HEDA: Multi-Attribute Unbounded Aggregation over Homomorphically Encrypted Database

Xuanle Ren
Le Su
Alibaba Group, {xuanle.rxl, le.su}@alibaba-inc.com

Zhen Gu
Sheng Wang
Alibaba Group, {guzhen.gz, sh.wang}@alibaba-inc.com

Feifei Li
Yuan Xie
Alibaba Group, {lifeifei, y.xie}@alibaba-inc.com

Song Bian
Kyoto University, Alibaba Group,
sbian@easter.kuee.kyoto-u.ac.jp

Chao Li
Zhejiang University,
lichao42@zju.edu.cn

Fan Zhang
Zhejiang University,
fanzhang@zju.edu.cn

ABSTRACT

Recent years have witnessed the rapid development of the encrypted database, due to the increasing number of data privacy breaches and the corresponding laws and regulations that caused millions of dollars in loss. These encrypted databases may rely on different techniques, such as cryptographic primitives and trusted execution environments. In this work, we investigate the feasibility of utilizing fully homomorphic encryption (FHE) to support unbounded database aggregation queries, which typically involve comparisons as filtering predicates and a final aggregation. These operators are theoretically supported by FHE, but need careful algorithm design to maximize the efficiency and have not been explored before. We creatively use two types of FHE schemes, *i.e.*, one for numerical and one for binary value, to enjoy their advantages respectively. To bridge the encrypted values between these two schemes for seamless query processing without client-server interaction, we propose a novel ciphertext transformation mechanism, which is of independent research interest, to close this gap. We further implement our system and test it over three TPC-H queries and a query over a real social media e-commerce database. Evaluation results show that, to process an aggregation query over $8k$ encrypted rows takes about 430 seconds. Although it is slower than plaintext processing in magnitudes and still has much room for improvement, as the very first work in this domain, our system demonstrates the feasibility of using FHE to process OLAP queries.

PVLDB Reference Format:
Xuanle Ren, Le Su, Zhen Gu, Sheng Wang, Feifei Li, Yuan Xie, Song Bian, Chao Li, and Fan Zhang. *HEDA*: Multi-Attribute Unbounded Aggregation over Homomorphically Encrypted Database. PVLDB, 16(4): 601 - 614, 2022.
doi:10.14778/3574245.3574248

1 INTRODUCTION

With the incomparable advantage in storage capacity and computing capability, cloud services have changed the landscape of how enterprises build their business. By moving the whole business logic and associated data to the cloud at a relatively reasonable cost, an enterprise could enjoy the hassle-free, one-stop service to manage their own business. Database Management Systems (DBMS) is one of the many examples that benefited from the cloud service. Cloud-based DBMS, such as Microsoft Azure SQL Server [43], AWS Aurora [3], Alibaba Cloud PolarDB [10], help millions of businesses to store and analyze their data on a daily basis. One of the fundamental assumptions in such DBMS is that the data owners *trust* the cloud service providers (CSP), as their invaluable data are in the hands of the CSP. However, this is not always the case: CSP could be compromised by cyber-attacks that leak its users' data. More severely, a malicious database administrator (DBA) could covertly steal sensitive information from its clients without being noticed. Traditional approaches such as transparent data encryption can mitigate such concern to a certain degree, by protecting the data at rest and against third-party intruders. However, these approaches still authorize the database system with the decryption key to obtain the plaintext data whenever computation is required.

Over the past decade, researchers have progressively proposed, improved and commercialized *encrypted database* to protect user data privacy while enabling encrypted data processing and maintaining query expressiveness. Cryptography-based approaches, such as CryptDB [51], MONOMI [59], Seabed [48] and SAGMA [31], explore various cryptographic primitives such as deterministic encryption (DET), searchable encryption (SE), partially homomorphic encryption (PHE) and order preserving/revealing encryption (OPE, ORE) *etc.*. Another streamline of work, such as TrustedDB [4], Cipherbase [2] and ObliDB [22], leverage on TEE to put the sensitive data into a hardware-based trust zone called enclave for processing, and encrypt both the data and processing results whenever they leave the enclave. The TEE approach eventually leads to commercialized encrypted database systems, *e.g.*, Microsoft SQL Server Always Encrypted [1] and Alibaba Cloud Operon [63].

Limitations still exist for the above two approaches. First, TEE-based solutions require the end users to fully trust the TEE hardware provider, and suffer from numerous side-channel attacks [45, 60]. Cryptographic solutions, although based on rigorous mathematical proofs, they suffer from query expressiveness limitations, *i.e.*, typically only support queries with aggregation (*e.g.*, `SUM` through PHE), range queries (OPE/ORE), or matching (DET/SE). More importantly, various attacks [11, 29, 34, 44] that target SE and OPE/ORE by exploring information leakage (*e.g.*, access pattern) apply.

Song Bian (sbian@easter.kuee.kyoto-u.ac.jp) is the corresponding author.
Xuanle Ren and Le Su contributed equally to this work.

Proceedings of the VLDB Endowment, Vol. 16, No. 4 ISSN 2150-8097.
doi:10.14778/3574245.3574248

Fully homomorphic encryption, known as the holy grail of cryptography, enables an unlimited number of additions and multiplications over the ciphertexts. The security of FHE does not rely on any hardware assumption, and is also fully oblivious and thus does not leak any access pattern. Further, from a theoretical perspective, FHE is Turing complete. This gives hope to constructing a DBMS that solely over FHE that can achieve various operations (*e.g.*, aggregation, equality and range queries) that used to rely on the combination of multiple cryptographic primitives (*e.g.*, the approach of CryptDB). Since the foundational work by Gentry [26], over the past decade, a series of works emerged in the direction of primitive improvement [9, 15, 16, 23, 27], hardware acceleration [24, 38, 54], and particularly, its applicability in cloud-based machine/deep learning [41, 53]. However, to the best of our knowledge, FHE has not been extensively explored in the encrypted database domain due to the challenge of translating SQL queries into FHE-compatible operators and designing database-orient solutions that fully leverage the potential of FHE.

In this work, we utilize FHE to support database aggregation queries with filtering predicates. Such queries, *e.g.* TPC-H Query 1, typically involves one or more aggregation operators (*e.g.*, `SUM`, `COUNT`), and several filtering predicates. These computations are, theoretically, additions and comparisons that most FHE schemes support. However, using FHE naïvely, with no client-server interaction and with a high security level against leakage attacks induces a significant amount of computational overheads. The challenges come from three aspects:

SQL and FHE compatibility: aggregation query first filters out non-satisfying rows and then aggregates those remaining values. However, FHE computations are oblivious, where the encrypted filtering results can not be decrypted by the server. These encrypted results have to propagate homomorphically according to the logical operators appeared in the query, and further with the aggregation attribute values to obtain the correct aggregation result;

Supporting unbounded aggregation queries: Users tend to issue analytical queries in various complexity according to different business needs, and thus prefer a system setup that is *independent* of the query that might be issued. How to utilize FHE to support such functionality has not yet been extensively explored;

Selecting the correct FHE schemes to improve SQL efficiency: FHE is notorious for its complicated mathematical structure and security parameter setting. A preferred FHE-based query analytical system should hide this complicacy from the end user, and build the system with multiple FHE schemes catered for different scenarios to enjoy the maximized efficiency.

To address the above challenges, we propose a system called *HEDA*[*], which could filter out the non-satisfying predicates in an oblivious way by multiplying the filtering results with the aggregation attribute blindly. Further, depending on the data and operation type, this work employs the TFHE [16] scheme to address encrypted binary operations, and the BFV [23] scheme for arithmetic operations. We leverage packed encryption in both approaches to encrypt a batch of plaintext into a single ciphertext, which improves the overall processing efficiency. Our main contributions are as follows:

- We comprehensively study the applicability of FHE to encrypted database domain. In particular, we thoroughly investigate and propose the translation mechanism of various SQL operators into FHE-compatible operations (Sec. 4);
- Our proposed solution supports aggregation queries with an *unbounded* number of filtering predicates. In a conventional leveled homomorphic encryption scheme, the encryption parameters depend on the query complexity, and the entire database needs to be re-encrypted if a new query involves more complicated computation. In contrast, the parameter setting in our scheme is *query independent* (Sec. 5);
- To support the aggregation queries (*i.e.*, filtering followed by aggregation operations) only through FHE, we propose a series of novel cryptographic techniques, seamlessly integrate the FHE schemes that handle binary and arithmetic operations, such that complicated SQL queries can be processed smoothly (Sec. 5). This mechanism is also of independent interest to the FHE research community;
- We conduct comprehensive experiments using real business data and TPC-H queries to validate our system, and demonstrate the feasibility of using FHE schemes to encrypt and process analytical queries (Sec. 6).

2 RELATED WORK

2.1 Homomorphic Encryption

Homomorphic encryption (HE) allows one to perform data manipulation over encrypted data. HE is a powerful tool for data privacy protection while accomplishing computational tasks. Breaking down to more fundamental operations, typically an HE scheme supports addition and/or multiplication. A partially homomorphic encryption (PHE) scheme supports only one operation, such as plaintext addition (*e.g.*, Paillier [46]) or multiplication (*e.g.*, RSA [55], ElGamal [21]). Somewhat homomorphic encryption (SWHE) and leveled homomorphic encryption (LHE) support both operations, but one could only perform a pre-defined, *limited* number of multiplications (though the number of the addition operation is unlimited). Since the seminal work [26], many FHE schemes that support a polynomial or an unlimited number of both operations have been proposed [9, 15, 16, 23, 27]. In addition, homomorphic conversion algorithms among such schemes are also proposed [8, 41]. Open-source libraries, such as IBM HElib [33], Microsoft SEAL [57], HEAAN [32], TFHE [16], and Palisade [47] are also available with some of the above cryptosystems implemented, with different realizations and optimization.

2.2 Cryptography-based Encrypted DBMS

Over the years, the advance in cryptography improved the efficiency and functionalities of the encrypted DBMS. Schemes proposed include using PHE [46], SE [12, 19] and OPE/ORE [7, 14]. A streamline of encrypted database research that employs these primitives emerge, following the pioneering work [30].

In [25], the authors propose to use the Paillier PHE scheme [46] and an OPE system to address the encrypted addition and filtering task in a query, with a packed plaintext concept. However, PHE greatly limits the query expressive, and many other operations could not be processed under the packed plaintext scenario.

[*] *HEDA* stands for **H**omomorphically **E**ncrypted **D**atabase **A**nalytics

CryptDB [51] and the subsequent MONOMI [59] employ DET, OPE, and Paillier to enable a much richer query set. However, by using the less secure cryptographic primitives, both schemes leak a non-trivial amount of plaintext information. To address this limitation, Arx [49] encrypts a DBMS with either AES, DET, or SE that is semantically secure. Nevertheless, when a query requires plaintext matching, Arx still leaks the search pattern or histogram count of that particular operation. Seabed [48], and subsequently [56] inspired by Seabed, proposed symmetrical versions of the additive and multiplicative PHE schemes. Despite both schemes using DET and OPE for equality and range operations that inherit the limitation of leaking plaintext information, the proposed PHE schemes have a ciphertext size that grows w.r.t the number of items being aggregated, which could be a problem in a real-life scenario where tens of millions of data need to be processed. A recent work, SAGMA [31], utilizes an SWHE and an SE scheme to handle aggregation queries with multiple filtering attributes. Apart from using SE that might potentially leak the search pattern for filtering conditions, the use of underlying SWHE that supports only one multiplication result on the server-side has to store an exponentially-growing set of monomials used for assisting the aggregation of multiple attributes. We also note an unpublished work [39], which has a similar concept to ours by utilizing LHE to address aggregated queries. The main difference is that the complexity of [39] is *query* dependent, *i.e.*, the client has to fix the encryption parameters *a priori* based on the to-be-issued query itself, and the system could not process (future) queries that are more complicated: the client has to decrypt and re-encrypt the entire database should such a need arises. On the contrary, *HEDA* is *query independent*, and supports an unbounded number of filtering predicates and thus more complex queries.

Works such as Cheetah [53] and F1 [24], focus more on FHE acceleration and optimization, and using queries like equality test or deployed with a K-V store to show its applicability. Works like [36, 58] discuss the conjunctive queries with comparison, but do not involve aggregation, and only stay as theoretical research. To address join-aggregate queries with multiple users, solutions based on secure multi-party computation (MPC) have been proposed over recent years (*e.g.*, SMCQL [5], Conclave [62], Secure Yannakakis [64], Secrecy [40] and Senate [50]). We note that *HEDA* is both orthogonal and complementary to MPC approaches: FHE can be used when a single participating party (*e.g.*, the client) is in the hold of all of the private information, and wishes to outsource some computations to another party (*e.g.*, the CSP), while the execution of MPC requires that none of the participating parties can recover all the secret data (e.g., additive secret sharing). Essentially, FHE and MPC focus on different scenarios. Hence, in this work, we focus on the use of FHE in designing secure protocol for outsourcing DBMS, and consider MPC-based DBMS schemes to be out of the scope of this work.

2.3 TEE-based Encrypted DBMS

There is another line of research, exploring the security brought by the TEE, to build an encrypted database and process the queries. Earlier attempts such as TrustedDB [4] and Cipherbase [2] explore the TEE with connection on PCI-X or PCIe. With more enclave resources, Haven [6] and EdgelessDB [28] employ TEE directly, instead of earlier works that use PCIe-connected TEE with physical isolation. Subsequent systems, such as StealthDB [61], EnclaveDB [52], and notably the already-commercialized Microsoft SQL Server Always Encrypted [1] and Alibaba Cloud Operon [63], provide a much richer set of DBMS functionalities and security guarantees. One of the advantages of the TEE-based encrypted DBMS is that they typically enjoy a very expressive set of queries (usually with no, or a minor difference compared to standard, non-encrypted SQL queries). However, the challenge is that the system users have to fully trust the TEE service provider, *i.e.*, Intel for SGX, AMD for SEV. This rather strong assumption may not appeal to users with highly sensitive data, and/or have no requirement for query expressive but rather fixed analytical queries. The pure cryptography based approach provides them with such an alternative.

Table 1: Summary of Notations

Notation	Description
n	The lattice dimension of the LWE/RLWE/RGSW ciphertexts
q	The ciphertext modulus
p, t	The plaintext moduli, where $t \leq p$
$\mathbb{Z}_q$	The ring of integers modulo q
$\mathbb{Z}_q^n$	The set of n-dimensional vectors over $\mathbb{Z}_q$
$\mathbb{Z}[x]$	The polynomial ring over $\mathbb{Z}$
$1[x]$	The polynomial whose coefficients are all 1's
$R_{n,q}$	The cyclotomic ring $\mathbb{Z}_q[X]/(X^n+1)$
a	A single field/ring element
$\mathbf{a}$	A vector
a_i	The i-th component in vector $\mathbf{a}$
r	The base of radix decomposition
ℓ	$\ell = \lfloor q/r \rfloor$
LWE	The set of LWE ciphertexts
RLWE	The set of RLWE ciphertexts
RGSW	The ciphertext encrypting the secret keys for LWE
$\mathcal{E}$(LWE)	The amount of errors contained in the LWE ciphertext

3 CRYPTOGRAPHIC PRELIMINARY

This section introduces the necessary preliminaries related to learning with errors (LWE) cryptography, a fundamental tool for many FHE schemes, together with some basic homomorphic algorithms that serve as the foundation for *HEDA*.

3.1 Notations

In this work, we use n for the lattice dimension, q for the ciphertext modulus, and t, p for the plaintext modulus. Using a separate modulus for filtering (t) from aggregation (p) allows more compact database/query encryption and more lightweight filtering. $\mathbb{Z}_q$ denotes the set of integers modulo q. R_q depicts quotient ring modulo q and an irreducible polynomial, generally taken to be the $2n$-th cyclotomic polynomial. We use normal lower-case letter for ring or field elements (*e.g.*, $a \in R_q$), and bold lower-case letter for vectors (*e.g.*, $\mathbf{a} \in \mathbb{Z}_q^n$). $1[x]$ denotes the polynomial $\sum_{i=0}^{n-1} x^i$, *i.e.*, an order-n polynomial whose coefficients are all 1's. A brief summary of the frequently used variables is provided in Table 1.

3.2 Ciphertext Formats

HEDA employs three encryption methods, namely the ring LWE (RLWE), LWE, and ring GSW (RGSW). Plaintext needs be encrypted to the format required by the specific encryption schemes. Fig. 1 shows an example of the multiplexer whose inputs are encrypted as LWEs while the select signal is encrypted as an RGSW. More precisely, RLWE is used for encrypting a batch of plaintexts, grouped

in vectorized format for a single-instruction multiple-data (SIMD) process, while LWE is used to encrypt a single plaintext value. We also transform ciphertexts between RLWE and LWE format during different processing stages, as detailed in Sec. 5. RGSW is used for encrypting the FHE private key for the bootstrapping, a step to reduce the propagated error during homomorphic computation.

(a) A plain multiplexer (b) An encrypted multiplexer

Figure 1: The multiplexers for plain data and encrypted data.

- **RLWE**: We use $\mathsf{RLWE}_{\mathbf{v},\mathbb{Z}_p}$ and $[\mathbf{v}]_{\mathbb{Z}_p}$ interchangeably to denote an RLWE ciphertext encrypting the vector $\mathbf{v}$ with a plaintext space $\mathbb{Z}_p$. Note that, by default, an RLWE encrypts a vector of plaintexts in a SIMD manner. In the symmetric-key version of the cryptosystem, the ciphertext is of the form

$$\mathsf{RLWE}_{\mathbf{v},\mathbb{Z}_p} = (\mathbf{a}, \mathbf{b}) = (\mathbf{a}, \mathbf{a} * \mathbf{s} + \mathbf{v} + \mathbf{e}), \tag{1}$$

  where $\mathbf{a} \in \mathbb{Z}_q^n$ is an n-dimensional vector that is uniformly randomly distributed over $\mathbb{Z}_q$ for some ciphertext modulus $q \in \mathbb{Z}$, $\mathbf{s} \in \mathbb{Z}_q^n$ is the secret vector, $*$ is the nega-cyclic convolution operator, $\mathbf{v} \in \mathbb{Z}_p^n$ is the plaintext vector for some plaintext modulus $p \leq q \in \mathbb{Z}$, and $\mathbf{e} \in \mathbb{Z}_q^n$ is some error vector. Eq. (1) is known as an RLWE ciphertext with the coefficient representation. Let $\mathbf{b} = [b_0 b_1 \cdots b_{n-1}]$. For RLWE ciphertexts, $\mathbf{b}$ represents the coefficients of some polynomial b, where

$$b = b_0 + b_1 x + b_2 x^2 + \cdots + b_{n-1} x^{n-1} \tag{2}$$

$$= \sum_{i=0}^{n-1} ((\mathbf{a} * \mathbf{s})_i + v_i + e_i) \cdot x^i. \tag{3}$$

  In other words, every coefficient of the polynomial b contains encryption of the corresponding plaintext value $v_i \in \mathbf{v}$.
- **LWE**: We use $\mathsf{LWE}_{v,\mathbb{Z}_p} \in \mathbb{Z}_q^{n+1}$ for an LWE ciphertext encrypting an integer v with plaintext space $\mathbb{Z}_p$. An LWE ciphertext has the form

$$\mathsf{LWE}_{v,\mathbb{Z}_p} = (\mathbf{a}, b) = (\mathbf{a}, \mathbf{a} \cdot \mathbf{s} + v + e), \tag{4}$$

  where $\mathbf{a}$ and $\mathbf{s}$ are the same as in an RLWE ciphertext, $\cdot$ is the inner product operator, $v \in \mathbb{Z}_p$ is the plaintext integer, and $e \in \mathbb{Z}_q$ is an error integer.
- **RGSW**: We use RGSW_{s_i} to specify an RGSW ciphertext encrypting one integer s_i. We use $\mathbf{s}$ to denote the plaintext vector of RGSW as it is only used to encrypt secret keys $s_i \in \mathbf{s}$ for $i \in |\mathbf{s}|$. Let G^{-1} be the decomposition function as used in [16], and $\mathbb{Z}_p^{\ell n} \ni \mathfrak{s}_\mathfrak{i} = G^{-1}(s_i)$ where ℓ is some small integer (*e.g.*, three or four), an RGSW is composed of the following terms:

$$\mathsf{RGSW}_{s_i} = (\mathfrak{a}, \mathbf{b}) = (\mathbf{a} + \mathfrak{s}_\mathfrak{i} || \mathbf{0}, \mathbf{a} \boxdot \mathbf{r} + \mathbf{e} + \mathbf{0} || \mathfrak{s}_\mathfrak{i}) \tag{5}$$

  where $\mathbf{a} \in \mathbb{Z}_q^{2\ell n}$ is a uniformly random matrix, $\mathbf{r} \in \mathbb{Z}_q^{\ell n}$ is some secret vector, $||$ is the concatenation operator, $\boxdot$ is a constant multiplication operator acting over the module of polynomials (*e.g.*, when $a = [\mathbf{a}_0\ \mathbf{a}_1]$, $a \boxdot \mathbf{r} = [\mathbf{a}_0 * \mathbf{r}\ \mathbf{a}_1 * \mathbf{r}]$ is still a vector of dimension $2\ell n$, and $*$ is the nega-cyclic convolution operator), $\mathfrak{s}_\mathfrak{i} \in \mathbb{Z}_p^{\ell n}$ is the decomposed plaintext vector, $\mathbf{0} \in \mathbb{Z}_q^{\ell n}$ is a vector of zeroes, and $\mathbf{e} \in \mathbb{Z}_q^{2\ell n}$ is the error vector. Visually, C_{s_i} can be represented by the following equation.

$$\mathsf{RGSW}_{\mathbf{s}} = \begin{pmatrix} \mathbf{a}_0 + \mathfrak{s}_\mathfrak{i} & \mathbf{a}_0 \boxdot \mathbf{r} + \mathbf{e}_0 \\ \mathbf{a}_1 & \mathbf{a}_1 \boxdot \mathbf{r} + \mathfrak{s}_\mathfrak{i} + \mathbf{e}_1 \end{pmatrix} \tag{6}$$

  where $\mathbf{a} = \mathbf{a}_0 || \mathbf{a}_1$ and $\mathbf{e} = \mathbf{e}_0 || \mathbf{e}_1$.

Algorithm 1 Homomorphic Gate

Require: c_a, c_b: the input LWE ciphertexts, **IKSK**: the identity key switching key, **BK**: the bootstrapping key.
Require: *scale*, *offset*: parameters to realize specific homomorphic gates.
Ensure: c_y: the output LWE ciphertext.
1: $c_{\text{add}} = scale \cdot (c_a + c_b) + (\mathbf{0}, offset)$
2: $c_{\text{rot}} = \mathsf{BlindRotate}(c_{\text{add}}, \mathbf{BK}, \mathsf{RLWE}_{1/8,\mathbb{Z}_p})$
3: $c_y = \mathsf{SampleExtract}(c_{\text{rot}}, 0)$

All ciphertexts in LWE cryptosystems contain some level of errors in the ciphertext for security reasons. When homomorphic operations are applied over such ciphertexts, the sizes of errors will be amplified, until a point where the ciphertexts become no longer decryptable. We use $\mathcal{E}(\cdot)$ to denote the size of the error contained in a ciphertext. For example, for the RLWE ciphertext $[\mathbf{v}]$ in Eq. (1), $\mathcal{E}([\mathbf{v}]) = ||\mathbf{e}||_2$ is the L_2 norm of the error vector $\mathbf{e}$ in $[\mathbf{v}]$.

3.3 Unbounded Circuit Evaluation over FHE

The two mainstream modes of evaluating circuits over HE are LHE and FHE. In LHE, circuits are with a prescribed level of depth, depending on the complexity of the computation (*e.g.*, the complexity of the SQL, number of filtering predicates, *etc.*). Any further circuit evaluation exceeding the prescribed depth will result in incorrect results. Meanwhile, FHE permits unbounded-depth circuit evaluation, and is more suitable for allowing flexible query statements.

Unfortunately, errors (*e.g.*, e, $\mathbf{e}$) exist in all of the ciphertext types we use in this work, and homomorphic evaluations over complex SQL queries increase the amount of errors in the ciphertext. Consequently, FHE requires frequent invocations of the bootstrapping procedure to reduce the amount of errors accumulated in the ciphertexts, and bootstrapping is known to be comparatively slow in the BGV and CKKS schemes [9, 15] (in the order of minutes [20]). Subsequently, FHEW [20] and TFHE [16] are proposed to dramatically reduce the latency of bootstrapping at the cost of reduced functionality. In short, FHE schemes based on TFHE permit an unbounded number of evaluations of only simple logic gates (*e.g.*, NAND gate). In particular, Alg. 1, Alg. 2 and Alg. 3 are the three key operations in evaluating a homomorphic gate.

Here, Alg. 1 is the overall procedure that can evaluate arbitrary homomorphic gates over encrypted inputs. Within Alg. 1, the core operation is Alg. 2, *i.e.*, BlindRotate, which is also referred to as the bootstrapping operation. Here, BlindRotate is applied to some input LWE ciphertext for two main purposes: to extract the result of the function evaluation (*e.g.*, the homomorphic logic gate function in

Algorithm 2 Blind Rotate/Bootstrapping

Require: c_{in}: the input LWE $= (\mathbf{a}_{\text{in}}, b_{\text{in}})$ ciphertext, **BK**: the bootstrapping key, TV: the input RLWE ciphertext.
Ensure: c_{rot}: the output RLWE ciphertext encrypting $x^{\lfloor (2n/q)(b-\mathbf{a}\cdot\mathbf{s})\rfloor} \cdot TV$

1: Let $\rho = 2n/q$
2: $c_{\text{rot}} = x^{-\lfloor \rho \cdot b_{\text{in}} \rfloor} \cdot (0, TV)$
3: **for** $i = 0$ to $n-1$ **do**
4: $\quad c_{\text{rot}} = \text{Decomposition}\big((x^{\lceil \rho \cdot \mathbf{a}_{\text{in}} \rfloor} \cdot c_{\text{rot}}) - c_{\text{rot}}\big) \cdot BK_i + c_{\text{rot}}$

Alg. 1), and to refresh the error in the input LWE ciphertext. As illustrated on Line 4 in Alg. 2, the essential component of BlindRotate is Decomposition, which decomposes an input RLWE ciphertext in $\mathbb{Z}_q^n$ into another RLWE ciphertext with smaller elements in $\mathbb{Z}_r^{2\ell n}$, where $r < q$. Alg. 3 is also known as the radix decomposition, as the algorithm decomposes every element of the ciphertext c_{in} into ℓ many r-radix integers. Unfortunately, as discussed in Sec. 5.4, the bootstrapping and decomposition procedures can only be applied to small-parameter LWE ciphertext, *i.e.*, $n \approx 1024$ and $\log_2 q \leq 64$. When the size of ciphertexts becomes larger (larger than 64-bit), the residual number system (RNS) technique needs to be applied, where a large integer a is decomposed into a set of smaller integers $(a_0 \bmod q_0, a_1 \bmod q_1, \cdots, a_l \bmod q_l)$.

4 SYSTEM OVERVIEW

This section first provides an intuitive example to illustrate how FHE could be used for SQL aggregation queries (Sec. 4.1), followed by describing the overall *HEDA* workflow (Sec. 4.2). This section further explains how different FHE primitives are used to implement SQL operators to give maximized performance (Sec. 4.3).

4.1 An Intuitive Example

Consider the query as in Listing 1 over the company employee table `emp` as shown in Table 2. The table contains five columns: employee ID, monthly salary, join date, affiliated department and vaccination status. We consider all but the ID column as sensitive fields and need to be encrypted for illustration purposes. The query calculates the total salary for employees who earn between `5000` to `6000` monthly and joined the company before 1st May 2022.

```
SELECT SUM(salary) FROM emp
WHERE salary BETWEEN 5000 AND 6000
      AND joindate < date '2021-05-01' + interval 1 year;
```

Listing 1: Salary SUM Query (Query S)

Table 2: Employee Table emp

ID	salary	joindate	dept	vacstatus
249111	5693.5	2022-04-01	IT	TRUE
250123	6354.7	2022-04-14	Sales	TRUE
287635	5120.6	2022-05-02	Admin	FALSE
234754	5800.5	2022-02-17	IT	TRUE

The SQL in Query S contains three comparisons and one aggregation. Two comparisons come from the `BETWEEN` operator, *i.e.*, the $\geq$ and $\leq$ arithmetic operator, and one comparison from the `joindate` predicate (we ignore the + for the `date` condition, to be

Algorithm 3 Decomposition

Require: $\mathbf{c}_{\text{in}}$: the input RLWE ciphertext in $\mathbb{Z}^{2\times n}$
Ensure: $\mathbf{c}_{\text{dec}}$: the output decomposed ciphertext in $\mathbb{Z}_q^{2\times 2\ell n}$

1: Let $offset_k = [\frac{1}{2r^k} \cdots \frac{1}{2r^k}] \in \mathbb{Z}^n$
2: Let $offset_{\text{acc}} = offset_\ell$
3: **for** $i = 0$ to $\ell - 1$ **do**
4: $\quad offset_{\text{acc}} = offset + offset_i$
5: Let $\mathbf{c}_{\text{ot}} = (\mathbf{a}_{\text{ot}}, \mathbf{b}_{\text{ot}}) = \mathbf{c}_{\text{in}}$
6: $\mathbf{c}_{\text{ot}} = \mathbf{c}_{\text{ot}} + (offset_{\text{acc}}, offset_{\text{acc}})$
7: **for** $i = 0$ to $\ell - 1$ **do**
8: $\quad \mathbf{c}_{\text{dec},i} = (\lceil r^i \cdot \mathbf{a}_{\text{ot}} \rfloor \bmod r) - \frac{r}{2} \cdot 1[x]$
9: $\quad \mathbf{c}_{\text{dec},i+\ell} = (\lceil r^i \cdot \mathbf{b}_{\text{ot}} \rfloor \bmod r) - \frac{r}{2} \cdot 1[x]$

explained in Sec. 4.2). The aggregation operator `SUM` is basically a finite number of additions. FHE could support these operations natively. The challenge is that, the FHE comparison produces *encrypted* results (*i.e.*, encrypted 0/1 bits), which the server could not decrypt, and thus could not further filter the non-satisfying rows as in the traditional database. However, the subtle point is, if these encrypted filtering results are multiplied by the corresponding values in the aggregation attribute, then the attribute value of those non-satisfying rows will become zero and *implicitly* be filtered out during the final aggregation.

Fig. 2 illustrates this process. We use [] to denote FHE encryption, $\otimes$ and $\oplus$ for FHE multiplication and addition, respectively. In the first step, the system executes independently over each filtering predicate and obtains encrypted 0/1 bits, followed by the FHE AND operation for all three *encrypted* comparison result column vectors (FHE AND is used as the SQL itself is a conjunctive query; OR could be handled similarly if exists). In the second step, by multiplying the combined encrypted filtering results with the aggregation attribute, it implicitly filters the non-satisfying row 2 and 3, and this won't affect the final aggregation results during the FHE addition process.

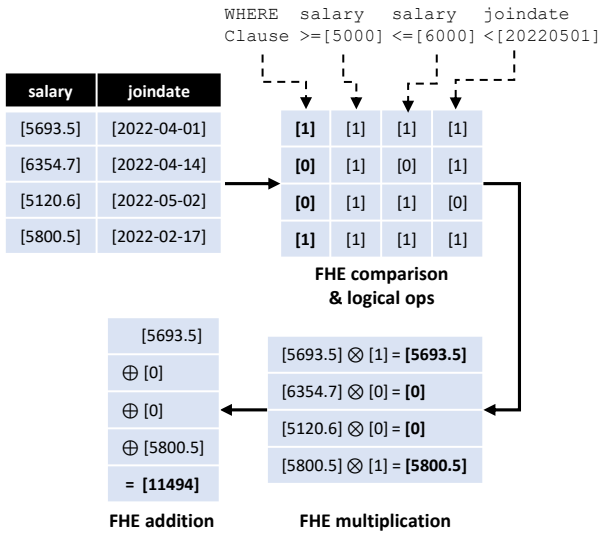


Figure 2: Illustration of the implicit filtering process.

4.2 *HEDA* Workflow

HEDA assumes a client-server model, and Fig. 3 illustrates the overall workflow. The client side is equipped with a client SDK, to handle **Data Preparation**, **Query Preparation**, and **Result Processing** phases. The server side, usually the CSP, stores the encrypted client data, as well as processes the client query through the *HEDA* Engine, and return the encrypted result to the client. *HEDA* does not require the user to pre- or post-process any of the input/output data and the query itself (as shown on the left most side of Fig. 3, the input, output and query are as per normal database usage), but rather handled by the client SDK entirely.

4.2.1 Data Preparation. The client SDK first needs to pre-process the user data, including scaling up the attribute values that were originally in float/double format to integer to meet the FHE plaintext space requirement. It also needs to convert other data types such as date or categorical values into integer format* (❶). For categorical values, the client side needs to maintain a string-to-integer mapping for later query issuance. Specifically, for categorical values with a binary domain, they are encoded into either 0 or 1. The client SDK then uses different FHE encryption schemes to encrypt integer and binary values (explained in Sec. 4.3) as in Step ❷, and uploads the encrypted values to CSP (❸).

4.2.2 Query Preparation. To issue an aggregation query, the client SDK needs to, as shown in Step ❹:

- value scale up and data type change;
- convert SQL operator into arithmetic operators (*e.g.*, `BETWEEN` to $\geq$ and $\leq$);
- pre-compute the trivial operations

followed by encrypting the sensitive fields with respective FHE encryption schemes according to the converted data type (❺).

4.2.3 Query Processing. By receiving the user query, the *HEDA* engine, at a high level, performs the steps described in the previous Sec. 4.1, and outputs the encrypted aggregation query result (❻). However, cryptographic design and optimization need careful treatment. We defer the detailed explanation to Sec. 5.

4.2.4 Result Processing. With the received encrypted aggregation results, the client SDK decrypts using the secret key to obtain the *scaled* plaintext result (❼), followed by dividing the scaling factor and return the true result (❽).

4.3 SQL Operator Support

4.3.1 Basic Operators. Through the above example, to complete the aggregation task, three high-level steps are needed: **comparison**, **multiplication** (to implicitly filter the rows), and **addition**. The latter two steps are relatively straightforward, FHE schemes such as BFV would be sufficient and efficient. For the comparison step, it would be the most computationally intensive step and deserve a closer look. By choosing different FHE schemes, the comparison step would have a vast efficiency difference.

There are two perspectives to consider for comparison: the *data type* and *comparison operator*. We categorize them into **binary equality**, **numerical inequality**, and **numerical equality**. Note that binary inequality is meaningless and do not discuss it further.
binary equality: the binary comparison could be a common filtering predicate in an aggregation query (*e.g.*, `read_status = 1/0` or `vaccinated_status = TRUE/FALSE`). FHE schemes such as BFV that are typically tailored for numerical value encryption could still handle binary values, but would be much less efficient as it requires a very large and deep comparison circuit. Instead, *HEDA* uses TFHE, an FHE scheme specially designed to process binary operations, to encrypt and compare binary values.
numerical inequality: the BFV scheme is used to encrypt and compare numerical values. To compare, *e.g.*, if $a \geq b$, the system computes $a - b$ and *homomorphically* extracts the most significant bit (MSB) from the subtraction result: an encrypted 0 indicates true and 1 otherwise (*i.e.*, $a < b$). For comparison operator $>$ (*e.g.*, if $a > b$), *HEDA* inverts the operands and compare instead if $b < a$ (*i.e.*, extract the MSB of $b - a$), and the same applies to the $\leq$ operator.
numerical equality: with the FHE support to numerical inequality operator, to check if $a = b$, it is equivalent to check if $a \geq b$ `AND` $a \leq b$, *i.e.*, to extract the MSB from both $\geq$ and $\leq$ comparison results, and feed the two MSBs into a homomorphic `AND` operation.

We further explain in detail how the above comparison operators are achieved using FHE schemes in Sec. 5.3.2.

4.3.2 `COUNT` *for packed encryption.* Dummy rows need to be introduced to form an RLWE ciphertext to fit the FHE parameter. Unlike traditional ways of padding a unique string/value that is out of the attribute domain, the FHE plaintext domain only permits integer or binary values. One way is to pad all zero-value rows, however, this padding approach may cause an inaccurate result. Take the same Table 2 example with a query that counts the number of employees who join before a specific date. As the dummy `joindate` are padded with 0 and thus satisfy the condition trivially, this will cause an inaccurate `COUNT` result. Note that, however, this would not affect queries with `SUM` operator: although some of the dummy rows may be wrongly included (*i.e.*, with encrypted 1 as filtering result), when multiplying with the to be aggregated dummy value, *e.g.*, salary (zero-valued padded), the result would be an encrypted 0 and thus not affecting the total sum.

To address this issue, during initial encryption each row is associated with an encrypted 0/1 tag: 0 for dummy and 1 for real data rows. When the execution of filtering predicates completes with the aggregated filtering result, the protocol requires an additional homomorphic AND operation between the filtering result and the encrypted row tag. This would implicitly remove those dummy rows, and the `COUNT` is calculated (*i.e.*, FHE addition) over this further filtered result.

4.3.3 `GROUP BY`. A straightforward way is to segregate the original query into several SQLs, each with additional equality comparison predicates. For example, in TPC-H Query 1, given `returnflag = {A,N,R}` and `linestatus = {O,F}`, the original SQL could be segregated into six SQLs without `GROUP BY`, each with two additional predicates, such as `returnflag = A AND linestatus = O`. However, this extension results in exponential complexity w.r.t the number of `GROUP BY` attributes and also their domain sizes. It quickly becomes impractical if the domain size is large. For `GROUP BY` condition with computation involved, *e.g.*, `GROUP BY`

*For date/time data type, UNIX time represents the data as a 32-bit integer, and easily supports various computations. For illustration, we use the YYYYMMDD format here.

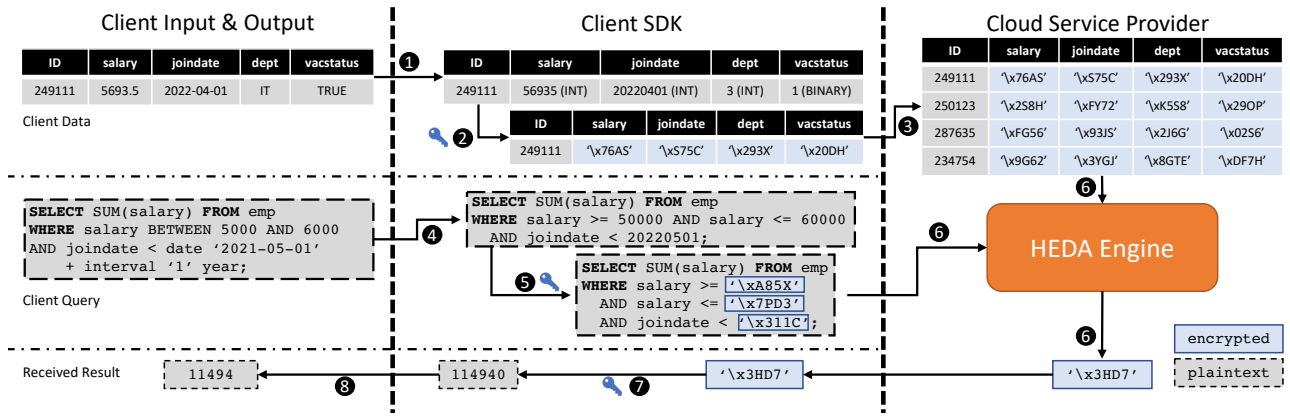


Figure 3: Illustration of the *HEDA* system flow.

`(1+taxrate)*orderprice`, theoretically, the server side could compute the condition blindly, followed by the same approach as above to process the query, but inherently this will incur extremely large overhead. To efficiently support unbounded `GROUP BY` condition using FHE is an important open question.

4.3.4 Other operators. Apart from the above-mentioned operators, other SQL operators such as `IN`, `BETWEEN`, `EXIST`, `ANY`, `ALL` are in principle supported by FHE, as they could be translated into comparison operation. `LIKE` and wildcards are difficult to support straightforwardly, as it focuses on string matching and may require complex string-to-integer encoding. For basic arithmetic operators, division and modulus are not or difficult to be supported as this is still an active research area in FHE, while bit-wise operators are supported straightforwardly.

5 CRYPTOGRAPHIC OPTIMIZATIONS

5.1 Formalizing the Threat Model

In this work, we adopt the classic threat model for generating secure queries over an outsourced database using FHE [31]. Here, we assume that client C encrypts and outsources the database $\mathcal{D}$ to the server $\mathcal{S}$. Since $\mathcal{S}$ does not possess any private information, the only possible attack occurs when $\mathcal{S}$ tries to learn as much information as possible from the data outsourced from C. Following typical privacy-preserving protocols over FHE, we assume that $\mathcal{S}$ is a semi-honest adversary. In other words, while $\mathcal{S}$ tries to learn as much as possible from the data provided by C, $\mathcal{S}$ honestly follows the prescribed protocol as illustrated in Sec. 5.2. In what follows, we outline the public and private data specified in our protocol and explain the reasons for such assumptions. We use $\mathcal{D}$ to denote the database, Q the query generated by the client and Attr as the attribute (*i.e.*, column labels) in $\mathcal{D}$.

Public Data: *HEDA* assumes the following database information to be public, *i.e.*, known to the server and third-party adversaries:

- $|\mathcal{D}|_r$: Size of the database, *i.e.*, the number of rows in $\mathcal{D}$. Most secure outsourcing of databases does not hide the size of the database [31, 51].
- $[\mathcal{D}]_c$: Number of attributes, *i.e.*, the number of columns in the database (the database schema is public information).
- $|Q|$: The number of filtering predicates in Q.
- NumType(Attr): The numerical type of an attribute, *e.g.*, a binary attribute or a numerical attribute.

Private Data: Under an outsourced DBMS setting, *HEDA* assumes the following properties of the database to be private, *i.e.*, kept secret to the server and any third-party adversaries:

- $\mathcal{D}_{row,col}$: Database entry value, $row \in |\mathcal{D}|$ and $col \in [\mathcal{D}]$.
- Q_i: The i-th filtering predicate value of Q for all $i \in |Q|$.

We also use Attr_j to refer the j-th column vector in $\mathcal{D}$, *i.e.*,

$$\text{Attr}_j = [\mathcal{D}_{0,j}\ \mathcal{D}_{1,j} \cdots \mathcal{D}_{|\mathcal{D}_r|,j}]^T \tag{7}$$

Adversary Goal and Protocol Security: As mentioned above, the adversary $\mathcal{S}$ tries to learn any of the following private data: $\mathcal{D}_{row,col}$ and Q_i, using all information available, such as the public data (*e.g.*, $|\mathcal{D}|$, $[\mathcal{D}]$) and ciphertext information (*e.g.*, the encrypted ciphertexts of $\mathcal{D}_{row,col}$ and Q_i). As explained in Sec. 4.2, *HEDA* follows a standard secure function outsourcing setup, where all predicate values in Q and items in $\mathcal{D}$ are encrypted under the FHE scheme described in Sec. 3. Therefore, the security of our protocol could be directly reduced to the security of the FHE ciphertexts. For an honest-but-curious server, the chosen-plaintext attack (CPA) security of the FHE schemes could safely guarantee that the query and database items could only be decrypted using the client's secret keys, and remain encrypted elsewhere. The concrete encryption parameters could be found in Sec. 6.1.1.

5.2 Protocol Overview

Given the encrypted database and query, the server operates on the ciphertexts following the steps shown in Fig. 4.

First, the encrypted filtering predicate values from the query are compared with the corresponding encrypted values from the database with the same attributes. This comparison could be binary equality (over Z_2) or numerical equality/inequality (over Z_t), according to the plaintext type and the comparison operator specified by the query (elaborated in Sec. 5.3.2). The parameter t is usually small as the plaintext space of the comparison attribute is typically small (*e.g.*, 32 or 64 bits). To multiply this encrypted comparison result with the aggregation attribute which is defined over Z_p ($p \gg t$), *HEDA* needs to *lift* the plaintext space from Z_2 (or Z_t) to Z_p. We name this novel process **parameter-lifting bootstrapping** PLB (detailed in Sec. 5.4).

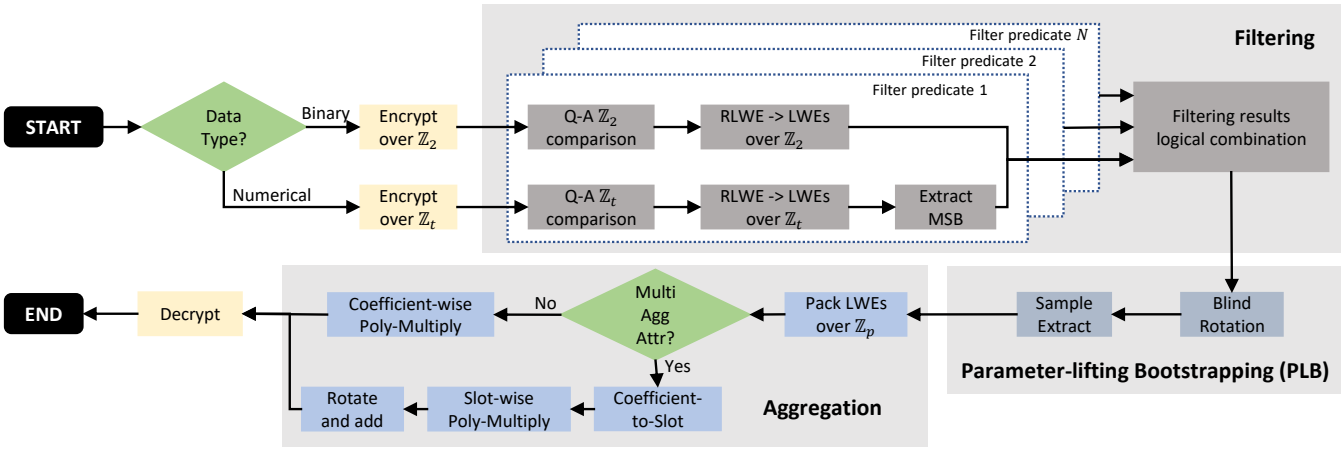


Figure 4: Illustration of the involved *HEDA* cryptographic steps.

However, before the bootstrapping, the packed comparison results, appearing as an RLWE ciphertext, need to be unpacked into LWE ciphertexts because bootstrapping can only be applied to individual coefficients (each coefficient corresponds to an LWE ciphertext) rather than the whole polynomial. This step is referred to as RLWE-to-LWEs conversion. Correspondingly, after the bootstrapping, the LWE ciphertexts are repacked into an RLWE ciphertext for the subsequent aggregation.

The parameter-lifting bootstrapping employs similar algorithms in TFHE, *i.e.*, blind rotation and sample extraction [16]. Blind rotation refers to a process in which the coefficients of polynomial P_A are rotated by the length of $[l]$, where $[l]$ is in encrypted form. Sample extraction refers to extracting the constant term of the resulting polynomial. Note that, after blind rotation and sample extraction, $[l]$ (over Z_2 or Z_t) is converted into the l-th coefficient of P_A (over Z_p), meaning that the filtering results are adapted to the domain where aggregation will be conducted.

In the final step, the filtered rows are aggregated (elaborated in Sec. 5.3.4). If the aggregation involves only one attribute (*e.g.*, `SUM(quantity)` as in TPC-H Query 1), then it could be achieved by multiplying the comparison results (within an RLWE ciphertext) with the encrypted aggregation column (equivalent to dot-product). However, if the aggregation involves more than one attribute (*e.g.*, `SUM(extendedprice*(1-discount))`), *HEDA* needs to convert the ciphertext from coefficient-wise to slot-wise using a linear transformation (*i.e.*, Coefficient-to-Slot) followed by slot-wise multiplications and additions. More detail are presented in Sec. 5.3.4.

5.3 Packed Filtering and Ciphertext Conversion

The main reason that lattice-based FHE schemes are preferred is their SIMD capability which could carry out the same computation over a large number (up to the dimension of the ciphertext, n) of plaintext slots using only one ciphertext. The SIMD method works well for homomorphic linear functions (*i.e.*, addition, multiplication, and aggregation) but not for non-linear ones (*e.g.*, numerical comparison* and filtering). In the most recent work [41], linear tasks and non-linear tasks are divided into two groups. The SIMD technique is only applied to linear layers, while non-linear functions are evaluated in a point-wise manner. In this work, *HEDA* is based on a similar approach, adopting the three types of ciphertexts as mentioned in Sec. 3, namely, LWE, RLWE, and RGSW.

Our main observation is that, filtering operations in database systems are mostly composed of simple comparisons that produce binary results. Therefore, the core idea of *HEDA* lies in leveraging conversions between RLWE and LWE ciphertexts to efficiently utilize the SIMD capability provided by the RLWE ciphertext, while retaining the ability of applying non-linear operations over the encrypted ciphertexts. With these techniques, *HEDA* is able to support both filtering and aggregation using a unified encryption method instead of using multiple ones. In particular, as mentioned in Sec. 5.2, both attribute comparisons and the final aggregations are carried out over RLWE ciphertext encrypting a pack of plaintext values, such that queries over a large database could be evaluated efficiently. More specifically, the cryptographic filter-aggregation procedure consists of the following four steps: query encryption, packed filtering, ciphertext conversion, and aggregation. In what follows, we explain each step in detail.

5.3.1 Query Encryption. Due to the underlying mathematical structure, a single filtering predicate value needs to be copied n times for the generation of one encrypted query to ensure the efficient SIMD computations. Concretely, when encrypting a query Q with $|Q|$ predicates, a total of $|Q|$ RLWE ciphertexts are generated, where the ciphertext for each of the i-th predicate is specified as

$$[Q_i] = (\mathbf{a}_i, \mathbf{b}_{Q_i}) = (\mathbf{a}_i, \mathbf{a}_i * \mathbf{s} + \mathrm{Copy}_n(Q_i) + \mathbf{e}), \tag{8}$$

where $\mathrm{Copy}_n(Q_i) = [Q_i\ Q_i \cdots Q_i]$ is a vector in $\mathbb{Z}_q^n$ constructed by duplicating the values of Q_i n times.

5.3.2 Packed Filtering. Filtering aims to find the rows that meet the predicates specified by the query generated in Sec. 5.3.1. We expect the filtering, as well as the aggregation, to be conducted in a SIMD manner because it is more efficient. In particular, the filtering starts with a SIMD subtraction over an RLWE ciphertext. However, carrying out the non-linear operation part in a numerical comparison or combining multiple filtering results cannot be

*Note that a binary comparison is regarded as a linear function because it can be realized using a homomorphic addition.

completed over RLWE ciphertexts. Thus, we need to convert the RLWE ciphertext into a number of LWE ciphertexts and carry out the PLB for individual LWE ciphertexts. After the PLB, the LWE ciphertexts are packed again for a SIMD-manner aggregation.

- **Binary equality**: For binary attributes, an equality comparison, equivalent to an XNOR, is implemented as an addition followed by a negation. Obviously, this step involves no non-linear functions, and therefore runs over RLWE ciphertexts.
- **Numerical inequality**: For numerical predicate values in the query (represented by an RLWE ciphertext $\mathbf{c}^{(q)}$) and database attribute values (represented by an RLWE ciphertext $\mathbf{c}^{(a)}$) such as `AGE > 30`, their inequality could be determined by subtracting the $\mathbf{c}^{(a)}$ and $\mathbf{c}^{(q)}$ values, and then extract the MSB of the resulting RLWE ciphertext (as detailed in Table 3).

Table 3: Inequality Comparison for Numerically-encoded Query Value $\mathbf{c}^{(q)}$ and Attribute Values $\mathbf{c}^{(a)}$.

Index	SIMD operation	MSB	comparison result
1	$\mathbf{c}^{(q)} - \mathbf{c}^{(a)}$	0	$\mathbf{c}^{(q)} \geq \mathbf{c}^{(a)}$
2	$\mathbf{c}^{(q)} - \mathbf{c}^{(a)}$	1	$\mathbf{c}^{(q)} < \mathbf{c}^{(a)}$
3	$\mathbf{c}^{(a)} - \mathbf{c}^{(q)}$	0	$\mathbf{c}^{(q)} \leq \mathbf{c}^{(a)}$
4	$\mathbf{c}^{(a)} - \mathbf{c}^{(q)}$	1	$\mathbf{c}^{(q)} > \mathbf{c}^{(a)}$

- **Numerical equality**: Numerical equality could be obtained by AND'ing the inequality comparison 1 and 3 in Table 3. In other words, $\mathbf{c}^{(q)} = \mathbf{c}^{(a)}$ is equivalent to $\mathbf{c}^{(q)} \geq \mathbf{c}^{(a)}$ AND $\mathbf{c}^{(q)} \leq \mathbf{c}^{(a)}$. This AND operation, due to its non-linear nature, could not execute over RLWE ciphertext. Instead, the RLWE ciphertext is unpacked into LWE ciphertexts which are then processed as a Boolean circuit over TFHE.
- **Multiple comparisons**: If the query contains multiple filtering predicates, then all comparison results need to be AND'ed. Similar as numerical equality, the AND operations are processed as a Boolean circuit over TFHE.

5.3.3 Ciphertext Conversion. After the homomorphic comparison, the results are packed into one RLWE ciphertext $[Q_i - \mathsf{Attr}_j]_{\mathbb{Z}_t}$*($Q_i$ and Attr_j correspond to the same column), which is basically the homomorphic subtraction between the query predicate value Q_i and the corresponding filtering attribute value Attr_j. However, such comparison results could not be directly fed into the aggregation process for two main issues. First, $[Q_i - \mathsf{Attr}_j]_{\mathbb{Z}_t}$ works over a plaintext modulus of $\mathbb{Z}_t$ for some t where $2 \leq t < p$, which are different from the plaintext space $\mathbb{Z}_p$ of the encrypted aggregation attribute, $[\mathsf{Attr}_{agg}]_{\mathbb{Z}_p}$. Hence, we need the ciphertext conversion operation to convert the ciphertext from $[Q_i - \mathsf{Attr}_j]_{\mathbb{Z}_t}$ to $[Q_i - \mathsf{Attr}_j]_{\mathbb{Z}_p}$, without increasing the amount of errors in the ciphertext. The second issue is that, while binary comparisons over $\mathbb{Z}_2$ directly produces binary results, comparisons over $\mathbb{Z}_t$ needs an additional MSB extraction function to actually produce this encrypted binary value. To achieve both goals, we devised the parameter-lifting bootstrapping technique PLB, where

$$\mathsf{LWE}_{f(v),\mathbb{Z}_p} = \mathsf{PLB}(\mathsf{LWE}_{v,\mathbb{Z}_t}, p) \tag{9}$$

The cryptographic details of PLB will be elaborated in Sec. 5.4.

*Here for notation simplicity, we assume $2 \leq t < p$, *i.e.*, this notation includes both the binary $t = 2$ and numerical $2 < t < p$ case.

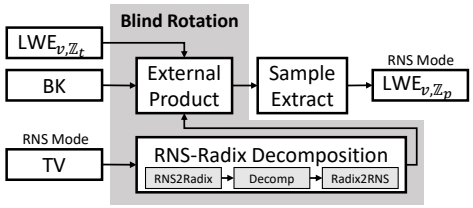


Figure 5: A conceptual illustration of the PLB functionality.

5.3.4 Aggregation. Given the lifted comparison results ($\mathbf{c}^{(r)}$) and the aggregation attribute ($\mathbf{c}^{(g)}$), aggregation could be achieved by a simple multiplication between them. Let $\mathbf{p}^{(r)}$ and $\mathbf{p}^{(g)}$ be the plaintexts (Eq. (10)) that correspond to ciphertexts $\mathbf{c}^{(r)}$ and $\mathbf{c}^{(g)}$. Multiplying $\mathbf{p}^{(r)}$ by $\mathbf{p}^{(g)}$ is actually a convolution with the dot-product result located at the coefficient of the term x^{n-1} (Eq. (11)):

$$\mathbf{p}^{(r)} = \sum_{i=0}^{n-1} a_i x^i \quad \text{and} \quad \mathbf{p}^{(g)} = \sum_{j=0}^{n-1} b_{n-1-j} x^j \tag{10}$$

$$\mathbf{p}^{(r)} \times \mathbf{p}^{(g)} = \sum_{i=0}^{n-1} c_i x^i, \text{ where } c_{n-1} = \sum_{i=0}^{n-1} a_i b_i \tag{11}$$

As mentioned in Sec. 5.2, the dot-product described in Eq. (11) only applies to the queries that only one attribute is involved in aggregation. For aggregations that involve more than one attribute, we need to re-encode the RLWE ciphertext as slot-wise format (referred to as Coefficient-to-Slot in Fig. 4). In a slot-wise encoded ciphertext, values are located in "slots" instead of coefficients, which enables slot-wise multiplication, and then followed by summing up the slots (achieved by the rotate-and-add in Fig. 4) [35].

5.4 Parameter-Lifting Bootstrapping

The key component in achieving the homomorphic evaluation described in Sec. 5.2 is the operator that we refer to as parameter-lifting bootstrapping. PLB converts a ciphertext with small plaintext space into a larger one without increasing the level of noises. In other words, we need to *lift* the filtering results that is encrypted either in $\mathbb{Z}_2$ or $\mathbb{Z}_t$ to $\mathbb{Z}_p$ for aggregation computation that is executed over $\mathbb{Z}_p$. PLB was not known possible (to the best of our knowledge) in current literature related FHE, including TFHE [16], BFV [23] and CKKS [15] along with their open-source implementations such as SEAL [57], PALISADE [47], HElib [33], Concrete [18], among others. Therefore, we devise the theory and the concrete implementation of PLB to make the overall protocol flow cryptographically viable.

Fig. 5 provides an illustration of the functionality of Eq. (9). Here, the main objective of PLB is to convert an LWE ciphertext with a plaintext v in the plaintext space $\mathbb{Z}_t$ to another LWE ciphertext encrypting the plaintext $f(v)$ over some plaintext space $\mathbb{Z}_p$, for some function f that depends on the bootstrapping technique used. The "lifting" property of PLB comes from the fact that $p \geq t$ and $\mathcal{E}(\mathsf{LWE}_{f(v),\mathbb{Z}_p}) = O(1)$. In other words, the input ciphertext is converted into the output ciphertext that attains a larger plaintext space with a constant amount of errors, permitting a number of further homomorphic computations to be carried out efficiently.

To achieve efficient PLB, the main challenge is to carry out radix decomposition under RNS-represented RLWE ciphertext. Essentially, as shown on Line 4 in Alg. 2, the original bootstrapping technique in TFHE [16] adopts radix decomposition to suppress the large noise amplification during the process of the homomorphic multiplication between the RLWE ciphertext c_{rot} and the i-th bootstrapping key (*i.e.*, RGSW ciphertext). When the ciphertext modulus for $\mathbf{c}_{\text{rot}}$ is small, *e.g.*, less than 64-bit, radix decomposition could be directly applied to each element in $\mathbf{c}_{\text{rot}}$. Nonetheless, when q becomes large, it needs to be split into l different RNS moduli $(q_0, q_1, \cdots, q_{l-1})$ to ensure that all q_i satisfy that $\log_2 q_i \leq 64$. RNS is generally applied to divide integers in ciphertexts into 64-bit chunks such that they could be efficiently processed by 64-bit processors. Unfortunately, the BlindRotate algorithm could not be directly applied to RNS-represented ciphertexts because we have multiple copies of $\mathbf{c}_{\text{rot}}$, each representing a small piece of $\mathbf{c}_{\text{rot}}$ in a particular RNS modulus. As a result, the product between BK_i and $\mathbf{c}_{\text{rot}}$ on Line 4 needs to be carried out in the RNS domain, which requires BK_i also to be in the RNS domain. The difficulty here is that representing the RGSW ciphertext BK_i in RNS splits the internal error within the RGSW ciphertext, and radix decomposition is not meaningful in suppressing errors in the RNS domain.

Therefore, to ensure that the output ciphertext $\mathbf{c}_{\text{rot}}$ to have a large plaintext space (and thus large ciphertext space) and small error size, we need to switch $\mathbf{c}_{\text{rot}}$ from the RNS domain back to the original radix domain and perform the homomorphic multiplication between $\mathbf{c}_{\text{rot}}$ and BK_i. The naïve approach in tackling this method is simply to restore $\mathbf{c}_{\text{rot}}$ back to the large ciphertext space over $\mathbb{Z}_q$. Note that, here, q could be up to 140-bit, and we point out that handling such large integer over any type of hardware results in low computational efficiency. Hence, to achieve large plaintext and small error growth, we devise the RNS-Radix decomposition technique that directly convert RNS-decomposed ciphertext into radix-decomposed ciphertext, without the need of handling large integers (*i.e.*, larger than 64-bit). Formally, let $(\mathbf{c}_0^{\text{Radix}}, \mathbf{c}_1^{\text{Radix}}, \cdots, \mathbf{c}_{J-1}^{\text{Radix}}) = \text{Radix}(\mathbf{c})$ and $(\mathbf{c}_0^{\text{RNS}}, \mathbf{c}_1^{\text{RNS}}, \cdots, \mathbf{c}_{J-1}^{\text{RNS}}) = \text{RNS}(\mathbf{c})$ be the radix and RNS representations of $\mathbf{c}$, respectively. The proposed RNS2Radix and Radix2RNS could be formulated as follows:

$$(\mathbf{c}_0^{\text{Radix}}, \cdots, \mathbf{c}_{I-1}^{\text{Radix}}) = \text{RNS2Radix}(\mathbf{c}_0^{\text{RNS}}, \cdots, \mathbf{c}_{J-1}^{\text{RNS}}), \tag{12}$$

where

$$\mathbf{c}_i^{\text{Radix}} = \lfloor \frac{\sum_{j=0}^{J-1} \mathbf{c}_j^{\text{RNS}} \cdot q_j \bmod q}{r^i} \rfloor = \lfloor \frac{\sum_{j=0}^{J-1} \mathbf{c}_j^{\text{RNS}} \cdot q_j}{r^i} - u\frac{q}{r^i} \rfloor \tag{13}$$

$$\text{and } u = \lfloor \frac{\sum_{j=0}^{J-1} \mathbf{c}_j^{\text{RNS}}}{q_j} \rfloor. \tag{14}$$

Note that all terms in Eq. (13) are relatively small, and could be easily handled on processors supporting 128-bit integer operation. Next, we point out that $r \ll q_j$ for all $j \in \{0, \cdots, J-1\}$, the summation of $\lceil J/I \rceil$ radix-decomposed ciphertexts still likely to lie within the range $[0, q_j)$. Therefore, Radix2RNS is almost a free operation, and requires much less computation compared to RNS2Radix.

6 EVALUATION

6.1 Experiment Setup

In this work, the setup for the evaluation of *HEDA* involves two parts: the testing SQL queries and associated database relations, and that of the corresponding FHE parameters.

6.1.1 SQLs and Relations. We test five SQLs in our experiments. We first test Query S with synthetic data as a warm-up. The second SQL is a simple funnel analysis query (Query F) from a real social media e-commerce platform. Sample of the corresponding relation, also extracted from real but de-identified dataset, is shown in Table 4. This relation contains a purchase date, and various user activity capturing. The actual relation contains more columns such as user ID, blogger ID *etc.*, we omit them here as they are not used for the experiment and for the de-identification purpose. This query calculates the total customer spending amount for those who have read a particular promoting blog and followed the blogger within a date range. It contains both binary and numerical comparisons.

```
SELECT SUM(amount) FROM customer
WHERE pur_date < date '2021-04-14'
      AND read = 1 AND follow = 1;
```

Listing 2: Funnel Analysis Query (Query F)

Table 4: 'customer' Table Sample for Funnel Analysis Query

pur_date	search	read	comment	follow	share	amount
2021-04-01	1	1	0	1	0	23
2021-04-01	0	1	1	1	0	49
2021-03-29	0	1	0	0	0	35
2021-03-28	1	1	1	1	1	99

The other three queries are from TPC-H benchmark, Query 1, 6 and 19. For Query 1, we remove the `GROUP BY`, `ORDER BY` operator (discussed in Sec. 7) and the `AVG` operator (as it could be computed at the client side with `SUM` and `COUNT` results). For Query 19 we remove the `p_partkey = l_partkey` condition as it involves multi-way join (discussed in Sec. 7). Query 6 is kept intact. We have pre-processed the data to fit FHE encryption.

6.1.2 HE Setups. The parameters of the FHE scheme are critically dependent on the plaintext domain of the operands in the query (but not the query complexity) and the domain of the database values. In particular, the required precision of the data items within $\mathcal{D}$ determines the plaintext space of FHE. The plaintext space then determines the FHE parameters including lattice dimension and ciphertext modulus. As illustrated in Table 5, comparisons can be carried out over either $\mathbb{Z}_2$ or $\mathbb{Z}_t$ where t can be as large as 32-bit integers. After the SIMD comparisons, PLB lifts the plaintext space to $\mathbb{Z}_p$ where $\lceil \log_2 p = 40 \rceil$. For small plaintext spaces, $\log_2 q \approx 32$ is sufficient. Meanwhile, for the larger plaintext space p, the value q needs to be as large as 140-bit. Such q is split into RNS moduli, within which each individual modulus is less than 60-bit.

6.1.3 Implementation. We implement *HEDA* based on the Microsoft Simple Encrypted Arithmetic Library (SEAL) [57], TFHEpp (a high-performance implementation of the TFHE scheme) [42] and OpenPegasus (linear transformation over RLWE ciphertext) [41]. We evaluate the benchmarks on the Intel Xeon Platinum 8163 CPU@2.5GHz.

Table 5: Summary of the Instantiated Parameters

Ciphertext Format	Plaintext Space	Ciphertext Parameters	RNS Moduli
LWE & RLWE	$\mathbb{Z}_2$ or $\mathbb{Z}_{t=2^{32}}$ $\mathbb{Z}_{p=2^{40}}$	$n = 1024$, $\lceil \log_2 q \rceil = 32$ $n = 8192$, $\lceil \log_2 q \rceil = 140$	N/A {60, 40, 40}
RGSW	$\mathbb{Z}_3$	$n = 8192$, $\lceil \log_2 q \rceil = 140$	{60, 40, 40}

Table 6: Performance of micro-benchmarks based on the Intel Xeon Platinum 8163 CPU@2.5GHz.

Function	Micro-benchmark	Latency (ms)
Filtering	comparison over $\mathbb{Z}_2/\mathbb{Z}_t$	1.5
	RLWE-to-LWEs (8192)	404
	gate bootstrapping	19
PLB	blind rotation of a single LWE	9,197
	sample extraction	0.048
Aggregation	packLWEs	30,574
	polynomial multiplication	38
	coefficient-to-slot	1,803
	slot-rotation	4.4
Others	key generation	160
	encryption	18
	decryption	6

6.2 Micro-benchmarks

Before evaluating the database queries, we first benchmark the latency of each cryptographic step in *HEDA* (called micro-benchmarks) with results shown in Table 6. We learn from Table 6 that packLWEs is the most time-consuming function (> 20 seconds). This is as expected because the function aims to pack 8192 LWEs into an RLWE, executing a total number of 8191 automorphism operations [13]. Since an automorphism operation involves 24 many 8192-point number theoretic transform (NTT) operations*, the function of packLWEs involves more than $196k$ many 8192-point NTTs.

Blind rotation of a single LWE is the second most time-consuming operation (> 9 seconds). Even though it seems much slower than the the TFHE scheme where blind rotation is originally proposed, this latency is within expectation because we employ a much larger polynomial degree (*i.e.*, $n = 8192$ vs. $n = 1024$) which means not only more computation overhead for rotating a single ciphertext but also a larger number of polynomials need to be processed (*i.e.*, RNS). To accelerate blind rotations, we introduced two major optimizations: lazy reductions and RLWE-scaling. In blind rotations, multiply-accumulate operations are frequently invoked. For the purpose of reducing the cost of modular reductions whenever a modular multiplication and a modular addition is called, we perform only one modular reduction after all multiplications and additions are performed in each iteration. Meanwhile, in RNS-to-Radix conversion, we also perform lazy reductions to avoid the costly divisions in Eq. (14). RLWE-scaling is another optimization in the context of TFHE scheme. In general, it split two high-degree polynomials into several low-degree polynomials and reduces the total count of computations linearly as the number of split polynomials grows. Many other optimizations can be applied in blind rotations such as bootstrapping key unrolling and choosing well-constructed moduli, but these optimizations are dependent on the specific input LWE ciphertext or the customized moduli, which leads to loss of generality. Considering that PLB involves a total number of 8192 blind rotations, PLB is approximately 8192 times slower than a blind rotation assuming a single-thread execution. Hence, PLB is also the performance bottleneck of *HEDA*.

Coefficient-to-slot is the third most time-consuming operation (~ 1.8 seconds). The coefficient-to-slot function aims to modify the plaintext format within its corresponding ciphertext, which is necessary when multiple attributes are involved in a single aggregation clause (*e.g.*, `SUM(extendedprice*(1-discount))`). This function involves a linear transformation over the RLWE ciphertext. As it is executed much less frequently (number of packed ciphertext are much less than the total number of rows), its impact on the overall performance is limited. All the other functions, consuming several to several hundreds of *ms*, are negligible w.r.t the overall latency.

*NTT is a generalization of the discrete Fourier transform (DFT) to finite fields. It enables fast convolution on integer sequences without any round-off errors, and therefore it is useful for multiplying large polynomials. An 8192-point NTT is used to transform an 8192-degree polynomial and composed of $48k$ integer modular multiplications.

Table 7: Performance breakdown (ms) for the benchmark databases with 8192 rows based on the Intel Xeon Platinum 8163 CPU@2.5GHz.

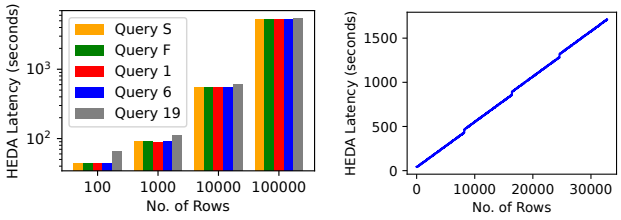
Database	Filtering	PLB	Aggregation	Others	Total
Query S	1,312 (0.3%)	398,366 (92.1%)	32,414 (7.5%)	371 (0.1%)	432,463
Query F	1,274 (0.3%)	393,487 (92%)	32,981 (7.6%)	390 (0.1%)	427,565
Query 1	424 (0.1%)	394,536 (92.1%)	32,684 (7.6%)	874 (0.2%)	428,518
Query 6	2,200 (0.5%)	400,517 (92.9%)	32,472 (7.5%)	386 (0.09%)	435,613
Query 19	21,292 (4.7%)	395,792 (87.8%)	32,472 (7.2%)	1,271 (0.3%)	450,827

6.3 DB Results

We further evaluate the SQL queries and their corresponding tables with results shown in Fig. 7. All databases are tested starting with 8192 rows, such that the whole database can be encrypted inside a single RLWE ciphertext, and the number of rows increased further in subsequent experiments. According to Table 7, all SQL queries take about 430 seconds (> 7 minutes) with 8192 rows.

Table 7 shows that the PLB is the major bottleneck for the overall performance (~ 92%) even with the acceleration of multi-threading. This can be explained by the fact that the PLB involves up to 8192 bootstrappings because the RLWE ciphertext has been converted into 8192 LWE ciphertexts during the filtering step. We adopt several optimizations to accelerate the PLB. First, as described in Sec. 5.4, we convert an RNS-based ciphertext into a radix-based one. More precisely, an RNS-based ciphertext, containing three components (corresponding to three moduli of 60-bit, 40-bit and 40-bit, respectively, as listed in Table 5), is converted into five radix-based components, each involving a 23-bit integer. Second, we adopt several approaches to optimize performance of a single bootstrapping, including lazy reductions and RLWE-scaling as elaborated in Sec. 6.2. Third, the PLB is further accelerated using multi-threading. The 8192 bootstrappings can be executed in parallel as they are independent of each other (Sec. 5.4). The Intel Xeon Platinum 8163 CPU supports up to 192 threads. Even though it is hard to achieve a speed-up of 192 times due to limited on-chip memory, multi-threading still achieves a significant acceleration for the PLB.

We then show the impact of database table size on the overall latency in Fig. 6. For all the table sizes, the latency of *HEDA* is near-linearly correlated to the number of rows in the table as shown in



(a) Latency of Queries **(b) Latency of Query F**

Figure 6: *HEDA* latency vs. the database table size.

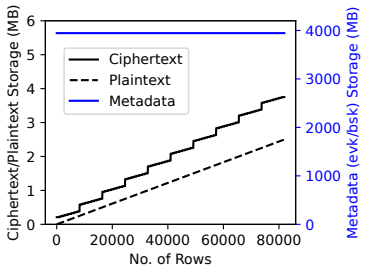


Figure 7: *HEDA* storage vs. the database table size.

Fig. 6a. The near-linearity is explained by the PLB whose latency is linear to the number of LWE ciphertexts (equal to the number of rows). If we zoom into Query F, for example, for a closer look, we observe an abrupt rise when the number of rows equals multiples of 8192, as shown in Fig. 6b. This phenomenon is caused by comparison and aggregation as they are based on RLWE ciphertext. More precisely, the computation on RLWE ciphertext, due to its SIMD nature, changes as a *step* function instead of a linear one.

Next, we evaluate the storage size for *HEDA* and compare the results with the plaintext case, as shown in Fig. 7. The encrypted database is about 1~2 times larger than the plaintext. Nevertheless, the bootstrapping key (bsk) and evaluation key (evk), used for evaluation over ciphertext, requires storage of 54MB and 3.9GB, respectively. We note that the key material size is a common challenge in the FHE domain and orthogonal to our work. This size can be hundreds of times smaller if the keys are generated on the fly [37]. The size of these keys does not scale with the database size, and therefore their impact is less significant for large databases.

In Table 8, we provide qualitative comparisons between our work and existing encrypted database solutions (*i.e.*, CryptDB [51] and SAGMA [31]). By leveraging FHE, we can significantly simplify the storage complexity without incurring overheads to both query latency as well query complexity. In particular, even in the case where the number of grouping attribute is one, the storage complexity of *HEDA* grows exactly linearly with the number of database elements (q and α are constants), instead of being dependent on the size of the query ($|Q|$ in Table 8) as in SAGMA. Second, in terms of security, most existing works, such as CryptDB and SAGMA, do not protect search frequency (while SAGMA provides some levels of empirical protections, the security cannot be formally proved). In contrast, *HEDA* proposes an end-to-end FHE-based DBMS protocol, where the security of both data items and the search frequencies are provably secure as discussed in Sec. 5.1. Lastly, for query latency,

Table 8: Qualitative and Complexity Comparisons

Work	Security $\mathcal{D}_{row,col}$	Security Search Freq.	$\mathcal{O}$(Storage)
[51]	Non-Provable	Not Protected	$\alpha\|\mathcal{D}_c\|\|\mathcal{D}_r\|^{\dagger}$
[31]	Provable	Leaks Bucket Freq.	$(B^{\star}-1+\|Q\|)\|\mathcal{D}_r\|\|\mathcal{D}_c\|$
Ours	Provable	Provable	$q(\beta\|\mathcal{D}_r\|)\|\mathcal{D}_c\|^{\ddagger}$

† α is a constant ‡ β is constant and $\beta \leq 1/1024$ * B is the bucket size in [31]

SAGMA reports an estimated aggregation time (excluding filtering since SAGMA uses SSE) around 40 seconds for $|\mathcal{D}_r| = 8,000$ rows at an 80-bit security level. In comparison, *HEDA* requires around 430 seconds for both the filtering and aggregation of $|\mathcal{D}_r| = 8,192$ rows at 128-bit security. *HEDA* does pay extra latency costs to protect access pattern. However, the query latency of *HEDA* can always be further reduced by parallelized hardware architecture, while the lost of access pattern security in SAGMA cannot be easily solved.

7 DISCUSSION

`MAX/MIN/ORDER BY`: These operators involve FHE-based sorting that comprises a number of comparisons and swaps. As shown in Sec. 5.3.2, comparison (over $\mathbb{Z}_t$) can be realized using a homomorphic subtraction, RLWE-to-LWEs conversion, and MSB extraction. Swapping two LWE ciphertexts that correspond to two integers can be achieved using a controlled multiplexer (CMUX, as illustrated in Fig. 1b of [17]). The comparison result, serving as the select signal, decides which input LWE ciphertext will be chosen and placed in front (or behind). One important open question is to implement and test the efficiency of such an approach.

Multi-way join: OLAP scenarios usually involve multi-table join predicates. Although fundamentally the operation is either a binary or numerical comparison that FHE could support, the challenge is each predicate involves two inputs from existing relations (compare to the TPC-H Query 1, 6 and 19 which one of the comparison operands is provided as the user input). We are working towards a solution that supports multi-way join without blowing up the complexity to the Cartesian product of all the involved relations.

8 CONCLUSION

In this work, we propose an FHE-based database analytical system, *HEDA*, to support SQL aggregation queries. As the first work that utilizes only FHE to support SQL queries, *HEDA* allows the users to issue aggregation queries with unbounded complexity. Further, by combining two different FHE constructions and creatively proposing the ciphertext transformation technique PLB, our system allows the users to encrypt and process their data based on the data type, to enjoy the benefit of both approaches without compromising the security. Through the comprehensive experiments, we demonstrated the feasibility of leveraging FHE to address SQL queries, making the very first step in this domain.

Despite the foundation laid by *HEDA*, it is still a much unexplored territory of FHE-based encrypted DBMS. One immediate urgency is to improve the efficiency to support larger database sizes through further cryptographic optimizations. Another line of valuable research is to widen the functionalities supported by FHE-based systems, such as `GROUP BY`, `ORDER BY` or multi-way join. These might be challenging and requires novel FHE techniques to address such needs raised by the database community.

REFERENCES

[1] 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 *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*.

[2] Arvind Arasu, Ken Eguro, Manas Joglekar, Raghav Kaushik, Donald Kossmann, and Ravi Ramamurthy. 2015. Transaction processing on confidential data using cipherbase. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE.

[3] AWS. 2022. Amazon Aurora. https://aws.amazon.com/rds/aurora/?c=db&sec=srv. Accessed: 2022-12-05.

[4] Sumeet Bajaj and Radu Sion. 2013. TrustedDB: A trusted hardware-based database with privacy and data confidentiality. *IEEE Transactions on Knowledge and Data Engineering* (2013).

[5] 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).

[6] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2015. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)* (2015).

[7] Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam O'neill. 2009. Order-preserving symmetric encryption. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 224–241.

[8] Christina Boura, Nicolas Gama, Mariya Georgieva, and Dimitar Jetchev. 2020. CHIMERA: Combining Ring-LWE-based fully homomorphic encryption schemes. *Journal of Mathematical Cryptology* (2020).

[9] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2014. (Leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)* 6, 3 (2014), 1–36.

[10] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, Bo Wang, Yuhui Wang, Haiqing Sun, Ze Yang, Zhushi Cheng, Sen Chen, Jian Wu, Wei Hu, Jianwei Zhao, Yusong Gao, Songlu Cai, Yunyang Zhang, and Jiawang Tong. 2021. PolarDB Serverless: A cloud native database for disaggregated data centers. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2477–2489. https://doi.org/10.1145/3448016.3457560

[11] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. 2015. Leakage-abuse attacks against searchable encryption. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*. 668–679.

[12] 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.

[13] Hao Chen, Wei Dai, Miran Kim, and Yongsoo Song. 2021. Efficient homomorphic conversion between (Ring) LWE ciphertexts. In *Applied Cryptography and Network Security*. Springer International Publishing, 460–479.

[14] Nathan Chenette, Kevin Lewi, Stephen A Weis, and David J Wu. 2016. Practical order-revealing encryption with limited leakage. In *International conference on fast software encryption*. Springer, 474–493.

[15] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 409–437.

[16] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachene. 2016. Faster fully homomorphic encryption: bootstrapping in less than 0.1 seconds. In *international conference on the theory and application of cryptology and information security*. Springer, 3–33.

[17] 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.

[18] Ilaria Chillotti, Marc Joye, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. 2020. CONCRETE: Concrete operates on ciphertexts rapidly by extending TFHE. In *WAHC 2020–8th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, Vol. 15.

[19] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. 2006. Searchable symmetric encryption: improved definitions and efficient constructions. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS '06)*.

[20] Léo Ducas and Daniele Micciancio. 2015. FHEW: Bootstrapping homomorphic encryption in less than a second. In *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 617–640.

[21] Taher ElGamal. 1985. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory* 31, 4 (1985), 469–472.

[22] Saba Eskandarian and Matei Zaharia. 2019. ObliDB: Oblivious query processing for secure databases. *Proceedings of the VLDB Endowment* 13, 2 (Oct. 2019), 169–183. https://doi.org/10.14778/3364324.3364331

[23] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat practical fully homomorphic encryption. *Cryptology ePrint Archive* (2012).

[24] Axel S. Feldmann, Nikola Samardzic, Aleksandar Krastev, Srinivas Devadas, Ronald G. Dreslinski, Karim M. El Defrawy, Nicholas Genise, Chris Peikert, and Daniel Sánchez. 2021. F1: A fast and programmable accelerator for fully homomorphic encryption. *54th Annual IEEE/ACM International Symposium on Microarchitecture* (2021).

[25] Tingjian Ge and Stan Zdonik. 2007. Answering aggregation queries in a secure system model. In *Proceedings of the 33rd international conference on Very large data bases*. 519–530.

[26] Craig Gentry. 2009. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*. 169–178.

[27] Craig Gentry, Amit Sahai, and Brent Waters. 2013. Homomorphic encryption from learning with errors: conceptually-simpler, asymptotically-faster, attribute-based. In *Annual Cryptology Conference*. Springer, 75–92.

[28] Edgeless Systems GmbH. 2022. EdgelessDB Official Website. https://www.edgeless.systems/products/edgelessdb. Accessed: 2022-12-05.

[29] Paul Grubbs, Kevin Sekniqi, Vincent Bindschaedler, Muhammad Naveed, and Thomas Ristenpart. 2017. Leakage-abuse attacks against order-revealing encryption. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 655–672.

[30] Hakan Hacigümüş, Bala Iyer, Chen Li, and Sharad Mehrotra. 2002. Executing SQL over encrypted data in the database-service-provider model. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*.

[31] Timon Hackenjos, Florian Hahn, and Florian Kerschbaum. 2020. SAGMA: Secure aggregation grouped by multiple attributes. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*.

[32] HEAAN. 2018. HEAAN Library. https://github.com/snucrypto/HEAAN. Accessed: 2022-12-05.

[33] IBM. 2021. HElib. https://github.com/homenc/HElib. Accessed: 2022-12-05.

[34] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. Access pattern disclosure on searchable encryption: ramification, attack and mitigation.. In *NDSS*, Vol. 20. Citeseer, 12.

[35] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. 2018. GAZELLE: A low latency framework for secure neural network inference. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 1651–1669.

[36] Seny Kamara and Tarik Moataz. 2018. SQL on structurally-encrypted databases. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 149–180.

[37] Andrey Kim, Maxim Deryabin, Jieun Eom, Rakyong Choi, Yongwoo Lee, Whan Ghang, and Donghoon Yoo. 2021. General bootstrapping approach for RLWE-based homomorphic encryption. *Cryptology ePrint Archive* (2021).

[38] Sangpyo Kim, Jongmin Kim, Michael Jaemin Kim, Wonkyung Jung, Minsoo Rhu, John Kim, and Jung Ho Ahn. 2021. BTS: An accelerator for bootstrappable fully homomorphic encryption. *arXiv preprint arXiv:2112.15479* (2021).

[39] Y.A.M. Kortekaas. 2020. *Access pattern hiding aggregation over encrypted databases*. Master's thesis.

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

[41] Wen-jie Lu, Zhicong Huang, Cheng Hong, Yiping Ma, and Hunter Qu. 2021. Pegasus: Bridging polynomial and non-polynomial evaluations in homomorphic encryption. In *2021 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 1057–1073.

[42] Kotaro Matsuoka. 2020. TFHEpp: pure C++ implementation of TFHE cryptosystem. https://github.com/virtualsecureplatform/TFHEpp. Accessed: 2022-12-05.

[43] Microsoft. 2022. Microsoft Azure SQL Server. https://azure.microsoft.com/en-us/products/azure-sql/#product-overview Accessed: 2022-12-05.

[44] Muhammad Naveed, Seny Kamara, and Charles V Wright. 2015. Inference attacks on property-preserving encrypted databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 644–655.

[45] Alexander Nilsson, Pegah Nikbakht Bideh, and Joakim Brorsson. 2020. A survey of published attacks on Intel SGX. *arXiv preprint arXiv:2006.13598* (2020).

[46] Pascal Paillier. 1999. Public-key cryptosystems based on composite degree residuosity classes. In *International conference on the theory and applications of cryptographic techniques*. Springer, 223–238.

[47] Palisade. 2022. Palisade lattice cryptography library. https://gitlab.com/palisade/palisade-release. Accessed: 2022-12-05.

[48] 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 *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*.

[49] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. 2019. Arx: An encrypted database using semantically secure encryption. *Proc. VLDB Endow.* (2019).

[50] 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 *30th USENIX Security Symposium (USENIX Security 21)*. 2129–2146.

[51] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: Protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*.

[52] Christian Priebe, Kapil Vaswani, and Manuel Costa. 2018. EnclaveDB: A secure database using SGX. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE.

[53] Brandon Reagen, Woo-Seok Choi, Yeongil Ko, Vincent T. Lee, Hsien-Hsin S. Lee, Gu-Yeon Wei, and David Brooks. 2021. Cheetah: Optimizing and accelerating homomorphic encryption for private inference. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA '21)*.

[54] M Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. 2020. HEAX: An architecture for computing on encrypted data. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1295–1309.

[55] Ronald L Rivest, Adi Shamir, and Leonard Adleman. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM* 21, 2 (1978), 120–126.

[56] Savvas Savvides, Darshika Khandelwal, and Patrick Eugster. 2020. Efficient confidentiality-preserving data analytics over symmetrically encrypted datasets. *Proc. VLDB Endow.* (2020).

[57] SEAL. 2022. Microsoft SEAL (release 4.0). https://github.com/Microsoft/SEAL. Accessed: 2022-12-05.

[58] Benjamin Hong Meng Tan, Hyung Tae Lee, Huaxiong Wang, Shuqin Ren, and Khin Mi Mi Aung. 2021. Efficient private comparison queries over encrypted databases using fully homomorphic encryption with finite fields. *IEEE Transactions on Dependable and Secure Computing* (2021).

[59] Stephen Tu, M. Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. 2013. Processing analytical queries over encrypted data. *Proc. VLDB Endow.* (2013).

[60] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. 2021. CacheOut: Leaking data on Intel CPUs via cache evictions. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 339–354.

[61] Dhinakaran Vinayagamurthy, Alexey Gribov, and Sergey Gorbunov. 2019. StealthDB: A scalable encrypted database with full SQL query support. *Proc. Priv. Enhancing Technol.* (2019).

[62] 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 2019*. 1–18.

[63] Sheng Wang, Yiran Li, Huorong Li, Feifei Li, Chengjin Tian, Le Su, Yanshan Zhang, Yubing Ma, Lie Yan, Yuanyuan Sun, Xuntao Cheng, Xiaolong Xie, and Yu Zou. 2022. Operon: An encrypted database for ownership-preserving data management. *Proc. VLDB Endow.* 15, 12 (2022), 3332–3345.

[64] Yilei Wang and Ke Yi. 2021. Secure Yannakakis: Join-aggregate queries over private data. In *Proceedings of the 2021 International Conference on Management of Data*. 1969–1981.