TIEREDOMAP: Skewness-Aware Oblivious Map

Juan Li[1], Xinle Cao[2], Huazhen Yu[3], Qianru Wang[4]
Shuyuan Zheng[5], Jian Liu[2],
[1]Hangzhou Normal University [2]Zhejiang University
[3]NUIST [4]Xidian University [5]The University of Osaka
{juanli,xinle,liujian2411}@zju.edu.cn
wangqianru@xidian.edu.cn zheng@ist.osaka-u.ac.jp

***Abstract*—Oblivious map (OMAP) is a fundamental primitive for encrypted databases, yet existing designs largely adhere to a uniform worst-case principle: every record incurs nearly the same time to retrieve, regardless of how often it is queried. Real-world workloads, however, are typically highly skewed, with a small hot set accounting for most requests. We argue that such skewness should be leveraged as a first-class design signal for oblivious retrieval, rather than treated solely as leakage to conceal.**

We present TIEREDOMAP, the first skewness-aware framework for OMAP, opening up a new design space for improving OMAP performance. TIEREDOMAP separates hot and cold records into separate and independent OMAP components to enable more efficient access to hot records without weakening the security guarantees of standard OMAPs. Moreover, its design naturally supports further performance gains under a small, explicit relaxation of security. To make TIEREDOMAP more practical, we also develop a complete mechanism to support dynamic workloads with evolving hot sets. Overall, our results show that oblivious accesses to records need not be governed solely by uniform worst-case behavior, and that skewness-aware structure represents a promising new direction orthogonal to existing OMAP design principles.

I. INTRODUCTION

Outsourcing data to cloud storage offers scalability and cost efficiency, but also exposes a powerful side channel: *access patterns*. As demonstrated in [1]–[4], even when the database contents are encrypted, an adversary who observes how encrypted records are accessed can still infer much sensitive information. Oblivious databases [5]–[8] address this threat by ensuring that the physical access pattern is independent of the logical query sequence. Among their core building blocks, oblivious map (OMAP) [9] is especially important: it can realize general encrypted key-value (KV) access while hiding which pair is being queried and updated. Yet despite steady progress [10]–[12], existing OMAPs still largely follow one guiding principle: *all pairs are treated uniformly*. For example, some classical OMAP constructions [9], [13], [14] always expect $O(\log N)$ interaction rounds and $O(\log^2 N)$ bandwidth to access any single pair in a database of N entries. Recent work [10], [15] improves constants or asymptotic factors of the complexities, but not this uniform organization. A popular pair and a rarely accessed pair are still expected to incur the same logical access structure, the same bandwidth cost, and essentially the same response time. This naturally motivates us to propose the following new question:

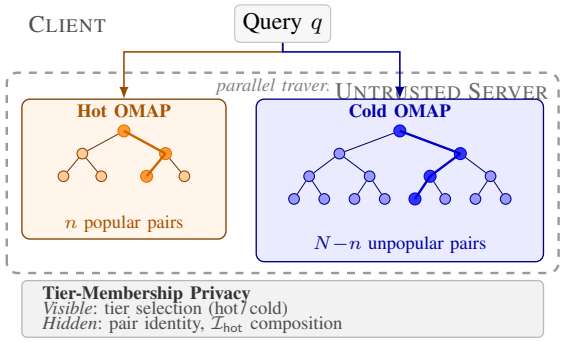


Fig. 1: TIEREDOMAP Architecture.

Is it really necessary to maintain the same uniform oblivious access pattern for popular and unpopular pairs?

Actually, as many real-world KV workloads are often highly skewed [16], [17], where a small fraction of pairs accounts for most requests, plaintext systems have long exploited this property through caching [18], tiering [19], and asymmetric layouts [20]. In contrast, oblivious systems [10], [15] have mostly treated skewness only as something to hide rather than something to utilize. As a result, their design space has remained centered on uniform worst-case behavior.

A. Our New Perspective

In this paper, we argue that the conventional view of uniform oblivious accessing is unnecessarily restrictive. In particular, workload skewness should be treated as a *first-class design signal*, rather than merely as leakage to be mitigated, as it can be leveraged to significantly accelerate oblivious accessing without relaxing its security guarantees. Our key observation is simple: although obliviousness requires the access patterns of popular and unpopular pairs to remain indistinguishable, the client can still be allowed to obtain query results faster for popular pairs. For example, even if the client must interact with the server for a fixed 10 rounds, it may obtain the result within the first 5 rounds for popular pairs, rather than only after all 10 rounds as in the case of unpopular pairs.

To instantiate this new perspective, we propose the TIEREDOMAP framework, illustrated in Figure 1. At a high level, TIEREDOMAP partitions a KV store with N entries into a hot tier $\mathcal{I}_{\mathsf{hot}}$, which contains the n most frequently queried pairs, and a cold tier $\mathcal{I}_{\mathsf{cold}}$, which stores the remaining $(N-n)$ pairs, while hiding the actual composition of the hot set from

TABLE I: Cost comparison for queries on hot pairs under both operating modes. λ: security parameter, N: maximum database capacity, n: maximum hot-set capacity ($n \ll N$). "Answer at" denotes the round at which the client obtains the target pair.

	Hot Bandwidth	Hot Rounds	Answer at	Leakage
Classical OMAPs [9], [13]	$O(\log^2 N + \log N)$	$O(\log N)$	$O(\log N)$	None
TIEREDOMAP (Classical, Full Obliv.)	$O(\log^2 N + \log N)$	$O(\log N)$	$O(\log n)$	None
TIEREDOMAP (Classical, Tier-Memb.)	$O(\log^2 n + \log N)$	$O(\log n)$	$O(\log n)$	Skewness
SOTA OMAP [10]	$O(\frac{\log^2 N}{\log\log N} + \log\lambda \log N)$	$O(\frac{\log N}{\log\log N} + \log\lambda)$	$O(\frac{\log N}{\log\log N} + \log\lambda)$	None
TIEREDOMAP (SOTA, Full Obliv.)	$O(\frac{\log^2 N}{\log\log N} + \log\lambda \log N)$	$O(\frac{\log N}{\log\log N} + \log\lambda)$	$O(\frac{\log n}{\log\log n} + \log\lambda)$	None
TIEREDOMAP (SOTA, Tier-Memb.)	$O(\frac{\log^2 n}{\log\log n} + \log\lambda \log n)$	$O(\frac{\log n}{\log\log n} + \log\lambda)$	$O(\frac{\log n}{\log\log n} + \log\lambda)$	Skewness

the server. It then builds two separate OMAPs over these tiers, referred to as the hot OMAP and the cold OMAP. For each query, the client probes both OMAPs in parallel so that the server cannot tell which tier contains the target pair. Since the hot tier is typically much smaller than the full dataset (i.e., $n \ll N$), queries to the hot OMAP can be resolved more quickly. As a result, when the queried key lies in $\mathcal{I}_{\mathsf{hot}}$, the client can recover the result before the execution on the cold OMAP completes. Although the client must continue interacting with the cold OMAP to preserve obliviousness, *this design still reduces the response time for accesses to popular pairs without any security relaxation!*

a) Design Flexibility: Interestingly, TIEREDOMAP is a promising new direction orthogonal to the design of existing oblivious algorithms. First, for oblivious KV stores, TIEREDOMAP is agnostic to the underlying OMAP construction and can therefore be integrated with any OMAP design to improve performance without weakening security. Moreover, it aligns naturally with a recent trend [21]–[23] in oblivious algorithm design: improving efficiency by permitting small, well-defined additional leakage. Specifically, when processing a batch of queries, the client can first execute all of them on the hot OMAP and then select only queries on cold items to continue the access to the cold OMAP. In this way, the client can improve overall performance, rather than merely response time, at the cost of revealing the workload skewness in the batch. Such a trade-off between security and efficiency has emerged as an important direction [24] in oblivious databases. In this work, TIEREDOMAP supports both a fully oblivious mode, as in traditional OMAPs, and this trade-off mode for batching queries, which we refer to as *tier-membership mode*. To demonstrate these advantages, we integrate TIEREDOMAP with the classical OMAP [9], [13] and a recent state-of-the-art OMAP [10] as backends in both modes, and summarize the resulting benefits in Table I.

Our contributions are summarized as follows:

1) **A skewness-aware perspective on OMAP.** We identify workload skewness as a first-class design signal for oblivious KV stores and show that hiding access patterns does not require all pairs to share the same response time.
2) **Tiered OMAP architecture.** We propose TIEREDOMAP, which organizes hot and cold pairs into separate OMAPs and supports two operating modes: *full obliviousness*, which preserves standard OMAP security while letting hot-pair response time track the hot tier rather than the full database, and *tier-membership privacy* for batching queries, which reveals only the workload skewness while further improving overall system performance.
3) **Dynamic maintenance and formalization.** We develop an online mechanism for evolving hot sets, formalize the security notions for both modes of TIEREDOMAP, and prove them under the standard honest-but-curious security model.
4) **Implementation and evaluation.** We implement TIEREDOMAP for both client/server and enclave-assisted [8], [25] settings and evaluate it under skewed workloads. In the client/server setting, TIEREDOMAP reduces hot-query answer rounds by up to 57% and reduces the measured hot-query response time by up to 58%; in the enclave setting, we demonstrate that the same tiered idea also effectively advances EnigMap [26], a famous enclave OMAP baseline.

The position of this work is to bridge the gap between the cryptographic primitive OMAP and its applications in real-world database workloads. It can be applied across various scenarios and settings to improve the practicality of oblivious KV stores, though we consider only typical settings in this paper for space constraints.

II. BACKGROUND

In this section, we introduce necessary background information and related works to highlight our uniqueness and novelty in utilizing workload skewness to oblivious algorithms.

A. Oblivious Primitives

a) ORAM tree: As a well-known oblivious primitive, oblivious RAM (ORAM) [5] guarantees the access patterns on *memory* independent of memory contents. In this work, we adopt the data structure of the widely used Path ORAM [6] as the underlying data structure. In detail, we organize N encrypted blocks into a complete binary tree of height $O(\log N)$ on the server, and we call this tree the *ORAM tree* for simplicity. Each block is assigned to a uniformly random leaf, and may be stored in an arbitrary node along the root-to-leaf path of that leaf. To access a block, the client reads the entire root-to-leaf path (downloading $O(\log N)$ blocks), retrieves the

target, reassigns it to a fresh random leaf, and writes the path back. Because every access touches a full root-to-leaf path with a random destination, the server's view is independent of the actual access sequence.

b) OMAP: This primitive is presented to extend obliviousness on general KV stores [10], it provides two interfaces:

- $\mathsf{Setup}(DB, N, \lambda) \to (st_C, st_S)$: set up the client and server states st_C, st_S from given database DB, parameter N and security parameter λ. Here N is used to regulate the maximal database allowed [10], [27].
- $\mathsf{Access}(st_C, st_S, k, v, op) \to (st'_C, st'_S, v')$: execute an operation $op \in \{\mathsf{search}, \mathsf{insert}, \mathsf{delete}\}$ on the key-value (KV) pair (k, v), and return the updated client/server states st'_C, st'_S together with the associated value v'. Specifically, search retrieves the pair identified by key k and either reads or updates its value without changing the key, while insert and delete modify both the key and value components.

The security requirement is that, for any two equal-length operation sequences, the server's observed access patterns are computationally indistinguishable (we refer to [10], [27] due to space contraints). The typical OMAP constructions [9], [13], [15] organize KV pairs as a search tree, and stores this tree inside the ORAM tree. Such a search tree is named *oblivious search tree*. To search for a pair, the client traverses from the root to a leaf of the search tree. Taking the oblivious AVL tree [9] as an example, the client traverses $O(\log N)$ nodes for a database of N entries, implying $O(\log N)$ interaction rounds. Each node visit requires one ORAM access of $O(\log N)$ blocks. Hence, the total bandwidth is $O(\log^2 N)$.

B. Related Work

a) Oblivious DB: Emerging security concerns [28]–[30] surrounding encrypted databases have motivated extensive research on oblivious databases across a wide range of settings [8], [31]–[34]. In this work, we focus on obliviousness for KV stores, a popular database paradigm in industry [35]. Existing OMAP-based solutions [12], [26], [27] for KV stores often fail to deliver satisfactory efficiency, especially in the client/server model where the client interacts with the server over WANs. To this end, recent works [21], [22], [24] explore relaxing the adversary model or allowing additional leakages to improve performance. In particular, Pancake [24] and Waffle [21] dispense with ORAM trees entirely and achieve significant speedups through relaxed security models. By contrast, TIEREDOMAP adheres to the strong security model [10], [13] and shows that efficiency can still be improved substantially by exploiting workload skewness without sacrificing security. Another important line of work [11], [26] optimizes OMAP efficiency with the aid of enclaves. Since TIEREDOMAP treats OMAP as a black-box tool, it can also be combined with these enclave-based works to further enhance their efficiency.

Beyond KV stores, numerous works [31], [36]–[39] study obliviousness for more complex databases, especially relational databases [8], [14], [15]. These works design oblivious algorithms for richer functionalities such as joins [38], range queries [15], and aggregation [40]. They often rely on OMAP and treat the oblivious KV store as a fundamental building block. Consequently, improvements at the KV-store layer may benefit these systems as well, although we leave that exploration to future work. Notably, these systems also do not exploit workload skewness.

b) Distributional PIR: To the best of our knowledge, although workload skewness has long been exploited in plaintext systems [41], [42], Lehmkuhl et al. [43] are the first to apply it in the context of privacy-preserving computation. However, their work targets private information retrieval (PIR) [44], a setting distinct from oblivious databases. We argue that oblivious databases are a more natural fit for exploiting workload skewness because PIR is unfriendly to updating the popular set online, which is typically dynamic in production workloads. Moreover, their approach uses workload skewness in a *lossy* manner: the client routes each query to the hot set with high probability, so queries for cold keys fail when they are directed to the hot set. In contrast, TIEREDOMAP avoids this problem and exploits skewness to accelerate hot-key lookups with nearly no degradation in overall performance.

III. PROBLEM OVERVIEW

In this section, we provide an overview of TIEREDOMAP's design, clarifying how workload skewness can be exploited to improve oblivious access in practice. Besides, we also present formal notions to strictly characterize the security of TIEREDOMAP in two modes.

A. System Model

a) Settings and Assumptions: We adopt the standard encrypted-database setting in which a client outsources an encrypted KV store to an untrusted server [10], [45], [46]. The server is *honest-but-curious*: it follows the protocol but tries to infer information from the observed access pattern during query processing. Within this standard threat model, TIEREDOMAP is motivated by three additional assumptions that are both practical and central to our design.

1) **Workload Skewness.** The workload is highly skewed: a small subset of pairs receives a substantial fraction of all requests. This assumption is well supported in production traces, which often follow Zipf [16], Pareto [47], or power-law-like [48] distributions. For example, Facebook's Memcached traces [17] show that the top 1% of keys account for a striking fraction of requests.
2) **Early Response.** Traditional OMAPs effectively couple the client's response time to the completion of the full oblivious access, so all pairs inherit nearly the same retrieval latency. We believe this coupling is stronger than what obliviousness fundamentally requires. Even when a protocol must eventually complete a fixed amount of oblivious work, it is practically valuable for the client to obtain the answer as early as possible. The typical application is to enable the client to obtain its private notes, photos, and documents earlier, even if the oblivious execution is still running in the background.

3) **Optional Trade-off.** Following recent encrypted database systems that offer tunable leakage–performance trade-offs [21], [22], [49], we allow the client to choose whether *workload skewness* is revealed for enhanced efficiency when batching [25], i.e., how many queries in a batch of queries target hot pairs. This also enables fine-grained control over security: the client can flexibly adjust the size of the hot set to control security level.

b) Design Overview: The key idea behind TIEREDOMAP is to separate popular and unpopular pairs into two different tiers rather than treating all pairs uniformly. Given a KV store with at most N entries, TIEREDOMAP maintains a hot tier $\mathcal{I}_{\mathsf{hot}}$ of capacity n and a cold tier $\mathcal{I}_{\mathsf{cold}}$ of capacity $N-n$, each implemented as an independent OMAP, we refer to them as the hot OMAP and cold OMAP. The cost of an oblivious access scales with structure size, and since typically $n \ll N$, the hot OMAP is substantially cheaper to traverse. However, since the client is usually assumed to have limited local storage in oblivious databases [10], [14], [50], it often cannot store hot-set membership locally to determine if a given query targets a hot pair. This raises a natural question: how does the client decide which tier to query?

Our solution is to have the client probe both OMAPs in parallel on every query. Note that since the hot OMAP is smaller, its traversal finishes first. If the queried pair resides in $\mathcal{I}_{\mathsf{hot}}$, the client already holds the answer while the cold-side traversal is still in progress. This parallel-probe design reveals a property that standard OMAPs overlook: *the client's response time and the protocol's total execution time need not coincide*. In a standard OMAP, the client is generally expected to wait for the entire traversal to finish before obtaining any result, so response time equals execution time for every query. TIEREDOMAP decouples the two: for hot pairs, the response time tracks the cost of the smaller hot tier, while the execution continues independently until all oblivious work completes. Building on this decoupling, TIEREDOMAP offers two natural operating modes:

- **Full obliviousness (FO) mode.** The client always completes both traversals. The server therefore observes identical traffic for every query, regardless of which tier holds the target pair, and learns nothing beyond what a standard OMAP reveals. Yet the client obtains the answer as soon as the hot-side traversal finishes, improving the response time for hot pairs without weakening the security guarantee.
- **Tier-membership (TM) mode.** Once the hot OMAP returns the answer, the client aborts the remaining cold-side traversal. This reveals one additional bit per query (whether the pair is currently in the hot or cold tier), but reduces both the response time and the total execution cost for the dominant hot-query population. This trade-off aligns with the recent trend [21], [22] of improving efficiency through small, controlled leakage.

While FO mode is clean and practical, directly applying TM mode to individual queries can be too sensitive: when the hot set is small, e.g., 1% of the dataset, revealing whether one query hits the hot tier may expose useful information. Fortunately, batching [21], [25] can make this leakage much weaker. The client processes the whole batch on the hot tier, but continues only the cold misses together on the cold tier, while hiding which original queries are cold. Thus the server learns only the number of hot and cold queries in the batch. This batch-level skewness is much less sensitive and can often be estimated from public workload statistics [16], [17]. Therefore, throughout this paper, we always consider TM mode only in this batched form to ensure its practicality.

c) Remark: The importance of our design goes beyond a single protocol. More fundamentally, it shows that workload skewness induces a new and legitimate design space for OMAP: the pairs can be organized asymmetrically, as long as the asymmetry is disciplined by a formal leakage model rather than left as an ad-hoc heuristic. This architecture is also not tied to any particular deployment model. While the description above uses the client/server setting for concreteness, the two-tier organization and the decoupling of response time from execution time carry over naturally to proxy-based [21], [25] and trusted-enclave deployments [15], [51]: for hot queries, the proxy/enclave can respond to the client once the traversal on hot OMAP completes, and then execute the remaining protocols by itself for obliviousness.

B. Security Definitions

We define two security levels to formalize the privacy interface created by our asymmetric organization. Recall from §II-A that an OMAP provides a unified interface $\mathsf{Access}(st_C, st_S, k, v, op)$ with $op \in \{\mathsf{search}, \mathsf{insert}, \mathsf{delete}\}$. We write $\sigma = (q_1, \ldots, q_m)$ for a sequence of m Access calls and $\mathsf{AP}(\sigma)$ for the resulting access pattern observed by the passive adversary (which captures the untrusted server). Note that all keys and values are typically assumed to have fixed lengths and be padded if necessary [21], so that individual record sizes do not leak any sensitive information.

The public parameters of TIEREDOMAP are the maximum database capacity N and the maximum hot-set capacity n. Since the OMAP supports insert and delete, the actual number of entries may vary over time; N and n denote the upper bounds with which the underlying data structures are set up. The value of n is chosen by the client according to its desired acceleration ratio to hot-pair queries and is considered public. Given a database DB with at most N entries and a hot set $\mathcal{I}_{\mathsf{hot}}$ whose size is never larger than n, the client first runs OMAP setup to the two tiers and then issues σ. We write $\mathsf{AP}(DB, \mathcal{I}_{\mathsf{hot}}, \sigma)$ for the complete access pattern observed by the adversary, covering both Setup and the subsequent Access calls. During the access phase, the hot set may evolve due to the dynamic maintenance mechanism, but its size always remains at most n.

a) Full Obliviousness: The FO mode extends the standard OMAP security guarantee [10], [27] to the tiered setting, with only the additional parameter n revealed to the server.

Definition 1 (Full Obliviousness)**.** TIEREDOMAP satisfies *full obliviousness* if for any two databases DB_0, DB_1 of size at

most N with hot sets $\mathcal{I}_{\mathsf{hot}}^{(0)}, \mathcal{I}_{\mathsf{hot}}^{(1)}$ of size at most n, and any two operation sequences σ_0, σ_1 of equal length, the access patterns are computationally indistinguishable:

$$\mathsf{AP}(DB_0, \mathcal{I}_{\mathsf{hot}}^{(0)}, \sigma_0) \overset{c}{\approx} \mathsf{AP}(DB_1, \mathcal{I}_{\mathsf{hot}}^{(1)}, \sigma_1).$$

The server's view depends only on the public parameters N, n and the operation number m. It reveals neither which pairs are accessed, nor the evolving composition of $\mathcal{I}_{\mathsf{hot}}$, nor which type of operation is performed. TIEREDOMAP achieves this by completing a full traversal of all underlying structures (for $\mathcal{I}_{\mathsf{hot}}$ and $\mathcal{I}_{\mathsf{cold}}$) on every Access call, so the tiered layout introduces no additional leakage, it only affects *when* the client internally obtains the result.

b) Tier-Membership: In TM mode, we consider the security notion for batch query processing and protect the tier label of individual queries. The workload skewness in the batch is the *only* additional information revealed beyond N, n, and m. Formally, given the batched operations $\sigma = (q_1, \ldots, q_m)$ and the initial hot set $\mathcal{I}_{\mathsf{hot}}$, we write $\mathsf{Prop}(\sigma)$ for the workload skewness of σ, i.e., the proportion of hot/cold queries in σ.

Definition 2 (TM Obliviousness)**.** TIEREDOMAP satisfies *tier-membership* obliviousness if for any two databases DB_0, DB_1 of size at most N with hot sets $\mathcal{I}_{\mathsf{hot}}^{(0)}, \mathcal{I}_{\mathsf{hot}}^{(1)}$ of size at most n, and any two batched operation sequences σ_0, σ_1 of equal length that produce the same workload skewness $\mathsf{Prop}(\sigma_0) = \mathsf{Prop}(\sigma_1)$, the access patterns are computationally indistinguishable:

$$\mathsf{AP}(DB_0, \mathcal{I}_{\mathsf{hot}}^{(0)}, \sigma_0) \overset{c}{\approx} \mathsf{AP}(DB_1, \mathcal{I}_{\mathsf{hot}}^{(1)}, \sigma_1).$$

Note that the two instances may involve entirely different databases, keys, and hot sets, as long as N, n, and actual workload skewness match. Besides, the above notion does not consider multiple batches for simplicity, it is trivial to extend the notion to multiple batches, i.e., batches in the same position always have the same length and workload skewness.

IV. TIEREDOMAP

We now present the complete design of TIEREDOMAP in two parts:

1) The core algorithms that realize skewness-aware oblivious access under a static hot set (§IV-A).
2) A lightweight online mechanism that tracks a shifting hot set under dynamic workload distributions (§IV-B).

Together, these components demonstrate that once skewness is treated as a first-class design signal, the resulting framework is practical under realistic workload dynamics and diverse deployment scenarios. For clarity, we mainly describe the protocol in the client/server setting. The same two-tier organization can also be adapted to proxy- and enclave-based deployments with some modifications. For example, in the client/server setting, FO probes both OMAPs in parallel to reduce interaction rounds. Inside an enclave, where network RTT is not the main bottleneck, FO can instead probe the hot tier first and then continue with the cold tier.

A. Algorithms

TIEREDOMAP treats OMAP as a black box: it requires only a standard oblivious Setup routine and a unified Access interface, and can therefore be instantiated with any existing OMAP construction [9], [10], [15], [27]. For brevity, we omit internal OMAP states from the notation: all client and server states are maintained implicitly within each OMAP instance. Then we formally describe the core procedures in Algorithm 1.

a) Setup: The client takes the database DB, an initial hot set $\mathcal{I}_{\mathsf{hot}}$, the maximum database and hot-set capacities (N, n), and a security parameter λ as inputs to initialize two tiered OMAPs. The hot set $\mathcal{I}_{\mathsf{hot}}$ contains the estimated most popular pairs, and $\mathcal{I}_{\mathsf{cold}}$ is defined as its complement. Like distributional PIR [43], which assumes the popularity distribution is known a priori, TIEREDOMAP can leverage prior workload knowledge (e.g., an estimated Zipf parameter or historical access logs) to select $\mathcal{I}_{\mathsf{hot}}$. When such knowledge is unavailable, the client may start with an arbitrary assignment; the dynamic maintenance mechanism described in § IV-B will converge the hot set to a near-optimal composition online. With the two identified disjoint sets, the client creates two OMAP instances, OMAP_h and OMAP_c, with capacities n and $N-n$ respectively. Since each OMAP is set up obliviously for a fixed capacity, the resulting setup trace depends only on the public parameters N and n, not on the specific identities of the popular pairs.

As a standard optimization in oblivious databases [9], each OMAP instance can internally use two separate ORAM trees: one for a lightweight index and one for the actual data payloads. This index/data separation reduces per-round bandwidth when the full record size is much larger than index size.

b) Access: For each request (k, v, op), the client assigns and executes the operation on both OMAPs simultaneously, where $mode \in \{\mathsf{FO}, \mathsf{TM}\}$ selects the full-obliviousness or tier-membership mode. The protocol is symmetric for search and delete: since the client does not know which tier holds the target pair, it has to issue the same operation to both tiers. The OMAP that contains the pair finds and operates on it; the other completes the same traversal but simply does not find the target pair. The only asymmetric case is insert, where the pair is known not to exist in the database. Here the client invokes a local policy $\mathsf{SelectTier}(k)$ to choose the destination tier and replaces the non-target side with a dummy search. This substitution does not change the obliviousness in the server's view, because the underlying OMAP already hides the operation type.

Importantly, the two OMAP calls proceed *round by round in parallel*: in each interaction round, the client retrieves the ORAM paths from both OMAP instances in a single network message and writes them back together via piggybacking [10]. Since OMAP_h has capacity $n \ll N$, it should complete each operation much earlier. This not only enables the client to obtain hot-pair results faster, but also gives rise to the two operating modes:

Algorithm 1 Core Procedures

Setup$(DB,\ \mathcal{I}_{\mathsf{hot}},\ N,\ n,\ \lambda)$ ▷ one-time setup

$\mathcal{I}_{\mathsf{cold}} \leftarrow \{(k,v) \in DB : k \notin \mathcal{I}_{\mathsf{hot}}\}$
$\mathsf{OMAP}_h \leftarrow \mathsf{OMAP.Setup}(\mathcal{I}_{\mathsf{hot}},\ n,\ \lambda)$ ▷ hot OMAP
$\mathsf{OMAP}_c \leftarrow \mathsf{OMAP.Setup}(\mathcal{I}_{\mathsf{cold}},\ N{-}n,\ \lambda)$ ▷ cold OMAP

Access$(k,\ v,\ op,\ mode)$ ▷ called on every query

$op_h \leftarrow op;\ op_c \leftarrow op$
if $op =$ insert **then**
  $\tau \leftarrow \mathsf{SelectTier}(k)$ ▷ route to correct tier
  if $\tau =$ hot **then**
    $op_c \leftarrow$ search
  else
    $op_h \leftarrow$ search
  end if
end if
▷ *run both OMAPs round by round in parallel*
$\mathsf{OMAP}_h.\mathsf{Access}(k, v, op_h) \parallel \mathsf{OMAP}_c.\mathsf{Access}(k, v, op_c)$
when OMAP_h succeeds **do**
if $mode = \mathsf{TM}$ **then**
  abort OMAP_c ▷ early termination
else
  continue OMAP_c with dummy ▷ full obliviousness
end if

- **FO Mode:** The client can always complete both OMAP traversals, keeping the execution trace fixed so that access patterns remain fully oblivious to the server.
- **TM Mode:** Alternatively, when OMAP_h *successfully* completes the operation, the client can directly abort OMAP_c to save cost, although this reveals which tier holds the target pair. This can be naively extended to query batching and achieves leaking only workload skewness if the OMAP protocol allows multiple parallel Access.

Here the "successful completion" of OMAP_h means that the target pair is either found or inserted in OMAP_h. For non-insert operations, if OMAP_h does not find the target pair, the operation on OMAP_h is treated as unsuccessful and the client must continue traversing OMAP_c to completion.

c) Splitting the cold OMAP: In TIEREDOMAP, every query operates two OMAP instances in parallel, so a hot query still traverses the cold OMAP before obtaining its result. Since OMAP_c is built over $N{-}n$ pairs, this traversal introduces a bandwidth cost that depends on N, even though we expect the hot-query complexity is related to only n. For instance, with the widely used AVL-tree OMAP [9], [13] as underlying OMAP, a hot query visits only the first $\lceil \log_2 n \rceil$ levels of the AVL tree stored in OMAP_c, but each node resides in an ORAM tree of capacity proportional to N, so the total bandwidth is $O(\log n \cdot \log N)$, still dependent on N.

To eliminate this dependence, we partition the AVL-tree nodes in OMAP_c into two ORAM trees: an upper tree $\mathcal{I}^{\mathsf{up}}_{\mathsf{cold}}$ of capacity n that stores the top n nodes, and a lower tree $\mathcal{I}^{\mathsf{low}}_{\mathsf{cold}}$ of capacity $N{-}n$ that stores the remaining $N{-}2n$ nodes. The client now traverses the hot OMAP and ORAM tree for $\mathcal{I}^{\mathsf{up}}_{\mathsf{cold}}$ simultaneously in $O(\log n)$ rounds at $O(\log^2 n)$ bandwidth; a single dummy access to the ORAM tree for $\mathcal{I}^{\mathsf{low}}_{\mathsf{cold}}$ at cost $O(\log N)$ completes the fixed-shape access template. This splitting trick is inspired by recursive ORAM [6] and can be adapted to most existing OMAP constructions [10], [15], [27].

B. Dynamic Maintenance

In realistic workloads, popular pairs often shift over time, so a skewness-aware design should be able to track the hot set adaptively. We present a lightweight online mechanism that obliviously adjusts $\mathcal{I}_{\mathsf{hot}}$ in TIEREDOMAP, allowing the client to recover a high hit rate on $\mathcal{I}_{\mathsf{hot}}$ even after the workload distribution changes dramatically. Remarkably, this mechanism is especially designed for the FO mode under the client/server setting due to the limitations of settings and existing OMAP protocols, which will be introduced in Section VI.

a) Overview: We model workload dynamics in terms of *epochs*, each consisting of B_{obs} consecutive operations. Each pair additionally records its access frequency in recent epochs so that the client can use this information to estimate per-pair popularity. At the beginning of each epoch, the client starts to obliviously swap pairs between OMAP_c and OMAP_h according to the last-epoch estimates, collecting the most popular pairs in OMAP_h. At a high level, we achieve the swap via a local cache in the client side. The client continually flows pairs from both OMAP instances through this cache during normal operations. Their relative orders are determined by the popularity estimates, and the client routes the most popular entries to OMAP_h and the least popular ones to OMAP_c. The epoch size B_{obs} and cache size control the sensitivity to distribution changes and swap accuracy. Such epoch-based adaptation and cache-based replacement are standard in data management [52]–[54].

b) Initialization: We identify all crucial components for the dynamic maintenance as follows:

- **Epoch metadata** (cnt, ep, fp)**.** Each KV pair carries three auxiliary fields alongside its value in the OMAP: a counter cnt recording the number of accesses within the current epoch, a stamp ep indicating the last epoch in which the pair was accessed, and the frequency fp recording the previous-epoch access count.
- **Local cache** $\mathcal{B}$**.** A fixed-capacity client-side buffer of size B (typically $B \in [4, 16]$) that stages entries extracted from both OMAP instances for frequency comparison.
- **Global state variables.** Besides the epoch length B_{obs}, the client maintains three state variables locally: ❶ a global counter t tracking the total number of operations executed; ❷ a scan pointer ptr that helps the client cycle through pairs in OMAP_h to the cache; and ❸ a swap state $st \in \{\mathsf{idle}, \mathsf{hot\text{-}pend}, \mathsf{cold\text{-}pend}\}$ ensuring that at most one swap is in flight at any time.

We describe the setup of these components with the Init routine in Algorithm 2. To convert a static TIEREDOMAP instance into its dynamic form, the client augments each stored pair from (k, v) to (k, v, cnt, ep, fp) to track per-epoch access statistics, with default metadata $cnt{=}0$, $ep{=}{-}1$, and $fp{=}0$. As all pairs apply the same default values, this augmentation can be materialized lazily during normal accesses, i.e., only pairs

Algorithm 2 Dynamic Hot-Set Maintenance

Init($\mathsf{OMAP}_h, \mathsf{OMAP}_c, B$) ▷ lift static to dynamic

for each stored pair (k, v) in $\mathsf{OMAP}_h \cup \mathsf{OMAP}_c$ **do**
  $cnt \leftarrow 0$; $ep \leftarrow -1$; $fp \leftarrow 0$
  oblivious rewrite $(k, v) \rightarrow (k, v, cnt, ep, fp)$
end for
Client locally sets:
  $t \leftarrow 0$; $st \leftarrow \mathsf{idle}$; $\mathcal{B} \leftarrow \emptyset$; $ptr \leftarrow -1$
for $i = 1$ to B **do**
  $x \leftarrow \mathsf{SearchNext}(\mathsf{OMAP}_h, ptr)$
  obliviously extract x from OMAP_h
  $ptr \leftarrow x.key$; $\mathcal{B} \leftarrow \mathcal{B} \cup \{x\}$
end for

UpdateMeta(k, cnt, ep, fp, t) ▷ epoch metadata update

$e = \lfloor t / B_{obs} \rfloor$
if $ep < e$ **then**
  ▷ *first access in current epoch*
  if $ep = e - 1$ **then**
    $fp \leftarrow cnt$ ▷ last epoch's count
  else
    $fp \leftarrow 0$ ▷ stale, too old
  end if
  $cnt \leftarrow 1$; $ep \leftarrow e$
else
  $cnt \leftarrow cnt + 1$ ▷ same epoch, increment
end if

CacheAdmit(x, dir) ▷ admit entry x from $dir \in \{\mathsf{hot}, \mathsf{cold}\}$

$\mathcal{B} \leftarrow \mathcal{B} \cup \{x\}$ ▷ $|\mathcal{B}| = B+1$ temporarily
if $dir = \mathsf{hot}$ **then**
  $w \leftarrow \arg\max_{b \in \mathcal{B}} b.fp$
  insert w into $\mathcal{I}_{\mathsf{hot}}$; $\mathcal{B} \leftarrow \mathcal{B} \setminus \{w\}$ ▷ best → hot
else
  $y \leftarrow \arg\min_{b \in \mathcal{B}} b.fp$
  insert y into $\mathcal{I}_{\mathsf{cold}}$; $\mathcal{B} \leftarrow \mathcal{B} \setminus \{y\}$ ▷ worst → cold
end if

that are actually touched need an explicit rewrite, amortizing the one-time expensive startup cost.

After metadata initialization, the client creates the state variables and fills the cache $\mathcal{B}$ with pairs from OMAP_h. We adopt a linear scan to populate the cache via the pointer ptr and a successor function SearchNext. WLOG, we assume all keys are non-negative integers and initialize $ptr \leftarrow -1$. $\mathsf{SearchNext}(\mathsf{OMAP}_h, ptr)$ is defined to perform either search or delete on the pair in OMAP_h whose key is the smallest one exceeding ptr; this function can be directly realized with a single OMAP Access call. The client invokes it B times in delete mode, each time extracting the target pair from OMAP_h into $\mathcal{B}$ and advancing ptr to the extracted key.

c) Mechanism: After initialization, the dynamic maintenance runs alongside normal query processing and produces a deterministic oblivious execution trace. Algorithm 2 defines two helper routines: UpdateMeta for epoch-metadata update and CacheAdmit for cache-based migration decisions. According to the functionalities, the complete mechanism can be divided into three parts:

- **Cold promotion** (reactive). When a normal access targets a pair x in OMAP_c while $st = \mathsf{idle}$, the client updates x's metadata via UpdateMeta. If $x.fp$ exceeds $\min_{b \in \mathcal{B}} b.fp$, the pair is popular enough to deserve a place in the cache: the client deletes x from OMAP_c and calls $\mathsf{CacheAdmit}(x, \mathsf{cold})$, which evicts the least popular cache entry y and admits x. The state becomes $st \leftarrow \mathsf{cold\text{-}pend}$, indicating that y should be inserted into OMAP_c.
- **Hot demotion** (proactive). Every B_1 normal accesses, if $st = \mathsf{idle}$, the client scans the next pair z in OMAP_h via SearchNext and reads its metadata (without incrementing cnt, since this is not a real access). If $z.fp$ is lower than $\max_{b \in \mathcal{B}} b.fp$, the pair is less popular than the best cache entry and should be swapped out: the client deletes z from OMAP_h and calls $\mathsf{CacheAdmit}(z, \mathsf{hot})$, which returns the most popular cache entry w to OMAP_h and keeps z in $\mathcal{B}$ for further observation. The state becomes $st \leftarrow \mathsf{hot\text{-}pend}$, indicating that w should be inserted into OMAP_h. If z is already more popular than every cache entry, no swap occurs and ptr simply advances.
- **Pending insertion** (periodic). The client periodically issues one insertion to each of OMAP_h and OMAP_c. If $st = \mathsf{hot\text{-}pend}$, the insertion to OMAP_h carries the real entry w and the one to OMAP_c is a dummy. Symmetrically, if $st = \mathsf{cold\text{-}pend}$, the insertion to OMAP_c carries y and the other is a dummy. If $st = \mathsf{idle}$, both are dummies. After a real insertion completes, the client sets $st \leftarrow \mathsf{idle}$ for next cold promotion or hot demotion.

To guarantee obliviousness, the above three parts must execute at fixed, public rates so that the server cannot infer how the swap actually works. We introduce three interval parameters (B_1, B_2, B_3): ❶ every B_1 accesses the client performs a scan on OMAP_h; ❷ every B_2 accesses it performs an insertion on OMAP_h; and ❸ every B_3 accesses it performs an insertion on OMAP_c. These regular operations do not always carry real work; when no swap is pending, the client still executes a dummy of identical shape at the prescribed rate. For instance, a hot-demotion scan is launched every B_1 accesses even when $st \neq \mathsf{idle}$. In that case, the scan is simply a dummy.

d) Piggybacked execution: Another advantage of TIEREDOMAP is that the client can use piggyback [55] to run the dynamic mechanism without any additional interaction rounds under the client/server model. Cold promotion is naturally completed during the normal access that discovers the popular cold pair. Hot demotion and pending insertion are logically independent of the normal access, so the client can execute them alongside the normal access within the same interaction round. Such piggyback is common practice in oblivious databases [10], [56]. One potential concern is that, while demotion and insertion do not conflict with normal accesses in logic, they may conflict in certain OMAP constructions. For example, in AVL-tree-based OMAPs [9], [13], the tree cannot handle two insertions or deletions at the same time. When this happens, the client pauses the demotion or insertion, replaces it with a dummy of identical shape, and defers it to the next scheduled regular access (controlled by interval parameters) whose operation is a search.

Although this strategy requires the client to occasionally wait for a search operation, such a delay is negligible in practice: search covers both reads and updates, so it accounts for the vast majority of traffic in typical KV-store workloads, while insert and delete occupy only a small fraction, hence deferred steps can be caught up quickly. Most importantly, we emphasize that even if the hot set is adjusted more slowly in insertion/deletion-intensive scenarios, TIEREDOMAP still remains highly beneficial. Since no additional interaction rounds are introduced, the worst-case latency is determined solely by the cold OMAP, which is no larger than a standard OMAP. In other words, TIEREDOMAP *accelerates all hot queries while never degrading cold queries!*

V. SECURITY PROOF

We prove that TIEREDOMAP satisfies the two security definitions from §III-B. The first proof treats OMAP_h and OMAP_c as black-box OMAP instances whose security is guaranteed by the standard OMAP security [10]. The second proof is for the batched TM mode. In this mode, the client first processes the whole batch on the hot side, then routes only the cold queries to the cold side without exposing which input positions are cold. Consistent with [10], [14], [33], we focus only on access patterns and exclude timing side-channel attacks [57]–[59], whose mitigation is orthogonal to this work.

Theorem 1. TIEREDOMAP *in FO mode satisfies Definition 1, assuming the underlying OMAP protocols are secure.*

Proof sketch. In FO mode, every query completes one full access to OMAP_h and one full access to OMAP_c, regardless of which tier stores the target pair. Dynamic maintenance also runs at fixed public intervals; a scheduled maintenance slot has the same observable shape whether it carries a real swap or a dummy. Hence any two equal-length operation sequences induce equal-length operation sequences on both underlying OMAP instances. Since OMAP_h and OMAP_c are independent secure OMAPs, a standard hybrid argument replaces their access patterns one by one, which gives $\mathsf{AP}^{(0)} \overset{c}{\approx} \mathsf{AP}^{(1)}$. The full proof appears in Appendix A of the full version [60]. □

For the TM mode, while we consider only batching for this mode, we assume these queries are executed sequentially: given a batch of queries σ, the client sequentially processes them on the hot OMAP, then continues to sequentially execute the remaining cold queries on the cold OMAP. Our implementation in Section VI actually complies with such a paradigm.

Theorem 2. TIEREDOMAP *in batched TM mode satisfies Definition 2, assuming the underlying OMAPs are secure.*

Proof. Let $(DB_0, \mathcal{I}^{(0)}_{\mathsf{hot}}, \sigma_0)$ and $(DB_1, \mathcal{I}^{(1)}_{\mathsf{hot}}, \sigma_1)$ be any two inputs satisfying the conditions of Definition 2, with $|\sigma_0| = |\sigma_1| = m$ and $\mathsf{Prop}(\sigma_0) = \mathsf{Prop}(\sigma_1)$. Let c be the number of cold queries in σ_0 or σ_1. We write the access pattern as $\mathsf{AP} = (\mathsf{AP}_h, \mathsf{AP}_c)$ and prove $\mathsf{AP}^{(0)} \overset{c}{\approx} \mathsf{AP}^{(1)}$ below.

Step 1: Equal-length operation sequences.
We argue that both inputs produce the same number of OMAP operations on each instance.

- *Setup.* Each OMAP is initialized with a fixed capacity (n or $N-n$), so the setup access pattern depends only on (N, n).
- *Hot-side phase.* The client processes all m queries sequentially on OMAP_h. Thus both inputs produce exactly m complete hot-side OMAP operations. The results of these operations are kept inside the client, so the server does not learn which positions in the batch are hot or cold.
- *Cold-side phase.* The client locally selects the c cold queries and processes only these queries on OMAP_c. The selection is not visible to the server. The server observes only that c complete cold-side OMAP operations are executed.

Therefore, both inputs generate equal-length operation sequences $\vec{y}^{(0)}_h, \vec{y}^{(1)}_h$ on OMAP_h and $\vec{y}^{(0)}_c, \vec{y}^{(1)}_c$ on OMAP_c, with $|\vec{y}^{(0)}_h| = |\vec{y}^{(1)}_h|$ and $|\vec{y}^{(0)}_c| = |\vec{y}^{(1)}_c|$. Their lengths are fixed by the public parameters and the leaked workload skewness.

Step 2: Hybrid argument.
Since OMAP_h and OMAP_c are independent instances, we decompose the indistinguishability as follows.

1) OMAP_h security: $\mathsf{AP}_h(\vec{y}^{(0)}_h) \overset{c}{\approx} \mathsf{AP}_h(\vec{y}^{(1)}_h)$. By Step 1, $\vec{y}^{(0)}_h$ and $\vec{y}^{(1)}_h$ are equal-length operation sequences on the same OMAP instance, so this directly follows from OMAP security.
2) OMAP_c security: $\mathsf{AP}_c(\vec{y}^{(0)}_c) \overset{c}{\approx} \mathsf{AP}_c(\vec{y}^{(1)}_c)$. Since both inputs have the same workload skewness, they produce the same number c of cold-side operations. Thus the two cold-side operation sequences have equal length, so this also follows from OMAP security.
3) Independence: since the two OMAPs do not share any state, their access patterns are independent. Combining (1) and (2), we obtain $(\mathsf{AP}^{(0)}_h, \mathsf{AP}^{(0)}_c) \overset{c}{\approx} (\mathsf{AP}^{(1)}_h, \mathsf{AP}^{(1)}_c)$, i.e., $\mathsf{AP}^{(0)} \overset{c}{\approx} \mathsf{AP}^{(1)}$.

The simulator needs only the public parameters and the leaked workload skewness to reproduce the same access-template length. It does not need the identities of hot or cold pairs, nor the positions of cold queries in the batch. The total distinguishing advantage is bounded by the sum of two OMAP-security terms, which is negligible. □

Multiple batches. Both Definition 2 and Theorem 2 are stated for one batch, but can be extended to multiple batches. If the client executes a sequence of batches, the leakage becomes the vector of per-batch workload skewness values. For any two executions with the same number of batches, the same size for each corresponding batch, and the same workload skewness in each corresponding batch, the above proof applies to every batch independently. The full access-pattern sequence then follows by a standard hybrid over batches.

VI. EVALUATION

In this section, we evaluate the TIEREDOMAP framework across both the client/server and enclave settings to demonstrate its impact in practice. Our experimental results answer the following three general research questions

RQ1. How does TIEREDOMAP speed up OMAPs with different skewness settings under the client/server paradigm?

RQ2. How effectively does the cache-based mechanism maintain the hot set under dynamic workloads?

RQ3. How does TIEREDOMAP perform when deployed inside the enclave under both FO and TM modes?

In summary, ❶ TIEREDOMAP reduces hot-query response time and improves average query latency without security relaxation, e.g., 58% on hot-query response time and 26% on average response time; ❷ the maintenance mechanism tracks a shifting hot set within $\sim 1.4\%$ of the static optimum and recovers monotonically and quickly after abrupt drift; ❸ in the enclave setting, TIEREDOMAP reduces server-side latency by up to 62.8% against the optimized EnigMap baseline.

A. Setup and Baselines

We implement TIEREDOMAP in C++ with OpenSSL as the cryptographic library. The prototype includes both the client/server model over TCP sockets and the SGX enclave model. For client/server experiments, the client runs on a local MacBook Air with an Apple M3 processor, 8 CPU cores, and 16 GB memory. The storage server runs on an Alibaba Cloud instance with an Intel Xeon Platinum 8369B processor, 64 vCPUs, 128 GB normal memory, and Ubuntu 22.04.5. The two machines communicate over TCP with 39 ms RTT and 100 MB/s bandwidth. This setting does not use a TEE or trusted memory. For TEE experiments, we use the same Alibaba Cloud SGX instance with an additional 128 GB trusted-memory region. To keep consistent with prior works [10], [26], we report end-to-end latency in the client/server setting and server-side time in the enclave setting. All implementations use Path ORAM [6] as the underlying ORAM tree and follow the typical forms: each ORAM tree node holds four blocks, the index and data are stored in two different ORAM trees to optimize bandwidth. For workloads, we generate queries according to the Zipf distribution, as used in YCSB [48] and Waffle [21]. We vary the skewness parameter s and the database size. The main client/server experiments use $N \in \{2^{16}, 2^{20}, 2^{24}\}$, and the TEE experiments scale up to 2^{24}. Throughout the evaluation, the *hit rate* is the theoretical value $\sum_{k \in \mathcal{I}_{\mathsf{hot}} \cup \mathcal{B}} p_k$, where p_k is key k's Zipf probability and $\mathcal{B}$ is the local maintenance cache.

Baselines. We compare against three representative OMAP protocols under the client/server model:

- AVL-OMAP [9] is the classical oblivious AVL-tree OMAP and is widely used in oblivious database systems [8], [26].
- B^+-OMAP [15] replaces the AVL tree with a B^+ tree, reducing the tree height and interaction rounds.
- B^+-ORAM [10] is the SOTA client/server OMAP and uses ORAM-based design rather than oblivious search tree.

For each baseline, we empirically compare the original version with the corresponding TIEREDOMAP-enhanced version. For TEE experiments, our comparison object is Enigmap [26], the extensively optimized version of AVL-OMAP for enclave.

Limitations. Our experiments cover both the client/server and enclave settings, but we evaluate only FO mode in the client/server setting. The reason is that batched TM is difficult to realize efficiently with existing client/server OMAP backends [9], [27], [50]. Most of them are built on sequential oblivious search trees, where concurrent operations are hard to support; for example, an AVL-based OMAP cannot naturally execute two insertions at the same time. Thus, although batched TM is a valid protocol mode, we evaluate it only in the enclave setting, where the batch can be processed sequentially inside the trusted environment and network RTT is not the main bottleneck. In this setting, the enclave can also directly refresh the hot set periodically, so we do not combine the enclave experiments with the dynamic maintenance mechanism.

B. Client/Server Performance

In this part, we evaluate the client/server deployment under the FO mode on static workload distributions where the hot set $\mathcal{I}_{\mathsf{hot}}$ captures popular items well. For each database size and baseline, we enhance the approach with TIEREDOMAP and then measure per-query bandwidth, answer rounds, response time, and total runtime. All reported client/server results execute the complete hot-side and cold-side templates; hence the server observes the same protocol shape regardless of whether the queried key is hot or cold. Table II and Figure 2 report the results on Zipf (s=1.0) queries per configuration. The key and value sizes are set separately to 4 B and 4 KB.

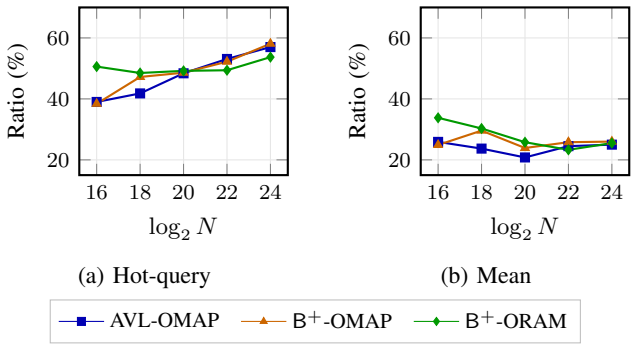
a) Main results: Table II summarizes the main experimental results. For $N \in \{2^{16}, 2^{20}, 2^{24}\}$, we fix $n = 2^{10}$. Even when $N = 2^{24}$, the fixed hot tier captures 43.6% of the Zipf (s=1.0) query mass, although $n/N = 2^{-14}$. Thus, a small hot tier can affect a large fraction of the workload. The key point is that this benefit does not rely on any security relaxation: the complete FO execution still performs the cold-side template, but the client can retrieve a hot value as soon as the hot-side traversal completes.

The results support two observations. ❶ Hot queries obtain their answers substantially earlier. At $N = 2^{24}$, the hot-query answer-round reduction reaches 57% for both AVL-OMAP and B^+-OMAP, and 50% for B^+-ORAM. The corresponding measured WAN response-time reductions are 57%, 58%, and 54%, respectively. This is a pure full-obliviousness gain: the server still observes the full access template, and therefore learns no tier label. ❷ Cold queries remain close to the corresponding original baselines. Their answer rounds match the corresponding original baselines, and the measured response times remain close because the cold traversal dominates the client/server round cost and the hot-side work is in parallel with it. Consequently, TIEREDOMAP improves the response time of the hot fraction without penalizing cold requests in the common WAN setting where interaction rounds dominate the processing time [10], [61].

The bandwidth column in Table II reports the complete FO execution rather than the early answer point. Recall TIEREDOMAP accesses both hot and cold tiers to achieve obliviousness, it has to pay additional bandwidth cost. Nevertheless, as the bottleneck of client/server scenarios is not the bandwidth, this cost does not affect the advantages of TIEREDOMAP in

TABLE II: Full-oblivious client/server performance: baselines vs. TIEREDOMAP (n=1024, s=1.0, WAN). Bandwidth reports the complete FO execution; answer rounds/time report when the client obtains the value.

			Bandwidth (KB)		**Answer Rounds**				**Response Time (ms)**			
N	Baseline	Hit%	Base	FO	Base	Hot	Cold	Mean	Base	Hot	Cold	Mean
2^{16}	AVL-OMAP	64.4	889.3	1556.7	73	46	73	55.6	5573.2	3397.2	5455.9	4130.9
	B$^+$-OMAP		1582.3	2280.5	25	16	25	19.2	1880.2	1158.8	1871.2	1412.7
	B$^+$-ORAM		675.0	1551.2	31	16	31	21.3	2402.3	1185.5	2318.2	1589.2
2^{20}	AVL-OMAP	52.0	1187.2	1971.6	88	46	88	66.2	6688.9	3452.2	7296.1	5297.2
	B$^+$-OMAP		2402.1	3156.5	31	16	31	23.2	2315.4	1191.1	2380.2	1761.8
	B$^+$-ORAM		843.3	1839.9	31	16	31	23.2	2338.4	1188.7	2325.2	1734.2
2^{24}	AVL-OMAP	43.6	1539.8	2441.2	106	46	106	79.8	8029.2	3449.3	8013.5	6022.4
	B$^+$-OMAP		3394.7	4205.4	37	16	37	27.8	2796.9	1171.8	2765.5	2070.3
	B$^+$-ORAM		1027.4	2128.6	32	16	32	25.0	2522.6	1168.7	2427.5	1878.4



(a) Hot-query (b) Mean

Fig. 2: FO answer-latency reduction with varying N.

optimizing latency. We remark that the bandwidth overhead seems more significant in B$^+$-ORAM since this SOTA baseline is much more bandwidth-efficient than the other two methods.

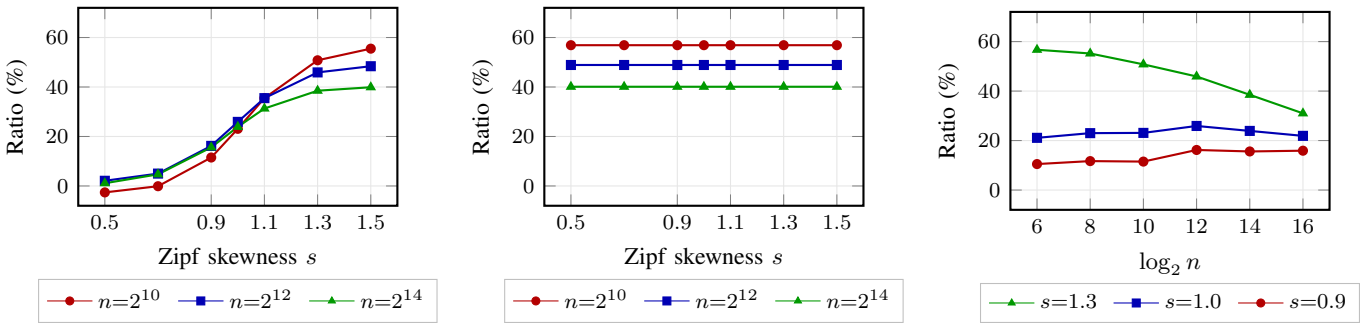
b) Scalability experiments: We next examine how TIEREDOMAP's advantage scales with the database size N. We vary N from 2^{16} to 2^{24} with $n = 1024$ and $s = 1.0$ under full obliviousness, and report the *latency reduction ratio*, i.e., the percentage of response time saved relative to the corresponding baseline. Figure 2 plots this ratio for hot queries and for all queries on average. There are two clear trends emerging from the results:

- The reduction ratio on hot query generally increases with N, rising from roughly 38–49% at $N = 2^{16}$ to 54–58% at $N = 2^{24}$. This is consistent with theoretical analysis: the baseline cost grows with $\log N$, whereas the hot-tier cost is governed by $\log n$ and remains constant as N scales. Consequently, the gap between the two widens as N increases.
- The average reduction ratio is relatively stable around 21–31% across all database sizes. Although individual hot queries benefit more from tiering at larger N, the hit rate simultaneously decreases (from 64% at $N = 2^{16}$ to 43% at $N = 2^{24}$) because a fixed-size hot set captures a smaller fraction of the Zipf mass. These two effects largely cancel out, keeping the average improvement relatively steady.

Together, these observations reveal a fundamental trade-off between the hit rate and the speedup to hot query. The client should choose n according to its target workload and acceleration goals. Notably, even in the most challenging setting ($N = 2^{24}$, $n/N = 2^{-14}$), the hit rate still reaches 43%, hot-query latency is reduced by 54–58%, and the average reduction across all queries reaches 25–26%, demonstrating that TIEREDOMAP provides meaningful acceleration across a wide range of database scales.

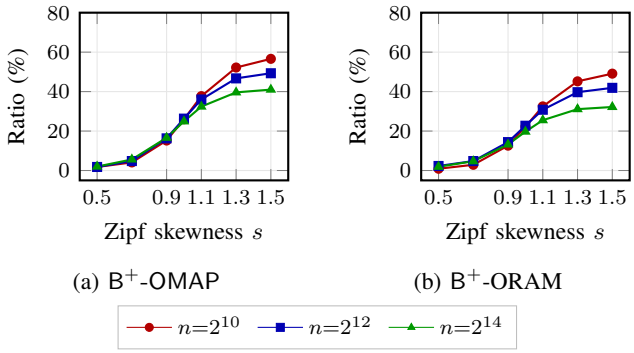
c) Skewness vs. hot-set size: In prior experiments we fix the Zipf parameter at $s = 1.0$. We now sweep s from 0.5 to 1.5 and jointly vary the hot-set size n to understand their combined effect on FO answer latency. We focus on analyzing the results of AVL-OMAP in Figure 3, with $N = 2^{24}$ and $n \in \{2^{10}, 2^{12}, 2^{14}\}$. Subfigure (3a) shows that the average latency reduction grows steadily with s because more queries fall into the hot set as the workload becomes more concentrated. For $n = 2^{10}$, the average reduction rises from 12% at $s = 0.9$ to 23% at $s = 1.0$ and reaches 56% at $s = 1.5$. One can observe that for a given n, the average reduction becomes increasingly close to the hot-query reduction reported in Subfigure (3b) as s increases: at $s = 1.5$ the average (56%) is close to the hot-query ceiling (57%), indicating that almost all queries are served by the hot tier. This confirms that TIEREDOMAP becomes more effective under more skewed workloads. For example, when $s = 1.5$, all tested hot-set sizes achieve about 40–56% average reduction.

For the hot-set size n, Subfigure (3b) shows that smaller n gives a higher hot-query speedup, because a smaller hot OMAP is more efficient to traverse. However, n cannot be set arbitrarily small. Too few hot pairs would lower the hit rate and offset the per-query gain. Subfigure (3c) confirms this trade-off by sweeping n from 2^6 to 2^{16} under three skewness levels. At $s = 1.0$, the average reduction forms an inverted U curve that peaks near $n \approx 2^{12}$ ($\sim$26%), because a further increase in n enlarges the hot OMAP without capturing enough additional popular pairs. At $s = 1.3$, the hit rate is already saturated and the curve decreases monotonically from $\sim$57% to $\sim$31%, as the growing hot-OMAP cost increasingly outweighs the hit-rate gain. At $s = 0.9$, moderate skewness limits the overall benefit and the curve stays modest ($\sim$10–16%) regardless of n. In practice, we recommend that the client select n based on its target hot-query speedup, which is fully determined by n alone and does not depend on the



(a) Avg. reduction vs. skewness (b) Hot-query reduction vs. skewness (c) Avg. reduction vs. hot-set size

Fig. 3: Latency reduction vs. skewness and hot-set size (N=2^{24}, AVL).



(a) B$^+$-OMAP (b) B$^+$-ORAM

Fig. 4: Average Latency reduction vs. skewness for B$^+$-OMAP and B$^+$-ORAM (N=2^{24}).

workload distribution. Tuning n to the observed skewness would itself leak distributional information.

We also repeat the skewness sweep for B$^+$-OMAP and B$^+$-ORAM to validate the above observations and show the average reduction in Figure 4. The results show that both backends exhibit the same qualitative trends: the average reduction grows with s although at different ratios. For B$^+$-OMAP, the reduction at $s = 1.0$ reaches 26% ($n = 2^{10}$) and increases to 57% when $s = 1.5$, since a more skewed workload sends more queries to the lightweight hot tier. For B$^+$-ORAM, the reduction is more moderate (22% at $s = 1.0$ for $n = 2^{10}$ and 49% at $s = 1.5$) because B$^+$-ORAM, as the state-of-the-art, already has a much more efficient baseline than the other two OMAPs, leaving less room for improvement. Nevertheless, the absolute improvements remain substantial, confirming that TIEREDOMAP is effective across diverse OMAP backends.

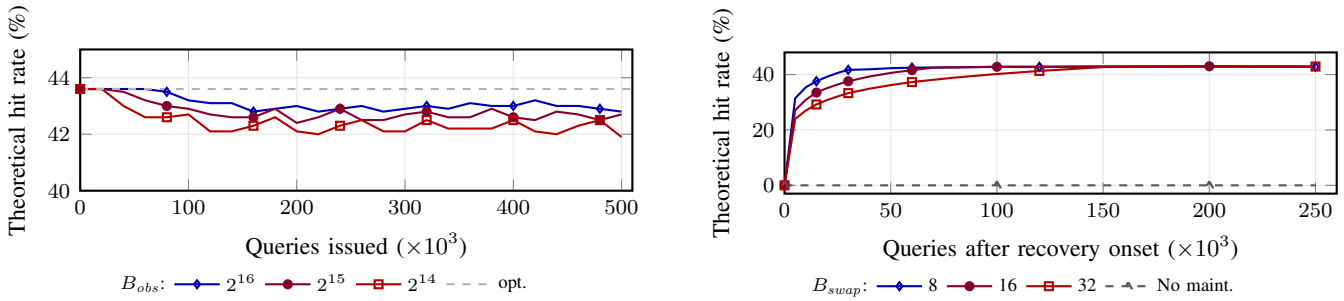
C. Dynamic Hot-Set Maintenance

We now evaluate the dynamic maintenance mechanism. Recall that the mechanism is parameterized by the epoch length B_{obs} for frequency estimation, the cache size B, and three fixed-rate interval parameters (B_1, B_2, B_3) that govern the hot-demotion scan, hot-OMAP insertion, and cold-OMAP insertion rates, respectively. For simplicity we set B_1=B_2=B_3=B_{swap} throughout. We aim to verify three properties: (1) under a stationary distribution the mechanism preserves a near-optimal hit rate, and (2) after an abrupt workload shift it recovers the correct hot set within a predictable number of queries, and (3) maintenance does not noticeably change latency. The hit-rate experiments fix $N = 2^{24}$, $n = 2^{10}$, $B = 8$, and Zipf skewness $s = 1.0$.

For the stationary workload, we first initialize TIEREDOMAP with the correct hot set, fix $B_{swap} = 2^5$, and issue 500 K queries under three observation windows $B_{obs} \in \{2^{14}, 2^{15}, 2^{16}\}$. Figure 5a plots the theoretical hit rate over time. All three curves start at the 43.6% static optimum and remain stable throughout the workload. A larger B_{obs} yields more accurate frequency estimates: with $B_{obs} = 2^{16}$ the hit rate stays within $\sim$0.7% of the optimum, and even the smallest window ($B_{obs} = 2^{14}$) settles at $\sim$42.2%, about 1.4% below. This confirms that the dynamic mechanism introduces negligible loss under a stationary distribution with a properly chosen observation window.

For recovery, we next test an abrupt workload shift. We fix $B_{obs} = 2^{16}$ and vary $B_{swap} \in \{2^3, 2^4, 2^5\}$. Because frequency estimates are refreshed only at epoch boundaries, the mechanism cannot react until the first post-shift epoch completes, so Figure 5b plots the hit rate from that point onward. All three configurations converge to a comparable steady state ($\sim$42–43%), confirming that estimation accuracy is governed by B_{obs} alone. The key difference is convergence speed: $B_{swap} = 2^3$ reaches $\sim$42% within roughly 30 K queries, whereas $B_{swap} = 2^5$ requires approximately 150 K. This is consistent with the one-swap-at-a-time design: a full scan of the $n - B$ stored entries takes $(n - B) \cdot B_{swap}$ queries, and because a demotion only fires when the candidate's frequency falls strictly below the best cache entry, multiple passes are needed to replace the entire hot set. Together with the static experiment, these results demonstrate that TIEREDOMAP's dynamic maintenance effectively tracks workload changes while preserving the hit-rate quality of the static configuration.

For maintenance overhead, we compare dynamic FO with a no-maintenance control using the same epoch-encoded block format. The reported cost is amortized over all queries: real maintenance steps are triggered only at B_{swap}-periodic boundaries, not on every query. With $N \in \{2^{16}, 2^{20}\}$, $B_{obs} = B_{swap} = 32$, and $B = 8$, piggybacked maintenance increases average total bandwidth by 1.42% for AVL-OMAP, 1.70–1.72% for B$^+$-OMAP, and 0.79–0.94% for B$^+$-ORAM. The



(a) Static Zipf distribution

(b) Abrupt workload shift

Fig. 5: Dynamic hot-set maintenance under a static workload and after a workload shift (N=2^{24}, n=1024, s=1.0).

answer and complete round counts are unchanged, and the response-time overhead is at most 0.13%. Thus dynamic maintenance slightly increases bandwidth, but does not noticeably increase latency in the WAN setting.

D. TEE Deployment

We deploy TIEREDOMAP inside the TEE setting on the Alibaba Cloud SGX server described earlier. The comparison baseline is EnigMap [26], a highly optimized enclave OMAP rather than a naive AVL implementation. This setting tests whether tiering still helps after the baseline has already applied enclave-specific optimizations. All results report server-side time and exclude network RTT. Because the hot tier is small, we implement it as a packed directory rather than a tree-structured OMAP. The enclave scans this directory obliviously to locate the target entry.

We use two value settings. The 256 B map-side setting follows EnigMap's map-only workload: OMAP entries and hot-directory entries store key-value pairs directly, so no separate data ORAM is used. This isolates the enclave OMAP cost. The 4 KB full-query setting separates index and data: OMAPs and the hot directory store 8-byte references, and the enclave retrieves the 4 KB value from a disk-backed data ORAM after the index lookup. Therefore, only the 4 KB numbers include data ORAM time. Each data point uses $n = 1024$, Zipf $s = 1.0$, and $Q = 200$ queries. The experimental results are shown in Table III.

In FO mode, TIEREDOMAP still executes both the hot and cold tiers for every query; we report both the total time and the time when a hot answer becomes available. In TM mode, we evaluate only the batched form: the enclave first processes the hot directory for the whole batch and then continues only the cold misses. Thus the leakage is only the aggregate hot/cold composition of the batch. It is shown that tiering remains effective inside the enclave. With 256 B map-side values, FO total time is slightly higher than EnigMap because it still runs both tiers, but the FO answer time is reduced by 37.7–52.1%. BatchTM avoids cold work for hot queries and reduces total server-side time by 41.5–57.1% relative to EnigMap and by 44.9–62.9% relative to FO total. At $N = 2^{24}$, EnigMap takes 5.27 ms, FO completes in 5.59 ms but answers in 3.28 ms, and BatchTM completes in 3.08 ms.

TABLE III: TEE performance against EnigMap. Times are server-side latencies in milliseconds (n=1024, s=1.0, Q=200); BTM denotes BatchTM. FO ans./E and BTM/E are relative to EnigMap, while BTM/FO is relative to FO total.

N	Latency (ms)					Reduction (%)		
	Enig.	FO tot.	FO ans.	BTM	Hit%	FO ans./E	BTM/E	BTM/FO
256 B map								
2^{16}	2.19	2.53	1.05	0.94	64.4	52.1	57.1	62.9
2^{20}	3.56	3.91	1.97	1.80	52.0	44.7	49.3	53.9
2^{24}	5.27	5.59	3.28	3.08	43.6	37.7	41.5	44.9
4 KB value								
2^{16}	2.39	2.53	0.93	0.89	64.4	61.2	62.8	64.8
2^{20}	3.84	4.01	1.94	1.85	52.0	49.3	51.7	53.9
2^{24}	9.45	8.93	5.71	5.72	43.6	39.6	39.5	36.0

For 4 KB full-query values, the data ORAM becomes a larger part of the cost, so the relative gain is smaller at large scale. Even so, FO answer time is reduced by 39.6–61.2%, and BatchTM reduces total server-side time by 39.5–62.8% relative to EnigMap and by 36.0–64.8% relative to FO total. At $N = 2^{24}$, EnigMap takes 9.45 ms, FO completes in 8.93 ms but answers in 5.71 ms, and BatchTM completes in 5.72 ms. These results match the expected behavior: the packed hot directory removes most of the map cost, while large payloads still require data-ORAM work. In addition, the storage overhead in our evaluation is also marginal. In TEE, the hot tier adds about 2.1 MB in the 256 B map setting and 148 KB in the 4 KB reference setting; relative to the baseline OMAP storage, this is at most 2.6% in our smallest $N = 2^{16}$ experiment and below 0.02% at $N = 2^{24}$. The similar cost also appears under the client/server, the hot OMAP capacity is still below 0.2% of the baseline when $n = 1024$ and $N = 2^{20}$, and maintenance only adds small metadata inside existing blocks.

VII. CONCLUSION

In this work, we present TIEREDOMAP, a tiered OMAP framework that treats workload skewness as a first-class design signal. By separating hot and cold pairs into independent OMAPs, TIEREDOMAP reduces client/server hot-query response time by up to 58% with negligible cold-query overhead, and reduces enclave server-side time by up to 62.8% against the baseline. These results show that OMAP performance need not follow a uniform worst-case design; skewness-aware organization is an orthogonal optimization axis.

REFERENCES

[1] M. S. Islam, M. Kuzu, and M. Kantarcioglu, "Access pattern disclosure on searchable encryption: Ramification, attack and mitigation." in *Ndss*, vol. 20, 2012, p. 12.

[2] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart, "Leakage-abuse attacks against searchable encryption," in *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, 2015, pp. 668–679.

[3] S. Lambregts, H. Chen, J. Ning, and K. Liang, "Val: Volume and access pattern leakage-abuse attack with leaked documents," in *European symposium on research in computer security*. Springer, 2022, pp. 653–676.

[4] E. A. Markatou and R. Tamassia, "Full database reconstruction with access and search pattern leakage," in *International Conference on Information Security*. Springer, 2019, pp. 25–43.

[5] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *Journal of the ACM (JACM)*, vol. 43, no. 3, pp. 431–473, 1996.

[6] E. Stefanov, M. v. Dijk, E. Shi, T.-H. H. Chan, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path oram: an extremely simple oblivious ram protocol," *Journal of the ACM (JACM)*, vol. 65, no. 4, pp. 1–26, 2018.

[7] T. Hoang, C. D. Ozkaptan, G. Hackebeil, and A. A. Yavuz, "Efficient oblivious data structures for database services on the cloud," *IEEE Transactions on Cloud Computing*, vol. 9, no. 2, pp. 598–609, 2018.

[8] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa, "Oblix: An efficient oblivious search index," in *2018 IEEE symposium on security and privacy (SP)*. IEEE, 2018, pp. 279–296.

[9] X. S. Wang, K. Nayak, C. Liu, T. H. Chan, E. Shi, E. Stefanov, and Y. Huang, "Oblivious data structures," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 215–226.

[10] X. Cao, W. Feng, J. Liu, J. Zhou, W. Fang, L. Wang, Q. Xu, C. Yang, and K. Ren, "Towards practical oblivious map," *Cryptology ePrint Archive*, 2024.

[11] Y. Guo, H. Chen, H. H. Chen, Y. Luo, X. Wang, and C. Wang, "Bolt: Bandwidth-optimized lightning-fast oblivious map powered by secure hbm accelerators," in *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, 2025, pp. 2594–2608.

[12] R. Wang, S. Lv, X. Li, H. Gong, Z. Liu, T. Li, and L. Guo, "Bomap: A round-efficient construction of oblivious maps," *IEEE Transactions on Dependable and Secure Computing*, 2025.

[13] J. G. Chamani, I. Demertzis, D. Papadopoulos, C. Papamanthou, and R. Jalili, "Graphos: Towards oblivious graph processing," *Proceedings of the VLDB Endowment*, vol. 16, no. 13, pp. 4324–4338, 2023.

[14] Z. Chang, D. Xie, S. Wang, and F. Li, "Towards practical oblivious join," in *Proceedings of the 2022 International Conference on Management of Data*, 2022, pp. 803–817.

[15] S. Eskandarian and M. Zaharia, "Oblidb: Oblivious query processing for secure databases," *Proceedings of the VLDB Endowment*, vol. 13, no. 2.

[16] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and zipf-like distributions: Evidence and implications," in *IEEE INFOCOM'99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No. 99CH36320)*, vol. 1. IEEE, 1999, pp. 126–134.

[17] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, 2012, pp. 53–64.

[18] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab *et al.*, "Scaling memcache at facebook," in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013, pp. 385–398.

[19] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[20] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Wisckey: Separating keys from values in ssd-conscious storage," *ACM Transactions On Storage (TOS)*, vol. 13, no. 1, pp. 1–28, 2017.

[21] S. Maiyya, S. C. Vemula, D. Agrawal, A. El Abbadi, and F. Kerschbaum, "Waffle: An online oblivious datastore for protecting data access patterns," *Proceedings of the ACM on Management of Data*, vol. 1, no. 4, pp. 1–25, 2023.

[22] I. Demertzis, D. Papadopoulos, C. Papamanthou, and S. Shintre, "{SEAL}: Attack mitigation for encrypted databases via adjustable leakage," in *29th USENIX security symposium (USENIX Security 20)*, 2020, pp. 2433–2450.

[23] Y. Du, D. Genkin, and P. Grubbs, "Snapshot-oblivious rams: sub-logarithmic efficiency for short transcripts," in *Annual International Cryptology Conference*. Springer, 2022, pp. 152–181.

[24] P. Grubbs, A. Khandelwal, M.-S. Lacharité, L. Brown, L. Li, R. Agarwal, and T. Ristenpart, "Pancake: Frequency smoothing for encrypted data stores," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2451–2468.

[25] E. Dauterman, V. Fang, I. Demertzis, N. Crooks, and R. A. Popa, "Snoopy: Surpassing the scalability bottleneck of oblivious storage," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 655–671.

[26] A. Tinoco, S. Gao, and E. Shi, "{EnigMap}:{External-Memory} oblivious map for secure enclaves," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 4033–4050.

[27] D. S. Roche, A. Aviv, and S. G. Choi, "A practical oblivious map data structure with secure deletion and history independence," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 178–197.

[28] M.-S. Lacharité, B. Minaud, and K. G. Paterson, "Improved reconstruction attacks on encrypted data using range query leakage," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 297–314.

[29] L. Xu, L. Zheng, C. Xu, X. Yuan, and C. Wang, "Leakage-abuse attacks against forward and backward private searchable symmetric encryption," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 3003–3017.

[30] E. A. Markatou, F. Falzon, Z. Espiritu, and R. Tamassia, "Attacks on encrypted response-hiding range search schemes in multiple dimensions," *Proceedings on Privacy Enhancing Technologies*, 2023.

[31] L. Reichert, G. R. Chandran, P. Schoppmann, T. Schneider, and B. Scheuermann, "Menhir: an oblivious database with protection against access and volume pattern leakage," in *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, 2024, pp. 1675–1690.

[32] D. Sohn, K. Jiang, N. Hammer, and J. Rogers, "Alchemy: A query optimization framework for oblivious sql," *Proceedings of the VLDB Endowment*, vol. 18, no. 9, 2025.

[33] Z. Chang, D. Xie, S. Wang, F. Li, and Y. Shen, "Towards practical oblivious join processing," *IEEE Transactions on Knowledge and Data Engineering*, vol. 36, no. 4, pp. 1829–1842, 2023.

[34] Y. Zheng, W. Wang, S. Wang, Z. Hua, and Y. Gao, "Obliusky: Oblivious user-defined skyline query processing in the cloud," *IEEE Transactions on Services Computing*, vol. 18, no. 1, pp. 314–327, 2024.

[35] M. Kleppmann, *Designing data-intensive applications: the big ideas behind reliable, scalable, and maintainable systems; First edition*. Sebastopol: O'Reilly, 2017. [Online]. Available: https://bib-pubdb1.desy.de/record/642116

[36] A. Mavrogiannakis, X. Wang, I. Demertzis, D. Papadopoulos, and M. Garofalakis, "{OBLIVIATOR}:{OBLIVIous} parallel joins and other {OperATORs} in shared memory environments," in *34th USENIX Security Symposium (USENIX Security 25)*, 2025, pp. 8521–8540.

[37] X. Hu and Z. Wu, "Optimal oblivious algorithms for multi-way joins," in *28th International Conference on Database Theory (ICDT 2025)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2025, pp. 25–1.

[38] S. Krastnikov, F. Kerschbaum, and D. Stebila, "Efficient oblivious database joins," *Proceedings of the VLDB Endowment*, vol. 13, no. 11.

[39] R. Pontes, B. Portela, M. Barbosa, and R. Vilaça, "Codbs: A cascading oblivious search protocol optimized for real-world relational database indexes," in *2021 40th International Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2021, pp. 176–186.

[40] P. Gupta, Y. Li, S. Mehrotra, N. Panwar, S. Sharma, and S. Almanee, "Obscure: Information-theoretic oblivious and verifiable aggregation queries." *Proc. VLDB Endow.*, vol. 12, no. 9, pp. 1030–1043, 2019.

[41] A. Pavlo, C. Curino, and S. Zdonik, "Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012, pp. 61–72.

[42] S. VM, "Dynamic resource management using skewness and load prediction algorithm for cloud computing," *International Journal on Recent and Innovation Trends in Computing and Communication*, vol. 2, no. 4, pp. 798–804, 2014.

[43] R. Lehmkuhl, A. Henzinger, and H. Corrigan-Gibbs, "Distributional private information retrieval," in *34th USENIX Security Symposium (USENIX Security 25)*, 2025, pp. 3377–3396.

[44] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan, "Private information retrieval," *Journal of the ACM (JACM)*, vol. 45, no. 6, pp. 965–981, 1998.

[45] R. A. Popa, C. M. Redfield, N. Zeldovich, and H. Balakrishnan, "Cryptdb: processing queries on an encrypted database," *Communications of the ACM*, vol. 55, no. 9, pp. 103–111, 2012.

[46] X. Cao, J. Liu, H. Lu, and K. Ren, "Cryptanalysis of an encrypted database in sigmod'14," *Proceedings of the VLDB Endowment*, vol. 14, no. 10, pp. 1743–1755, 2021.

[47] P. Barford and M. Crovella, "Generating representative web workloads for network and server performance evaluation," in *Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, 1998, pp. 151–160.

[48] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 143–154.

[49] L. Qiu, G. Kellaris, N. Mamoulis, K. Nissim, and G. Kollios, "Doquet: Differentially oblivious range and join queries with private data structures," *Proceedings of the VLDB Endowment*, vol. 16, no. 13, 2023.

[50] X. Cao, Y. Li, D. Bogatov, J. Liu, and K. Ren, "Secure and practical functional dependency discovery in outsourced databases," in *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 2024, pp. 1645–1658.

[51] C. Priebe, K. Vaswani, and M. Costa, "Enclavedb: A secure database using sgx," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 264–278.

[52] G. Einziger, R. Friedman, and B. Manes, "Tinylfu: A highly efficient cache admission policy," *ACM Transactions on Storage (ToS)*, vol. 13, no. 4, pp. 1–31, 2017.

[53] Z. Yang, M. Hoseinzadeh, A. Andrews, C. Mayers, D. T. Evans, R. T. Bolt, J. Bhimani, N. Mi, and S. Swanson, "Autotiering: Automatic data placement manager in multi-tier all-flash datacenter," in *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)*. IEEE, 2017, pp. 1–8.

[54] N. Megiddo and D. S. Modha, "{ARC}: A {Self-Tuning}, low overhead replacement cache," in *2nd USENIX Conference on File and Storage Technologies (FAST 03)*, 2003.

[55] W. S. Lai, "An analysis of piggybacking in packet networks," *Computer Networks (1976)*, vol. 6, no. 4, pp. 279–290, 1982.

[56] L. Ren, C. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. Van Dijk, and S. Devadas, "Constants count: Practical improvements to oblivious {RAM}," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 415–430.

[57] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang, "Detecting privileged side-channel attacks in shielded execution with déjá vu," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, 2017, pp. 7–18.

[58] C. W. Fletchery, L. Ren, X. Yu, M. Van Dijk, O. Khan, and S. Devadas, "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)*. IEEE, 2014, pp. 213–224.

[59] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa, "Strong and efficient cache side-channel protection using hardware transactional memory," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 217–233.

[60] J. Li, X. Cao, H. Yu, Q. Wang, S. Zheng, and J. Liu, "TieredOMap: Skewness-aware oblivious map," Cryptology ePrint Archive, Paper 2026/823, 2026. [Online]. Available: https://eprint.iacr.org/2026/823

[61] Z. Ye, X. Wang, Z. Liu, D. Zhu, and J. Ma, "Obir-tree: An efficient oblivious index for spatial keyword queries on secure enclaves," *Proceedings of the ACM on Management of Data*, vol. 3, no. 1, pp. 1–24, 2025.

APPENDIX A
FULL PROOF OF FO SECURITY

Proof. We prove the theorem by comparing two valid executions. Let the two executions use databases DB_0, DB_1, hot sets $\mathcal{I}_{\mathsf{hot}}^{(0)}, \mathcal{I}_{\mathsf{hot}}^{(1)}$, and operation sequences σ_0, σ_1 satisfying Definition 1, with $|\sigma_0| = |\sigma_1| = m$. We write the access pattern as $\mathsf{AP} = (\mathsf{AP}_h, \mathsf{AP}_c)$, where AP_h and AP_c denote the access sequences on OMAP_h and OMAP_c respectively. We prove $\mathsf{AP}^{(0)} \overset{c}{\approx} \mathsf{AP}^{(1)}$ below.

Step 1: Equal-length operation sequences.
We argue that both inputs produce the same number of OMAP operations on each instance.

- *Setup.* Each OMAP is initialized with a fixed capacity (n or $N-n$), so the setup access pattern depends only on (N, n).
- *Queries.* In full obliviousness mode, every user query triggers one full traversal of OMAP_h and one full traversal of OMAP_c, regardless of which tier holds the target pair or which operation type is performed.
- *Maintenance.* The dynamic mechanism adds operations to each OMAP at fixed public intervals (B_1, B_2, B_3). Whether a scheduled slot carries a real swap or a dummy is determined by the client-local cache and swap state, which are invisible to the server; in either case the slot contributes exactly one additional OMAP operation. Epoch metadata is updated within the normal OMAP read-write cycle and does not produce any extra operations.
- *Piggybacked operations.* Each piggybacked maintenance operation can be modeled as an independent OMAP operation executed sequentially after the primary one. Piggybacking imposes one constraint: a piggybacked insert or delete is deferred (replaced by a dummy) when the primary operation on the same OMAP is also an insert or delete. This constraint depends only on the public schedule and the operation types in the user's query sequence, so it applies identically to both inputs and does not affect the equal-length argument.

Therefore, both inputs generate equal-length operation sequences $\vec{y}_h^{(0)}, \vec{y}_h^{(1)}$ on OMAP_h and $\vec{y}_c^{(0)}, \vec{y}_c^{(1)}$ on OMAP_c, with $|\vec{y}_h^{(0)}| = |\vec{y}_h^{(1)}|$ and $|\vec{y}_c^{(0)}| = |\vec{y}_c^{(1)}|$.

Step 2: Hybrid argument.
Since OMAP_h and OMAP_c are independent instances, we decompose the indistinguishability as follows.

1) OMAP_h security: $\mathsf{AP}_h(\vec{y}_h^{(0)}) \overset{c}{\approx} \mathsf{AP}_h(\vec{y}_h^{(1)})$. By Step 1, $\vec{y}_h^{(0)}$ and $\vec{y}_h^{(1)}$ are equal-length operation sequences on the same OMAP instance, so this directly follows the OMAP security guarantee.
2) OMAP_c security: $\mathsf{AP}_c(\vec{y}_c^{(0)}) \overset{c}{\approx} \mathsf{AP}_c(\vec{y}_c^{(1)})$. The same argument applies to the independent instance OMAP_c.
3) Independence: since the two OMAPs do not share any state, their access patterns are independent. Combining (1) and (2), we obtain $(\mathsf{AP}_h^{(0)}, \mathsf{AP}_c^{(0)}) \overset{c}{\approx} (\mathsf{AP}_h^{(1)}, \mathsf{AP}_c^{(1)})$, i.e., $\mathsf{AP}^{(0)} \overset{c}{\approx} \mathsf{AP}^{(1)}$.

The total distinguishing advantage is bounded by the sum of two OMAP-security terms, which is negligible. □