRLWE).
*Improving precision requires larger parameters. $^{++}$Supporting encrypted aggregations over comparison results.

performance penalties induced by FHE computations are still the main barrier against its real-world deployment.

1.2.2 Comparing Encrypted Data over FHE. As sketched in Table 1, the single most important limiting factor to the performance of FHE-based EDB is the excessive amount of homomorphic comparisons that stem from the filtering and logic aggregation statements. Here, we give a comprehensive review of existing homomorphic comparison algorithms and their computational characteristics. Roughly speaking, ciphertext-ciphertext comparison 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 encipher 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 adopts 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.

2 FHE PRIMITIVES

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., $\mathbf{a}$) for vectors, tilde lowercase letters (e.g., $\tilde{a}$) for polynomials, and bold uppercase letters (e.g., $\mathbf{A}$) for matrices. Throughout this work, We use λ to denote the security parameter and p/P for different plaintext moduli (generally $P > p$). $q/Q/Q'$ 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 $\mathbb{Z}[X]/(X^N + 1) \bmod Q$. 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 RGSW.

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

$$\mathsf{LWE}_{\mathbf{s}}^{n,q}(m) = (b, \mathbf{a}) = (< -\mathbf{a}, \mathbf{s} > + \Delta m + e, \mathbf{a}). \tag{1}$$

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

• $\mathsf{RLWE}_{\tilde{s}}^{N,Q}(\tilde{m})$: An RLWE ciphertext is formulated as

$$\mathsf{RLWE}_{\tilde{s}}^{N,Q}(\tilde{m}) = (\tilde{b}, \tilde{a}) = (-\tilde{a} \cdot \tilde{s} + \Delta\tilde{m} + \tilde{e}, \tilde{a}). \tag{2}$$

for a polynomial of encoded messages $\tilde{m} \in R_{N,P}$ under a secret key $\tilde{s} \in R_{N,Q}$ and $\Delta = \left\lceil \frac{Q}{P} \right\rfloor$.

• $\mathsf{RGSW}_{\tilde{s}}^{N',Q'}(m)$: Given a decomposition size l, the RGSW encryption of a message $m \in \mathbb{Z}_p$ under the secret key $\mathbf{s} \in \mathbb{R}_{N',Q'}$ is defined as $\mathsf{RGSW}_{\tilde{s}}^{N',Q'}(m) = (\mathbf{B}, \mathbf{A}) \in \mathbb{Z}_{Q'}^{2l \times 2}$.

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 degradations 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 ciphertexts as

$$\widehat{\mathsf{LWE}}_{\mathbf{s}}^{n,q}(\hat{m}) = \big(\mathsf{LWE}(m_0), \cdots, \mathsf{LWE}(m_{\omega-1})\big), \tag{3}$$

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

$$\widehat{\mathsf{RLWE}}_{\tilde{s}}^{N,Q}(\hat{\tilde{m}}) = \big(\mathsf{RLWE}(\tilde{m}_0), \cdots, \mathsf{RLWE}(\tilde{m}_{\omega-1})\big), \tag{4}$$

where $\tilde{m}_i = \sum_{j=0}^{N-1} m_{i,j} X^j \in R_{N,p}$, and $\hat{\tilde{m}} = \{\tilde{m}_0, \tilde{m}_1, \ldots, \tilde{m}_{\omega-1}\} = \sum_{i=0}^{\omega-1} \tilde{m}_i \beta^i \in R_{N,P}$ for some radix base β. In other words, here, a large plaintext polynomial $\hat{\tilde{m}} \in R_P$ is cut into ω chunks of $\tilde{m}_i \in R_p$, where each $\tilde{m}_i$ is encrypted as an RLWE ciphertext $\mathsf{RLWE}(\tilde{m}_i)$.

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

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 $\mathsf{LWE}(m)$ and $\mathsf{RLWE}(m)$ 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 $\tilde{m}$ encrypted in $\mathsf{RLWE}_s^{N,Q}(\tilde{m})$).

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

• $poly(\mathsf{RLWE}(\tilde{m}))$: For any polynomial $poly(x)$, $poly(\mathsf{RLWE}(\tilde{m}))$ represents the homomorphic evaluation of $poly$ over the input ciphertext $\mathsf{RLWE}(\tilde{m})$ encrypting $\tilde{m}$.

• $\mathtt{Automorphism}(\mathsf{RLWE}(\tilde{m}), \mathfrak{d})$: For a given ciphertext $\mathsf{RLWE}(\tilde{m}[X])$, $\mathtt{Automorphism}(\mathsf{RLWE}(\tilde{m}[X], \mathfrak{d})$ outputs a new ciphertext $\mathsf{RLWE}_{\text{out}}(\tilde{m}[X^{\mathfrak{d}}])$, i.e., the coefficient of the plaintext polynomial is rearranged by the parameter $\mathfrak{d}$.

• $\mathtt{ExternalProduct}(\mathsf{RGSW}(\tilde{d}), \mathsf{RLWE}(\tilde{m}))$: $\mathtt{ExternalProduct}$ is a special type of homomorphic multiplication, where $\mathtt{ExternalProduct}(\mathsf{RGSW}(\tilde{d}), \mathsf{RLWE}(\tilde{m})) = \mathsf{RLWE}(\tilde{d} \cdot \tilde{m})$. In general, $\mathtt{ExternalProduct}$ 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.

• $\mathtt{CMUX}(\mathsf{RGSW}(\mathfrak{t}), \mathsf{LWE}(a), \mathsf{LWE}(b))$: For inputs $\mathsf{LWE}(a)$ and $\mathsf{LWE}(b)$, given a control signal $\mathsf{RGSW}(\mathfrak{t})$ that encrypts a binary plaintext $\mathfrak{t} \in \{0, 1\}$, $\mathtt{CMUX}(\mathsf{RGSW}(\mathfrak{t}), \mathsf{LWE}(a), \mathsf{LWE}(b))$ homomorphically computes $\mathfrak{t}\ ?\ \mathsf{LWE}(a)\ :\ \mathsf{LWE}(b)$, i.e., the function selects $\mathsf{LWE}(a)$ if $\mathfrak{t} = 1$ and $\mathsf{LWE}(b)$ if $\mathfrak{t} = 0$.

• $\mathtt{BlindRotate}(\mathsf{LWE}_{\mathbf{s}}^{n,q}, \mathbf{BK}, \tilde{TV})$: $\mathtt{BlindRotate}$ takes as input an LWE ciphertext $\mathsf{LWE}_{\mathbf{s}}^{n,q}$, a bootstrapping key $\mathbf{BK}$, and a polynomial $\tilde{TV}$ (i.e., the test vector in [29]), and generates an RLWE ciphertext $\mathsf{RLWE}(X^{-\rho}\tilde{TV})$, where $\rho = \left\lceil 2n \cdot (b + \sum_{i=1}^{\ell} s_i \cdot a_i)/q \right\rfloor$.

• $\mathtt{HomGate}(\mathsf{LWE}(m_0), \mathsf{LWE}(m_1), \mathtt{OP})$: Given two LWE ciphertexts $\mathsf{LWE}(m_0)$ and $\mathsf{LWE}(m_1)$ with a two-input logic gate $\mathtt{OP}$, $\mathtt{HomGate}(\mathsf{LWE}(m_0), \mathsf{LWE}(m_1), \mathtt{OP})$ produces $\mathsf{LWE}(\mathtt{OP}(m_0, m_1))$. Here, $\mathtt{OP}$ 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`, `LWEstoRLWE` 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`$(\mathsf{RLWE}(\tilde{m}))$: `RLWEtoLWEs` converts $\mathsf{RLWE}^{N,Q}_{\tilde{s}}$ ciphertext to a set of N $\mathsf{LWE}^{n,q}_{\mathbf{s}}$ ciphertexts. As defined in [29], `RLWEtoLWEs` outputs N LWE ciphertexts $\mathbf{ct} = (ct_0, ct_1, ..., ct_{N-1})$ where ct_i encrypts the i-th plaintext coefficient of $\tilde{m}$.
- `LWEstoRLWE`$(\mathsf{LWE}_0, \cdots, \mathsf{LWE}_{N-1})$: `LWEstoRLWE` converts a set of N $\mathsf{LWE}^{n,q}_{\mathbf{s}}$ ciphertexts to one $\mathsf{RLWE}^{N,Q}_{\tilde{s}}$ ciphertext, i.e., the inverse of `RLWEtoLWEs`.
- `LWEtoRGSW`$(\mathsf{LWE}, \mathbf{BK})$: `LWEtoRGSW` converts an $\mathsf{LWE}^{n,q}_{\mathbf{s}}$ ciphertext to an $\mathsf{RGSW}^{N',Q'}_{\tilde{\mathbf{s}}}$ 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 structures 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 $\mathcal{D}$ by the client. Here, each table $\mathcal{T} \in \mathcal{D}$, will be encrypted utilizing the `TableEncrypt` function, producing the encrypted table $[\mathcal{T}]$. More specifically, an encrypted Table $[\mathcal{T}]$ comprises a number of encrypted data columns that can be categorized into two classes based on the attribute properties, namely, the filter attribute columns ($\mathsf{Attr}^{\mathsf{cmp}}$) and the aggregation attribute columns ($\mathsf{Attr}^{\mathsf{agg}}$). Different classes of attribute columns can have different encrypted data types and ciphertext structures, which will be further elaborated in Section 3.2. After table encryption, the client generates the associated evaluation keys (i.e., **BK** and **KSK**) and transfers both the encrypted tables $[\mathcal{T}]$ along with the evaluation keys to the server.

② **Client Query Encryption**: When querying the outsourced encrypted table $[\mathcal{T}]$, 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 $Q = (\mathcal{P}_0 \diamond \mathcal{P}_1 \diamond \ldots \diamond \mathcal{P}_{|Q|-1}, \mathsf{Attr}^{\mathsf{agg}}, \mathsf{Agg})$. Here, each $\mathcal{P}_i = (\mathsf{Attr}^{\mathsf{cmp}}_i, \mathsf{cmp}_i, b_i)$ represents a predicate consisting of the i-th filtering attribute $\mathsf{Attr}^{\mathsf{cmp}}_i$, the comparison operator cmp_i, and a predicate value b_i. Different predicates are concatenated with some logic function $\diamond$, which can be AND $\wedge$ or OR $\vee$. Agg denotes the aggregation function to be applied on the aggregation attribute $\mathsf{Attr}^{\mathsf{agg}}$. The `QueryEncrypt` function homomorphically encrypts the predicate values in each of $\mathcal{P}_i$, yielding the encrypted predicate $[\mathcal{P}]_i$ ciphertexts. Subsequently, we obtain the encrypted query $[Q]$ formatted as $([\mathcal{P}]_0 \diamond \ldots \diamond [\mathcal{P}]_{|Q|-1}, \mathsf{Attr}^{\mathsf{agg}}, \mathsf{Agg})$. 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 $[\mathcal{T}]$. The main computations involved in homomorphic query evaluations are the homomorphic filtering and homomorphic aggregation functions. First, in the filtering stage, each encrypted predicate $[\mathcal{P}]_i$ is evaluated over the encrypted table using the `FilterPred` (more details in Section 3.3) function and produces the filtered result $[\mathcal{F}]_i$, which is a set of ciphertexts encrypting 1 if the predicate $\mathcal{P}_i$ on such item is true and 0 if false. Next, after the predicate evaluation, the filtering results $\{[\mathcal{F}]_i\}$ 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 $[\mathcal{F}]$. Here, $[\mathcal{F}]$ is an array of $|\mathcal{T}|_{\mathrm{row}}$ LWE ciphertexts encrypting either 1 (true) or 0 (false), indicating whether the i-th row of $\mathcal{T}$ is selected or not. Lastly, the aggregation functions Agg are executed over the aggregated $\mathsf{Attr}^{\mathsf{agg}}$ column to produce the final result $[\mathcal{R}]$, which can be either $\widehat{\mathsf{LWE}}$ or $\widehat{\mathsf{RLWE}}$ ciphertexts, depending on the aggregation function. Finally, the round of query evaluation finishes when the client decrypts $[\mathcal{R}]$ 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.

- $\widehat{\mathsf{LWE}}$: The modular variant of the LWE ciphertext as defined in Section 2.1.2. $\widehat{\mathsf{LWE}}$ is mainly employed for encrypting Boolean values in ArcEDB. Meanwhile, $\widehat{\mathsf{LWE}}$ can also encrypt the intermediate results during the query evaluation.
- Coefficient/Slot/Exponent-based $\widehat{\mathsf{RLWE}}$: The modular version of RLWE ciphertext defined in Section 2.1.2. The plaintext encoding for $\widehat{\mathsf{RLWE}}$ is much more complex than that of $\widehat{\mathsf{LWE}}$. 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 $\pi : \mathbb{Z} \to R$ 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 $\hat{a} = \{a_0, a_1 \cdots, a_{\omega-1}\}$, the modular exponent RLWE ciphertext is defined as:

$$\widehat{\mathsf{RLWE}}\big(\pi(\hat{a})\big) = \big(\mathsf{RLWE}(X^{a_0}), \cdots, \mathsf{RLWE}(X^{a_{\omega-1}})\big). \tag{5}$$

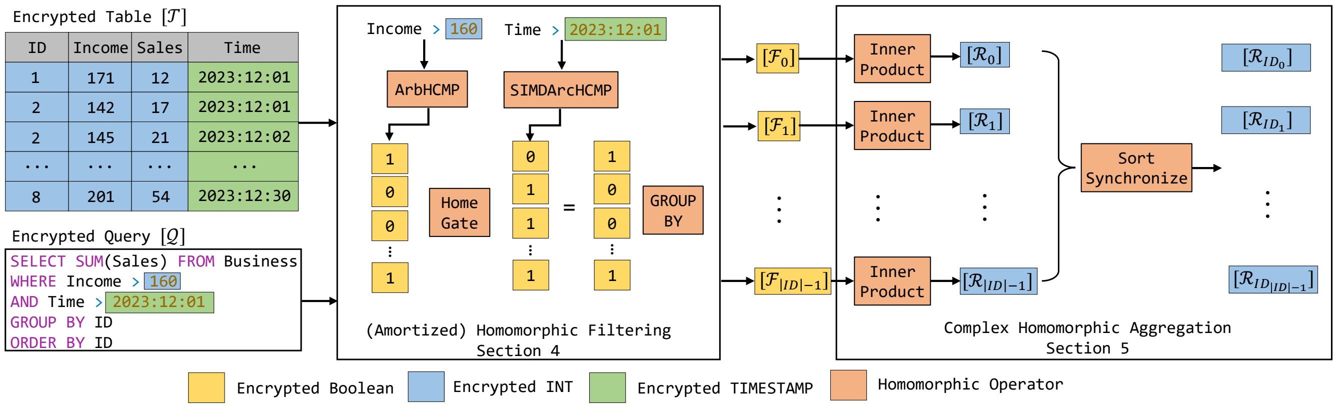


Figure 1: The system-level overview of ArcEDB.

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 $\widehat{\text{RGSW}}$: The modular version of the RGSW ciphertext defined in Section 2.1.2. In ArcEDB, $\widehat{\text{RGSW}}$ 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 $\hat{b} = \{b_0, b_1, \cdots b_{\omega-1}\}$, the corresponding ciphertext is formulated as:

$$\widehat{\text{RGSW}}\big(\pi(-\hat{b})\big) = \big(\text{RGSW}(X^{-b_0}), \cdots, \text{RGSW}(X^{-b_{\omega-1}})\big) \quad (6)$$

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 ($\widehat{\text{RLWE}}(\hat{a})$), slot ($\widehat{\text{RLWE}}(\mathcal{E}(\hat{a}))$), or exponent ($\widehat{\text{RLWE}}(\pi(\hat{a}))$). 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 $\widehat{\text{RLWE}}$ 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, EBINARY 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 $\hat{\hat{m}}_i$. 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 $\hat{\hat{m}}$ is encrypted into modular RLWE ciphertexts with exponent encoding, i.e., $\widehat{\text{RLWE}}(\pi(\hat{\hat{m}}))$.

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

• **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`$(\mathcal{T}) \rightarrow [\mathcal{T}]$: Encrypts the database table $\mathcal{T}$ and yields the encrypted table $[\mathcal{T}]$. This process involves encrypting both the filter and the aggregation columns in $\mathcal{T}$.

• `QueryEncrypt`$(Q) \rightarrow [Q]$: Encrypts a SQL query $Q = (\mathcal{P}_0 \diamond \mathcal{P}_1 \diamond \ldots \diamond \mathcal{P}_{|Q|-1}, \mathsf{Attr}^{\mathsf{agg}}, \mathsf{Agg})$ into an encrypted query $[Q] = ([\mathcal{P}]_0 \diamond [\mathcal{P}]_1 \diamond \ldots \diamond [\mathcal{P}]_{|Q|-1}, \mathsf{Attr}^{\mathsf{agg}}, \mathsf{Agg})$. In ArcEDB, each encrypted predicate $[\mathcal{P}]_i$ is always encrypted in the form of a negative exponent modular RGSW ciphertexts $\widehat{\mathsf{RGSW}}(\pi(-\hat{b_i}))$, where b_i is the corresponding predicate value b_i.

• `Query`$([Q], [\mathcal{T}]) \rightarrow [\mathcal{R}]$: Evaluates an encrypted query $[Q] = ([\mathcal{P}]_0 \diamond [\mathcal{P}]_1 \diamond \ldots \diamond [\mathcal{P}]_{|Q|-1}, \mathsf{Attr}^{\mathsf{agg}}, \mathsf{Agg})$ on the encrypted table $[\mathcal{T}]$. Essentially, the querying process is to invoke the `FilterPred` and `Aggregation` APIs consecutively, detailed as follows.

- `FilterPred`$([\mathcal{P}], [\mathcal{T}]) \rightarrow [\mathcal{F}]$: Filters an encrypted predicate $[\mathcal{P}]$ on the encrypted table $[\mathcal{T}]$ and outputs $|\mathcal{T}|_{\mathrm{row}}$ LWE ciphertexts $[\mathcal{F}] = \mathsf{LWE}_0, \cdots \mathsf{LWE}_{|\mathcal{T}|_{\mathrm{row}}-1}$. The function utilizes advanced homomorphic comparison operators `HCMP`, `ArbHCMP`, and `SIMDArbHCMP`, detailed in Section 4, to enhance the precision and efficiency of encrypted predicate evaluation.
- `Aggregation`$([\mathcal{F}], \mathsf{Attr}^{\mathsf{agg}}, \mathsf{Agg}, [\mathcal{T}]) \rightarrow [\mathcal{R}]$: Aggregates a specified column $\mathsf{Attr}^{\mathsf{agg}}$ by a function Agg on the encrypted table $[\mathcal{T}]$ and the encrypted filter result $[\mathcal{F}]$.

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

• `GROUP BY`$([Q], \mathsf{Attr}^{\mathsf{grpby}}, [\mathcal{T}]) \rightarrow \{[\mathcal{R}]_i\}$: Evaluates the `SELECT` statement query $[Q]$ with the `GROUP BY` attribute $\mathsf{Attr}^{\mathsf{grpby}}$ on the encrypted table $[\mathcal{T}]$. Similar to [16, 99], to implement `GROUP BY`, we issues multiple copies of the query $[Q]$, each augmented with an additional equality test for the group attribute.

• `ORDER BY`$(\mathsf{Attr}^{\mathsf{sort}}, [\mathcal{T}]) \rightarrow [\mathcal{T}]'$: Evaluates the `ORDER BY` statement on the encrypted table $[\mathcal{T}]$ based on a target attribute $\mathsf{Attr}^{\mathsf{sort}}$. `ORDER BY` is essentially a series of homomorphic comparisons appended by data swapping based on the comparison results. Note that existing approaches [16] only focus on sorting the $\mathsf{Attr}^{\mathsf{sort}}$ column alone and do not account for the synchronization of other columns based on the sorted column. ArcEDB introduces a novel homomorphic `ORDER BY` technique built upon our exponent-based encoding, enabling efficient synchronized sorting across multiple columns. More details can be found in Section 5.2.

3.4 Threat Model and Security Guarantees

The security objective of ArcEDB is to protect the outsourced database $\mathcal{D}$ owned by the client C against a semi-honest server $\mathcal{S}$, 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:

• $|\mathcal{T}|_{\mathrm{row}}, |\mathcal{T}|_{\mathrm{col}}$: the number of rows as well as the number of columns in the table $\mathcal{T} \in \mathcal{D}$.

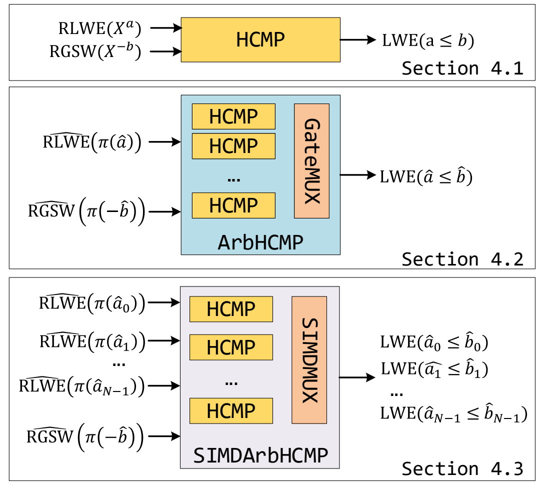


Figure 2: An overview of the homomorphic filtering in ArcEDB.

• $|Q|$: the number of filtering predicates in a SQL query Q.

• Attr, |Attr|: The attribute label (e.g., gender, date) and the range of the attribute (e.g., |Gender| = 2).

• $\diamond$: The concatenating logic function (e.g., $\wedge$, $\vee$) between the filtering predicates in a SQL query.

• Agg: the aggregation functions (e.g., `SUM`, `MIN`) in Q.

Private Data:

• $\mathcal{T}_{i,j}$, for $i \in |\mathcal{T}|_{\mathrm{row}}$, $j \in |\mathcal{T}|_{\mathrm{col}}$: exact values of the database items for all $\mathcal{T} \in \mathcal{D}$.

• $\mathcal{P}_i \in Q$, for $i \in |Q|$: the predicate values in the SQL query.

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 $\mathcal{D}$. 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

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

SQL statement	#ArbHCMP	#HomGate	#CMUX	#LWEtoRGSW	#LWEstoRLWE	#+	#·	#Automorphism
SELECT	$\lvert\mathcal{T}\rvert_{\text{row}} \cdot \lvert Q\rvert$	$\lvert\mathcal{T}\rvert_{\text{row}} \cdot (\lvert Q\rvert - 1)$	0	0	0	0	0	0
SUM	0	0	0	0	1	$\log_2 \lvert\mathcal{T}\rvert_{\text{row}}$	1	$\log_2 \lvert\mathcal{T}\rvert_{\text{row}}$
COUNT	0	0	0	0	0	$\lvert\mathcal{T}\rvert_{\text{row}} - 1$	0	0
MIN/MAX	$\lvert\mathcal{T}\rvert_{\text{row}} - 1$	0	$2\lvert\mathcal{T}\rvert_{\text{row}} - 1$	$\lvert\mathcal{T}\rvert_{\text{row}} - 1$	0	0	0	0
GROUP BY	$\lvert\mathcal{T}\rvert_{\text{row}} \cdot \lvert\text{Attr}^{\text{grpby}}\rvert$	0	0	0	0	0	0	0
ORDER BY	0	0	$\lvert\mathcal{T}\rvert^2_{\text{row}} - \lvert\mathcal{T}\rvert_{\text{row}}$	0	0	$\lvert\mathcal{T}\rvert_{\text{row}} - 1$	0	0
JOIN	$\lvert\mathcal{T}_a\rvert_{\text{row}} \cdot \lvert\mathcal{T}_b\rvert_{\text{row}}$	0	0	0	0	0	0	0

$\lvert Q\rvert$, $\lvert\mathcal{T}\rvert_{\text{row}}$, $\lvert\text{Attr}^{\text{grpby}}\rvert$ are publicly known to the server.

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, HE^3DB [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 HE^3DB [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:

$$\tilde{C} = \tilde{TV} \cdot \pi(a) \cdot \pi(-b) \mod (X^N + 1), \tag{7}$$

where $\tilde{TV} = 1 + X + ... + X^{N-1}$ 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 TV shifts right, making the zeroth coefficient $\tilde{C}$ equal to 1. Conversely, if $a > b$, 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].

Algorithm 1: Basic hmomorphic comparison operator $\texttt{HCMP}_{\leq}$

Input : Two ciphertexts $ct_a = \text{RLWE}(X^a)$, $ct_b = \text{RGSW}(X^{-b})$ with plain modulus p.
Output : An LWE ciphertext $ct_O = \text{LWE}(c)$ where $c = 1$ if $a \leq b$, otherwise $c = 0$.

1 $ct \leftarrow \texttt{ExternalProduct}(ct_b, ct_a)$ $\mu \leftarrow 2^{-1} \mod p$
2 $\tilde{TV} \leftarrow \mu + \mu X + ... + \mu X^{N-1}$
3 $ct \leftarrow \tilde{TV} \cdot ct + \mu$ $ct_O \leftarrow \texttt{RLWEtoLWEs}(ct)[0]$

Return : $ct_O = \text{LWE}(c)$

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 $\text{RLWE}(X^a)$ and $\text{RGSW}(X^{-b})$, yielding $ct = \text{RLWE}(X^{a-b})$. Subsequently, on Line 2–3, we construct a test vector $\tilde{TV} = \mu + \mu X + ... + \mu X^{N-1}$ where μ denotes the inverse of 2 modulo p. In the following phase (Line 4), we evaluate $TV \cdot ct + \mu$ to obtain $\text{RLWE}(X^{a-b} \cdot TV + \mu)$. It is observed that the constant term of $\text{RLWE}(X^{a-b} \cdot TV + \mu)$ results in either $\mu + \mu = 1$ or $-\mu + \mu = 0$, depending on whether $a \leq b$. Lastly, on Line 5, we apply the `RLWEtoLWEs` function and extract the zeroth coefficient of $\text{RLWE}(X^{a-b} \cdot TV + \mu)$ to produce the output LWE ciphertext ct_O. The ciphertext ct_O encrypts either 1 or 0, indicating whether the data item a satisfies the predicate condition. For other comparison operators like $>$, $\geq$, $<$, $==$, and $\neq$, we only need to slightly modify the test vector TV. Detailed explanations for these modifications are provided in Table A3.

Algorithm 2: Homomorphic MUX operator GateMUX

Input : Three LWE ciphertexts $ct_d = \mathsf{LWE}^{n,q}(d)$, $ct_a = \mathsf{LWE}^{n,q}(a)$, $ct_b = \mathsf{LWE}^{n,q}(b)$, where $d, a, b \in \{0, 1\}$ and plain modulus p.
Input : A bootstrapping key **BK**.
Output: An LWE ciphertext $ct_O = \mathsf{LWE}(d\ ?\ a : b)$.
1 $ct_y = ct_d + 4ct_a + 2ct_b$ $scale \leftarrow \lfloor p/16 \rfloor$, $offset \leftarrow \lfloor q/32 \rfloor$
2 $ct_{linear} \leftarrow scale \cdot ct_y + offset$
3 $\mathbf{T} \leftarrow \{0, 0, 1, 0, 0, 1, 1, 1\}$ $\tilde{TV} = \sum_{y=0}^{7} \sum_{j=0}^{n/8-1} \mathbf{T}[y] \cdot X^{j+yn/8}$
4 $ct_{rot} \leftarrow \mathtt{BlindRotate}(ct_{linear}, \mathbf{BK}, \tilde{TV})$
5 $ct_O \leftarrow \mathtt{RLWEtoLWEs}(ct_{rot})[0]$
Return : $ct_O = \mathrm{LWE}(d\ ?\ a : b)$

Algorithm 3: Arbitrary-precision homomorphic comparison operator $\mathtt{ArbHCMP}_{\leq}$

Input : A modular RLWE ciphertext $ct_a = \widehat{\mathsf{RLWE}}(\pi(\hat{a}))$, where $\hat{a} = \sum_{i=0}^{\omega-1} a_i \cdot N^i$, a modular RGSW ciphertext $ct_b = \widehat{\mathsf{RGSW}}(\pi(-\hat{b}))$ where where $\hat{b} = \sum_{i=0}^{\omega-1} b_i \cdot N^i$.
Input : A bootstrapping key **BK**, modular ciphertext size ω.
Output: An LWE ciphertext $ct_O = \mathsf{LWE}(\hat{a} \leq \hat{b})$.
1 $\omega \leftarrow ct_a.size()$
2 **if** $\omega == 1$ **then**
3 $\quad ct_O \leftarrow \mathtt{HCMP}_{\leq}(ct_a[0], ct_b[0])$
4 **else**
5 $\quad ct_d \leftarrow \mathtt{HCMP}_{==}(ct_a[\omega - 1], ct_b[\omega - 1])$
6 $\quad ct_0 \leftarrow \mathtt{ArbHCMP}\leq(ct_a[0 : \omega - 1], ct_b[0 : \omega - 1], \mathbf{BK})$
7 $\quad ct_{\omega-1} \leftarrow \mathtt{HCMP}_{\leq}(ct_a[\omega - 1], ct_b[\omega - 1])$
8 $\quad ct_O \leftarrow \mathtt{GateMUX}(ct_d, ct_0, ct_{\omega-1}, \mathbf{BK})$
Return : $ct_O = \mathrm{LWE}(c)$

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 η-bit integer into a sequence of ω chunks of lower precision $\log N$-bit integers, and encrypts 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

$$a_1 == b_1\ ?\ a_0 \leq b_0\ :\ a_1 \leq b_1 \tag{8}$$

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 gate 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 $ct_d = \mathsf{LWE}^{n,q}(d)$, $ct_a = \mathsf{LWE}^{n,q}(a)$, $ct_b = \mathsf{LWE}^{n,q}(b)$, 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 $d\ ?\ a\ :\ b$. On Line 6–7, we apply BlindRotate and RLWEtoLWEs to output the LWE ciphertext ct_O encrypting $d\ ?\ a\ :\ b$.

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 η-bit integers, let $\omega = \lceil \eta/N \rceil$, 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 $\mathrm{Attr}^{\mathtt{cmp}}$ 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 $\widehat{\mathsf{RLWE}}(\pi(\hat{a})) = (\mathsf{RLWE}(X^{a_0}), ..., \mathsf{RLWE}(X^{a_{\omega-1}}))$ and ct_b be the modular RGSW ciphertext that encrypts the $\omega \log N$-bit queried attribute value $\hat{b}$ with $\widehat{\mathsf{RGSW}}(\pi(-\hat{b})) = (\mathsf{RGSW}(X^{-b_0}), ..., \mathsf{RGSW}(X^{-b_{\omega-1}}))$. Suppose that cmp is the type of comparison to be performed (where cmp can be one of the $\leq, <, \geq, >, ==, \neq$ 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

Algorithm 4: Amortized arbitrary-precision homomorphic comparison operator $\texttt{SIMDArbHCMP}_{\leq}$

Input : N modular RLWE ciphertexts $\mathbf{ct_a} = (\widehat{\mathsf{RLWE}}(\pi(\hat{a}_0)), \widehat{\mathsf{RLWE}}(\pi(\hat{a}_1)), ..., \widehat{\mathsf{RLWE}}(\pi(\hat{a}_{N-1})))$ for $\hat{a}_i$ a modular RGSW ciphertext $ct_b = \widehat{\mathsf{RGSW}}(\pi(-\hat{b}))$.
Input : A batch bootstrapping key **BTK**.
Input : An LWE key switching key **KSK**.
Output : N LWE ciphertexts $\mathbf{ct_O} = (\mathsf{LWE}(\hat{a}_0 \leq \hat{b}), \mathsf{LWE}(\hat{a}_1 \leq \hat{b}), ..., \mathsf{LWE}(\hat{a}_{N-1} \leq \hat{b}))$.

$\omega \leftarrow \mathbf{ct_a}[0].size()$
if $\omega == 1$ **then**
  for $i = 0$ **to** $N-1$ **do**
    $\mathbf{ct_O}[i] \leftarrow \texttt{HCMP}_{\leq}(\mathbf{ct_a}[i][0], ct_b[0])$
else
  for $i = 0$ **to** $N-1$ **do**
    $\mathbf{ct_d}[i] \leftarrow \texttt{HCMP}_{==}(\mathbf{ct_a}[i][\omega-1], ct_b[\omega-1])$
    $\mathbf{ct}_{\omega-1}[i] \leftarrow \texttt{HCMP}_{\leq}(\mathbf{ct_a}[i][\omega-1], ct_b[\omega-1])$
  $\mathbf{ct_0} \leftarrow \texttt{SIMDArbHCMP}_{\leq}(\mathbf{ct_a}[0:N][0:\omega-1], ct_b[0:\omega-1], \mathbf{BTK}, \mathbf{KSK})$
  $ct_O \leftarrow \texttt{SIMDCMUX}(\mathbf{ct_d}, \mathbf{ct_0}, \mathbf{ct}_{\omega-1}, \mathbf{BTK}, \mathbf{KSK})$
Return : $\mathbf{ct_O}$

the detailed procedures for $\texttt{ArbHCMP}_{\leq}$ in Algorithm 3. First, when $\omega == 1$, we can directly apply $\texttt{HCMP}_{\leq}$ on $ct_a[0]$ and $ct_b[0]$ and get the comparison result (Line 2 – 3). Otherwise when $\omega \geq 2$, supposing $\texttt{ArbHCMP}_{\leq}$ can perform $(\omega-1)\cdot\log N$-bit precision comparison, we transform the expression $\hat{a} \leq \hat{b}$ to the following equation

$$a_{\omega-1} == b_{\omega-1} ? \{a_0, ..., a_{\omega-2}\} \leq \{b_0, ..., b_{\omega-2}\} : a_{\omega-1} \leq b_{\omega-1} \quad (9)$$

Thus, on Line 5 – 7, we individually evaluate the comparisons in Equation (9) utilizing HCMP operator and $(\omega-1)\cdot\log N$-bit precision ArbHCMP operator to obtain three LWE ciphertexts $ct_d, ct_0, ct_{\omega-1}$ encrypting $a_{\omega-1} == b_{\omega-1}, \{a_0, ..., a_{\omega-2}\} \leq \{b_0, ..., b_{\omega-2}\}$, and $a_{\omega-1} \leq b_{\omega-1}$. Finally, on Line 8, we perform the MUX operation on $ct_d, ct_0, ct_{\omega-1}$ based on the GateMUX to get the final LWE ciphertext result encrypting $\hat{a} \leq \hat{b}$. By recursively evaluating $a_i \leq b_i$ using HCMP and combining the comparison of the result by GateMUX without extra conversion, we can construct ArbHCMP 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 SIMDArbHCMP 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 HCMP building block to construct SIMDArbHCMP, we only need to design a new homomorphic MUX algorithm in a SIMD manner, i.e., the SIMDCMUX operator, to accelerate homomorphic comparisons over large-size DB columns. The main procedure of SIMDCMUX 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. In contrast to the GateMUX operator, which conducts a single MUX operation on three-input LWE ciphertexts, SIMDCMUX is able to perform N MUX operations simultaneously on $3N$ LWE ciphertexts. Due to the space limitation, the concrete constructions for SIMDCMUX is depicted in Algorithm 6.

Based on the amortized homomorphic MUX operator SIMDCMUX 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 $\mathbf{ct_a} = (\widehat{\mathsf{RLWE}}(\pi(\hat{a}_0)), \widehat{\mathsf{RLWE}}(\pi(\hat{a}_1)), ..., \widehat{\mathsf{RLWE}}(\pi(\hat{a}_{N-1})))$ be the L modular RLWE ciphertexts encrypt $\omega \log N$-bit DB integer item $\hat{a}_0, \hat{a}_1, ..., \hat{a}_{N-1}$ and ct_b be the modular RGSW ciphertext encrypts $\omega \log N$-bit query attribute $\hat{b}$ with $ct_b = \widehat{\mathsf{RGSW}}(\pi(-\hat{b}))$. Suppose that cmp is the type of comparison to be performed. The SIMDArbHCMP 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 $\texttt{SIMDArbHCMP}_{\leq}$ in Algorithm 4. The main process is similar to Algorithm 3 but change the Line 6 in Algorithm 3 with SIMDArbHCMP 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 $[\mathcal{F}]$, which encompass $|\mathcal{T}|_{\text{row}}$ 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, $|\mathcal{T}|_{\text{row}}$ LWE ciphertexts encrypting the filtered results must first be packed into an RLWE ciphertext $[\mathcal{F}^{\text{rlwe}}]$ through the LWEstoRLWE 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 $[\mathcal{F}^{\text{rlwe}}]$ and a vector $I = (1, 1, \ldots, 1)$. Similarly, the SUM function involves an inner product between $[\mathcal{F}^{\text{rlwe}}]$ and the to-be-aggregated column ciphertext Attr^{agg}. 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 exponent 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 `ExpToBase`

Input : L modular RLWE ciphertexts $\mathbf{ct_a} = \left(\widehat{\mathsf{RLWE}}(\pi(\hat{a_0})), \widehat{\mathsf{RLWE}}(\pi(\hat{a_1})), \cdots, \widehat{\mathsf{RLWE}}(\pi(\hat{a_{L-1}})\right)$ where $\hat{a}_i = \sum_{j=0}^{\omega-1} a_{i,j}\beta^j$.

Input : Modular RLWE ciphertext dimension N.

Output: An RLWE ciphertext $ct_O = \widehat{\mathsf{RLWE}}(\hat{\hat{a}})$, where $\hat{\hat{a}} = \sum_{j=0}^{\omega-1} \beta^j \sum_{i=0}^{L-1} a_{i,j} X^i$.

1 Initialize $\tilde{TV} \leftarrow 0 + X + 2X^2 + \ldots + (N-1)X^{N-1}$
2 **for** $i = 0$ **to** $L-1$ **do**
3 **for** $j = 0$ **to** $\omega - 1$ **do**
4   $ct_{i,j} \leftarrow \mathsf{Automorphism}(\mathbf{ct_a}[i][j], 2N-1)$
5   $ct_{i,j} \leftarrow \tilde{TV} \cdot ct_{i,j}$
6   $\mathsf{LWE}_{i,j} \leftarrow \mathsf{RLWEtoLWEs}(ct_{i,j})[0]$
7 **for** $j = 0$ **to** $\omega - 1$ **do**
8 $ct_j \leftarrow \mathsf{LWEstoRLWE}(\mathsf{LWE}_{0,j}, \mathsf{LWE}_{1,j}, \cdots \mathsf{LWE}_{L-1,j})$
9 $ct_O \leftarrow (ct_0, ct_1, \cdots, ct_{\omega-1})$

Return: $ct_O = \widehat{\mathsf{RLWE}}(\hat{\hat{a}})$

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 $\mathbf{ct_a} = \left(\widehat{\mathsf{RLWE}}(\pi(\hat{a_0})), \widehat{\mathsf{RLWE}}(\pi(\hat{a_1})), \cdots, \widehat{\mathsf{RLWE}}(\pi(\hat{a_{L-1}})\right)$ where $\hat{a}_i = \sum_{j=0}^{\omega-1} a_{i,j}\beta^j$. The conversion process begins by performing the automorphism (Line 4) $X \rightarrow X^{2N-1}$ on the exponent encoding ciphertext to obtain $\widehat{\mathsf{RLWE}}(\pi(-\hat{a}_i)) = \left(\mathsf{RLWE}(X^{-a_{i,0}}), \ldots, \mathsf{RLWE}(X^{-a_{i,\omega-1}})\right)$. Subsequently, a plaintext multiplication (Line 5) is conducted between the test polynomial $\tilde{TV} = 0 + X + 2X^2 + \ldots + (N-1)X^{N-1}$ and $\widehat{\mathsf{RLWE}}(\pi(-\hat{a}_i))$, resulting in $\left(\mathsf{RLWE}(X^{a_{i,0}} \cdot \tilde{TV}), \ldots, \mathsf{RLWE}(X^{a_{i,\omega-1}} \cdot \tilde{TV})\right)$. After extracting the zeroth coefficient of each RLWE ciphertext (Line 6) in $\widehat{\mathsf{RLWE}}(\pi(-\hat{a}_i))$, we will get $\widehat{\mathsf{LWE}}_i$ encrypting $\hat{a}_i$. The final step (Line 8) is to pack the L modular LWE ciphertext $\widehat{\mathsf{LWE}}_0, \widehat{\mathsf{LWE}}_1, \ldots, \widehat{\mathsf{LWE}}_{L-1}$ to a modular RLWE ciphertext ct_O encrypting $\hat{\hat{a}} = \sum_{j=0}^{\omega-1} \beta^j \sum_{i=0}^{L-1} a_{i,j} X^i$. 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 one 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

To-be-sorted column $\boldsymbol{w} = [5, 3, 7, 6]$

$\boldsymbol{Id}[i] = \sum_{j=0}^{3} \boldsymbol{w}_i < \boldsymbol{w}_j \quad \Rightarrow \quad \boldsymbol{Id} = [1, 0, 3, 2]$

Synchronize another column $\boldsymbol{v} = [4, 9, 2, 8]$

$\hat{v} = \sum_{i=0}^{3} \boldsymbol{v}_i X^{\boldsymbol{Id}[i]}$
$= 4\boldsymbol{X^1} + 9\boldsymbol{X^0} + 2X^3 + 8\boldsymbol{X^2}$
$= 9\boldsymbol{X^0} + 4\boldsymbol{X^1} + 8\boldsymbol{X^2} + 2X^3$
$\Downarrow$ Extract coefficients sequentially
$[9, 4, 8, 2]$

Figure 3: Toy example for `HomSort` with L = 4.

Appendix C.3. Our `HomSort` algorithm works in a 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 work for a to-be-sorted column $\mathbf{w} \in \mathbb{Z}_p^4$. Here, we first compute the order index of each item in the through $\mathbf{Id}[i] = \sum_{j=0}^{4}(\mathbf{w}_i < \mathbf{w_j})$, resulting in $\mathbf{Id} = [1, 0, 3, 2]$. This indicates that the $\mathbf{w}[i]$ can be placed in the $\mathbf{Id}[i]$ position in the sorted column $\mathbf{w_s}$. The subsequent step involves converting the position values $\mathbf{Id}[i]$ into the exponent form $X^{\mathbf{Id}[i]}$, followed by an inner product with a to-be-synchronized column $\mathbf{v}$ to produce $\tilde{v}$. The crucial observation is that, the resulting polynomial $\tilde{v}$ will automatically swap the coefficients due to the exponent indices. After extracting the coefficients of $\tilde{v}$, we obtain the synchronized column, and its order matches with that of the sorted-$\mathbf{w}$. Due to space limitations, more details on the sorting and synchronizing algorithms are outlined in Algorithm 7 and Algorithm 8.

Remark: Similar to sorting, other logic aggregation algorithms such as `MIN` or `MAX` can also be accelerated by the exponent-encoding-based sorting approach. As later shown in Section 6.2.3, with a properly encrypted index column, ArcEDB can significantly outperform existing solutions.

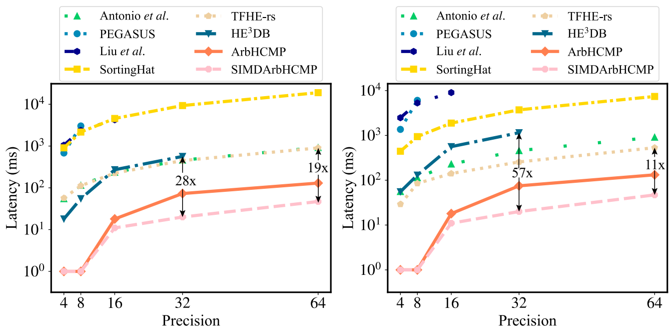
6 EVALUATION

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

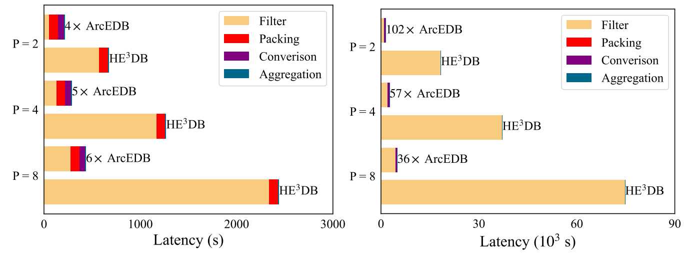
We implemented ArcEDB using C++17 and complied with it using GCC 11.4.0. Our implementation is based on Microsoft SEAL [100], TFHEpp [106] and OpenFHE [89]. The experiments were carried out on two Intel(R) Xeon(R) Gold 5318Y processors with 512 GB of RAM. We configured the parameters of ArcEDB to provide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



(a) Relational Comparison

(b) Equality Comparison

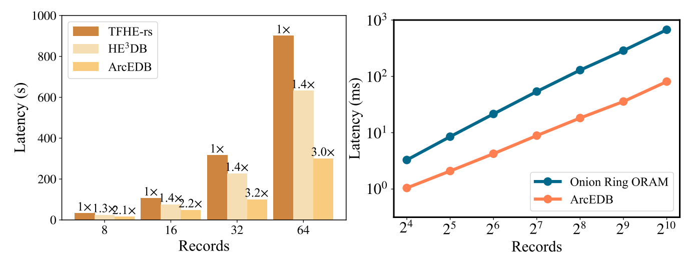
Figure 4: Benchmark results for homomorphic predicate evaluation with the relational comparison operator and equality comparison operator.



(a) Latency with 1K records.

(b) Latency with 32K records.

Figure 5: Breakdown of SQL filter-aggregation latency with different numbers of predicates (P). The filter phase includes predicate evaluation and combination. The packing phase converts LWE filter results to RLWE ciphertext for aggregation. The aggregation phase aggregates the specific column.



(a) Column Sorting

(b) Column Synchronization

Figure 6: The (a) sorting and (b) synchronization performance of `ORDER BY` operator over databases with different sizes.

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 `SIMDArbHCMP` operators against the leading existing solutions [16, 32, 55, 76, 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 $>, \leq, <, \leq$) 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 `SIMDArbHCMP` perform consistently better on both comparison tasks. Specifically, `ArbHCMP` is 6×−56× faster than the SOTA method and `SIMDArbHCMP` 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.

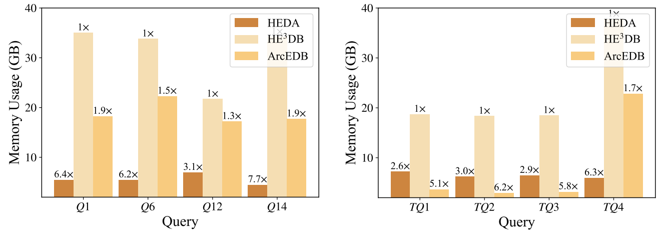
6.2.2 Filter-Aggregation. In the evaluation of filter-aggregation performance, we conduct a thorough comparison between ArcEDB and the SOTA FHE-based EDB solution [16] on a sum query with conjunctions of 2, 4, and 8 predicates applied on 1K and 32K records. For smaller datasets (1K records), we utilize `ArbHCMP` as the primary comparison operator and for larger datasets (32K records), `SIMDArbHCMP` performs faster for its effective batch processing capability. Figure 5 illustrates the breakdown in query execution time of ArcEDB and HE^{3}DB [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^{3}DB [16] for datasets with 10K records with `ArbHCMP` and 36× to 102× acceleration for datasets with 32K records owing to the use of `SIMDArbHCMP`.

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^{3}DB [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

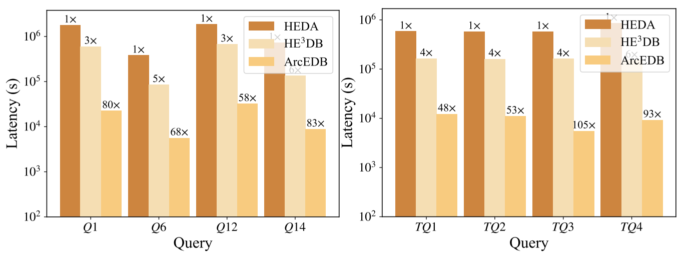
To answer the **RQ2**, we test the performance of ArcEDB using the TPC-H benchmarks [37] and real-world time-series DB queries [39, 44, 88]. We compare our results against the best-performing FHE-based frameworks HEDA [99] and HE^{3}DB [16] (details for the SQL are listed in the Figure A2).

For memory usage, as illustrated in Figure 7, we can achieve on average 1.6× less memory than HE^{3}DB [16] over 16-bit-precision



(a) Relational DB with 32K records.(b) Time-series DB with 32K records.

Figure 7: Peak memory usage comparisons on SQL queries.



(a) Relational DB with 32K records.(b) Time-series DB with 32K records.

Figure 8: Latency performance comparisons on SQL queries.

32K database query and 4.7× less memory over 32K time-series database query. Meanwhile, HEDA [99] has smaller memory consumption 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^{3}DB [16] and 72× than HEDA [99] over 16-bit-precision 32K database query[2]. Moreover, since HEDA [99] and HE^{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^{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^{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. 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.

[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.

REFERENCES

[1] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. 2004. Order preserving encryption for numeric data. In *International Conference on Management of Data*. 563–574.

[2] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. 2018. *Homomorphic Encryption Security Standard*. Technical Report. HomomorphicEncryption.org.

[3] Martin R. Albrecht, Rachel Player, and Sam Scott. 2015. On the concrete hardness of Learning with Errors. *Journal of Mathematical Cryptology* 9, 3 (2015), 169–203.

[4] Alibaba-Gemini-Lab. [n. d.]. Pegasus: Bridging Polynomial and Non-polynomial Evaluations in Homomorphic Encryption. https://github.com/Alibaba-Gemini-Lab/OpenPEGASUS

[5] Sebastian Angel, Hao Chen, Kim Laine, and Srinath T. V. Setty. 2018. PIR with Compressed Queries and Amortized Query Processing. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 962–979.

[6] Panagiotis Antonopoulos, Arvind Arasu, Kunal D Singh, Ken Eguro, Nitish Gupta, Rajat Jain, Raghav Kaushik, Hanuma Kodavalla, Donald Kossmann, Nikolas Ogg, et al. 2020. Azure SQL database always encrypted. In *ACM SIGMOD International Conference on Management of Data*. 1511–1525.

[7] Arvind Arasu, Ken Eguro, Manas Joglekar, Raghav Kaushik, Donald Kossmann, and Ravi Ramamurthy. 2015. Transaction processing on confidential data using cipherbase. In *IEEE International Conference on Data Engineering*. IEEE, 435–446.

[8] AWS. 2023. Machine Learning on AWS. https://aws.amazon.com/machine-learning/?nc2=h_ql_sol_use_ml. Accessed: 2023-01-01.

[9] Azure. 2023. Azure Machine Learning. https://azure.microsoft.com/en-us/products/machine-learning/. Accessed: 2023-01-01.

[10] Maurice Bailleu, Jörg Thalheim, Pramod Bhatotia, Christof Fetzer, Michio Honda, and Kapil Vaswani. 2019. SPEICHER: Securing LSM-based Key-Value Stores using Shielded Execution. In *USENIX Conference on File and Storage Technologies*. 173–190.

[11] Sumeet Bajaj and Radu Sion. 2011. TrustedDB: a trusted hardware based database with privacy and data confidentiality. In *ACM SIGMOD International Conference on Management of data*. 205–216.

[12] Johes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel Kho, and Jennie Rogers. 2016. SMCQL: Secure querying for federated databases. *arXiv preprint arXiv:1606.06808* (2016).

[13] Bruno Beauquier and Éric Darrot. 2002. On Arbitrary Size Waksman Networks and Their Vulnerability. *Parallel Processing Letters* 12, 3-4 (2002), 287–296.

[14] Loris Bergerat, Anas Boudi, Quentin Bourgerie, Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. 2023. Parameter Optimization and Larger Precision for (T) FHE. *Journal of Cryptology* 36, 3 (2023), 28.

[15] Song Bian, Dur-e-Shahwar Kundi, Kazuma Hirozawa, Weiqiang Liu, and Takashi Sato. 2021. APAS: Application-Specific Accelerators for RLWE-Based Homomorphic Linear Transformations. *IEEE Transactions on Information Forensics and Security* 16 (2021), 4663–4678.

[16] 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 *ACM SIGSAC Conference on Computer and Communications Security*. 2930–2944.

[17] Jean-Philippe Bossuat, Christian Mouchet, Juan Ramón Troncoso-Pastoriza, and Jean-Pierre Hubaux. 2021. Efficient Bootstrapping for Approximate Homomorphic Encryption with Non-sparse Keys. In *EUROCRYPT*. Springer, 587–617.

[18] Zvika Brakerski. 2012. Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP. In *CRYPTO*. 868–886.

[19] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2012. (Leveled) fully homomorphic encryption without bootstrapping. In *ACM Transactions on Computation Theory*. 309–325.

[20] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. 2015. Leakage-abuse attacks against searchable encryption. In *ACM SIGSAC Conference on Computer and Communications Security*. 668–679.

[21] David Cash, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Cătălin Roşu, and Michael Steiner. 2013. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Annual cryptology conference*. Springer, 353–373.

[22] Hao Chen, Ilaria Chillotti, and Ling Ren. 2019. Onion ring ORAM: efficient constant bandwidth oblivious RAM from (leveled) TFHE. In *ACM SIGSAC Conference on Computer and Communications Security*. 345–360.

[23] Hao Chen, Wei Dai, Miran Kim, and Yongsoo Song. 2021. Efficient Homomorphic Conversion Between (Ring) LWE Ciphertexts. In *Applied Cryptography and Network Security*. Springer, 460–479.

[24] Hao Chen, Zhicong Huang, Kim Laine, and Peter Rindal. 2018. Labeled PSI from fully homomorphic encryption with malicious security. In *ACM SIGSAC Conference on Computer and Communications Security*. 1223–1237.

[25] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. 2018. Bootstrapping for Approximate Homomorphic Encryption. In *EUROCRYPT*. Springer.

[26] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. 2017. Homomorphic Encryption for Arithmetic of Approximate Numbers. In *ASIACRYPT*. 409–437.

[27] Jung Hee Cheon, Dongwoo Kim, and Duhyeong Kim. 2020. Efficient Homomorphic Comparison Methods with Optimal Complexity. In *ASIACRYPT*. Springer, 221–256.

[28] Jung Hee Cheon, Dongwoo Kim, Duhyeong Kim, Hun-Hee Lee, and Keewoo Lee. 2019. Numerical Method for Comparison on Homomorphically Encrypted Numbers. In *ASIACRYPT*. Springer, 415–445.

[29] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2020. TFHE: Fast Fully Homomorphic Encryption Over the Torus. *Journal of Cryptology* 33, 1 (2020), 34–91.

[30] Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. 2021. Improved Programmable Bootstrapping with Larger Precision and Efficient Arithmetic Circuits for TFHE. In *ASIACRYPT*. 670–699.

[31] Google Cloud. 2023. Cloud SQL. https://cloud.google.com/sql/. Accessed: 2023-01-01.

[32] Kelong Cong, Debajyoti Das, Jeongeun Park, and Hilder V. L. Pereira. 2022. SortingHat: Efficient Private Decision Tree Evaluation via Homomorphic Encryption and Transciphering. In *ACM SIGSAC Conference on Computer and Communications Security*. ACM, 563–577.

[33] Kelong Cong, Radames Cruz Moreno, Mariana Botelho da Gama, Wei Dai, Ilia Iliashenko, Kim Laine, and Michael Rosenberg. 2021. Labeled psi from homomorphic encryption with reduced computation and communication. In *ACM SIGSAC Conference on Computer and Communications Security*. 1135–1150.

[34] Henry Corrigan-Gibbs and Dan Boneh. 2017. Prio: Private, Robust, and Scalable Computation of Aggregate Statistics. In *USENIX Conference on Networked Systems Design and Implementation*. 259–282.

[35] KU Leuven COSIC. [n. d.]. Private decision tree evaluation via Homomorphic Encryption and Transciphering. https://github.com/KULeuven-COSIC/SortingHat

[36] Victor Costan and Srinivas Devadas. 2016. Intel SGX explained. *Cryptology ePrint Archive* (2016).

[37] Transaction Processing Performance Council. 2022. *TPC BENCHMARK^{TM} H Standard Specification*. Technical Report. San Francisco,CA.

[38] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. 2006. Searchable symmetric encryption: improved definitions and efficient constructions. In *ACM SIGSAC Conference on Computer and Communications Security*. 79–88.

[39] Emma Dauterman, Mayank Rathee, Raluca Ada Popa, and Ion Stoica. 2022. Waldo: A Private Time-Series Database from Function Secret Sharing. In *IEEE Symposium on Security and Privacy*. IEEE, 2450–2468.

[40] Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Saurabh Shintre. 2020. SEAL: Attack mitigation for encrypted databases via adjustable leakage. In *USENIX Security Symposium*. 2433–2450.

[41] Srinivas Devadas, Marten van Dijk, Christopher W Fletcher, Ling Ren, Elaine Shi, and Daniel Wichs. 2016. Onion ORAM: A constant bandwidth blowup oblivious RAM. In *Theory of Cryptography Conference*. Springer, 145–174.

[42] Léo Ducas and Daniele Micciancio. 2015. FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second. In *EUROCRYPT*. 617–640.

[43] Saba Eskandarian and Matei Zaharia. 2019. ObliDB: Oblivious Query Processing for Secure Databases. *Proceedings of the VLDB Endowment*. 13, 2 (2019), 169–183.

[44] Muhammad Faisal, Jerry Zhang, John Liagouris, Vasiliki Kalavri, and Mayank Varia. 2023. TVA: A multi-party computation system for secure and expressive time series analytics. USENIX Association, 5395–5412.

[45] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat Practical Fully Homomorphic Encryption. *IACR Cryptol. ePrint Arch.* (2012), 144.

[46] FBT-TFHE. [n. d.]. Revisiting the functional bootstrap in TFHE. https://github.com/antoniocgj/FBT-TFHE

[47] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. 2021. Scalable Memory Protection in the PENGLAI Enclave. In *USENIX Symposium on Operating Systems Design and Implementation*. 275–294.

[48] Benjamin Fuller, Mayank Varia, Arkady Yerukhimovich, Emily Shen, Ariel Hamlin, Vijay Gadepally, Richard Shay, John Darby Mitchell, and Robert K Cunningham. 2017. Sok: Cryptographically protected database search. In *IEEE Symposium on Security and Privacy*. IEEE, 172–191.

[49] Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. 2016. TWORAM: efficient oblivious RAM in two rounds with applications to searchable encryption. In *Annual International Cryptology Conference*. Springer, 563–592.

[50] Craig Gentry, Shai Halevi, and Nigel P Smart. 2012. Homomorphic evaluation of the AES circuit. In *CRYPTO*. Springer, 850–867.

[51] Craig Gentry, Amit Sahai, and Brent Waters. 2013. Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based. In *CRYPTO*. 75–92.

[52] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. 2018. Pump up the volume: Practical database reconstruction from volume leakage on range queries. In *ACM SIGSAC Conference on Computer and Communications Security*. 315–331.

[53] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. 2019. Learning to reconstruct: Statistical learning theory and encrypted database attacks. In *IEEE Symposium on Security and Privacy*. IEEE, 1067–1083.

[54] Zichen Gui, Oliver Johnson, and Bogdan Warinschi. 2019. Encrypted Databases: New Volume Attacks against Range Queries. In *ACM SIGSAC Conference on Computer and Communications Security*. 361–378.

[55] Antonio Guimarães, Edson Borin, and Diego F. Aranha. 2021. Revisiting the functional bootstrap in TFHE. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2021), 229–253.

[56] Timon Hackenjos, Florian Hahn, and Florian Kerschbaum. 2020. SAGMA: Secure Aggregation Grouped by Multiple Attributes. In *International Conference on Management of Data*. ACM, 587–601.

[57] Shai Halevi, Yuriy Polyakov, and Victor Shoup. 2019. An Improved RNS Variant of the BFV Homomorphic Encryption Scheme. In *Topics in Cryptology–CT-RSA*. 83–105.

[58] Shai Halevi and Victor Shoup. 2014. Algorithms in HElib. In *CRYPTO*. Springer, 554–571.

[59] Kyoohyung Han, Minki Hhan, and Jung Hee Cheon. 2019. Improved Homomorphic Discrete Fourier Transforms and FHE Bootstrapping. *IEEE Access* 7 (2019), 57361–57370.

[60] HE^{3}DB. [n. d.]. HE^{3}DB: An Efficient and Elastic Encrypted Database Via Arithmetic-And-Logic Fully Homomorphic Encryption. https://github.com/zhouzhangwalker/HE3DB

[61] Zhicong Huang, Wen-jie Lu, Cheng Hong, and Jiansheng Ding. 2022. Cheetah: Lean and Fast Secure Two-Party Deep Neural Network Inference. In *USENIX Security Symposium*. USENIX Association, 809–826.

[62] Ilia Iliashenko and Vincent Zucca. 2021. Faster homomorphic comparison operations for BGV and BFV. *Proc. Priv. Enhancing Technol.* 2021, 3 (2021), 246–264.

[63] Yuval Ishai, Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. 2016. Private large-scale databases with distributed searchable symmetric encryption. In *Cryptographers' Track at the RSA Conference*. Springer, 90–107.

[64] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. Access pattern disclosure on searchable encryption: ramification, attack and mitigation.. In *Network & Distributed System Security Symposium*.

[65] Simon Johnson, Raghunandan Makaram, Amy Santoni, and Vinnie Scarlata. 2021. Supporting intel sgx on multi-socket platforms. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/supporting-intel-sgx-on-mulit-socket-platforms.pdf.

[66] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha P. Chandrakasan. 2018. GAZELLE: A Low Latency Framework for Secure Neural Network Inference. In *USENIX Security Symposium*. USENIX Association, 1651–1669.

[67] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'neill. 2016. Generic attacks on secure outsourced databases. In *ACM SIGSAC Conference on Computer and Communications Security*. 1329–1340.

[68] Alhassan Khedr, Glenn Gulak, and Vinod Vaikuntanathan. 2015. SHIELD: scalable homomorphic implementation of encrypted data-classifiers. *IEEE Trans. Comput.* 65, 9 (2015), 2848–2858.

[69] Taehoon Kim, Joongun Park, Jaewook Woo, Seungheun Jeon, and Jaehyuk Huh. 2019. Shieldstore: Shielded in-memory key-value storage with sgx. In *Proceedings of the Fourteenth EuroSys Conference*. 1–15.

[70] Evgenios M Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. 2020. The state of the uniform: Attacks on encrypted databases beyond the uniform query distribution. In *IEEE Symposium on Security and Privacy*. IEEE, 1223–1240.

[71] Evgenios M Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. 2021. Response-hiding encrypted ranges: Revisiting security via parametrized leakage-abuse attacks. In *IEEE Symposium on Security and Privacy*. IEEE, 1502–1519.

[72] Y.A.M. Kortekaas. 2020. Access Pattern Hiding Aggregation over Encrypted Databases. http://essay.utwente.nl/83874/

[73] Yongwoo Lee, Joon-Woo Lee, Young-Sik Kim, Yongjune Kim, Jong-Seon No, and HyungChul Kang. 2022. High-Precision Bootstrapping for Approximate Homomorphic Encryption by Error Variance Minimization. In *EUROCRYPT*. Springer, 551–580.

[74] John Liagouris, Vasiliki Kalavri, Muhammad Faisal, and Mayank Varia. 2021. Secrecy: Secure collaborative analytics on secret-shared data. *arXiv preprint arXiv:2102.01048* (2021).

[75] Zheli Liu, Xiaofeng Chen, Jun Yang, Chunfu Jia, and Ilsun You. 2016. New order preserving encryption model for outsourced databases in cloud environments. *Journal of Network and Computer Applications* 59 (2016), 198–207.

[76] Zeyu Liu, Daniele Micciancio, and Yuriy Polyakov. 2022. Large-Precision Homomorphic Sign Evaluation Using FHEW/TFHE Bootstrapping. In *ASIACRYPT*, Shweta Agrawal and Dongdai Lin (Eds.). Springer, 130–160.

[77] Zeyu Liu and Yunhao Wang. 2023. Amortized Functional Bootstrapping in less than 7ms, with $\tilde{O}(1)$ polynomial multiplications. In *ASIACRYPT*. 1–29.

[78] Wen-jie Lu, Zhicong Huang, Cheng Hong, Yiping Ma, and Hunter Qu. 2021. PEGASUS: Bridging Polynomial and Non-polynomial Evaluations in Homomorphic Encryption. In *IEEE Symposium on Security and Privacy*. IEEE, 1057–1073.

[79] Wenjie Lu, Jun-Jie Zhou, and Jun Sakuma. 2018. Non-interactive and Output Expressive Private Comparison from Homomorphic Encryption. In *Asia Conference on Computer and Communications Security*. ACM, 67–74.

[80] Rasoul Akhavan Mahdavi and Florian Kerschbaum. 2022. Constant-weight PIR: Single-round Keyword PIR via Constant-weight Equality Operators. In *USENIX Security Symposium*. USENIX Association, 1723–1740.

[81] Rasoul Akhavan Mahdavi, Haoyan Ni, Dimitry Linkov, and Florian Kerschbaum. 2023. Level Up: Private Non-Interactive Decision Tree Evaluation using Levelled Homomorphic Encryption. In *ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2945–2958.

[82] Kotaro Matsuoka, Ryotaro Banno, Naoki Matsumoto, Takashi Sato, and Song Bian. 2021. Virtual Secure Platform: A Five-Stage Pipeline Processor over TFHE. In *USENIX Security Symposium*. 4007–4024.

[83] Samir Jordan Menon and David J Wu. 2022. Spiral: Fast, high-rate single-server PIR via FHE composition. In *IEEE Symposium on Security and Privacy*. IEEE, 930–947.

[84] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. 2018. Oblix: An Efficient Oblivious Search Index. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 279–296.

[85] MonogoDB. 2023. Application-Driven Analytics. https://www.mongodb.com/use-cases/analytics. Accessed: 2023-01-01.

[86] Muhammad Haris Mughees, Hao Chen, and Ling Ren. 2021. OnionPIR: Response Efficient Single-Server PIR. In *ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2292–2306.

[87] Muhammad Haris Mughees and Ling Ren. 2023. Vectorized Batch Private Information Retrieval. In *IEEE Symposium on Security and Privacy*. IEEE, 437–452.

[88] Syeda Noor Zehra Naqvi, Sofia Yfantidou, and Esteban Zimányi. 2017. Time series databases and influxdb. *Studienarbeit, Université Libre de Bruxelles* 12 (2017).

[89] openfheorg. [n. d.]. OpenFHE - Open-Source Fully Homomorphic Encryption Library. https://github.com/openfheorg/openfhe-development

[90] Simon Oya and Florian Kerschbaum. 2021. Hiding the access pattern is not enough: Exploiting search pattern leakage in searchable encryption. In *USENIX Security Symposium*. 127–142.

[91] Antonis Papadimitriou, Ranjita Bhagwan, Nishanth Chandran, Ramachandran Ramjee, Andreas Haeberlen, Harmeet Singh, Abhishek Modi, and Saikrishna Badrinarayanan. 2016. Big Data Analytics over Encrypted Datasets with Seabed. In *USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 587–602.

[92] Vasilis Pappas, Fernando Krell, Binh Vo, Vladimir Kolesnikov, Tal Malkin, Seung Geol Choi, Wesley George, Angelos Keromytis, and Steve Bellovin. 2014. Blind seer: A scalable private DBMS. In *IEEE Symposium on Security and Privacy*. IEEE, 359–374.

[93] Benny Pinkas and Tzachy Reinman. 2010. Oblivious RAM revisited. In *CRYPTO*. Springer, 502–519.

[94] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. 2019. Arx: An Encrypted Database using Semantically Secure Encryption. *Proceedings of the VLDB Endowment* (2019).

[95] Rishabh Poddar, Sukrit Kalra, Avishay Yanai, Ryan Deng, Raluca Ada Popa, and Joseph M Hellerstein. 2021. Senate: A Maliciously-Secure MPC Platform for Collaborative Analytics. In *USENIX Security Symposium*. 2129–2146.

[96] Rishabh Poddar, Stephanie Wang, Jianan Lu, and Raluca Ada Popa. 2020. Practical volume-based attacks on encrypted databases. In *IEEE European Symposium on Security and Privacy*. IEEE, 354–369.

[97] Raluca A. Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: protecting confidentiality with encrypted query processing. In *ACM Symposium on Operating Systems Principles*. ACM, 85–100.

[98] Christian Priebe, Kapil Vaswani, and Manuel Costa. 2018. EnclaveDB: A secure database using SGX. In *2018 IEEE Symposium on Security and Privacy*. IEEE, 264–278.

[99] 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* (2022).

[100] SEAL 2022. Microsoft SEAL (release 4.1.0). https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA..

[101] Nigel P. Smart and Frederik Vercauteren. 2014. Fully homomorphic SIMD operations. *Designs, codes and cryptography* 71, 1 (2014), 57–81.

[102] Emil Stefanov, Marten Van Dijk, Elaine Shi, T-H Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2018. Path ORAM: an extremely simple oblivious RAM protocol. *J. ACM* 65, 4 (2018), 1–26.

[103] G Edward Suh, Charles W O'Donnell, and Srinivas Devadas. 2007. Aegis: A single-chip secure processor. *IEEE Design & Test of Computers* 24, 6 (2007), 570–580.

[104] Stephen Tu, M. Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. 2013. Processing analytical queries over encrypted data. *Proceedings of the VLDB Endowment* (2013), 289–300.

[105] Dhinakaran Vinayagamurthy, Alexey Gribov, and Sergey Gorbunov. 2019. StealthDB: a Scalable Encrypted Database with Full SQL Query Support. *Proc. Priv. Enhancing Technol.* 2019, 3 (2019), 370–388.

[106] virtualsecureplatform. [n. d.]. TFHEpp. https://github.com/virtualsecureplatform/TFHEpp

[107] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. 2019. Conclave: secure multi-party computation on big data. In *Proceedings of the Fourteenth EuroSys Conference*. 1–18.

[108] Chenghong Wang, Johes Bater, Kartik Nayak, and Ashwin Machanavajjhala. 2022. IncShrink: Architecting Efficient Outsourced Databases using Incremental MPC and Differential Privacy. In *International Conference on Management of Data*. ACM, 818–832.

[109] Sheng Wang, Yiran Li, Huorong Li, Feifei Li, Chengjin Tian, Le Su, Yanshan Zhang, Yubing Ma, Lie Yan, Yuanyuan Sun, et al. 2022. Operon: An encrypted database for ownership-preserving data management. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3332–3345.

[110] Yilei Wang and Ke Yi. 2021. Secure Yannakakis: Join-Aggregate Queries over Private Data. In *International Conference on Management of Data*. 1969–1981.

[111] Zama. 2022. TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data. https://github.com/zama-ai/tfhe-rs.

[112] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. 2016. All your queries are belong to us: the power of {File-Injection} attacks on searchable encryption. In *USENIX Security Symposium*. 707–720.

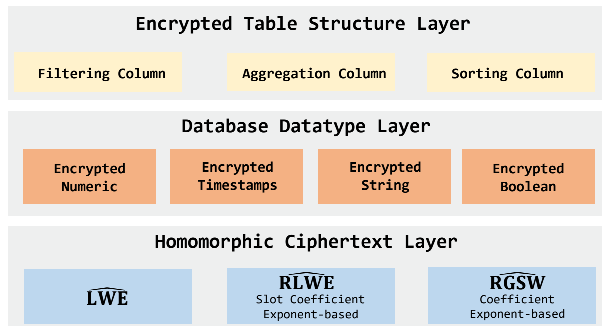
[113] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. 2017. Opaque: An oblivious and encrypted distributed analytics platform. In *USENIX Symposium on Networked Systems Design and Implementation*. 283–298.

Appendix Table A1: Summary of Notations

Notation	Description
$\mathcal{D}$	The database
$\mathcal{T}$	The data table in the database
$\mathcal{Q}$	The SQL query
$\mathcal{P}$	The SQL query predicate
Agg	The aggregation function in SQL query
Attr	The attribute label in database table
$\lvert\mathcal{T}\rvert_{\text{row}}$	The number of rows in table $\mathcal{T}$
$\lvert\mathcal{T}\rvert_{\text{col}}$	The number of columns in table $\mathcal{T}$
$\lvert\mathcal{Q}\rvert$	The number of predicates in query $\mathcal{Q}$
$[\mathcal{T}]$	The encrypted database table
$[\mathcal{Q}]$	The encrypted SQL query
$[\mathcal{P}]$	The encrypted predicate
$[\mathcal{F}]$	The encrypted filter result
$[\mathcal{R}]$	The encrypted query result
λ	The security parameter
p	The plaintext modulus
q	The ciphertext modulus for LWE and $\widehat{\text{LWE}}$
Q	The ciphertext modulus for RLWE and $\widehat{\text{RLWE}}$
Q'	The ciphertext modulus for RGSW and $\widehat{\text{RGSW}}$
n	The lattice dimension for LWE and $\widehat{\text{LWE}}$
N	The lattice dimension for an RLWE and $\widehat{\text{RLWE}}$
N'	The lattice dimension for an RGSW and $\widehat{\text{RGSW}}$
l	The number of RLWE in RGSW
$\mathbb{Z}_q^n$	The set of n-vectors over $\mathbb{Z}_q$
R	The cyclotomic ring $\mathbb{Z}[X]/(X^N+1)$
R_Q	The cyclotomic ring $\mathbb{Z}_{\mathbb{Q}}[X]/(X^N+1)$
χ	The noise distribution
Δ	The scaling factor
β	The radix base
ω	The modular ciphertext chunks
$\mathfrak{d}$	The automorphism parameter
$\mathbf{a}$	An element in vector domain
$\mathbf{a}_i$	The i-th element of $\mathbf{a}$
$\tilde{a}$	An element in polynomial ring
$\tilde{a}_i$	The i-th coefficient of $\tilde{a}$
$\mathbf{A}$	An element in matrix domain
$\hat{a}$	A modular integer
$\hat{\tilde{m}}$	A modular polynomial
$\text{LWE}_{\mathbf{s}}^{n,q}(m)$	An LWE ciphertexts encrypting m with parameters (n,q) and secret $\mathbf{s}$
$\text{RLWE}_{\tilde{s}}^{N,Q}(\tilde{m})$	An RLWE ciphertexts encrypting $\tilde{m}$ with parameters (N,Q) and secret $\tilde{s}$
$\text{RGSW}_{\tilde{s}}^{N',Q'}(\mathbf{m})$	An RGSW ciphertext encrypting $\mathbf{m}$ with parameters (N',Q') and secret $\tilde{s}$
$\widehat{\text{LWE}}_{\mathbf{s}}^{n,q}(\hat{m})$	A modular LWE ciphertexts encrypting $\hat{m}$ with parameters (n,q) and secret $\mathbf{s}$
$\widehat{\text{RLWE}}_{\tilde{s}}^{N,Q}(\hat{\tilde{m}})$	An RLWE ciphertexts encrypting $\hat{\tilde{m}}$ with parameters (N,Q) and secret $\tilde{s}$
$\widehat{\text{RGSW}}_{\tilde{s}}^{N',Q'}(\hat{\tilde{m}})$	An RGSW ciphertext encrypting $\hat{\tilde{m}}$ with parameters (N',Q') and secret $\tilde{s}$

Appendix Table A2: Summary of Operators

Operator	Description
π	The exponent encoding
$\diamond$	Homomorphic matrix-vector multiplication
`Automorphism`	Homomorphic automorphism [50]
`ExternalProduct`	Homomorphic External Product [29]
$+, -, \cdot$	Addition, subtraction and multiplication
`CMUX`	Homomorphic selector [29]
`BlindRotate`	Blind rotate [29]
`PBS`	Programmable bootstrapping [30]
`HomGate`	Homomorphic gate [29, 42]
$\bar{ct}$	The NOT gate result of ct
`RLWEtoLWEs`	Converting RLWE to LWEs (a.k.a, sample extract index [29])
`LWEstoRLWE`	Converting LWEs to RLWE (a.k.a, repack [23, 78])
`LWEtoRGSW`	Converting LWE to RGSW (a.k.a., circuit bootstrapping [29])



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

Appendix Table A3: Test vector of `HCMP` on different comparison operator $\leq, <, \geq, >, ==, <>$.

Comparison	Test Vector	μ
$a \leq b$	$TV = \mu + \mu X + \cdots + \mu X N - 1$	$1/2$
$a < b$	$TV = -\mu + \mu X + \cdots + \mu X N - 1$	$1/2$
$a \geq b$	$TV = \mu - \mu X - \cdots - \mu X N - 1$	$1/2$
$a > b$	$TV = -\mu - \mu X - \cdots - \mu X N - 1$	$1/2$
$a == b$	$TV = \mu$	1

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 $\leq, <, \geq, >, ==$ on Table A3, and the $<>$ operator can be evaluated by the NOT gate of the result of $==$.

We detail how to construct high-precision comparisons based on low-precision comparisons on operator $\leq, <, \geq, >, ==, <>$ in Table A4.

Appendix Table A4: Constructing high-precision comparison based on low-precision comparisons.Taking $a = \{a_0 a_1\}$ and $b = \{b_0 b_1\}$ as an example.

Comparison	Expression
$a \le b$	$a_1 == b_1 ? a_0 \le b_0 : a_1 \le b_1$
$a < b$	$a_1 == b_1 ? a_0 < b_0 : a_1 < b_1$
$a \ge b$	$a_1 == b_1 ? a_0 \ge b_0 : a_1 \ge b_1$
$a > b$	$a_1 == b_1 ? a_0 > b_0 : a_1 > b_1$
$a == b$	$a_1 == b_1 \wedge a_0 == b_0$
$a <> b$	$a_1 <> b_1 \vee a_0 <> b_0$

Appendix Table A5: The Proposed Parameter Sets

Ciphertext Format	Parameters
LWE & $\widehat{\text{LWE}}$	$n = 1024, \lceil \log_2 q \rceil = 32$
RLWE & $\widehat{\text{RLWE}}$	$N = 1024, \lceil \log_2 Q \rceil = 32$ $N = 4096, \lceil \log_2 Q \rceil = 109$ $N = 32768, \lceil \log_2 Q \rceil = 720$
RGSW & $\widehat{\text{RGSW}}$	$N' = 1024, \lceil \log_2 Q' \rceil = 32$ $N' = 2048, \lceil \log_2 Q' \rceil = 64$ $N' = 4096, \lceil \log_2 Q' \rceil = 109$

C.2 SIMD Homomorphic MUX

Our core concept in designing SIMDCMUX 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 SIMDCMUX presented in Algorithm 6 involves the following steps. Given three sets of N-sized LWE ciphertexts $\{\text{LWE}_{\mathbf{s}}^{n,q}(\mathfrak{t}_i)\}$, $\{\text{LWE}_{\mathbf{s}}^{n,q}(\mathfrak{a}_i)\}$, $\{\text{LWE}_{\mathbf{s}}^{n,q}(\mathfrak{b}_i)\}$, 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 $b_i + \mathbf{a}_i \cdot \mathbf{s}_i$ is equal to $(4\mathfrak{t}_i + 2\mathfrak{a}_i + \mathfrak{b}_i) \cdot \lfloor q/8 \rfloor + \lfloor q/16 \rfloor$ 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 $\mathbf{b} = [b_0, b_1, ..., b_{N-1}]$ and a ciphertext matrix $\mathbf{A} = [\mathbf{a_0}, \mathbf{a_1}, ..., \mathbf{a_{N-1}}]^T \in \mathbb{Z}_q^{N \times n}$. Next on line 5 – 6, we apply the homomorphic matrix multiplication operations [58, 66, 78] to evaluate $\mathbf{A} \diamond \mathbf{BTK}$ and get $ct = \text{RLWE}_{\tilde{s}'}^{N,Q}(\mathcal{E}(\mathbf{As}))$. After homomorphically adding $\mathbf{b}$ to ct, we obtain $ct_l = \text{RLWE}_{\tilde{s}'}^{N,Q}(\mathcal{E}(\mathbf{As} + \mathbf{b}))$. Here ct_l encrypts $\mathbf{A} \cdot \mathbf{s} + b = \mathbf{m} + t\mathbf{q}$, and the $t\mathbf{q}$ term is automatically removed as ct_l is an RLWE ciphertext with plain modulus q. Thus, ct_l is encrypting $\mathbf{m}$ where $m[i] = b_i + \mathbf{a}_i \cdot \mathbf{s}_i$. The next step (line 7 – 8) is the evaluation of a specific polynomial $p_{mux}(x) : \mathbb{Z}q \to \mathbb{Z}q$. We define the polynomial $p_{mux}(x)$ as

$$p_{mux}(x) = \begin{cases} \lfloor q/p \rfloor & \lceil x / \lfloor q/8 \rfloor \rfloor \in [1,2] \cup [3,4] \cup [6,8] \\ 0 & \lceil x / \lfloor q/8 \rfloor \rfloor \in [0,1] \cup [2,3] \cup [4,6] \end{cases} \tag{A1}$$

Simplicity, the value $\mathbf{m}[i] = b_i + \mathbf{a}_i \cdot \mathbf{s}+$ will falls into the interval $[(4\mathfrak{t}_i + 2\mathfrak{a}_i + \mathfrak{b}_i) \cdot \lfloor q/8 \rfloor, (4\mathfrak{t}_i + 2\mathfrak{a}_i + \mathfrak{b}_i + 1) \cdot \lfloor q/8 \rfloor]$. If $\mathfrak{t}_i ? \mathfrak{a}_i$: $\mathfrak{b}_i$ is 1, we define the value of $p_{mux}(x)$ on the the interval $[(4\mathfrak{t}_i + 2\mathfrak{a}_i + \mathfrak{b}_i) \cdot \lfloor q/8 \rfloor, (4\mathfrak{t}_i + 2\mathfrak{a}_i + \mathfrak{b}_i + 1) \cdot \lfloor q/8 \rfloor]$ is $\lfloor q/p \rfloor$, otherwise 0. As explained in Section 2.2, $p_{mux}(x)$ can be directly applied to the RLWE ciphertext ct_l. Thus, after line 8, we get the ciphertext $ct_p = \text{RLWE}_{\tilde{s}'}^{N,Q}(\mathcal{E}(\mathfrak{c}_0, \mathfrak{c}_1, ..., \mathfrak{c}_{N-1}))$ and $\mathfrak{c}_i = (\mathfrak{t}_i ? \mathfrak{a}_i : \mathfrak{b}_i) \cdot \lfloor q/p \rfloor$. Next, we follow [17, 25, 73] to apply SlotToCoeff on ct_p and gets $ct_c = \text{RLWE}_{\tilde{s}'}^{N,Q}(\tilde{\mathfrak{c}})$. Here $\tilde{\mathfrak{c}}_i = (\mathfrak{t}_i ? \mathfrak{a}_i : \mathfrak{b}_i) \cdot \lfloor q/p \rfloor$. After we perform the modulus switching (line 9), we change back the ciphertext with plain modulus p and ciphertext modulus q and gets $ct_m = \text{RLWE}_{\tilde{s}'}^{N,q}(\tilde{\mathfrak{c}})$. Now, the value $\tilde{\mathfrak{c}}_i$ is equal to $(\mathfrak{t}_i ? \mathfrak{a}_i : \mathfrak{b}_i)$. Finally, we extract the N LWE ciphertexts from ct_m and key switch the LWE ciphertext to the original parameters (line 12 – 14) and gets $(\text{LWE}_{\mathbf{s}}^{n,q}(\mathfrak{c}_0), \text{LWE}_{\mathbf{s}}^{n,q}(\mathfrak{c}_1), ..., \text{LWE}_{\mathbf{s}}^{n,q}(\mathfrak{c}_{N-1}))$ where $\mathfrak{c}_i = (\mathfrak{t}_i ? \mathfrak{a}_i : \mathfrak{b}_i)$ and the noise of the output is independent of the input ciphertext.

Algorithm 6: SIMDCMUX

Input : $3N$ input LWE ciphertexts with plain modulus p $(\text{LWE}_{\mathbf{s}}^{n,q}(\mathfrak{t}_0), \text{LWE}_{\mathbf{s}}^{n,q}(\mathfrak{t}_1)..., \text{LWE}_{\mathbf{s}}^{n,q}(\mathfrak{t}_{N-1}),$ $(\text{LWE}_{\mathbf{s}}^{n,q}(\mathfrak{a}_0), \text{LWE}_{\mathbf{s}}^{n,q}(\mathfrak{a}_1)..., \text{LWE}_{\mathbf{s}}^{n,q}(\mathfrak{a}_{N-1}),$ $(\text{LWE}_{\mathbf{s}}^{n,q}(\mathfrak{b}_0), \text{LWE}_{\mathbf{s}}^{n,q}(\mathfrak{b}_1)..., \text{LWE}_{\mathbf{s}}^{n,q}(\mathfrak{b}_{N-1}).$

Input : A Batch bootstrapping key $\mathbf{BTK} = \text{RLWE}_{\tilde{s}'}^{N,Q}(\mathcal{E}(\mathbf{s}))$ with the plain modulus q.

Input : An LWE key switching key **KSK**.

Output : N LWE ciphertexts $\mathbf{ct_O} = (\text{LWE}_{\mathbf{s}}^{n,q}(\mathfrak{c}_0), \text{LWE}_{\mathbf{s}}^{n,q}(\mathfrak{c}_1), ..., \text{LWE}_{\mathbf{s}}^{n,q}(\mathfrak{c}_{N-1}))$ where $\mathfrak{c}_i = \mathfrak{t}_i ? \mathfrak{a}_i : \mathfrak{b}_i$.

1 $scale \leftarrow \lfloor p/8 \rfloor, offset \leftarrow \lfloor q/16 \rfloor$
2 **for** $i = 0$ **to** $N - 1$ **do**
3 $\quad (b_i, \mathbf{a_i}) \leftarrow scale \cdot (4\text{LWE}(\mathfrak{t}_i) + 2\text{LWE}(\mathfrak{a}_i) + \text{LWE}(\mathfrak{b}_i))$
4 $\quad b_i \leftarrow b_i + offset$
5 Let $\mathbf{b} \leftarrow [b_0, b_1, ..., b_{N-1}]$, $\mathbf{A} \leftarrow [\mathbf{a_0}, \mathbf{a_1}, ..., \mathbf{a_{N-1}}]^T \in \mathbb{Z}_q^{N \times n}$
6 $ct \leftarrow \mathbf{A} \diamond \mathbf{BTK}$; ▷ $ct = \text{RLWE}_{\tilde{s}'}^{N,Q}(\mathcal{E}(\mathbf{As}))$
7 $ct_l \leftarrow ct + (\mathcal{E}(\mathbf{b}), 0)$; ▷ $ct_l = \text{RLWE}_{\tilde{s}'}^{N,Q}(\mathcal{E}(\mathbf{As} + \mathbf{b}))$
8 Generate the evaluation polynomial $p_{mux}(x)$
9 $ct_p \leftarrow p_{mux}(ct_l)$; ▷ $ct_p = \text{RLWE}_{\tilde{s}'}^{N,Q}(\mathcal{E}(\mathfrak{c}_0, \mathfrak{c}_1, ..., \mathfrak{c}_{N-1}))$
10 $ct_c \leftarrow \text{SlotToCoeff}(ct_p)$; ▷ $ct_c = \text{RLWE}_{\tilde{s}'}^{N,Q}(\tilde{\mathfrak{c}})$
11 $ct_m \leftarrow \text{ModSwitch}(ct_c)$; ▷ $ct_m = \text{RLWE}_{\tilde{s}'}^{N,q}(\tilde{\mathfrak{c}})$
12 $\mathbf{ct_O} \leftarrow \text{RLWEtoLWEs}(ct_m)$; ▷ $\mathbf{ct_O}[i] = \text{LWE}_{s'}^{N,q}(\mathfrak{c}_i)$
13 **for** $i = 0$ **to** $N - 1$ **do**
14 $\quad \mathbf{ct_O}[i] \leftarrow \text{KeySwitch}(\mathbf{ct_O}[i], \mathbf{KSK})$; ▷ $\text{LWE}_{\mathbf{s}}^{n,q}(\mathfrak{c}_i)$

Return : $\mathbf{ct_O}$

C.3 Homomorphic Sorting and Synchronization

We present the detail algorithm for SortSynchronize in Algorithm 7 and HomSort in Algorithm 8.

The Algorithm 7 includes the following steps. Given L modular RGSW ciphertext $\mathbf{ct} = (\widehat{\text{RGSW}}(\pi(\hat{a}_0)), \widehat{\text{RGSW}}(\pi(\hat{a}_1)), \cdots, \widehat{\text{RGSW}}(\pi(\hat{a}_{L-1}))$ where $\hat{a}_i = \sum_{j=0}^{\omega-1} a_{i,j} 2^j$ and L another modular RLWE ciphertexts

Appendix Table A6: Complexity Comparisons Between Liu *et al.* [76], HE^{3}DB [16], `ArbHCMP` and `SIMDArbHCMP` in ArcEDB.

		Bootstrap complexity	Precision (k)	#bootstrap	Bootstrap parameters
Liu *et al.* [76]		$2 \cdot \lceil k/4 \rceil - 1$	16	7	$n = 2048, \lceil \log_2 q \rceil = 29$
			32	15	$n = 4096, \lceil \log_2 q \rceil \leq 109$
			64	31	$n = 4096, \lceil \log_2 q \rceil \leq 109$
HE^{3}DB [17]		$2 \cdot \lceil k/5 \rceil - 1$	16	7	$n = 2048, \lceil \log_2 q \rceil = 64$
			32	13	$n = 2048, \lceil \log_2 q \rceil = 64$
			64	25	$n = 4096, \lceil \log_2 q \rceil = 128$
ArcEDB	`ArbHCMP`	$\lceil k/10 \rceil - 1$	16	1	$N = 1024, \lceil \log_2 Q \rceil = 32$
			32	3	$N = 1024, \lceil \log_2 Q \rceil = 32$
			64	6	$N = 1024, \lceil \log_2 Q \rceil = 32$
	`SIMDArbHCMP`	$\lceil k/12 \rceil - 1$	16	1	$N = 32768, \lceil \log_2 Q \rceil = 660$
			32	2	$N = 32768, \lceil \log_2 Q \rceil = 660$
			64	5	$N = 32768, \lceil \log_2 Q \rceil = 660$

$\left(\widehat{\mathsf{RLWE}}(\hat{m}_0), \cdots, \widehat{\mathsf{RLWE}}(\hat{m}_{L-1})\right)$ where $\hat{m}_i = \sum_{i=0}^{\omega-1} m_{i,j}\beta^i$. First in Line 3, for each modular RLWE ciphertext $\widehat{\mathsf{RLWE}}(\hat{m}_i)$, we multiply the $\widehat{\mathsf{RLWE}}(\hat{m}_i)$ with $X^0, X^1, \cdots, X^{L-1}$ and result $\mathbf{ct}^{exp} = \left(X^0\widehat{\mathsf{RLWE}}(\hat{m}_i), \cdots, X^{L-1}\widehat{\mathsf{RLWE}}(\hat{m}_i)\right)$. Next in Line 4-6, we utilize the modular RGSW ciphertext as the control signal and evaluate a CMUX tree on $\mathbf{ct}^{exp}$ to get $\widehat{\mathsf{RLWE}}(\hat{m}_i X^{\hat{a}_i})$. After that, in Line 7, we summarize the L modular ciphertext and obtain $ct_{res} = \sum_{i=0}^{L-1} \widehat{\mathsf{RLWE}}(\hat{m}_i X^{\hat{a}_i})$. 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 $\left(\widehat{\mathsf{LWE}}(\hat{m}_{s_0}), \cdots, \widehat{\mathsf{LWE}}(\hat{m}_{s_{L-1}})\right)$. The sequence $s_0, \cdots s_{L-1}$ satisfy $\hat{a}_{s_0} \leq \hat{a}_{s_1} \leq \ldots \hat{a}_{s_{L-1}}$.

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

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^{3}DB [16].

Algorithm 7: `SortSynchronize`

Input : L modular RGSW ciphertexts $\mathbf{ct} = \left(\widehat{\mathsf{RGSW}}(\pi(\hat{a}_0)), \widehat{\mathsf{RGSW}}(\pi(\hat{a}_1)), \cdots, \widehat{\mathsf{RGSW}}(\pi(\hat{a}_{L-1}))\right)$ where $\hat{a}_i = \sum_{j=0}^{\omega-1} a_{i,j}2^j$.

Input : L another modular RLWE ciphertexts $\left(\widehat{\mathsf{RLWE}}(\hat{m}_0), \cdots, \widehat{\mathsf{RLWE}}(\hat{m}_{L-1})\right)$ where $\hat{m}_i = \sum_{i=0}^{\omega-1} m_{i,j}\beta^i$.

Output : L synchronized modular LWE ciphertexts $\left(\widehat{\mathsf{LWE}}(\hat{m}_{s_0}), \cdots, \widehat{\mathsf{LWE}}(\hat{m}_{s_{L-1}})\right)$. The sequence $s_0, \cdots s_{L-1}$ satisfy $a_{s_0} \leq a_{s_1} \leq \ldots a_{s_{L-1}}$.

1 Initialize $ct_{res} = \widehat{\mathsf{RLWE}}(0)$
2 **for** $i = 0$ **to** $L-1$ **do**
3 $\quad \mathbf{ct}^{exp} \leftarrow \left(X^0\widehat{\mathsf{RLWE}}(\hat{m}_i), \cdots, X^{L-1}\widehat{\mathsf{RLWE}}(\hat{m}_i)\right)$
4 $\quad$ **for** $j = 0$ **to** $\log L - 1$ **do**
5 $\quad\quad$ **for** $t = 0$ **to** $2^{\log L - i - 1} - 1$ **do**
6 $\quad\quad\quad \mathbf{ct}^{exp}_{2^{j+1}\cdot t} \leftarrow \mathsf{CMUX}(\mathbf{ct}[i][j], \mathbf{ct}^{exp}_{2^{j+1}\cdot t + 2^j}, \mathbf{ct}^{exp}_{2^{j+1}\cdot t})$
7 $\quad ct_{res} \leftarrow ct_{res} + \mathbf{ct}^{exp}_0$
8 **for** $i = 0$ **to** $L-1$ **do**
9 $\quad \mathbf{ct}_O \leftarrow \mathsf{RLWEtoLWEs}(ct_{res})[i]$

Return : ct_O

```
SELECT COUNT(*) FROM MedicalHistory
WHERE (systolic < 90 OR diastolic < 50 OR
       weight_gain > 2 OR heart_rate < 40
       OR heart_rate > 90) AND (time BETWEEN
       2021:07:01:00:00 AND 2021:08:01:00:00)
```

(a) TQ1.

```
SELECT COUNT(*) FROM MobileHealth
WHERE glucose < 70 OR glucose > 100 AND
      time > 2023:12:01:00:00
GROUP BY time(1m)
```

(b) TQ2.

```
SELECT COUNT(passenger_count) FROM passengers
WHERE time = 2021:07:01:00:00
      AND VendorID = 2 AND RatecodeID = 2
```

(c) TQ3.

```
SELECT SUM(fare_amount) FROM fare
WHERE (time BETWEEN 2016:01:01:00:00 AND
       2016:01:03:00:00)
```

(d) TQ4.

Appendix Figure A2: The detail time-series queries benchmark in Section 6.3.

Algorithm 8: The homomorphic sorting operator HomSort

Input : L modular RLWE ciphertexts $\mathbf{ct_a} = \left(\widehat{\mathsf{RLWE}}\big(\pi(\hat{a}_0)\big), \cdots, \widehat{\mathsf{RLWE}}\big(\pi(\hat{a}_{L-1})\big)\right)$.

Input : L another modular RLWE ciphertexts $\mathbf{ct_m} = \big(\widehat{\mathsf{RLWE}}(\hat{m}_0), \cdots, \widehat{\mathsf{RLWE}}(\hat{m}_{L-1})\big)$ where $\hat{m}_i = \sum_{i=0}^{\omega-1} m_{i,j}\beta^i$.

Output : L synchronized modular LWE ciphertexts $\big(\widehat{\mathsf{LWE}}(\hat{m}_{s_0}), \cdots, \widehat{\mathsf{LWE}}(\hat{m}_{s_{L-1}})\big)$. The sequence $s_0, \cdots s_{L-1}$ satisfy $a_{s_0} \le a_{s_1} \le \ldots a_{s_{L-1}}$

for $i = 0$ **to** $L-1$ **do**
  for $j = 0$ **to** $i-1$ **do**
    $\mathbf{A}[i][j] \leftarrow 1 - \mathbf{A}[j][i]$
  $\mathbf{A}[i][i] \leftarrow \mathsf{RLWE}(0)$
  for $j = i+1$ **to** $L-1$ **do**
    $\mathbf{A}[i][j] \leftarrow \mathsf{ArbHCMP}_{<}\big(\mathbf{ct_a}_i, \mathbf{ct_a}_j\big)$
Initialize $\mathbf{Id} = \big(\mathsf{LWE}_0(0), \cdots, \mathsf{LWE}_{L-1}(0)\big)$
for $i = 0$ **to** $L-1$ **do**
  for $j = 0$ **to** $L-1$ **do**
    $\mathbf{Id}[i] \leftarrow \mathbf{Id}[i] + A[i][j]$
for $i = 0$ **to** $L-1$ **do**
  $\mathbf{BitId}[i] \leftarrow \mathsf{Decompose}(\mathbf{Id}[i])$
  for $j = 0$ **to** $\log L - 1$ **do**
    $\mathbf{BitId}[i][j] \leftarrow \mathsf{LWEtoRGSW}(\mathbf{BitId}[i][j])$
$\mathbf{ct}_O \leftarrow \mathsf{SortSynchronize}(\mathbf{BitID}, \mathbf{ct_m})$

Return : $\mathbf{ct}_O$

D.3 Clarifications for Figure 4

We conclude the concrete complexity comparisons between Liu *et al.* [76], HE^3DB [16] and ArcEDB in Table A6. Moreover, we provide more clarifications for experimental differences between HE^3DB [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× 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× gain). Therefore, for 16-bit comparisons, ArcEDB is around 14× 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×–28×.