Enc^{2}DB: A Hybrid and Adaptive Encrypted Query Processing Framework

No Author Given

No Institute Given

Abstract. Cloud computing has become an essential infrastructure for efficient data management systems. Following the framework, data owners are now outsourcing their data to cloud service providers (CSP), which shall then be expected to provide Database Service (DBaaS). This change in database usage brings in a *deviation* of data ownership and usage, leading to inevitable privacy issues that prevent users from outsourcing database services to CSPs.
Existing solutions to this problem rely on either property-preserving cryptography schemes or a trusted execution environment (TEE). However, either solution suffers from a series of limitations. In light of this, we propose and implement a framework, namely Enc^{2}DB, following a hybrid strategy that preserves the pros of both types of approaches. Within the system, we present a micro-benchmarking test and a self-adaptive mode switch strategy that can dynamically choose the best execution path, cryptography or TEE, to answer a given query. We implement the framework over a pair of off-the-shelf RDBMSs. Empirical study over TPC-C justifies that Enc^{2}DB outperforms existing TEE or cryptographic solutions.

Keywords: privacy · trusted execution environment · query processing.

1 Introduction

Cloud computing has become an essential infrastructure for data management systems. However, while storage and computing in the cloud bring great convenience, the deviation of data ownership and usage also brings privacy risks. To ensure confidentiality, data can be kept encrypted on the CSP. Traditional data encryption [28] preserves the confidentiality of data at rest. Securing data at rest allows users to use cloud storage without exposing sensitive information, but prevents users from performing SQL queries effectively or efficiently.

A possible solution is homomorphic encryption, including fully homomorphic encryption (FHE) [14] and partially homomorphic encryption (PHE), which can provide the operability of the data while preserving data confidentiality. Property-Preserving Encryption (PPE) [21] can preserve some attributes of plaintext data, such as orders. However, simply relying on these encryption schemes may introduce a significant time or space cost. Furthermore, it cannot answer queries that contain both comparison and algebraic operations at the same time.

Another alternative is to use a trusted execution environment (TEE) [25], which creates a secure area on the CPU, so encrypted data can be decrypted and computed in TEE. Due to that, many encrypted database systems based on enclaves have emerged [8,17,19,27]. Although it functionally supports all types of tasks found in SQL, including comparison and algebraic operations, TEE may introduce an additional burden when switching pages into and out of the Enclave, which is a small fraction of the main memory. Given the limitations of both approaches, it will be more practical if a system can benefit from the pros of both and avoid the cons as much as possible. Driven by that, we are motivated to propose a hybrid and adaptive encryption query processing solution that can switch between both approaches depending on the query task at run-time. However, there exist a series of challenges to achieve that. First, the above goal requires monitoring the internal state of the Enclave, which is protected as a black-box. Second, the run-time switch requires evaluating the cost of both approaches, which asks for an extra cost model and query optimizer.

In this paper, we present a system, namely Enc^{2}DB, implemented in both PostgreSQL and openGauss, an open source database proposed by Huawei. Enc^{2}DB proposes a hybrid solution that uses both software (cryptography) and hardware (TEE) to improve the efficiency of ciphertext data query and realizes fully encrypted storage and execution of query workload, as well as transparent processing of user-side query requests. Our main contributions are as follows:

– We propose and implement both the software-based and TEE-enabled mode to answer SQL queries over encrypted data.
– We propose a cyphertext-aware indexing mechanism over ORE, to further improve query efficiency in the software-based mode.
– We propose a hybrid self-adaptive strategy that dynamically switches to the most appropriate mode (*i.e.,* software-based and TEE-enabled) at runtime.

In the following, we first introduce related work in Section 2 and then explore the system architecture in Section 3, as well as the details of software and TEE-enabled modes. In Section 4, we introduce a series of optimizations. The empirical study is provided in Section 5, and we conclude in Section 6.

2 Related Work

2.1 Cryptography Schemes

Homomorphic Encryption Homomorphic encryption is proposed for mathematical calculation of the ciphertext without obtaining the key. According to the types of ciphertext calculation, it can be divided into FHE and PHE. In 2009, Gentry [14] implemented the first fully homologous encryption scheme. Although over the years, the algorithm of FHE has experienced significant improvement in the aspect of efficiency, it is impractical for real-world usage. In comparison, PHE supports only limited types of ciphertext calculation, but provides practical efficiency. PHE is divided mainly into additive homomorphic encryption (AHE) (*e.g.,* Paillier [20]) and multiplicative homomorphic encryption (MHE) (*e.g.,* El-Gamal [13]). At present, we adopt the state-of-the-art symmetric AHE and MHE solution, namely SAHE and SMHE proposed in Symmetria [26]. They retain the full range of homomorphic operations that asymmetric schemes support while offering the same level of semantic security (IND-CPA).

Property Preserving Encryption For range queries involving comparisons, we turn to PPE, which has been used in CryptDB [23], Monomi [28] and Seabed [22]. The ciphertext of the PPE retains some attributes of the plaintext, *e.g.,* the order of the underlying plaintext values. The relative order of the plaintext values can be obtained directly by comparing the encrypted values under Order Preserving Encryption (OPE) [5]. Boldyreava [9] proposes an attribute-preserving encryption scheme, but it is a deterministic scheme that preserves the frequency information of plaintext data. Currently, the Order-Revealing Encryption scheme (ORE) [10] and its branch scheme are nondeterministic solutions, which can reveal less information under the premise of guaranteeing higher operational efficiency, thus providing better security than OPE.

2.2 Trusted Execution Environment

Trusted Execution Environment (TEE) [25,31] builds a secure area on the CPU to ensure that the programs and data loaded there are protected in terms of confidentiality and integrity. The principle of TEE is to divide the hardware and software resources of the system into two execution environments: the trusted execution environment and the common execution environment, which are securely isolated. All major CPU vendors have implemented their TEE (*e.g.,* ARM TrustZone, Intel SGX, and AMD SEV) to provide a secure execution environment, commonly referred to as an enclave [1,3,16].

Encrypted database systems can employ TEE to preserve confidentiality in query processing by decrypting the ciphertext and performing complex calculations over plaintext within TEE. SGX is a representative TEE framework proposed by Intel, which encapsulates the security operations of legitimate software in an enclave to protect it from malware attacks. That is, even the OS or the VMM (Hypervisor) cannot access or affect the code and data inside the enclave. EnclaveDB [24] is based on SGX and processes encrypted queries in an enclave, which allows only precompiled queries and assumes that all data can fit in memory. The always encrypted Azure database [6] uses some enclave-based defined functions for computation of ciphertext. However, such a non-intrusive design leads to possible information leakage and performance degradation. FE-in-GaussDB [32] combines encryption algorithms with TEE to securely perform various operations on ciphertext data, including matching, comparison, etc.

3 System Architecture

In this section, we introduce the basic structure of Enc^{2}DB (Encrypted Database with Enclave), which provides two deployment modes, *i.e.,* software-only and TEE-enabled. To start with, we first propose the software-only mode, namely EncDB (Encrypted Database). After that, we discuss how TEE can cooperate

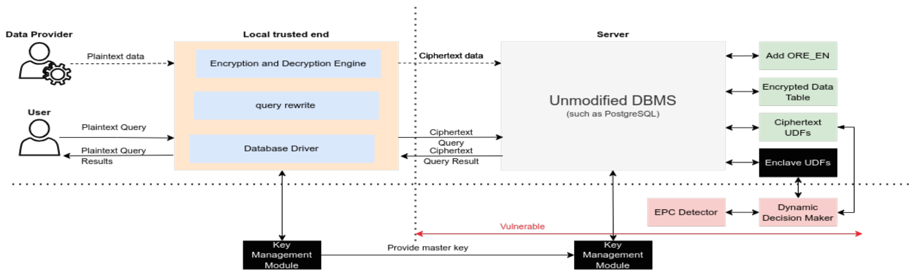


Fig. 1. System Architecture with components differentiated by mode: TEE-enabled (black and red) and Software-based (green).

with EncDB by enabling complex tasks to run inside the Enclave, leading to the second mode, Enc^{2}DB (detailed in Section 3.3).

The complete system is divided into two parts: the application server (trusted client) and the untrusted server. Its architecture is shown in Figure 1, which includes the following components deployed in both the client and the cloud.

Encryption and Decryption Engine: holds the encryption key, encrypts plaintext data and decrypts the result received from the cloud. The encryption module also generates and stores meta-encryption data, *i.e.,* the mapping of plaintext operators to the relevant ciphertext ones, the supported encryption schemes towards different data types in the database.

Query Rewriter: convert plaintext queries into ciphertext queries that are sent through the network to the server.

Database Driver: PostgreSQL supports multiple database application programming interfaces (such as libpq, JDBC, ODBC, etc.) to submit query requests. The application driver helps the client to send the rewritten encrypted SQL to the server and receive the encrypted result from the cloud.

Server-side Computing Services: Leverage a set of user-defined functions (UDFs) to perform operations on encrypted data.

The pipeline to answer an SQL query in the architecture is as follows. The data provider uploads plaintext data, and the client encrypts each column under one or more encryption schemes according to the expected operations, replacing the original column name with a random string for anonymization. The user enters a plaintext query, which is rewritten and sent to the server in the form of a ciphertext counterpart. The server implements the relevant operation logic for the ciphertext calculation in advance, performs a query and operation on the encrypted SQL, and returns the encrypted result obtained to the client. The client decrypts the received ciphertext to obtain the plaintext query result. The whole process is completely transparent to the user, and the server is unaware of the content of either the query or the result.

3.1 Security model

Our goal is to ensure that the servers return the correct results while unable to obtain the plaintext of the stored data. We assume that the database server (*i.e.,* the adversary) is honest-but-curious, *i.e.,* strictly follows the query protocol but curious about the data. Our security study and proof in [4] show that our software-only mode (with cipher index) guarantees IND-OCPA, and the other modes are at least as secure as the software-only mode.

For sensitive columns that are encrypted and outsourced to the database server, they should be kept secret against the server before and after queries. In particular, side-channel attacks and access pattern leakage are beyond the scope of this work.

3.2 Software-based mode, EncDB

In our software-only implementation mode, a column in a table is to be stored in ciphertext form under one (or more) of AHE, MHE, ORE, and AES. Given an SQL statement, the column names and operands are, respectively, replaced by desensitized names and ciphertext under a predefined encryption logic. This process takes place at the user's local trusted end and is deployed by a small connection pool, which has many advantages such as transparency to the upper layers, no client awareness, and easy migration. After receiving the encrypted query statements, the server will perform the relevant ciphertext calculation according to the pre-configured user-defined functions and finally returns the encrypted query results to the user, and the local trusted end will decrypt and obtain the plaintext results, which is completely transparent to the application.
Advantages. A major benefit is that it has no hardware requirement and has fewer deployment costs.
Disadvantages. Obviously, there exists large data redundancy, as a column is copied and encrypted into many ciphertext ones under different encryption schemes, resulting in a large overhead. In addition, the query is done using UDFs with a huge computational cost. Moreover, some complex expressions are hard to compute in this mode. For example, none of the current encryption schemes can support homomorphic addition/multiplication and comparison simultaneously. For queries containing both operations over the same operands, another round of interaction between the client and the server is required.

The software-only mode is a major research direction in cryptographic database systems [15,23,28,30]. The EncDB in our system is an advanced solution in this trend that absorbs many of the advantages of the previous work, especially those symmetric encryption schemes. Based on this implementation, we further introduce TEE and propose a hybrid solution that benefits from both software and hardware security support. In addition, we also present a self-adaptive strategy to enable both techs to cooperate at runtime, which results in a large improvement in the overall performance of the system.

3.3 TEE-enabled mode, Enc^{2}DB

The TEE-enabled mode aims to improve the efficiency and address the unsupported operations under software-only mode. In the implementation of the TEE enabled mode, UDFs are packaged as independent trusted bridging functions, which are used to enter secure memory, *e.g.,* Enclave, decrypt encrypted data, and encrypt the results according to the required encryption mode after performing the corresponding computation task. Finally, the computed results are returned to the database system, which also maintains the high security of the data to the host. Since the computation is performed in the secure area, we never need to worry about homomorphic or property-preserving functionality in the encryption scheme. Instead, any symmetric encryption can be considered, as long as it is efficient and secure enough. In addition, compared to the presence of various encrypted copies (each corresponds to a specific encryption scheme) in software-only mode, only one encrypted copy is stored on the server side. Suppose that Intel SGX is used in Figure 1, trusted bridging functions are declared in SGX via the Enclave Definition Language (EDL) and are enclosed as UDFs to interact with the database.

Notably, since all secure calculations in Enclave are performed on plaintext data, keys that exist on the local trusted side must be securely transferred to Enclave instances running in the cloud over untrusted channels. The details of the implementation of UDFs and key transfer are provided in [4]

3.4 Self-adaptive switch between TEE-enabled and software modes

Using trusted bridging functions to wrap UDFs can support arbitrary type of query over ciphertexts, but trusted hardware also suffers from a series of limitations, *i.e.,* insufficient safe memory space for SGX [7,8], which can lead to severe page replacement in concurrent transaction processing scenarios, even worse than that of software-only implementations. Therefore, outsourcing all queries to trusted hardware may introduce extra cost. Driven by that, we select to mitigate the problem by processing the encrypted query using both software and TEE-enabled modes together. Depending on whether the processing is dynamically switched between both modes at run-time or not, we propose two strategies, namely, static switch and self-adaptive switch.

Static mode switch Specifically, in the static switch strategy, for operations efficiently addressed by the software mode, we choose not to rely on TEE. To this end, we conducted theoretical and empirical studies on encryption schemes to determine which can be efficiently addressed in software-only mode (shown later in Figure 4(a)). ORE has the highest cost, which is consistent with the fact that ORE has a larger space and time complexity than other encryption schemes. Intuitively, ORE-related operations is the first target we must kill (*i.e.,* transfer the task to TEE) in software-only mode.

All operations involving OREs can be replaced with AES with the help of TEE to save time and space. Assuming that ORE operations are statically replaced using AES and supported in TEE, an SQL query shall replace the corresponding columns involved in all comparison predicates using AES columns, instead of ORE ones. On the server side, UDFs involving comparison over columns are all moved to Enclave, who shall decrypt the AES ciphertext, perform comparison over plaintext, and return the comparison results. This static configuration to perform a fixed type of predicates on ciphertext operands, *i.e.,* comparison, over TEE is referred to as a static switching mechanism.

If all predicates of the ciphertext operands are moved and fully rely on TEE, the static switch becomes the TEE-enabled mode shown in Section 3.3. This simple replacement of an encrypted column to reduce space-time overhead does improve the overall performance of an encrypted database system. If all types of predicates over ciphertext operands are handed over to SGX, it works well in small-scale or low-concurrency scenarios. However, in high-concurrency scenarios, SGX's secure memory space, *i.e.,* 128MB in SGXv1 and 64GB (512 in principle) in SGXv2 [12], will soon be full, resulting in severe paging. According to the previous analysis, if a page fault occurs, all operations will take almost twice as long as those without a page fault [7,11].

Self-adaptive mode switch As discussed above, the limited space of Enclave will inevitably introduce too much paging task in high-concurrency transactional scenarios. An ideal solution towards this problem is to obtain the state of the current SGX in real time and dynamically determine whether the current ciphertext predicates shall be performed in software mode or TEE-enabled one. Unfortunately, the official SGX SDK does not provide such an interface to return the remaining capacity of the enclave page caches (EPC) in real time.

To address that, we present a microbenchmark to estimate the residual capacity of EPC at run-time. Intuitively, if the remaining capacity of Enclave memory is insufficient, page replacement will occur, which will significantly affect efficiency; then the running time of a UDF calculation task executed in SGX can indirectly reflect the status of the remaining capacity of EPC. In other words, if the execution time of a predefined task in SGX is significantly higher than its expectation, a page replacement probably occurs. In this way, the current remaining capacity of the EPC can be dynamically inferred. In the sequel, we refer to this strategy as the Enclave microbenchmark. Specifically, in a new thread, we predefine a particular task and trigger it with a certain time interval. This task is executed in Enclave and the execution time of this task is recorded. Because it is a fixed task, the expected running time is also fixed.

Remarkably, there is no restriction on the predefined task of the benchmark, as long as the access of the task test itself to the data should be random enough to resist against the caching effect of the page replacement algorithm, so that the task can cause enough page fault exceptions. Thus, differentiation in run-time can be easily observed. Generally, if the data access of the task is not random enough, there will be less page fault, so it cannot accurately reflect whether page replacement is happening at present. After an exhaustive empirical study, which shall be shown in Section 5.2, in Enc^{2}DB we adopt the binary search after quick sorting in Enclave memory as a benchmark task.

Cost estimation model The main task of the self-adaptive switch strategy is to dynamically switch to the correct mode according to the current load of the EPC. The execution mode for each particular predicate over ciphertext can be viewed as different physical operators (*i.e.,* software mode or TEE-enable mode, each corresponds to a UDF) in the query execution plan (QEP). We shall also deploy a cost model to enable dynamic switch between physical operators, *i.e.,* UDFs. For ease of discussion, we refer to the UDF cost of the software mode and the TEE-enabled mode as C_{soft} and C_{TEE}, respectively.

$$C_{soft} = C_{calc} + C_{decide} \tag{1}$$

C_{soft} is shown in Equation (1), where C_{Calc} is the cost in UDF execution, $C_{decision}$ is the cost of the decision itself. Obviously, C_{decide} depends mainly on the number of UDFs of the same kind calculated on the same path. For different UDFs, C_{Calc} is different. For instance, in the symmetric cryptographic homomorphic algorithm, the efficiency of AHE is always higher than that of MHE. An estimate of C_{calc} can be given based on the computational flow of different UDFs and their complexity. In addition, compiled assembly instructions or run-time CPU clock cycles based on UDF code can be used as a reference to estimate C_{calc}. For C_{TEE}, the total cost can be found as follows.

$$C_{TEE} = C_{fixed} + C_{calc} + C_{runtime} + C_{decide} \tag{2}$$

C_{fixed} is the start-up cost of the TEE. For example, in SGX, it is reflected mainly as the $ECALL/OCALL$ invocation overhead. C_{calc} is the computational cost of UDF execution. Since the main logic of execution on trusted hardware is to decrypt AES, perform the computation on plaintext, and then encrypt the result according to the encryption format required by the UDF, the dynamic part of the computational cost here depends on the parameters of the UDF, *i.e.,* the specified encryption scheme requirements. In decision making, it is necessary to dynamically estimate this part of the computational cost for different parameters. $C_{runtime}$ is the additional load overhead of the trusted hardware at run-time, *e.g.,* in SGX it is mainly expressed as the additional computational cost during page replacement. Its value continuously varies with the severity of page replacement. The main measurement method is to dynamically estimate $C_{runtime}$ from the microbenchmark test introduced above.

4 Further Optimizations

4.1 Cipher index

B(+) tree is a fundamental tool to accelerate queries with range or equivalent predicates. In an encrypted database, the client encrypts the numeric data and transmits the ciphertext as a string, which is typically stored in the form of “text” or “blob” on the server side. Due to the different representation forms of plaintext and ciphertext, the classical comparison logic of the plaintext database is no longer applicable to ciphertext. Correspondingly, B(+)-tree cannot directly support the query over ciphertext, *i.e.,* ORE in our software-only mode.

In this part, we present an index scheme that supports equivalent query, range query, and related aggregate functions over ORE ciphertext and is compatible with both PostgreSQL and openGauss. Meanwhile, we ensure the transparency of the user, who only need to enter the following to create server-side ciphertext indexes in the same way as with plaintext databases.

```sql
CREATE INDEX idx_name ON tb_name(col_name);
```

Intuitively, one possible solution to this index is to replace the comparison operator within the B(+) tree using an ORE-based UDF that can return comparison results over the ORE ciphertext. However, the cost model and the query optimizer are unaware of the cost of this modified B(+) tree, so that the query plan may not correctly employ this index, *i.e.,* incorrectly uses Seq SCAN instead of Index SCAN. To address that, our model relies on UDT (user-defined type) and UDO (user-defined operation) instead of purely UDF over ORE ciphertext. Detailed implementations for UDT and UDO, as well as how range queries are executed accordingly, are described in [4].

4.2 within-SGX caching

In the design of a cryptographic database system, cipher calculation is costly, so it is a waste of resources, especially when a repeated query (*resp.,* sub-query) occurs. If we can reduce the duplicate cipher computation, it will not only improve efficiency but also save much secure memory space. As AES ciphertext are repeatedly decrypted to perform predicate computation within SGX, it is possible to cache the cipher-to-plain text mapping of AES within SGX, such that the follow-up decryption task towards the same AES ciphertext can be accelerated significantly. LRU or LFU can be potential choice for implementing the cache algorithm. As cached data exists in limited secure memory, the cache algorithm also needs to save memory as much as possible, so LRU is a more appropriate choice. Experiments show that ciphertext caching has a 15%-20% improvement over the TPC-C benchmark. For the aspect of cache capacity, in Enc^{2}DB we set it configurable, so that it can be adjusted according to different usage scenarios and properties of different SGX platforms.

4.3 SGX task pool for batched ecall

In an Enclave program, access to the enclave program from the untrusted side is done through a predefined bridge function called ECALL. The verification of ECALL is also a computationally expensive process, so if a ECALL call enters the Enclave, whose computation task is too simple, the verification of ECALL accounts for most of the cost, especially in face of a large number of concurrent short tasks. Hence, it is important to reduce the number of ECALL with respect to small tasks. In particular, SGX has proposed *Switchless Call* [2] technique to reduce this type of overhead by deploying worker threads inside the Enclave to asynchronously acquire tasks for execution. However, according to *Switchless Call*, work threads can only be specified statically and the size of the task pool is fixed, which makes it impossible for cryptographic database systems to achieve more fine-grained adjustment and optimization according to the characteristics of ciphertext (UDF) computation tasks.

To address that, we deploy a ECALL task pool at the untrusted end. Whenever a thread calls ECALL, a task object, containing its ID and parameters, is generated and stored in the task pool. The working thread at the trusted end takes the ECALL task from the task pool and executes it. After a certain number of tasks in the pool or a certain period, all tasks are transferred to the processing pool, at which time the work thread in the safe zone will iterate through the tasks from and execute them accordingly. After all tasks in the processing pool have been executed, new tasks in the pool can be accepted. The reason for dividing the pool into two parts is that threads in the safe zone cannot share the same lock mechanism with threads in the nonsafe zone, because the lock object for non-safe threads is provided by the standard library, while the lock object for safe zone threads is provided by the SGX development library. This pattern effectively prevents the EENTER/EEXIT instructions from being called, thus reducing additional overhead. In particular, when the task pool is full or all work threads are busy, the ECALL call degenerates into a normal form of call.

More importantly, as *Switchless Call* is a multithreaded model, and a database system following that model, *e.g.,* PostgreSQL, cannot effectively use *Switchless Call* technique and must implement an interprocess concurrency control instead. Due to that, the task pool proposed above is fundamental.

5 Evaluation

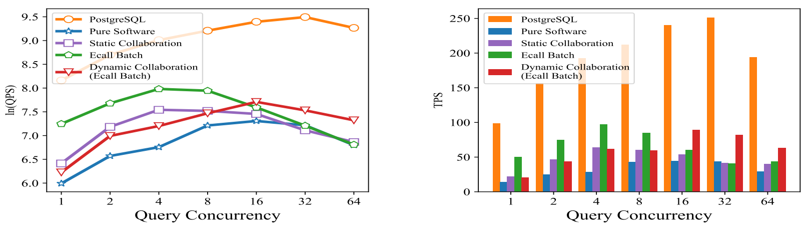
We verify the performance of our system in both the software-only mode and the TEE-enabled mode in the TPC-C benchmark implemented using sysbench[18] with the Sysbench-TPCC test script[29]. The experimental study is conducted on an Intel SGX-enabled machine with an EPC size of 128MB, equipped with an 8-core, 16-thread Intel Xeon E-2288G CPU with 512KB, 2MB, and 16MB of L1,L2,L3 cache, respectively. In particular, although larger EPCs can be supported on SGXv2 [12], there are also paging faults for large databases. That is, our self-adaptive mode switch solution also applies.

We compare our system with a pair of representative baselines, including the original (plaintext) implementation of the same database, *i.e.,* PostgreSQL/openGauss[1], and the state-of-the-art software-based encrypted database, namely Symmetrial [26]. Since the TPC-C test cannot manually control the read/write ratio, we also perform experiments on a synthetic dataset (1 million records) by allowing configurable read/write ratio.

5.1 Overall throughput

The overall performance in both the aspects of latency and TPS (#Transactions per second) by varying the concurrency is shown in Figure 2, covering the baseline (plaintext) system, the software-only mode, the static (mode switch) TEE-enabled mode w/wo task pool, and the dynamic (mode switch) TEE-enabled mode. Since the CPU contains 16 cores, all solutions show a decrease in QPS when the concurrency number exceeds 16. The software-only mode constantly exhibits a $< 10\times$ performance loss compared to the plaintext database. In the static TEE-enabled mode, severe page replacement occurs in high concurrency

[1] We implement Enc^{2}DB on both PostgreSQL and openGauss, as the results on both system are consistent we select to showcase only that of PostgreSQL due to space limit and popularity among the audience.



(a) QPS (b) TPS

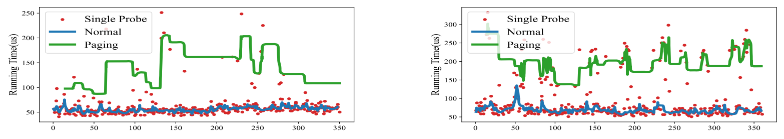
Fig. 2. Overall results on TPC-C

scenario, performs even worse than the software-only mode. In the low concurrent scenario, the performance is better and the loss is $< 5\times$ compared to the plaintext baseline. In low-concurrency scenarios, task pooling performs the best among all solutions due to the elimination of computational overhead caused by ECALLs. The task-pool mode is more advance for multi-threading, but performs not so good in multi-process tests. However, since PostgreSQL's concurrent implementation is multiprocess mode, it does not perform as well in high-concurrency scenarios as the static TEE-enabled mode without pool. The dynamic TEE-enabled mode does not perform as well as the other two, *i.e.,* static TEE-enabled mode w/wo task pool, in the lower concurrency scenarios, due to the low incidence of page replacement. Instead, it performs best in high-concurrency scenarios where page replacement occurs frequently. In addition, SGX task pool shows advance performance in all TEE-enabled mode, thus in the rest experiments it is by default turned on in TEE-enabled mode solutions.

5.2 Self-adaptive mode switch test

In the dynamic mode switch scheme of the TEE-enabled mode shown in Section 3.4, we present a micro-benchmark to estimate the state of the current EPC operation, so as to switch between the ciphertext predicate solution self-adaptively. Figure 3 shows the time cost of the microbenchmark task during the TPC-C test, where the scatter refers to the time it takes for each microbenchmark task to be executed, and the dashed line indicates the average running time of the current mode with (replacement state) or without page replacement (normal state), respectively. Obviously, the execution time of the microbenchmark task in the replacement state is generally twice as long as that in the normal state. It can be seen that, in the high-concurrency scenario, the probing tasks take more time. The estimation results are in line with the ground-truth state inside the Enclave, hence it can inform us of the real-time state of secure memory for self-adaptive mode switch. In addition, we also conducted experiments to explore performance by varying the types of microbenchmark probe and the volumes of task workload (see details in [4]).

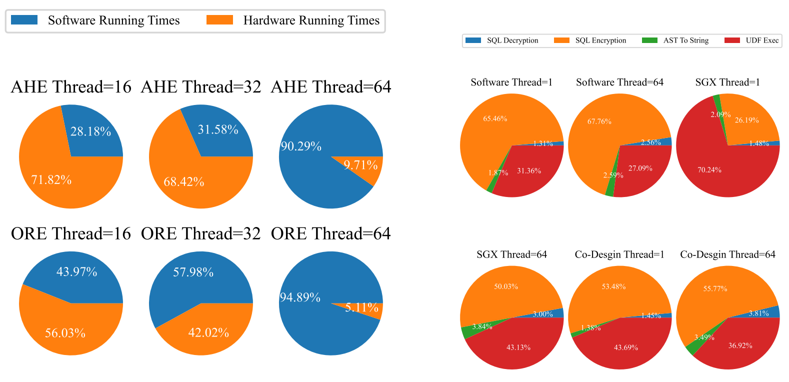
Figure 4(a) shows the number of times homomorphic addition and ORE are executed in different concurrency settings. Obviously, with increasing concurrency, the SGX memory space is gradually occupied; at this time, under the regulation of the self-adaptive mode switch, the UDF calculation using Enclave becomes less, and more UDFs are calculated outside Enclave, via software-only mode. The main reason is that, in the high-concurrency scenario, too many UDFs



(a) thread=4 (b) thread=64

Fig. 3. Extra cost of Probing Task during TPC-C

enter the feasible hardware at the same time; thus, more data need to be processed simultaneously inside. As page replacement occurs, the micro-benchmark probing task execution time will increase. With the help of the micro-benchmark, Enc^{2}DB dynamically schedules the execution mode of UDFs, to control the use of EPC space to reduce the extra overhead of page replacement.



(a) (b)

Fig. 4. (a) Time-consumption for different phases under TEE-enabled mode with self-adaptive switch strategy; (b) Time-consumption for components in Enc^{2}DB

The percentage of time consumed for different phases under TPC-C is shown in Figure 4(b). In the software-only mode, the most time-consuming module is SQL Encryption. Due to the high proportion of insert operations in TPC-C tests, encryption dominates in the pie chart. If there is more pure read request, its proportion will be reduced. The second consuming part is the execution of the UDF predicate, which reflects the proportion of ciphertext calculations involved in the query. It is worth noting that in the implementation of static mode switch, when the concurrency number is 1, UDF evaluation takes more time than encryption. The reason is that after replacing ORE with AES (under TEE-enabled mode), all plaintext columns now correlate to exactly one (AES) column (instead of 4 *i.e.,* AHE, MHE, ORE and AES), so the required encryption operation time will be reduced significantly. This is also one of the important reasons why the efficiency of TEE-enabled mode with static mode switch strategy is higher than that of software-only mode.

In addition, we further tested the UDF execution time for different ciphertext computations, including AHE, MHE and ORE for the addition, multiplication and comparison predicates, respectively, as shown in Figure 5.

In the case of a single thread, *i.e.,* left of Figure 5, no paging in SGX is triggered, the static mode switch with task pool performs the best, mainly be-

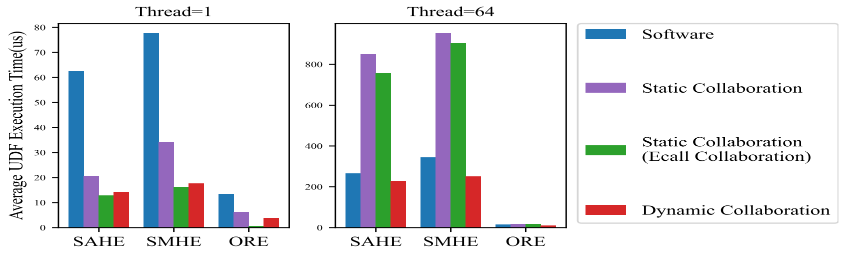
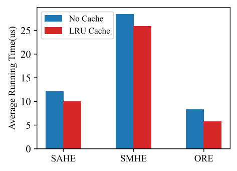
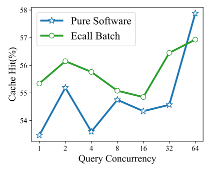


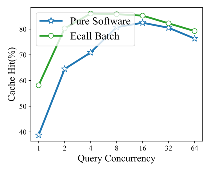
Fig. 5. Time consumption for different predicates under different modes



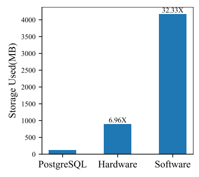
(a) cache module



(b) cache hit



(c) replacement



(d) storage space

Fig. 6. (a) With-in SGX cache; (b-c) client-side cache effect; (d) storage expansion

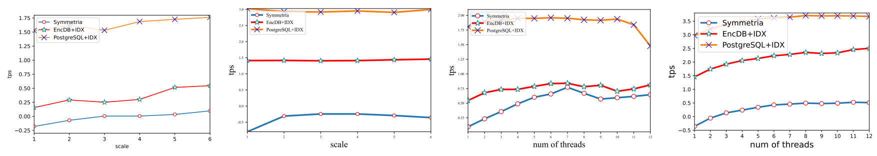
cause the extra overhead of ECALL is eliminated in this mode. In comparison, the self-adaptive mode switch with task pool performs worse than the static one. This means that the execution path will dynamically select the TEE to realize at this time, but the microbenchmark probing itself will incur time overhead.

In high-concurrency setting, where the client initiates 64 connection requests at the same time, the phenomenon is reversed because page substitution occurs for static mode switch strategy, resulting in a significant discount in efficiency. At this time, regardless of whether the task-pool mode is turned on or not, the computational overhead is about four times that of the software-only mode. In this group of experiments, the self-adaptive mode switch is the most efficient solution because TEE can be dispatched dynamically, which minimizes the cost of page substitution while making full use of the advantages of software computing.

5.3 Effect of within-SGX cache

We have also conducted experiments to test the overall improvement of system performance of the within-SGX cache module, which is deployed in secure memory to cache AES-decrypted data. The experimental results are shown in Figure 6(a). From the results of the experiment, the cache module can provide a 10%-40% improvement in efficiency.

Additionally, the cache module is deployed on the TEE of the server, which eventually significantly improves the efficiency as shown in Figure 6(a). Inspired by that, we also try to deploy a cache module in the client's AES decryption component and explore its performance; details are shown in Figure 6(b) and Figure 6(c). Figure 6(b) shows the cache hit rate. Its performance is around 50%, and the improvement is not obvious enough. Figure 6(c) shows the cache replacement rate, that is, the probability that the cache is full and the old data need to be replaced each time the cache is missed. The higher the rate, the worse the cache performs. The replacement rate on the client side is also too high, and the effect is not ideal. The main reason is that most of the decrypted data are



(a) r/w balanced (b) read-only (c) r/w balanced (d) read-only

Fig. 7. TPS in different scenarios: (a-b) single thread; (c-d) multi-thread

expression results, the randomness of which is too large. Therefore, we select not to implement the AES decryption cache on the client side in Enc^{2}DB.

5.4 Storage expansion

Besides, we also compare the expansion of the storage space of our solutions. The experiment is set to the TPC-C test script with parameters $table = 1$ and $scale = 1$. The specific experimental data are shown in Figure 6(d).

In software-only mode, the expansion of storage space is at least 30 times larger than that of the original database, while in the TEE-enabled mode, the expansion of storage space is about 7 times. The main reason is that in the TEE-enabled mode, the ORE encryption columns, whose ciphertext takes up too many bytes, are eliminated.

5.5 Effects of cipher index in software-only mode

For ease of discussion, we name the software-only mode EncDB and the mode with cipher index introduced in Section 4.1 as EncDB+IDX. We test the QPS performance of EncDB+IDX, Symmetria and PostgreSQL under different data volumes and concurrency settings in read-only and read/write workloads. The read-only test is performed with respect to the "stock level" transaction, and the proportion of equivalent query and range query execution is 1:1. In the read-/write balancing scenario, all transactions supported by TPC-C are executed, including read/write statements.

A joint index for the equal predicate columns corresponding to "d_id" and "d_w_id" is established on the "district" table. The "order_line" table constructs a joint index for "the DET column of ol_w_id, the DET column of ol_d_id, and the ORE column of ol_o_id". The "stock" table builds a joint index for the "DET column of s_w_id, DET column of s_i_id". Here $\%d$ is a random generated value. TPC-C supports users to customize the volume of the table, and we perform this group of experiments over six different volumes, *i.e.,* with $0.3, 0.6, \ldots, 1.8$ million tuples in "order_line" table, respectively.

In the scenario of read/write balanced, EncDB+IDX performs significantly better than Symmetria. Although the write operation involves the time-consuming task of creating and updating indexes, experiments show that the write operation for large tables does not change the performance advantage of EncDB+IDX. As shown in Figure 7(a), the TPS of EncDB+IDX is 1.2 times higher than that of Symmetria. The throughput advantages of EncDB+IDX and postgresql+IDX, which maintain the index structure, do not decrease with the increase in table capacity, showing the excellent performance of reading and writing on large tables. In the read-only scenario, as shown in Figure 7(b), the TPS of EncDB+IDX is about 45-160 times larger than Symmetria.

In addition, we also tested the performance under high-concurrency scenarios. We fixed the volume of "order_line" as 1.8 million tuples, and varied the number of concurrent threads from 1 to 12. As shown in Figure 7(c), when the number of threads is greater than 7, the TPS of EncDB+IDX and Symmetria exhibit a fluctuating but downward trend. As the number of threads increases, PostgreSQL+IDX also suffers performance degradation.

We have also conducted extensive experiments on the aspect of QPS and latency (detailed in [4]), the phenomenon of which is in line with that of TPS.

6 Conclusion

In this work, we present and implement Enc^{2}DB[2], an encrypted database built on PostgreSQL and openGauss. On the one hand, Enc^{2}DB implements a ciphertext index via UDT and UDO that is easily configured and natively supported by the query optimizer. On the other hand, Enc^{2}DB can be deployed in either software-only mode or TEE-enabled mode, each corresponds to different practical scenarios. In addition, in TEE-enabled mode, we, for the first time, present a self-adaptive mode switch strategy that dynamically chooses the suitable mode to execute a given query. The switch strategy fully utilizes the benefit of both cryptographic schemes and TEE at run-time.

References

1. Arm trustzone (2009), https://documentation-service.arm.com/static/5f212796500e883ab8e74531?token=
2. Intel(r) software guard extensions sdk for linux* os. (2018), https://download.01.org/intel-sgx/linux-2.2/docs/Intel_SGX_Developer_Reference_Linux_2.2_Open_Source.pdf
3. Intel (2023), https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions.html
4. Anonymous Technical Report. Enc^{2}DB: A Hybrid and Adaptive Encrypted Query Processing Framework. https://anonymous.4open.science/r/paper-2618/Enc2DB_DASFAA24.pdf
5. Agrawal, R., Kiernan, J., Srikant, R., Xu, Y.: Order preserving encryption for numeric data. In SIGMOD 2004.
6. Antonopoulos, P., Arasu, A., Singh, K.D., Eguro, K., Gupta, N., Jain, R., Kaushik, R., Kodavalla, H., Kossmann, D., Ogg, N., et al.: Azure sql database always encrypted. In SIGMOD 2020.
7. Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'keeffe, D., Stillwell, M.L., et al.: Scone: Secure linux containers with intel sgx. In OSDI 2016.
8. Bailleu, M., Thalheim, J., Bhatotia, P., Fetzer, C., Honda, M., Vaswani, K.: Speicher: Securing lsm-based key-value stores using shielded execution. In FAST 2019.

[2] We shall release the source code once this work is published.

9. Boldyreva, A., Chenette, N., O'Neill, A.: Order-preserving encryption revisited: Improved security analysis and alternative solutions. In CRYPTO 2011.
10. Boneh, D., Lewi, K., Raykova, M., Sahai, A., Zhandry, M., Zimmerman, J.: Semantically secure order-revealing encryption: Multi-input functional encryption without obfuscation. In EUROCRYPT 2015.
11. Costan, V., Devadas, S.: Intel sgx explained. Cryptology ePrint Archive (2016)
12. El-Hindi, M., Ziegler, T., Heinrich, M., Lutsch, A., Zhao, Z., Binnig, C.: Benchmarking the second generation of intel SGX hardware. In DaMoN 2022.
13. ElGamal, T.: A public key cryptosystem and a signature scheme based on discrete logarithms. IEEE TIT **31**(4), 469–472 (1985)
14. Gentry, C.: Fully homomorphic encryption using ideal lattices. In STOC 2009.
15. Hacigümüş, H., Iyer, B., Li, C., Mehrotra, S.: Executing sql over encrypted data in the database-service-provider model. In SIGMOD 2002.
16. Kaplan, D., Powell, J., Woller, T.: Amd sev, http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf
17. Kim, T., Park, J., Woo, J., Jeon, S., Huh, J.: Shieldstore: Shielded in-memory key-value storage with sgx. In EuroSys 2019.
18. Kopytov, A.: Sysbench. https://github.com/akopytov/sysbench (2017)
19. Mishra, P., Poddar, R., Chen, J., Chiesa, A., Popa, R.A.: Oblix: An efficient oblivious search index. In S&P 2018.
20. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In EUROCRYPT 1999.
21. Pandey, O., Rouselakis, Y.: Property preserving symmetric encryption. In EUROCRYPT 2012.
22. Papadimitriou, A., Bhagwan, R., Chandran, N., Ramjee, R., Haeberlen, A., Singh, H., Modi, A., Badrinarayanan, S.: Big data analytics over encrypted datasets with seabed. In OSDI 2016.
23. Popa, R.A., Redfield, C.M., Zeldovich, N., Balakrishnan, H.: Cryptdb: processing queries on an encrypted database. Communications of the ACM **55**(9), 103–111 (2012)
24. Priebe, C., Vaswani, K., Costa, M.: Enclavedb: A secure database using sgx. In S&P 2018.
25. Sabt, M., Achemlal, M., Bouabdallah, A.: Trusted execution environment: what it is, and what it is not. In Trustcom 2015.
26. Savvides, S., Khandelwal, D., Eugster, P.: Efficient confidentiality-preserving data analytics over symmetrically encrypted datasets. In PVLDB 2020.
27. Sun, Y., Wang, S., Li, H., Li, F.: Building enclave-native storage engines for practical encrypted databases. In PVLDB 2021.
28. Tu, S., Kaashoek, M.F., Madden, S., Zeldovich, N.: Processing analytical queries over encrypted data. In PVLDB 2013.
29. Vadim, T., Alexey, S., Alexey, K., Sebastian, D.: sysbench-tpcc. https://github.com/Percona-Lab/sysbench-tpcc/ (2018)
30. Wong, W.K., Kao, B., Cheung, D.W.L., Li, R., Yiu, S.M.: Secure query processing with data interoperability in a cloud database environment. In SIGMOD 2014.
31. Xia, S., Zhu, Z., Zhu, C., Zhao, J., Chard, K., Elmore, A.J., Foster, I.T., Franklin, M.J., Krishnan, S., Fernandez, R.C.: Data station: Delegated, trustworthy, and auditable computation to enable data-sharing consortia with a data escrow. In PVLDB 2022.
32. Zhu, J., Cheng, K., Liu, J., Guo, L.: Full encryption: An end to end encryption mechanism in gaussdb. In PVLDB 2021.