Enabling Index-free Adjacency in Oblivious Graph Processing with Delayed Duplications

Weiqi Feng
University of Massachusetts Amherst
weiqifeng@umass.edu

Xinle Cao*
OceanBase, Ant Group
caoxinle.cxl@antgroup.com

Adam O'Neill
University of Massachusetts Amherst
adamoneill@umass.edu

Chuanhui Yang
OceanBase, Ant Group
rizhao.ych@oceanbase.com

ABSTRACT

Obliviousness has been regarded as an essential property in encrypted databases (EDBs) for mitigating leakage from access patterns. Yet despite decades of work, practical oblivious graph processing remains an open problem. In particular, all existing approaches fail to enable the design of *index-free adjacency* (IFA), i.e., each vertex preserves the physical positions of its neighbors. However, IFA has been widely recognized as necessary for efficient graph processing and is fundamental in native graph databases (e.g., Neo4j).

In this work, we propose a core technique named *delayed duplication* to resolve the conflict between IFA and obliviousness. To the best of our knowledge, we are the first to address this conflict with both practicality and strict security. Based on the new technique, we utilize elaborate data structures to develop a new EDB named Grove for processing expressive graph queries. The experimental results demonstrate that incorporating IFA makes Grove impressively outperform the state-of-the-art work across multiple graph-processing tasks, such as neighbor query and t-hop query.

PVLDB Reference Format:
Weiqi Feng, Xinle Cao, Adam O'Neill, and Chuanhui Yang. Enabling Index-free Adjacency in Oblivious Graph Processing with Delayed Duplications. PVLDB, 19(8): 1826 - 1839, 2026.
doi:10.14778/3811243.3811255

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

1 INTRODUCTION

Encrypted databases (EDBs) [26, 33, 62, 64, 73] have emerged as a promising solution in cloud computing, particularly as database outsourcing becomes increasingly popular [35, 85]. By leveraging the computational power and storage capacity of remote servers, the client can manage its databases more efficiently and conveniently. However, ensuring data confidentiality during outsourcing is critical, as the client must prevent untrusted servers from accessing or inferring sensitive information. EDBs address this challenge by enabling queries to be processed directly over encrypted data, thereby preserving both functionality and security. Over the past two decades, EDB systems have seen significant progress [19, 44, 56, 62, 64]. Despite these advancements, *access patterns*, i.e., the access frequency and order of items, can still reveal substantial information about underlying plaintexts stored in EDBs, evidenced by a long line of attacks [23, 34, 37, 39, 45, 46]. This vulnerability has underscored the need for oblivious algorithms [21, 42, 57], which ensure the untrusted server learns only query type and the size of query results, without revealing any other sensitive information from access patterns during query execution. Consequently, oblivious query processing has become a key requirement for strengthening the security guarantees of EDBs.

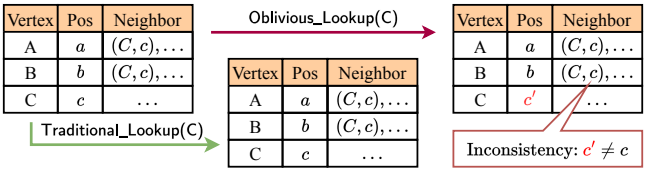


Figure 1: The conflict between index-free design and oblivious algorithms. Obliviousness requires the position of C to be changed after visiting C, thus incurs inconsistency.

1.1 The Conflict in Oblivious Graph

While obliviousness has been a central topic [14, 16, 21, 42] in the community, designing efficient oblivious algorithms for various EDB functionalities remains challenging. One particularly recent and pressing issue is enabling oblivious query processing over graph-structured data [4, 14, 84, 86]. As highlighted by Appan et al. [4], all existing approaches do not realize *index-free adjacency* (IFA) [5, 60], a fundamental property of native graph databases such as Neo4j [59] that is crucial for efficient graph processing.

Specifically, IFA requires each vertex to directly preserve the physical positions of its neighbors. In this way, the client can retrieve the neighbors of a given vertex at only constant cost. However, these stored positions create a conflict between IFA and oblivious algorithms, even on very simple graphs. Consider an undirected graph with only three vertices $\{A, B, C\}$ where C is a neighbor of both A and B. Under IFA, both A and B store the position of C to enable efficient neighbor retrieval. We illustrate the resulting conflict using this simple graph in Figure 1:

*Xinle Cao is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 19, No. 8 ISSN 2150-8097.
doi:10.14778/3811243.3811255

Table 1: Comparison between Grove and prior works on oblivious graph.

Solution	Obliviousness	Scenarios		Neighbor Query	
		Client/Server	Enclave-based	Index-free	Tool/Technique used
Opaque [84]	○	–	✓	✗	*complex* relational databases
Compass [86]	◐	✓	–	✗	*linear* client-side storage
GraphOS [14]	●	–	✓	✗	*expensive* oblivious map
AHR [4]	○	✓	–	✓	smart pointer for *limited* programs
Grove	◐	✓	–	✓	*secure and practical* delayed duplication

[a] ●= Double obliviousness, ◐= Obliviousness, ○= Obliviousness with some additional leakages
[b] Double obliviousness [14, 56] implies the enclave-based EDBs achieve obliviousness in both server's storage and hardware enclaves.

- **Non-oblivious algorithms** do not necessarily change the positions of vertices. When the client retrieves C in traditional non-oblivious ways, it reads C and then rewrites it, leaving C's position unchanged. Consequently, the position of C preserved inside A and B remains valid and can still be used to locate C.
- **Oblivious algorithms** in contrast, require updating a vertex's position whenever it is visited. As a result, if the client executes an oblivious lookup on C, it updates C's position in this process for obliviousness. The updated position is a fresh random number and thus can be inconsistent with the one stored inside A and B.

1.2 The Shortage of Existing Solutions

The conflict between IFA and obliviousness actually has motivated plenty of works [6, 14, 78, 84, 86] to pursue alternative solutions for oblivious graph processing. Nevertheless, the absence of IFA makes most of them suffer from serious practicality issues. To date, only Appan et al. [4] attempt to realize IFA in oblivious graphs for limited programs. We outline the features and limitations of representative approaches; a summary appears in Table 1.

- *Opaque* [84] supports graph processing via oblivious relational databases. While such a conversion is common in databases [76], achieving it efficiently and obliviously is complex. As noted by [14], Opaque has to execute costly oblivious joins [42] to support BFS traversal and leaks unexpected sensitive information.
- *Compass* [86] targets the well-known HNSW algorithm [53], a graph-based approximate nearest neighbor search solution. It supports oblivious graph processing by having the client preserve the position of each vertex locally. This design yields a client-side storage linear in the number of vertices, which is clearly impractical for very large databases and multi-client settings [48].
- *GraphOS* [14] is an *enclave-based* EDB presented at VLDB'24 for oblivious graph processing. In its design, each vertex stores neighbor identifiers as a global index, and an oblivious map (OMAP) maps identifiers to positions. As a result, the client retrieves neighbors via OMAP, incurring substantial overhead.
- *AHR protocol* refers to the recent work of Appan, Heath, and Ren [4]. It remarkably enables an index-free design via a novel technique named *smart pointer*. Nevertheless, it provides obliviousness only for programs that utilize the same number of memory accesses and can reveal additional information when applied to process some graph queries, e.g., a single vertex lookup.

These drawbacks motivate new techniques that support IFA and oblivious graph processing with strong security and practical performance. We therefore propose an EDB that ❶ is graph-specific; ❷ requires only sublinear client-side storage; ❸ maintains the IFA design; ❹ provides strict obliviousness.

1.3 Our Contributions

This work aims to enable IFA in oblivious graph processing to improve performance without compromising strict obliviousness guarantees. To tackle the inherent conflict outlined in Section 1.1, we propose a core technique called *delayed duplication*, which allows the client to update vertex positions in an oblivious and practical manner. Building on this technique, we design Grove (**G**raph **R**etrieval with **O**bliviousness, **V**ersatility, and **E**fficiency), a new system for oblivious graph processing. It delivers substantial performance improvements over GraphOS and provides stronger security guarantees than the AHR protocol. Our main contributions are:

- **In Section 3, we introduce *delayed duplication*, a new technique that supports position updates for IFA.** It adopts small meta blocks to notify all neighbors of a vertex the updated positions when the vertex is visited. We derive a practical bound for the allocation of storage to these meta blocks. Unlike prior works [4, 14], this technique eliminates the need for the expensive OMAP and avoids relaxing security guarantees.
- **In Section 4, we develop a new EDB system,** Grove**, for practical oblivious graph processing**. To integrate the delayed duplication technique, we carefully design the system's data structures and algorithms to support efficient basic operations and graph-specific queries, particularly graph traversal tasks such as neighbor query and t-hop query.
- **In Section 6, we evaluate** Grove **alongside the SOTA EDB systems GraphOS [14] for oblivious graph processing.** Experimental results demonstrate that Grove achieves significant performance improvements in both fundamental and graph-specific queries. We have open-sourced implementations of Grove and GraphOS under the standard client/server paradigm at the repository https://github.com/weiqins/daoram for future research.

2 BACKGROUND

This section first provides an overview of the threat model and related work, then introduces the foundational preliminaries.

2.1 Threat Model

In this work, we consider the typical threat model in EDBs [4, 12, 54]. The adversary $\mathcal{A}$ is assumed to be *honest-but-curious*, i.e., it honestly executes predefined protocols but is curious about sensitive information. It therefore attempts to infer database contents from its observations. Regarding capabilities, $\mathcal{A}$ can compromise the server

for extended periods to observe the encrypted data and query execution, as modeled by an untrusted cloud server. A significant point recently concerned is whether $\mathcal{A}$ can inject queries [2, 33, 52]. While disallowing $\mathcal{A}$ to inject queries may bring new performance gain on KV stores [52] and even other types of databases [2], we follow traditional oblivious graph works [4, 14, 84] in permitting this ability and treat these works as comparison objects. Generally, we aim to achieve the following security goals:

- **Obliviousness on single vertex:** For each vertex, we require it to obliviously *preserve and update* the positions of its neighbors for enabling IFA with both security and correctness.
- **Obliviousness on graphs:** With oblivious IFA on each vertex, we aim to achieve obliviousness on complete graphs. While the allowed leakage profile can vary from graph queries, we always provide a security guarantee no weaker than that in GraphOS [14] under the *client/server model*.

We use semantically secure encryption for the database and queries. Each retrieved item is re-encrypted with fresh randomness. Consistent with [2, 11, 16], we exclude the timing attacks and malicious adversaries. This work follows many prior works [10, 16, 28, 71] in adopting the client/server model, where the client and server communicate over realistic networks. In this setting, network latency makes interaction rounds the primary bottleneck [77]. An important assumption [4, 10, 11] we adopt under this model is the client-side storage cannot scale *linearly* to the graph for practicality.

2.2 Preliminary

Notations. We use $|I|$ to denote the cardinality of a given set I. For a complete binary tree of height L, the root is at level 0, and the leaves are at level $L-1$. The client and server are denoted as C and $\mathcal{S}$, respectively. For graph notations, we express a graph as $G = (V, E)$ where V and E separately signify the vertex and edge set. We generally consider undirected graphs as it is natural to support directed graphs by marking the direction of edges.

Oblivious RAM (ORAM). The foundational mechanism we rely on is ORAM, which is originally proposed to achieve obliviousness on accessing memory [32]. It can be logically considered as an oblivious key-value (KV) store where keys are consecutive integers. We take Path ORAM, one of the most popular ORAMs in EDBs [11, 16, 21], as our underlying mechanism. Informally, to implement this obliviousness mechanism for n items, the client C first outsources a complete binary tree to the server $\mathcal{S}$ with the following properties:

- The binary tree, also refer as the ORAM tree, contains at least n leaf nodes, with its height denoted by l, where $l \geq \lceil \log_2 n \rceil + 1$.
- Each node in the tree is called a *bucket* and consists of Z blocks. Each block within a bucket stores either an encrypted item r from C's database or some random strings.
- Each item r associated with a key k is assigned a path number $pn \leftarrow\$ \{0, 1, ..., 2^l - 1\}$, and the ciphertext of (k, r, pn) is stored along *the path from the root node to the pn-th leaf node.*

Here we briefly explain how the ORAM tree is used to obliviously access the item r associated with k and pn with three steps.

(1) *Retrieval:* C retrieves all the blocks along the path corresponding to pn. Then it decrypts these blocks, extracts the items, and stores them locally. Upon locating the item r from all local items, C performs the desired operation on this item and samples a new path number pn' for it.
(2) *Eviction:* C rearranges all local items to blocks in the path retrieved above such that they are as close as possible to the leaf node associated with their path numbers. For items which cannot be placed in the blocks, C temporarily stores them in a local area named *stash* and tries to arrange them in the future.
(3) *Upload*: After eviction, C fulfills empty retrieved blocks with random strings and then encrypts all retrieved blocks with new randomness. C uploads these encrypted blocks to $\mathcal{S}$ and $\mathcal{S}$ uses them to cover the path corresponding to pn.

After the access, the item r is assigned with a new path number pn'. Its ciphertext is stored in either the path associated to pn' or stash.

Oblivious Map (OMAP). The other oblivious primitive we use is OMAP, which can be thought as an extension of ORAM. It supports *arbitrary* KV stores and more complex operations, e.g., insertions. The typical OMAP constructions [21, 79] organize KV stores as a search tree like AVL tree. The OMAP functionalities are then achieved via oblivious operations on the tree. For instance, C obliviously traverses the search tree for oblivious lookup. In detail, each tree node is treated as an item to be stored in the ORAM tree. It preserves both the keys and path numbers of its children. So when C downloads a tree node, it can identify where to retrieve the children of this node and further complete the tree traversal. We illustrate this storage form in Figure 2 and remark such a form will be our standard to store graph vertices under IFA, i.e., all vertices are stored within the ORAM tree and a vertex preserves the keys and path numbers of its neighbors. This enables C to efficiently download the neighbors of a given vertex.

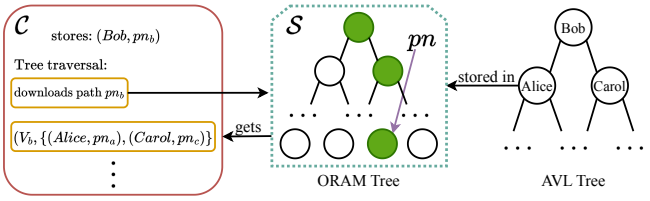


Figure 2: Example of oblivious AVL tree. Each tree node is stored in the ORAM tree randomly. For tree traversal, C visits a node (e.g., root node Bob) to get the keys and path number of its children (e.g., $\{(Alice, pn_a), (Carol, pn_c)\}$).

3 DELAYED DUPLICATION

In this section, we identify the key problem our work addresses: enabling IFA in oblivious graph processing. We then introduce our core technique, *delayed duplication*, to resolve this challenge. This technique forms the foundation of our proposed EDB Grove.

3.1 Problem Statement

We first outline the problem context in oblivious graph processing. As introduced in Section 2.2, most existing EDBs achieve obliviousness via adopting an ORAM tree as the underlying mechanism. To process graph data, each vertex (and even edge) is treated as one item to be placed in a path determined by its assigned random path

number pn. More importantly, whenever a vertex is visited, C updates its pn to a new random value and places it on the corresponding path. This mechanism guarantees any access to a vertex/edge just operates a random path in the view of the passive adversary, providing the requested obliviousness.

Index-free Adjacency (IFA). To accelerate the efficiency of graph processing, a popular design in graph databases [59, 63, 83] is to adopt IFA for vertices. That is, each vertex directly preserves both the identifiers and storage positions of its neighbors to enable finding neighbors at the cost of *constant* complexity. To transfer such a design in oblivious graph, besides identifiers, each vertex preserves assigned path numbers of its neighbors as their storage positions.

However, it can be problematic when adopting IFA in oblivious graphs. This is because non-oblivious graphs do not need to change the positions during visiting vertices while oblivious graphs have to update the positions. In detail, consider C visits a vertex v in oblivious graph and updates its position. As all v's neighbors preserve the position of v, now C has to address the following problem:

How to update the position of v preserved by the neighbors with both efficiency and obliviousness?

A straightforward approach [86] is to store only the identifiers of neighbors within each vertex, while C maintains a global mapping from identifiers to vertex positions. For instance, in a social network, when C obliviously retrieves a vertex with attribute Name = Alice and discovers a neighbor with Name = Bob, it locally queries the position associated with Bob to obtain the target path number, and then successfully downloads the corresponding vertex. However, this method incurs $O(|V|)$ client-side storage overhead for a graph $G = (V, E)$, which becomes impractical for large-scale graphs. In contrast, GraphOS [14] moves the identifier-to-position mapping to the server side and enables C to access it obliviously via OMAP, thereby preserving obliviousness while significantly reducing client storage. Nevertheless, this improvement comes at the cost of efficiency, as OMAP remains an expensive primitive [10].

3.2 Delayed Duplication

In this section, we present the design of *delayed duplication*, detailing its core components step by step to demonstrate how it enables IFA with practical efficiency.

Example and Intuition. To illustrate our technique, consider a simple scenario: a vertex v_0 has d neighbors $V_1 = \{v_1, \ldots, v_d\}$, where $d \geq 2$. These vertices are randomly stored in the ORAM tree, and each vertex in V_1 maintains the identifier and current position of v_0, i.e., $v_0.pn$. After v_0 is visited by the client C, its position is updated for obliviousness, denoted as $v_0.pn \rightarrow v_0.pn'$. To maintain position consistency, the stored references to v_0's position within all vertices in V_1 must also be updated. A naive approach would be to download and modify all vertices in V_1 to reflect the new position. However, this solution is impractical for two reasons:

- *Bandwidth blowup:* This approach increases bandwidth usage by d times, as now C must download $d + 1$ vertices rather than one.
- *Cascading inconsistency:* Since downloading each vertex in V_1 also triggers its own position update (for obliviousness), new inconsistencies are introduced for each downloaded neighbor, leading to a recursive consistency maintenance problem.

Motivated by the above shortcomings, we address the position consistency in another new perspective. Instead of downloading all vertices in V_1 to update their references to v_0's new position, we only need to **notify** them of the update $v_0.pn \rightarrow v_0.pn'$. Specifically, for each neighbor v_i with assigned path number pn_i, we create a new **small** block containing the message $\{v_0.pn \rightarrow v_0.pn'\}$, and assign its path number as pn_i to guarantee it will be evicted into the path corresponding to pn_i in the ORAM tree. Consequently, when C later visits v_i by retrieving all blocks on path pn_i, it can detect this message block for notification and update v_i's stored reference to v_0's position accordingly. This strategy avoids downloading any vertex in V_1 during the notification phase, requiring only the insertion of some small message blocks. We refer to these blocks as *delayed duplications*, as they actually delay the position update by duplicating v_0's position information within the tree.

With the new fundamental insight on intuition, we now introduce the concrete technical challenges on constructions as follows. We will address them one by one to complete our constructions.

C1. **Limited tree capacity**: The size of ORAM tree is originally set according to the number of vertices/edges. As new inserted auxiliary blocks, delayed duplications raise concerns about bucket overflows and an increase in tree size.

C2. **Communication overhead**: The insertion of delayed duplications towards ORAM tree can increase both the number of interaction rounds and the overall bandwidth usage.

C3. **Duplication management**: Repeated visits to the same vertex can produce multiple delayed duplications for each of its neighbors, which must be managed efficiently to ensure both correctness and practical feasibility.

3.2.1 Data Structure (C1). C1 results from that duplications require additional storage as they are used to store the update information about vertices, but bucket and blocks are initialized for storing only vertices/edges in the setup. To avoid sacrificing much additional storage costs on the duplications, we utilize an important fact that a duplication requires much less information to be recorded compared to a real vertex since the duplication preserves only the position update information. To this end, we adopt the *meta-block* design [7, 66, 72], adding Y smaller blocks to each bucket alongside the Z regular blocks. For clarity, we refer to the regular blocks as *data blocks* and the smaller ones as *meta blocks*. Vertices are stored in data blocks, while delayed duplications are stored in meta blocks. We describe the detailed storage form of vertices and duplications as following to illustrate the storage utilization:

- **Vertex form:** In an undirected graph, a vertex v should maintain its information as the main contents, i.e., its identifier $v.\mathrm{id}$, position $v.pn$, and attribute values $v.value$. Then for each of its neighbors, v maintains the identifier and position of this neighbor for IFA, and also the edge between them for weighted graph. Accordingly, the data structure for vertex v is defined as follows:
$$\left\{v.\mathrm{id}, \left(v.pn, v.value, \{(g_i.\mathrm{id}, g_i.pn, g_i.edge)\}_{i=1}^{d}\right)\right\}$$
where g_i denotes the neighbor of v.
- **Duplication form:** When a vertex v is visited, it broadcasts the position update to all its neighbors. In detail, for its neighbor g_i with assigned path number pn_i, the duplication is created as $\{v.\mathrm{id}, v.pn', pn_i\}$ where $v.pn'$ is v's updated position. We do not

store the identifier of g_i since the duplication is valid as long as it is stored along the path corresponding to pn_i. When C downloads path pn_i during query processing, it will detect both this duplication and g_i to update v's position inside g_i.

With description above, it is obvious that the duplications costs much less storage than vertices, especially in applications where each vertex has many attributes such that the attribute values occupy the main storage, e.g., knowledge graph [75].

3.2.2 Duplication Insertion (C2). The introduction of delayed duplications can incur new communication overhead. Consider a vertex v with d neighbors: each vertex access operation generates d delayed duplication to notify the d neighbors with position updates. Recall we adopt Path ORAM as the underlying ORAM, which theoretically allows only *one* duplication to be inserted into the ORAM tree when retrieving and uploading one random path. So the insertion of d duplications implies C needs to retrieve and upload (at least) d random paths, resulting in additional communication overhead. To optimize this for efficient duplication insertion, we implement two key measures: ❶ we notice that duplication insertion can exclusively target the *small* meta blocks, thus we require C to reduce the bandwidth via downloading only meta blocks in the d designated paths for insertion. ❷ we integrate the duplication insertion with regular vertex operations for a piggyback-style [3] insertion. When C retrieves a complete path (containing both data and meta blocks) during visiting a vertex, it simultaneously retrieve meta blocks in d paths within **the same interaction round** to insert duplications. The above design guarantees that the communication overhead, i.e., the costs on bandwidth and interaction rounds, is limited compared with the total costs of query processing.

Remarkably, retrieving d random paths actually does not suffice for our duplication insertion. This is because the assigned path numbers of duplications are not random, while the analysis of Path ORAM aims to only items with uniformly random path numbers. To illustrate, suppose C accesses the same vertex v repeatedly for μ times. For its neighbor g_i located at path pt_i, C must notify g_i of the position updates with μ duplications. Moreover, all the μ duplications are inserted towards the same path pt_i, which is basically different from the randomized mechanism in Path ORAM. Consequently, retrieving d random paths provides no theoretical guarantee against *overflow*—i.e., the absence of empty meta blocks in the target path to accommodate new duplications. To ensure a rigorous overflow bound, we adopt the aggressive eviction strategy from [29, 66]: C selects the d paths in **reverse lexicographical order** instead of choosing randomly. Combined with the de-duplication strategy detailed in the next section, this enables us to derive a practical upper bound on Y (i.e., the number of meta blocks for each bucket) to rule out the overflow.

3.2.3 De-duplication Strategy (C3). The final and most critical step is de-duplication. When a vertex v is visited multiple times, C generates multiple duplications for the same neighbor, all of which are inserted towards the same path. Without an effective de-duplication, overflow becomes highly likely due to the limited capacity of meta blocks. We formalize the objectives of de-duplication below:

(1) **Utilization**: minimize the number of redundant duplications for the same neighbor within a path;
(2) **Correctness**: ensure that only the most recent duplication for a given neighbor remains valid.

A duplication $\{v.\mathrm{id}, v.pn', pn_i\}$ can be uniquely identified by the tuple $(v.\mathrm{id}, pn_i)$, referred as identifier tuple. All such duplications target the neighbors of v located in path pn_i, but carry position updates in different time points. Since only the latest update is truly valid, C can remove any prior duplications sharing the same identifier tuple, thereby improving space utilization. To determine update order without additional information, we avoid explicit timestamps (which would increase meta block size) and instead leverage the structural properties of the ORAM tree: we treat the *level* of a duplication within the ORAM tree as its logical timestamp, and always identify the one closest to the root as the valid one. Note that the latest duplication is inserted last and thus exactly resides at the shallowest level (closest to the root), which ensures the correctness of our de-duplication.

Furthermore, extensive de-duplication can be performed when C downloads an entire path including data and meta blocks. This full-path access enables C to consolidate all update information and notify them to the corresponding neighbor vertices, after which all duplications targeting that path can be safely removed.

3.3 Overflow Analysis

In this section, we provide both a theoretical bound and practical guideline for setting the meta-block capacity Y to control overflow.

3.3.1 Theoretical Bound. We begin by presenting a formal theorem that establishes an upper bound on the overflow probability under specific application parameters. This allows us to choose Y rigorously such that the overflow probability is negligible in the security parameter λ. For example, when $\lambda = 128$, the overflow probability is less than 2^{-128}!

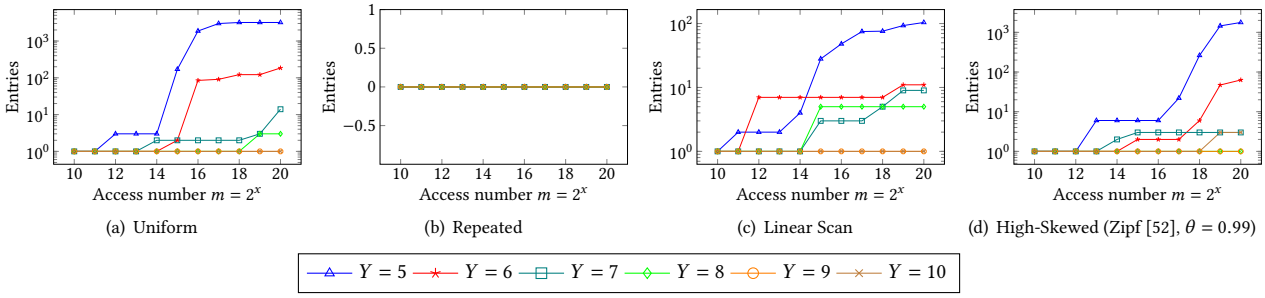
Theorem 3.1. *Suppose that, on input n items, C initializes a height-L ORAM tree where each bucket is associated with Y meta blocks. Assume each access to the ORAM tree (the retrieval and upload of one path) incurs exactly d new duplications, and d additional paths are chosen via the RL path selection for inserting the duplications. Then, after m accesses, the probability of overflow is bounded by:*

$$\sum_{j=\sigma}^{L-1} 2^j \left\lceil \frac{md}{2^j} \right\rceil \left[1 - \sum_{i=0}^{\lfloor Y/d \rfloor} \binom{2^j}{i} \left(\frac{1}{2^{j+1}}\right)^i \left(1 - \frac{1}{2^{j+1}}\right)^{2^j - i} \right],$$

where σ is the smallest integer such that $2^\sigma > Y$.

We defer the proof of this theorem to the full version [25]. To illustrate its practical implications, we give some concrete examples. Table 2 lists the parameters used to apply delayed duplication to real-world datasets. For each dataset, we configure the ORAM based on its maximum vertex degree and size, and set the number of vertex accesses to $m = |V|$ to compute the required capacity Y . The results show that the storage overhead introduced by our technique (column **Pro**) remains modest even under a negligible overflow probability. A more comprehensive evaluation, including bandwidth and runtime, will be provided in Section 6.

3.3.2 Practical Guideline. Actually, a small amount of overflow is permissible in practice. For example, Path ORAM allows $O(\log n)$ overflowed items to reside in the client-side stash. Similarly, the



(a) Uniform (b) Repeated (c) Linear Scan (d) High-Skewed (Zipf [52], $\theta = 0.99$)

Figure 3: Maximal duplications inside the stash for meta blocks ($|V| = 2^{15}, |E| = 10|V|$).

Table 2: Meta-block overhead on real-world graphs.

Dataset	$\|V\|$	Data	Meta	Cap	Pro
Enron [41]	36,692	12 KB	1.5 KB	70	11.1%
Amazon [47]	548,552	88 KB	1.2 KB	50	1.35%
Genome [43]	108,077	1.2 MB	5.4 KB	90	0.45%

Data/**Meta**: storage per bucket (data and meta blocks, respectively).
Cap: number of meta blocks allocated per bucket.
Pro: percentage of meta-block storage relative to total bucket storage.

client C can temporarily store overflowed duplications for meta blocks in its stash, enabling much smaller values of Y. This introduces a tunable trade-off between client-side memory usage and server-side storage overhead (and correspondingly, bandwidth). By relaxing the requirement of negligible overflow probability, we can adopt more practical parameter choices.

To empirically evaluate this trade-off, we simulate delayed duplication on a synthetic circulant graph with 2^{15} vertices, where each vertex has exactly 10 neighbors. All vertices are stored in an ORAM tree with padded meta blocks. We measure the number of duplications accumulated in the client's stash for meta blocks under varying values of Y and different access distributions over 2^{20} accesses. The results in Figure 3 show that as Y increases, the number of duplications in the stash dramatically decreases. Notably, when $Y = 10$, the stash contains only 0 or 1 duplication! In contrast, our theoretical bound predicts up to 330 duplications due to the negligible probability requirement and our relaxation to Theorem 3.1 (as detailed in [25]). Despite using the theoretical bound in our deployment, our Grove still outperforms the SOTA system GraphOS. We provide more experimental results about setting the practical values of Y on larger graphs in the appendix of [25].

We further remark that the overflow behavior of delayed duplication differs significantly from that of Path ORAM. Specifically, overflow is the most frequent under the uniform access distribution for delayed duplications. For instance, when $Y = 5$, the client stash accumulates 3195 entries under a uniform distribution, but only 1774 under the standard high-skewed Zipf distribution ($\theta = 0.99$) after 2^{20} accesses. It is exactly the complex behavior of duplications that incurs the new amazing observation and much relaxed theoretical bound.

4 GROVE: GRAPH PROCESSING

As demonstrated by native graph databases (particularly *Neo4j* [59]), IFA is a fundamental principle that enables efficient preservation and traversal of relationships between vertices. Our technique, *delayed duplication*, is the first to realize IFA in the context of oblivious graph databases. In this section, we leverage this capability to design Grove, an EDB for processing complete graph queries in an oblivious manner. We particularly highlight its advantages in handling neighbor queries and graph traversal tasks, which benefit most significantly from the IFA design.

Design Goal. When presenting *delayed duplication* in Section 3.2, we assumed C already has the content of a target vertex in order to obliviously retrieving its neighbors from an ORAM tree. However, we are yet to address: *how can C first download the target vertex obliviously?* This step is essential for enabling ORAM-tree based oblivious graph algorithms, but it cannot be done in a practical way yet. For example, Compass [86] requires C to locally preserve the positions of all vertices in the ORAM tree for retrieving them. This requires $O(|V|)$ client-side storage and thus can be infeasible. In this section, our goal is to design new data structures and mechanisms, based on the delayed duplication technique, to enable the oblivious retrieval of any target vertex with practicality, and finally support complete and efficient oblivious graph processing.

Degree Padding. Existing oblivious graph EDBs [14, 86] treat the degree of individual vertex as sensitive information and permit leakage of only the maximum vertex degree in the graph. To achieve this, Grove always processes each vertex as if it has K neighbors, where K denotes the maximum vertex degree in the graph. For instance, when accessing a vertex using delayed duplication, Grove inserts K (real or dummy) duplications even if the vertex has no actual neighbor. Similar degree padding is applied throughout initialization and all query processing phases, such as setting the ORAM block size based on K and always retrieving K (real or dummy) neighbors during a neighbor query.

4.1 Data Structures

Data structures. Grove adopts two ORAM trees ($\mathcal{T}_G, \mathcal{T}_P$) to allow C to download any target vertex and process graph queries:

- **Graph Tree $\mathcal{T}_G$:** This ORAM tree stores all vertices for processing graph queries. To follow IFA design, each vertex preserves the vertex content together with its edge/neighbor information. Besides the edges' description, these information include neighbor's positions in $\mathcal{T}_G$ to enable C directly access the neighbors within $\mathcal{T}_G$. Delayed duplications are applied here to enable IFA.
- **Position tree $\mathcal{T}_P$:** This ORAM tree is used to build an OMAP to store all vertices and their corresponding positions in $\mathcal{T}_G$. It stores much less information than $\mathcal{T}_G$ as it is designed to obtain only vertices' positions in $\mathcal{T}_G$.

Consider a graph query where C begins with target vertex v. Grove first retrieves v's position in $\mathcal{T}_G$ through a lookup on OMAP $\mathcal{T}_P$, followed by fetching the vertex data from $\mathcal{T}_G$ based on the obtained position information. With the fetched vertex, C is able to execute the complete graph query, incorporating delayed duplication and specialized graph algorithms. Notably, $\mathcal{T}_G$ additionally maintains each vertex's corresponding position in $\mathcal{T}_P$. During graph processing operations, updates to vertex positions in $\mathcal{T}_G$ create inconsistency with the positions stored in $\mathcal{T}_P$. To resolve this, we require $\mathcal{T}_G$ to record vertex positions in $\mathcal{T}_P$ such that C can utilize delayed duplications again here to notify $\mathcal{T}_P$ the updated positions.

For ease of deployment, we extract meta blocks from $(\mathcal{T}_G, \mathcal{T}_P)$ as ORAM trees $(\mathcal{M}_G, \mathcal{M}_P)$. This allows C to apply nearly the same logic and interfaces to manage the four ORAM trees instead of additional complex mechanisms to handle meta blocks separately. With this optimization, Grove's architecture is illustrated in Figure 4.

4.1.1 Initialization. During the initialization for graph $G = \{(V, E)\}$, C converts it to KV pairs for $\mathcal{T}_G$ and $\mathcal{T}_P$. For vertex v, denote its positions in $\mathcal{T}_P$ and $\mathcal{T}_G$ as pn^P and pn^G, respectively. Then its pair for $\mathcal{T}_P$ is $(v.\text{id}, \{v.pn^P, v.pn^G\})$ while the pair for $\mathcal{T}_G$ is

$$\left\{v.\text{id}, \left(v.pn^P, v.pn^G, v.value, \{(g_i.\text{id}, g_i.pn^G, g_i.edge)\}_{i=1}^{K}\right)\right\}$$

where K denotes the maximal number of v's neighbors, and g_i denotes the neighbor of v. We pad the neighbor number of each vertex to K to guarantee all pairs in $\mathcal{T}_G$ have the same size. With the KV pairs above, it is trivial to adopt existing algorithms [49, 71] to initialize the ORAM trees and write the pairs into $(\mathcal{T}_P, \mathcal{T}_G)$.

An optional optimization for delayed duplication is the technique proposed in [4], which utilizes newly inserted intermediate vertices to sparsify a graph. For example, if one vertex A preserves 10 edges connected to neighbors $\{B_1, ..., B_{10}\}$, then we can create two new vertices $\{A_1, A_2\}$ as A's new neighbors and separately connect them with $\{B_1, ..., B_5\}$ and $\{B_6, ..., B_{10}\}$. To identify an intermediate vertex in the graph, we set its value to particular strings like "`intermediate vertex`". Consequently, now A preserves only two edges to A_1 and A_2, and each vertex is connected by at most 5 vertices, even if $K = 10$. In the remaining sections, we always denote the maximal number of vertices connected to a vertex after sparsification as D, and we explain the sparsification benefits below.

Under the context of delayed duplications, downloading a vertex needs to produce K duplications and $O(K \log |V|)$ bandwidth costs. Fortunately, the sparsification reduces the duplication number requested to D and now the corresponding bandwidth is only $O(D \log |V'|)$ where V' denotes the vertex set after sparsification.

4.2 Queries on Static Graphs

In this section, we introduce the interfaces for queries on static graphs, especially for graph traversal tasks. We leave the interfaces and extension to dynamic graphs in Section 4.3. The difference between static and dynamic graphs lies only in whether the insertion and deletion of vertices and edges are allowed.

Lookup and Update. We first introduce the basic operations, i.e., lookup and update to vertex and edge. The two basic operations differ only in the client-side process to the target object, so we unify them with the same interface. To operate a vertex v, C first accesses $\mathcal{T}_P$ to obtain v's position in $\mathcal{T}_G$. Then it retrieves v from $\mathcal{T}_G$ and

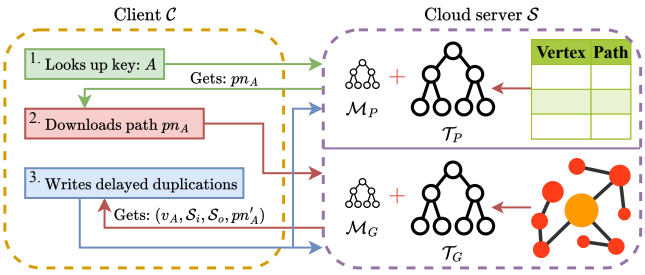


Figure 4: The Complete Architecture of Grove.

possibly locally updates it. To write back v, C updates its positions in both $\mathcal{T}_G$ and $\mathcal{T}_P$. At the same time, C utilizes delayed duplications to notify all v's neighbors the updated position of v in $\mathcal{T}_G$. Recall that v's KV pair for $\mathcal{T}_G$ preserves all neighbors' positions in $\mathcal{T}_G$ such that the duplications can be inserted into correct paths. For edge lookup and update, as the IFA design stores all edge information inside the vertices, C operates an edge by operating the vertices connected by the target edge.

Neighbor Query. Given a neighbor query to find all neighbors of vertex v, C completes it with three steps: ❶ C accesses $\mathcal{T}_P$ to find v's position in $\mathcal{T}_G$; ❷ with the obtained position, C accesses $\mathcal{T}_G$ to download v's KV pair which includes the identifiers and positions of v's neighbors, and then proceeds to the query via downloading all these neighbors from $\mathcal{T}_G$ based on their positions; ❸ finally, C updates the positions of all downloaded vertices (including v and its neighbors), and notify the update information to all related objects in $\mathcal{T}_G$ and $\mathcal{T}_P$ via delayed duplications. As the pair of each vertex for $\mathcal{T}_G$ preserves its positions in $(\mathcal{T}_P, \mathcal{T}_G)$ and neighbors' positions in $\mathcal{T}_G$, C knows the correct paths to insert the duplications.

Recall that intermediate vertices are used to sparsify the graph. Thus given a vertex v for neighbor query, Grove actually executes the following subroutines in step ❷:

(1) Since the graph has been sparsified, so C first retrieves v and its D neighbors in the sparsified graph. Denote these D neighbors as the neighbor set S, then S may include intermediate vertices.
(2) Once C notice intermediate vertices in S, it retrieves all neighbors of intermediate vertices in S. Then C replaces the intermediate vertices in S with these new retrieved vertices.
(3) C repeats step (2) until S include no intermediate vertex, then it treats S as the final and correct neighbor set.

For obliviousness, the above process implies C retrieves a total of $1 + D + \ldots + D^{w-1} + K$ vertices in step ❷ within $w + 1$ interaction rounds, where w is the least number such that $D^w \geq K$.

t-hop Query. The well-known t-hop query [70] is used to retrieve all vertices whose distances from the start vertex are no more than t. To achieve it, we extend it to a series of neighbor queries. Given the start vertex v, C obtains its neighbor set S_1 via a neighbor query. Then C issues neighbor queries to all vertices in S_1 in parallel and denote the new neighbor set of them as S_2. Repeating this process for t times, C gets t sets $\{S_1, S_2, ..., S_t\}$ as the final results.

All the above operations are padded to the worst-case performance for obliviousness. In other words, the access number and orders to the four ORAM trees are always deterministic to the operation type, the values of (D, K), while the sizes of the ORAM trees

are determined by (D, K), the vertex and edge size, and the number of vertices in the sparsified graph denoted by $|V'|$.

Graph Traversal. With all interfaces above, Grove supports general graph queries which can be expressed as the combination of these interfaces. For example, Grove can achieve a variety of graph queries such as fixed-length path query [18], shortest path query in unweighted graphs [68], and even connected component queries [81]. We select t-length traversal [53] as one typical query to implement, which traverses a t-length path in the graph to find a target vertex. Such a task is recently popular as it is the foundation of the famous HNSW algorithm [53]. Clearly, Grove can achieve t-length traversal by t continual (optimized) graph queries.

4.3 Extension to Dynamic Graphs

In this section, we describe insertion and deletion to extend Grove on dynamic graphs. We present them separately because IFA and graph sparsification lead to new differences in them.

Limited Maximal Degrees. Edge insertions and deletions are particularly straightforward when the graph is relatively sparse, i.e., the degree of each vertex is bounded by a small constant. In such cases, Grove does not need intermediate vertices. Thus, edge updates can be efficiently performed by directly modifying the adjacency information of the vertices connected by the target edge. For instance, to delete an edge, the two vertices connected by it are retrieved, and the corresponding edge information are removed from their KV pairs. Similarly, vertex insertion or deletion involves updating the ORAM trees by inserting or removing the vertex itself along with all its incident edges.

Sparsified Graphs. When vertices in the graph have high degrees and trigger sparsification, edge insertions and deletions must account for intermediate vertices. In this context, inserting an edge may actually involve adding it between two intermediate vertices introduced to maintain sparsity. We summarize the procedure for inserting or deleting an edge e between vertices v_1 and v_2 as follows:

(1) C retrieves both v_1 and v_2 from the ORAM trees.
(2) C retrieves all intermediate vertices associated with v_1 and v_2.
(3) C locally determines and modifies the target intermediate vertices for inserting or deleting the edge e.
(4) C writes back all local vertices to the ORAM trees and applies delayed duplications to maintain IFA.

Similarly, vertex insertion or deletion is performed by updating both the vertex itself and all its incident edges in the sparsified structure. Importantly, to ensure obliviousness, C must pad the number of accesses in step (2) so that each vertex appears to have the maximal degree K, regardless of its actual degree. This padding can become prohibitively expensive when vertex degrees vary widely across the graph. Therefore, C is provided with an optional trade-off between security and efficiency: it can choose to leak more degree information besides K to significantly improve the performance of insertions and deletions, e.g., if the actual degree is larger than $K/2$.

5 THEORETICAL ANALYSIS

5.1 Performance Analysis.

The query cost in Grove consists of the accesses on the four ORAM trees especially $(\mathcal{T}_P, \mathcal{T}_G)$ since $\mathcal{M}_P$ and $\mathcal{M}_G$ are designed for the *small* meta blocks and do not occupy independent interaction rounds under our piggyback strategy. Meta-block cost is dataset dependent and typically small relative to total query cost. Moreover, the complexities for meta block costs cannot be explicitly calculated due to the complex relation between the bound Y and $|V|$. Therefore, here we mainly count costs on $(\mathcal{T}_P, \mathcal{T}_G)$ to compare Grove with prior SOTA work named GraphOS [14], we will provide an empirical evaluation of cost on meta blocks in Section 6.

Table 3: Performance of Grove and GraphOS where K is the largest vertex degree in the graph and $\beta = 1+D+\ldots+D^{w-1}+K$.

Operation	EDB	Costs
`Lookup` (Vertex)	GraphOS	1 $\mathcal{T}_{OS}$ call
	Grove	1 $\mathcal{T}_P$ call + 1 $\mathcal{T}_G$ call
`Neighbor query`	GraphOS	$(2K+1)$ $\mathcal{T}_{OS}$ calls
	Grove	1 $\mathcal{T}_P$ call + β $\mathcal{T}_G$ calls
t-`hop query`	GraphOS	$(2(K+K^2+\ldots+K^t)+1)$ $\mathcal{T}_{OS}$ calls
	Grove	1 $\mathcal{T}_P$ call + $(1+K+\ldots+K^{t-1})\beta$ $\mathcal{T}_G$ calls
t-`length traversal`	GraphOS	$(2tK+1)$ $\mathcal{T}_{OS}$ calls
	Grove	1 $\mathcal{T}_P$ call + $\beta \cdot t$ $\mathcal{T}_G$ calls
`Insertion` (Edge)	GraphOS	$(5K+1)$ $\mathcal{T}_{OS}$ calls
	Grove	2 $\mathcal{T}_P$ calls + 2β $\mathcal{T}_G$ calls

Instead of complexity, we use the calls of interfaces on different data structures to decompose the query cost in Grove and GraphOS. This breaks the limitation of specific foundational protocols used. In general, we use three interfaces to decompose the cost:

- *$\mathcal{T}_{OS}$ call* is the operation on the underlying OMAP of GraphOS, whose size is $O(|V|+|E|)$.
- *$\mathcal{T}_P$ call* implies the operation on the OMAP of $\mathcal{T}_P$, whose size is only $O(|V'|)$ where V' is the vertex set in the sparsified graph.
- *$\mathcal{T}_G$ call* indicates an simple ORAM access on $\mathcal{T}_G$ rather than an expensive OMAP operation. And the size of $\mathcal{T}_G$ is $O(|V'|)$.

Specifically, with the OMAP protocol [14] used in GraphOS, an operation on OMAP whose size is $O(n)$, the interaction rounds and bandwidth are $O(\log n)$ and $O(\log^2 n)$, respectively. So we can conclude that an $\mathcal{T}_P$ call is often cheaper than $\mathcal{T}_{OS}$ call as $|V'| < |V|+|E|$ holds in most graphs. In particular, $\mathcal{T}_G$ call is the cheapest, which requests only one interaction roundtrip and $O(\log n)$ bandwidth on an ORAM tree with size $O(n)$.

With the above decomposition, now we list the costs on typical queries of Grove in Table 3. For fairness, we hide the vertex degree information during neighbor queries in both Grove and GraphOS. The complexities can be calculated according to specific OMAP protocols, e.g., with OMAP in [14], the bandwidth for a neighbor query in Grove and GraphOS are $O((2K-1)\log^2(|V|+|E|))$ and $O(\log^2|V'|+\beta\log|V'|)$ where $\beta = 1+D+\ldots+D^{w-1}+K$. Clearly, Grove performs better in most queries as it has only a few $\mathcal{T}_P$ calls and completes graph traversal mainly via the cheap $\mathcal{T}_G$ calls.

5.2 Security Analysis.

We follow the typical security notion in oblivious algorithms [56, 69] and EDBs [62, 67] to define the security of Grove. Intuitively, we require for each operation in Grove, the corresponding query execution leaks nothing beyond the allowed leakage. We describe the specific leakage profiles on the given graph $G = (V, E)$ and query q below and propose the formal security notion in Theorem 5.1.

- *Database leakage $\mathcal{L}_1(G)$* implies the leakages about G. It is revealed during both initialization and query processing, including

❶ the vertex number and edge number in the (sparsified) graph; ❷ the maximal size of vertex and edge; ❸ the maximal vertex degree before and after sparsification, i.e., K and D.

- *Query leakage* $\mathcal{L}_2(q)$ indicates the query type information, which is common in EDBs [74]. In detail, it leaks the query type from { `Lookup, Update, Neighbor query,` t`-hop query,` t`-length traversal` } where the value of t also belongs to leakage.

Our simulation-based security notion follows [13, 62] to require the view of the adversary $\mathcal{A}$ on Grove is simulatable with the permitted leakages. Formally, $\mathcal{A}$ is trying to distinguish the real world and ideal world. Given a graph database G and a sequence of queries $\mathcal{Q} = \{q_1, q_2, ..., q_m\}$. In the real world, $\mathcal{A}$ observes how Grove executes $\mathcal{Q}$ on G in the server side, denoted by $\mathsf{View}^{\mathsf{Grove}}_{\mathcal{A}}(G, \mathcal{Q}, \lambda)$. On the contrary, in the ideal world, a simulator Sim simulates the execution given the leakage $\{\mathcal{L}_1(G), \mathcal{L}_2(\mathcal{Q})\}$ where $\mathcal{L}_2(\mathcal{Q}) := \{\mathcal{L}_2(q_1), ..., \mathcal{L}_2(q_m)\}$. We denote the $\mathcal{A}$'s view in the ideal world as $\mathsf{View}^{\mathsf{Sim}}_{\mathcal{A}}(\mathcal{L}_1(G), \mathcal{L}_2(\mathcal{Q}), \lambda)$. Then the security guarantee holds as the following theorem (the proof is deferred to the full version [25]).

Theorem 5.1 (Grove security). *For any graph database* $G = (V, E)$ *and a sequence of queries* $\mathcal{Q}$*, there exists a polynomial-time (PPT) simulator* Sim *such that the view of any PPT adversary* $\mathcal{A}$ *in the real world and ideal world are computationally indistinguishable:*

$$\mathsf{View}^{\mathsf{Grove}}_{\mathcal{A}}(G, \mathcal{Q}, \lambda) \overset{c}{\equiv} \mathsf{View}^{\mathsf{Sim}}_{\mathcal{A}}(\mathcal{L}_1(G), \mathcal{L}_2(\mathcal{Q}), \lambda).$$

6 EXPERIMENTAL EVALUATION

This section presents a comprehensive evaluation of Grove. We design experiments to answer the following questions:

(1) What overhead do delayed duplications introduce in order to enable the IFA design in oblivious graphs?
(2) Does Grove outperform the SOTA work GraphOS [14] in multiple workloads, including:
   a. foundational operation such as lookup and neighbor query
   b. graph-specific tasks such as t-hop and graph traversal
   c. extension to dynamic graphs, i.e., insertion and deletion.

We mainly evaluate operations on vertices here, and leave the edge operations in the full version [25] as edge operations are completed via accessing the corresponding vertices under our design. Next, we detail our experimental setup, datasets, and metrics. Both Grove and GraphOS use the same security parameters $\lambda = 128$ and hardware. As GraphOS was originally designed within enclaves, we adapt and extensively optimize it to enable a fair comparison.

Experimental Setup. We implement all components in Python 3.12. All cryptographic primitives are provided by the pycryptodome package [1]. For ORAM trees $\mathcal{T}_P$ and $\mathcal{T}_G$, each bucket has four blocks, **the bucket capacity** Y **for meta blocks is set according to Theorem 3.1 and we guarantee the overflow probability is strictly smaller than** $2^{-\lambda}$**.** The entities $\mathcal{C}$ and $\mathcal{S}$ run on separate machines and communicate over a WAN. To reflect realistic conditions, $\mathcal{C}$ is in Virginia, and $\mathcal{S}$ is in California; the measured average round-trip latency between them is 35 ms. $\mathcal{S}$ is an Alibaba Cloud instance that owns an Intel Xeon Platinum 8160 with 32 cores at 2.10 GHz and 64 GB of memory, while $\mathcal{C}$ is a constrained instance with 2 vCPUs from an Intel Xeon Platinum 8269CY at 2.50 GHz and 8 GB of memory. The bandwidth between them is fixed at 100 Mbps.

Batching. We adopt the AVL-tree-based OMAP [14, 79] used in GraphOS as the underlying OMAP in our implementations. We conduct high batching (i.e., piggyback) on both the OMAP and ORAM trees in two systems to fully optimize them under the client/server model. That is, we batch all available independent requests to issue and execute them within *the minimum roundtrip*:

- GraphOS relies solely on OMAP operations to process queries. For instance, a neighbor query that retrieves K neighbors requires K separate OMAP lookup operations. Since these lookups are often independent, they can be batched together and executed in the same number of interaction rounds as a single OMAP operation. In our implementation, we ensure that all available independent OMAP operations are fully batched. Consequently, the total number of interaction rounds incurred by GraphOS is determined only by the number of *sequentially dependent* OMAP operations and the underlying OMAP protocol.
- Grove, by contrast, leverages its IFA design to avoid costly OMAP operations in graph-specific queries. Batching is applied to OMAP operations only during insertions. In most query scenarios, aside from a single OMAP operation to fetch the starting vertex or edge, Grove processes multiple independent ORAM paths to retrieve both data and meta blocks. We batch the retrieval and write-back of these paths into a single roundtrip for efficiency.

The batching technique described above is a standard optimization in oblivious algorithms [10, 15, 80] for reducing client-server interaction latency. This ensures that our experimental deployment reflects the true end-to-end performance of both systems. In particular, all experiments are conducted under identical network conditions, guaranteeing a fair comparison.

Datasets. Consistent with prior oblivious works [10, 15, 36], we use synthetic datasets that vary vertex number $|V|$, edge number $|E|$ and the maximum vertex degree K due to the obliviousness property [15]. We vary $|V|$ from 2^{10} to 2^{30} to evaluate performance across graph scales. Due to storage and costly setup constraints, we conduct real experiments only on graphs with up to 2^{20} vertices. For larger scales (2^{21} to 2^{30}), we simulate the system behavior and the simulation details are provided in the full version [25]. To instantiate the complete graphs, we generate each graph with a proportional number of edges, i.e., $|E| = K|V|$, where K is set to 10 and 100. When evaluating Grove with $K = 100$, we sparsify graphs with $D = 10$. We use V' to denote the vertex set after sparsification. For convenience, we still report results using the original dataset vertex count; however, when $K = 100$ the underlying Grove datasets include newly inserted intermediate vertices. For each dataset, the per-vertex value size is 64 B, 4 KB, and 22 KB to reflect different application scenarios.

Metrics. There are four main metrics considered throughout our experiments. Besides server-side storage, we test the query processing time (also called runtime) as the key efficiency metric; the interaction rounds as the inherent efficiency metric independent of latency, and the bandwidth to reflect the communication volume.

6.1 Evaluation of Delayed Duplications

We first measure the overhead introduced by delayed duplication in enabling the IFA design for oblivious graphs. Note that each duplication consists only of a vertex identifier and two path numbers,

Table 4: Percentage of bandwidth and runtime from delayed duplications relative to total query processing ($K = 10$).

Graph Size		$\|V\| = 2^{15}$			$\|V\| = 2^{20}$			$\|V\| = 2^{25}$			$\|V\| = 2^{30}$		
Metrics		Band-W	Band-I	Runtime	Band-W	Band-I	Runtime	Band-W	Band-I	Runtime	Band-W	Band-I	Runtime
Lookup	64 B	74.52%	15.12%	3.42%	70.08%	11.65%	3.27%	66.35%	9.38%	3.14%	62.78%	7.68%	3.02%
	4 KB	33.15%	2.98%	1.05%	32.85%	2.72%	0.97%	32.55%	2.51%	0.90%	32.25%	2.35%	0.84%
	22 KB	12.95%	1.42%	0.44%	12.82%	1.28%	0.39%	12.70%	1.18%	0.36%	12.58%	1.12%	0.35%
Neighbor	64 B	83.15%	10.15%	3.18%	85.42%	9.92%	3.27%	86.58%	9.68%	3.34%	87.35%	9.48%	3.40%
	4 KB	52.15%	1.98%	0.88%	54.55%	1.95%	0.93%	55.68%	1.92%	0.97%	56.22%	1.90%	1.00%
	22 KB	18.72%	0.75%	0.32%	20.35%	0.75%	0.35%	21.22%	0.75%	0.37%	21.78%	0.75%	0.39%

[a] **Band-W**: The (worst-case) bandwidth overhead attributable to delayed duplications when choosing Y based on the theoretical bound in Theorem 3.1.
[b] **Band-I: The (improved) bandwidth overhead attributable to delayed duplications when choosing Y according to the practical guideline.**

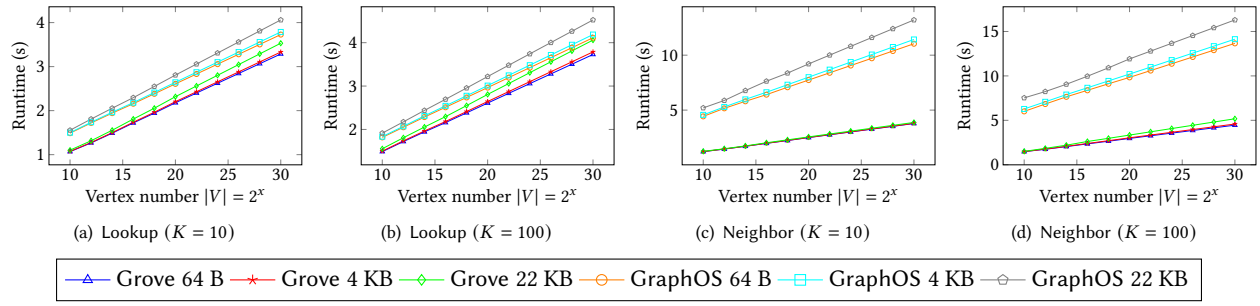


Figure 5: Performance of fundamental operations.

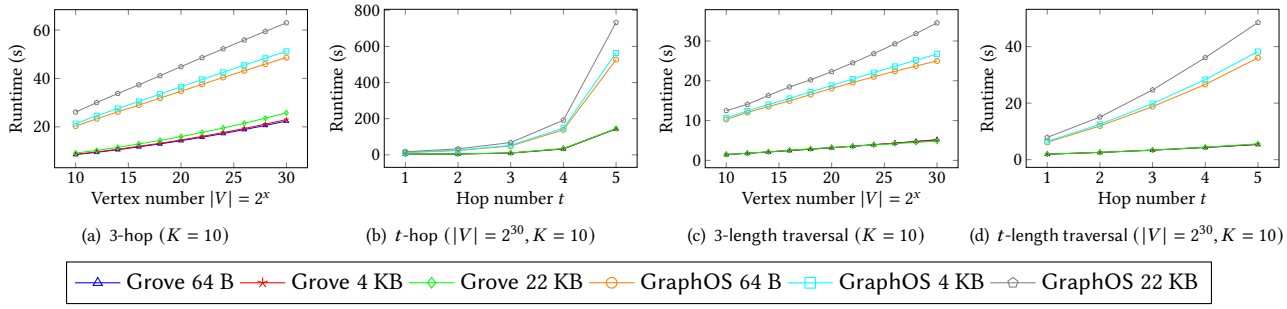


Figure 6: Performance of graph queries.

incurring a fixed cost per bucket. Consequently, as the vertex size increases, the relative overhead of duplications becomes increasingly negligible. To comprehensively evaluate this overhead across a wide range of vertex sizes, we vary the vertex payload from 64 B to 22 KB. The experimental results are shown in Table 4 where we pick up the foundational operations lookup and neighbor query as test objects. Recall that duplications do not occupy additional interaction rounds due to our piggyback strategy, so we present the proportions of processing duplication relative to the total query processing on only bandwidth and runtime. Importantly, in addition to evaluating the bandwidth cost (i.e., column Band-W) and runtime under the theoretical bucket capacity bound for meta blocks, we also report the bandwidth cost in column Band-I when Y is set heuristically. As discussed in Section 3.3.2, this heuristic value reflects a practical optimization available to data users. **Throughout our experiments, we adopt a conservative configuration that ensures negligible overflow probability** by using large bucket capacities for meta blocks—typically $Y \in [300, 350]$, whereas feasible heuristic values can be as low as 10. Even under this strict assumption, Grove still outperforms GraphOS, demonstrating the inherent efficiency of the IFA design in oblivious graph processing.

There are three important facts observed from Table 4: ❶ the overhead is stable under different vertex numbers, validating its great scalability; ❷ the bandwidth and runtime overhead significantly decrease with vertex size increases from 64 B to 22 KB, implying the duplications can perform better in applications with larger vertex size; ❸ the processing time is always incremental, even when the vertex size is as small as 64 B and the bandwidth cost reaches 74.52%, the runtime overhead is only 3.42%! The gap between bandwidth and runtime overhead is because the interaction rounds are the main bottleneck under the client/server model. Remarkably, despite the 12.58% $\sim$ 74.52% bandwidth overhead, Grove still outperforms GraphOS in bandwidth as shown in the next section. Additionally, we evaluate larger K values with $D = 10$ to assess scalability with respect to vertex degree; due to space constraints, the results appear in the appendix of the full version [25].

6.2 Comparison between Systems

We begin our comparison of Grove and GraphOS by characterizing server-side storage and setup time. Table 5 presents the storage required to instantiate both systems, revealing a significant gap. GraphOS consistently incurs substantially higher storage overhead

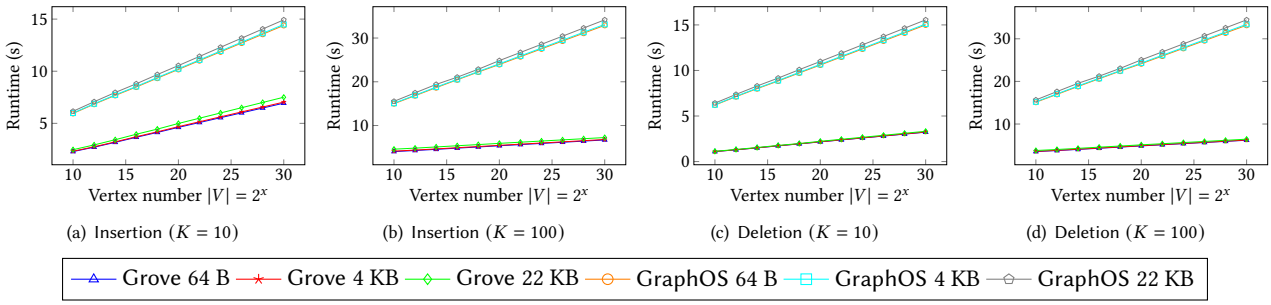


Figure 7: Performance of insertion and deletion.

because it stores vertices and edges as equal entities in a single OMAP of size $O(\log(|V| + |E|))$. In contrast, Grove embeds edge directly within vertices, resulting in an OMAP whose size scales as $O(\log |V|)$. For example, when $|V| = 2^{20}$ and the vertex size is 22 KB, Grove requires only 181 GB of server storage, whereas GraphOS consumes 2.8 TB, a **15.5**× increase. The setup time for GraphOS under this configuration exceeds 317 hours, making real-world evaluation impractical beyond $|V| = 2^{20}$. This motivates our simulation experiments for larger scales ($|V| = 2^{25}$ and 2^{30}), where storage costs are calculated based on the setup parameters and algorithms. In real deployments across $|V| \in \{2^{10}, 2^{11}, \ldots, 2^{20}\}$, Grove achieves a setup time speedup of 80.6% to 94.6% and reduces server-side storage by 74.6% to 92.5%, consistently outperforming GraphOS across all evaluated graph sizes.

6.2.1 Foundational Operations. We evaluate the performance of Grove and GraphOS on foundational operations, namely lookup and neighbor queries. We omit the update since it adopts the same interface as lookup. Results for runtime, interaction rounds, and bandwidth are shown in Figure 5 and Table 6. For both operations, we observe Grove outperforms GraphOS across all three metrics.

Although IFA is primarily designed to optimize graph traversal, Grove also achieves faster lookups with fewer interaction rounds and lower bandwidth, as shown in Figure 5 and Table 6. This stems from the fact that both systems rely on an underlying OMAP, but Grove uses a significantly smaller one. The results in Figure 5, for the 64 B vertex size, Grove is 11.8% to 28.2% faster than GraphOS across all tested graph sizes in lookup queries. The performance gap narrows as K increases because additional intermediate vertices enlarge Grove 's OMAP, thereby reducing the size gap between the two systems, e.g., the maximum speedup drops to 18.1% when $K = 100$ and the vertex size is 64 B. Moreover, the speedup is also affected by vertex size. As the vertex size grows, the relative overhead of delayed duplications becomes increasingly negligible, leading to greater performance gains. For instance, under $K = 10$, the maximum lookup speedup rises to 28.4% and 29.1% when the vertex size is increased to 4 KB and 22 KB, respectively.

The speedup on neighbor query is more substantial since this operation involves graph traversal, which IFA targets to improve. In Figures 5(c) and 5(d), the speedup ranges from 65.8% to 76.0%. The improvement stems from many fewer interaction rounds; for $|V| = 2^{20}$ the count drops from 108 in GraphOS to 31 in Grove as shown in Table 6. IFA enables Grove to complete the operation with a single OMAP access rather than multiple sequential OMAP accesses in GraphOS. Bandwidth declines accordingly, for example, from 37.8 MB to 26 MB when $|V| = 2^{20}$ with 22 KB vertex values.

Table 5: Server storage cost comparison ($K = 10$).

Graph Size $\lvert V\rvert$	2^{10}	2^{15}	2^{20}	2^{25}	2^{30}
64 B (Grove)	5.6 MB	180 MB	5.6 GB	180 GB	5.8 TB
64 B (GraphOS)	18 MB	576 MB	18 GB	576 GB	18 TB
4 KB (Grove)	37 MB	1.2 GB	37 GB	1.2 TB	38 TB
4 KB (GraphOS)	522 MB	16 GB	522 GB	16 TB	522 TB
22 KB (Grove)	181 MB	5.7 GB	181 GB	5.8 TB	181 TB
22 KB (GraphOS)	2.8 GB	88 GB	2.8 TB	88 TB	2.8 PB

6.2.2 Graph-specific Queries. To further demonstrate how IFA accelerates graph traversal, we evaluate t-hop query and t-length traversal. We expect Grove to deliver larger speedups: for a series of neighbor retrievals, IFA lets Grove complete each traversal in a single interaction round, whereas GraphOS requires multiple OMAP accesses and corresponding multiple rounds.

In Figure 6, we vary the vertex count $|V|$ and hop number t independently to evaluate the t-hop query. With t fixed to 3, Grove achieves speedups between 54.0% and 65.9%. As $|V|$ increases, Grove 's advantage widens, since repeated OMAP operations in GraphOS scale more quickly with $|V|$ than the single-round neighbor traversal in Grove. With $|V|$ fixed instead, the cost of both systems grows rapidly as the number of retrieved neighbors often increases exponentially with t, substantially increasing bandwidth and client computation. For large t, client computation becomes the main bottleneck, which makes the benefit of fewer interaction rounds in Grove less pronounced. Even then, the speedup can be approximated from bandwidth differences: since Grove transfers less total data, it continues to improve over GraphOS. Across the reported t values, Grove attains a maximum speedup of 71.5% to 84.5% in runtime and improvement of 42.3% to 45.1% on bandwidth.

We evaluate the t-length traversal using the same design, varying t and $|V|$ independently, as shown in Figure 6. Similarly, Grove 's advantage widens as $|V|$ grows. When varying t, t-length traversal differs from t-hop: the per-step bandwidth is essentially constant for both systems and much smaller than in the t-hop setting, so the plotted lines have roughly constant slopes and interaction rounds dominate the time. Since Grove 's interaction rounds grow much more slowly with t than those of GraphOS, larger t yields greater

Table 6: Comparison on interaction round and bandwidth under $|V| = 2^{20}$, $K = 10$, and vertex size of 22 KB.

Operation	Grove		GraphOS	
	Roundtrip	Bandwidth	Roundtrip	Bandwidth
`Lookup`	30	2.54 MB	36	3.22 MB
`Neighbor`	31	26.01 MB	108	37.82 MB
`Insertion`	31	28.98 MB	108	157.66 MB
`Deletion`	30	2.54 MB	108	157.66 MB
`3-hop query`	33	1.91 GB	252	3.2 GB
`3-length trav`	33	72.95 MB	252	107.05 MB

runtime speedups, consistent with the results that at $t = 1$ Grove achieves a speedup of 75.5% and at $t = 5$ a speedup of 88.4%.

6.2.3 Dynamic Graphs. Lastly, we evaluate dynamic graph operations, i.e., insertion and deletion. In Grove, insertion adds the new vertex and identifies neighbor paths to place delayed duplications correctly as update metadata. In contrast, GraphOS must insert the vertex, insert each incident edge, and download neighbor vertices to update stored information, which results in many more interaction rounds and substantially higher bandwidth. As demonstrated in Figure 7, Grove achieves an insertion speedup of 49.7% to 79.6%. As K increases and Grove introduces additional layers of intermediate vertices, insertion becomes less parallelizable because locating neighbor information requires retrieving these intermediate vertices layer by layer. GraphOS has a fixed number of interaction rounds with respect to K, while its bandwidth alone grows with K. Even so, the initial gap in interaction rounds is large, and with many intermediate layers the neighbor count is also large, making bandwidth the primary bottleneck. Consequently, Grove continues to outperform GraphOS as K grows.

For deletion, GraphOS follows the same pattern as insertion: it removes incident edges and updates neighbor vertices, which again incurs many interaction rounds and high bandwidth. In Grove, when no intermediate vertices are present, deletion is nearly identical to lookup: delayed duplications notify neighbors of the deletion, for example by setting the updated path to a negative value. If intermediate vertices exist, Grove retrieves the relevant intermediate vertices to inform neighbors, mirroring insertion. Figure 7 shows that Grove improves deletion performance by 75.8% to 82.5%.

7 RELATED WORK

As many applications model their data as graphs [61], encrypted graphs have recently attracted growing attention across multiple communities. In addition to the representative works discussed in Section 1.2, several other studies also explore encrypted graphs.

Non-oblivious Graph Encryption. In the context of EDBs, a series of structured encryption (STE) schemes [9, 17, 22, 30, 55] have been proposed to process encrypted graphs. These schemes encrypt the graph while permitting limited leakage during query processing to enable specific functionality. For example, Liu et al. [50] leak the order of distances between vertices for shortest-distance queries. Compared with oblivious graph processing, such approaches generally reveal more leakage to the adversary and are therefore more vulnerable to attacks [24, 31]. Their main advantage is speed: they typically require fewer interaction rounds and lower bandwidth. Grove and these STE schemes target different scenarios; clients can choose between them to trade off security and practicality.

Other Oblivious Graphs. Wang et al. [79] are the first to consider oblivious graph processing, but they are limited to specific graphs (e.g., trees) and thus avoid the conflict with IFA. OblivGM [78] supports oblivious attributed subgraph matching but requires non-colluding servers. Keller et al. [40] also propose the oblivious algorithm for Dijkstra's shortest path algorithm [20] in the secure multi-party computation setting [8]. All these settings differ from our single-server data-outsourcing scenario. Bhattacharjee et al. [6] also discuss oblivious property graph databases in a vision paper, but do not provide a concrete protocol. In summary, our work fulfills the study for oblivious graph processing in enabling IFA under the typical setting of obliviousness for practical systems.

7.1 Discussion

Enclave-based EDBs Besides the client/server model, another promising line of work [21, 56, 74, 82] leverages trusted execution environments (TEEs) within the server to act as an avatar for the client, thereby avoiding interaction delays. Our design can also be adapted to this setting and apply existing optimizations [21, 27, 38, 49, 56, 65, 74, 82] to enhance the performance. For instance, Li et al. [49] propose an efficient and oblivious bulk loading algorithm for Path ORAM initialization, which can accelerate the initialization of ORAM trees in Grove. Similarly, the AVL-OMAP implementation can be optimized using techniques from EnigMap [74], such as packing parent nodes and their children to reduce page swaps for enclaves. For GraphOS, not only can its oblivious graph processing inside the enclave be utilized, but its improved oblivious eviction can also be beneficial. These synergies indicate that Grove is compatible with TEEs and has significant potential to leverage ongoing advances in oblivious systems research.

IFA design. In this work, we actually adapt the original IFA design in Neo4j to suit our new context, namely, the client/server model and obliviousness constraint. In Neo4j, vertices, edges, and attributes are stored as separate objects to manage. Each vertex holds pointers to its first outgoing and incoming edges, and edges form linked lists that reference adjacent vertices. In contrast, to realize IFA in oblivious graphs, Grove prioritizes vertex operations and graph traversal by treating each vertex and its associated edges and attributes as a unified unit (similar to recent works [4, 86]). We recommend that data users adapt the storage model to their specific workloads, while leveraging delayed duplication as the core technique for IFA to enable efficient oblivious graph processing.

8 CONCLUSION

In this work, we aim to address a key problem of obliviousness in EDBs, namely enabling IFA in oblivious graphs for enhanced performance. Compared with prior works, our new technique, delayed duplication, is the first to solve this problem efficiently without relaxing security guarantees. We combine this technique with carefully designed data structures to develop an EDB, Grove, that processes graph queries both practically and obliviously. Our empirical evaluation shows that Grove substantially outperforms the SOTA system GraphOS [14] under the client/server paradigm. We release the open-source library at https://github.com/weiqins/daoram.

REFERENCES

[1] [n.d.]. Python package pycryptodome. ([n. d.]). https://github.com/Legrandin/pycryptodome

[2] Haseeb Ahmed, Nachiket Rao, Abdelkarim Kati, Florian Kerschbaum, and Sujayya Maiyya. 2025. OasisDB: An Oblivious and Scalable System for Relational Data. *Cryptology ePrint Archive* (2025).

[3] Tatsuya Aida, So Hasegawa, Maki Arai, Ryosuke Isogai, Yozo Shoji, and Mikio Hasegawa. 2024. Optimization of End-to-End Throughput on Piggyback Network with drone-to-drone millimeter-wave communications. In *2024 International Conference on Information Networking (ICOIN)*. IEEE, 751–755.

[4] Ananya Appan, David Heath, and Ling Ren. 2024. Oblivious Single Access Machines - A New Model for Oblivious Computation, See [51], 3080–3094. https://doi.org/10.1145/3658644.3690352

[5] arXiv. 2024. *Introduction to Index-Free Adjacency in Graph Databases*. Technical Report 2412.18143v1. arXiv. https://arxiv.org/html/2412.18143v1 Preprint, December 2024.

[6] Bishwajit Bhattacharjee, Nafis Ahmed, Renée Miller, and Sujaya Maiyya. 2025. [Vision] Towards oblivious property graph databases. In *Proceedings of the 8th Joint Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*. 1–6.

[7] Erik-Oliver Blass, Travis Mayberry, and Guevara Noubir. 2017. Multi-client Oblivious RAM Secure Against Malicious Servers. In *ACNS 17: 15th International Conference on Applied Cryptography and Network Security (Lecture Notes in Computer Science)*, Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi (Eds.), Vol. 10355. Springer, Cham, Switzerland, Kanazawa, Japan, 686–707. https://doi.org/10.1007/978-3-319-61204-1_34

[8] Elette Boyle, Kai-Min Chung, and Rafael Pass. 2015. Large-Scale Secure Computation: Multi-party Computation for (Parallel) RAM Programs. In *Advances in Cryptology – CRYPTO 2015, Part II (Lecture Notes in Computer Science)*, Rosario Gennaro and Matthew J. B. Robshaw (Eds.), Vol. 9216. Springer Berlin Heidelberg, Germany, Santa Barbara, CA, USA, 742–762. https://doi.org/10.1007/978-3-662-48000-7_36

[9] Ning Cao, Zhenyu Yang, Cong Wang, Kui Ren, and Wenjing Lou. 2011. Privacy-preserving query over encrypted graph-structured data in cloud computing. In *2011 31st International Conference on Distributed Computing Systems*. IEEE, 393–402.

[10] Xinle Cao, Weiqi Feng, Jian Liu, Jinjin Zhou, Wenjing Fang, Lei Wang, Quanqing Xu, Chuanhui Yang, and Kui Ren. 2024. Towards Practical Oblivious Map. *Proc. VLDB Endow.* 18, 3 (Nov. 2024), 688–701. https://doi.org/10.14778/3712221.3712235

[11] Xinle Cao, Yuhan Li, Dmytro Bogatov, Jian Liu, and Kui Ren. 2024. Secure and Practical Functional Dependency Discovery in Outsourced Databases . In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE Computer Society, Los Alamitos, CA, USA, 1645–1658. https://doi.org/10.1109/ICDE60146.2024.00134

[12] Xinle Cao, Jian Liu, Hao Lu, and Kui Ren. 2021. Cryptanalysis of an encrypted database in SIGMOD'14. *Proceedings of the VLDB Endowment* 14, 10 (2021), 1743–1755.

[13] David Cash, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. 2013. Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries. In *Advances in Cryptology – CRYPTO 2013, Part I (Lecture Notes in Computer Science)*, Ran Canetti and Juan A. Garay (Eds.), Vol. 8042. Springer Berlin Heidelberg, Germany, Santa Barbara, CA, USA, 353–373. https://doi.org/10.1007/978-3-642-40041-4_20

[14] Javad Ghareh Chamani, Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Rasool Jalili. 2023. GraphOS: Towards Oblivious Graph Processing. *Proc. VLDB Endow.* 16, 13 (Sept. 2023), 4324–4338. https://doi.org/10.14778/3625054.3625067

[15] Zhao Chang, Dong Xie, and Feifei Li. 2016. Oblivious RAM: A dissection and experimental evaluation. *Proceedings of the VLDB Endowment* 9, 12 (2016), 1113–1124.

[16] Zhao Chang, Dong Xie, Sheng Wang, and Feifei Li. 2022. Towards Practical Oblivious Join. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 803–817. https://doi.org/10.1145/3514221.3517868

[17] Melissa Chase and Seny Kamara. 2010. Structured Encryption and Controlled Disclosure. In *Advances in Cryptology – ASIACRYPT 2010 (Lecture Notes in Computer Science)*, Masayuki Abe (Ed.), Vol. 6477. Springer Berlin Heidelberg, Germany, Singapore, 577–594. https://doi.org/10.1007/978-3-642-17373-8_33

[18] Martino Ciaperoni, Nikolaos Tziavelis, Athanasios Katsamanis, and Panagiotis Karras. [n.d.]. Fast and Space-Efficient Fixed-Length Path Optimization. ([n. d.]).

[19] Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Saurabh Shintre. 2020. SEAL: Attack Mitigation for Encrypted Databases via Adjustable Leakage. In *USENIX Security 2020: 29th USENIX Security Symposium*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 2433–2450.

[20] E. W. Dijkstra. 1959. A note on two problems in connexion with graphs. *Numer. Math.* 1, 1 (Dec. 1959), 269–271. https://doi.org/10.1007/BF01386390

[21] Saba Eskandarian and Matei Zaharia. 2019. ObliDB: oblivious query processing for secure databases. *Proc. VLDB Endow.* 13, 2 (Oct. 2019), 169–183. https://doi.org/10.14778/3364324.3364331

[22] Francesca Falzon, Esha Ghosh, Kenneth G. Paterson, and Roberto Tamassia. 2024. PathGES: An Efficient and Secure Graph Encryption Scheme for Shortest Path Queries, See [51], 4047–4061. https://doi.org/10.1145/3658644.3670305

[23] Francesca Falzon, Evangelia Anna Markatou, Akshima, David Cash, Adam Rivkin, Jesse Stern, and Roberto Tamassia. 2020. Full Database Reconstruction in Two Dimensions. In *ACM CCS 2020: 27th Conference on Computer and Communications Security*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM Press, Virtual Event, USA, 443–460. https://doi.org/10.1145/3372297.3417275

[24] Francesca Falzon and Kenneth G Paterson. 2022. An efficient query recovery attack against a graph encryption scheme. In *European Symposium on Research in Computer Security*. Springer, 325–345.

[25] Weiqi Feng, Xinle Cao, Adam O'Neill, and Chuanhui Yang. 2025. Enabling Index-free Adjacency in Oblivious Graph Processing with Delayed Duplications. Cryptology ePrint Archive, Paper 2025/2047. https://eprint.iacr.org/2025/2047

[26] Bernardo Ferreira, Bernardo Portela, Tiago Oliveira, Guilherme Borges, Henrique Domingos, and João Leitão. 2022. Boolean Searchable Symmetric Encryption With Filters on Trusted Hardware. *IEEE Transactions on Dependable and Secure Computing* 19, 2 (2022), 1307–1319. https://doi.org/10.1109/TDSC.2020.3012100

[27] Christopher W. Fletcher, Ling Ren, Albert Kwon, Marten van Dijk, and Srinivas Devadas. 2015. Freecursive ORAM: [Nearly] Free Recursion and Integrity Verification for Position-based Oblivious RAM. *SIGPLAN Not.* 50, 4 (March 2015), 103–116. https://doi.org/10.1145/2775054.2694353

[28] Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. 2016. TWORAM: Efficient Oblivious RAM in Two Rounds with Applications to Searchable Encryption. In *Advances in Cryptology – CRYPTO 2016, Part III (Lecture Notes in Computer Science)*, Matthew Robshaw and Jonathan Katz (Eds.), Vol. 9816. Springer Berlin Heidelberg, Germany, Santa Barbara, CA, USA, 563–592. https://doi.org/10.1007/978-3-662-53015-3_20

[29] Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit S. Jutla, Mariana Raykova, and Daniel Wichs. 2013. Optimizing ORAM and Using It Efficiently for Secure Computation. In *PETS 2013: 13th International Symposium on Privacy Enhancing Technologies (Lecture Notes in Computer Science)*, Emiliano De Cristofaro and Matthew K. Wright (Eds.), Vol. 7981. Springer Berlin Heidelberg, Germany, Bloomington, IN, USA, 1–18. https://doi.org/10.1007/978-3-642-39077-7_1

[30] Esha Ghosh, Seny Kamara, and Roberto Tamassia. 2021. Efficient graph encryption scheme for shortest path queries. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. 516–525.

[31] Anselme Goetschmann. 2020. *Design and Analysis of Graph Encryption Schemes*. Master's thesis. ETH Zurich.

[32] Oded Goldreich and Rafail Ostrovsky. 1996. Software protection and simulation on oblivious RAMs. *J. ACM* 43, 3 (May 1996), 431–473. https://doi.org/10.1145/233551.233553

[33] Paul Grubbs, Anurag Khandelwal, Marie-Sarah Lacharité, Lloyd Brown, Lucy Li, Rachit Agarwal, and Thomas Ristenpart. 2020. Pancake: Frequency smoothing for encrypted data stores. In *29th USENIX Security Symposium (USENIX Security 20)*. 2451–2468.

[34] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. 2019. Learning to reconstruct: Statistical learning theory and encrypted database attacks. In *2019 IEEE symposium on security and privacy (SP)*. IEEE, 1067–1083.

[35] Zichen Gui, Kenneth G. Paterson, and Tianxin Tang. 2023. Security Analysis of MongoDB Queryable Encryption. In *USENIX Security 2023: 32nd USENIX Security Symposium*, Joseph A. Calandrino and Carmela Troncoso (Eds.). USENIX Association, Anaheim, CA, USA, 7445–7462.

[36] Thang Hoang, Ceyhun D Ozkaptan, Gabriel Hackebeil, and Attila Altay Yavuz. 2018. Efficient oblivious data structures for database services on the cloud. *IEEE Transactions on Cloud Computing* 9, 2 (2018), 598–609.

[37] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation, See [58].

[38] Seyni Kane and Anis Bkakria. 2024. A privacy-preserving graph encryption scheme based on oblivious RAM. In *IFIP Annual Conference on Data and Applications Security and Privacy*. Springer, 101–108.

[39] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. 2016. Generic Attacks on Secure Outsourced Databases. In *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM Press, Vienna, Austria, 1329–1340. https://doi.org/10.1145/2976749.2978386

[40] Marcel Keller and Peter Scholl. 2014. Efficient, Oblivious Data Structures for MPC. In *Advances in Cryptology – ASIACRYPT 2014, Part II (Lecture Notes in Computer Science)*, Palash Sarkar and Tetsu Iwata (Eds.), Vol. 8874. Springer Berlin Heidelberg, Germany, Kaoshiung, Taiwan, R.O.C., 506–525. https://doi.org/10.1007/978-3-662-45608-8_27

[41] Bryan Klimt and Yiming Yang. 2004. The enron corpus: a new dataset for email classification research. In *Proceedings of the 15th European Conference on Machine Learning* (Pisa, Italy) *(ECML'04)*. Springer-Verlag, Berlin, Heidelberg, 217–226. https://doi.org/10.1007/978-3-540-30115-8_22

[42] Simeon Krastnikov, Florian Kerschbaum, and Douglas Stebila. 2020. Efficient oblivious database joins. *Proc. VLDB Endow.* 13, 12 (July 2020), 2132–2145. https://doi.org/10.14778/3407790.3407814

[43] Ranjay Krishna, Yuke Zhu, Oliver Groth, Justin Johnson, Kenji Hata, Joshua Kravitz, Stephanie Chen, Yannis Kalantidis, Li-Jia Li, David A. Shamma, Michael S. Bernstein, and Li Fei-Fei. 2017. Visual Genome: Connecting Language and Vision Using Crowdsourced Dense Image Annotations. *Int. J. Comput. Vision* 123, 1 (May 2017), 32–73. https://doi.org/10.1007/s11263-016-0981-7

[44] Mehmet Kuzu, Mohammad Saiful Islam, and Murat Kantarcioglu. 2012. Efficient Similarity Search over Encrypted Data. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering (ICDE '12)*. IEEE Computer Society, USA, 1156–1167. https://doi.org/10.1109/ICDE.2012.23

[45] Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. 2018. Improved reconstruction attacks on encrypted data using range query leakage. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 297–314.

[46] Steven Lambregts, Huanhuan Chen, Jianting Ning, and Kaitai Liang. 2022. VAL: Volume and Access Pattern Leakage-Abuse Attack with Leaked Documents. In *ESORICS 2022: 27th European Symposium on Research in Computer Security, Part I (Lecture Notes in Computer Science)*, Vijayalakshmi Atluri, Roberto Di Pietro, Christian Damsgaard Jensen, and Weizhi Meng (Eds.), Vol. 13554. Springer, Cham, Switzerland, Copenhagen, Denmark, 653–676. https://doi.org/10.1007/978-3-031-17140-6_32

[47] Jure Leskovec, Lada A. Adamic, and Bernardo A. Huberman. 2007. The dynamics of viral marketing. *ACM Trans. Web* 1, 1 (May 2007), 5–es. https://doi.org/10.1145/1232722.1232727

[48] Dongjie Li, Siyi Lv, Yanyu Huang, Yijing Liu, Tong Li, Zheli Liu, and Liang Guo. 2021. Frequency-hiding order-preserving encryption with small client storage. *Proceedings of the VLDB Endowment* 14, 13 (2021), 3295–3307.

[49] Xiang Li, Yunqian Luo, and Mingyu Gao. 2024. BULKOR: Enabling Bulk Loading for Path ORAM. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 4258–4276.

[50] Chang Liu, Liehuang Zhu, Xiangjian He, and Jinjun Chen. 2018. Enabling privacy-preserving shortest distance queries on encrypted graph data. *IEEE Transactions on Dependable and Secure Computing* 18, 1 (2018), 192–204.

[51] Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie (Eds.). 2024. *ACM CCS 2024: 31st Conference on Computer and Communications Security*. ACM Press, Salt Lake City, UT, USA.

[52] Sujaya Maiyya, Sharath Chandra Vemula, Divyakant Agrawal, Amr El Abbadi, and Florian Kerschbaum. 2023. Waffle: An online oblivious datastore for protecting data access patterns. *Proceedings of the ACM on Management of Data* 1, 4 (2023), 1–25.

[53] Yu A Malkov and Dmitry A Yashunin. 2018. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence* 42, 4 (2018), 824–836.

[54] Charalampos Mavroforakis, Nathan Chenette, Adam O'Neill, George Kollios, and Ran Canetti. 2015. Modular order-preserving encryption, revisited. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 763–777.

[55] Xianrui Meng, Seny Kamara, Kobbi Nissim, and George Kollios. 2015. GRECS: Graph Encryption for Approximate Shortest Distance Queries. In *ACM CCS 2015: 22nd Conference on Computer and Communications Security*, Indrajit Ray, Ninghui Li, and Christopher Kruegel (Eds.). ACM Press, Denver, CO, USA, 504–517. https://doi.org/10.1145/2810103.2813672

[56] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. 2018. Oblix: An efficient oblivious search index. In *2018 IEEE symposium on security and privacy (SP)*. IEEE, 279–296.

[57] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. 2018. Oblix: An Efficient Oblivious Search Index. In *2018 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Francisco, CA, USA, 279–296. https://doi.org/10.1109/SP.2018.00045

[58] NDSS 2012 2012. *ISOC Network and Distributed System Security Symposium – NDSS 2012*. The Internet Society, San Diego, CA, USA.

[59] Inc. Neo4j. 2025. Neo4j: The Graph Database. https://neo4j.com Accessed: 2025-08-20.

[60] Yue Pang, Lei Zou, and Yu Liu. 2023. IFCA: index-free community-aware reachability processing over large dynamic graphs. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 2220–2234.

[61] Georgios A Pavlopoulos, Maria Secrier, Charalampos N Moschopoulos, Theodoros G Soldatos, Sophia Kossida, Jan Aerts, Reinhard Schneider, and Pantelis G Bagos. 2011. Using graph theory to analyze biological networks. *BioData mining* 4, 1 (2011), 10.

[62] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. 2019. Arx: an encrypted database using semantically secure encryption. *Proc. VLDB Endow.* 12, 11 (July 2019), 1664–1678. https://doi.org/10.14778/3342263.3342641

[63] Jaroslav Pokornỳ. 2015. Graph databases: their power and limitations. In *IFIP International Conference on Computer Information Systems and Industrial Management*. Springer, 58–69.

[64] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2012. CryptDB: processing queries on an encrypted database. *Commun. ACM* 55, 9 (Sept. 2012), 103–111. https://doi.org/10.1145/2330667.2330691

[65] Leonie Reichert, Gowri R Chandran, Phillipp Schoppmann, Thomas Schneider, and Björn Scheuermann. 2024. Menhir: an oblivious database with protection against access and volume pattern leakage. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*. 1675–1690.

[66] Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. 2015. Constants Count: Practical Improvements to Oblivious RAM. In *USENIX Security 2015: 24th USENIX Security Symposium*, Jaeyeon Jung and Thorsten Holz (Eds.). USENIX Association, Washington, DC, USA, 415–430.

[67] Marcel Rosu. 2014. Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation. In *Proceedings 2014 Network and Distributed System Security Symposium*.

[68] Raimund Seidel. 1995. On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of computer and system sciences* 51, 3 (1995), 400–403.

[69] Elaine Shi. 2020. Path oblivious heap: Optimal and practical oblivious priority queue. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 842–858.

[70] Yunjiao Song, Xinrui Ge, Jia Yu, Rong Hao, and Ming Yang. 2024. Enabling Privacy-Preserving K K-Hop Reachability Query Over Encrypted Graphs. *IEEE Transactions on Services Computing* 17, 3 (2024), 893–904.

[71] Emil Stefanov, Marten Van Dijk, Elaine Shi, T.-H. Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2018. Path ORAM: An Extremely Simple Oblivious RAM Protocol. *J. ACM* 65, 4, Article 18 (April 2018), 26 pages. https://doi.org/10.1145/3177872

[72] Emil Stefanov, Elaine Shi, and Dawn Xiaodong Song. 2012. Towards Practical Oblivious RAM, See [58].

[73] Yuanyuan Sun, Sheng Wang, Huorong Li, and Feifei Li. 2021. Building enclave-native storage engines for practical encrypted databases. *Proc. VLDB Endow.* 14, 6 (Feb. 2021), 1019–1032. https://doi.org/10.14778/3447689.3447705

[74] Afonso Tinoco, Sixiang Gao, and Elaine Shi. 2023. {EnigMap}:{External-Memory} Oblivious Map for Secure Enclaves. In *32nd USENIX Security Symposium (USENIX Security 23)*. 4033–4050.

[75] Bayu Distiawan Trisedya, Jianzhong Qi, and Rui Zhang. 2019. Entity alignment between knowledge graphs using attribute embeddings. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 33. 297–304.

[76] Chad Vicknair, Michael Macias, Zhendong Zhao, Xiaofei Nan, Yixin Chen, and Dawn Wilkins. 2010. A comparison of a graph database and a relational database: a data provenance perspective. In *Proceedings of the 48th annual ACM Southeast Conference*. 1–6.

[77] Ruixuan Wang, Siyi Lv, Xiang Li, Haoshuai Gong, Zheli Liu, Tong Li, and Liang Guo. 2025. BOMAP: A Round-Efficient Construction of Oblivious Maps. *IEEE Transactions on Dependable and Secure Computing* (2025).

[78] Songlei Wang, Yifeng Zheng, Xiaohua Jia, Hejiao Huang, and Cong Wang. 2022. OblivGM: Oblivious attributed subgraph matching as a cloud service. *IEEE Transactions on Information Forensics and Security* 17 (2022), 3582–3596.

[79] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T.-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. 2014. Oblivious Data Structures. In *ACM CCS 2014: 21st Conference on Computer and Communications Security*, Gail-Joon Ahn, Moti Yung, and Ninghui Li (Eds.). ACM Press, Scottsdale, AZ, USA, 215–226. https://doi.org/10.1145/2660267.2660314

[80] Zhiqiang Wu and Rui Li. 2023. OBI: a multi-path oblivious RAM for forward-and-backward-secure searchable encryption.. In *NDSS*.

[81] Haoxuan Xie, Yixiang Fang, Yuyang Xia, Wensheng Luo, and Chenhao Ma. 2023. On querying connected components in large temporal graphs. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–27.

[82] Zikai Ye, Xiangyu Wang, Zesen Liu, Dan Zhu, and Jianfeng Ma. 2025. OBIR-tree: An Efficient Oblivious Index for Spatial Keyword Queries on Secure Enclaves. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–24.

[83] Chao Zhang, Angela Bonifati, and M Tamer Özsu. 2023. Indexing Techniques for Graph Reachability Queries. *arXiv preprint arXiv:2311.03542* (2023).

[84] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. 2017. Opaque: An oblivious and encrypted distributed analytics platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 283–298.

[85] Jinwei Zhu, Kun Cheng, Jiayang Liu, and Liang Guo. 2021. Full encryption: an end to end encryption mechanism in GaussDB. *Proc. VLDB Endow.* 14, 12 (July 2021), 2811–2814. https://doi.org/10.14778/3476311.3476351

[86] Jinhao Zhu, Liana Patel, Matei Zaharia, and Raluca Ada Popa. 2025. Compass: encrypted semantic search with high accuracy. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. 915–938.