LETIndex: A Secure Learned Index with TEE

Shuting Cao
Tsinghua University
Beijing, China
cst23@mails.tsinghua.edu.cn

Zeping Niu
Tsinghua University
Beijing, China
niuzp20@mails.tsinghua.edu.cn

Guoliang Li
Tsinghua University
Beijing, China
liguoliang@tsinghua.edu.cn

ABSTRACT

Trusted execution environment (TEE) offers a promising approach to building encrypted databases, which keep data confidential for users. However, designing an efficient index for TEE databases remains a significant challenge. Due to the limited enclave memory and system call support in enclaves, traditional indexes incur massive context switches (including enclave entry and exiting), which cause performance regression. Existing approaches, such as introducing rich execution environment (REE) buffer pools or index parameter optimization, may not alleviate these problems effectively. To address these limitations, we propose LETIndex, an efficient learned dynamic index designed for TEE databases. LETIndex adopts LSM-structured Piecewise Geometric Model (PGM) indexes and an adaptive prefetch mechanism to support lookup, range queries, and updates with significantly reduced context switches and disk I/O overhead. Experimental results show that LETIndex achieves superior performance compared to existing approaches on the SOSD benchmark. We demonstrate LETIndex with two real scenarios, binary join and multi-tale join.

PVLDB Reference Format:
Shuting Cao, Zeping Niu, and Guoliang Li. LETIndex: A Secure Learned Index with TEE. PVLDB, 18(12): 5403 - 5406, 2025.
doi:10.14778/3750601.3750682

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/chiiips/LETIndex.

1 INTRODUCTION

Indexes are essential for speeding up database query execution. In recent years, with the advancement of trusted execution environments (TEEs) such as Intel SGX and AMD SEV, researchers have increasingly focused on building encrypted databases within secure and isolated enclaves. However, designing an index for TEE database is difficult due to some inherent characteristics of TEE. First, the size of enclave memory is limited. For the commonly used Intel SGX (software guard extensions), SGXv1 supports only 128 MB of memory space in the enclave. While SGXv2 offers some improvements, it remains constrained in cloud multi-tenant scenarios. Second, the overhead of switching between enclaves and the rich execution environment (REE) is also non-negligible, ranging from 10000 to 18000 cycle [8]. Such context switches are relatively frequent because the enclave is a part of the user process (in ring 3). When executing a system call (e.g., reading data from a file), it first exits the enclave using an OCall and then re-enters the enclave with an ECall after completing the system call. Existing designs of the TEE index typically adopt a straightforward adaptation of traditional indexes like B+tree, supplemented by optimizations such as multi-level buffer pools[6]. These traditional index based designs cannot effectively mitigate the two issues mentioned above, primarily due to the difficulty of maintaining a small index size.

In recent years, a new type of index, known as learned indexes, has been proposed and extensively studied. Learned indexes leverage simple regression models, such as piecewise linear models, to approximate the cumulative distribution function (CDF) of the data, significantly reducing the index size [7]. Introducing learned indexes into the design of TEE indexes is a promising solution, though many challenges remain.

Challenge 1. TEE-induced Amplification of "Last Mile" Search Penalty. Firstly, Given a query key, learned indexes predict the corresponding position in the sorted array of tuples. Almost all existing learned indexes introduce errors during the prediction process. Therefore, an additional "last mile" search [3, 7] within the error bound must be performed to correct the discrepancy. The "last mile" search is typically one of the most time-consuming parts of the entire search process, and this overhead is further amplified in a TEE setting. For the read of a single file block, TEE introduces two *REE-TEE* context switches, including some time-consuming security operations (e.g. integrity verification). Since a typical "last mile" search requires multiple such block accesses, the accumulated overhead makes it prohibitively expensive in TEE environments.

Challenge 2. Context Switch Overhead in Structure Updates. Secondly, When a learned index performs an update, it may trigger a structural modification operation (SMO), such as when a node is full. An SMO may trigger a series of node modifications and writebacks. In conventional learned indexes without TEE, these operations incur negligible overhead since the nodes typically reside in the memory buffer pool. However, in a TEE index setting, nodes may be placed in a REE buffer pool, forcing each SMO to trigger multiple context switches.

Challenge 3. Efficient Layout of Index Storage. Thirdly, it is crucial to maximize the efficient use of enclave memory when organizing the storage structure of learned indexes. Compared to typical disk indexing scenarios, the TEE index design needs to consider utilizing both TEE and REE memory. Therefore, when designing the storage layout of a learned index, it is essential to consider its placement in TEE memory, REE memory, and external storage to maximize efficiency, which presents a significant challenge.

To address the above challenges, we propose LETIndex, an efficient index designed for TEE databases, supporting bulk load, lower

Proceedings of the VLDB Endowment, Vol. 18, No. 12 ISSN 2150-8097.
doi:10.14778/3750601.3750682

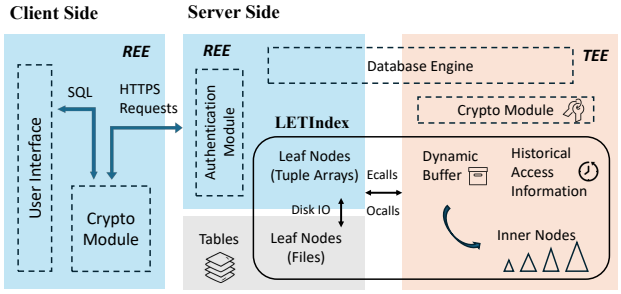

Figure 1: System Architecture.

bound, range queries, and updates. To mitigate the performance degradation caused by the "last mile" search, we utilize statistical information about the relative positions between the predicted and real positions of historical query keys, adaptively prefetch multiple blocks and reduce the overhead of context switches. For more efficient utilization of TEE memory, we adopt Piecewise Geometric Models (PGM) [2] to fit the data distribution of sorted keys. `LETIndex` consists of multiple PGMs with exponentially increasing sizes. The inner nodes of PGM are a compact set of linear models, achieving an extremely small size. By prioritizing the placement of all internal PGM nodes within the enclave and performing batch updates, we achieve a low amortized cost per update operation (including insertions, deletions, and modifications).

We evaluate our index with the SOSD [4] benchmark on both synthetic and real-world datasets. The experimental results show that `LETIndex` achieves up to 22x higher throughput compared to existing TEE indexes (B+tree-based) under write-only and balanced workloads. On read-only workloads, `LETIndex` has more than 15% throughput improvement. Our demonstration presents two scenarios: binary join and multi-table join. In our demonstration, users can select encrypted columns to create TEE indexes, utilize predefined SQL templates to generate queries containing binary join or multi-table join, and explore the visualized workflow of `LETIndex` in response to the queries.

2 LETINDEX OVERVIEW

Figure 1 illustrates the architecture of our system. The client possesses the encryption keys. Before queries are sent to the server side, the crypto module encrypts all keys corresponding to sensitive columns. Then the query is sent to the cloud server and executed by the database engine after authentication. When accessing data, if an index exists on the target column, our system utilizes `LETIndex` for efficient search. The encrypted keys in the query will be decrypted in TEE to keep confidentiality. `LETIndex` performs required operations like point search and returns a result. The server then returns query results containing both plaintext and ciphertext columns, while the client decrypts the encrypted parts locally.

2.1 Index Layout

When designing an index layout, the core principle is to maximize both data storage efficiency and buffer hit ratio in TEE memory. We utilize the Piecewise Geometric Model (PGM) [2] to approximate data distributions, which is well known for achieving small index sizes through optimal piecewise linear approximation of data. Similar to DPGM [2], we adopt an LSM [5] (Log-Structured Merge)-like organization of multiple PGMs to support update operations. `LETIndex` manages exponentially growing PGM indexes by retaining all internal nodes in TEE memory (achievable by tuning prediction error bound ϵ) while storing encrypted leaf nodes in REE memory or on disk. It also maintains a dynamic buffer to cache the most recently updated tuples in TEE.

2.2 Adaptive Prefetch

A unique challenge in TEE-based databases lies in the performance penalty during disk data reads (and even REE memory reads). Besides the data volume itself, the frequency of read operations significantly impacts performance due to the context-switching overhead between TEE and REE. While learned indexes excel in pure in-memory settings, their disk-based implementations suffer significant performance degradation for introducing prediction error. Given a prediction error bound ϵ, let p be the predicted position of a query key at the leaf level by the last used inner node. The precise position of the key lies within the range $[p - \epsilon, p + \epsilon]$. No matter how the error bound ϵ is adjusted (which is even not supported to be assigned in some kinds of learned index), misalignment between the search range and block size (typically 4KB) inevitably occurs. This may result in searches spanning two or even more disk blocks, leading to multiple context switches and more severe performance degradation compared to conventional disk scenarios. Moreover, the probability that the error bound covers blocks containing the query key also varies. For example, the error bound may only cover the last tuple in the first block, which means the first block is less likely to contain the query key. We propose an adaptive prefetch mechanism leveraging historical access information. `LETIndex` maintains metadata in TEE-resident memory that records the correct search direction for each range search starting at position p. This metadata is then used to determine whether to prefetch the left or right block adjacent to p. Our experiments show that the adaptive prefetch mechanism achieves approximately 20% improvement in index throughput (as figure 5 illustrates).

2.3 Operation Supports

`LETIndex` supports point queries, range queries, insertions, deletions, and updates. Upon receiving each query, the encrypted keys in the query are first decrypted within the TEE to facilitate subsequent query execution. For point queries, `LETIndex` first searches in the dynamic buffer within TEE memory, which stores the recently inserted data. If the key is not found in the dynamic buffer, `LETIndex` initiates a search from the smallest PGM. It first predicts the target tuple's location at the leaf node level based on the query key, then performs an exponential search or binary search within the predicted range. This process typically incurs 1-2 times of TEE-REE context switches. If the key is still not found, then `LETIndex` searches in the next PGM. Bloom filters can be employed to accelerate this search process. Range queries follow a similar execution pattern but require additional result merging. For dynamic operations, `LETIndex` takes an append-only strategy that converts all queries into insert operations.

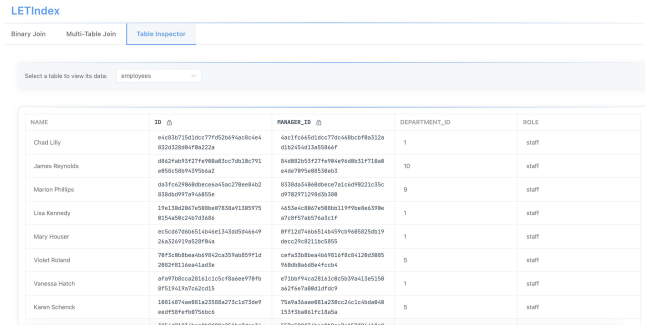

Figure 2: Page of Table Inspector.

3 DEMONSTRATION

In this section, we present the user interface of `LETIndex`, which includes three pages: table inspector, binary join, and multi-table join.

Table Inspector. Figure 2 is a screenshot of the table inspector page. This page allows users to inspect the tables involved in both binary and multi-table joins. Users can select a table from the dropdown menu to explore its structure and data. Taking the `employees` table as an example, after selecting the table and clicking the query button, the page displays the first 10 rows of data. To protect the company's organizational hierarchy, the `ID` and `MANAGER_ID` column is encrypted, and the table presents the encrypted version of these two columns. Since the client also holds a copy of the key, users can freely encrypt and decrypt data. Thus, clicking on the header of the encrypted column allows users to view the decrypted `MANAGER_ID`.

Scenario 1 - Binary Join As shown in the left subfigure of figure 3, users can select two different indexes to accelerate the execution of binary joins: `LETIndex` and a B+Tree-based TEE index. Users may also experiment by running the query without any index to observe the execution time, thereby gaining a clearer understanding of the performance improvement achieved through `LETIndex`. The query for the binary join is displayed in the left text box. Upon selecting an index and clicking the submit button, the execution begins (①). After the execution is completed, users can view the execution time and check the returned results at the bottom of the page (②). In addition, if users choose to use `LETIndex`, our system visually demonstrates how `LETIndex` processes queries for a specific `manager_id` to help users better understand the index structure (③).

Scenario 2 - Multi-Table Join Compared to binary joins, the scale of indexing has a greater impact in multi-table join scenarios due to the potential need to construct indexes for multiple tables. In a real-world cloud service scenario, the sharing of server resources among multiple tenants intensifies the pressure on enclave memory resources. For multi-table join, our system provides several SQL templates, all of which are multi-table join queries from TPC-H (as illustrated in the right subfigure of figure 3). Users can select an SQL template from the dropdown list, and then choose an index to use (①). The query execution time and results are displayed at the bottom of the page (②). Module ③ presents the query execution tree and time cost of each join operation.

4 EVALUATION

We evaluate the performance of `LETIndex` with three experiments. First, we run the SOSD benchmark to evaluate the performance of `LETIndex` on different workloads and datasets. Then we evaluate the performance of `LETIndex` on binary join and multi-table join, which are two common scenarios in real-world applications.

Experimental Setup. We conducted experiments on a server with a 2.70GHz Intel(R) Xeon(R) Platinum 8369B CPU, 32GB RAM, and 512GB SSD, running Ubuntu 22.04.5 LTS. Our implementation of `LETIndex` is based on Intel SGX 2.25 and Gramine 1.8[1].

SOSD Benchmark. For the SOSD benchmark, we used YCSB [1] to generate write-only, read-only, and balanced workloads, each with 150 million records. We compared `LETIndex` with two baselines, `B+Tree_ALL_Cache` and `B+Tree_Selected_Cache`. Both are B+Tree-based TEE indexes with separate buffer pools in TEE and REE. The first caches both internal and leaf nodes in both pools, while the second only caches internal nodes in the TEE buffer. Figure 4 shows that `LETIndex` outperforms both baselines across three different workloads. It is worth noting that LETIndex delivers exceptional performance under the write-only workload. The reason lies in the fact that B+Tree-based indexes may encounter node splits during updates. Node splits involve multiple levels of nodes, resulting in poor spatial locality, as data may be scattered across the TEE buffer pool, REE buffer pool, or even disk, leading to frequent context switches. In contrast, `LETIndex` uses an LSM-like structure to batch updates, enhancing spatial locality, reducing context switches, and amortizing update costs. Under the read-only workload, `LETIndex` outperforms `B+Tree_Selected_Cache` by approximately 15%, which is not easy for learned indexes in disk-based indexing scenarios. This is attributed to our adaptive prefetch strategy. To assess the impact of prefetching, we conducted an ablation on the read-only workload, comparing the throughput of `LETIndex` (prefetch) and `LETIndex_naive` (no prefetch). As shown in figure 5, the application of adaptive prefetch improves the index performance by approximately 20%.

Binary Join. For binary joins, we adopted a synthetic data table `employees` (150 million lines) and performed a join operation on the `ID` and `MANAGER_ID` columns to retrieve employee-manager relationships. We built indexes on the encrypted `ID` column to accelerate the join. `LETIndex` achieves a 17% reduction in execution time compared to `B+Tree_Selected_Cache` (514.34s vs. 619.74s).

Multi-Table Join. To evaluate the performance improvement of `LETIndex` for multi-table joins, we tested TPC-H Query 7 and compared the join time with `LETIndex` and `B+Tree_Cache` (same as `B+Tree_Selected_Cache`). The scale factor was set to 0.1. The comparison of join times across different tables is shown in Table 1. It can be observed that `LETIndex` provides greater optimization in join time compared to `B+Tree_Cache`.

ACKNOWLEDGMENTS

This paper was supported by National Key R&D Program of China (2023YFB4503600), NSF of China (62525202, 62232009), Shenzhen Project (CJGJZD20230724093403007), Zhongguancun Lab, and Beijing National Research Center for Information Science and Technology (BNRist). Guoliang Li is the corresponding author.

[1] https://github.com/gramineproject/gramine

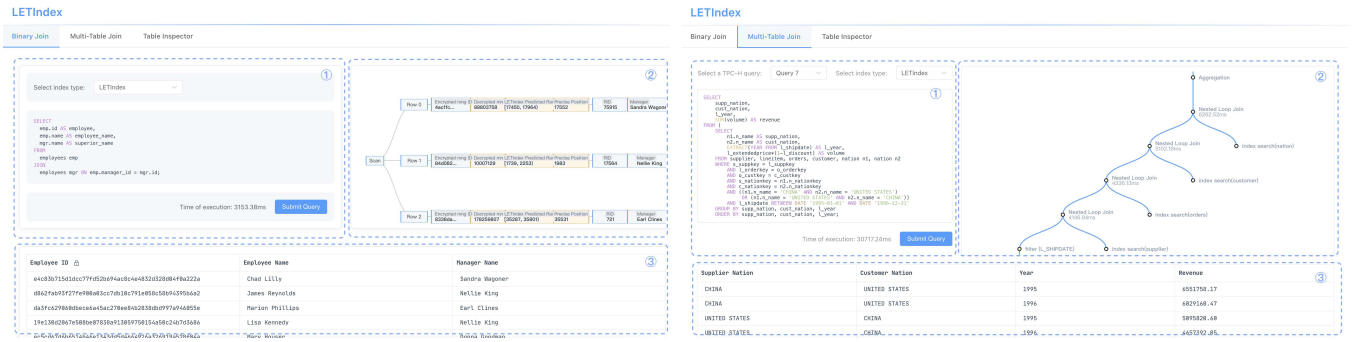

Figure 3: Screenshots of binary join page (the left subfigure) and multi-table join page (the right subfigure).

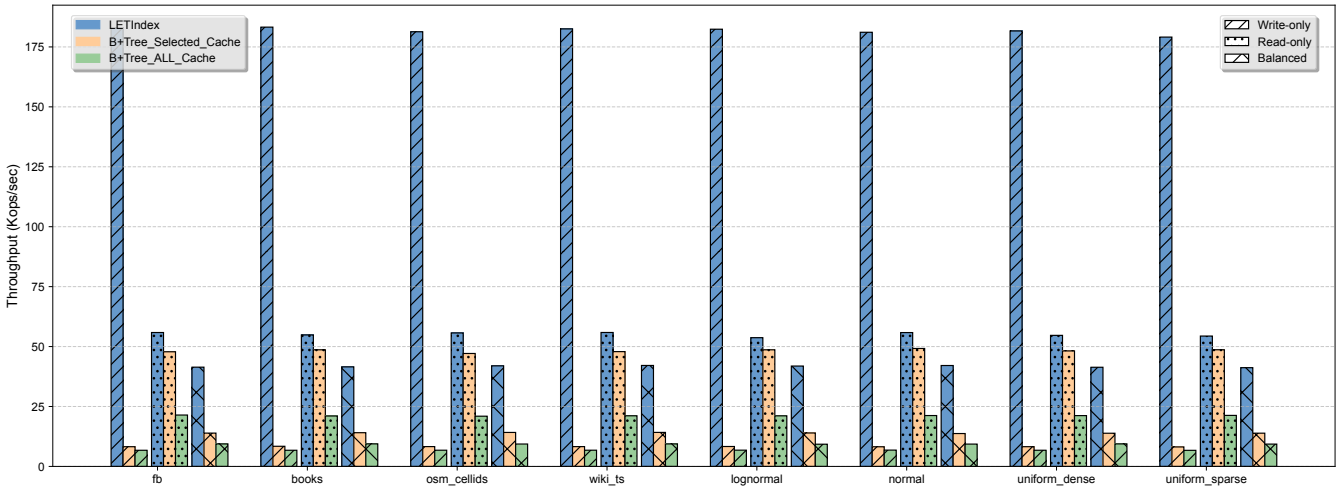

Figure 4: Throughput of `LETIndex` against two baselines on the SOSD benchmark.

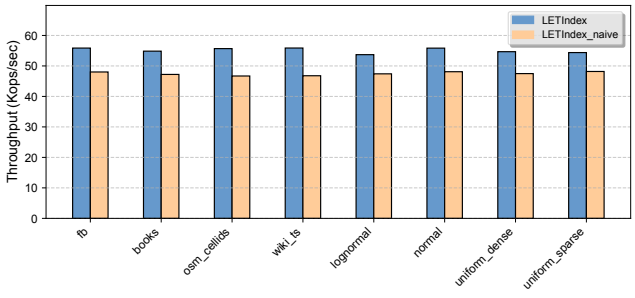

Figure 5: Ablation study of adaptive prefetch.

Table 1: Time Cost of Different Joins

Join (s)	`LETIndex`	`B+Tree_Cache`
lineitem ⋈ *supplier*	4.145	5.578
lineitem ⋈ *orders*	4.341	5.661
orders ⋈ *customer*	5.128	5.593
nation ⋈ *customer* ⋈ *supplier*	6.265	9.387

REFERENCES

[1] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB *(SoCC '10)*. 143–154.

[2] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proc. VLDB Endow.* 13, 8 (April 2020), 1162–1175.

[3] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures *(SIGMOD '18)*. 489–504.

[4] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking Learned Indexes. *Proc. VLDB Endow.* 14, 1 (2020), 1–13.

[5] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Inf.* 33, 4 (June 1996), 351–385. https://doi.org/10.1007/s002360050048

[6] Yuanyuan Sun, Sheng Wang, Huorong Li, and Feifei Li. 2021. Building enclave-native storage engines for practical encrypted databases. *Proc. VLDB Endow.* 14, 6 (2021), 1019–1032.

[7] Zhaoyan Sun, Xuanhe Zhou, and Guoliang Li. 2023. Learned Index: A Comprehensive Experimental Evaluation. *Proc. VLDB Endow.* 16, 8 (April 2023), 1992–2004.

[8] Ofir Weisse, Valeria Bertacco, and Todd Austin. 2017. Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves. *SIGARCH Comput. Archit. News* 45, 2 (June 2017), 81–93.