A Workload-Aware Encrypted Index for Efficient Privacy-Preserving Range Queries

Dong Wang
dong.wang@chd.edu.cn
Chang'an University

Ningning Cui*
willber1988@163.com
Chang'an University

Jianxin Li
jianxin.li@ecu.edu.au
Edith Cowan University

Jianzhong Qi
jianzhong.qi@unimelb.edu.au
The University of Melbourne

Jianliang Xu
xujl@comp.hkbu.edu.hk
Hong Kong Baptist University

Hui Lu
luhui@gzhu.edu.cn
Guangzhou University

ABSTRACT

Recent advances in workload-aware indexes have attracted growing attention for their ability to optimize index efficiency by learning query distributions. However, these architectures remain fundamentally incompatible with sensitive data scenarios that require encrypted index storage and privacy-preserving queries. Meanwhile, existing privacy-preserving solutions make it difficult for encrypted indexes to be workload-aware due to the complexity of cryptographic protocols, which prevents accurate cost estimation for a given workload. To address these limitations, this paper studies the workload-aware encrypted index for efficient privacy-preserving range queries. We propose P^{3}RQ-Bitmap, a XOR-encrypted bitmap index powered by a lightweight Pseudo Random Function (PRF)-based comparison protocol. This index supports efficient privacy-preserving range queries while being workload-aware. Building upon this, we further propose P^{3}RQ-WBTree, a workload-aware encrypted tree index that optimizes query efficiency through adaptive data partitioning guided by a gradient descent-optimized cost model. The index comes with buffer and reconstruct strategies to support dual updates for both data and workload. Extensive theoretical analysis and experiments demonstrate that P^{3}RQ-WBTree achieves at least 83× faster query performance compared to SOTA schemes.

PVLDB Reference Format:
Dong Wang, Ningning Cui, Jianxin Li, Jianzhong Qi, Jianliang Xu, and Hui Lu. A Workload-Aware Encrypted Index for Efficient Privacy-Preserving Range Queries. PVLDB, 19(4): 740 - 752, 2025.
doi:10.14778/3785297.3785313

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/CHAN-dong/WBTree.git.

1 INTRODUCTION

Modern data-driven applications increasingly rely on efficient range queries to support data analytics, location-based services, and real-time decision-making [4, 16, 31]. Recently, inspired by the seminal work [18], a new class of index structures known as workload-aware indexes has attracted significant attention. Unlike traditional indexes that only consider data distribution, these structures are capable of simultaneously learning query distribution to optimize index efficiency. For instance, workload-aware designs have been applied to tree construction optimization [12, 21, 33], space-filling curve parameter learning [13, 29], and space partitioning strategies [11, 27, 28] for range queries, achieving at least an 8× improvement in query efficiency compared to traditional data-aware indexes (e.g., R-tree, KD-tree). However, these methods inherently require direct access to plaintext data, making them unsuitable for privacy-sensitive domains such as healthcare, finance, and IoT, where queries must be executed over encrypted data [17, 32, 35].

With the exponential growth of sensitive data, there is an urgent demand for encrypted data storage and privacy-preserving query mechanisms [1, 5–8, 20, 22, 23, 25, 34, 42, 46]. However, the adoption of privacy-preserving range queries remains limited due to their significantly lower efficiency compared to plaintext counterparts. This inefficiency arises not only from the computational cost of cryptographic operations but more fundamentally from the difficulty of adapting advanced plaintext index techniques (i.e., workload-aware indexes) to encrypted environments. Table 1 summarizes key technological advancements in this field.

Table 1: Summary of existing schemes vs. ours

Scheme	Encrypt	Security	Index	Workload	Update
PORE [6]	ORE	POPF-CCA	Linear Scan	✗	✗
BlockOPE [5]	OPE	IND-OCPA	Binary-tree	✗	✗
PPTR [46]	RMM	IND-CPA	Linear Scan	✗	✗
PSDQ [25]	ASPE	IND-CPA	R-tree	✗	✗
Miao [42]	HE	IND-CPA	Radix-tree	✗	Data
GRSRT [23]	HE	IND-CPA	Radix-tree	✗	Data
PHRQ [34]	HE	IND-CPA	B+-tree	✗	✗
PSRQ [24]	HVE	IND-CPA	Bloom Filter	✗	✗
SKSE [40]	HVE	IND-CPA	Quad-tree	✗	✗
VSRQM [8]	HMAC	IND-CPA	KD-tree	✗	✗
ServeDB [44]	HMAC	IND-CPA	Binary-tree	✗	✗
[37, 39, 45]	LDP	Static Attacks	Hierarchical-tree	✗	✗
[10, 19, 43]	✗	✗	Learned-tree	✗	Data
[11, 13, 28]	✗	✗	Learned-tree	✓	✗
[21, 29, 33]	✗	✗	Learned-tree	✓	Dual
P^{3}RQ-Linear	PRF	IND-CPA	Linear Scan	✗	Data
P^{3}RQ-Bitmap	PRF	IND-CPA	Bitmap	✗	Data
P^{3}RQ-WBTree	**PRF**	**IND-CPA**	**Learned-tree**	**✓**	**Dual**

Security Strictness: IND-CPA (Indistinguishability under Chosen Plaintext Attack) > IND-OCPA (Indistinguishability under Ordered Chosen Plaintext Attack) > POPF-CCA (Pseudorandom Order-Preserving Function under Chosen Ciphertext Attack). **Notes:** '✓' means the approach meets the condition; '✗' means it breaks the condition; "Data" means supporting only data updates; "Dual" means supporting both data and workload updates.

*Corresponding author.

Proceedings of the VLDB Endowment, Vol. 19, No. 4 ISSN 2150-8097.
doi:10.14778/3785297.3785313

Early privacy-preserving range queries schemes, such as Order-Preserving/Revealing Encryption (OPE/ORE) [1, 5, 6, 20], trade security for fast range queries, rendering them vulnerable to inference attacks [15]. Subsequent approaches, including Homomorphic Encryption (HE) [22, 23, 34, 42], Asymmetric Scalar Product-Preserving Encryption (ASPE) [7, 25], Hidden Vector Encryption (HVE) [24, 40, 41], and Randomizable Matrix Multiplication (RMM) [46], strengthen security guarantees but incur prohibitive computational overhead. Although hash-based methods (e.g., HMAC) combined with Bloom Filters (BF) allow for high-speed range queries, they also introduce false positives in the query results [8, 44]. In parallel, unlike encryption-based schemes, Local Differential Privacy (LDP) methods resist statistical inference attacks by injecting noise, but this usually comes at the cost of reduced accuracy [37, 39, 45]. In addition, most privacy-preserving range query schemes can construct tree-based indexes to enable sub-linear queries, such as implementations combining R-tree [36], KD-tree [8], Quad-tree [41], B+-tree [34], Binary-tree [5, 44], and Radix-tree [23, 42].

Unfortunately, the above approaches fail to exploit workload-aware optimization opportunities. On one hand, existing encrypted schemes cannot be directly applied to workload-aware indexing through a "build first, then encrypt" approach. This is not only because certain plaintext indexes, such as Learned Indexes in [13, 28], cannot be directly encrypted with hidden ordering, but more importantly, because workload-aware methods rely on plaintext query cost models to guide index construction, which fundamentally break down once queries are encrypted. On the other hand, encryption protocols are inherently difficult to make workload-aware, since their cryptographic operations (e.g., homomorphic computations and ciphertext expansion) introduce non-linear and data-independent costs. This makes it challenging to build accurate cost models for guiding partitioning, unlike in plaintext settings, where query latency can be directly estimated from data distribution and access selectivity. To bridge this critical gap, in this paper, we take the first step toward workload-aware encrypted index for efficient privacy-preserving range queries. We identify two main challenges:

(i) How to design a lightweight workload-aware and privacy-preserving range queries scheme? To address this, we first propose a privacy-preserving data comparison protocol based on the lightweight Pseudo Random Function (PRF). Then, we combine this idea with an XOR-encrypt bitmap index to propose the P^{3}RQ-Bitmap, which is a novel index structure that supports efficient privacy-preserving range queries while enabling accurate estimation of encrypted query costs. *(ii) How can the workload-aware encrypted index be constructed using the above scheme to improve query performance?* We further propose P^{3}RQ-WBTree, a novel tree structure that leverages P^{3}RQ-Bitmap as nodes and incorporates workload information to enhance query efficiency. We formulate a cost model based on the workload to quantify storage and query overheads for each node. Guided by this cost model, we develop a greedy strategy to recursively split nodes and identify the minimal-cost tree structure. This process integrates learning linear models with gradient descent optimization to efficiently determine optimal split borders. Additionally, we integrate buffer and reconstruct strategies into P^{3}RQ-WBTree to support dual updates on both data and workload.

In summary, we make the following contributions:

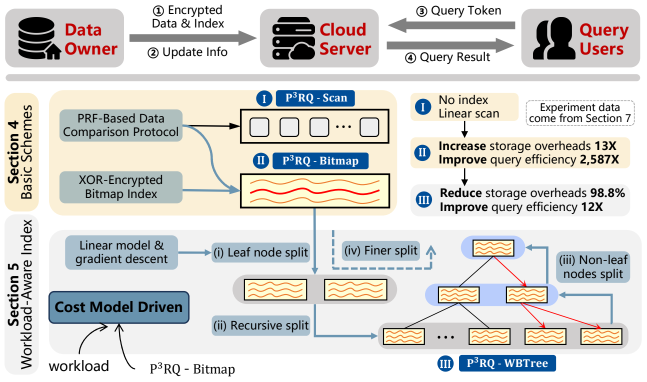

Data Owner
① Encrypted Data & Index
② Update Info
Cloud Server
③ Query Token
④ Query Result
Query Users
Section 4 Basic Schemes
PRF-Based Data Comparison Protocol
XOR-Encrypted Bitmap Index
I P³RQ - Scan
II P³RQ - Bitmap
I No index Linear scan
Experiment data come from Section 7
II Increase storage overheads 13X Improve query efficiency 2,587X
III Reduce storage overheads 98.8% Improve query efficiency 12X
Section 5 Workload-Aware Index
Linear model & gradient descent
(i) Leaf node split
(iv) Finer split
(iii) Non-leaf nodes split
Cost Model Driven
(ii) Recursive split
workload
P³RQ - Bitmap
III P³RQ - WBTree

Figure 1: System overview. There are three progressively optimized schemes: P^{3}RQ-Linear, P^{3}RQ-Bitmap, and the workload-aware index P^{3}RQ-WBTree.

- To the best of our knowledge, this is the first work to study workload-aware encrypted indexes for efficient privacy-preserving range queries.
- We propose a lightweight PRF-based privacy-preserving comparison protocol, and an XOR-encrypted bitmap index P^{3}RQ-Bitmap for efficient privacy-preserving range queries.
- We propose a novel workload-aware encrypted index P^{3}RQ-WBTree to further improve query and storage efficiency. Additionally, this index supports efficient data and workload updates.
- We conduct theoretical analysis and extensive experiments to demonstrate that P^{3}RQ-WBTree outperforms SOTA approaches, achieving up to 83× speedup in query efficiency.

2 PROBLEM FORMULATION

As shown in Figure 1, our query system consists of three entities: (i) Data Owner, (ii) Cloud Server, and (iii) Query Users. The specific query process is as follows.

- **Data Owner (DO).** The DO holds a dataset $\mathcal{D}$ along with the corresponding query workload $\mathcal{W}$. The DO can encrypt the dataset $\mathcal{D}$ into $\mathcal{D}^*$ and construct the encrypted index $\mathcal{I}^*$ for $\mathcal{D}^*$ with the assistance of $\mathcal{W}$. Then, the DO sends $\mathcal{D}^*$ and $\mathcal{I}^*$ to cloud storage (Step ①). Subsequently, the DO can also update the index based on new data or workload changes (Step ②).
- **Query users (QU).** The QU can initiate a query request $\mathcal{Q}$. It encrypts the query to generate a token $\mathcal{TD}$ and sends it to the cloud server for query processing (Step ③).
- **Cloud Server (CS).** The CS is responsible for storing the outsourced data and processing queries. Upon receiving the token $\mathcal{TD}$, it processes the privacy-preserving range query based on the encrypted index $\mathcal{I}^*$. After this, CS gets the encrypted result objects $\mathcal{R}$ and sends them to the QU (Step ④).

The threat model of our schemes is consistent with most prior work. The DO and QU are assumed to be fully trusted. The CS, however, is modeled as 'honest-but-curious' [24, 34, 40–42, 46]. It faithfully executes the designated protocol but may attempt to infer meaningful information about the data objects or the QU's query patterns. The primary security objectives of our schemes are:

- **Data Privacy.** All data in the original database $\mathcal{D}$ should be protected from the adversary.

- **Query Privacy.** The content of the range query Q should be protected from the adversary.

Based on the setup above, we study the problem of efficient privacy-preserving range queries. The range data can reside in arbitrary dimensions, and for simplicity of description without loss of generality, we assume it is two-dimensional. In contrast to existing methods, our solution enables query workload-aware optimization, which we formally define as follows:

Definition 1 (Workload-Aware, Privacy-Preserving Range Queries). Given a dataset $\mathcal{D} = \{o_1, o_2, ..., o_n\}$ where each object o_i consists of a geo-coordinate tuple $\{m_x, m_y\}$. Along with this, there is a query workload $\mathcal{W}$ consisting of a set of queries $\{Q\}$, which reflects QU's query distribution. The scheme proceeds as follows: Dataset $\mathcal{D}$ is encrypted into $\mathcal{D}^* = \{o_1^*, o_2^*, \cdots, o_n^*\}$, and an encrypted index $\mathcal{I}^*$ is constructed for $\mathcal{D}^*$ using $\mathcal{W}$. Given a range query $Q = \{\{q_x^l, q_y^l\}, \{q_x^r, q_y^r\}\}$ representing rectangular bounds, a search token $\mathcal{TD}$ is generated based on Q, which can be evaluated against the encrypted index $\mathcal{I}^*$ to retrieve the corresponding encrypted results $\mathcal{R}$. For each result $o^* \in \mathcal{R}$ corresponding $o = \{m_x, m_y\}$, holds $(q_x^l \leq m_x \leq q_x^r) \wedge (q_y^l \leq m_y \leq q_y^r)$.

3 OVERVIEW

As shown in Figure 1, we first propose two basic indexing and query schemes P^{3}RQ-Linear and P^{3}RQ-Bitmap, which are prepared for the workload-aware encrypted index scheme P^{3}RQ-WBTree.

Basic Schemes. To enable querying over encrypted data while avoiding the use of complex cryptographic primitives, we first propose a privacy-preserving data comparison protocol based on the lightweight Pseudo Random Function (PRF) (*See Section 4.1*). However, when directly applying this protocol to enable privacy-preserving range queries (the P^{3}RQ-Linear scheme), it results in an $O(\theta \cdot n)$ PRF computations, where n denotes the dataset size and θ denotes the data binary length. To reduce this overhead, we integrate the protocol with an XOR-encrypted bitmap index and propose the P^{3}RQ-Bitmap scheme, which lowers the PRF computation cost to $O(\theta)$ (*See Section 4.2*). Nevertheless, this optimization introduces $O(n)$ bitwise query computations, and the storage overhead increases super-linearly with the dataset size n.

Workload-Aware Index Scheme. We further exploit using P^{3}RQ-Bitmap as tree nodes to construct a tree structure for reducing both query and storage complexity. While this tree-based approach successfully reduces the bitwise computation complexity from $O(n)$ to $O(\log n)$, it simultaneously increases the PRF computation complexity from $O(\theta)$ to $O(\theta \cdot \log n)$. Consequently, conventional tree indexes (e.g., R-trees) fail to balance PRF and bitwise computations, ultimately resulting in degraded query efficiency.

We aim to make the tree structure workload-aware so that it can adaptively balance PRF and bitwise computations. To achieve this, we propose the P^{3}RQ-WBTree scheme. First, we formulate a cost model based on the workload to evaluate the query and storage costs associated with each tree node (*See Section 5.2*). Then, we use this cost model to drive P^{3}RQ-WBTree construction for tree cost minimization (*See Section 5.3*). This mainly consists of four steps. We store all objects in a node initially, then find an optimal split border to partition the node such that the post-split cost is minimized. During this process, we use linear models and gradient descent to quickly locate the split border (Step (i)). This partitioning strategy is applied recursively until the node cost can no longer be reduced (Step (ii)). Additionally, we construct a P^{3}RQ-Bitmap as a non-leaf node to index the leaf nodes, which can also be partitioned using the same strategy (Step (iii)). After tree formulation, we further split all nodes more finely in a bottom-up manner to ensure no further splits are possible (Step (iv)). During queries, the top-down traversal enables fast node filtering, achieving sub-linear query efficiency (*See Section 5.4*). Furthermore, we implement a buffer and reconstruct strategies to accommodate updates in both data distributions and query workloads (*See Section 5.5*).

Table 2: The summary of notations

Notation	Description
$\mathcal{D}/\mathcal{D}^*, o/o^*$	dataset/encrypted dataset, object/encrypted object
$\mathcal{W}, \mathcal{TD}, \mathcal{R}$	workload, query token, and query result
$o = \{m_x, m_y\}$	geo-coordinate tuple
$Q = \{\{q_x^l, q_y^l\}, \{q_x^r, y_y^r\}\}$	geo-coordinate query
$k, r, \mathsf{F}(k/r, \cdot)$	secret key, random value, and PRF function
$v_1 v_2 \cdots v_\theta$	binary representation of message data
$e/u, et/ut$	encrypted bit strings in Definition 3
w_q/w_s	query/storage cost model weight
p_n, p_q, p_s	parameters used for cost model computing
$\mathcal{N}, \mathcal{M}, b, c$	tree node, linear model, space split border, node cost
η, ζ	parameters used for tree index update

4 BASIC SCHEME

We start with a novel PRF-based privacy-preserving data comparison protocol. We then incorporate the protocol with an XOR-encrypted bitmap index to propose P^{3}RQ-Bitmap, enabling efficient privacy-preserving range queries.

4.1 PRF-Based Data Comparison

Chenette et al. [6] first proposed a PRF-based ORE scheme, denoted as PRF-ORE, which encrypts each binary bit of data using a lightweight PRF primitive, enabling efficient ciphertext comparisons. However, this approach suffers from critical security vulnerabilities as it leaks data ordering information [15]. To address this limitation, we enhance their scheme to achieve IND-CPA security [9], which is a rigorous security standard. Our solution guarantees that the ciphertexts et_1, et_2 of messages m_1, m_2 cannot be directly compared. Only the encrypted token ut_q of a query message m_q can be used to compare against a ciphertext et. The PRF and the enhanced protocol are formally defined as follows:

Definition 2 (Pseudo Random Function, PRF). For a randomly chosen key $k \in \{0,1\}^\lambda$, a function family $\mathsf{F} : \{0,1\}^\lambda \times \{0,1\}^x \to \{0,1\}^y$ is called a PRF if, for any efficient adversary, the function $\mathsf{F}(k, \cdot)$ is computationally indistinguishable from a truly random function $\mathsf{G} : \{0,1\}^x \to \{0,1\}^y$.

Definition 3 (PRF-Based Privacy-Preserving Data Comparison). The data comparison protocol is constituted of four polynomial algorithms as follows:

- $\mathsf{Setup}(1^\lambda) \to k$. The setup algorithm chooses a secret key $k \leftarrow \{0,1\}^\lambda$ for PRF $\mathsf{F}(k, \cdot)$.
- $\mathsf{Encrypt}(k, m) \to et$. The algorithm first chooses a random value $r \leftarrow \{0,1\}^\lambda$, and represents the message m in binary as $v_1 \cdots v_\theta$ with each $v_i \in \{0,1\}$. For each position i with $v_i = 0$, it constructs a '0-string' $(i, v_1 v_2 \cdots v_i | 0^{\theta - i})$, where "|" denotes

string concatenation and $0^{\theta-i}$ denotes a sequence of θ-i zeros. The algorithm then computes u_i and e_i using the PRF:

$$u_i = \mathsf{F}(k, (i, v_1v_2 \cdots v_i|0^{\theta-i})), e_i = \mathsf{F}(r, u_i). \tag{1}$$

Finally, the algorithm outputs the ciphertext $et = (r, e_1, e_2, ..., e_\theta)$.

- $\mathsf{TokenGen}(k, m_q) \rightarrow ut$. The algorithm represents the query message m_q in binary as $v_1 \cdots v_\theta$. For each position i with $v_i = 1$, it constructs a '1-string' $(i, v_1v_2 \cdots (v_i - 1)|0^{\theta-i})$ and then computes u_i using the PRF:

$$u_i = \mathsf{F}(k, (i, v_1v_2 \cdots (v_i - 1)|0^{\theta-i})). \tag{2}$$

Finally, the algorithm outputs the token $ut = (u_1, u_2, ..., u_\theta)$.

- $\mathsf{Compare}(ut, et) \rightarrow \{1, 0\}$. The compare algorithm parses $et = (r, e_1, e_2, ..., e_\theta)$, $ut = (u_1, u_2, ..., u_\theta)$. If there exists u_j and $\mathsf{PRF}(r, u_j) \in et$, it means $ut > et$, and the algorithm outputs 1; otherwise, $ut \leq et$ and the algorithm outputs 0.

Example 1. *Given $\theta = 3$, $m_1 = (110)_2$, $m_2 = (011)_2$, and $m_q = (100)_2$. The protocol works as follows.* ❶ ***Encryption****. For m_1, the algorithm generates a random r_1. Since $v_3 = 0$, the corresponding '0-string' is $(3, 110|)$. It then computes $e_1 = F(r_1, F(k, (3, 110)))$ and outputs the ciphertext $et_1 = (r_1, e_1)$. For m_2, the algorithm generates a random r_2. Since $v_1 = 0$, so the '0-string' is $(1, 0|00)$. It computes $e_2 = F(r_2, F(k, (1, 000)))$ and outputs the ciphertext $et_2 = (r_2, e_2)$.* ❷ ***Token generation****. Since $v_1 = 1$, the algorithm constructs the '1-string' $(1, 0|00)$ and computes $u_1 = F(k, (1, 000))$.* ❸ ***Comparison****. The algorithm checks $et_1 = (r_1, e_1)$ and observes that $F(r_1, u_1) \neq e_1$, which indicates that $ut_q \leq et_1$. It then checks $et_2 = (r_2, e_2)$ and observes that $F(r_2, u_1) = e_2$, which indicates that $ut_q > et_2$.*

Extending to privacy-preserving range queries. Based on Definition 3, given a query $[q^l, q^r]$, we can determine in the ciphertext domain whether a message m intersects with $[q^l, q^r]$.

$$\begin{aligned} m \in [q^l, q^r] &\Leftrightarrow q^l \leq m \leq q^r \Leftrightarrow \neg(q^l > m) \wedge (q^r + 1) > m \\ &\Leftrightarrow \neg\mathsf{Compare}(ut(q^l), et(m)) \wedge \mathsf{Compare}(ut(q^r + 1), et(m)) \end{aligned} \tag{3}$$

Furthermore, if the encrypted object is a range $[m^l, m^r]$ instead of a message m, we can also determine in the ciphertext domain whether the range $[m^l, m^r]$ intersects with $[q^l, q^r]$.

$$\begin{aligned} [m^l, m^r] \cap [q^l, q^r] &\Leftrightarrow q^l \leq m^r \wedge q^r \geq m^l \Leftrightarrow \neg(q^l > m^r) \wedge q^r + 1 > m^l \\ &\Leftrightarrow \neg\mathsf{Compare}(ut(q^l), et(m^r)) \wedge \mathsf{Compare}(ut(q^r + 1), et(m^l)) \end{aligned} \tag{4}$$

In above, the $et = \mathsf{Encrypt}(k, \cdot)$ and $ut = \mathsf{TokenGen}(k, \cdot)$ are simplified to $et(\cdot)$ and $ut(\cdot)$, respectively. With these properties, PRF-based privacy-preserving range queries (P^{3}RQ-Linear) can be easily implemented. Each object's location is encrypted via Encrypt. Upon query, a token is generated using TokenGen, and objects are linearly scanned with Compare (Equation (3)) to identify matches.

Analysis. In the above scheme, the main computational overhead during queries comes from PRF computations. Querying a single object requires $O(\theta)$ PRF computations, meaning that querying an entire dataset with n objects requires $O(\theta \cdot n)$ PRF computations.

4.2 P^{3}RQ-Bitmap

Considering the high computational cost of P^{3}RQ-Linear, we propose a bitmap-based index, P^{3}RQ-Bitmap, which indexes all PRF values in a lightweight XOR-encrypted bitmap. During queries, XOR operations replace individual PRF computations, allowing batch comparisons and significantly accelerating the query.

Algorithm 1: P^{3}RQ-Bitmap.Construct($k, \mathcal{D}, r$)

```
Input: Dataset $\mathcal{D}$, secret key $k$ and a random value $r$
Output: Encrypted bitmap index $\mathcal{I}^* \leftarrow \mathcal{B}$
initialize two bitmap $\mathcal{PB}$ and $\mathcal{B}$;
for $j$-th object $o = \{\{m^l_x, m^l_y\}, \{m^r_x, m^r_y\}\}$ in $\mathcal{D}$ do
    // if o is a point, then $m^l_x = m^r_x, m^l_y = m^r_y$
    for each $m^{pos}_{dim}$ in $\{m^l_x, m^l_y, m^r_x, m^r_y\}$ do
        get binary representation $v_1 \cdots v_\theta$ of $m^{pos}_{dim}$;
        for each $v_i = 0$ do
            $pc \leftarrow (i, v_1v_2 \cdots v_i|0^{\theta-i})|r|dim|pos$;
            $\mathcal{PB}[pc][j] \leftarrow 1$;
for each $pc$ in $\mathcal{PB}$ do
    parse $pc \rightarrow (i, v_1v_2 \cdots v_i|0^{\theta-i})|r|dim|pos$;
    $\alpha \leftarrow \mathsf{F}(k|1, (i, v_1v_2 \cdots v_i|0^{\theta-i}))$;
    $\beta \leftarrow \mathsf{F}(k|2, (i, v_1v_2 \cdots v_i|0^{\theta-i}))$;
    $\alpha' \leftarrow \mathsf{F}(r|dim|pos, \alpha)$;
    $\beta' \leftarrow \mathsf{F}(r|dim|pos, \beta)$;
    $\mathcal{B}[\alpha'] \leftarrow \mathcal{PB}[pc]$.XOR(Bitset.Valueof($\beta'$));
return $\mathcal{B}$;
```

Algorithm 2: P^{3}RQ-Bitmap.TokenGen(k, Q)

```
Input: Secret key $k$ and query $Q$
Output: Query token $\mathcal{TD}$
initialize a query token $\mathcal{TD}$;
parse $Q \rightarrow \{\{q^l_x, q^l_y\}, \{q^r_x, q^r_y\}\}$;
for each $q^{pos}_{dim}$ in $\{q^l_x, q^l_y, (q^r_x + 1), (q^r_y + 1)\}$ do
    get binary representation $v_1 \cdots v_\theta$ of $q^{pos}_{dim}$;
    reverse the $pos$ and initialize set $ut^{pos}_{dim}$; // $x \rightarrow y$ or $y \rightarrow x$
    for each $v_i = 1$ do
        $\alpha \leftarrow \mathsf{F}(k|1, (i, v_1v_2 \cdots (v_i - 1)|0^{\theta-i}))$;
        $\beta \leftarrow \mathsf{F}(k|2, (i, v_1v_2 \cdots (v_i - 1)|0^{\theta-i}))$;
        $ut^{pos}_{dim}$.Add($\{\alpha, \beta\}$);
    $\mathcal{TD}$.Add($ut^{pos}_{dim}$);
return $\mathcal{TD}$;
```

Algorithm 3: P^{3}RQ-Bitmap.Query($\mathcal{TD}, \mathcal{B}, r$)

```
Input: Query token $\mathcal{TD}$, Bitmap index $\mathcal{B}$ and it's rand value $r$
Output: Query result $\mathcal{R}$
initialize a result set $\mathcal{R}$;
initialize a bitset $bs_1$ and fill with 1;
for each $ut^{pos}_{dim}$ in $\mathcal{TD} = \{ut^l_x, ut^l_y, ut^r_x, ut^r_y\}$ do
    initialize a bitset $bs_2$ and fill with 0;
    for each $\{\alpha, \beta\}$ in $ut^{pos}_{dim}$ do
        $\alpha' \leftarrow \mathsf{F}(r|dim|pos, \alpha)$;
        $\beta' \leftarrow \mathsf{F}(r|dim|pos, \beta)$;
        $bs\_dec \leftarrow \mathcal{B}[\alpha']$.XOR(Bitset.Valueof($\beta'$));
        $bs_2 \leftarrow bs_2$.OR($bs\_dec$);
    if $pos$ is $l$ then $bs_2 \leftarrow bs_2$.NOT();
    $bs_1 \leftarrow bs_1$.AND($bs_2$);
for each $bs_1[j]$ is 1 then add the $j$-th object $o^*_j$ to $\mathcal{R}$;
return $\mathcal{R}$;
```

Index construction. The P^{3}RQ-Bitmap construction algorithm is shown in Algorithm 1. It supports both range and point object encryption. Each object o is parsed into $\{\{m^l_x, m^l_y\}, \{m^r_x, m^r_y\}\}$; if $m^l_x = m^r_x, m^l_y = m^r_y$, o is a point object; otherwise a range object. The index construction consists of two main steps: ❶ We employ a plain bitmap $\mathcal{PB}$ to encode all packed string pc, where pc comprises the object's '0-string' $(i, v_1v_2 \cdots v_i|0^{\theta-i})$, dimensional dim, and positional attributes pos. The bitmap structure satisfies $\mathcal{PB}[pc][j] = 1$

if and only if the j-th object o_j contains pc (lines 2-7). ❷ Then, the $\mathcal{PB}$ is encrypted into a new bitmap $\mathcal{B}$ using a secret key k. Specifically, each pc is parsed and encrypted via PRF into α' and β'. The value $\mathcal{PB}[pc]$ is then XOR-encrypted with β' and stored at $\mathcal{B}[\alpha']$ (lines 9-15).

Query token generation. Algorithm 2 shows the token generation process for query $Q = \{\{q_x^l, q_y^l\}, \{q_x^r, q_y^r\}\}$. It should be noted that, as indicated by Equations (3) and (4), the positional attribute pos in the query message q_{dim}^{pos} must be reversed since the query compares the left message against the right object. In particular, if q_{dim}^{pos} has $pos = r$, then $q_{dim}^{pos} + 1$ must be encrypted instead (lines 4-5). Finally, each '1-string' $(i, v_1 v_2 \cdots (v_i - 1)|0^{\theta - i})$ are encrypted using secret key k to generate α and β, which are finally packed in $\mathcal{TD}$ (lines 6-10).

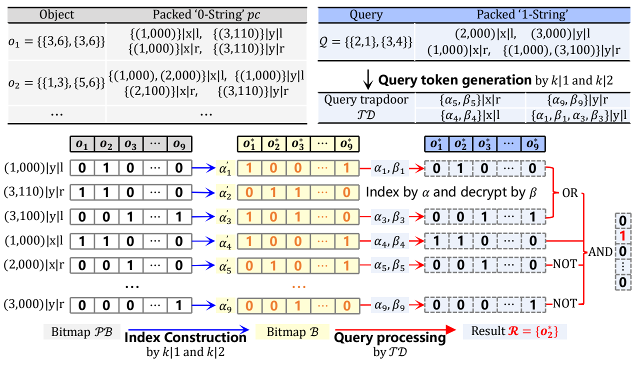
Query processing. With the token $\mathcal{TD}$, the query process is performed over the encrypted index $\mathcal{I}^* = \mathcal{B}$ as described in Algorithm 3. The procedure consists of three main steps: ❶ Each PRF value in $\mathcal{TD}$ is appended with its query dimension and position to compute α' and β' (lines 6-7). ❷ The value stored at $\mathcal{B}[\alpha']$ is XOR-decrypted using β' to obtain the bitset bs_dec (line 8). ❸ All bs_dec values decrypted from a token ut are combined using OR-bitwise, yielding bs_2. If ut corresponds to a left query message, bs_2 is inverted by NOT-bitwise. Then all bs_2 is aggregated via AND-bitwise to obtain bs_1 (lines 9-11). If $bs_1[j] = 1$, it indicates that the j-th object satisfies the query result (line 12).

Example 2. *Figure 2 illustrates the overall workflow of P^{3}RQ-Bitmap.* ❶ ***Index construction****. In the figure, o_1 and o_2 represent a point and a range object, respectively. The gray table shows each object's packed 0-binary string pc. Then a bitmap $\mathcal{PB}$ is built (e.g., $\mathcal{PB}[(3, 110)|y|r] = 110 \cdots 0$ indicates both o_1 and o_2 contain $(3, 110)|y|r$). Finally, $\mathcal{PB}$ is encrypted to obfuscate each bit, and the result is stored in the bitmap $\mathcal{B}$.* ❷ ***Token generation.*** *The blue table displays the packed 1-binary string of query Q, which is subsequently encrypted to generate the token $\mathcal{TD}$. Notably, the right messages $\{3, 4\}$ is replaced with $\{4, 5\}$ for encoding. And PRF values of right message 4 is packed with a left label (i.e., $\{\alpha_4, \beta_4\}|x|l$) add to $\mathcal{TD}$.* ❸ ***Query processing****. The red lines indicate the query process. Each token element is used in a PRF computation to index the corresponding bitset, which is then decrypted (e.g., α_1 indexes to α_1' and, using β_1, decrypts $100 \cdots 1$ to $010 \cdots 0$). Finally, bitwise OR, NOT, and AND operations on the decrypted bitsets yield the final query result (i.e., get $\mathcal{R} = \{o_2^*\}$ from bitset $010 \cdots 0$).*

Analysis. Compared to P^{3}RQ-Linear, P^{3}RQ-Bitmap reduces the PRF evaluations per query from $O(\theta \cdot n)$ to $O(\theta)$. However, the overall query time complexity remains linear in the dataset size n due to the $O(n)$ bitwise operations involved. In addition, a P^{3}RQ-Bitmap stores $O(n \log \theta)$ distinct PRF values, each occupying n bits, resulting in a total storage complexity of $O(n^2 \log \theta)$.

5 WORKLOAD-AWARE SCHEME

To reduce the $O(n)$ bitwise operations to sub-linear, an intuitive strategy is to recursively partition the dataset into a tree structure (e.g., R-tree, KD-tree), where each node organizes its objects using a P^{3}RQ-Bitmap. However, this tree structure also increases the PRF computation from O(θ) to $O(\theta \cdot \log n)$. While decreasing the number of objects per node reduces bitwise computation costs, this also leads to a taller tree structure, which increases the overall PRF



Object
Packed '0-String' pc
$o_1 = \{\{3,6\},\{3,6\}\}$
$\{(1,000)\}|x|l$, $\{(3,110)\}|y|l$
$\{(1,000)\}|x|r$, $\{(3,110)\}|y|r$
$o_2 = \{\{1,3\},\{5,6\}\}$
$\{(1,000),(2,000)\}|x|l$, $\{(1,000)\}|y|l$
$\{(2,100)\}|x|r$, $\{(3,110)\}|y|r$
Query
Packed '1-String'
$Q = \{\{2,1\},\{3,4\}\}$
$(2,000)|x|l$, $(3,000)|y|l$
$(1,000)|x|r$, $\{(1,000),(3,100)\}|y|r$
Query token generation by $k|1$ and $k|2$
Query trapdoor $\mathcal{TD}$
$\{\alpha_5,\beta_5\}|x|r$
$\{\alpha_9,\beta_9\}|y|r$
$\{\alpha_4,\beta_4\}|x|l$
$\{\alpha_1,\beta_1,\alpha_3,\beta_3\}|y|l$
$(1,000)|y|l$
$(3,110)|y|r$
$(3,100)|y|l$
$(1,000)|x|l$
$(2,000)|x|r$
$(3,000)|y|r$
Index by α and decrypt by β
OR
NOT
AND
Bitmap $\mathcal{PB}$
Index Construction by $k|1$ and $k|2$
Bitmap $\mathcal{B}$
Query processing by $\mathcal{TD}$
Result $\mathcal{R} = \{o_2^*\}$

Figure 2: Example of P^{3}RQ-Bitmap workflow, including the index construction of the P^{3}RQ-Bitmap $\mathcal{B}$ for objects o_1 and o_2, the query token generation for a query Q, and the query processing of the generated token $\mathcal{TD}$ in index $\mathcal{B}$.

computation overhead. Consequently, conventional trees may inadvertently degrade query efficiency, as they cannot control the tree structure to balance PRF and bitwise computations. In particular, our experimental results in Section 7 demonstrate that a standard KD-tree with P^{3}RQ-Bitmap will reduce query efficiency by at least 40×. In what follows, we present P^{3}RQ-WBTree, a workload-aware encrypted index that adaptively optimizes node layouts to balance PRF and bitwise computations.

5.1 P^{3}RQ-WBTree Structure

P^{3}RQ-WBTree is neither strictly binary nor height-balanced. Instead, each node adopts an adaptive fanout and children, determined by the distribution of queries and objects. It contains two types of nodes: leaf nodes and non-leaf nodes. Each node is equipped with a P^{3}RQ-Bitmap, which indexes either the objects stored in leaf nodes or the Minimum Bounding Rectangles (MBRs) of child nodes in non-leaf nodes.

Definition 4 (Leaf Node). The fields of the leaf node $\mathcal{N}$ of P^{3}RQ-WBTree are defined as:

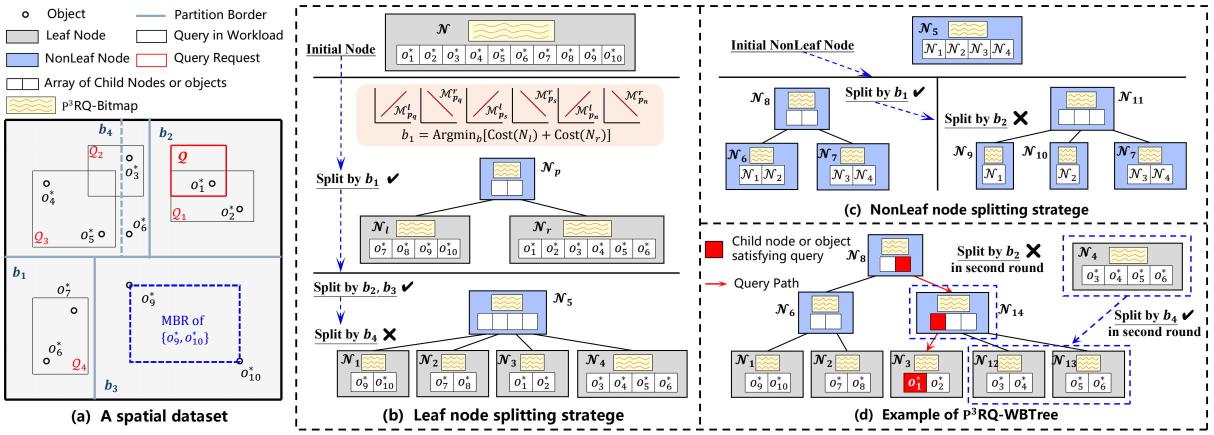
- $\mathcal{OA}$: An array storing the encrypted objects in $\mathcal{N}$, where $\mathcal{OA}[i]$ represents the i-th encrypted object;
- $r \leftarrow \{0, 1\}^\lambda$: A random value used to encrypt the Bitmap.
- $\mathcal{B} \leftarrow$ P^{3}RQ-Bitmap.Construct$(k, \mathcal{OA}, r)$: A P^{3}RQ-Bitmap constructed from the object array $\mathcal{OA}$.

Definition 5 (Non Leaf Node). The fields of the non-leaf node $\mathcal{N}$ of P^{3}RQ-WBTree are defined as:

- $\mathcal{CA}$: An array containing the child nodes of $\mathcal{N}$, where $\mathcal{CA}[i]$ represents the i-th child node;
- $r \leftarrow \{0, 1\}^\lambda$: A random value used to encrypt the Bitmap.
- $\mathcal{B} \leftarrow$ P^{3}RQ-Bitmap.Construct$(k, \mathcal{RA}, r)$: A P^{3}RQ-Bitmap constructed from a range array $\mathcal{RA}$, where $\mathcal{RA}[i]$ is the MBR of child node $\mathcal{CA}[i]$.

5.2 Node Cost Model

To evaluate the quality of a node, i.e., its query efficiency and storage cost, without fully constructing the node or executing queries, we introduce a cost model to accurately assess these aspects. Specifically, given a query workload $\mathcal{W}$, we formulate a cost model for



Object
Partition Border
Leaf Node
Query in Workload
NonLeaf Node
Query Request
Array of Child Nodes or objects
P³RQ-Bitmap
MBR of {o₉*, o₁₀*}
(a) A spatial dataset
Initial Node
b_1 = Argmin$_b$[Cost(N_l) + Cost(N_r)]
Split by b_1 ✔
Split by b_2, b_3 ✔
Split by b_4 ✘
(b) Leaf node splitting stratege
Initial NonLeaf Node
Split by b_1 ✔
Split by b_2 ✘
(c) NonLeaf node splitting stratege
Child node or object satisfying query
Query Path
Split by b_2 ✘ in second round
Split by b_4 ✔ in second round
(d) Example of P³RQ-WBTree

Figure 3: Example workflow of P^{3}RQ-WBTree. (a) depicts a spatial dataset with 10 objects and a query workload of 4 queries. (b) and (c) sequentially depict the P^{3}RQ-WBTree construction process for leaf nodes and non-leaf nodes, respectively. (d) depicts the finer splitting process and the final constructed index structure, together with the query processing for a query Q.

a P^{3}RQ-WBTree node $\mathcal{N}$ constructed over p_n objects (i.e., objects array size $|\mathcal{OA}| = p_n$ or child nodes size $|\mathcal{CA}| = p_n$) to measure the query cost and storage cost, as follows:

$$\text{Cost}(\mathcal{N}) = w_q \cdot \text{Query}(\mathcal{N}) + w_s \cdot \text{Storage}(\mathcal{N}) \quad (5)$$

Here, w_q and w_s are tunable parameters that control the relative weight of query and storage costs, respectively.

Query cost. The query cost of a node $\mathcal{N}$ refers to the time required to execute the workload $\mathcal{W} = \{Q\}$ on its P^{3}RQ-Bitmap. It consists of three components following the P^{3}RQ-Bitmap.Query algorithm: ❶ the time required to load the node from memory, ❷ the time of PRF computations for token elements, ❸ and the time of the bitset decryption and result aggregation via XOR, AND, OR, and NOT operations. The total query cost is given by:

$$\text{Query}(\mathcal{N}) = \text{T}_1 + p_q \cdot \text{T}_2 + p_n \cdot p_q \cdot \text{T}_3 \quad (6)$$

Here, p_q denotes the number of PRF values in query token (i.e., $\{\alpha, \beta\}$ in $\mathcal{TD}$) generated by P^{3}RQ-Bitmap.TokenGen(k, $\{Q\}$). The constants T$_1$, T$_2$, and T$_3$ represent the time to load a node from memory, the time for a PRF computation, and the time for one bitwise operation, respectively.

Storage cost. The storage cost $\mathcal{N}$ of a node contains three parts: ❶ the PRF values (i.e., α') storage in $\mathcal{B}$, ❷ the corresponding bit values storage in $\mathcal{B}$, and ❸ the array of object pointers. The total storage cost is formulated as:

$$\text{Storage}(\mathcal{N}) = 256 \cdot p_s + p_n \cdot p_s + 64 \cdot p_n \quad (7)$$

Here, p_s denotes the number of distinct PRF values (α') in $\mathcal{B}$. Each PRF value takes 256 bits, and each object pointer uses 64 bits.

5.3 P^{3}RQ-WBTree Construction

With the node cost model, we construct a P^{3}RQ-WBTree that minimizes the sum cost of all nodes. We design a greedy bottom-up algorithm for building the P^{3}RQ-WBTree. The detailed procedure is presented in Algorithm 4. Initially, all objects are stored in a single P^{3}RQ-Bitmap, forming the leaf node $\mathcal{N}_{root}$ (line 2). Then, $\mathcal{N}_{root}$ is dequeued from a queue $\mathcal{F}$ as $\mathcal{N}$ and partitioned by a set of split borders. As a result, $\mathcal{N}$ is divided into multiple leaf nodes, while a non-leaf parent node $\mathcal{N}_p$ is created to index them (lines 7–8). The newly created parent node $\mathcal{N}_p$ is further tried to split following the split borders of its leaf nodes. This process repeats iteratively, generating non-leaf nodes until a new parent node cannot be split (lines 9-11). However, two key issues arise in the construction process:

How to split a node $\mathcal{N}$ into multiple parts? The node split is a recursive process. Initially, node $\mathcal{N}$ is enqueued into a queue *que* (lines 15-16). Then, we repeatedly dequeue node $\mathcal{N}$ from *que*, split it into two nodes $\mathcal{N}_l$ and $\mathcal{N}_r$, by determining a space split border b (lines 17-29). For a leaf node, the split border b is found to minimize the sum cost of the resulting child nodes (lines 19-21). For a non-leaf node, b is the split border of one of its child leaf nodes in the same spatial space, and all child node split borders are captured in a set $\mathcal{S}$ (lines 22-23). If the split results in a node cost decrease, indicating a successful split, $\mathcal{N}_l$ and $\mathcal{N}_r$ are enqueued, and the process continues until the *que* is empty (lines 24-28).

Where to split a leaf node (line 20)? When a node $\mathcal{N}$ is split by border b into two child nodes $\mathcal{N}_l$ and $\mathcal{N}_r$, its parent node $\mathcal{N}_p$ is updated to $\mathcal{N}_p'$ to index new child nodes. Meanwhile, the overall cost will be affected, and we obtain the cost difference Δc.

$$\Delta c = \text{Cost}(\mathcal{N}_l) + \text{Cost}(\mathcal{N}_r) + \text{Cost}(\mathcal{N}_p') - \text{Cost}(\mathcal{N}) - \text{Cost}(\mathcal{N}_p) \quad (8)$$

Our objective is to determine the optimal b that minimizes Δc. However, an exhaustive search over the entire space would be computationally prohibitive. To address this, we transform Δc into a model function with respect to the split border b, and then apply gradient descent to find the optimal b. The process is detailed as follows: ❶ We use the boundary positions (i.e., $\{q^l, q^r\}$ of each Q) in the query workload $\mathcal{W} = \{Q\}$ as candidate split borders, since the query boundaries are the most critical positions influencing query cost. For each candidate border b, we can efficiently collect the cost model parameters $\{p_q^l, p_q^r, p_s^l, p_s^r, p_n^l, p_n^r\}$ for the nodes $\mathcal{N}_l$ and $\mathcal{N}_r$ that would result from a split at b, without explicitly constructing the nodes. All these parameters vary monotonically, either increasing or decreasing, as b increases. Consequently, these distributions can be effectively approximated using piecewise linear models, denoted as $\{\mathcal{M}_{pq}^l, \mathcal{M}_{pq}^r, \mathcal{M}_{ps}^l, \mathcal{M}_{ps}^r, \mathcal{M}_{pn}^l, \mathcal{M}_{pn}^r\}$. For example, $\mathcal{M}_{pq}^l(b_0)$ represents the predicted value of p_q^l for the left node $\mathcal{N}_l$ split by $b = b_0$ according to the linear model $\mathcal{M}_{pq}^l$. ❷ Then, the query and storage cost for the child nodes $\mathcal{N}_l$ and $\mathcal{N}_r$ can be

Algorithm 4: P^{3}RQ-WBTree.Construct($\mathcal{D}, \mathcal{W}$)

```
Input: The dataset $\mathcal{D}$ and a query workload $\mathcal{W}$
Output: A P$^3$RQ-WBTree and return the root node $\mathcal{N}_{root}$
Function Construct($\mathcal{D}, \mathcal{W}$)
    $\mathcal{N}_{root} \leftarrow$ new LeafNode($\mathcal{D}, \mathcal{W}$);
    Initialize a split border set $\mathcal{S} \leftarrow \emptyset$;
    Initialize a queue $\mathcal{F}$ to maintain the nodes requiring finer splitting;
    $\mathcal{F}$.Enqueue($\mathcal{N}_{root}$);
    while $\mathcal{F}$.NotEmpty() do
        $\mathcal{N} \leftarrow \mathcal{F}$.Dequeue();
        $\mathcal{N}_p \leftarrow$ NodeSplit($\mathcal{N}, \mathcal{S}$);
        while $\mathcal{N}_p$ is the newly created parent node after NodeSplit do
            $\mathcal{N} \leftarrow \mathcal{N}_p$;
            $\mathcal{N}_p \leftarrow$ NodeSplit($\mathcal{N}, \mathcal{S}$);
        $\mathcal{N}_{root} \leftarrow$ maxmin top node of $\mathcal{N}_{root}$ and $\mathcal{N}$;
    return $\mathcal{N}_{root}$;
Function NodeSplit($\mathcal{N}, \mathcal{S}, \mathcal{F}$)
    $\mathcal{N}_p \leftarrow \mathcal{N}$.Parent(); /* null if $\mathcal{N}$ has no parent */
    Initialize a queue $que$ and $que$.Enqueue($\mathcal{N}$);
    while $que$.NotEmpty() do
        $\mathcal{N}_q \leftarrow que$.Dequeue();
        if $\mathcal{N}_q$ instance of LeafNode then
            border $b \leftarrow$ get split border with cost model;
            $\mathcal{S}$.Add($b$);
        else
            border $b \leftarrow$ get split border in $\mathcal{S}$;
        ($\mathcal{N}_l, \mathcal{N}_r, \mathcal{N}_p'$) $\leftarrow$ split node $\mathcal{N}_q$ by $b$ and update parent node $\mathcal{N}_p$;
        $\Delta c = \text{Cost}(\mathcal{N}_l) + \text{Cost}(\mathcal{N}_r) + \text{Cost}(\mathcal{N}_p') - \text{Cost}(\mathcal{N}) - \text{Cost}(\mathcal{N}_p)$;
        if $\Delta c < 0$ then
            $\mathcal{N}_p \leftarrow \mathcal{N}_p'$;
            $que$.Enqueue($\mathcal{N}_l$); $que$.Enqueue($\mathcal{N}_r$);
            for each child node $\mathcal{N}_c$ of $\mathcal{N}_q$ do $\mathcal{F}$.Enqueue($\mathcal{N}_c$);
    return $\mathcal{N}_p$;
```

expressed by models as follows:

$$\begin{aligned}
\text{Query}(\mathcal{N}_l) &= \text{T}_1 + \mathcal{M}^l_{pq}(b) \cdot \text{T}_2 + \mathcal{M}^l_{pq}(b) \cdot \mathcal{M}^l_{pn}(b) \cdot \text{T}_3\\
\text{Query}(\mathcal{N}_r) &= \text{T}_1 + \mathcal{M}^r_{pq}(b) \cdot \text{T}_2 + \mathcal{M}^r_{pq}(b) \cdot \mathcal{M}^r_{pn}(b) \cdot \text{T}_3\\
\text{Storage}(\mathcal{N}_l) &= 256 \cdot \mathcal{M}^l_{ps}(b) + \mathcal{M}^l_{pn}(b) \cdot \mathcal{M}^l_{ps}(b) + 8 \cdot \mathcal{M}^l_{pn}(b)\\
\text{Storage}(\mathcal{N}_r) &= 256 \cdot \mathcal{M}^r_{ps}(b) + \mathcal{M}^r_{pn}(b) \cdot \mathcal{M}^r_{ps}(b) + 8 \cdot \mathcal{M}^r_{pn}(b)
\end{aligned} \tag{9}$$

❸ Moreover, for the parent node change, the query cost remains unaffected by the split border, and the difference in storage cost is negligible. This can be expressed as:

$$\text{Query}(\mathcal{N}_p') - \text{Query}(\mathcal{N}_p) = \begin{cases} \text{T}_1 + p_q \cdot \text{T}_2 + 2 \cdot p_q \cdot \text{T}_3 & \mathcal{N}_p = \text{null} \\ p_q \cdot \text{T}_3 & \mathcal{N}_p \neq \text{null} \end{cases} \tag{10}$$
$$\text{Storage}(\mathcal{N}_p') - \text{Storage}(\mathcal{N}_p) = 256 \cdot \log\theta + p_n \cdot \log\theta + \log\theta + p_s + 8$$

❹ By substituting Equation (9) and (10) into Equation (8), we obtain the model function of Δc with respect to the border b. We then efficiently determine the optimal split border b_{opt} by applying the gradient descent method to minimize $\Delta c(b)$:

$$\begin{aligned}
b_{opt} &= \text{Argmin}(\Delta c) = \text{Argmin}(\text{Cost}(\mathcal{N}_l) + \text{Cost}(\mathcal{N}_r))\\
&= \text{Argmin}(w_q \cdot (\text{Query}(\mathcal{N}_l) + \text{Query}(\mathcal{N}_r))\\
&\quad + w_s \cdot (\text{Storage}(\mathcal{N}_l) + \text{Storage}(\mathcal{N}_r)))
\end{aligned} \tag{11}$$

❺ Using the above method, we attempt splitting along both the x and y axes to obtain candidate borders b^x_{opt} and b^y_{opt}, and last select the one with the lower cost as the optimal split border.

Example 3. *In Figure 3, (a) shows an spatial space with encrypted dataset $\mathcal{D}^* = \{o_1^*, \cdots, o_{10}^*\}$ and a query workload $\mathcal{W} = \{Q_1, \cdots, Q_4\}$. (b) shows the process of constructing the P^{3}RQ-WBTree's leaf nodes for (a). Initially, all objects are stored in a leaf node $\mathcal{N}$. Then, by building the linear models for $\{p_q, p_s, p_n\}$ and using gradient descent to find the optimal split border b_1, the node is divided into two sub-nodes, $\mathcal{N}_l$ and $\mathcal{N}_r$, with a corresponding parent node $\mathcal{N}_p$. Using the same approach, $\mathcal{N}_l$ and $\mathcal{N}_r$ are further split at b_2 and b_3 until, at b_4, the condition $\Delta c(b_4) \geq 0$ is met. Finally, the leaf nodes $\{\mathcal{N}_1, \mathcal{N}_2, \mathcal{N}_3, \mathcal{N}_4\}$ are obtained and indexed by the non-leaf node $\mathcal{N}_5$. Then (c) shows the splitting process for non-leaf node $\mathcal{N}_5$. Following the split path from the leaf nodes, splits are again attempted at $\mathcal{S} = \{b_1, b_2, b_3\}$, but only the split at b_1 is successful. This results in the non-leaf nodes $\{\mathcal{N}_6, \mathcal{N}_7\}$ and their parent node $\mathcal{N}_8$, with $\mathcal{N}_8$ being the root of the tree, as it no longer requires further splitting. The final constructed P^{3}RQ-WBTree is shown in (d).*

Algorithm 5: P^{3}RQ-WBTree.Query($\mathcal{TD}, \mathcal{N}_{root}$)

```
Input: P$^3$RQ-WBTree root node $\mathcal{N}_{root}$ and query token $\mathcal{TD}$
Output: Query result $\mathcal{R}$
Initialize a queue $que$ and a result set $\mathcal{R}$;
$que$.Push($\mathcal{N}_{root}$);
while $que$.NotEmpty() do
    $\mathcal{N} \leftarrow que$.Pop();
    $\mathcal{R}_\mathcal{N} \leftarrow$ P$^3$RQ-Bitmap.Query($\mathcal{TD}, \mathcal{N}.\mathcal{B}, \mathcal{N}.path$);
    if $\mathcal{N}$ instance of LeafNode then
        $\mathcal{R}$.AddAll($\mathcal{R}_\mathcal{N}$); /* $\mathcal{R}_\mathcal{N}$ is the result objects in $\mathcal{N}$ */
    else
        $que$.PushAll($\mathcal{R}_\mathcal{N}$); /* $\mathcal{R}_\mathcal{N}$ is $\mathcal{N}$'s child nodes satisfying the query*/
return $\mathcal{R}$;
```

Finer splitting. We observe that a node $\mathcal{N}$ may not split because the cost of its parent node $\mathcal{N}_p$ increases more when trying to update to $\mathcal{N}_p'$ (i.e., Equation (10)). However, $\mathcal{N}_p$ might be split at a higher level, causing the parent of $\mathcal{N}$ to be updated from $\mathcal{N}_p$ to a new parent $\mathcal{N}_p''$. As a result, the parameters $\{p_q, p_s, p_n\}$ in the new parent node may decrease, potentially making $\mathcal{N}$ eligible for split again to enable $\Delta c < 0$ (i.e., Equation (8)). In other words, splitting a node can reduce the split cost of its children, potentially making them eligible for further refinement. Consider this, as shown in Algorithm 4, we maintain a queue $\mathcal{F}$ to track candidate nodes for finer splitting (line 29). After the initial index construction, these nodes are revisited in a bottom-up manner to assess whether additional splits are beneficial (lines 7-12). This iterative process continues until no node requires further refinement (line 6).

Example 4. *In Figure 3 (d), $\mathcal{N}_4$ is further split into $\mathcal{N}_{12}$ and $\mathcal{N}_{13}$ during finer splitting. This occurs because, after the tree is constructed, its parent node changes from the initially split node $\mathcal{N}_5$ to the smaller nodes $\mathcal{N}_6$ and $\mathcal{N}_7$, which results in a smaller increase in the parent node's query cost, thus reducing the overall split cost in border b_4.*

5.4 Query Processing

Algorithm 5 details the query processing in the P^{3}RQ-WBTree using the query token $\mathcal{TD}$. It utilizes a queue que for breadth-first tree traversal (lines 1-2). Within each accessed node $\mathcal{N}$, the P^{3}RQ-Bitmap query algorithm retrieves the node result $\mathcal{R}_\mathcal{N}$ that satisfies the query (lines 4-5). If $\mathcal{N}$ is a leaf node, $\mathcal{R}_\mathcal{N}$ is the objects satisfying the query and added to $\mathcal{R}$; if $\mathcal{N}$ is non-leaf node, $\mathcal{R}_\mathcal{N}$ is the child nodes satisfying the query and enqueued in que for further traversal (lines 6-9).

Example 5. *In Figure 3. When querying the red region in (a), the query path is marked in red within (d). The query begins at the root node $\mathcal{N}_8$ deploying P^{3}RQ-Bitmap.Query($\mathcal{TD}$, $\mathcal{N}_8.\mathcal{B}$, $\mathcal{N}_8.r$), determining that $\mathcal{N}_{14}$ satisfies the query. Next, $\mathcal{N}_{14}$.P^{3}RQ-Bitmap further narrows the search to the node $\mathcal{N}_3$. Finally, $\mathcal{N}_3$.P^{3}RQ-Bitmap identifies the encrypted object o_1^* and add to the query result $\mathcal{R}$.*

Analysis. We analyze the query and storage complexity of P^{3}RQ-WBTree in two cases. ❶ *Cold start:* the index is constructed using only the dataset $\mathcal{D}$ without workload $\mathcal{W}$. In this case, each node splits into two equal-sized children to maximize storage cost reduction, forming a balanced tree of height $O(\log n)$. Since each node is indexed by a P^{3}RQ-Bitmap, a leaf node with $O(1)$ objects has query and storage complexity of $O(\theta)$ and $O(\log \theta)$, respectively. Aggregating all nodes, the overall query complexity of P^{3}RQ-WBTree is $O(\theta \log n)$, and the storage complexity is $O(n \log \theta)$. ❷ *General case:* the index incorporates workload $\mathcal{W}$ and is no longer strictly balanced. Tree height may exceed $O(\log n)$ in sparse-query regions and be lower in dense regions. This adaptive structure minimizes total query cost rather than enforcing strict balance. Consequently, while the worst-case height can surpass $O(\log n)$, the average query complexity is typically below $O(\theta \log n)$.

5.5 Dual Update of P^{3}RQ-WBTree

In this subsection, we present the P^{3}RQ-WBTree update algorithm, enabling it to adapt to dynamic data and workload changes.

Node buffer strategy. When inserting an object into the tree, it is first indexed into the appropriate leaf node $\mathcal{N}$. If the insertion alters the MBR of $\mathcal{N}$, the parent node $\mathcal{N}_p$ is recursively updated upward. The update overhead for each node stems from modifications to its Bitmap $\mathcal{B}$. However, inserting a new object requires appending a new column of bits to $\mathcal{B}$, which has p_s rows and p_n columns. This entails recomputing all the α' and β' values across the p_s rows in order to encrypt the $(p_n + 1)$-th column. The resulting update complexity is $O(p_s)$. To mitigate this cost, we introduce a buffer strategy for P^{3}RQ-Bitmap. Specifically, during index construction, each $\mathcal{B}$ is pre-extended by $n \cdot \eta$ column of bits as a buffer to avoid row-wise recomputation for each insertion. When we add an object o into node $\mathcal{N}$, Algorithm 1 encrypts o to obtain the corresponding PRFs $\{\alpha'\}$ and bitsets $\{\mathcal{B}[\alpha']\}$. For values of α' that do not yet exist in $\mathcal{N}.\mathcal{B}$, the encrypted α' and $\mathcal{B}[\alpha']$ are sent to the server for storage. For values of α' that already exist in $\mathcal{N}.\mathcal{B}$, the server only needs to flip the bit $\mathcal{B}[\alpha'][p_n + 1]$. This strategy reduces the update complexity of PRF computations from $O(p_s)$ to $O(\theta)$.

incremental reconstructing strategy. Recall that in the construction of P^{3}RQ-WBTree, each node is split by identifying the optimal border b_{opt} of a spatial space, aiming to minimize the total cost of the nodes constructed in the subspaces. However, updates may alter the dataset and workload distribution in the space, making the original b_{opt} no longer the optimal split point. To address this, we propose a Cost Monitor Tree (CM-Tree), which monitors the node cost of each subspace after every split. If an update causes a significant cost imbalance between the subspaces, it indicates that the previous split is no longer effective, and the affected space should be re-partitioned to reconstruct the P^{3}RQ-WBTree.

Definition 6 (CM-Tree Node). A node $\mathcal{N}^m$ in the CM-Tree, which monitors the cost of a space (i.e., the total cost of the corresponding

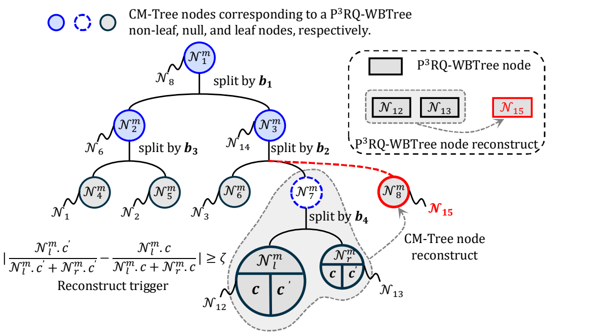


Figure 4: Example of the CM-Tree monitoring mechanism. After several updates, the split by b_4 no longer fits the corresponding space, triggering a subtree reconstruction.

P^{3}RQ-WBTree nodes constructed using a dataset $\mathcal{D}$ and a workload $\mathcal{W}$ in the space), contains the following fields:

- c, c': The monitored cost for index construct and after updates.
- $\mathcal{N}_l^m, \mathcal{N}_r^m$: The left and right child nodes of $\mathcal{N}^m$ for monitoring the split sub-dataset and sub-workload.

If the space monitor by node $\mathcal{N}^m$ is corresponding to a P^{3}RQ-WBTree node $\mathcal{N}$, it additionally includes:

- p_s: Number of PRF values storage in $\mathcal{B}$ generated by $\mathcal{D}$;
- p_q: Number of PRF values in query token generated by $\mathcal{W}$.

In above, the node monitored cost $\mathcal{N}^m.c$ is initialized as

$$\mathcal{N}^m.c = \mathcal{N}_l^m.c + \mathcal{N}_r^m.c + \mathrm{Cost}(\mathcal{N}) \cdot [\mathcal{N} \neq null] \tag{12}$$

When updates occur, the Cost($\mathcal{N}$) can be efficiently updated using Equation (5), (6), and (7) based on the updated $\mathcal{N}^m.p_s$ and $\mathcal{N}^m.p_q$. Thus, the updated cost $\mathcal{N}^m.c'$ can also be efficiently updated by Equation (12). Then, we introduce a tolerance parameter ζ. When the cost ratio between the child nodes of a node $\mathcal{N}^m$ deviates from its initial value by more than ζ, the tree structure of the monitored space is reconstructed. The trigger condition is defined as:

$$\left|\frac{\mathcal{N}_l^m.c'}{\mathcal{N}_l^m.c' + \mathcal{N}_r^m.c'} - \frac{\mathcal{N}_l^m.c}{\mathcal{N}_l^m.c + \mathcal{N}_r^m.c}\right| \geq \zeta \tag{13}$$

Example 6. *Figure 4 illustrates the CM-Tree monitoring mechanism. Each node monitors a partitioned space from Figure 3, and maintains the corresponding costs. Specifically, $\mathcal{N}_l^m.c = Cost(\mathcal{N}_{12})$, $\mathcal{N}_7^m.c = \mathcal{N}_l^m.c + \mathcal{N}_r^m.c$, $\mathcal{N}_3^m.c = \mathcal{N}_6^m.c + \mathcal{N}_7^m.c + Cost(\mathcal{N}_{14})$, and $\mathcal{N}_1^m.c = \mathcal{N}_2^m.c + \mathcal{N}_3^m.c$ is the total cost of all P^{3}RQ-WBTree nodes. We observe that $\mathcal{N}_{12}$ and $\mathcal{N}_{13}$ in the P^{3}RQ-WBTree are reconstruct into a new node $\mathcal{N}_{15}$ because the corresponding monitoring CM-Tree node $\mathcal{N}_7^m$ has triggered the reconstruction condition.*

Analysis. When the workload or data distribution shifts, our strategy enable the reconstruction first triggered at the leaf nodes of the CM-tree, which monitor $O(1)$ data, and subsequently proceeds level by level with costs $2^1O(1), 2^2O(1), \ldots$, until the root rebuilds over $O(n)$ data. ❶ In an extreme case, if $\log n$ consecutive updates each trigger reconstruction, the update complexity is $(\sum_{i=0}^{\log n} 2^i O(1))/\log n = O(n/\log n)$. ❷ Under normal cases, a node covering n elements usually be reconstructed after $O(n)$ object updates, so its reconstruct cost can be amortized to $O(1)$ per update. Since each update traverses $O(\log n)$ nodes, the amortized complexity is $O(\log n)$. Therefore, both above complexities are significantly more efficient than the $O(n)$ cost of a full index reconstruct.

Table 3: Complexity analysis of our schemes

Scheme	Query Complexity	Storage Complexity
P^{3}RQ-Linear	$t_1 O(\theta \cdot n)$	$s_1 O(\theta \cdot n)$
P^{3}RQ-Bitmap	$t_1 O(\theta) + t_2 O(\theta \cdot n)$	$s_1 O(n \log \theta) + s_2 O(n^2 \log \theta)$
P^{3}RQ-WBTree	$t_1 O(\theta \log n) + t_2 O(\theta \log n)$	$s_1 O(n \log \theta) + s_2 O(n \log \theta)$
KDTree	$t_1 O(\theta \log n) + t_2 O(\theta \log n)$	$s_1 O(n \log \theta) + s_2 O(n \log \theta)$

Notes: t_1, s_1 represent the computation and storage cost of the PRF, respectively; t_2, s_2 represent the bitwise computation and storage cost, respectively. Typically, a PRF value requires $s_1 = 256$ bits, while a bitwise value requires $s_2 = 1$ bit.

6 THEORETICAL ANALYSIS

6.1 Complexity Analysis

Based on the **Analysis** in Sections 4 and 5, we summarize the query complexity and storage complexity in Table 3. Here, KDTree refers to the scheme using KD-tree construction instead of the workload-aware method.

6.2 Security Analysis

In this subsection, we take P^{3}RQ-WBTree as an example to prove that our scheme is secure against IND-CPA [9]. Firstly, we define the leakage function $\mathcal{L} = \{\mathcal{L}_{sp}, \mathcal{L}_{rp}, \mathcal{L}_{qp}, \mathcal{L}_{ap}\}$ to evaluate the privacy leaks in P^{3}RQ-WBTree.

- *Size Pattern* ($\mathcal{L}_{sp} = \{n, |\mathcal{D}^*|, |\mathcal{I}^*|\}$): reveals the size of the encrypted dataset and index structure.
- *Result Pattern* ($\mathcal{L}_{rp} = \{o_j^*\}_{\forall j \in n, o_j^* \in \mathcal{R}_Q}$): indicates that the adversary can distinguish the structure of query results.
- *Query Pattern* ($\mathcal{L}_{qp} = \{\mathcal{TD}_i\}_{\forall i \in [|\mathcal{TD}|]}$): reveals whether two queries are identical by comparing their tokens.
- *Access Pattern* ($\mathcal{L}_{ap} = \{\mathcal{N}_j\}_{\forall j \in [|\mathcal{N}|], \mathcal{R}_Q \cap \mathcal{R}_{\mathcal{N}_j} \neq \emptyset}$): reveals whether two queries traverse a common node in the index.

Definition 7 (IND-CPA Security of P^{3}RQ-WBTree). Let a P^{3}RQ-WBTree scheme be denoted as Π = (Setup, Construct, TokenGen, Query). Given the leakage function $\mathcal{L}$, the IND-CPA security game between an adversary $\mathcal{A}$ and a challenger C is defined as follows:

- **Init**: On input a security parameter λ, the adversary $\mathcal{A}$ outputs a workload $\mathcal{W}$, two databases $\mathcal{D}_0, \mathcal{D}_1$ of equal size n, and two queries Q_0, Q_1, which are sent to the challenger C.
- **Setup**: The challenger C runs Setup(1^λ) to generate a secret key sk, which is kept private.
- **Phase 1**: The adversary $\mathcal{A}$ adaptively issues requests of the following two types: (i) *Ciphertext request.* On the j-th request, $\mathcal{A}$ submits a dataset $\mathcal{D}_j^{'}$. The challenger responds with the encrypted dataset and index $(\mathcal{D}_j^*, \mathcal{I}_j^*) \leftarrow$ Construct$(\mathcal{D}_j^{'}, \mathcal{W})$, where $\mathcal{D}_0$ and $\mathcal{D}_1$ generate identical leakage $\mathcal{L}_{sp}$. (ii) *Query request.* On the j-th request, $\mathcal{A}$ submits a query $Q_j^{'}$ and an encrypted index $\mathcal{I}^*$. The challenger responds with a token $\mathcal{TD}_j \leftarrow$ TokenGen$(sk, Q_j^{'})$ and the result $\mathcal{R}_j \leftarrow$ Query$(\mathcal{TD}_j, \mathcal{I}^*)$, where Q_0 and Q_1 on index $\mathcal{I}^*$ incur the same leakage $\mathcal{L}_{rp}, \mathcal{L}_{qp}, \mathcal{L}_{ap}$.
- **Challenge**: The challenger proceeds with one of the following experiments: (i) *Dataset challenge.* Using $\mathcal{D}_0, \mathcal{D}_1$ from **Init**, challenger C selects a random bit $b_d \in \{0, 1\}$, computes $\mathcal{D}_{b_d}^* \leftarrow$ Construct$(\mathcal{D}_{b_d}, \mathcal{W})$, and sends it to $\mathcal{A}$. (ii) *Query challenge.* Using Q_0, Q_1 from **Init** and the encrypted index $\mathcal{I}^*$, C selects a random bit $b_q \in \{0, 1\}$, computes $\mathcal{TD}_{b_q} \leftarrow$ TokenGen(sk, Q_{b_q}), $\mathcal{R}_{b_q} \leftarrow$ Query$(\mathcal{TD}_{b_q}, \mathcal{I}^*)$, and send them to $\mathcal{A}$.

Table 4: Parameter setting (bold is the default)

Parameter	Setting
Dataset	BPD **OSM** Uniform Skew
Workload	UNI LAP GAU **MIX**
# of evaluated dimension	1 **2** 3 4 5 6
# of objects (M)	0.2 0.4 0.6 0.8 **1.0** 5 10 50 100
# of query region	0.2% 0.4% **0.6%** 0.8% 1%
Weight rate w_q/w_s	0/1 1/8 1/32 1/1 8/1 **32/1** 1/0
Buffer rate η	0.1 **0.2** 0.3 0.4 0.5
Reconstruct rate ζ	0.1 **0.4** 0.7 1

- **Phase 2**: The adversary $\mathcal{A}$ may continue to adaptively issue requests as in Phase 1.
- **Guess**: Finally, $\mathcal{A}$ outputs a guess $b_d^{'}$ for b_d or $b_q^{'}$ for b_q.

We say that Π is secure against IND-CPA attacks on data and query privacy if for any polynomial-time adversary in the above game, it has at most a negligible advantage

$$\mathbf{Adv}_{\Pi,\mathcal{A}}^{\text{IND-SCPA-Data}}(1^\lambda) = |\mathbf{Pr}[b_d^{'} == b_d] - \frac{1}{2}| \leq \mathbf{negl}(\lambda).$$

$$\mathbf{Adv}_{\Pi,\mathcal{A}}^{\text{IND-SCPA-Query}}(1^\lambda) = |\mathbf{Pr}[b_q^{'} == b_q] - \frac{1}{2}| \leq \mathbf{negl}(\lambda).$$

where $\mathbf{negl}(\lambda)$ represents a negligible function.

Theorem 1 (Data and Query Privacy of P^{3}RQ-WBTree). P^{3}RQ-WBTree achieves $\mathcal{L}$-IND-CPA data privacy and query privacy if no probabilistic polynomial-time adversary can win the IND-CPA game defined in Definition 7 with non-negligible advantage.

Proof. To prove $\mathcal{L}$-IND-CPA security, consider an adversary $\mathcal{A}$ in the game of Definition 7. We show that $\mathcal{A}$ cannot distinguish between two datasets or between two queries. This holds as long as the encrypted index, query tokens, and results reveal only the leakage $\mathcal{L}$. ❶ **Data privacy.** Each object in an index node is encoded into PRF-derived values and an obfuscated bitmap using a fresh random value r per node. The uniqueness of r and the pseudorandomness of PRF ensure that identical plaintexts map to independent, indistinguishable ciphertext values. Moreover, the encrypted dataset sizes leaked through $\mathcal{L}_{sp}$ are identical for $\mathcal{D}_0$ and $\mathcal{D}_1$. Consequently, $\mathcal{A}$ cannot distinguish $\mathcal{D}_0^*$ from $\mathcal{D}_1^*$ unless it breaks the underlying PRF. ❷ **Query privacy.** For a query Q, the token $\mathcal{TD}$ is generated via PRF, and executing Q on the encrypted index $\mathcal{I}^*$ generates the result $\mathcal{R}$. Since queries Q_0 and Q_1 incur identical leakages $\mathcal{L}_{rp}, \mathcal{L}_{ap}$, their results are indistinguishable in shape. Furthermore, the pseudorandomness of PRF ensures that tokens $\mathcal{TD}_0$ and $\mathcal{TD}_1$ are computationally indistinguishable. Thus, $\mathcal{A}$ cannot distinguish $\mathcal{TD}_0$ from $\mathcal{TD}_1$ or $\mathcal{R}_0$ from $\mathcal{R}_1$ without distinguishing PRF outputs. Therefore, P^{3}RQ-WBTree achieves $\mathcal{L}$-IND-CPA data and query privacy. □

7 EXPERIMENTS

7.1 Experiment Setup

Datasets and Workloads. We evaluate performance on two real-world and two synthetic datasets: ❶ BPD, a two-dimensional POI dataset from the SLIPO project containing 1M objects [2]; ❷ OSM, OpenStreetMap-based POIs from the UCR STAR repository comprising 100M six-dimensional objects, including GPS coordinates, ID, timestamp, etc. [14]; ❸ Uniform, a synthetic six-dimensional dataset with a uniform spatial distribution containing 100M objects; and ❹ Skew, a synthetic six-dimensional dataset with a skewed spatial distribution containing 100M objects. Since no public query

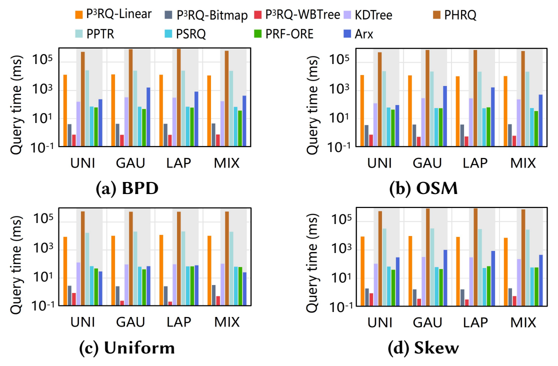


Figure 5: Query performance varying query workload

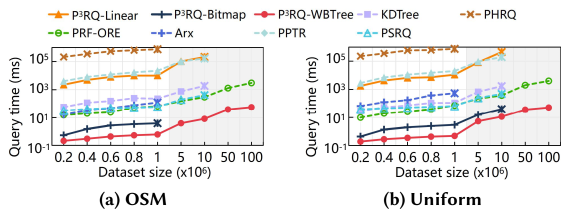


Figure 6: Query performance varying dataset size

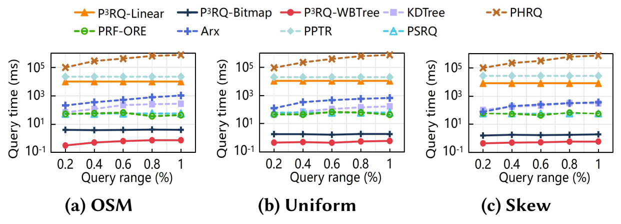


Figure 7: Query performance varying query range

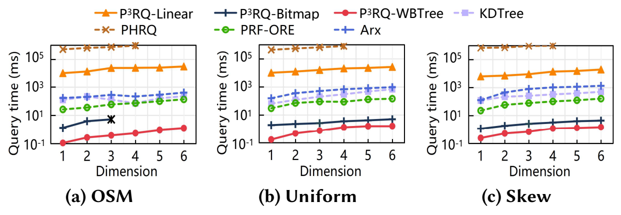


Figure 8: Query performance varying dimension

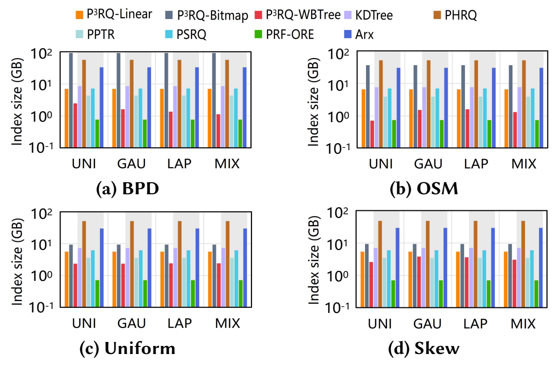


Figure 9: Index storage overhead varying query workload

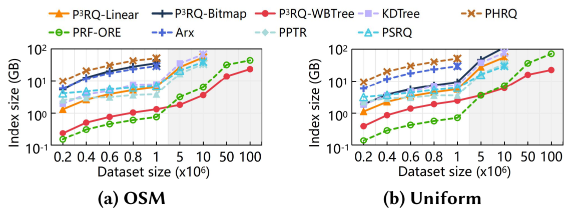


Figure 10: Index storage overhead varying dataset size

workloads exist for geo-textual datasets, we generate queries following prior methodologies [33]. For each dataset, we create four distinct workload types (800 queries each): ❶ UNI: Queries with centers uniformly sampled from the dataset; ❷ LAP: Sampled from a Laplace distribution; ❸ GAU: Sampled from a Gaussian distribution; ❹ MIX: Equal mix of UNI and LAP queries.

Setup. Our experiments adopt the parameters detailed in Table 4. Cryptographic primitives of PRF via SHA-256 are implemented using Java's standard libraries. All algorithms are executed on a workstation equipped with an Intel i9-12900 CPU (2.40 GHz), 128 GB RAM, and Windows 10 Professional.

Algorithm. To evaluate performance, we examine the following schemes: ❶ P^3RQ-Linear: Our basic PRF-based privacy-preserving range queries scheme without an index; ❷ P^3RQ-Bitmap: Our enhanced version of P^3RQ-Linear incorporating XOR-encrypted bitmap index; ❸ P^3RQ-WBTree: Our workload-aware encrypted index; ❹ KDTree: An extended scheme integrating P^3RQ-Bitmap with KD-tree, which is a widely adopted structure for privacy-preserving spatial queries [8]. ❺ PHRQ: A HE-enabled method with B+-tree index [34]; ❻ PPTR: A RMM-based encryption scheme [46]; ❼ PSRQ: A HVE-enabled method with BF-based index [24]; ❽ PRF-ORE: A PRF-based ORE scheme [6]; ❾: Arx: A GC-enabled scheme incorporating a path-repair mechanism for IND-CPA security and partial access pattern preservation [30].

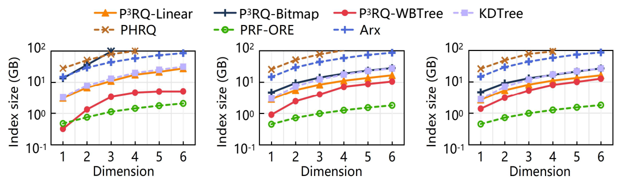
7.2 Query Performance

Impact of query workload. Figure 5 shows the impact of varying workloads on query performance across 1M object datasets. P^3RQ-WBTree outperforms the best IND-CPA secure SOTA scheme, the PSRQ, by a factor of 83×. P^3RQ-Bitmap significantly accelerates queries compared to P^3RQ-Linear (by 2,587× to 6,220×), while P^3RQ-WBTree further improves performance by at least 12×. Although KDTree has similar query complexity to P^3RQ-WBTree, its actual performance lags behind P^3RQ-Bitmap by at least 40×.

Impact of dataset size. As illustrated in Figure 6, the query time of all schemes grows as the dataset size increases from 0.2×10^6 to 100×10^6. However, most schemes can only handle limited dataset sizes due to unaffordable index storage overhead (see Figure 10). In contrast, our P^3RQ-WBTree is highly scalable and consistently achieves the best performance across all dataset sizes.

Impact of query range. As shown in Figure 7, the query time of PPTR, PSRQ, P^3RQ-Linear, P^3RQ-Bitmap, and PRF-ORE remains stable as the query range increases, while KDTree, PHRQ, Arx and P^3RQ-WBTree grow linearly. Despite this, P^3RQ-WBTree always maintains the best efficiency.

Impact of dataset dimension. As shown in Figure 8, our P^3RQ-WBTree is adapted to high-dimensional datasets. For instance, on the six-dimensional OSM dataset, P^3RQ-WBTree achieves up to 109× higher efficiency than the SOTA PRF-ORE scheme.



(a) OSM (b) Uniform (c) Skew

Figure 11: Index storage overhead varying dimension

Table 5: Index construction time cost of the comparisons and our schemes in various datasets with 1M objects (time: s)

Dataset (1×10^6 size)	BPD	OSM	Uniform	Skew
KDTree	92.84	80.03	80.51	75.31
PHRQ[34]	952.20	772.49	793.73	718.62
PPTR[46]	45.33	31.63	27.91	26.18
PSRQ[24]	201.54	182.46	165.18	161.57
PRF-ORE[6]	33.38	30.77	25.82	114.10
Arx[30]	91.02	92.86	122.51	91.58
P^3RQ-Linear	49.72	44.18	38.37	35.54
P^3RQ-Bitmap	**32.47**	**29.92**	**10.37**	**9.39**
P^3RQ-WBTree	126.73	91.76	110.58	108.89

7.3 Index Storage Size

Impact of query workload. Figure 9 shows that P^3RQ-WBTree achieves efficient storage, using at most 4%, 67%, 38%, 32%, 10% of the index sizes of PHRQ, PPTR, PSRQ, KDTree, and Arx, respectively. As expected, P^3RQ-Bitmap incurs high overhead, up to 13 × that of P^3RQ-Linear in worst-case. In contrast, P^3RQ-WBTree significantly reduces the storage overheads of P^3RQ-Bitmap, achieving 98.8% storage savings in optimal cases.

Impact of dataset size. Figure 10 shows that all schemes exhibit increasing index sizes with larger datasets. Among the IND-CPA secure schemes, only our P^3RQ-WBTree scales to datasets with 100M objects, while even incurring lower storage overhead than the less secure PRF-ORE scheme.

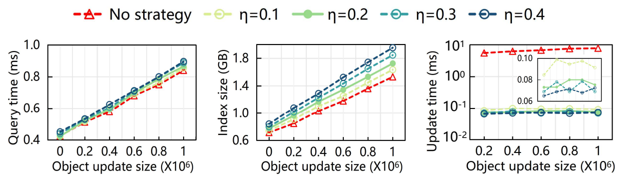
Impact of dataset dimension. As shown in Figure 11, P^3RQ-Bitmap on the OSM dataset is limited to low dimensions (95GB in three dimensions), whereas the optimized P^3RQ-WBTree consistently achieves the lowest storage overhead among IND-CPA secure schemes.

7.4 Index Construction Time

Table 5 compares the index construction time of all schemes. Our P^3RQ-Bitmap is comparable to PRF-ORE and outperforms the others, achieving average construction times of 8.5%, 55.2%, 9.3%, and 32.1% of those required by PBRQ, PPTR, PSRQ, and Arx respectively. Although P^3RQ-WBTree exhibits 7.9× longer construction time than P^3RQ-Bitmap, its time remains close to that of KDTree, with only a 13.7% increase. This marginal offline time investment enables substantial online efficiency.

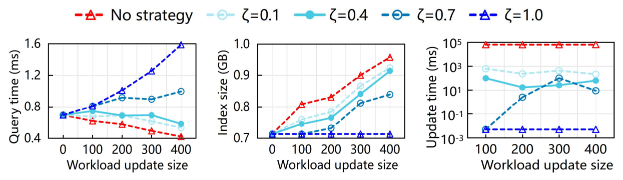
7.5 Index Update

Impact of node buffer strategy. Figure 12 evaluates the buffer strategy on update performance and its trade-offs in query efficiency and index size using the 1M OSM dataset with 1M object insertions. Every 200,000 updates, we measure index performance under different buffer rates η. Larger η values yield faster updates



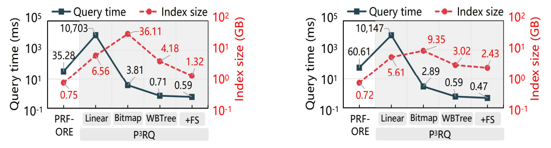
(a) Query time (b) Index size (c) Update time

Figure 12: Impact of buffer strategy for object updates



(a) Query time (b) Index size (c) Update time

Figure 13: Impact of reconstruct strategy for workload drift



(a) OSM (b) Uniform

Figure 14: Ablation study on query and storage performance

but incur higher query latency and index size. We set $\eta = 0.2$ as the default since it provides the best balance. Compared with no strategy, it improves update efficiency by 84× while query time increases by only 2.9% and storage by 11.4%.

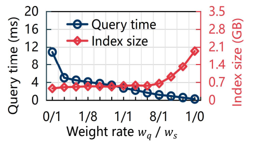
Impact of incremental reconstructing strategy. Figure 13 evaluates the reconstruct strategy under workload drift by updating 400 queries of the LAP workload on an index built from 1M OSM objects with 800 UNI queries workload. Every 100 updates, we measure performance under different tolerance rates ζ. This strategy avoids the high update overhead of frequent full index reconstructions, and larger ζ accelerates updates but weakens index repair effectiveness. In the extreme case $\zeta = 1.0$, no reconstructs occur and query efficiency drops super-linearly to 26.6%. We set $\zeta = 0.4$ as the default since it preserves 91.3% of query efficiency and 83.3% of index size while avoiding 99.9% of reconstruct time.

7.6 Ablation Study

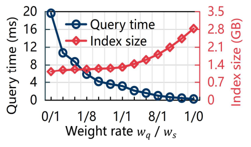
We compare our P^3RQ schemes with the baseline PRF-ORE, highlighting the trade-offs in query and storage performance incurred by the enhanced security, as well as the performance improvements achieved by each successive optimization, as shown in Figure 14. We observe that while P^3RQ-Linear decreases query and storage performance, both P^3RQ-Bitmap and P^3RQ-WBTree mostly improve performance and even largely surpass PRF-ORE. Among this, "+FS" denotes P^3RQ-WBTree with the finer splitting strategy.

7.7 Cost Model Performance

Figure 15 validates the accuracy of our cost model design. By adjusting the query weight w_q or storage weight w_s, the corresponding performance can be tuned. Specifically, increasing w_q improves

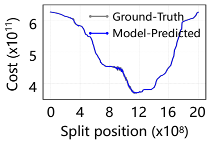


(a) OSM

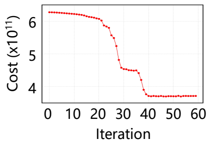


(b) Uniform

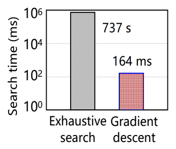
Figure 15: Index performance varying cost weight



(a) Node split cost vs. split border



(b) Cost vs. iteration in gradient descent



(c) Split border search time

Figure 16: Cost modeling to search optimal node split border on the OSM dataset with MIX workload

query efficiency, whereas increasing w_s reduces storage overhead. In general, we set $w_q/w_s = 32/1$ as the default configuration, which achieves high query efficiency while keeping storage overhead within an acceptable range.

To assess whether the piecewise-linear function using the gradient descent can efficiently and accurately search node splits, we conduct experiments on the OSM 1M dataset with the MIX workload, as shown in Figure 16. (a) compares the ground-truth cost curve with the cost estimated by our piecewise-linear models at different split borders. The results demonstrate that the models closely approximate the cost parameters $\{p_s, p_q, p_n\}$ and yield node split costs consistent with the ground truth. (b) illustrates the split border search process using gradient descent, which accurately identifies the split border with the lowest cost. (c) reports the runtime of exhaustive search and gradient descent, showing that our gradient descent-based method achieves up to 4,493× faster search efficiency.

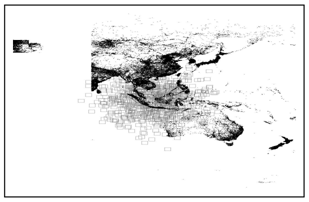
7.8 Index Partition Visualization

Figure 17 compares the partitioning effectiveness between P^{3}RQ-WBTree and KDTree on the OSM dataset with GAU workload. The KDTree exclusively optimizes for data distribution, resulting in partitions that precisely mirror the underlying data topology. In contrast, our P^{3}RQ-WBTree, through its dual optimization of both data and query distributions, generates complex adaptive partitioning geometries that vary significantly with different query-storage weight configurations. In addition, the number of partitions in P^{3}RQ-WBTree decreases as the query weight increases.

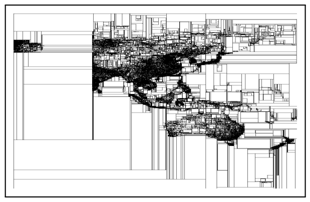
8 RELATED WORK

8.1 Privacy-Preserving Range Query Processing

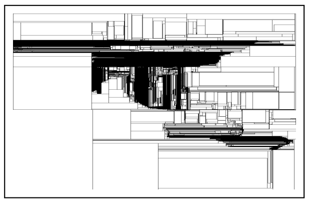
Privacy-preserving range query mechanisms have progressed from cryptographic to structural paradigms. Early schemes, such as OPE [1] and ORE [3, 6, 20], enable ciphertext comparison but inevitably leak order information, making them vulnerable to inference attacks [15]. To enhance security, methods based on HE [22, 23, 34, 42], ASPE [7, 25], RMM [46], and HVE [24, 40, 41] support secure computation on encrypted data, yet incur prohibitive costs on large datasets. To improve efficiency, Bloom Filter–based approaches



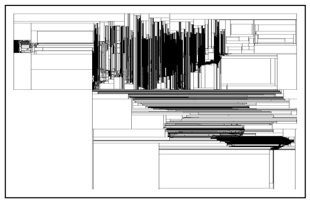
(a) OSM dataset with GAU workload



(b) KDTree



(c) P^{3}RQ-WBTree $w_q/w_s = 1/32$



(d) P^{3}RQ-WBTree $w_q/w_s = 32/1$

Figure 17: Visualization of partition for P^{3}RQ-WBTree and KDTree under OSM dataset with GAU workload

[8, 38, 44] leverage hash mappings but suffer from false positives. Structural indexes, including R-trees, KD-trees, Quad-trees, B+-trees, Binary-trees, and Radix-trees [5, 8, 23, 34, 36, 38, 41, 42, 44], have been incorporated into encrypted frameworks to achieve sub-linear query efficiency. However, none of these schemes support workload-aware optimization.

8.2 Workload-Aware Index

Workload-aware indexes improve efficiency by learning query distributions. For one-dimensional data, recent works [21, 26] design adaptive learned indexes emphasizing memory efficiency and scalability. In spatial settings, enhanced R-trees [12, 16] and disk-based systems [27] exploit query-aware node splitting and reinforcement learning to balance latency and update cost. For multi-dimensional data, space-filling curve mappings [13, 29] and workload-guided partitioning [11, 28] adaptively align index structures with query distributions. Extending further, WISK [33] integrates spatial and textual features for multi-modal queries. However, all these methods assume direct access to plaintext data, making them incompatible with privacy-preserving frameworks.

9 CONCLUSION

In this paper, we studied a workload-aware encrypted index for efficient privacy-preserving range queries. To this end, we first proposed a lightweight PRF-based comparison protocol and built the XOR-encrypted bitmap index P^{3}RQ-Bitmap, enabling workload-aware and privacy-preserving range queries. We further proposed the workload-aware encrypted index P^{3}RQ-WBTree, which optimizes query efficiency through adaptive data partitioning guided by a gradient descent-optimized cost model. Theoretical analysis and extensive experiments demonstrate that P^{3}RQ-WBTree achieves at least 83× faster query performance compared to SOTA schemes.

ACKNOWLEDGMENTS

The work is partially supported by Natural Science Foundation of Shaanxi Province General Program Project (No. 2025JC-YBMS-673) and Hong Kong RGC Project (No. 12201925).

REFERENCES

[1] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. 2004. Order-Preserving Encryption for Numeric Data. In *SIGMOD*. 563–574.

[2] Spiros Athanasiou, Michail Alexakis, Giorgos Giannopoulos, Nikos Karagiannakis, Yannis Kouvaras, Pantelis Mitropoulos, Kostas Patroumpas, and Dimitrios Skoutas. 2019. SLIPO: Large-Scale Data Integration for Points of Interest. In *EDBT*. 574–577.

[3] Dan Boneh, Kevin Lewi, Mariana Raykova, Amit Sahai, Mark Zhandry, and Joe Zimmerman. 2015. Semantically Secure Order-Revealing Encryption: Multi-input Functional Encryption Without Obfuscation. In *EUROCRYPT*. 563–594.

[4] Ruichu Cai, Zijie Lu, Li Wang, Zhenjie Zhang, Tom Z. J. Fu, and Marianne Winslett. 2017. DITIR: Distributed Index for High Throughput Trajectory Insertion and Real-time Temporal Range Query. *Proc. VLDB Endow.* 10, 12 (2017), 1865–1868.

[5] Zhihao Chen, Qingqing Li, Xiaodong Qi, Zhao Zhang, Cheqing Jin, and Aoying Zhou. 2022. BlockOPE: Efficient Order-Preserving Encryption for Permissioned Blockchain. In *ICDE*. 1245–1258.

[6] Nathan Chenette, Kevin Lewi, Stephen A. Weis, and David J. Wu. 2016. Practical Order-Revealing Encryption with Limited Leakage. In *FSE*. 474–493.

[7] Ningning Cui, Jianxin Li, Xiaochun Yang, Bin Wang, Mark Reynolds, and Yong Xiang. 2019. When Geo-Text Meets Security: Privacy-Preserving Boolean Spatial Keyword Queries. In *ICDE*. 1046–1057.

[8] Ningning Cui, Dong Wang, Huaijie Zhu, Jianxin Li, Jianliang Xu, and Xiaochun Yang. 2024. Enabling Verifiable and Secure Range Query in Multi-User Setting Under Cloud Environments. *IEEE TKDE* 36, 12 (2024), 8148–8163.

[9] Reza Curtmola, Juan A. Garay, Seny Kamara, and Rafail Ostrovsky. 2006. Searchable symmetric encryption: improved definitions and efficient constructions. In *CCS*. 79–88.

[10] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David B. Lomet, and Tim Kraska. 2020. ALEX: An Updatable Adaptive Learned Index. In *SIGMOD*. 969–984.

[11] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. 2020. Tsunami: A Learned Multi-dimensional Index for Correlated Data and Skewed Workloads. *Proc. VLDB Endow.* 14, 2 (2020), 74–86.

[12] Haowen Dong, Chengliang Chai, Yuyu Luo, Jiabin Liu, Jianhua Feng, and Chaoqun Zhan. 2022. RW-Tree: A Learned Workload-aware Framework for R-tree Construction. In *ICDE*. 2073–2085.

[13] Jian Gao, Xin Cao, Xin Yao, Gong Zhang, and Wei Wang. 2023. LMSFC: A Novel Multidimensional Index based on Learned Monotonic Space Filling Curves. *Proc. VLDB Endow.* 16, 10 (2023), 2605–2617.

[14] Saheli Ghosh, Tin Vu, Mehrad Amin Eskandari, and Ahmed Eldawy. 2019. UCR-STAR: the UCR spatio-temporal active repository. *ACM SIGSPATIAL Special* 11, 2 (2019), 34–40.

[15] Paul Grubbs, Richard McPherson, Muhammad Naveed, Thomas Ristenpart, and Vitaly Shmatikov. 2016. Breaking Web Applications Built On Top of Encrypted Data. In *CCS*. 1353–1364.

[16] Tu Gu, Kaiyu Feng, Gao Cong, Cheng Long, Zheng Wang, and Sheng Wang. 2023. The RLR-Tree: A Reinforcement Learning Based R-Tree for Spatial Data. *Proc. ACM Manag. Data* 1, 1 (2023), 63:1–63:26.

[17] Yu Guo, Hongcheng Xie, Cong Wang, and Xiaohua Jia. 2022. Enabling Privacy-Preserving Geographic Range Query in Fog-Enhanced IoT Services. *IEEE TDSC* 19, 5 (2022), 3401–3416.

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

[19] Pengfei Li, Hua Lu, Rong Zhu, Bolin Ding, Long Yang, and Gang Pan. 2023. DILI: A Distribution-Driven Learned Index. *Proc. VLDB Endow.* 16, 9 (2023), 2212–2224.

[20] Zheli Liu, Siyi Lv, Jin Li, Yanyu Huang, Liang Guo, Yali Yuan, and Changyu Dong. 2022. EncodeORE: Reducing Leakage and Preserving Practicality in Order-Revealing Encryption. *IEEE TDSC* 19, 3 (2022), 1579–1591.

[21] Yongping Luo, Peiquan Jin, Zhaole Chu, Xiaoliang Wang, Yigui Yuan, Zhou Zhang, Yun Luo, Xufei Wu, and Peng Zou. 2024. Morphtree: a polymorphic main-memory learned index for dynamic workloads. *VLDB J.* 33, 4 (2024), 1065–1084.

[22] Yinbin Miao, Lin Song, Xinghua Li, Hongwei Li, Kim-Kwang Raymond Choo, and Robert H. Deng. 2024. Privacy-Preserving Arbitrary Geometric Range Query in Mobile Internet of Vehicles. *IEEE TMC* 23, 7 (2024), 7725–7738.

[23] Yinbin Miao, Guijuan Wang, Xinghua Li, Hongwei Li, Kim-Kwang Raymond Choo, and Robert H. Deng. 2025. Efficient and Secure Geometric Range Search Over Encrypted Spatial Data in Mobile Cloud. *IEEE TMC* 24, 3 (2025), 1621–1635.

[24] Yinbin Miao, Yutao Yang, Xinghua Li, Zhiquan Liu, Hongwei Li, Kim-Kwang Raymond Choo, and Robert H. Deng. 2023. Efficient Privacy-Preserving Spatial Range Query Over Outsourced Encrypted Data. *IEEE TIFS* 18 (2023), 3921–3933.

[25] Yinbin Miao, Yutao Yang, Xinghua Li, Linfeng Wei, Zhiquan Liu, and Robert H. Deng. 2024. Efficient Privacy-Preserving Spatial Data Query in Cloud Computing. *IEEE TKDE* 36, 1 (2024), 122–136.

[26] Yuxuan Mo and Yu Hua. 2025. LOFT: A Lock-free and Adaptive Learned Index with High Scalability for Dynamic Workloads. In *EuroSys*. 475–491.

[27] Moin Hussain Moti, Panagiotis Simatis, and Dimitris Papadias. 2022. Waffle: A Workload-Aware and Query-Sensitive Framework for Disk-Based Spatial Indexing. *Proc. VLDB Endow.* 16, 4 (2022), 670–683.

[28] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning Multi-Dimensional Indexes. In *SIGMOD*. 985–1000.

[29] Sachith Pai, Michael Mathioudakis, and Yanhao Wang. 2024. WaZI: A Learned and Workload-aware Z-Index. In *EDBT*. 559–571.

[30] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. 2019. Arx: An Encrypted Database using Semantically Secure Encryption. *Proc. VLDB Endow.* 12, 11 (2019), 1664–1678.

[31] Yuya Sasaki. 2022. A Survey on IoT Big Data Analytic Systems: Current and Future. *IEEE IOT* 9, 2 (2022), 1024–1036.

[32] Shuai Shang, Xiong Li, Rongxing Lu, Jianwei Niu, Xiaosong Zhang, and Mohsen Guizani. 2022. A Privacy-Preserving Multidimensional Range Query Scheme for Edge-Supported Industrial IoT. *IEEE IOT* 9, 16 (2022), 15285–15296.

[33] Yufan Sheng, Xin Cao, Yixiang Fang, Kaiqi Zhao, Jianzhong Qi, Gao Cong, and Wenjie Zhang. 2023. WISK: A Workload-aware Learned Index for Spatial Keyword Queries. *Proc. ACM Manag. Data* 1, 2 (2023), 187:1–187:27.

[34] Lili Sun, Yonggang Zhang, Yandong Zheng, Weiyu Song, and Rongxing Lu. 2023. Towards Efficient and Privacy-Preserving High-Dimensional Range Query in Cloud. *IEEE TSC* 16, 5 (2023), 3766–3781.

[35] Pengxu Tian, Cheng Guo, Kim-Kwang Raymond Choo, Xinyu Tang, and Lin Yao. 2023. A Privacy-Preserving Hybrid Range Search Scheme Over Encrypted Electronic Medical Data in IoT Systems. *IEEE IOT* 10, 17 (2023), 15314–15324.

[36] Qiuyun Tong, Yinbin Miao, Hongwei Li, Ximeng Liu, and Robert H. Deng. 2023. Privacy-Preserving Ranked Spatial Keyword Query in Mobile Cloud-Assisted Fog Computing. *IEEE TMC* 22, 6 (2023), 3604–3618.

[37] Ning Wang, Yaohua Wang, Zhigang Wang, Jie Nie, Zhiqiang Wei, Peng Tang, Yu Gu, and Ge Yu. 2023. PrivNUD: Effective Range Query Processing under Local Differential Privacy. In *ICDE*. IEEE.

[38] Peng Wang and Chinya V. Ravishankar. 2013. Secure and Efficient Range Queries on Outsourced Databases using Rp-trees. In *ICDE*. 314–325.

[39] Tianhao Wang, Bolin Ding, Jingren Zhou, Cheng Hong, Zhicong Huang, Ninghui Li, and Somesh Jha. 2019. Answering Multi-Dimensional Analytical Queries under Local Differential Privacy. In *SIGMOD*. 159–176.

[40] Xiangyu Wang, Jianfeng Ma, Feng Li, Ximeng Liu, Yinbin Miao, and Robert H. Deng. 2021. Enabling Efficient Spatial Keyword Queries on Encrypted Data With Strong Security Guarantees. *IEEE TIFS* 16 (2021), 4909–4923.

[41] Xiangyu Wang, Jianfeng Ma, Ximeng Liu, Robert H. Deng, Yinbin Miao, Dan Zhu, and Zhuoran Ma. 2020. Search Me in the Dark: Privacy-preserving Boolean Range Query over Encrypted Spatial Data. In *INFOCOM*. 2253–2262.

[42] Xiangyu Wang, Jianfeng Ma, Ximeng Liu, Yinbin Miao, Yang Liu, and Robert H. Deng. 2023. Forward/Backward and Content Private DSSE for Spatial Keyword Queries. *IEEE TDSC* 20, 4 (2023), 3358–3370.

[43] Jiacheng Wu, Yong Zhang, Shimin Chen, Yu Chen, Jin Wang, and Chunxiao Xing. 2021. Updatable Learned Index with Precise Positions. *Proc. VLDB Endow.* 14, 8 (2021), 1276–1288.

[44] Songrui Wu, Qi Li, Guoliang Li, Dong Yuan, Xingliang Yuan, and Cong Wang. 2019. ServeDB: Secure, Verifiable, and Efficient Range Queries on Outsourced Database. In *ICDE*. 626–637.

[45] Jianyu Yang, Tianhao Wang, Ninghui Li, Xiang Cheng, and Sen Su. 2020. Answering Multi-Dimensional Range Queries under Local Differential Privacy. *Proc. VLDB Endow.* 14, 3 (2020), 378–390.

[46] Chuan Zhang, Liehuang Zhu, Chang Xu, Jianbing Ni, Cheng Huang, and Xuemin Shen. 2022. Location Privacy-Preserving Task Recommendation With Geometric Range Query in Mobile Crowdsensing. *IEEE TMC* 21, 12 (2022), 4410–4425.