OasisDB: An Oblivious and Scalable System for Relational Data

Haseeb Ahmed
University of Waterloo
h25ahmed@uwaterloo.ca

Nachiket Rao
University of Waterloo
ns2rao@uwaterloo.ca

Abdelkarim Kati
University of Waterloo
akati@uwaterloo.ca

Florian Kerschbaum
University of Waterloo
florian.kerschbaum@uwaterloo.ca

Sujaya Maiyya
University of Waterloo
smaiyya@uwaterloo.ca

ABSTRACT

We present OasisDB, an oblivious and scalable RDBMS framework designed to securely manage relational data while protecting against access and volume pattern attacks. Inspired by plaintext RDBMSs, OasisDB leverages existing oblivious key value stores (KV-stores) as storage engines and securely scales them to enhance performance. Its novel multi-tier architecture allows for independent scaling of each tier while supporting multi-user environments without compromising privacy. We demonstrate OasisDB's flexibility by deploying it with two distinct oblivious KV-stores, PathORAM and Waffle, and show its capability to execute a variety of SQL queries, including point and range queries, joins, aggregations, and (limited) updates. Experimental evaluations on the Epinions dataset show that OasisDB scales linearly with the number of machines. When deployed with a plaintext KV-store, OasisDB introduces negligible overhead in its multi-tier architecture compared to a plaintext database, CockroachDB. We also compare OasisDB with ObliDB and Obliviator, two oblivious RDBMSs, highlighting its advantages with scalability and multi-user support.

PVLDB Reference Format:
Haseeb Ahmed, Nachiket Rao, Abdelkarim Kati, Florian Kerschbaum, and Sujaya Maiyya. OasisDB: An Oblivious and Scalable System for Relational Data. PVLDB, 18(11): 4478 - 4491, 2025.
doi:10.14778/3749646.3749707

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/dsg-uwaterloo/ObliSQL.

1 INTRODUCTION

With encrypted databases, applications outsourcing their data can benefit from a cloud's scalability, persistence, and cost effectiveness without compromising data confidentiality and privacy. However, attacks on encrypted databases have proven their ineffectiveness in preserving data privacy by observing the access patterns and access frequencies of an application [10, 19, 30–32, 34, 36, 37, 40, 54, 55, 59, 74]. For example, the access frequency distribution of encrypted keywords searched on encrypted emails can reveal the plaintext keywords within the emails [34].

	Multi-user setting	*Scalable*	*H/W agnostic*	*Supports updates*
Opaque [75]	✗	✗	✗	✗
ObliDB [22]	✗	✗	✗	✓ *(Complex)*
Obliviator [51]	✗	✗	✗	✗
SEAL [19]	✗	✓	✓	✗
OasisDB	✓	✓	✓	✓ *(Limited)*

Table 1: Comparing OasisDB with other oblivious RDBMs.

Oblivious data systems mitigate such attacks by obfuscating the access frequency distribution [11, 17–19, 22, 29, 47, 48, 62, 67, 70, 75]. Commonly deployed techniques for obliviousness include Oblivious RAM or ORAM [26], frequency smoothing [29, 42], and dynamic keyword shuffling [48]. These schemes protect an application's data from honest-but-curious (or semi-honest) adversaries who can view the data accesses on the cloud server along with any message exchanges between the trusted and untrusted domains.

Most existing oblivious data systems store data in key-value format, supporting a single key `Get` or `Put` request [11, 17, 18, 29, 47, 48, 62, 64, 67, 70], with many enabling scalability [18, 64, 66, 70]. However, organizations maintain structured data, often captured in the relational data model consisting of tables, attributes (or columns), and tuples (or rows). Relational data also enables more sophisticated ways to query the underlying data by supporting conditional selections of certain tuples, range searches, aggregations (e.g., SUM, AVG), joins of tables, and ordering of tuples.

A handful of solutions propose and prototype oblivious relational data management systems (RDBMs) - Opaque [75], ObliDB [22], Obliviator [51], and SEAL [19]. However, these systems' support for a richer set of queries destroys the benefits of scalability usually associated with cloud storage. Hence, the state-of-the-art requires one to choose between functionality-rich queries and low scalability, or simple keyword queries and high scalability. In particular, we identify four drawbacks with existing, functionality-rich systems as summarized in Table 1. (i). Unlike plaintext RDBMSs, these systems operate in a single-user setting and process client queries *sequentially*, leading to a substantial performance gap between plaintext and oblivious RDBMSs. (ii) Opaque [75], ObliDB [22], and Obliviator[51] cannot scale[1] to accommodate increasing client workloads. While SEAL [19] enables horizontal scaling, without multi-user support, there is no throughput gains reported in [19].

[1] [75] and [51] supports executing a single request in parallel but not horizontal scaling.

Proceedings of the VLDB Endowment, Vol. 18, No. 11 ISSN 2150-8097.
doi:10.14778/3749646.3749707

Moreover, SEAL (tunably-) leaks the shard that serves a given client request with the leakage increasing as the scale factor grows. (iii) Opaque, ObliDB, and Obliviator require specialized hardware enclaves to execute oblivious relational queries, which may not be supported by all cloud providers. (iv). SEAL, Opaque, and Obliviator do not allow applications to update the data.

The above drawbacks indicate a clear need for an oblivious RDBMS that (i) supports multi-user settings, serving client requests in parallel, (ii) scales throughput as the number of machines serving requests increases, (iii) is hardware agnostic, and (iv) accommodates updates to application data. The contributions of this paper, proposed as an oblivious RDBMS ***OasisDB***, make significant strides towards achieving all four of these goals.

The first fundamental question we address in designing OasisDB is how to shard a relational database – consisting of one or more tables with varying numbers of columns and rows – across multiple machines for scalability. In answering this question, we take inspiration from commercial plaintext RDBMSs such as YugaBytеDB [73], Google's Spanner [16], CockroachDB [68], and TiDB [69] that store relational data on top of an underlying key-value storage engine, which facilitates scalability. This approach gives OasisDB a huge advantage in leveraging an existing oblivious key-value store (KV-store) as its storage engine and avoids the need to design custom storage engines, as seen in [22, 51, 75].

Even with an oblivious key-value storage engine, achieving all four goals is non-trivial and raises either privacy or performance challenges. For example, naively scaling the system by sharding the plaintext tuples across multiple KV-stores can break security because a single shard consisting of popular entries will receive many more requests than the other shards, thus revealing the locality of popular entries to an adversary. Simply shifting entries across shards introduces significant inter-shard coordination, potentially impacting performance. Additionally, revealing the exact result volume of each SQL query makes the system vulnerable to volume pattern attacks [19, 24, 25, 30, 32, 36, 38, 49, 50, 59].

Contributions and overview:

1. OasisDB as a framework: We develop secure mechanisms for handling relational data and queries on top of an oblivious KV-store as the storage engine. This design decision allows OasisDB to integrate with many existing (or newly developed) oblivious KV-stores seamlessly. As a result, OasisDB can be viewed as a framework for securely scaling non-scalable oblivious KV-stores to support relational data. To demonstrate its versatility, we implement OasisDB using two distinct oblivious KV-stores: PathORAM [67], a widely used scheme in oblivious KV-stores, and Waffle [48], a recent, highly efficient KV-store. These datastores differ significantly in their threat models, bandwidth and storage overheads, and performance characteristics. By building OasisDB atop these two diverse KV-stores, we illustrate its adaptability and extensibility. Additionally, we formally define the requirements an oblivious KV-store must meet to be compatible with OasisDB (§4).

2. Scalable multi-tier architecture: To achieve scalability, we propose a novel multi-layer architecture where each layer can be scaled independently. Specifically, following the commonly deployed system model for multi-user environments [17, 29, 48, 62, 64, 70], we employ a trusted proxy to route concurrent requests between multiple users and the storage server. Unlike traditional approaches, the proxy in OasisDB is composed of multiple logical processes, which can be distributed across a small number of physical machines, thus removing scalability bottlenecks.

OasisDB 's architecture comprises of three layers (Figure 1). The top layer, called `Executor`, interacts with the cloud storage and executes the oblivious data access mechanisms defined by PathORAM or Waffle. The middle layer, called `Batcher`, batches multiple requests from the bottom layer such that each `Executor` receives a uniform load. Maintaining this equal workload is crucial to preserving obliviousness, as we discuss in §4. The bottom layer, called `Resolver`, translates a client-issued SQL query into key-values to be retrieved from the storage. OasisDB's design obfuscates the sizes of SQL query results, effectively mitigating volume pattern attacks. We evaluate this by launching such attacks against OasisDB and demonstrate that it successfully defends against them (§6).

3.Oblivious SQL processing: We leverage OasisDB's design to demonstrate how to securely process a variety of SQL queries (§5). In particular, OasisDB supports oblivious conditional selections, range queries, joins, aggregates, order bys, and (highly restricted) update SQL queries.

4. Open-sourced prototype of OasisDB: Finally, we provide an open-sourced implementation of OasisDB integrated with an optimized PathORAM design [64] and Waffle [48]. Experimental evaluations on Epinions dataset [4] provide empirical evidence that OasisDB scales linearly with the number of machines (§7). We also compare OasisDB against CockroachDB [68], a plaintext baseline, and highlight that the performance penalty in OasisDB's design is negligible and primarily stems from the oblivious KV-stores. Comparing OasisDB with two oblivious baselines, ObliDB [22] and Obliviator [51] on Big Data Benchmark [5] indicates the benefit of multi-user support and scalability of OasisDB.

A motivating example: A clinic using an Electronic Health Record (EHR) system may rely on cloud storage for cost-effective handling of large records like MRI scans and genome profiles. But privacy laws may require the clinic to protect patient data from unauthorized entities. The clinic in this example only needs to provision a few machines deployed locally on its premise to run OasisDB, which acts as a gateway between the clients (e.g., doctors, analysts, patients) and the cloud storage. OasisDB enables the clinic to leverage the cloud storage without revealing the data or any other sensitive information to the cloud, while ensuring high performance.

2 BACKGROUND ON OBLIVIOUS KV STORES

This section provides a background on two oblivious key-value stores, PathORAM [67] and Waffle [48], which serve as sample alternative storage engines for developing OasisDB.

PathORAM: Given a sequence of logical accesses, PathORAM [67] generates a sequence of seemingly random physical accesses on the server. PathORAM securely stores N encrypted data blocks on untrusted storage by organizing them as a binary tree. Each tree node, termed a *bucket*, holds exactly Z blocks. PathORAM assigns each block to a random path identified by its leaf-id and stores the block in one of the buckets along the path from the root to leaf. When a client requests an object, PathORAM retrieves the entire path containing the object (or block), thereby obfuscating which specific block is being accessed. After each access to an object, it re-assigns the object to a randomly chosen path in the tree. This ensures that the server observes uniformly random path accesses, irrespective of the skew in requested data objects.

Path ORAM relies on a single trusted proxy positioned between clients and untrusted storage to execute the obliviousness protocol. Upon receiving a read or write request, the proxy performs the following steps: 1) It retrieves the requested object's current path ID p from a local data structure *position map*, a map of each object to its assigned path. 2) It reads the entire path p from the storage server and temporarily stores all the blocks in a local cache-like structure called the *stash*. 3) The proxy then assigns a new, random path p' to the requested object and updates the position map accordingly. 4) For write requests, it updates the object's value in the *stash*. 5) Lastly, it *evicts* or writes-back path p by pushing blocks from the *stash* to the lowest non-full bucket that intersects with both p and p'. If no such bucket has space, the blocks remain in the *stash*.

Within the honest-but-curious adversarial model, ORAM assumes the adversary to be *active* meaning that the adversary can inject queries (e.g., by compromising clients). Providing obliviousness under such a powerful adversary incurs a lower-bound of *O(logN)* bandwidth overhead [27, 43, 57] with each client request accessing *O(logN)* objects from the server.

Waffle: Waffle [48] is a recent oblivious KV-store designed to overcome the *logN* bandwidth overhead of ORAM by assuming a *passive persistent adversary*, which has the same capabilities as an active adversary except that it cannot inject queries. Under such an adversary, Waffle protects an application's data using techniques that allow tuning certain parameters to deliberately trade off privacy for higher performance.

Waffle protects against access pattern attacks by first reading an encrypted key (and its value), deleting it, and re-writing it with fresh encryption such that the adversary cannot infer if the newly written key-value pair corresponds to the same object or a different object. However, just employing the above technique leaks timing-based side channel because a popular object will be read and re-written much more frequently compared to an unpopular object. Waffle mitigates the above leakage by guaranteeing that *every* outsourced key will be accessed within $\alpha + 1$ accesses after being written on the server. It achieves this by batching real queries with fake queries on least recently accessed objects. An application can choose the number of fake queries per batch: higher the number of fake queries, lower the α value and higher the security, but higher is also the bandwidth overhead.

Waffle also employs client-side caching at the proxy to handle highly skewed workloads efficiently by storing popular objects in the cache. This enables Waffle to retrieve unpopular objects from the server at a faster rate, reducing α. Waffle provides another guarantee β, which dictates the minimum number of accesses between reading an object from the server and re-writing it back. In essence, Waffle ensures that for *any* sequence of objects requested by clients, the sequence of server accesses will be α, β-uniform with:

$$\alpha = \left\lceil \max\left(\frac{(N-C)-(B-f_D)}{B-R-f_D}, \frac{D}{f_D}\right)\right\rceil \qquad \beta = \left\lfloor \frac{C}{B-f_D+R} - 1 \right\rfloor$$

where N is the number of objects, C the cache size, B the batch size, R the number of real queries in a batch, and D an optional number of dummy objects (added for additional security) with f_D number of queries to the fake objects per batch. All parameters can be configured by the application, providing a trade-off between privacy and performance.

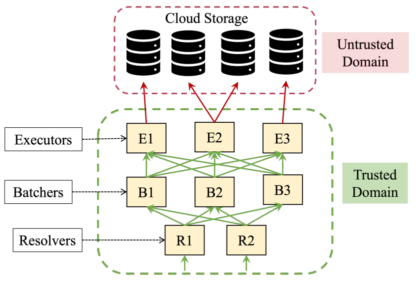


Figure 1: OasisDB Architecture.

3 SYSTEM AND THREAT MODEL

System Model: OasisDB considers an architecture where an application outsources its data to the cloud and relies on a local trusted proxy to route client requests. The proxy-architecture, which is well-adapted in academic [17, 29, 47, 48, 62, 64, 66, 67, 70] and production systems [3, 46, 52, 58, 65], effectively handles multi-user settings and serves client requests in parallel. OasisDB enables applications to securely store and query relational data by leveraging oblivious key-value storage engines (we explain how OasisDB converts tabular data into key-value pairs in §4).

Catered towards scalability, OasisDB partitions the state and functionality of the central trusted proxy across multiple layers as shown in Figure 1. In particular, the proxy consists of three layers – `Resolver`, `Batcher`, and `Executor`– each of which can be scaled horizontally independent of the other layers. Additionally, OasisDB assumes a scalable cloud storage architecture and distributes key-value pairs across multiple server instances. The proxy machines and cloud storage servers reside on different trust domains and hence, communicate over WAN. While the current design assumes that proxy machines do not fail, we aim to extend OasisDB to guarantee fault-tolerance in future work.

Threat Model: OasisDB considers honest-but-curious adversaries who control the cloud and can observe an application's data access patterns to launch attacks using techniques such as *frequency analysis*, *l_p-optimization* or volume pattern attacks [19, 24, 25, 30, 32, 36, 38, 41, 44, 49, 50, 53, 59] to learn *any* non-public information about the data. An adversary can also observe (encrypted) message exchanges between the trusted and the untrusted domains. But it cannot observe or influence message exchanges between the clients and the proxy or across proxy machines within the trusted domain. We consider attacks performed by malicious adversaries or timing attacks based on response times [6, 62] or application query workload to be out of scope. Along with hiding access patterns, OasisDB's batching mechanism obfuscates volume pattern leakages by blurring the boundaries of individual SQL queries (§6).

Since OasisDB serves as a secure framework incorporating multiple oblivious KV-stores, the adversary's capabilities within the honest-but-curious model may vary. In fact, we build OasisDB with PathORAM [67] and Waffle [48] as two alternatives, both of which have slightly different adversarial model as explained in §2.

4 OASISDB DESIGN

UID	Name	Age	Salary
100	Alice	35	10000
101	Bob	42	7500
102	Carol	29	10000

(a) Base relational table `Emp`

Key	Value
Emp | Name | 100	Alice
Emp | Age | 100	35
Emp | Salary | 100	10000
Emp | Salary_idx | 10000	100 | 102
Emp | Salary_idx | 7500	101

(b) Converted key-value pairs

Figure 2: Key-value pairs of `UID=100` and index on `Salary`

4.1 Data Representation

1. How to shard tabular data?

We begin with the fundamental design choice of why OasisDB represents data in key-value format. A primary goal of OasisDB is to scale, which is typically achieved by sharding data across multiple processes executed on different machines. Sharding tables similar to MySQL [63] by partitioning rows of tables exposes the physical locations of each table in a shard to the server. Hiding access patterns would require customized obliviousness techniques applied per table, as seen in [2, 22, 51, 75].

Solution: Inspired by many plaintext databases supporting relational queries [16, 68, 69, 73], OasisDB converts tabular data into key-value pairs for ease of scalability. Specifically, a single row translates to multiple key-value pairs with each pair as ('$T_{name}|C_{name}|pk$' : C^{pk}_{value}) where T_{name} is the table name, C_{name} the column name, pk the primary key and C^{pk}_{value} that column's value for pk, as shown in Figure 2b. This enables our system to easily partition key-values across multiple machines while benefiting from re-using existing oblivious KV-stores.

2. Where and how to store indices?

A naive approach is to process SQL queries by scanning the entire table and filtering rows satisfying a predicate. However, this incurs significant bandwidth and computation overhead in oblivious databases. For example, in ORAM-based systems, where each request retrieves *O(logN)* objects, scanning a table of size $|T|$ results in a prohibitive bandwidth cost of $O(|T| * \log |T|)$.

Indexing the data and storing the index locally at the proxy ensures high security but incurs significant storage and computational overhead. Whereas, outsourcing the index structure to the cloud and allowing it to traverse the index to find matching rows expose query-dependent access patterns to an adversary. This leakage is particularly problematic for index structures like B-trees but making them oblivious requires multiple rounds of communication to read each tree level obliviously [12], which can severely reduce system throughput.

Solution: OasisDB addresses this challenge by creating inverted indices and outsourcing them to the cloud. Inverted indices also benefit from easy translation to key-value pairs, enabling OasisDB to retrieve index entries in one round. As seen in the example in Figure 2b, each index key has the format of ($T_{name}|C_{name}\_idx|idx\_key$) and the value is a list of primary keys of rows containing the index key in column C_{name}. For security, OasisDB combines *all* key-value pairs – be it from the tabular data or from the index – into a single set before partitioning the data for outsourcing.

4.2 System Overview

Having described how OasisDB represents tabular and index data, we now discuss the system design, starting with a naive approach.

Naive approach 1: Because OasisDB stores data in key-value pairs, the simplest way to execute SQL queries at scale is to partition data across multiple oblivious KV-store instances, with each shard handling SQL queries. This eliminates the server as a bottleneck by sharding its data as well. However, existing oblivious KV-engines require significant modifications with this approach as they do not understand the semantics of SQL queries.

Naive approach 2: A simple fix for the above challenge is to add a new component, a *Query Resolver*, that processes SQL queries, translates each query to its appropriate key-value requests, and routes them to the respective KV-shards. However, this solution still leaves several challenges unresolved.

Challenge 1: Ensuring global obliviousness. While each KV-shard ensures obliviousness of its portion of the data, this does not guarantee obliviousness of data *across* shards, especially under skewed workloads. If two identical SQL queries repeatedly access the same set of KV-shards, an adversary may infer that those shards contain the relevant data for those queries. In contrast, if two distinct queries access different sets of shards, their access patterns will appear uniform. The difference in shard access frequencies thus leaks the skew in the workload to an adversary.

Challenge 2: Shuffling data across shards. One way to fix the issue is to shuffle each accessed object to a random shard after each request, which ensures that any two SQL queries access random shards. This can be done in two ways: (i). A centralized resolver maps each accessed key across SQL queries to random KV-shards, similar to [6, 66]. This introduces coordination between conflicting SQL queries accessing the same keys and notably, the resolver becomes a scalability bottleneck. However, the KV-stores remain unchanged. (ii). Alternatively, allowing KV-shards to shuffle data among themselves removes the resolver bottleneck but requires meticulous coordination not only between KV-shards but also with the resolver to track the locations of objects. Inter-shard coordination necessitates non-trivial changes to the underlying KV-stores, defeating OasisDB's goal of reusing existing KV-stores.

Challenge 3: Hiding query result volumes. Beyond access patterns, the result sizes of SQL queries impose a major challenge in oblivious RDBMSs. Shuffling or not, naively requesting the exact number of key-values of a SQL query from the server reveals the size of the query, which can be exploited to recover data and/or queries [19, 24, 25, 30, 32, 36, 38, 49, 50, 59]. Whereas fully secure solutions, such as padding to the worst-case size ($O(|T|)$ for point/range queries, $O(|T1| * |T2|)$ for joins), impose impractical overheads.

Solution: OasisDB integrates key features from both naive approaches while addressing their shortcomings. From the first, it adopts server sharding alongside KV-shards. As shown in Figure 1, its top layer, `Executors`, runs the KV-store's oblivious schemes on a subset of data. From the second, it incorporates the query resolver, `Resolver`, for communicating with the clients and translating SQL queries to key-values. Additionally, OasisDB introduces `Batcher` to eliminate the need for cross-shard shuffling while ensuring global obliviousness (§4.4). The simple yet effective design of the `Batcher` also protects against volume pattern attacks (§6).

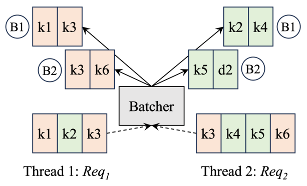


Figure 3: `Batcher` intermixing and distributing key-value requests from multiple SQL queries and routing to two `Executors` in two batches $B1$ and $B2$ with a batch size $B_R = 2$.

The following sections explains each layer of OasisDB.

4.3 `Resolver` Layer

The first layer of OasisDB consists of one or more `Resolvers` that receive potentially concurrent SQL requests from clients. This layer executes the query processing logic and maintains metadata essential for efficient query execution, as we explain in §5. For instance, it tracks which columns of a table have an index. Additionally, OasisDB assumes that incoming queries are both syntactically and semantically valid but the system's design can be easily extended to incorporate a validation module if needed.

A `Resolver` transforms a SQL query into a set of key-value pairs to be retrieved (or updated) on the server with §5 detailing the specific key-value pairs accessed for each query type. Upon receiving a SQL query, a `Resolver` assigns it a unique request ID, req_{id}, and constructs a request of the form $\{req_{id}, keys, values\}$, where $keys$ represent tabular or index keys while each entry in $values$ either contains an updated value for the corresponding key or remains empty for read requests. This design ensures that the request structure remains identical for both read and update queries, maintaining uniformity in query processing. Once the request is generated, the `Resolver` forwards it to a randomly chosen `Batcher`. Upon receiving all the necessary key-value responses from the above layers, a `Resolver` may further perform some post-processing such as sorting or aggregations before responding to the client.

The lifespan of a single SQL query can be relatively long, as it typically involves at least two sequential rounds of communication with the storage server—for instance, retrieving an index first, followed by accessing the corresponding tabular data. Given this, it is crucial for `Resolvers` to scale horizontally by distributing SQL queries across multiple processes. However, this distribution introduces challenges in handling update requests and thus, limits the type of updates that OasisDB handles, as we discuss in §5.5.

4.4 `Batcher` Layer

The primary security risk in sharding an oblivious KV-store proxy arises when SQL queries access some server shards more than the other, revealing the skew in client workloads. For example, in PathORAM, a shard accessed by a frequent SQL query will (obliviously) read more paths from its subtree compared to the other subtrees, violating the global oblivious property of uniform accesses over the entire dataset. Additionally, retrieving the key-values of a SQL query in isolation reveals the volume pattern of each request. We observe that the root cause for both leakages is the query-dependent, non-uniform request load to each shard.

Based on this insight, OasisDB addresses both issues by introducing the `Batcher` layer that securely batches key-value requests to ensure uniform workload to all `Executors`, regardless of any skew in the client requests. When a `Batcher` receives a request from a `Resolver` containing one or more key-value pairs, it hashes each key to determine its `Executor` and enqueues the request in the corresponding queue. The `Batcher` monitors these queues and, once each queue reaches a pre-configured batch size B_R, it issues a batch of B_R key-value requests, potentially containing a mix of `Get` and `Put` operations, to each `Executor`.

However, this design presents a challenge: some queues may not receive B_R requests for an extended period, causing delays for clients awaiting responses. To mitigate this, a background thread runs a timer with a pre-configured timeout. If the timer expires and at least one queue contains a request, the system fills all batches up to B_R by inserting fake key-value requests if necessary (§4.5 details how OasisDB handles them) before sending the batched requests to all `Executors`. Note that a batch of B_R requests may include duplicate keys requested by multiple `Resolvers`.

To provide additional protection against volume pattern leakages and to enable multi-user settings, a `Batcher` processes concurrent requests from `Resolvers` and intermixes the key-values (with potential duplicates) requested by multiple SQL queries. On the flip side, key-value pairs of a single SQL query might be split across multiple queues, as seen in Figure 3 where key-values from two SQL queries are intermixed in batches sent to two `Executors`. This raises a subtle challenge of how to route the responses back to the correct SQL query.

When a `Batcher` receives a request from a `Resolver`, it spawns a separate thread to process the request. Once responses arrive from the `Executors`, the `Batcher` must correctly match each key-value response to the corresponding thread and return the values to the appropriate `Resolver`. Achieving this requires careful thread synchronization and interrupt mechanisms to wake up waiting threads. OasisDB addresses this challenge using channels (implemented as Go channels in our prototype). Specifically, each request thread creates a channel and associates it with key-value pairs being queued. When a response arrives from an `Executor`, the `Batcher` identifies the corresponding channel and forwards the response through it. Once all responses for a request are received, the handling thread sends a final response back to the `Resolver`.

4.5 `Executor` Layer

The top layer of OasisDB, consisting of one or more `Executors`, is responsible for obliviously accessing key-value pairs from the untrusted storage servers. OasisDB's modular design allows an application to choose from many – but not all – oblivious KV-stores and scales them for oblivious query processing; this section formally defines what KV-stores can be integrated with OasisDB. To demonstrate the flexibility of this approach, we present OasisDB with two different oblivious KV-stores: PathORAM [67] and Waffle [48].

During initialization, OasisDB transforms all tabular and index data into key-value pairs, as described in §4.1. It then uses hashing to randomly distribute all key-value pairs denoted as N, across s available `Executors`, with each shard storing $\lceil N/s \rceil$ objects.

When an `Executor` receives a batch of B_R key-value requests from a `Batcher`, it executes the oblivious protocol dictated by the chosen KV-store for its assigned subset of data. For instance, a PathORAM-based `Executor` retrieves B_R random paths per batch of requests [67], while a Waffle-based `Executor` processes the batch by generating α, β-uniform accesses to the server [48], independent of the skew in client workload. Importantly, a batch of B_R requests may contain duplicate requests to the same key-value pair, which are handled by the oblivious KV-stores. Since the original PathORAM [67] design processes requests sequentially, we optimize its throughput in our implementation by handling batched requests in parallel using techniques from [62, 64]. Once an `Executor` completes its execution of B_R read or write requests on the server, it returns the responses to the `Batcher` that issued the batch.

OasisDB introduces a minor but necessary modification to the underlying oblivious KV-store. As discussed in §4.4, if a queue for a given `Executor` has fewer than B_R requests when the timeout occurs, the `Batcher` fills the batch with dummy requests. This means that an `Executor` may receive requests for non-existent keys. If the `Executor` were to ignore such requests, an adversary could infer the number of such dummy requests from the varying request load across different storage shards, breaking obliviousness. To prevent this, OasisDB requires the underlying KV-store to generate fake accesses for these dummy requests. Specifically, a PathORAM-based `Executor` issues fake read-path requests, while a Waffle-based `Executor` adds fake accesses to real objects. In both cases, the `Executor` responds with dummy values to the `Batcher`. This ensures uniform number of accesses to each database shard.

Oblivious KV-stores compatible with OasisDB: We now discuss: What criteria must an oblivious KV scheme meet to ensure compatibility with OasisDB?

Let O be an obliviousness scheme that hides access patterns for a set of outsourced key-value pairs. Consider a sequence of m accesses, $A_m = \{(op_1, k_1, v_1), (op_2, k_2, v_2), \cdots (op_m, k_m, v_m)\}$ where each op_i indicates a read or write operation, $k_i \in D$ is the accessed key from the database D, and v_i the updated value for writes (or $\perp$ for reads). An oblivious scheme O takes A_m as input and produces a transcript $\tau(D)$ of server accesses over the outsourced database D such that τ guarantees an obliviousness 'property' `prop`. The exact definition of `prop` depends on the specific scheme. In PathORAM, the property guarantees that $\tau(D)$ accesses randomly chosen paths on the server, whereas in Waffle, $\tau(D)$ exhibits α, β-uniformity. In both cases, the properties guarantee that the server accesses in $\tau(D)$ appear to be independent of the actual accesses in A_m thereby preserving obliviousness.

Now consider a sharded KV-store where D is partitioned into s disjoint chucks such that $D = (d_1 || d_2 .. || d_s)$. Applying the oblivious scheme O to each shard produces a transcript $\tau(d_i)$. Assume a secure batching scheme $\mathcal{B}$ exists that (i) generates batches of requests in consecutive rounds, and (ii) ensures that in each round, every shard receives the same number m' of requests. Then in each round j, the oblivious schemes across s shards produce $\tau(d_1)_j, \tau(d_2)_j, \cdots \tau(d_s)_j$. This implies that the overall transcript $\tau(D)$ is given by

$$\tau(D) = \sum_{j=1}^{r} [\tau(d_1)_j || \tau(d_2)_j || \cdots || \tau(d_s)_j]$$

where r is the number of the rounds the batching scheme $\mathcal{B}$ runs for. In simpler words, $\tau(D)$ over the entire database is the combination of transcripts generated by all shards d_1 to d_s across all rounds of requests processed by $\mathcal{B}$.

Definition 1. *An oblivious scheme O is sharding-safe iff the obliviousness property* `prop` *guaranteed for each shard implies that a derived property* `prop'` *holds over the entire dataset when accesses are generated using the secure batching scheme $\mathcal{B}$. Formally,*

$$\texttt{prop}\{\tau(d_1)\}, \texttt{prop}\{\tau(d_2)\}, \cdots \texttt{prop}\{\tau(d_s)\} \implies \texttt{prop}'\{\tau(D)\}.$$

The oblivious schemes of PathORAM and Waffle are *sharding-safe* according to Definition 1. In PathORAM, partitioning a single tree into s subtrees and applying its scheme independently to each shard ensures that every shard accesses the same number of randomly chosen paths from its subtree. This preserves the derived obliviousness property `prop'`, identical to `prop`, over the entire dataset D. Similarly, applying Waffle's scheme to s shards ensures that each shard accesses its portion of the data the same number of times while preserving α, β-uniformity, which is the obliviousness property of Waffle. As a result, both PathORAM and Waffle can be integrated into OasisDB without compromising privacy.

Pancake [29] is a recent oblivious KV-store that hides access patterns by smoothening the access frequencies of outsourced objects. However, simply sharding the data into smaller subsets d_i and applying Pancake's frequency-smoothing technique independently to each shard fails to ensure uniform access patterns across the entire dataset D, even when using a secure batching scheme. This is because shards containing popular keys will exhibit higher overall access frequencies than those maintaining unpopular keys, as discussed in [70]. Consequently, Pancake is not sharding-safe according to Definition 1 and cannot be directly integrated into OasisDB. That said, Pancake can still be sharded using specialized techniques tailored to its specific scheme, as shown in [70].

5 OBLIVIOUS QUERY PROCESSING USING OASISDB

After outlining the roles and responsibilities of each layer in OasisDB, this section delves into how OasisDB executes SQL queries obliviously. While OasisDB incorporates specific techniques for query execution, its modular architecture allows database administrators to modify the query processing logic within the `Resolver` layer without requiring changes to the other layers. OasisDB supports point, range, join, aggregate, order by, and group by operators with composite queries combining two or more of these operators.

5.1 Point Queries

We begin with a basic SQL query containing one or more point predicates in the `WHERE` clause. A `Resolver` upon receiving a SQL query first identifies indexed columns in the `WHERE` clause. For all indexed columns, it generates a `Get` request to retrieve the corresponding index key-value pairs, as shown in Figure 2b. The request traverses via `Batchers` and `Executors` to the server, which responds with the values for the index keys. Each value contains concatenated primary keys associated with the index key. The `Resolver` then identifies the overlapping primary keys across all retrieved keys and sends a `Get` request to fetch the projected columns in the query.

This approach of leveraging an inverted index to process point queries privately is also utilized in searchable encryption schemes such as Kamara and Moataz [35] and SEAL [19]. However, the scheme in [35] is non-oblivious and, therefore, susceptible to access pattern attacks. SEAL [19] mitigates this leakage by employing ORAM. Nevertheless, SEAL's tunable-leakage design exposes the sub-ORAM responsible for frequently queried tuples, revealing access patterns. OasisDB protects from both types of leakages.

5.2 Range Queries

Next we consider queries where one or more predicates in the `WHERE` clause consist of a range filter with a lower bound l and/or upper bound u.

Plaintext databases typically handle range queries efficiently by using indexing structures like B^{+}-trees and traversing them to locate the leaf nodes corresponding to bounds l and/or u. However, traversing an outsourced B-tree to find the desired range requires *logN* (i.e., the tree height) *rounds* of communication between the proxy and the cloud server [12], often over a WAN. Such communication delays significantly degrade system performance.

Alternatively, indexing mechanisms designed for range searches on encrypted data (as discussed in [20]) typically build indexes over the entire *domain* of a column being filtered. While this approach works for attributes like `age` or `height/weight` with relatively small domains, it becomes impractical for attributes with large or indefinite domains, such as `salary` or `date-of-birth`.

Naively addressing the above two challenges by expanding a range query into a series of index-based point queries (§5.1) can lead to significant bandwidth and computation wastage because the expansion may include many point queries on non-existent data.

OasisDB addresses this inefficiency by employing Bloom filters [8] to filter out non-existent index keys. During initialization, OasisDB creates an index for each range attribute based on the database instance rather than the attribute's domain and inserts the index keys into a Bloom filter maintained by all `Resolvers`. At query time, the Bloom filter allows a `Resolver` to filter out non-existent keys early in the process. The `Resolver` then sends a `Get` request for the index keys that pass the Bloom filter and proceeds with the second round to retrieve the tabular data.

Note that a Bloom filter returning false positives for non-existent index keys does not impact the correctness of a range query because `Executors` treat them identical to dummy keys (as explained in §4.5) and return a dummy response, which are filtered out by `Resolvers`. Since Bloom filters never return false negatives, OasisDB returns correct range responses. Similar to point queries, range queries incur two sequential rounds of communication - one to retrieve the index and one for the data. The benefit of Bloom filters is demonstrated in our experiments §7.4.

5.3 Joins

Join queries in SQL combine two (or more) tables based on a specified join condition and are of the form '`SELECT col_names FROM t1, t2 WHERE t1.attr =t2.attr`', joining tables `t1` and `t2` on an attribute `attr`. *Equi-joins* consist of an equality predicate, whereas the rest are considered *non-equi-joins*. OasisDB currently supports binary equi- and non-equi-joins, although the `Resolver` can be easily extended to support multiway-joins.

Existing schemes such as SEAL [19] support oblivious joins by either streaming each `attr` value from table `t1` and performing an oblivious point query on `t2` to retrieve matching records (equivalent to an index nested loop join) or by utilizing the technique from [35] to pre-compute a join-map. The join-map identifies primary keys from tables `t1` and `t2` with matching `attr` values, which can then be offloaded to the server and streamed to join the tables. Whereas, Chang et al.[13] use ORAM-based B-tree indexing to retrieve matching join attributes. However, these approaches incur high bandwidth overhead due to streaming an entire table or join-map, or require multiple communication rounds to traverse the B-tree obliviously. Reducing this overhead by storing the pre-computed join-map locally requires non-trivial local storage.

To achieve high throughput while minimizing storage and bandwidth requirements, OasisDB uses Bloom filters to locally store pairs of primary keys from tables `t1` and `t2` that satisfy the join condition during initialization. For join queries with additional filter conditions (as observed in all join queries used for benchmarking [4, 5] OasisDB), a `Resolver` first retrieves the index data from the server and extracts the primary keys of the filtered table(s). If both tables include a filter, the `Resolver` computes the cross-product of all filtered primary keys from the two tables and checks them against the Bloom filter. For pairs that pass the Bloom filter, the `Resolver` retrieves the columns projected in the join query. Similar to point and range queries, this approach incurs two sequential rounds of communication: one to retrieve the index and another for the data.

Bloom filters return false positives with non-zero probability, which can lead to returning tuples that do not satisfy the join condition. To address this, OasisDB retrieves the join attribute `attr` from both tables for the primary keys passing the filter along with fetching the projected columns. The system returns only the tuples from `t1` and `t2` that actually match the join attribute, and discards the other retrieved tuples.

5.4 Aggregates, Order By, and Group By

OasisDB supports aggregates such as `COUNT, SUM, AVG, MIN, MAX`, along with `ORDER BY ASC|DESC`, and `GROUP BY` operators. Depending on the presence of point, range, and/or join conditions, OasisDB first retrieves the relevant columns required for query evaluation using the techniques discussed in the previous sections. Once the data is retrieved, OasisDB applies post-processing techniques at the `Resolver` layer to compute aggregates, sort the results, or group them as specified in the query.

5.5 Updates

OasisDB provides limited support for updating tabular data, specifically allowing updates to a single column in a table filtered by the table's primary key. It also restricts updates to columns without an index or Bloom filter. Although restrictive, this design is sufficient to handle all four update statements in the Epinions dataset [4], a real-world customer review application. Among existing oblivious RBDMSs [19, 22, 75], only ObliDB [22] supports updates by enforcing serial execution of queries, which significantly limits its throughput, as we show in §7.1. The distributed design of OasisDB, along with concurrency, cannot trivially support complex update queries; extending it for such operations would require a transactional layer executing distributed commits *obliviously*. No such work exists today and addressing this challenge is a promising avenue for future work.

Upon receiving an allowed update request, a `Resolver` generates a `Put(k,v)` request to modify the specified key-value on the server based on the table name, updating column name, and the primary key in the update query (see Figure 2). However, OasisDB does not support updates of the read-modify-write format.

Serializing SQL Requests: The constraints on update queries along with the linearizability of the underlying KV-store allow OasisDB to maintain a serialized execution. Since an update request r_u in OasisDB modifies only a single cell, any read request r_r either observes the updated value (indicating r_u occurred before r_r) or does not (indicating r_r occurred before r_u). This ensures a serial order between any two requests, r_u and r_r.

6 SECURITY OF OASISDB

OasisDB is a scalable and secure framework built on an oblivious KV-store to maintain and process relational data. Informally, assuming the query predicates and update values are encrypted, OasisDB ensures the following privacy guarantees: (i) it hides the exact rows satisfying query predicates of a SQL query, (ii) it conceals the relative frequency of rows satisfying user queries, and (iii) it obscures the exact result volume of SQL queries.

Recall the definition of a sharding-safe oblivious KV-store in Definition 1, which is satisfied by both PathORAM [67] and Waffle [48] used in OasisDB.

Theorem 1. *Let $U(m)$ be an independently, uniformly drawn random number in $[1, m]$. Given PathORAM [67], a sharding-safe oblivious KV-store that satisfies obliviousness property $\tau(d) = \sum_{k=1}^{n} U(|d|)$, OasisDB using PathORAM satisfies the obliviousness property*

$$\tau(D) = \sum_{j=1}^{r} \sum_{i=1}^{s} \sum_{k=1}^{B_R} U(|d_i|)$$

where s is the number of PathORAM shards and r the number of batched requests generated by OasisDB.

Proof. PathORAM's obliviousness property per shard ensures that each shard accesses a uniform, independent random path from its subtree for each key-value request. To achieve the global obliviousness property, OasisDB ensures that the same number of random paths be accessed per subtree across all shards. Hence, the obliviousness property of PathORAM can be repeatedly composed a fixed number of times due to the independence of the random variables. Note that if different shards were to access varying numbers of random paths, this would introduce leakage, potentially revealing the locality of certain objects (e.g., popular objects).

In detail, OasisDB's security hinges on the functionality of the `Batcher` layer and demonstrating that its operations remain independent of the specific key-value pairs generated by the input SQL workload. When `Resolvers` translate SQL queries into key-value requests and forward them to `Batchers`, each `Batcher` sends batches of B_R key-values at a time to all `Executors`, regardless of the specific objects requested by client queries. Additionally, OasisDB's modification of `Executors` to process non-existent keys as if they were regular keys (as detailed in §4.5) ensures that the accesses generated on the server is a function of B_R, i.e., each PathORAM shard reads exactly B_R paths per batch. This creates storage accesses that satisfy the global obliviousness property. □

Theorem 2. *Given Waffle [48], a sharding-safe oblivious KV-store satisfying the obliviousness property α, β-uniformity, OasisDB using Waffle satisfies the obliviousness property:*
$\lfloor \alpha/B_R \rfloor (s-1)B_R + \alpha$, $\lceil \beta/B_R \rceil (s-1)B_R + \beta$-uniformity.

Proof. Waffle defines α, β-uniformity (§2), which, loosely speaking, ensures that each key on the server is read (and deleted) within at most α accesses after being written, while each key in the cache remains for at least β accesses. In Waffle, the obliviousness property per shard holds if each shard is initialized with identical system parameters, enabling the computation of α, β-bounds. In addition, OasisDB ensures uniform accesses to shards, preventing any key-value pair from violating these bounds across the entire database.

To derive the composed bound, we need to compute the minimum and maximum number of accesses to other shards between reading (and deleting) a key in a shard. This amounts to $(s-1)B_R$ accesses per intermediate batch with at least $\lfloor \alpha/B_R \rfloor$ and at most $\lceil \beta/B_R \rceil$ such batches. The derived bound, which holds for the entire database in OasisDB, follows accordingly. This represents a worst-case scenario for α, β-bounds. Because accesses by a proxy shard are confined to its server shard ensuring local α, β-uniformity, the obliviousness is further reinforced. □

Proposition. *Assuming OasisDB employs a sharding-safe oblivious KV-store with an obliviousness property `prop`, the accesses generated by OasisDB in serving SQL queries will satisfy the global obliviousness property `prop'`.*

Proof. From the arguments for OasisDB with PathORAM and Waffle, we can assert that with a sharding-safe oblivious KV-store, OasisDB preserves the global obliviousness property `prop'`. □

6.1 Volume pattern protection in OasisDB

Volume pattern attacks [19, 24, 25, 30, 32, 36, 38, 49, 50, 59] exploit the result sizes of SQL queries to infer either protected query predicates or the underlying data itself. These attacks work as follows at a high level: a persistent adversary has access to the transcript of interactions between the system being evaluated (e.g., OasisDB) and the cloud database. The transcript includes encrypted keyword or range query predicates and any information disclosed in the leakage profile of the system. For example, a volume revealing oblivious database would reveal the exact result size of each query as part of its leakage profile.

These attacks often rely on the ability to isolate each query's volume precisely. Furthermore, the attacker needs to collect certain number of queries and observe the above leakages to mount a successful attack. Auxiliary information on query distribution and data density assumptions can reduce the number of required queries or refine the attacker's reconstruction process [7, 30–32, 40]. The adversary, which has access to auxiliary information and the observed leakage patterns, attempts to match encrypted query tokens or searched values to plaintext keywords or values from the auxiliary data. Several recent systems [9, 19, 33, 56, 61, 71] mitigate such attacks by per-query obfuscation of volumes.

Our approach. OasisDB does not employ per-query volume-hiding strategies. Instead, it prevents an adversary from reliably isolating the result volumes of individual SQL queries in the first place, by generating uniform workload of exactly B_R requests per batch to the KV-stores. Specifically, if r is the collective result size of either one or more SQL queries, OasisDB sends i rounds of $B_R \times s$ key-value requests to the server, where s is the number of `Executors` and i is the smallest value such that $(i-1) \times B_R \times s < r \leq i \times B_R \times s$. This ensures that even during periods of low activity, the server still observes multiples of B_R key-value accesses. However, the choice of B_R can impact the level of protection OasisDB provides against volume pattern attacks because lower B_R values hint at the actual volume of SQL queries.

To assess the efficacy of OasisDB's defense mechanism, we launch two volume pattern attacks: i) keyword-based attack [7] for point queries, and ii) range-based attack [31] for range queries. While for space constraints, [1] presents the detailed attacks with results, we present the findings of the range attack here.

The range attack [31] considers a single shard database with no fake or index key accesses to the database, and hence can be viewed as a 'worst case attack' for OasisDB. We apply the attack for two different datasets, MIMIC-PC [23], a medical dataset, and Salaries [28], a human resource dataset. The attack compares three batching techniques: 1) `NoBatch`, which performs no batching, revealing the exact volume per SQL query, 2) `FixedQ`, which batches a fixed number of SQL queries, resulting in varying number of key-values per batch retrieved from the server, and 3) `FixedKV`, which fixes the size of key-values retrieved from the server to B_R per batch. Through these attacks, we aim to answer the following questions.

1. Is fixed size batching necessary for volume pattern hiding? The attack on the Mimic-PC dataset shows that, with as few as 1,000 observed queries, the error in recovery drops to 7% for `NoBatch` and 11% for `FixedQ`[2]. In contrast, `FixedKV` (OasisDB's approach) yields a recovery error of 96%, effectively equivalent to random guessing. These results underscore the importance of fixed-size key-value batching in defending against such attacks.

2. Can any B_R value successfully protect against volume attacks? To answer this question, we perform range attacks [31] on the `FixedKV` setting by choosing $B_R \in \{2, 50, 100\}$. On Mimic-PC [23] dataset, the attack fails to recover any data even at $B_R = 2$, while for the Salaries [28] dataset, the attack achieves a partial recovery[2] at $B_R = 2$ and 50, with an error rate of 38–45%. Only at $B_R = 100$ does the attack become fully ineffective. The difference stems from the domain size of the attacked range attribute: Mimic-PC's has a domain size of 2684, whereas Salaries' has only 395. This suggests that administrators should configure OasisDB with a higher B_R when sensitive and frequently queried attributes have small domains; otherwise, even smaller B_R values can offer sufficient protection.

7 EXPERIMENTAL EVALUATION

This section investigates the performance of OasisDB, integrated with Waffle [48] and PathORAM [67] as KV-stores, and compare both with other baselines across various query types. In particular, we answer the following questions:

- How does OasisDB perform compared to its baselines? (§7.1)
- How does OasisDB scale? Which components of the system contribute the most to its performance? (§7.2)
- How does fake requests and skewed workloads affect the system's performance? (§7.3)
- How does its query optimizations impact performance? (§7.4)

Implementation and Experimental Setup: We implement OasisDB in Go, integrating Waffle [48] and an optimized PathORAM implementation [1] as the underlying KV-stores. Experiments run on machines with Intel E5-series CPUs, 10 Gbps networking, and up to 256GB RAM. A multi-threaded open-loop client generates concurrent SQL queries to measure throughput and latency. We emulate a WAN link with a default 10ms round-trip delay between the trusted proxy and untrusted Redis [60] cloud storage. We refer readers to the full version [1] for further details.

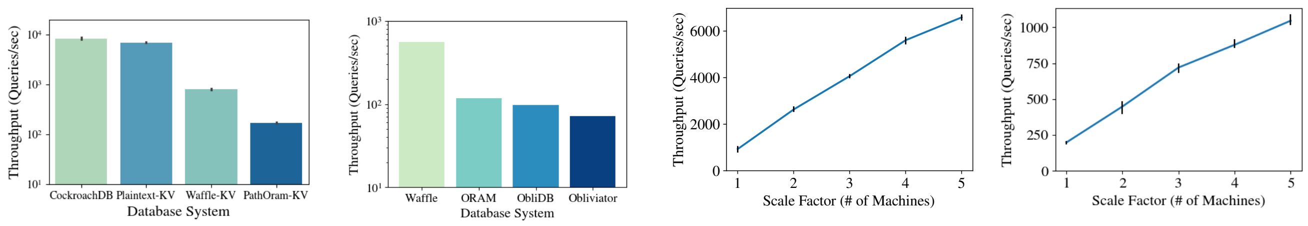
Baselines: We compare OasisDB with three baselines:
1. CockroachDB [68] acts as an insecure baseline in evaluating the privacy overhead incurred by OasisDB. We chose CockroachDB because its architecture, similar to OasisDB, builds on a scalable KV storage engine to store relational data and support SQL queries [15].
2. ObliDB [22] acts as one of the oblivious RDBMS baselines. ObliDB utilizes hardware enclaves (i.e., Intel SGX) to obliviously process SQL queries on the server. However, ObliDB assumes the presence of oblivious memory within the enclave — a functionality not provided by Intel SGX. Hence, its evaluations are without accounting for the associated costs of oblivious memory. ObliDB is deployed on an Intel SGX v2 (Xeon E2374G) machine with 128GB RAM.
3. Obliviator [51] is a recent oblivious system supporting SQL queries using hardware enclaves. It differs from ObliDB in two significant ways: i) It removes the assumption of oblivious memory, providing a stronger security guarantee, and ii) It enables multi-threaded query execution, similar to OasisDB. We run Obliviator on Azure with DC32ds_v3 VM.[3]

We note that both ObliDB [22] and Obliviator [51] lack support for multi-user environments and process queries sequentially. Neither systems are scalable and by default, they do not hide query volume patterns. This limits their performance, particularly in WAN settings. However, both systems remain suitable for cloud-based deployments where on-premise resources are scarce and their limitations are acceptable.

Datasets and query types: We evaluate OasisDB and its baselines on two datasets:
1. Epinions [4]: Dataset based on epinions.com, a consumer review platform. This dataset models user interactions with other users and products wherein users rate and review the products (or items). We use Benchbase [21] to generate a dataset comprising of over 1 million rows distributed across five tables and assign primary and foreign keys with uniform distribution. 1M rows of the dataset results in over 6M key-values, including any constructed indexes. Unless stated otherwise, our workload consists of 13 selection queries (Point, Range, Aggregate and Join) and 4 update queries, with predicates chosen randomly from the dataset.
2. Big Data Benchmark [5]: It consists of two tables, `PageRank` and `UsersVisited`, with information on multiple websites and their visits with 360K and 350K rows. We compare the oblivious

[2]As a reference, an error close to or above 50% suggests that the attack is ineffective, while an error below 50% indicates partial or practical recovery.

[3]We were unable to run ObliDB on Azure as it requires EPID-based SGX attestation, which is incompatible with Azure's datacenter attestation protocol (DCAP).



(a) Plaintext baseline comparison **(b) Oblivious baseline comparison** **(c) Scaling with Waffle** **(d) Scaling with PathORAM**

Figure 4: Baseline and scaling experiments of OasisDB

baselines ObliDB [22] and Obliviator [51] on this dataset. The workload for this dataset consists of 3 queries (Point, Range, and Joins).

Of the 20 queries across the two benchmarking workloads, most being composite queries, Table 2 depicts the distribution of various SQL operations tested in the experiments.

Op. Type	Point	Range	Join	Aggr.	Grp. By	Ord. By	Update
Cnt.	11	4	4	5	1	3	4

Table 2: Distribution of operator types in the experiments.

7.1 Comparison with Baselines

Comparison with CockroachDB: This experiment evaluates the cost of achieving obliviousness by comparing OasisDB with a plaintext baseline, CockroachDB[14, 68]. We use BenchBase[21] to benchmark CockroachDB on the Epinions dataset [4] with an identical workload to OasisDB. In addition to the two oblivious KV-stores, we compare CockroachDB with an insecure variant of OasisDB, where the `Executor` layer employs a plaintext KV-store, which bypasses data encryption, decryption, and any oblivious mechanisms to access data. This version of OasisDB will identify the overhead introduced by the `Resolver` and `Batcher` layers of OasisDB. For this experiment, OasisDB is deployed with a scale factor of one, while CockroachDB runs on a single-node cluster (i.e., without scaling or replication). The client process generates maximum 1000 concurrent requests across all baselines.

The results shown in Figure 4a highlight that the plaintext KV version of OasisDB achieves identical throughput to CockroachDB. The two systems also incur identical average query latency with 137ms for CockroachDB and 141ms for OasisDB plaintext version. These results indicate that OasisDB's `Resolver` and `Batcher` layers introduce minimal overhead.

Using Waffle as the KV-store in OasisDB (with $\alpha = 5000$) results in a **9.2**× drop in throughput compared to CockroachDB, and the average query latency increases to 891 ms – **5.5**× higher than that of CockroachDB. When using PathORAM as the KV-store, we observe an even greater performance degradation relative to Waffle, due to PathORAM's stronger threat model, which incurs a $logN$ bandwidth overhead. These results underscore that OasisDB 's performance is heavily influenced by the choice of underlying KV-store.

Comparison with ObliDB and Obliviator: This experiment compares OasisDB with ObliDB [22] and Obliviator [51], two oblivious RDBMSs that leverage Intel SGX for oblivious query processing. The experiment uses the Big Data Benchmark [5] consisting of point, range, join, aggregate, group by, and order by operators. We introduce a 10ms round-trip WAN latency to both systems to simulate client-server communication, mirroring OasisDB 's setup. Since both baselines are not scalable, OasisDB is deployed with a scale factor of 1 for a fair comparison.

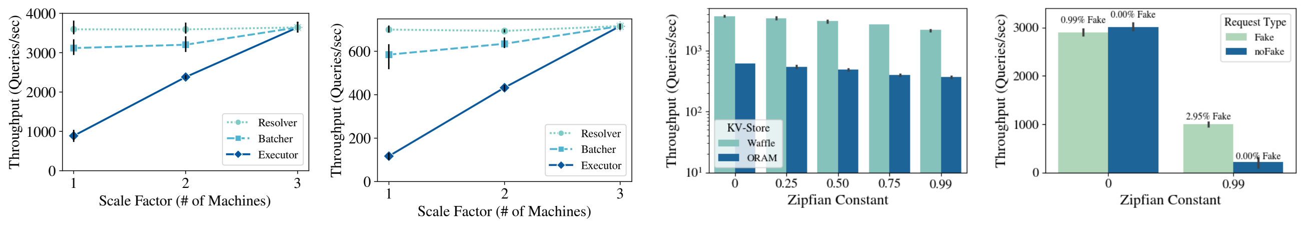
As shown in Figure 4b, OasisDB achieves **5.7x** and **7.7x** the throughput of ObliDB and Obliviator respectively when using Waffle, and **1.22x** and **1.64x** the throughput when using PathORAM. The lower throughput of both baselines stem from their sequential query processing, whereas OasisDB batches multiple requests to amortize communication and computation costs, improving throughput. However, batching increases query latency: ObliDB and Obliviator incur an average latency of under 20ms, whereas OasisDB incurs over 800ms latency. Although batching increases latency, it is necessary to hide volume patterns, which neither of the baselines ensure in this experiment.

7.2 System Scaling

This experiment evaluates OasisDB's ability to scale with the number of machines – a primary design feature of the system - by increasing the number of machines from one to five such that each machine deploys one `Resolver`, `Batcher`, and `Executor` process. For example, a scale factor of three implies three `Resolvers`, `Batchers`, and `Executors`, totaling nine processes deployed across three machines. This and the following experiments use Epinions data with clients generating 17 different SQL queries including updates.

Figures 4c and 4d illustrate the results of this experiment for OasisDB with Waffle and PathORAM, respectively. As shown, both versions of OasisDB exhibit linear scalability, demonstrating OasisDB 's ability to efficiently distribute workload across multiple machines. Specifically, as the scale factor increases, OasisDB distributes client requests across multiple `Resolvers`, which further distribute the load across multiple `Batchers`, and ultimately onto multiple `Executors`, improving system throughput.

Per-Layer Scaling: While the previous experiment demonstrated the overall scalability of OasisDB, it does not reveal which layer of its architecture contributes the most to this behavior. Therefore, we conduct a layer-wise scalability analysis by scaling one layer at a time from one to three processes, while keeping the other two layers fixed at three processes each. Hence, the end result of each experiment reaches the same configuration of three processes per



(a) Per layer scaling - Waffle **(b) Per layer scaling - PathORAM** **(c) Changing workload skew** **(d) Effect of fake KV requests**

Figure 5: Per layer scaling for the two versions of OasisDB and the effect of skew and fake requests on OasisDB.

layer. This experiment follows the same process of co-location as mentioned in the previous experiment but with varying the number of processes in one layer at time.

Figures 5a and 5b illustrate that the `Executor` layer has the largest impact on performance for both Waffle and PathORAM with the throughput increasing by ~**4x** for both compared to scale 1. This is because `Executors` execute resource-intensive oblivious algorithm for both versions and as they scale, the workload per process reduces, leading to a substantial improvement in performance. For the `Resolver` layer, we anticipated it to play a more vital role in scaling as each `Resolver` maintains context for each query across multiple rounds of communication with the server. The results are counterintuitive and indicate that `Resolvers` contribute the least to scaling. `Batchers` similarly exhibit a lower improvement of $\sim 20\%$ as the layer scales. From the insights of this experiment, an application can choose how many processes to deploy per layer depending on the required throughput.

7.3 Internal Experiments

This sections presents internal experiments on how OasisDB behaves under varying skew and the effect that fake queries have on the system. The full version of the paper [1] also presents an additional experiment measuring the performance changes under varying Round-Trip Time (RTT) between the trusted and untrusted domains.

Effect of varying skew: Thus far, client-generated workloads have selected query predicates uniformly from the set of possible attribute values. This experiment evaluates the impact of skew in these workloads, with results shown in Figure 5c. The experiment uses the default scale factor of three. We observe that throughput decreases by 0.4x for both KV-stores as the workload skew increases from uniform to 0.99 Zipf. This is because, under skewed workloads, many KV requests target the same `Executor`, leading to queue buildup at that node. In contrast, uniform workloads distribute requests evenly across all `Executors`. This pattern occurs even in plaintext RDBMSs; for instance, CockroachDB exhibits a 15.9% throughput drop when moving from a uniform to Zipf(0.99) distribution. Importantly, this performance degradation does not affect the security of OasisDB, as throughput is a client-observed metric. Each server shard continues to observe a stream of B_R key-value accesses, independent of access skew. Two of the volume pattern attacks launched on OasisDB — targeting uniform and skewed workloads — show negligible recovery success [1].

Effect of fake key-value requests in OasisDB: Recall from §4.4 that OasisDB adds fake requests if a preset timer times out and some queues have less than B_R requests. Because fake requests add additional overhead on the system, two natural questions that arise are: *i) Is adding fake queries to reach B_R batch size necessary for security?* and *ii) What effect does it have on system performance?* Section 6.1 answered the first question, establishing that the `FixedKV` mechanism is integral to mitigating volume pattern attacks. This section discusses the performance implications of fake requests.

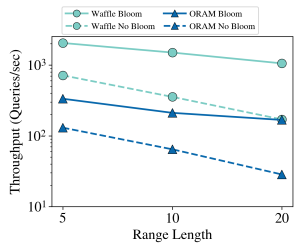
We experimentally evaluate the effect of adding fake key-value requests by comparing OasisDB with a `NoFake` version that removes this padding. Although for space constraints, we only present the performance with Waffle as the KV-store, the trend also held for PathORAM. The setup consists of 3 `Executors`, 1 `Batcher`, and 1 `Resolver` for both the `Fake` and `NoFake` versions, with equal number of concurrent clients injecting queries. Because security requires each `Executor` to receive B_R key-values, the `NoFake` version waits until each queue in a `Batcher` receives B_R requests.

Figure 5d depicts the measured throughput for both versions and the percent of fake requests added. We observe that OasisDB only adds 0.99% fake requests and has 3.65% lower throughput than the `NoFake` version when clients draw queries from a uniform distribution. However, when clients generate a skewed query workload, although the percent of fake queries in OasisDB increases to 2.95%, its throughput is **3.6x** higher than the `NoFake` version. This difference occurs because under skewed workloads, not all `Executor` queues receive equal request load, and hence, the `NoFake` version has to wait much longer until all queues receive B_R requests, significantly reducing the throughput. This experiment underscores the benefit of adding fake requests despite incurring wasted compute and communication.

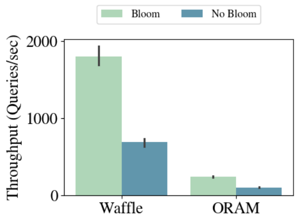
7.4 Oblivious Query Processing

We now evaluate range and join queries in OasisDB, comparing throughput with and without Bloom filters cross two key-value stores, Waffle and PathORAM.

For **Range Queries** (Figure 6a), we assess the impact of the Bloom filter in preventing unnecessary key retrievals. The `Bloom` approach efficiently filters keys without any false positives, whereas the `NoBloom` approach retrieves all range expanded keys from the server. The results indicate that Bloom filters improve the throughput by **3.3x** over the `NoBloom` approach for both KV-stores. As the range length grows, the throughput reduces for both KV-stores due



(a) Range queries



(b) Join queries

Figure 6: Range and Join queries with & without Bloom filters

to increased bandwidth and computation overhead, with PathORAM exhibiting a more significant drop due to its higher sensitivity to increased KV pair retrievals.

For **Join Queries** (Figure 6b), both `Bloom` and `NoBloom` approach retrieve the index keys of the tables being joined in the first round, obtaining the primary keys of tuples satisfying point and/or range predicates. The `Bloom` approach identifies potential joins from the cross product of these keys locally at the `Resolver`, whereas the `NoBloom` approach can only learn the join result by fetching the join attribute for the retrieved keys from round one. As explained in §5.3, OasisDB only incurs two rounds of communication, retrieving the projected attributes in round two, whereas `NoBloom` requires three rounds along with the overhead of fetching the attributes that do not join. On average the `Bloom` approach retrieves 6.2 KV pairs per query whereas the `NoBloom` one fetches 45 KV pairs. This results in `Bloom` improving the throughput by **2.5x** over `NoBloom` for Waffle and **4x** for PathORAM, indicating the benefit of Bloom filters.

8 RELATED WORK

This section reviews existing works related to OasisDB, categorizing them into scalable and non-scalable oblivious query processing systems. Broadly, existing scalable systems lack SQL support, while oblivious SQL processing systems do not scale, distinguishing them from OasisDB. Note that we focus the discussion on oblivious query processing systems and exclude those that rely on alternative privacy schemes such as MPC or homomorphic encryption.

Scalable Oblivious Data-stores: Several existing works have explored scalable oblivious key-value storage systems [6, 18, 64, 66, 70]. These systems achieve high-throughput and scalability but are limited to basic GET/PUT operations and do not support SQL semantics. Secondly, these systems are often tightly coupled with the underlying oblivious processing technique. Earlier works such as ObliviStore [66] and CURIOUS [6] cater towards sharding ORAM proxies for scalability but rely on a single load balancing process for security, which becomes a bottleneck. Snoopy [18] relies on TEEs for concealing access patterns and *linear scans the entire database per batch of requests*. ShortStack [70]'s scalability design is hard-wired to Pancake [29]'s oblivious scheme and does not generalize to other oblivious KV stores. Similarly, Treebeard [64]'s scheme is tied to tree-based ORAM schemes and cannot adapt to other oblivious KV schemes. Unlike these systems, OasisDB is agnostic to the underlying oblivious retrieval mechanism, and more importantly it supports SQL queries while hiding access patterns and obfuscating volume patterns.

Oblivious Query Processing: Many works propose oblivious query processing techniques. Works such as [12, 13, 39, 45] discuss oblivious mechanisms to process specific type of SQL queries such as joins or ranges using either TEEs or black-box ORAM approaches. They differ from OasisDB in their limited support for general SQL queries and their lack of scalability.

With regard to oblivious systems that support a wider range of SQL queries, works such as [2, 22, 51, 61, 75] rely on TEEs. Menhir[61] offers support to only single table queries using ORAM and differential privacy to prevent access pattern and volume pattern leakage, but it cannot perform joins. ObliDB [22], Opaque [75], Arasu and Kaushik's [2] and Oblivator [51] enable a wider range of SQL-like queries using customized query processing algorithms for each type of query. Additionally, Opaque and ObliDB rely on the existence of oblivious memory within the TEEs, which is currently not supported by any commercial TEE providers. These works primarily vary from OasisDB in their lack of scalability and support for multi-user settings.

SEAL [19] presents a range of searchable encryption techniques that support point, range, join, and aggregate queries with configurable security parameters. However, SEAL also relies on ORAM along with oblivious data structures [72] to ensure privacy, which can introduce significant performance overheads. Moreover, SEAL trades-off privacy in exchange for scalability, unlike OasisDB, and only supports single-user settings.

9 CONCLUSION

We present OasisDB- an oblivious and scalable RDBMS framework for storing and processing relational data. It addresses critical challenges in existing systems, such as supporting multi-user settings, ensuring scalability, and handling updates while hiding access and volume pattern leakages. OasisDB leverages existing oblivious KV-stores and proposes a distributed multi-layer architecture, with each layer scaling independently. We demonstrate the flexibility of OasisDB by integrating it with two distinct oblivious KV-stores, PathORAM and Waffle and discuss how OasisDB supports a wide range of SQL queries, while obfuscating query result sizes to thwart volume pattern attacks. Experimental results on real-world datasets, Epinions, show that OasisDB scales linearly with the number of machines. Compared to ObliDB and Obliviator, two oblivious RDBMS, as baselines, OasisDB exhibits benefits due to scalability and multi-user support. Future work on OasisDB will focus on support for OLTP workloads with serializablity guarantees and expanding query types to nearest neighbor searches.

ACKNOWLEDGMENTS

We gratefully acknowledge the support of NSERC for grants RGPIN-2023-03448, RGPIN-2023-03244, IRC-537591, the Government of Ontario and the Royal Bank of Canada for funding this research. This project was made possible in-part through the support of the National Cybersecurity Consortium and the Government of Canada (CSIN) (project id 2024-1261). This work benefited from the use of the CrySP RIPPLE Facility and the Tembo cluster at the University of Waterloo.

REFERENCES

[1] Haseeb Ahmed, Nachiket Rao, Abdelkarim Kati, Florian Kerschbaum, and Sujayya Maiyya. 2025. OasisDB: An Oblivious and Scalable System for Relational Data. Cryptology ePrint Archive, Paper 2025/1263. https://eprint.iacr.org/2025/1263

[2] Arvind Arasu and Raghav Kaushik. 2013. Oblivious query processing. arXiv preprint arXiv:1312.4012 (2013).

[3] Baffle. 2025. https://baffle.io. Accessed: 2025-07-11.

[4] Benchbase Epinions Dataset. 2025. https://github.com/cmu-db/benchbase/wiki/epinions. Accessed: 2025-07-11.

[5] Big Data Benchmark. 2025. https://amplab.cs.berkeley.edu/benchmark/. Accessed: 2025-07-11.

[6] Vincent Bindschaedler, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, and Yan Huang. 2015. Practicing oblivious access on cloud storage: the gap, the fallacy, and the new way forward. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS). 837–849.

[7] Laura Blackstone, Seny Kamara, and Tarik Moataz. 2020. Revisiting leakage abuse attacks. In 27th Annual Network and Distributed System Security Symposium (NDSS).

[8] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. Commun. ACM 13, 7 (1970), 422–426.

[9] Dmytro Bogatov, Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. 2021. εpsolute: Efficiently querying databases while providing differential privacy. In Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. 2262–2276.

[10] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. 2015. Leakage-abuse attacks against searchable encryption. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS). 668–679.

[11] Anrin Chakraborti and Radu Sion. 2019. ConcurORAM: High-Throughput Stateless Parallel Multi-Client ORAM. In 26th Annual Network and Distributed System Security Symposium (NDSS).

[12] Zhao Chang, Dong Xie, Feifei Li, Jeff M Phillips, and Rajeev Balasubramonian. 2021. Efficient oblivious query processing for range and knn queries. IEEE Transactions on Knowledge and Data Engineering 34, 12 (2021), 5741–5754.

[13] 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. 803–817.

[14] CockroachDB. 2025. https://www.cockroachlabs.com/blog/pebble-rocksdb-kv-store/. Accessed: 2025-07-11.

[15] CockroachDB - KV store layer. 2025. https://www.cockroachlabs.com/glossary/distributed-db/key-value-kv-layer/. Accessed: 2025-07-11.

[16] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. ACM Transactions on Computer Systems (TOCS) 31, 3 (2013), 1–22.

[17] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. 2018. Obladi: Oblivious serializable transactions in the cloud. In 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI). 727–743.

[18] Emma Dauterman, Vivian Fang, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. 2021. Snoopy: Surpassing the scalability bottleneck of oblivious storage. In Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP). 655–671.

[19] Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Saurabh Shintre. 2020. SEAL: Attack Mitigation for Encrypted Databases via Adjustable Leakage. In 29th USENIX Security Symposium (USENIX Security 20). 2433–2450.

[20] Ioannis Demertzis, Stavros Papadopoulos, Odysseas Papapetrou, Antonios Deligiannakis, Minos Garofalakis, and Charalampos Papamanthou. 2018. Practical private range search in depth. ACM Transactions on Database Systems (TODS) 43, 1 (2018), 1–52.

[21] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. PVLDB 7, 4 (2013), 277–288. http://www.vldb.org/pvldb/vol7/p277-difallah.pdf

[22] Saba Eskandarian and Matei Zaharia. 2019. ObliDB: Oblivious Query Processing for Secure Databases. Proc. VLDB Endow. 13, 2 (2019), 169–183.

[23] Alistair EW Johnson, Tom J Pollard, Lu Shen, H Lehman Li-Wei, Mengling Feng, Mohammad Ghassemi, Benjamin Moody, Leo Anthony Szolovits, Peter Celi, and Roger G Mark. 2016. MIMIC-III, a freely accessible critical care database. In Scientific Data, 3(1).

[24] Sky Faber, Stanislaw Jarecki, Hugo Krawczyk, Quan Nguyen, Marcel Rosu, and Michael Steiner. 2015. Rich queries on encrypted data: Beyond exact matches. In Computer Security–ESORICS 2015: 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part II 20. Springer, 123–145.

[25] Francesca Falzon, Evangelia Anna Markatou, Zachary Espiritu, and Roberto Tamassia. 2022. Range search over encrypted multi-attribute data. Cryptology ePrint Archive (2022).

[26] Oded Goldreich. 1987. Towards a Theory of Software Protection and Simulation by Oblivious RAMs. In Proceedings of the 19th Annual ACM Symposium on Theory of Computing (STOC). 182–194.

[27] Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. J. ACM 43, 3 (1996), 431–473.

[28] Government Digital Service. Organogram of staff roles & salaries of Government Legal Department. 2018. https://www.data.gov.uk/dataset/34d08a53-6b96-4fb6-b043-627e2b25840d/organogram-government-legal-department. Accessed Mar 10, 2025.

[29] 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. 2451–2468.

[30] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. 2018. Pump up the volume: Practical database reconstruction from volume leakage on range queries. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. 315–331.

[31] 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). 1067–1083.

[32] Zichen Gui, Oliver Johnson, and Bogdan Warinschi. 2019. Encrypted databases: New volume attacks against range queries. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. 361–378.

[33] Shanshan Han, Vishal Chakraborty, Michael T Goodrich, Sharad Mehrotra, and Shantanu Sharma. 2023. Veil: A Storage and Communication Efficient Volume-Hiding Algorithm. Proceedings of the ACM on Management of Data 1, 4 (2023), 1–27.

[34] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. Access pattern disclosure on searchable encryption: ramification, attack and mitigation.. In 19th Annual Network and Distributed System Security Symposium (NDSS).

[35] Seny Kamara and Tarik Moataz. 2018. SQL on structurally-encrypted databases. In Advances in Cryptology–ASIACRYPT 2018: 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2–6, 2018, Proceedings, Part I 24. Springer, 149–180.

[36] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'neill. 2016. Generic attacks on secure outsourced databases. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS). 1329–1340.

[37] Evgenios M Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. 2019. Data recovery on encrypted databases with k-nearest neighbor query leakage. In 2019 IEEE Symposium on Security and Privacy (SP). 1033–1050.

[38] Evgenios M Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. 2021. Response-hiding encrypted ranges: Revisiting security via parametrized leakage-abuse attacks. In 2021 IEEE Symposium on Security and Privacy (SP). IEEE, 1502–1519.

[39] Simeon Krastnikov, Florian Kerschbaum, and Douglas Stebila. 2020. Efficient oblivious database joins. arXiv preprint arXiv:2003.09481 (2020).

[40] 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.

[41] Marie-Sarah Lacharité and Kenneth G Paterson. 2015. A note on the optimality of frequency analysis vs. ℓ_p-optimization. Cryptology ePrint Archive (2015).

[42] Marie-Sarah Lacharité and Kenneth G Paterson. 2017. Frequency-smoothing encryption: preventing snapshot attacks on deterministically encrypted data. Cryptology ePrint Archive (2017).

[43] Kasper Green Larsen and Jesper Buus Nielsen. 2018. Yes, there is an oblivious RAM lower bound!. In Advances in Cryptology–CRYPTO 2018: 38th Annual International Cryptology Conference. 523–542.

[44] Jingwei Li, Chuan Qin, Patrick PC Lee, and Xiaosong Zhang. 2017. Information leakage in encrypted deduplication via frequency analysis. In 2017 47th Annual IEEE/IFIP international conference on dependable systems and networks (DSN). IEEE, 1–12.

[45] Rui Li, Alex X Liu, Ann L Wang, and Bezawada Bruhadeshwar. 2014. Fast range query processing with strong privacy protection for cloud computing. Proceedings of the VLDB Endowment 7, 14 (2014), 1953–1964.

[46] Lookout. 2025. https://www.lookout.com. Accessed: 2025-07-11.

[47] Sujaya Maiyya, Seif Ibrahim, Caitlin Scarberry, Divyakant Agrawal, Amr El Abbadi, Huijia Lin, Stefano Tessaro, and Victor Zakhary. 2022. QuORAM: A Quorum-Replicated Fault Tolerant ORAM Datastore. In 31st USENIX Security Symposium. 3665–3682.

[48] 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 (SIGMOD) 1, 4 (2023), 1–25.

[49] Evangelia Anna Markatou, Francesca Falzon, Zachary Espiritu, and Roberto Tamassia. 2023. Attacks on encrypted response-hiding range search schemes in multiple dimensions. Proceedings on Privacy Enhancing Technologies (2023).

[50] Evangelia Anna Markatou and Roberto Tamassia. 2019. Full database reconstruction with access and search pattern leakage. In International Conference on Information Security. Springer, 25–43.

[51] Apostolos Mavrogiannakis, Xian Wang, Ioannis Demertzis, Dimitrios Papadopoulos, and Minos Garofalakis. 2025. OBLIVIATOR: Oblivious Parallel Joins and other Operators in Shared Memory Environments. Cryptology ePrint Archive (2025).

[52] Navajo Systems. 2025. https://tinyurl.com/yc4z5nyf. Accessed: 2025-07-11.

[53] Muhammad Naveed, Seny Kamara, and Charles V Wright. 2015. Inference attacks on property-preserving encrypted databases. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. 644–655.

[54] Simon Oya and Florian Kerschbaum. 2021. Hiding the Access Pattern is Not Enough: Exploiting Search Pattern Leakage in Searchable Encryption.. In USENIX Security Symposium. 127–142.

[55] Simon Oya and Florian Kerschbaum. 2022. IHOP: Improved Statistical Query Recovery against Searchable Symmetric Encryption through Quadratic Optimization. In 31st USENIX Security Symposium. 2407–2424.

[56] Sarvar Patel, Giuseppe Persiano, Kevin Yeo, and Moti Yung. 2019. Mitigating leakage in secure cloud-hosted data structures: Volume-hiding for multi-maps via hashing. In Proceedings of the 2019 ACM SIGSAC conference on computer and communications security. 79–93.

[57] Giuseppe Persiano and Kevin Yeo. 2019. Lower bounds for differentially private RAMs. In Advances in Cryptology–EUROCRYPT 2019: 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques. 404–434.

[58] Perspecsys. 2025. https://tinyurl.com/45ubwnef. Accessed: 2025-07-11.

[59] Rishabh Poddar, Stephanie Wang, Jianan Lu, and Raluca Ada Popa. 2020. Practical volume-based attacks on encrypted databases. In 2020 IEEE European Symposium on Security and Privacy (EuroS&P). IEEE, 354–369.

[60] Redis. 2025. https://redis.io/. Accessed: 2025-07-11.

[61] Leonie Reichert, Gowri R Chandran, Phillipp Schoppmann, Thomas Schneider, and Björn Scheuermann. 2024. Menhir: An Oblivious Database with Protection against Access and Volume Pattern Leakage. In Proceedings of the 19th ACM Asia Conference on Computer and Communications Security (Singapore, Singapore) (ASIA CCS '24). Association for Computing Machinery, New York, NY, USA, 1675–1690. https://doi.org/10.1145/3634737.3657005

[62] Cetin Sahin, Victor Zakhary, Amr El Abbadi, Huijia Lin, and Stefano Tessaro. 2016. TaoStore: Overcoming asynchronicity in oblivious data storage. In 2016 IEEE Symposium on Security and Privacy (SP). 198–217.

[63] Scalability in MySQL. 2025. https://www.mysql.com/products/cluster/scalability.html. Accessed: 2025-07-11.

[64] Amin Setayesh, Cheran Mahalingam, Emily Chen, and Sujaya Maiyya. 2025. Treebeard: A Scalable and Fault Tolerant ORAM Datastore. Cryptology ePrint Archive, Paper 2025/1082. https://eprint.iacr.org/2025/1082

[65] Skyhigh Networks. 2025. https://www.skyhighsecurity.com. Accessed: 2025-07-11.

[66] Emil Stefanov and Elaine Shi. 2013. Oblivistore: High performance oblivious cloud storage. In 2013 IEEE Symposium on Security and Privacy (SP). 253–267.

[67] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: an extremely simple oblivious RAM protocol. In Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS). 299–310.

[68] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. 2020. CockroachDB: The Resilient Geo-Distributed SQL Database. In Proceedings of the 2020 ACM International Conference on Management of Data (SIGMOD). 1493–1509.

[69] TiDB. 2025. https://www.pingcap.com. Accessed: 2025-07-11.

[70] Midhul Vuppalapati, Kushal Babel, Anurag Khandelwal, and Rachit Agarwal. 2022. SHORTSTACK: Distributed, Fault-tolerant, Oblivious Data Accesss. In 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI). 719–734.

[71] Jianfeng Wang, Shi-Feng Sun, Tianci Li, Saiyu Qi, and Xiaofeng Chen. 2022. Practical volume-hiding encrypted multi-maps with optimal overhead and beyond. In Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. 2825–2839.

[72] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. 2014. Oblivious Data Structures. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (Scottsdale, Arizona, USA) (CCS '14). Association for Computing Machinery, New York, NY, USA, 215–226. https://doi.org/10.1145/2660267.2660314

[73] YugaByte: Key-value data model. 2025. https://docs.yugabyte.com/preview/develop/data-modeling/common-patterns/keyvalue/. Accessed Feb 28, 2025.

[74] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. 2016. All your queries are belong to us: The power of file-injection attacks on searchable encryption.. In USENIX Security Symposium, Vol. 2016. 707–720.

[75] 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.