# 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 the well-known neighbor query and $t$-hop query.

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

## 1 INTRODUCTION

Encrypted databases (EDBs) [30, 38, 72, 74, 84] have emerged as a practical and promising solution in cloud computing, particularly as database outsourcing becomes increasingly popular in business settings [42, 98]. By leveraging the computational power and storage capacity of remote servers, the client (also referred to as the data owner in literature [12, 94]) 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 [21, 49, 52, 72, 74]. 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 [26, 40, 45, 46, 53, 54]. This vulnerability has underscored the need for oblivious algorithms [23, 50, 65], 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.

*Xinle Cao is the corresponding author.

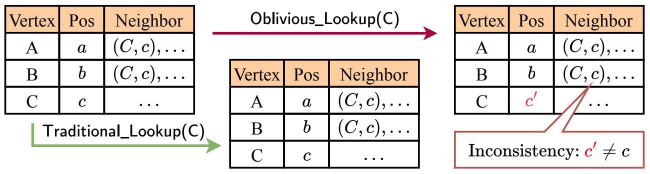


**Figure 1: The conflict between index-free design and oblivious algorithms. Obliviousness requires the position of $C$ to be changed after visiting $C$, thus incurs inconsistency.**

### 1.1 The Conflict in Oblivious Graph

While obliviousness has been a central topic [15, 17, 23, 50] 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, 15, 97, 99]. As highlighted by Appan et al. [4], all existing approaches do not realize *index-free adjacency* (IFA) [5, 70], a fundamental property of native graph databases such as Neo4j [68] 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:

- **Non-oblivious algorithms** do not necessarily change the positions of vertices. When the client executes a `lookup` for $C$ 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 $B$.

| Solution | **Obliviousness** | **Scenarios** | | **Neighbor Query** | |
|---|---|---|---|---|---|
| | | Client/Server | Enclave-based | Index-free | Tool/Technique used |
| Opaque [97] | ○ | – | ✓ | ✗ | *complex* relational databases |
| Compass [99] | ◐ | ✓ | – | ✗ | *linear* client-side storage |
| GraphOS [15] | ● | – | ✓ | ✗ | *expensive* oblivious map |
| AHR [4] | ○ | ✓ | – | ✓ | *insecure* smart pointer |
| Grove | ◐ | ✓ | – | ✓ | *secure and practical* delayed duplication |

[a] ●= Double obliviousness, ◐= Obliviousness, ○=Relaxed obliviousness
[b] Double obliviousness [15, 66] implies the enclave-based EDBs achieve obliviousness in both server's storage and hardware enclaves.

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

## 1.2 The Shortage of Existing Solutions

The conflict between IFA and obliviousness actually has motivated plenty of works [6, 15, 89, 97, 99] 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, but at the cost of security relaxation. We outline the features and limitations of representative approaches; a summary appears in Table 1.

- *Opaque* [97] supports graph data and queries via oblivious relational databases. While such a conversion is common in traditional databases [87], achieving it efficiently and obliviously is complex. As noted by [15], Opaque has to execute costly oblivious joins [50] to support BFS traversal and still leaks unexpected sensitive information.
- *Compass* [99] targets the well-known HNSW algorithm [62], 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 [56].
- *GraphOS* [15] 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 vis a novel technique named *smart pointer*. Nevertheless, it relaxes the security guarantee and thus does not truly achieve oblivious graph processing. For example, the adversary may observe the access frequency of a vertex in this protocol, which violates the requirement of obliviousness.

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, 15], 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 [15] 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, 63]. The adversary, denoted as $\mathcal{A}$, is assumed to be *honest-but-curious*, i.e., it honestly executes the protocols defined by the client but is curious about sensitive information. It therefore attempts to infer database contents from its observations. Regarding capabilities, we assume $\mathcal{A}$ can compromise the server for extended periods to observe the encrypted database and query execution, as modeled by an untrusted cloud server. A significant point recently concerned is whether $\mathcal{A}$ can inject queries [2, 39, 61]. While disallowing $\mathcal{A}$ to inject queries may bring new performance gain on KV stores [61] and even other types of databases [2], we follow traditional oblivious graph works [4, 15, 97] 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 [15] 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 recent oblivious works [2, 11, 17], we exclude the timing attacks and malicious adversaries; please refer to [19, 31, 41] for corresponding countermeasures.

**Client/Server Paradigm.** This work follows many prior obliviousness systems [2, 10, 17, 32, 82] 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 [88]. Another line of works [23, 66, 85] utilize enclaves inside the server as the avatar of the client, avoiding the interaction delay; their bottleneck lies in the bandwidth, i.e., the number of items accessed for obliviously accessing one item. While Grove is compared with others primarily in the client/server model, we also demonstrate it requires less bandwidth and thus has the potential to benefit enclave-based oblivious graph processing. Besides, another important assumption [4, 10, 11] we adopt under this model is the client-side storage costs cannot scale *linearly* to the number of vertices or edges for practicality.

## 2.2 Related Work

As many applications model their data as graphs (e.g., biological networks [71]), 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, 18, 25, 34, 64] have been proposed to process encrypted graph data and queries. These schemes encrypt the graph while permitting limited leakage during query processing to enable specific functionality. For example, Liu et al. [59] leak the order of distances between vertices to support shortest-distance queries. Compared with oblivious graph processing, such approaches generally reveal more leakage to the adversary and are therefore more vulnerable to attacks [27, 35]. 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. [91] are the first to consider oblivious graph processing, but they limit the scope to specific graphs (e.g., trees) and thus avoid the conflict with IFA. OblivGM [89] supports oblivious attributed subgraph matching but requires multiple non-colluding servers. Keller et al. [47] also propose the oblivious algorithm for Dijkstra's shortest path algorithm [22] 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 or implementation. In summary, our work fulfills the study for oblivious graph processing especially in enabling IFA under the typical setting of obliviousness, and proposes a more secure and practical system.

## 2.3 Preliminary

We introduce the notation and brief construction of essential primitives used throughout the remainder of the paper.

**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 $L-1$. For ease of presentation, we denote the client and server 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 throughout this paper, it is natural to support directed graphs by marking the direction of edges.

**Oblivious RAM (ORAM).** The fundational mechanism we rely on is ORAM, which is originally proposed to achieve obliviousness on accessing memory [37]. 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, 17, 23], 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.* It consists of a constant number of 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 $pn \xleftarrow{\$} \{0, 1, ..., 2^l\}$, 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 next ORAM accesses.
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 $pt'$. Its ciphertext is stored in either the path associated to $pt'$ or the client-side stash. This validates our claim in Section 1.1 that oblivious algorithms update a vertex's position after visiting it.

**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 [23, 91] organize KV stores as a search tree like AVL tree. The OMAP functionalities are then achieved via oblivious operations on the tree. For instance, $C$ can obliviously traverse the search tree for oblivious lookup. In detail, each tree node is treated as an item to be stored in the ORAM tree.

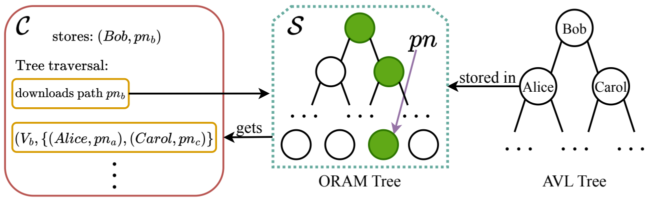


**Figure 2: Example of oblivious AVL tree. Each tree node is stored in the ORAM tree randomly. For tree traversal, $C$ visits a node (e.g., root node *Bob*) to get the keys and path number of its children (e.g., $\{(Alice, pn_a), (Carol, pn_c)\}$).**

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.

## 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 for oblivious graph processing.

### 3.1 Problem Statement

We first outline the problem context in oblivious graph processing. As introduced in Section 2.3, 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 [68, 73, 96] is to adopt index-free adjacency (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 the identifiers, each vertex should preserve the 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 for obliviousness. In detail, consider $C$ visits a vertex $v$ in oblivious graph and thus is going to update 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 [99] 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 [15] 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 a relatively expensive cryptographic primitive in practice [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 $K$ neighbors $V_1 = \{v_1, \ldots, v_K\}$, where $K \geq 2$. 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 $K$ times, as now $C$ must download $K + 1$ 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. This leads to a recursive consistency maintenance problem—resolving one inconsistency spawns $K$ new ones.

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 insert a new **small** block containing the message $\{v_0.pn \rightarrow v_0.pn'\}$ along 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, 76, 83], 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 $v$.id, position $v.pn$, and attribute values $v$.value. Then for each of its neighbors, $v$ should maintain the identifier and position of this neighbor for IFA, and also the edge between them for weighted graph. Accordingly, the data structure (KV pair) of a vertex $v$ is defined as follows:

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

  where $K$ denotes the number of $v$'s neighbors, and $g_i$ denotes the neighbor of $v$. For obliviousness, we padding the neighbor number of each vertex to the maximal number in the graph.
- **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

$$\{v.\text{id}, v.pn', pn_i\}$$

  where $v.pn'$ is the updated position of $v$. 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 the reference to $v$'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 [86].

*3.2.2 Duplication Insertion (C2).* The introduction of delayed duplications can incur new communication overhead. Consider a vertex $v$ with $K$ neighbors: each vertex access operation generates $K$ delayed duplication to notify the $K$ 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 path. So the insertion of $K$ duplications implies $C$ needs to retrieve and upload $K$ 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 $K$ 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 $K$ 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, $C$ cannot select $K$ random paths to insert duplications. In the original Path ORAM [82], random path selection suffices for writing items into the ORAM tree because each item is assigned to a uniformly random path during eviction. In contrast, duplications are deterministically mapped to fixed paths based on graph topology. To illustrate, suppose $C$ accesses the same vertex $v$ repeatedly for $\mu$ times. For its neighbor $g_i$ located at path $pt_i$, $C$ must notify $g_i$ the position updates with $\mu$ duplications. Moreover, all the $\mu$ duplications are inserted towards the same path $pt_i$, which is basically different from the randomized mechanism in Path ORAM. Consequently, retrieving $K$ 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 [33, 76]: $C$ selects the $K$ 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. As previously noted, when a vertex $v$ is visited for multiple times, $C$ may generate multiple duplications for the same neighbor, all of which are inserted into the same path. Without an effective de-duplication mechanism, overflow becomes highly likely due to the limited capacity of meta blocks. We formalize the objectives of de-duplication as follows:

(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 $\{v.\text{id}, v.pn', pn_i\}$ can be uniquely identified by the tuple $(v.\text{id}, pn_i)$, 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. Combined with the earlier insertion strategy, the de-duplication mechanism enables a rigorous analysis of overflow in the following section.

## 3.3 Overflow Analysis

We now derive the appropriate number of meta blocks to allocate per bucket, denoted by $Y$. We first describe the *reverse lexicographical (RL)* path selection [33, 76] and its key properties, and then propose our results about overflow.

**RL Path Selection.** Within a height-$L$ ORAM tree, path selection is governed by an $L$-bit integer $s := s_{L-1}s_{L-2}\dots s_0$. Starting from the root node at level 0, the traversal direction at each branching point is determined by the current bit value: a left branch is taken when $s_0 = 0$, and a right branch when $s_0 = 1$. Proceeding sequentially through the bit sequence from $s_0$ to $s_{L-1}$ yields a unique path through the tree structure. The RL path selection variant employs *modular* arithmetic for path evolution, executing the increment operation $s \leftarrow s+1$ after each path selection. As demonstrated in [33, 76], this strategy guarantees aggressive eviction for the ORAM tree to keep small client-side storage costs. It exhibits important properties and guarantees Theorem 3.1 for delayed duplications.

Property 1. *For a height-L ORAM tree under RL path selection, any level-j bucket ($0 \le j < L$) satisfies:*

- *the meta blocks associate with it are deterministically accessed once every $2^j$ path selections.*
- *if the i-th path takes its left (resp. right) branch, the $(i+2^j)$-th path will deterministically take its right (resp. left) branch.*

Theorem 3.1. *Suppose that, on input an integer n, 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 K new duplications, and K additional paths are chosen via the RL path selection for inserting the duplications. Then, after m accesses, the probability of overflow is bounded by:*

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

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

Proof. We derive the probability bound through three steps. First, we introduce a simplified eviction strategy with lower meta block utilization than our actual strategy, serving as a relaxation for analysis. Second, we bound the maximum duplications per bucket in the ORAM tree under this simplified strategy. Third, we compute the overall overflow probability by aggregating individual bucket overflow probabilities across the tree.

Following the theorem's assumptions, each ORAM access generates $K$ duplications and thus $C$ selects meta blocks in another $K$ paths for inserting duplications. For simplicity, we treat the parallel $K$ path selections as $K$ sequential selections to insert the $K$ duplications one by one. This does not affect our analysis as parallel selections and insertions actually imply a higher utilization [10, 90, 93].

(1) **Simplified Eviction Strategy.**
In addition to the de-duplication mechanism described in Section 3.2.3, we employ a simplified eviction strategy for handling duplications. Given one path whose meta blocks are downloaded, all duplications are pushed into this path as deeply as possible, irrespective of whether there are empty meta blocks. An overflow is triggered if any bucket in this path is required to store more than $Y$ duplications.

(2.1) **Overflow in one Bucket for $\alpha$ Duplications.**
We now analyze the overflow probability within a bucket $\mathcal{B}$ at level $j$ along the path selected by the RL strategy. The de-duplication strategy guarantees we only need to consider duplications with distinct identifier pairs, simplifying the analysis. The simplified eviction strategy guarantees that a newly inserted duplication remains in bucket $\mathcal{B}$ iff the path number of its target neighbor lies in the left (resp. right) subtree of $\mathcal{B}$, while the selected path lies in the right (resp. left) subtree of $\mathcal{B}$. This event occurs with probability $2^{-j-1}$. Given $\alpha$ distinct duplications to be inserted—each targeting different neighbors—let $X$ denote the number of duplications that remain in $\mathcal{B}$. Then, the overflow probability in $\mathcal{B}$ is bounded by the event that more than $Y$ duplications remain, i.e., $\Pr[X > Y]$.
However, as discussed in Section 3.2.2, up to $K$ duplications may target the same neighbor and path number. In other words, these duplications can be treated as a whole in eviction instead of independent evictions, extending the duplication size by $K$. Therefore, we calculate a very relaxed upper bound of overflow probability by directly requiring there are at most $\lfloor \frac{Y}{K} \rfloor$ duplications to remain in $\mathcal{B}$:

$$\Pr\left[X \le \left\lfloor \frac{Y}{K} \right\rfloor\right] = \sum_{i=0}^{\lfloor Y/K \rfloor} \binom{\alpha}{i} \left( \frac{1}{2^{j+1}} \right)^i \left( 1 - \frac{1}{2^{j+1}} \right)^{\alpha - i}.$$

(2.2) **Combine all Accesses and Duplications.**
For a bucket $\mathcal{B}$ at level $j$, Property 1 implies it is regularly selected by the RL strategy every $2^j$ accesses. Between two consecutive selections to it, meta blocks of $\mathcal{B}$ temporarily store all duplications whose path numbers lies in the same subtree of $\mathcal{B}$. For example, if the first selection is the left branch, all duplications towards the right branch are kept in $\mathcal{B}$'s meta blocks. Upon the next selection to the right branch, these retained duplications are pushed further down, and $\mathcal{B}$'s meta blocks then hold duplications towards the left branch. Note between two regular selections, at most $2^j$ new duplications are inserted into the ORAM tree. So the probability that overflow happens on $\mathcal{B}$ can be calculated by letting $\alpha = 2^j$:

$$\Pr\left[X > \left\lfloor \frac{Y}{K} \right\rfloor\right] = 1 - \sum_{i=0}^{\lfloor Y/K \rfloor} \binom{2^j}{i} \left( \frac{1}{2^{j+1}} \right)^i \left( 1 - \frac{1}{2^{j+1}} \right)^{2^j - i}.$$

The theorem assumes $K$ duplications are produced per ORAM access. We now bound the probability that an overflow occurs in bucket $\mathcal{B}$ within these $m$ ORAM accesses. Since $\mathcal{B}$ is accessed every $2^j$ path selection, the total number of such intervals is at most $\left\lceil \frac{mK}{2^j} \right\rceil$. Thus, the overflow probability in $\mathcal{B}$ is at most:

$$\left\lceil \frac{mK}{2^j} \right\rceil \Pr\left[X > \left\lfloor \frac{Y}{K} \right\rfloor\right].$$

This is induced from the union bound over events $\{A_i\}_{i\in[k]}$ [28], which states that:

$$\Pr\left[\bigcup_{i=1}^{k} A_i\right] \le \sum_{i=1}^{k} \Pr[A_i] \ . \tag{1}$$

(3) **Combine all Buckets in the Tree.**
Finally, we compute the probability that an overflow occurs in any bucket of the ORAM tree over $m$ ORAM accesses. Applying Equation 1 across all levels of the ORAM tree, we obtain:

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

where $\sigma$ is the smallest integer such that $2^\sigma > Y$. To ensure a negligible overflow probability, we require the entire expression to be less than $2^{-\lambda}$ where $\lambda$ is the security parameter. □

**Remark.** Actually, the upper bound derived in the proof is rather loose due to several strong simplifying assumptions:

(1) The simplified eviction strategy is not as aggressive as that in Path ORAM and does not utilize the client-side stash, thus the simplification relaxes the bound.
(2) We apply Equation 1 to calculate the upper bound of overflow in different periods of time and buckets. This inevitably amplifies the overall probability.
(3) Recall each ORAM tree also retrieves a complete path including meta blocks, which can also be used for inserting duplications, we do not count this in our proof.

In Table 2, we present the specific information about meta block settings according to real-world datasets. Given a graph $G = (E, V)$, we assume $K = 10$ and $m = n = |V|$ to simulate the storage overhead of meta blocks. It is shown that the storage cost of meta blocks is incremental compared with the total bucket storage cost. We will provide a more detailed and thorough evaluation of meta block cost (e.g., the bandwidth and processing time) in Section 6.

## 4 GROVE: GRAPH PROCESSING

As demonstrated by native graph databases (particularly *Neo4j* [68]), 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.

*4.0.1 Data Structures.* When presenting *delayed duplication* in Section 3.2, we assumed the client $C$ 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 the client 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 [99] 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 mechanism, based on the delayed

| Dataset | Size $\|V\|$ | Data | Meta | Pro |
|---|---|---|---|---|
| Enron email [48] | 36,692 | 12 KB | 1.5 KB | 11.1% |
| Amazon product [55] | 548,552 | 88 KB | 1.2 KB | 1.35% |
| VisualGenome [51] | 108,077 | 1.2 MB | 5.4 KB | 0.45% |

**Table 2: Meta block information in reality. Data and Meta refer to the storage for data and meta blocks per bucket. Pro is the proportion of meta blocks on the total bucket storage.**

duplication technique, to enable the oblivious retrieval of any target vertex with practicality, and finally support complete and efficient oblivious graph processing.

### 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 operation 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 mechanisms 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 the 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 implementation logic and interfaces to manage the four ORAM trees instead of additional complex mechanisms to handle meta blocks separately. With this optimization, the complete architecture of Grove is illustrated in Figure 3.

*4.1.1 Initialization.* During the initialization of Grove for a graph $G = \{(V, E)\}$, the client $C$ converts the graph to KV pairs for $\mathcal{T}_G$ and $\mathcal{T}_P$. For each 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 [57, 82] to initialize the ORAM trees and write the pairs into $(\mathcal{T}_P, \mathcal{T}_G)$. Interestingly, as there is no duplication in the initialization, the

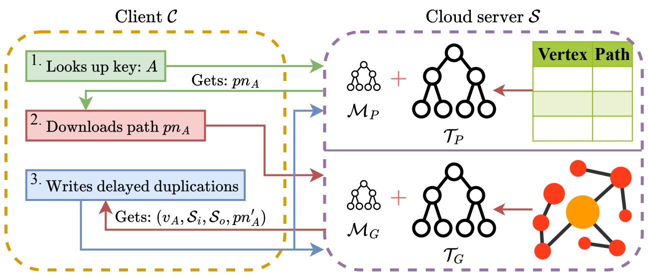


**Figure 3: The Complete Architecture of** Grove.

trees ($\mathcal{M}_P$, $\mathcal{M}_G$) can be directly initialized as empty where "empty" means there is even no random string inside meta blocks.

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, i.e., the union of real vertices and intermediate vertices.

## 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 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 graph 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) for obliviousness, 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 for correctness.

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 neighbor set $S$, then $S$ may include intermediate vertices.
(2) Once $C$ notice intermediate vertices in $S$, it retrieves all neighbors of vertices in $S$. Then $C$ replaces $S$ with the set of the 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` [81] is used to retrieve all vertices whose distances between the start vertex are no more than $t$. Its applications include relationship discovery [92], impact prediction [69], and friend recommendation [58]. 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 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 [20], shortest path query in unweighted graph [79], and even connected component queries [95]. We select `t-length traversal` [62] 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 [62]. Clearly, Grove can achieve $t$-length traversal by $t$ continual (optimized) graph queries.

Other graph queries can also be achieved similarly via trivially translating some classical algorithms with our oblivious interfaces. In this work, we actually focus on the acceleration of IFA to foundational interfaces and leave the deployment and optimizations for more complex graph queries in the future.

**Remark.** Grove targets the client/server scenario where the communication between $C$ and $\mathcal{S}$ is expensive. This makes Grove impractical to execute some classical graph algorithms like Dijkstra's algorithm [22] which need to traverse all vertices. To overcome this issue, some enclave-based graph EDBs are proposed. They significantly reduce the communication cost with an avatar of $C$ inside $\mathcal{S}$. They regard the bandwidth as the key performance factor. We

| 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 $\mathcal{T}_P$ calls + $\beta$ $\mathcal{T}_G$ calls |
| $t$-`hop query` | GraphOS | $(2(K+K^2+\ldots+K^t)+1)$ $\mathcal{T}_{OS}$ calls |
| | Grove | 1 $\mathcal{T}_P$ calls + $(1+K+\ldots+K^{t-1})\beta$ $\mathcal{T}_G$ calls |
| $t$-`length traversal` | GraphOS | $(2tK+1)$ $\mathcal{T}_{OS}$ calls |
| | Grove | 1 $\mathcal{T}_P$ calls + $\beta \cdot t$ $\mathcal{T}_G$ calls |
| `Insertion` | GraphOS | $(5K+1)$ $\mathcal{T}_{OS}$ calls |
| | Grove | 2 $\mathcal{T}_P$ calls + $2(\beta-K)$ $\mathcal{T}_G$ calls |

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

will show Grove has a much smaller bandwidth cost than the SOTA enclave-based graph EDB (GraphOS [15]), which implies Grove also has the potential to be applied within enclaves for enhanced graph query performance.

## 4.3 Extension to Dynamic Graphs

In this section, we describe insertion and deletion operations that extend Grove to dynamic graphs. We present them separately because IFA and graph sparsification lead to algorithmic differences across applications.

**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 PERFORMANCE AND SECURITY 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 [15], we will provide an empirical evaluation of cost on meta blocks in Section 6.

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. For example, both GraphOS and $\mathcal{T}_P$ in Grove are established via an OMAP, their complexities can vary based on different underlying OMAP protocols [10, 77, 91]. In general, we use the following 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 [15] 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 most cheaper, 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 query` in both Grove and GraphOS. The complexities can be calculated according to specific OMAP protocols, e.g., with OMAP in [15], the bandwidth for `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 [66, 80] and EDBs [72, 78] 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 [85]. 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.

The simulation-based security notion of Grove follows [14, 24, 72] 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 $Q = \{q_1, q_2, ..., q_m\}$. In the real world, $\mathcal{A}$ observes how Grove executes $Q$ on $G$ in the server side, denoted by $\mathsf{View}_{\mathcal{A}}^{\mathsf{Grove}}(G, Q, \lambda)$. On the contrary, in the ideal world, a simulator $\mathsf{Sim}$ simulates the execution given the leakage $\{\mathcal{L}_1(G), \mathcal{L}_2(Q)\}$ where $\mathcal{L}_2(Q) := \{\mathcal{L}_2(q_1), ..., \mathcal{L}_2(q_m)\}$. We denote the $\mathcal{A}$'s view in the ideal world as $\mathsf{View}_{\mathcal{A}}^{\mathsf{Sim}}(\mathcal{L}_1(G), \mathcal{L}_2(Q), \lambda)$. Then the security guarantee holds as the following theorem.

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

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

Proof. We defer the detailed proof to Appendix A of the extended version [29]. Here we give a brief security intuition. The semantically secure encryption in Grove guarantees that $\mathcal{A}$ can infer information only through access patterns, as timing attacks are excluded. Then Grove's obliviousness property guarantees the accesses to the ORAM trees leak only the permitted information. For example, a `Lookup` operation always sequentially picks up $h$ random paths in $\mathcal{T}_P$ where $h = \lceil 1.44 \log |V'| \rceil$, then a random path in $\mathcal{T}_G$, and finally $D$ paths on $\mathcal{M}_G$. The execution trace is fixed and hence can be simulated by $\mathsf{Sim}$ with the values of $|V'|$ and $D$. $\square$

## 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 [15] 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, as edge operations are completed via accessing the corresponding vertices under IFA. Next, we detail our experimental setup, datasets, and metrics. Both Grove and GraphOS use the same security parameters, hardware, and batching strategies. As GraphOS was originally designed within enclaves, we adapt it to enable a fair comparison. Our results show that delayed duplication adds only modest overhead; a small increase in client-side storage can substantially reduce server-side storage and improve query performance; and Grove consistently outperforms GraphOS across all workloads considered. In addition, we release open-source implementations of Grove and a client/server-optimized GraphOS at https://github.com/weiqins/daoram.

**Experimental Setup.** We implement all components in Python 3.12 and adopt an AVL-tree-based OMAP [15, 91] in GraphOS as our underlying OMAP due to its highly parallelizable nature. For ORAM trees $\mathcal{T}_P$ and $\mathcal{T}_G$, each bucket has four blocks. All cryptographic primitives are provided by the pycryptodome package [1]. The entities $C$ and $\mathcal{S}$ run on separate machines and communicate over a WAN. To reflect realistic conditions, $C$ is in Virginia, USA and $\mathcal{S}$ is in California, USA; the measured average round-trip latency between them is 35 ms. $\mathcal{S}$ is an Alibaba Cloud instance with an Intel Xeon Platinum 8160 with 32 cores at 2.10 GHz and 64 GB of memory, while $C$ is a more constrained instance with 2 vCPUs from an Intel Xeon Platinum 8269CY at 2.50 GHz and 8 GB of memory. The bandwidth between the two instances is fixed at 100 Mbps, as the performance bottleneck comes from the number of interaction rounds rather than the bandwidth itself.

**Datasets.** Consistent with prior oblivious works [10, 16, 43] and also GraphOS [15], we use synthetic datasets that vary vertex number $|V|$, edge number $|E|$ and the maximum vertex degree $K$ due to the obliviousness property [16]. We set $|V|$ from $2^{10}$ to $2^{20}$ and equip them with proportional edges, i.e., $|E| = K|V|$. For each $|V|$, we evaluate $K = 10$ and $K = 100$. When evaluating Grove with $K = 100$, we sparsify the 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 4 KB and 22 KB, following the real datasets in Table 2.

**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 as the efficiency metric when interaction delay is excluded. Notably, bandwidth in obliviousness refers to the communication volume. Under the client/server model, bandwidth alone has limited impact on end-to-end efficiency because interaction cost dominates. We nevertheless report bandwidth to highlight the potential of running Grove inside secure enclaves.

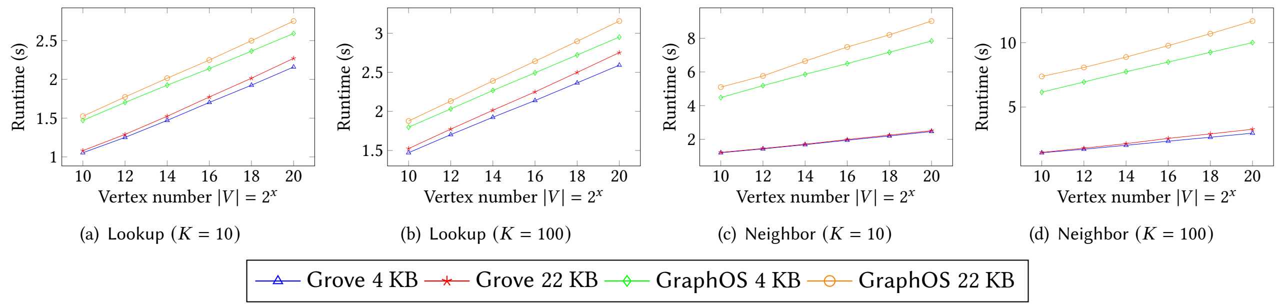
### 6.1 Evaluation of Delayed Duplications

We first measure the overhead introduced by delayed duplication in enabling the IFA design for oblivious graphs. In Section 3.3, we have used Table 2 to show that the storage costs of meta blocks are very limited since each meta block consists of only several integers. With the vertex size increasing, the storage overhead is more and more incremental. For example, when the vertex size is over 22 KB [55], the storage overhead is no more than 1.35%. As vertices in graphs often have large size like 300 KB in [51], the storage costs on meta blocks actually are ignorable relative to graph storage.

For query processing, we recall that duplications do not occupy additional interaction rounds due to our piggyback strategy. So we use Table 4 to present the proportions of processing duplication relative to the total query processing on only bandwidth and runtime. We generate graphs with different vertex numbers and vertex sizes under $K = 10$ to observe the scalability of the duplication technique. We pick up the foundational query `Lookup` and `Neighbor` query as the test objects. It is shown that ❶ the overhead is stable under different vertex numbers, validating its great scalability; ❷ similar to storage costs, the bandwidth and runtime overhead significantly decrease with vertex size increases from 4 KB to 22 KB,

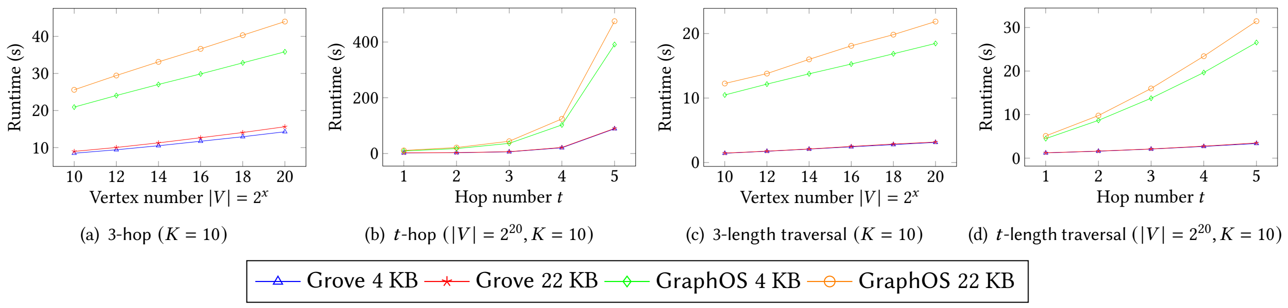
| Graph Size | | $\lvert V\rvert = 2^{10}$ | | $\lvert V\rvert = 2^{14}$ | | $\lvert V\rvert = 2^{18}$ | | $\lvert V\rvert = 2^{20}$ | |
|---|---|---|---|---|---|---|---|---|---|
| **Metrics** | | Bandwidth | Runtime | Bandwidth | Runtime | Bandwidth | Runtime | Bandwidth | Runtime |
| Lookup | 4 KB | 29.72% | 1.00% | 28.57% | 0.91% | 25.45% | 0.85% | 24.35% | 0.81% |
| | 22 KB | 11.87% | 0.43% | 11.92% | 0.42% | 11.79% | 0.42% | 11.63% | 0.40% |
| Neighbor | 4 KB | 43.09% | 1.48% | 50.15% | 1.68% | 53.12% | 1.82% | 53.90% | 1.90% |
| | 22 KB | 13.13% | 0.44% | 17.11% | 0.58% | 19.20% | 0.67% | 19.90% | 0.71% |

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



(a) Lookup ($K = 10$)
(b) Lookup ($K = 100$)
(c) Neighbor ($K = 10$)
(d) Neighbor ($K = 100$)

**Figure 4: Performance of fundamental operations.**



(a) 3-hop ($K = 10$)
(b) $t$-hop ($|V| = 2^{20}, K = 10$)
(c) 3-length traversal ($K = 10$)
(d) $t$-length traversal ($|V| = 2^{20}, K = 10$)

**Figure 5: Performance of graph queries.**

implying the duplications can perform better in applications with larger vertex size; ❸ the processing time for duplications costs only $0.40\% \sim 1.90\%$ of the total runtime, which is clearly incremental, although the bandwidth overhead is substantial. The gap between bandwidth and runtime overhead is because the interaction rounds are the main bottleneck under the client/server model. Remarkably, despite the $11.63\% \sim 53.90\%$ 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 Appendix B.1 of the extended version [29]. We also conduct an additional experiment to quantify the client/server storage trade-off by using smaller buckets for delayed duplications and to measure the resulting effect on processing time.

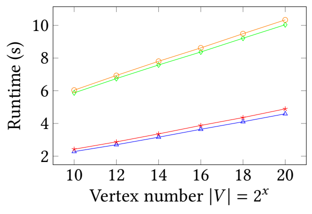
## 6.2 Comparison between Systems

We begin our comparison of the two systems by characterizing their setup performance and server-side storage. Grove outperforms GraphOS in both metrics, particularly for dense graphs. This advantage stems from the fact that GraphOS stores both vertices and edges in the underlying OMAP, whereas Grove stores edges within vertices, resulting in an OMAP size determined solely by the number of vertices. Empirically, Grove achieves a setup speedup of 80.6% to 94.6% and reduces server-side storage by 74.6% to 92.5% across datasets with $|V|$ ranging from $2^{10}$ to $2^{20}$. We provide detailed breakdowns of these results, along with a comparison of client storage, in Appendix B.2 of the extended version [29].

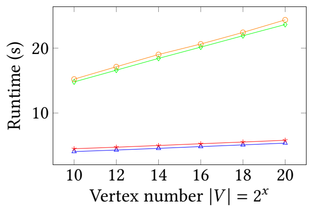
*6.2.1 Foundational Operations.* We evaluate the performance of Grove and GraphOS on foundational operations, namely Lookup and `Neighbor` query. We omit Update since it adopts the same interface as Lookup. Results for runtime, interaction rounds, and bandwidth are shown in Figure 4 and Table 5. For both operations, we observe Grove outperforms GraphOS across all three metrics.

Although IFA is designed to mainly optimize graph traversal, Grove still performs Lookup faster, with fewer interaction rounds and lower bandwidth, as shown in Table 3. This results from both systems relying on an underlying OMAP while Grove uses a smaller one. In Figure 4, across all tested graph sizes with 4 KB value, Grove is 13.2% to 28.6% faster than GraphOS in Lookup. The performance gain narrows as $K$ increases because the additional intermediate vertices enlarge the OMAP used by Grove, reducing the OMAP size gap between the two systems. Larger vertex values have the opposite effect: when $K = 10$ and the value is 22 KB, the maximum improvement rises to 39.9%. The increase results from GraphOS having steeper bandwidth growth with value size, since its edge-retrieval step is as expensive as a vertex retrieval.

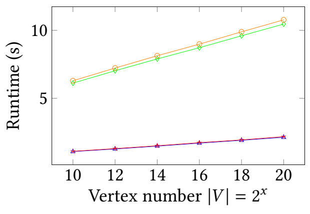
The speedup on `Neighbor` query is more substantial since this operation involves neighbor traversal, which IFA targets to improve. In Figures 4(c) and 4(d), the speedup ranges from 68.5% to 76.1%.



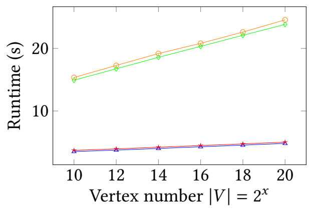
(a) Insertion ($K = 10$)



(b) Insertion ($K = 100$)



(c) Deletion ($K = 10$)



(d) Deletion ($K = 100$)

**Figure 6: Performance of insertion and deletion.**

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 3. 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 MB to 26 MB when $|V| = 2^{20}$ with 22 KB vertex values.

*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 5, we vary the vertex count $|V|$ and hop number $t$ independently to evaluate the $t$-hop query. With $t$ fixed, Grove achieves speedups between 59.4% and 64.1%. 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 $t$ (for example, when retrieval downloads the entire dataset), 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 68.5% to 77.9% in runtime and improvement of 40.3% to 42.4% on bandwidth.

We evaluate the $t$-length traversal using the same design, varying $t$ and $|V|$ independently, as shown in Figure 5. For the same reasons as above, 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 runtime speedups, consistent with the results that at $t = 1$ Grove achieves a speedup of 76.1% and at $t = 5$ a speedup of 88.2%.

*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 6, Grove achieves an insertion speedup of 52.5% to 70.1%. 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 6 shows that Grove improves deletion performance by 75.5% to 81.9%.

| Operation | Grove | | GraphOS | |
|---|---|---|---|---|
| | Round | Bandwidth | Round | 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 |

**Table 5: Comparison on interaction round and bandwidth under $|V| = 2^{20}$, $K = 10$, and vertex size of 22 KB.**

## 7 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 [15] under the client/server paradigm. We release an open-source library at https://github.com/weiqins/daoram to support further development of oblivious graphs.

## 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 [60], 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] Srdjan Capkun and Franziska Roesner (Eds.). 2020. *USENIX Security 2020: 29th USENIX Security Symposium*. USENIX Association.

[14] 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

[15] 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

[16] 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.

[17] 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

[18] 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

[19] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. 2017. Detecting Privileged Side-Channel Attacks in Shielded Execution with Déjà Vu. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security* (Abu Dhabi, United Arab Emirates) *(ASIA CCS '17)*. Association for Computing Machinery, New York, NY, USA, 7–18. https://doi.org/10.1145/3052973.3053007

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

[21] Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Saurabh Shintre. 2020. SEAL: Attack Mitigation for Encrypted Databases via Adjustable Leakage, See [13], 2433–2450.

[22] 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

[23] 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

[24] Sky Faber, Stanislaw Jarecki, Hugo Krawczyk, Quan Nguyen, Marcel-Catalin Rosu, and Michael Steiner. 2015. Rich Queries on Encrypted Data: Beyond Exact Matches. In *ESORICS 2015: 20th European Symposium on Research in Computer Security, Part II (Lecture Notes in Computer Science)*, Günther Pernul, Peter Y. A. Ryan, and Edgar R. Weippl (Eds.), Vol. 9327. Springer, Cham, Switzerland, Vienna, Austria, 123–145. https://doi.org/10.1007/978-3-319-24177-7_7

[25] Francesca Falzon, Esha Ghosh, Kenneth G. Paterson, and Roberto Tamassia. 2024. PathGES: An Efficient and Secure Graph Encryption Scheme for Shortest Path Queries, See [60], 4047–4061. https://doi.org/10.1145/3658644.3670305

[26] 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

[27] 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.

[28] William Feller et al. 1971. *An introduction to probability theory and its applications*. Vol. 963. Wiley New York.

[29] Weiqi Feng, Xinle Cao, Adam O'Neil, and Chuanhui Yang. 2025. Enabling Index-free Adjacency in Oblivious Graph Processing with Delayed Duplications. (2025). https://github.com/WeiqiNs/daoram

[30] 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

[31] Christopher W. Fletchery, Ling Ren, Xiangyao Yu, Marten Van Dijk, Omer Khan, and Srinivas Devadas. 2014. Suppressing the Oblivious RAM timing channel while making information leakage and program efficiency trade-offs. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 213–224. https://doi.org/10.1109/HPCA.2014.6835932

[32] 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

[33] 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

[34] 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.

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

[36] Oded Goldreich. 2004. *Foundations of Cryptography, Volume 2*. Cambridge university press Cambridge.

[37] 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

[38] Paul Grubbs, Anurag Khandelwal, Marie-Sarah Lacharité, Lloyd Brown, Lucy Li, Rachit Agarwal, and Thomas Ristenpart. 2020. Pancake: Frequency Smoothing for Encrypted Data Stores, See [13], 2451–2468.

[39] 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.

[40] 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.

[41] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. 2017. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 217–233. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/gruss

[42] 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.

[43] 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.

[44] IEEE S&P 2020 2020. *2020 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Francisco, CA, USA.

[45] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation, See [67].

[46] 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

[47] 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

[48] 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

[49] Evgenios M. Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. 2020. The State of the Uniform: Attacks on Encrypted Databases Beyond the Uniform Query Distribution, See [44], 1223–1240. https://doi.org/10.1109/SP40000.2020.00029

[50] 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

[51] 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

[52] 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

[53] 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.

[54] 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

[55] 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

[56] 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.

[57] 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.

[58] Yakun Li, Lei Hou, and Juanzi Li. 2023. Preference-aware graph attention networks for cross-domain recommendations with collaborative knowledge graph. *ACM transactions on information systems* 41, 3 (2023), 1–26.

[59] 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.

[60] 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.

[61] 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.

[62] 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.

[63] 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.

[64] Xianrui Meng, Seny Kamara, Kobbi Nissim, and George Kollios. 2015. GRECS: Graph Encryption for Approximate Shortest Distance Queries, See [75], 504–517. https://doi.org/10.1145/2810103.2813672

[65] 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

[66] 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.

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

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

[69] Jianwei Niu, Jinyang Fan, Lei Wang, and Milica Stojinenovic. 2014. K-hop centrality metric for identifying influential spreaders in dynamic large-scale social networks. In *2014 IEEE Global Communications Conference*. IEEE, 2954–2959.

[70] 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.

[71] 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.

[72] 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

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

[74] 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

[75] Indrajit Ray, Ninghui Li, and Christopher Kruegel (Eds.). 2015. *ACM CCS 2015: 22nd Conference on Computer and Communications Security*. ACM Press, Denver, CO, USA.

[76] 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.

[77] Daniel S Roche, Adam Aviv, and Seung Geol Choi. 2016. A practical oblivious map data structure with secure deletion and history independence. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 178–197.

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

[79] 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.

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

[81] 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.

[82] 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

[83] Emil Stefanov, Elaine Shi, and Dawn Xiaodong Song. 2012. Towards Practical Oblivious RAM, See [67].

[84] 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

[85] 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.

[86] 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.

[87] 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.

[88] 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).

[89] 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.

[90] Xiao Wang, Hubert Chan, and Elaine Shi. 2015. Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 850–861.

[91] 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

[92] Yan Wang, Amar Henni, Azade Fotouhi, and Leopold Mvondo Ze. 2022. Leveraging K-hop based Graph for the Staffing Recommender System with Parametric Geolocation. In *2022 International Conference on Computational Science and Computational Intelligence (CSCI)*. IEEE, 664–669.

[93] Wai Kit Wong, David Wai-lok Cheung, Ben Kao, and Nikos Mamoulis. 2009. Secure kNN computation on encrypted databases. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data* (Providence, Rhode Island, USA) *(SIGMOD '09)*. Association for Computing Machinery, New York, NY, USA, 139–152. https://doi.org/10.1145/1559845.1559862

[94] Wai Kit Wong, Ben Kao, David Wai Lok Cheung, Rongbin Li, and Siu Ming Yiu. 2014. Secure query processing with data interoperability in a cloud database environment. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 1395–1406.

[95] 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.

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

[97] 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.

[98] 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

[99] 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.

## A SECURITY PROOF

We now prove Theorem 5.1. The proof can be completed with game transitions on a hybrid game. In this game, the data structures in the server side consist of only random strings instead of ciphertexts. To keep the distribution of access patterns on them identical to $\mathsf{View}_{\mathcal{A}}^{\mathsf{Grove}}(G, Q, \lambda)$, the client $C$ runs $Q$ on $G$ with local Grove to produce the access patterns. In particular, it addresses overflow on meta blocks by directly padding new additional meta blocks when overflow happens. Denote the view of the adversary in this hybrid game as $\mathsf{View}_{\mathcal{A}}^{\mathsf{Hybrid}}(G, Q, \lambda)$. Clearly, this hybrid game actually only replaces semantically secure ciphertexts in the server side with random strings and remove the negligible overflow probability. Then the security of semantically secure encryption [36] guarantees:

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

Otherwise, we can construct an adversary via them to break the underlying semantically secure encryption scheme. Now we construct a simulator Sim to prove the views in the hybrid game and ideal world are indistinguishable.

(1) Sim sets up the ORAM trees in the server side according to $\mathcal{L}_1(G)$ and fill them with random strings. The sizes of these ORAM trees are identical to those in the real world since they are solely determined by $\mathcal{L}_1(G)$, e.g., the vertex number (after sparsification) $|V'|$ determines the height of ORAM trees.

(2) For query $q_i$, Sim produces the access patterns on the simulated ORAM trees with both $\mathcal{L}_2(q_i)$ and $\mathcal{L}_1(G)$. The two leakages decide how to produce the access patterns, i.e., the interaction rounds and the number of paths in each ORAM tree to retrieve in each round. So Sim follows the two leakages to issue interaction and retrieve random paths. Here we list two foundational queries from { `Lookup`, `Update`, `Neighbor query`, $t$-`hop query`, $t$-`length traversal` } to explain the simulation.
   - The Lookup operation fixes an execution trace with the following sequential ORAM accesses: first $h$ ORAM accesses to $\mathcal{T}_P$ where $h = \lceil \log |V'| \rceil$, then one ORAM access to $\mathcal{T}_G$, finally $D$ ORAM accesses to $\mathcal{M}_G$.
   - The `Neighbor query` fixes an execution trace with sequential ORAM accesses: first $h$ ORAM accesses to $\mathcal{T}_P$, then $1 + D^1 + \ldots + D^{w-1} + K$ accesses to $\mathcal{T}_G$ for the start vertex and its neighbors where $w$ is the smallest integer such that $D^w \geq K$, finally $K$ accesses to $\mathcal{M}_P$ and $K \cdot D$ accesses to $\mathcal{M}_G$.

   As each ORAM access to $\mathcal{T}_P$ and $\mathcal{T}_G$ just retrieve a random path while each ORAM access to $\mathcal{M}_P$ and $\mathcal{M}_G$ selects a deterministic path according to the reverse lexicographical order, SIM clearly can follow the execution trace to pick up paths for simulating the corresponding query with the permitted query type information $\mathcal{L}_2(q)$.

In this way, Sim performs identically to the server-side execution in the hybrid game except that they select ORAM paths and produce random strings independently. As both of them just select random paths on $(\mathcal{T}_P, \mathcal{T}_G)$ and the same deterministic paths on $(\mathcal{M}_P, \mathcal{M}_G)$, it holds that the distributions of their path selections are identical. Then combined with random strings, we can conclude

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

which completes our proof.

## B ADDITIONAL EXPERIMENTS

In this section, we present the details of the additional experiments mentioned in Section 6, completing the evaluation of the additional costs introduced by incorporating the delayed duplication technique and further demonstrating Grove's advantages over GraphOS.

### B.1 Duplication Costs

We begin by examining how the cost of delayed duplication scales with the maximum number of neighbors per vertex, $K$. Recall that when $K = 100$, the synthetic graph is sparsified with $D = 10$, keeping it the same as when $K = 10$. The major factor affecting the theoretical per-bucket bound $Y$ on delayed duplications is then the introduction of additional intermediate vertices, which enlarges $|V|$ by at most 10×. We perform this additional preprocessing because $Y$ is far less sensitive to increases in the total number of vertices $|V|$ than to increases in $D$. As a result, the percentages of bandwidth and runtime attributable to delayed duplications at $K = 100$, reported in Table 6, are very close to those at $K = 10$, reported in Table 4. Overall, increasing $K$ while keeping $D$ fixed has only a minimal effect on the runtime and bandwidth attributable to delayed duplication.

Secondly, we note that the probability bound derived in Theorem 3.1 decreases quite slowly as $Y$ increases. This yields a practical trade-off: by setting $Y$ to a fraction of its theoretically-secure bound and accepting a slightly higher overflow probability, we shift the burden of storing some delayed duplications to the client-side stash while reducing server storage and per-operation bandwidth. In Table 7, we evaluate this trade-off on the 4 KB value graph. Because keys have the same size in the 22 KB graph, the size of delayed duplications is identical there, so similar behavior is expected. With $Y$ set to 20% of the theoretical bound, we observe a peak increase in the client stash of 263.8 KB while performing $2^{20}$ `Neighbor` queries, and a reduction of 443.52 KB in the bandwidth of each `Neighbor` query. We stress that the client-side stash size fluctuates and reaches its maximum only after many operations, whereas the decreases in server storage and bandwidth are stable. In addition, we note that when $Y$ is reduced, the processing time for delayed duplication also decreases, though this time already constitutes only a small portion of the overall runtime. Overall, the client $C$ can choose to hold more delayed duplications in its stash to significantly reduce the extra overhead incurred by incorporating the delayed duplication technique.

### B.2 Setup and Storage Comparison

In this subsection, we compare Grove and GraphOS in setup time and server-side storage. Under the client/server paradigm, the client $C$ performs a one-time setup and can do substantial local computation during this phase. A practical method that limits client-local storage and enables fast setup is to instantiate the complete binary tree for ORAM/OMAP locally, then, for each path, fill it with dummy data, encrypt it, and upload it to the server. It follows that both setup time and server-side storage depend on the size of the ORAM/OMAP trees. Consequently, Grove consistently outperforms GraphOS on both metrics, especially on dense graphs. The reason

| Graph Size | | $\lvert V\rvert = 2^{10}$ | | $\lvert V\rvert = 2^{14}$ | | $\lvert V\rvert = 2^{18}$ | | $\lvert V\rvert = 2^{20}$ | |
|---|---|---|---|---|---|---|---|---|---|
| Metrics | | Bandwidth | Storage | Bandwidth | Storage | Bandwidth | Storage | Bandwidth | Storage |
| Lookup | 4 KB | 30.01% | 1.03% | 28.84% | 0.93% | 25.71% | 0.87% | 24.60% | 0.83% |
| | 22 KB | 11.99% | 0.44% | 12.04% | 0.43% | 11.91% | 0.43% | 11.75% | 0.41% |
| Neighbor | 4 KB | 43.66% | 1.52% | 50.81% | 1.73% | 53.81% | 1.86% | 54.60% | 1.94% |
| | 22 KB | 13.26% | 0.45% | 17.33% | 0.60% | 19.44% | 0.69% | 20.15% | 0.73% |

**Table 6: Percentage of bandwidth and runtime from delayed duplications relative to total query processing** $(K = 100, D = 10)$**.**

| Utilization of theoretical bound | Server Storage | Lookup | | Neighbor | |
|---|---|---|---|---|---|
| | | Client stash | Bandwidth | Client stash | Bandwidth |
| 40% | -5.79 GB | +0.02 KB | -30.24 KB | +0.82 KB | -332.64 KB |
| 30% | -6.76 GB | +0.15 KB | -35.28 KB | +13.2 KB | -388.08 KB |
| 20% | -7.72 GB | +16.8 KB | -40.32 KB | +263.8KB | -443.52 KB |

**Table 7: Trade-off introduced by adjusting bound** $Y$ **for delayed duplication** $(\lvert V\rvert = 2^{20}, K = 10$**, and vertex value of 4 KB**$)$**.**

| Graph Size | | $\lvert V\rvert = 2^{10}$ | | $\lvert V\rvert = 2^{14}$ | | $\lvert V\rvert = 2^{18}$ | | $\lvert V\rvert = 2^{20}$ | |
|---|---|---|---|---|---|---|---|---|---|
| Metrics | | Time (h) | Storage (GB) | Time (h) | Storage (GB) | Time (h) | Storage (GB) | Time (h) | Storage (GB) |
| 4 KB | Grove | 0.04 | 0.04 | 0.37 | 0.63 | 3.78 | 29.71 | 12.12 | 106.88 |
| | GraphOS | 0.18 | 0.53 | 2.88 | 8.46 | 47.28 | 135.23 | 191.40 | 541.16 |
| 22 KB | Grove | 0.02 | 0.19 | 0.56 | 3.68 | 6.11 | 58.72 | 19.19 | 232.88 |
| | GraphOS | 0.25 | 2.89 | 4.38 | 46.20 | 76.33 | 739.27 | 317.68 | 2957.08 |

**Table 8: Comparison between the time consumed in the setup phase and the total storage utilized by the server** $(K = 10)$**.**

is that GraphOS stores both vertices and edges as KV pairs inside the OMAP, whereas Grove stores edges inside vertices, so the OMAP size is determined only by $\lvert V\rvert$. As a result, GraphOS must upload more paths during setup, and each path is more expensive. Empirically, we observe a speedup ranging from 80.6% to 94.6%.

For storage, although the delayed duplication ORAM trees $\mathcal{M}_P$ and $\mathcal{M}_G$ add overhead, Grove still achieves a reduction of 74.6% to 92.5%. This advantage grows for graphs with larger vertex values, because the sizes of $\mathcal{M}_P$ and $\mathcal{M}_G$ are independent of those values. Hence these portions become smaller as vertex values increase, which further enlarges the server-side storage advantage. Finally, if $C$ has ample local storage during initialization, it can encrypt the entire complete binary tree for ORAM/OMAP locally and upload it in bulk; in that setting, the setup-time speedup of Grove approximates the server-side storage reduction factor.