

OasisDB: An Oblivious and Scalable System for Relational Data

by

Haseeb Ahmed

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2025

© Haseeb Ahmed 2025

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

We present OasisDB, an oblivious and scalable RBDMS 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, an oblivious RDBMS, highlighting its advantages with scalability and multi-user support.

Acknowledgements

I would like to thank my advisor, Prof. Sujaya Maiyya, for her steadfast guidance, encouragement and support throughout the course of my degree. Her mentorship and support has been instrumental in shaping the direction and quality of this project.

I am also thankful to Nachiket Rao for his support during the development and evaluation stages of the project. His technical assistance and suggestions significantly contributed to the successful implementation of the system.

I would like to also extend my appreciate to Professor Florian Kerschbaum and his student Abdelkarim Kati for their constructive feedback, assistance and guidance.

Finally, I would like to thank my family, friends, and my wife Aima for their love, patience, and support throughout my graduate studies.

Table of Contents

Author’s Declaration	ii
Abstract	iii
Acknowledgements	iv
List of Figures	vii
List of Tables	viii
1 Introduction	1
1.1 Contributions and overview:	3
2 Background on oblivious key-value stores	5
2.1 PathORAM	5
2.2 Waffle	6
3 System and Threat Model	8
4 OasisDB Design	10
4.1 Data Representation	10
4.2 System Overview	12
4.3 Resolver Layer	13

4.4	Batcher Layer	14
4.5	Executor Layer	16
5	Oblivious Query Processing using OasisDB	19
5.1	Point Queries	19
5.2	Range Queries	20
5.3	Joins	21
5.4	Aggregates, Order By, and Group By	22
5.5	Updates	22
6	Security of OasisDB	24
7	Experimental Evaluation	28
7.1	Comparison with Baselines	31
7.2	System Scaling	33
7.3	Oblivious Query Processing	34
7.3.1	Join Queries	34
7.3.2	Range Queries	35
7.3.3	Performance under Skewed Workloads	36
8	Related work	38
9	Conclusion and Future Work	40
	References	41

List of Figures

3.1	OasisDB Architecture.	9
4.1	(a,b) Data representation in OasisDB. (c,d) Naively sharding an Oblivious Key-Value Store (OKVS) leads to challenges in query resolution.	11
4.2	Batcher intermixing and distributing key-value requests from multiple SQL and routing to two Executors in two batches $B1$ and $B2$ with a batch size $B_R = 2$	15
7.1	Throughput comparison for plaintext and oblivious baselines, along with horizontal scaling behavior of OasisDB using Waffle and PathORAM. . . .	29
7.2	Performance of OasisDB under two configurations: (a, b) show how throughput scales with increasing query layers, and (c, d) evaluate the performance of join queries using Waffle and PathORAM respectively.	32
7.3	Range query for the versions of OasisDB	36

List of Tables

1.1	Comparing OasisDB with other oblivious SQL query processing systems. .	2
7.1	Performance under Skewed and Uniform Distributions – Waffle KV Store .	37

Chapter 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 [86, 79, 89, 90, 40, 59, 78, 80, 96, 140, 151, 193, 141]. For example, performing an encrypted keyword search to retrieve encrypted documents can reveal the frequencies with which each keyword is searched. Mapping this access frequency distribution to the distribution of keywords commonly used in English can reveal with high probability the underlying plaintext keywords in the encrypted documents [86].

Oblivious data systems mitigate such attacks by obfuscating the access frequency distribution [170, 121, 77, 158, 56, 57, 63, 195, 41, 179, 119, 59]. Commonly deployed techniques for obliviousness include Oblivious RAM or ORAM [73], frequency smoothing [77, 98], and dynamic keyword shuffling [121]. 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 [170, 121, 77, 158, 56, 57, 41, 179, 119, 161], with many enabling scalability [168, 57, 179, 161]. However, many 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 oblivious relational data management systems (RDBMs)

	<i>Multi-user setting</i>	<i>Scalable</i>	<i>H/W agnostic</i>	<i>Supports updates</i>
AK [19]	✗	✗	✗	✗
Opaque [195]	✗	✗	✗	✗
ObliDB [63]	✗	✗	✗	✓ _(Complex)
SEAL [59]	✗	✓	✓	✗
OasisDB	✓	✓	✓	✓ _(Limited)

Table 1.1: Comparing OasisDB with other oblivious SQL query processing systems.

- Opaque [195], ObliDB [63], SEAL [59], and [19] by Arasu and Kaushik (AK). However, these systems’ support for a richer set of query type 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 1.1. (i). These systems process client queries *sequentially*, operating in a single-user setting. In contrast, practical plaintext RDBMSs handle concurrent client requests in a multi-user setting, leading to a substantial performance gap between plaintext and oblivious RDBMSs. (ii) Opaque [195], ObliDB [63], and [19] cannot scale¹ to accommodate growing number of client requests. While SEAL [59] enables horizontal scaling, due to a lack of support for multi-user settings, there is no throughput gains reported in [59]. Moreover, SEAL (tunably-) leaks the shard that serves a given client request with the leakage increasing as the scale factor grows. (iii) Opaque [195], ObliDB [63], and [19] require specialized hardware enclaves to execute oblivious relational queries, which may not be supported by all cloud providers. (iv). SEAL [59], Opaque [195], and [19] 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. In particular, OasisDB supports oblivious conditional selections, range queries, joins, aggregates, order bys, and (highly restricted) update SQL queries.

The first fundamental question we address in designing OasisDB is how to shard a

¹Opaque [195] supports executing a single request in parallel but not horizontal scaling.

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 YugaByteDB [188], Google’s Spanner [55], CockroachDB [52], and TiDB [175] 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 [195, 63, 19].

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. Or revealing the exact result volume of each SQL query makes the system vulnerable to volume pattern attacks. Meanwhile, naively handling updates in multi-user settings raises database correctness concerns because one user might be reading stale data that is actively being updated by another user.

1.1 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 [170], a widely used oblivious KV-store, and Waffle [121], a recent, highly efficient alternative. These KV-stores 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 (chapter 4).

2. Scalable multi-tier architecture: To achieve scalability in OasisDB, 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 [158, 77, 121, 56, 161, 179], 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 (chapter 4). The top-most layer, called *Executor*, interacts with the cloud storage servers and execute the oblivious data access mechanisms defined by PathORAM or Waffle. The middle layer, called *Batcher*, batches multiple requests from the bottom-most layer such that each *Executor* receives the uniform load. Maintaining this equal workload is crucial to preserving obliviousness, as we discuss in §4.4. The bottom-most layer, called *Resolver*, translates a client-issued SQL query into a series of keys and values to be retrieved from the storage servers. This design of OasisDB obfuscates SQL query result sizes, making volume pattern attacks computationally difficult to carry out (chapter 6).

3. Oblivious SQL processing: We leverage OasisDB’s design to demonstrate how to securely process a variety of SQL queries, including point and range queries, joins, aggregates, ordering, and (limited) updates (chapter 5).

4. Open-sourced prototype of OasisDB: Finally, we provide an open-sourced implementation of OasisDB integrated with an optimized PathORAM design [161] and Waffle [121]. Experimental evaluations on Epinions dataset and workload [27] provide empirical evidence that OasisDB scales linearly with the number of machines (chapter 7). We also compare OasisDB against CockroachDB [171], 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 an oblivious baseline, ObliDB [63] indicates the benefit of multi-user support and scalability of OasisDB.

Chapter 2

Background on oblivious key-value stores

This section provides a background on two oblivious key-value stores, PathORAM [170] and Waffle [121], which serve as sample alternative storage engines for developing OasisDB.

2.1 PathORAM

Path ORAM [170] 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 performs a read or write operation, PathORAM retrieves the entire path containing the object (or block), thereby obfuscating which specific block is being accessed. However, repeatedly accessing the same path whenever a specific object is requested can reveal the skew in the workload. PathORAM counters this by re-assigning any accessed 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, the proxy 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. The requested object is stored in a 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 guarantees under such a powerful adversary incurs a lower-bound of $O(\log N)$ bandwidth overhead [74, 104, 146]. This implies that serving a single client request will access $O(\log N)$ objects from the server.

2.2 Waffle

Waffle [121] is a recent oblivious KV-store designed to overcome the $\log N$ 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.

Most access pattern attacks exploit the frequency with which each outsourced encrypted object is accessed on the server. Waffle thwarts such 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. In the extreme case, some outsourced objects may never be requested by clients and hence remain unaccessed on the server forever.

Waffle mitigates the above leakage by guaranteeing that *every* outsourced key will be accessed within $\alpha + 1$ accesses after being written on the server. Waffle achieves this by batching real queries with fake queries wherein the objects for fake queries are picked based on the recency of their last access. Specifically, Waffle uses a binary search tree balanced on the last access timestamps of each object; the objects with the least timestamp, indicating that these are the least recently accessed objects, will be chosen for fake queries. 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. Please refer to the paper for more details [121]. In essence, Waffle ensures that for any sequence of objects requested by clients, the sequence of server accesses will be α, β -uniform, with the following α and β values:

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

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.

Chapter 3

System and Threat Model

System Model: OasisDB considers an architecture where an application outsources its data to the cloud and relies on a trusted proxy to route client requests. The proxy-architecture, which is well-adapted in academic [170, 56, 158, 168, 119, 121, 77, 179, 161] and production systems [25, 134, 110, 165, 147], 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 chapter 4).

Catered towards scalability, OasisDB partitions the state and functionality of the central trusted proxy across multiple layers as shown in Figure 3.1. 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.

In particular, the proxy consists of three layers: the top-most layer, called **Executor**, consists of the processes that execute oblivious accesses over key-value pairs (these are proxies as defined in systems such as PathORAM [170] and Waffle [121]). Each **Executor** process executes the oblivious access protocols for a disjoint set of key-value pairs without coordinating with other **Executors**. **Executors** are the only processes in OasisDB to communicate with the cloud storage. The middle layer, called **Batcher**, collects requests from the layer below and securely batches them before forwarding the requests to **Executors**. Each **Batcher** communicates with all **Executors**. Finally, the lowest-layer,

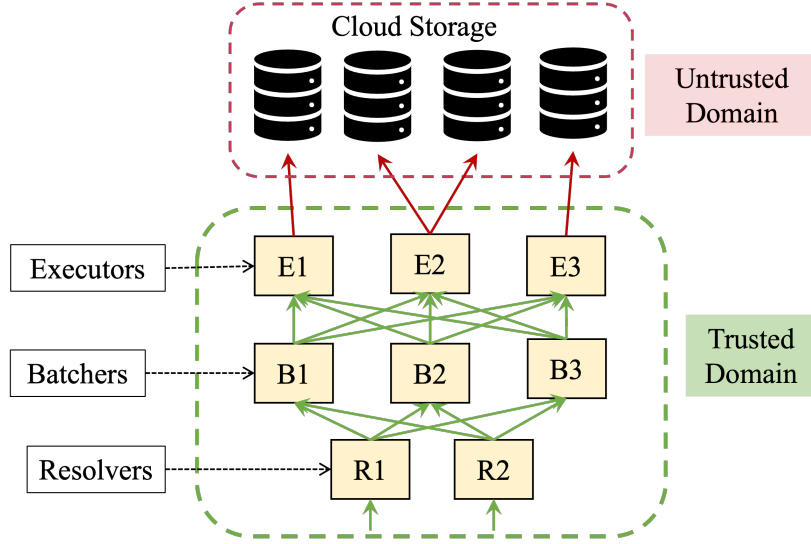


Figure 3.1: OasisDB Architecture.

called **Resolver**, resolves SQL queries issued by clients into key-value pairs that can be read from or updated in the cloud storage. For each client request, a **Resolver** randomly chooses a **Batcher** to request the key-value data.

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 [135, 97, 105, 89, 80, 151, 59, 78, 125, 91, 64, 65, 124] 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 application query workload or response times 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, as we discuss further in chapter 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 [170] and Waffle [121] as two alternatives, both of which have slightly different adversarial model as explained in chapter 2; OasisDB can accommodate both these adversarial models.

Chapter 4

OasisDB Design

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 [159] 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 [63, 195, 19].

Solution: Inspired by many plaintext databases supporting relational queries [171, 175, 55, 188], 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 $(T_{name}|C_{name}|pk : C_{value}^{pk})$ where T_{name} is the table name, C_{name} the column name, pk the primary key and C_{value}^{pk} that column's value for pk , as shown in Figure 4.1b. 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, especially when query selectivity¹ is low. In ORAM-based

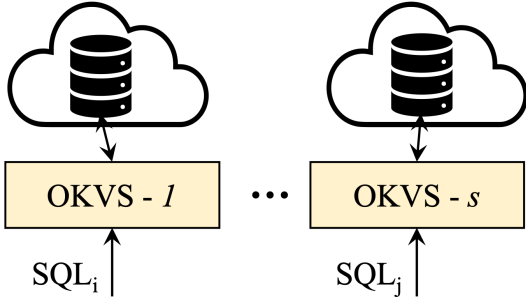
¹Selectivity here refers to the probability that a row satisfies a query's predicate.

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

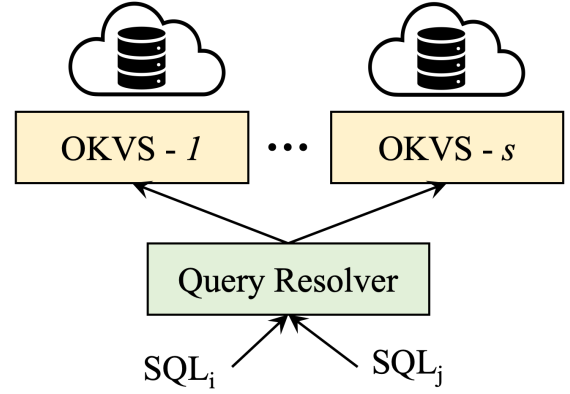
(a) Base table Emp

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) Key-value pairs for UID=100 and index on Salary



(c) Naive solution 1: Simply shard an OKVS



(d) Naive solution 2: Introduce a Query Resolver

Figure 4.1: (a,b) Data representation in OasisDB. (c,d) Naively sharding an Oblivious Key-Value Store (OKVS) leads to challenges in query resolution.

systems, where each request retrieves $O(\log N)$ objects, scanning a table of size $|T|$ results in a prohibitive bandwidth cost of $O(|T| * \log |T|)$.

Indexing the data while being oblivious poses overheads with client-side storage or bandwidth use. Storing an index structure 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 [44], which can severely reduce system throughput.

Solution: OasisDB addresses this challenge by creating inverted indices and outsourcing them to the cloud. Another benefit of representing data in key-value format is that inverted indices easily translate to key-value pairs, enabling OasisDB to retrieve index entries in one round. As seen in the example in Figure 4.1b, each index key has the format of $(T_{name}|C_{name-idx}|index_key)$ 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 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 an oblivious KV-store like PathORAM or Waffle, with each shard handling SQL queries (Figure 4.1c). 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 is to add a new component, a Query Resolver, that stores schema information, translates SQL queries to appropriate key-value requests, and routes them to the respective KV-shards, as shown in Figure 4.1d. 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. 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 different, leaking query-dependent information 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 [168, 30]. This introduces coordination between conflicting SQL queries requesting overlapping 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 [89, 80, 151, 59, 78, 125, 91, 64, 65, 124]. 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 3.1, its top-most 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. The simple yet effective design of the **Batcher** also protects against volume pattern attacks.

The following sections explains each layer of OasisDB.

4.3 Resolver Layer

The first layer of OasisDB consists of one or more **Resolver** processes that receive SQL requests from clients. This layer maintains query-dependent metadata essential for efficient

query execution, as we explain in chapter 5. For instance, it tracks which columns of a table have an index. While storing metadata as a file on the cloud and reading it for each request is a straightforward option, OasisDB instead keeps metadata locally on each **Resolver** to minimize WAN communication between the trusted domain and the cloud. 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.

Resolver transforms a SQL query into a set of key-value pairs to be retrieved (or updated) on the server with chapter 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 data keys while each entry in *values* either contains an update 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 sending the final response 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 request uneven number of keys from different shards. For example, in PathORAM, shards accessed by a repeated SQL query will (obviously) read more paths from their subtrees than others, violating the global obliviousness property of uniform accesses over the entire dataset. Additionally, retrieving the key-values requested per SQL query in isolation reveals the volume pattern of client requests. 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**,

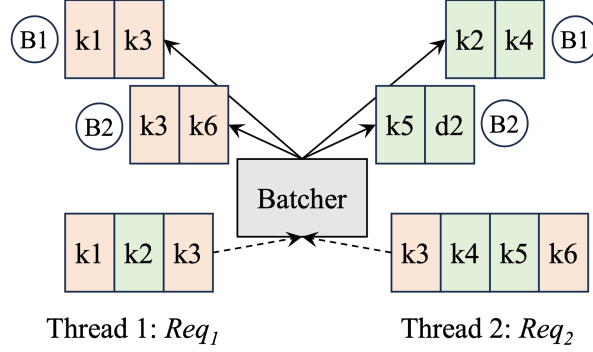


Figure 4.2: **Batcher** intermixing and distributing key-value requests from multiple SQL and routing to two **Executors** in two batches $B1$ and $B2$ with a batch size $B_R = 2$.

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 requests by inserting fake key-value requests (§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 due to multiple **Resolver** requests; the **Batcher** does not perform de-duplication and simply forwards all requests as they are to their respective **Executors**.

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 4.2 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-most layer of OasisDB, consisting of one or more **Executors**, is responsible for obliviously accessing key-value pairs from the untrusted storage server. 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 [170] and Waffle [121].

During initialization, OasisDB first 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 requests from a **Batcher**, it executes the oblivious protocol dictated by the chosen KV-store for its assigned subset of key-value pairs. For instance, a PathORAM-based **Executor** retrieves B_R random paths per batch of requests, while a Waffle-based **Executor** processes the batch by generating α, β -uniform accesses to the server. 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 [170] design processes requests sequentially, we optimize its throughput in our implementation by handling batched requests in parallel using techniques from [158, 161]. 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: Can any oblivious KV-store be seamlessly integrated into OasisDB? If not, what criteria must an oblivious KV scheme meet to ensure compatibility with OasisDB?

Let $\mathcal{O}$ be an obliviousness scheme that hides access patterns for a set of outsourced key-value pairs. Consider a sequence of m accesses, $A_m = \{(op_1, k_1, v_1), \dots (op_m, k_m, v_m)\}$ 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 $\mathcal{O}$ 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 a set of buckets along 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 chunks such that $D = (d_1 || d_2 \dots || d_s)$. Applying the oblivious scheme $\mathcal{O}$ 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 $\tau(d_1)_j, \tau(d_2)_j, \dots \tau(d_s)_j$. This implies that the overall transcript $\tau(D)$ is given by

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

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 $\mathcal{O}$ 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,*

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

The oblivious schemes of PathORAM and Waffle are *sharding-safe* when shard accesses are generated using the secure batching scheme $\mathcal{B}$. In PathORAM, partitioning a single tree into s smaller 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 $\mathbf{prop}'$, identical to $\mathbf{prop}$, over the entire dataset D . Similarly, in Waffle, applying its 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 [77] 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 using 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 [179]. 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 [179].

Chapter 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 adapt and modify the query processing logic within the **Resolver** layer without requiring changes to the other layers.

5.1 Point Queries

We begin with a basic SQL query on a single table containing one or more predicates in the **WHERE** clause. A **Resolver** upon receiving a SQL query first identifies the columns in the **WHERE** clause with an index. For all indexed columns, it generates a **Get** request to retrieve the corresponding index key-value pairs. In the example table shown in Figure 4.1b, index key for **salary** will query for $(employee|salar_idx|predicate_val)$. The request traverse through the **Batcher** and **Executor** layers to the server, which responds with the values for the index keys. Each value contains concatenated primary keys associated with the indexed value. The **Resolver** then identifies the overlapping primary keys across all retrieved keys and sends a **Get** request to fetch the selected 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 [87] and SEAL [59]. However, the scheme in [87] is non-oblivious and, therefore, susceptible to access pattern

attacks. SEAL [59] mitigates this leakage by employing ORAM. Nevertheless, SEAL’s tunable-leakage design can expose the sub-ORAM responsible for frequently queried tuples, as these sub-ORAMs may receive a higher number of requests compared to others, revealing access patterns. OasisDB protects from both types of leakages.

5.2 Range Queries

Next we consider range queries wherein one or more predicates in the **WHERE** clause of a SQL query consists of a range filter with a lower bound l and upper bound u .

Plaintext databases typically handle range queries efficiently by using indexing structures like B⁺-trees, which store information about the current state of the database. These structures are traversed to locate the leaf nodes corresponding to bounds l and/or u . However, traversing an outsourced B⁺-tree to find the desired range requires $\log N$ (i.e., the tree height) *rounds* of communication between the proxy and the cloud server [44], often over a WAN. Such communication delays significantly degrade system performance.

Alternatively, indexing mechanisms designed for range searches on encrypted data (as discussed in [60]) 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. This is because a range query expansion may include many point queries for keys that do not exist in the database instance.

OasisDB addresses this inefficiency by employing Bloom filters [32] 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 insert 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. If a range attribute lacks a Bloom filter but has an index, OasisDB handles the range query by converting it into a series of index-based point queries, potentially incurring additional compute and communication overhead for non-existent index keys. The benefit of Bloom filters is demonstrated in our experiments §7.3.

5.3 Joins

Join queries in SQL combine two (or more) tables based on a specified join condition. Join queries are of the form ‘`SELECT col_names FROM t1, t2 WHERE t1.attr = t2.attr`’ that joins tables `t1` and `t2` on a joinable attribute `attr`. If the join predicate consists of `=` condition, such joins are called equi- joins, and otherwise non-equi-joins. OasisDB supports both equi- and non-equi binary oblivious joins (two tables).

Existing schemes such as SEAL [59] support oblivious join queries 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 [87] 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. Alternatively, Chang et al.[45] propose using 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 (our benchmarking observed about 30MB join-map chapter 7).

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 in the Epinions dataset [27] used for benchmarking 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 requested 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.

If only one table, say **t1**, has a filter, OasisDB follows the same process but computes the cross-product of the filtered primary keys from **t1** with all primary keys from **t2** and checks them against the Bloom filter. To avoid storing primary keys of a table locally, OasisDB generates unique primary key ranges for each table and stores these ranges in **Resolvers**, while treating application-level primary keys as regular table columns. If no filter predicates exist for either table, a **Resolver** computes the cross-product of all primary keys from **t1** and **t2** locally and retrieves the requested columns for key pairs flagged as true by the Bloom filter.

Bloom filters return false positive 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, while simultaneously fetching the columns selected in the query. 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**, ordering of the results in ascending or descending order, and grouping of results based on specified attribute values. 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 [27]. Among existing oblivious data systems supporting relational queries [63, 195, 59], only ObliDB [63] supports updates by enforcing serial execution of queries, which significantly limits its throughput, as we show in §7.1.

Upon receiving an allowed update request, a **Resolver** generates a **Put** request to modify the specified key-value on the server. If the application uses an arbitrary primary

key while OasisDB has generated a custom primary key, the **Resolver** first retrieves the custom primary key via the index on the application’s primary key (e.g., **UID**). The retrieved custom primary key is then used to execute the update. However, OasisDB does not support updates of the read-modify-write format.

Serializing SQL Requests: To ensure the correct execution of read and write operations, OasisDB enforces serial execution of SQL queries. The constraints on update queries along with the linearizability of the underlying KV-store allow OasisDB to maintain this guarantee. 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 . The distributed design of OasisDB cannot trivially support complex update queries; extending it for such operations would require a transactional layer capable of ensuring distributed commits. Addressing this challenge is a promising avenue for future work.

Chapter 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 conditions and update values are encrypted, OasisDB ensures the following privacy guarantees: (i) it hides the exact rows satisfying query predicates for point, range, join, or update queries, (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 (Definition 1), which asserts that if each proxy shard (i.e., **Executor** in OasisDB) executing the oblivious protocol for a subset d_i of data upholds the scheme’s oblivious property **prop**, then the global property **prop’** holds for the entire dataset D . Both PathORAM [170] and Waffle [121] used in OasisDB satisfy this condition.

Theorem 1. *Let $U(m)$ be an independently, uniformly drawn random number in $[1, m]$. Given PathORAM [170], 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, data-dependent 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 a fixed number of key-value requests (B_R 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. $\square$

Theorem 2. *Given Waffle [121], a sharding-safe oblivious KV-store satisfying obliviousness property α, β -uniformity, OasisDB using Waffle satisfies the obliviousness property:*

$$\lfloor \alpha/B_R \rfloor (s-1)B_R + \alpha, \lceil \beta/B_R \rceil (s-1)B_R + \beta\text{-uniformity}.$$

Proof. Waffle defines α, β -uniformity (chapter 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. $\square$

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' . $\square$

Discussion on volume pattern leakage-abuse attacks: Leakage-abuse attacks can be categorized into three main variants: (i). reconstruction attacks, which aim to recover the exact values in a dataset; (ii). approximate reconstruction attacks, which seek an approximate mapping of the dataset’s values; (iii). count reconstruction attacks, which recover (approximately) how frequently each distinct value appears in the data. These attacks typically require a significant leakage profile - that is, the access pattern, the volume pattern, the equality pattern (also known as search pattern), and the structure pattern, which reveals if two queries have a common subquery - coupled with access to auxiliary information on the underlying data and or queries to enable meaningful recovery rates. 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 number of required queries or refine the attacker’s reconstruction process [96, 78, 79, 80].

Volume pattern range attacks [89, 80, 151, 59, 78, 125, 91, 64, 65, 124] – the most common type of volume attacks – specifically exploit the result sizes of SQL queries to infer either protected query predicates or the underlying data itself. These attacks often rely on assumptions of uniform query and/or dense data distributions, or the ability to isolate each query’s volume precisely. For multi-dimensional data, they may also require careful constructions of 1D “slices” (e.g., prime-sized or neighboring range queries), which are then aggregated to reconstruct value distributions. Several recent systems [59, 82, 156, 145, 180, 33] mitigate such attacks by per-query obfuscation of volumes or by frequency-smoothing techniques.

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 merging key-value requests from multiple SQL queries and generating uniform workload of exactly B_R requests per batch to the KV-stores. Consequently, while we do not claim to eliminate all forms of volume-based leakage, an attacker with access to OasisDB’s volume leakage cannot trivially correlate a single query with a specific volume, nor map these observed volumes to a known distribution, rendering volume pattern attacks significantly more challenging — if not computationally infeasible. To illustrate that, consider a domain of size N , which has $O(N^2)$ possible range queries. If m such queries are observed in parallel, an adversary would have to consider up to $(N^2)^m$ combinations. Even for moderate values $N = 10^5$ and $m = 25$, this combination space becomes 10^{250} , making an

exhaustive enumeration attack infeasible. In our deployment, we have over a million rows across five tables (and over six million key-value pairs), with a minimum client workload of 100 concurrent queries — an even larger scale that further exacerbates the infeasibility of launching a practical volume-based attack. We plan to evaluate the feasibility and effectiveness of volume pattern attacks on OasisDB-generated traces in future work.

Discussion on skewed workloads: When a skewed request distribution is introduced into our system, — so that a small number of popular keys or shards receive far more traffic than the rest — the client’s throughput and latency change in a predictable way, under uniform load, all executors share real work and dummy padding evenly, but as the skew grows, the popular executors are responsible for a disproportionate share of genuine operations, while the others largely process dummy requests. Because those few machines become the bottleneck, the client’s overall throughput falls and per-request latency climbs, even though it continues dispatching at a fixed rate across all nodes. To any external observer — including the untrusted cloud— each executor still appears to be equally busy, since dummy traffic now masks the imbalance in the skewed distribution, maintaining constant sized requests, being sent at the same time. The result is purely a performance degradation and can be observed at the client’s end, which in our system is within the trusted zone.

Chapter 7

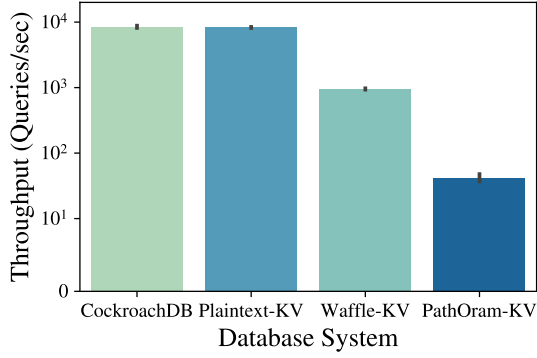
Experimental Evaluation

This section investigates the performance of OasisDB, integrated with Waffle [121] and PathORAM [170] as KV-stores, and compare both with other baselines across various query types. In particular, we answer the following questions:

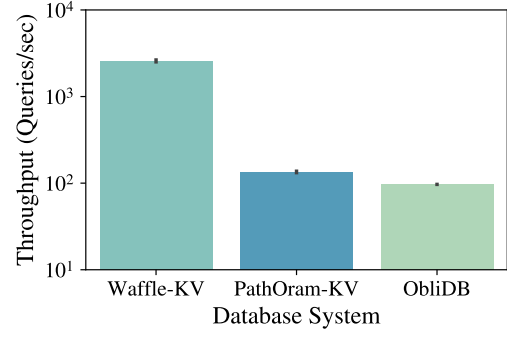
- How does OasisDB perform compared to insecure and oblivious baselines? (§7.1)
- How does OasisDB scale? Which components of the system contribute the most to its performance? (§7.2)
- How do OasisDB’s optimizations across different query types chapter 5 impact performance? (§7.3)

Implementation: We implement OasisDB in Go with $\sim 4k$ lines of code. The implementation is compatible with both GRPC and Thrift RPC libraries to interface with sharding-safe KV stores. Open-source implementation can be found on: <https://github.com/dsg-uwaterloo/ObliSQL>. For the two KV-stores, the experiments integrate the open-source implementation of Waffle [121], written in C++, and an optimized PathORAM implementation written in Go. Vanilla PathORAM [170] reads and writes tree paths on the server sequentially. Based on the optimization proposed in [161], our PathORAM implementation accesses a batch of requests in parallel such that any overlapping nodes of the tree across the batch is retrieved only once.

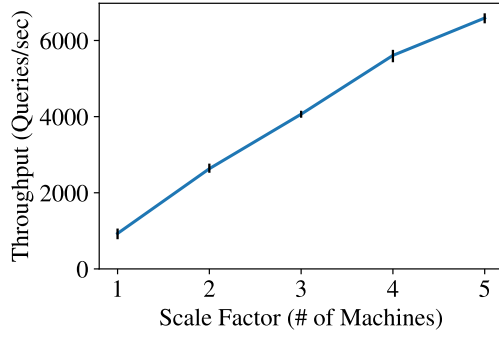
Experimental Setup: The experimental setup consists of machines with 2x Intel E5-2620v2 CPUs with 12 cores, 32 GB RAM, with 10 Gbps Ethernet connectivity. Unless



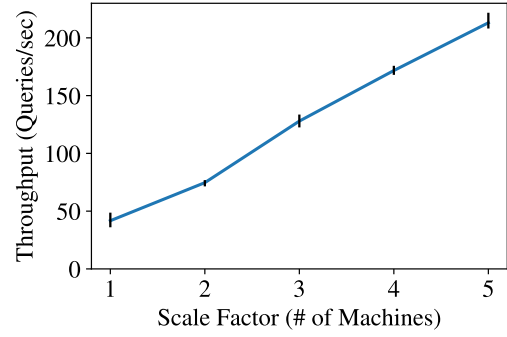
(a) Plaintext baseline comparison



(b) Oblivious baseline comparison



(c) Throughput scaling with Waffle



(d) Throughput scaling with PathORAM

Figure 7.1: Throughput comparison for plaintext and oblivious baselines, along with horizontal scaling behavior of OasisDB using Waffle and PathORAM.

otherwise noted, the experiments collocated the three layers of OasisDB on a single machine for a given scale factor. A multi-threaded client generates concurrent SQL queries mimicking the multi-user setting and measures the average latency and throughput for each experiment.

To emulate a cloud database server, we use a powerful high-performance machine with 4× Intel E5-4620v2 processors (32 Cores), 256GB RAM, and 10Gbps Ethernet connectivity. We use a commercial database, Redis [155], as the cloud storage engine and place all server shards on this machine. For simplicity, each **Executor** in our experiments connects to a single server shard. For example, with three **Executors**, the data on the server is partitioned into three shards, each mapped to one **Executor**. Furthermore, we emulate the WAN access link between the trusted domain and the cloud storage server by introducing a 10ms round-trip latency across the proxy and server machines.

For our SGX-based tests, we run enclaves on an Intel(R) Xeon(R) E-2374G CPU, which provides hardware support for both AVX2 and AVX-512 instruction sets, and features a default maximum Enclave Page Cache (EPC) size of 0.5GB for Intel SGX.

Baselines: We compare OasisDB with two baselines:

1. *CockroachDB* [171, 52] acts as an insecure baseline in evaluating the privacy overhead incurred by OasisDB. We choose CockroachDB as the baseline because its architecture resembles that of OasisDB of relying on a scalable KV storage engine to store relational data and execute SQL queries [53].

2. *ObliDB* [63] acts as an oblivious RDBMS baseline. ObliDB utilizes hardware enclaves (i.e., Intel SGX) to obviously process SQL queries on the server. It supports two storage modes: a tree-based structure for indexed queries and a flat structure for linear scan based queries. However, ObliDB assumes the presence of oblivious memory within the enclave — a functionality not provided by Intel SGX. Hence, the evaluations on ObliDB is without accounting for the associated costs of oblivious memory. It also lacks support for multi-user environments and processes queries sequentially, and does not scale across multiple machines. This limits its performance, particularly in WAN settings. However, ObliDB remains suitable for cloud-based deployments where on-premise resources are scarce and its limitations are acceptable.

Workloads and default parameters: We evaluate OasisDB and CockroachDB on Epinions dataset [27] 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 [61] 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. Each reported data point in the experiments is an average of three runs.

For comparison with ObliDB, we use the Big Data Benchmark [29], consisting of a table on PageRank information of multiple websites. This dataset size is 360K rows. This comparison experiment chooses a different baseline than the other experiments because evaluating ObliDB on a different dataset requires non-trivial changes. The workload for this baseline includes both point and range queries for OasisDB and ObliDB, constructed similarly to other experiments.

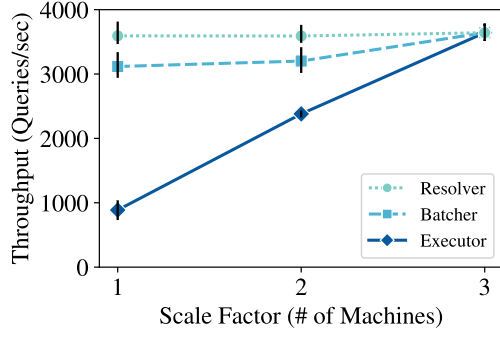
Regarding the configurable parameters for the KV-stores, for Waffle, the experiments set the system parameters such as the batch size and cache size such that the security parameter, α is 5000 [121] (the exact parameter values can be found in OasisDB’s Github repo). This implies that each of the 6 million outsourced key is read within 5000 server accesses after the key is written to the server. Whereas for PathORAM [170], the experiments set Z , the number of objects per bucket, to 4.

7.1 Comparison with Baselines

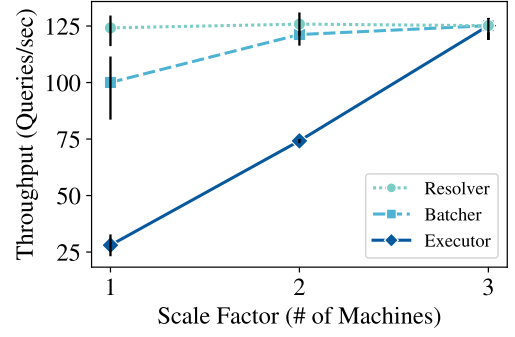
Comparison with CockroachDB: This experiment evaluates the cost of achieving obliviousness by comparing OasisDB with its a plaintext baseline, CockroachDB[171, 52]. We use BenchBase[61] to benchmark CockroachDB on the Epinions dataset [27] with an identical workload to OasisDB. In addition to the two 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 7.1a 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 118ms for OasisDB plaintext version. These results indicate that OasisDB’s **Resolver** and **Batcher** layers introduce minimal overhead.

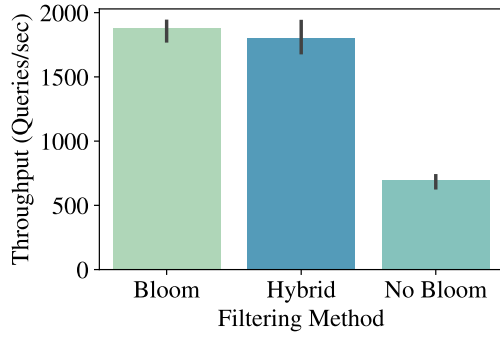
With Waffle as the KV-store in OasisDB, throughput decreases by **7.9x** compared to CockroachDB, while the average query latency rises to 931ms, which is **5.9x** higher



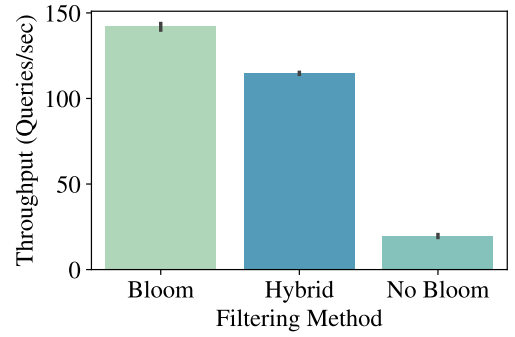
(a) Per-layer scaling with Waffle



(b) Per-layer scaling with PathORAM



(c) Join queries with Waffle



(d) Join queries with PathORAM

Figure 7.2: Performance of OasisDB under two configurations: (a, b) show how throughput scales with increasing query layers, and (c, d) evaluate the performance of join queries using Waffle and PathORAM respectively.

than CockroachDB. With PathORAM as the KV-store, we notice a larger degradation in performance relative to Waffle as the KV-store. This is due to the even greater overhead introduced by PathORAM due to its stronger threat model, which introduces a $\log N$ bandwidth overhead. This experiment highlights that the performance of OasisDB is primarily dictated by the choice of the underlying KV-store.

Comparison with ObliDB: This experiment compares OasisDB with ObliDB [63], an oblivious query processing system that leverages Intel SGX enclave for executing relational queries. We run ObliDB on Intel SGX V2 (Xeon E-2374G) with 128Gb RAM and introduce a 10ms round-trip WAN latency to simulate client-server communication, mirroring OasisDB’s setup. Since ObliDB is not scalable, OasisDB is deployed with a scale factor of 1 for a fair comparison.

As shown in Figure 7.1b, OasisDB achieves **26x** the throughput of ObliDB when using Waffle and **1.4x** the throughput when using PathORAM. The lower throughput of ObliDB stems from its sequential query processing, whereas OasisDB batches multiple requests to amortize communication and computation costs, improving throughput. However, batching increases query latency: ObliDB’s average latency is 10.33ms whereas OasisDB with Waffle has 380.3ms latency and with PathORAM has 362.6 ms.

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 13 types of SQL queries including point, range, join, aggregate, and update queries.

Figures 7.1c and 7.1d 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**.

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 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 7.2a and 7.2b illustrate that the **Executor** layer has the largest impact on performance for both Waffle and PathORAM with the throughput increasing by $\sim 3\times$ for both compared to scale 1. This is because **Executors** execute the expensive 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 negligible improvement of $\approx 20\%$ as the layer scales. From the insights of this experiment, an application to choose how many processes to deploy per layer depending on the required throughput.

7.3 Oblivious Query Processing

This section presents experiments demonstrating how OasisDB’s query processing optimizations enhance performance.

```
1 SELECT review.rating, item.title FROM review, item
2 WHERE item.i_id = review.i_id
3 AND item.i_id = ? AND review.creation_date = ?;
```

Query 7.1: Join Query.

```
1 SELECT rating FROM review WHERE u_id between ? and ?;
```

Query 7.2: Range Query

7.3.1 Join Queries

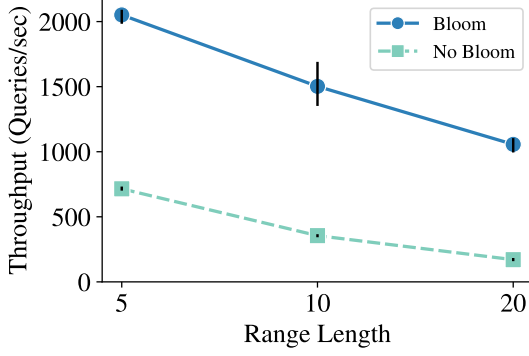
We begin by focusing on join queries and analyzing the effect of Bloom filters in OasisDB’s join processing. This experiment executes the join query in Query 7.1 with the client concurrently issuing the same join query but with varying predicates. We compare three filtering approaches:

- **Bloom:** This approach (i) retrieves primary keys for `item.i_id` and `review.creation_date` by querying the index keys. (ii) Uses a Bloom filter to identify potential joins from the cross product of these keys. (iii) For the keys that pass the filter, it fetches `review.rating` and `item.title` without join validation. Due to the Bloom Filter’s false positives, results may be incorrect. There are two rounds of communication – once for index keys and once for requested columns.
- **Hybrid:** This approach mirrors (i) and (ii) detailed above but adds validation in (iii). It retrieves both the selected columns and the join columns `item.i_id` and `review.i_id` in one round to verify true joins and returns valid results. Albeit retrieving more keys than the first approach, this approach also incurs two rounds of communication with the server and returns only correct results.
- **No Bloom:** This approach first retrieves filtered primary keys, using index keys. It then queries the join columns for all filtered primary keys for both tables. For the tuples that join, it retrieves the requested columns. This incurs three rounds of communication, reading more key-value pairs in total.

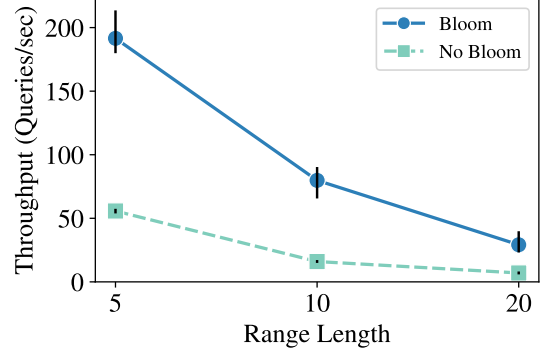
Figures 7.2c and 7.2d depict the results for Waffle and PathORAM. The Hybrid approach improves the throughput by 1.59x for Waffle and 5x for PathORAM compared to the No Bloom approach. Whereas the Hybrid approach incurs a penalty of 4.2% for Waffle and 19.7% for PathORAM compared to the Bloom approach in exchange for mitigating any false positive results. Bloom filter reduces the number of key-values retrieved from the server: the No Bloom approach retrieved 46 key-values per join query on average, the Hybrid averaged at 6.2 key-values per query, and the Bloom approach at 4.1 key-values. This reduction directly improves the throughput as seen in Figures 7.2c and 7.2d. PathORAM exhibits higher difference in throughput across the three approaches because each additional requested key-value incurs a logarithmic bandwidth and computational overhead compared to Waffle’s constant overheads.

7.3.2 Range Queries

This experiment evaluates the impact of Bloom filter on range queries to determine their effectiveness in preventing unnecessary key-value retrievals. Clients issue Query 7.2 concurrently while varies the length of the range query from 5 to 20 and measures the change in throughput for both the Bloom and No Bloom approaches. As the range increases, the number of keys retrieved per request grows, impacting the overall query execution time.



(a) Range queries - Waffle



(b) Range queries - PathORAM

Figure 7.3: Range query for the versions of OasisDB

Note that Bloom filter for range queries do not result in false positive results as explained in §5.2.

Figures 7.3a and 7.3b illustrate the performance changes for OasisDB with Waffle and PathORAM. The throughput drops as the range size increases. Since the Epinions dataset we use is uniformly distributed, as the range grows, fewer keys are filtered out by the Bloom filter, reducing the system throughput. And PathORAM has higher sensitivity to increased key-value retrievals, the throughput drops starkly for PathORAM as the range increases. Bloom filters on average improve the throughput by **3.3x** for both KV-stores.

7.3.3 Performance under Skewed Workloads

In this experiment, we benchmark our system under skew. We evaluate our system at a scale factor of 3, with 3,000 concurrent requests being by the client. Under the 0.99 Zipf distribution, average throughput falls to 3,824 QPS (verses 4,707 QPS), a drop of roughly 19%, while the average latency rises to 768 ms from 626 ms (a 23% increase), in comparison with a uniform (0 Zipf) distribution. High skew drives up the dummy-requests overhead: the system inserts fake keys at a rate of 7.11%, nearly double the 3.64% observed under uniform traffic. As we mentioned in Chapter 6 heavier skew concentrates real work on a subset of executors, lowering overall throughput, raising latency, and increasing the proportion of dummy operations. We expect to observe a similar trend in other KV stores under skewed workloads as well.

Skew	Throughput	Latency (ms)	Dummy Key Overhead
0.00	3 823.53	767.6	7.11 %
0.99	4 707.00	625.6	3.64 %

Table 7.1: Performance under Skewed and Uniform Distributions – Waffle KV Store

Chapter 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 [57, 179, 168, 161, 30]. 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 [168] and CURIOUS [30] cater towards sharding ORAM proxies for scalability but rely on a single load balancing process for security, which becomes a bottleneck. Snoopy [57] relies on TEEs for concealing access patterns and *linear scans the entire database per batch of requests*. ShortStack [179]’s scalability design is hard-wired to Pancake [77]’s oblivious scheme and does not generalize to other oblivious KV stores. Similarly, Treebeard [161]’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 [106, 45, 44, 93] 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 [156, 19, 195, 63, 128] rely on TEEs. Menhir[156] 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 [63], Opaque [195], Arasu and Kaushik’s [19] and a concurrent work, Oblivator [128] 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 [59] 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 [182] 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.

Chapter 9

Conclusion and Future Work

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, including point and range queries, joins, aggregations, and limited updates, 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 an ObliDB, an oblivious RDBMS, as a baseline, OasisDB exhibits benefits due to scalability and multi-user support.

Future work on OasisDB will focus on support for OLTP workloads with serializability guarantees and expanding query types to nearest neighbor searches. Additional directions include integrating transactional semantics to support ACID operations and incorporating fault-tolerance mechanisms such as replication and failure recovery to improve reliability in the distributed setting. Moreover, extending SQL support to encompass more expressive operators—such as the LIKE clause for pattern matching—and complex sub queries would further enhance the framework’s applicability for more diverse relational workloads.

References

- [1] 2021-22 Survey of Canadian businesses on privacy-related issues. https://www.priv.gc.ca/en/opc-actions-and-decisions/research/explore-privacy-research/2022/por_2021-22_bus/#fig03. Accessed January 18, 2023.
- [2] Cloud storage of corporate data. <https://www.statista.com/statistics/1062879/worldwide-cloud-storage-of-corporate-data/>. Accessed January 18, 2023.
- [3] Default Replica Count For CockroachDB. <https://www.cockroachlabs.com/docs/stable/configure-replication-zones.html>. Accessed Jan 10, 2021.
- [4] Default Replica Count For Spanner. <https://cloud.google.com/spanner/docs/instances>. Accessed Jan 10, 2021.
- [5] Growth of Global Data Privacy Laws. <https://stealthbits.com/blog/growth-of-global-data-privacy-laws/>. Accessed: 2021-10-25.
- [6] High stakes of privacy loss for marginalized communities. <https://www.newamerica.org/oti/reports/centering-civil-rights-privacy-debate/for-marginalized-communities-the-stakes-are-high/>. Accessed Jan 10, 2021.
- [7] Increasing Cyber-attacks on Cloud Services. <https://compliance-group.com/cyber-attacks-on-cloud-services-rise-630/>. Accessed: 2021-10-25.
- [8] Netflix uses AWS for all compute and storage needs. <https://aws.amazon.com/solutions/case-studies/netflix/>. Accessed October 5, 2021.

- [9] Spotify backend infrastructure moves to Google Cloud. <https://variety.com/2016/digital/news/spotify-goes-cloud-no-more-data-centers-1201712891/>. Accessed October 5, 2021.
- [10] Statistics of cloud computing. <https://www.cloudzero.com/blog/cloud-computing-statistics>. Accessed January 18, 2023.
- [11] Twitter selects AWS to power user feeds. <https://press.aboutamazon.com/news-releases/news-release-details/twitter-selects-aws-strategic-provider-serve-timelines/>. Accessed October 5, 2021.
- [12] Nahla Aburawi, Alexei Lisitsa, and Frans Coenen. Querying encrypted graph databases. In Proceedings of the 4th International Conference on Information Systems Security and Privacy. SCITEPRESS-Science and Technology Publications, 2018.
- [13] Divyakant Agrawal and Amr El Abbadi. An efficient and fault-tolerant solution for distributed mutual exclusion. ACM Transactions on Computer Systems (TOCS), 9(1):1–20, 1991.
- [14] Divyakant Agrawal and Amr El Abbadi. The generalized tree quorum protocol: An efficient approach for managing replicated data. ACM Transactions on Database Systems (TODS), 17(4):689–717, 1992.
- [15] Marcos K Aguilera and Douglas B Terry. The many faces of consistency. IEEE Data Eng. Bull., 39(1):3–13, 2016.
- [16] MJ. Amiri, Sujaya Maiyya, D. Agrawal, and A. El Abbadi. Seemore: A fault-tolerant protocol for hybrid cloud environments. International Conference on Data Engineering (ICDE), 2020.
- [17] MJ. Amiri, D. Shu, Sujaya Maiyya, D. Agrawal, and A. El Abbadi. Ziziphus: Scalable Data Management Across Byzantine Edge Servers. https://sites.cs.ucsb.edu/~sujaya_maiyya/assets/papers/ziziphus.pdf, 2021. Accessed: 2021-11-11.
- [18] HIMSS Analytics. Himss analytics cloud survey, 2014.
- [19] Arvind Arasu and Raghav Kaushik. Oblivious query processing. arXiv preprint arXiv:1312.4012, 2013.

- [20] V. Arora, RKS. Babu, Sujaya Maiyya, D. Agrawal, A. El Abbadi, X. Xue, et al. Dynamic timestamp allocation for reducing transaction aborts. In 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), pages 269–276. IEEE, 2018.
- [21] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. OptORAMa: Optimal oblivious RAM. J. ACM, 70(1):1–70, 2022.
- [22] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Enoch Peserico, and Elaine Shi. Optimal oblivious parallel RAM. In Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pages 2459–2521, 2022.
- [23] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, and Elaine Shi. Oblivious RAM with worst-case logarithmic overhead. J. Cryptology, 36(2):7, 2023.
- [24] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems, pages 53–64, 2012.
- [25] Baffle. <https://baffle.io>.
- [26] Johes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel N Kho, and Jennie Rogers. Smcql: Secure query processing for private data networks. Proc. VLDB Endow., 10(6):673–684, 2017.
- [27] Benchbase Epinions Dataset. <https://github.com/cmu-db/benchbase/wiki/epinions>.
- [28] Ryan Berryhill, Wojciech Golab, and Mahesh Tripunitara. Robust shared objects for non-volatile main memory. In LIPIcs-Leibniz International Proceedings in Informatics, volume 46. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.
- [29] Big Data Benchmark. <https://amplab.cs.berkeley.edu/benchmark/>.
- [30] Vincent Bindschaedler, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, and Yan Huang. 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), pages 837–849, 2015.

- [31] Marina Blanton, Aaron Steele, and Mehrdad Alisagari. Data-oblivious graph algorithms for secure computation and outsourcing. In Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security, pages 207–218, 2013.
- [32] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. Communications of the ACM, 13(7):422–426, 1970.
- [33] Dmytro Bogatov, Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O’Neill. ϵ psolute: Efficiently querying databases while providing differential privacy. In Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, pages 2262–2276, 2021.
- [34] Dan Boneh, Eyal Kushilevitz, Rafail Ostrovsky, and William E Skeith. Public key encryption that allows pir queries. In Annual International Cryptology Conference, pages 50–67. Springer, 2007.
- [35] Elette Boyle, Kai-Min Chung, and Rafael Pass. Oblivious parallel RAM and applications. In Theory of Cryptography: 13th International Conference (TCC), pages 175–204, 2016.
- [36] Elette Boyle and Moni Naor. Is there an oblivious ram lower bound? In Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science, pages 357–368, 2016.
- [37] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. Tao: Facebook’s distributed data store for the social graph. In 2013 USENIX Annual Technical Conference (USENIXATC 13), pages 49–60, 2013.
- [38] Ning Cao, Zhenyu Yang, Cong Wang, Kui Ren, and Wenjing Lou. Privacy-preserving query over encrypted graph-structured data in cloud computing. In 2011 31st International Conference on Distributed Computing Systems, pages 393–402. IEEE, 2011.
- [39] David Cash, Andrew Drucker, and Alexander Hoover. A lower bound for one-round oblivious ram. In Theory of Cryptography: 18th International Conference, TCC 2020, Durham, NC, USA, November 16–19, 2020, Proceedings, Part I 18, pages 457–485. Springer, 2020.

- [40] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS), pages 668–679, 2015.
- [41] Anrin Chakraborti and Radu Sion. ConcurORAM: High-throughput stateless parallel multi-client ORAM. In 26th Annual Network and Distributed System Security Symposium (NDSS), 2019.
- [42] T-H Hubert Chan, Yue Guo, Wei-Kai Lin, and Elaine Shi. Oblivious hashing revisited, and applications to asymptotically efficient ORAM and OPRAM. In Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, pages 660–690, 2017.
- [43] T.-H. Hubert Chan and Elaine Shi. Circuit OPRAM: unifying statistically and computationally secure ORAMs and OPRAMs. In Theory of Cryptography: 15th International Conference (TCC), pages 72–107, 2017.
- [44] Zhao Chang, Dong Xie, Feifei Li, Jeff M Phillips, and Rajeev Balasubramonian. Efficient oblivious query processing for range and knn queries. IEEE Transactions on Knowledge and Data Engineering, 34(12):5741–5754, 2021.
- [45] Zhao Chang, Dong Xie, Sheng Wang, and Feifei Li. Towards practical oblivious join. In Proceedings of the 2022 International Conference on Management of Data, pages 803–817, 2022.
- [46] Melissa Chase and Seny Kamara. Structured encryption and controlled disclosure. In International conference on the theory and application of cryptology and information security, pages 577–594. Springer, 2010.
- [47] Binyi Chen, Huijia Lin, and Stefano Tessaro. Oblivious parallel RAM: improved efficiency and generic constructions. In Theory of Cryptography: 13th International Conference (TCC), pages 205–234, 2016.
- [48] Shun Yan Cheung, Mostafa H Ammar, and Mustaque Ahamad. The grid protocol: A high performance scheme for maintaining replicated data. IEEE Transactions on Knowledge and Data Engineering, 4(6):582–592, 1992.
- [49] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. Journal of the ACM (JACM), 45(6):965–981, 1998.

- [50] Kai-Min Chung, Zhenming Liu, and Rafael Pass. Statistically-secure oram with $\tilde{O}(\log^2 n)$ overhead. In Advances in Cryptology–ASIACRYPT 2014: 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, ROC, December 7-11, 2014, Proceedings, Part II 20, pages 62–81. Springer, 2014.
- [51] Cloud Adoption Statistics. <https://bit.ly/3ZdCzpt>. Accessed Feb 10, 2023.
- [52] CockroachDB. <https://www.cockroachlabs.com/blog/pebble-rocksdb-kv-store/>.
- [53] CockroachDB - KV store layer. <https://www.cockroachlabs.com/glossary/distributed-db/key-value-kv-layer/>.
- [54] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC), pages 143–154, 2010.
- [55] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google’s globally distributed database. ACM Transactions on Computer Systems (TOCS), 31(3):1–22, 2013.
- [56] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. Obladi: Oblivious serializable transactions in the cloud. In 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI), pages 727–743, 2018.
- [57] Emma Dauterman, Vivian Fang, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. Snoopy: Surpassing the scalability bottleneck of oblivious storage. In Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP), pages 655–671, 2021.
- [58] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Voshall, and Werner Vogels. Dynamo: amazon’s highly available key-value store. ACM SIGOPS operating systems review, 41(6):205–220, 2007.
- [59] Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Saurabh Shintre. Seal: Attack mitigation for encrypted databases via adjustable

- leakage. In 29th USENIX Security Symposium (USENIX Security 20), pages 2433–2450, 2020.
- [60] Ioannis Demertzis, Stavros Papadopoulos, Odysseas Papapetrou, Antonios Deligianakis, Minos Garofalakis, and Charalampos Papamanthou. Practical private range search in depth. ACM Transactions on Database Systems (TODS), 43(1):1–52, 2018.
 - [61] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. PVLDB, 7(4):277–288, 2013.
 - [62] Jack Doerner and Abhi Shelat. Scaling oram for secure computation. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pages 523–535, 2017.
 - [63] Saba Eskandarian and Matei Zaharia. Oblidb: Oblivious query processing for secure databases. Proc. VLDB Endow., 13(2):169–183, 2019.
 - [64] Sky Faber, Stanislaw Jarecki, Hugo Krawczyk, Quan Nguyen, Marcel Rosu, and Michael Steiner. 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, pages 123–145. Springer, 2015.
 - [65] Francesca Falzon, Evangelia Anna Markatou, Zachary Espiritu, and Roberto Tamassia. Range search over encrypted multi-attribute data. Cryptology ePrint Archive, 2022.
 - [66] Francesca Falzon and Kenneth G Paterson. An efficient query recovery attack against a graph encryption scheme. Cryptology ePrint Archive, 2022.
 - [67] Xinrui Ge, Jia Yu, Hanlin Zhang, Jianli Bai, Jianxi Fan, and Neal N Xiong. Spps: A search pattern privacy system for approximate shortest distance query of encrypted graphs in iiot. IEEE Transactions on Systems, Man, and Cybernetics: Systems, 52(1):136–150, 2021.
 - [68] Craig Gentry, Kenny A Goldman, Shai Halevi, Charanjit Julta, Mariana Raykova, and Daniel Wichs. Optimizing oram and using it efficiently for secure computation. In Privacy Enhancing Technologies: 13th International Symposium, PETS 2013, Bloomington, IN, USA, July 10-12, 2013. Proceedings 13, pages 1–18. Springer, 2013.

- [69] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In Proceedings of the nineteenth ACM symposium on Operating systems principles, pages 29–43, 2003.
- [70] Esha Ghosh, Seny Kamara, and Roberto Tamassia. Efficient graph encryption scheme for shortest path queries. In Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security, pages 516–525, 2021.
- [71] David K Gifford. Weighted voting for replicated data. In Proceedings of the seventh ACM symposium on Operating systems principles, pages 150–162. ACM, 1979.
- [72] Anselme Goetschmann. Design and analysis of graph encryption schemes. Master’s thesis, ETH Zurich, 2020.
- [73] Oded Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In Proceedings of the 19th Annual ACM Symposium on Theory of Computing (STOC), pages 182–194, 1987.
- [74] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. J. ACM, 43(3):431–473, 1996.
- [75] Michael T Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious ram simulation. In Automata, Languages and Programming: 38th International Colloquium, ICALP 2011, Zurich, Switzerland, July 4-8, 2011, Proceedings, Part II 38, pages 576–587. Springer, 2011.
- [76] James N Gray. Notes on data base operating systems. Operating systems, pages 393–481, 1978.
- [77] Paul Grubbs, Anurag Khandelwal, Marie-Sarah Lacharité, Lloyd Brown, Lucy Li, Rachit Agarwal, and Thomas Ristenpart. Pancake: Frequency smoothing for encrypted data stores. In 29th USENIX Security Symposium, pages 2451–2468, 2020.
- [78] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. 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, pages 315–331, 2018.
- [79] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. Learning to reconstruct: Statistical learning theory and encrypted database attacks. In 2019 IEEE Symposium on Security and Privacy (SP), pages 1067–1083, 2019.

- [80] Zichen Gui, Oliver Johnson, and Bogdan Warinschi. Encrypted databases: New volume attacks against range queries. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, pages 361–378, 2019.
- [81] Peeyush Gupta, Yin Li, Sharad Mehrotra, Nisha Panwar, Shantanu Sharma, and Sumaya Almanee. Obscure: Information-theoretically secure, oblivious, and verifiable aggregation queries on secret-shared outsourced data. IEEE Transactions on Knowledge and Data Engineering (TKDE), 34(2):843–864, 2022.
- [82] Shanshan Han, Vishal Chakraborty, Michael T Goodrich, Sharad Mehrotra, and Shantanu Sharma. Veil: A storage and communication efficient volume-hiding algorithm. Proceedings of the ACM on Management of Data, 1(4):1–27, 2023.
- [83] Hashicorp Raft. <https://github.com/hashicorp/raft>.
- [84] Health Care Cloud Computing Trends. <https://bit.ly/3ZyCWdI>. Accessed Feb 10, 2023.
- [85] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems (TOPLAS), 12(3):463–492, 1990.
- [86] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: ramification, attack and mitigation. In 19th Annual Network and Distributed System Security Symposium (NDSS), 2012.
- [87] Seny Kamara and Tarik Moataz. 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, pages 149–180. Springer, 2018.
- [88] Antonios Katsarakis, Vasilis Gavrielatos, MR Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay Nagarajan. Hermes: a fast, fault-tolerant and linearizable replication protocol. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 201–217, 2020.
- [89] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O’neill. Generic attacks on secure outsourced databases. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS), pages 1329–1340, 2016.

- [90] Evgenios M Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. Data recovery on encrypted databases with k-nearest neighbor query leakage. In 2019 IEEE Symposium on Security and Privacy (SP), pages 1033–1050, 2019.
- [91] Evgenios M Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. Response-hiding encrypted ranges: Revisiting security via parametrized leakage-abuse attacks. In 2021 IEEE Symposium on Security and Privacy (SP), pages 1502–1519. IEEE, 2021.
- [92] Tim Kraska, Gene Pang, Michael J Franklin, Samuel Madden, and Alan Fekete. Mdcc: Multi-data center consistency. In Proceedings of the 8th ACM European Conference on Computer Systems, pages 113–126, 2013.
- [93] Simeon Krastnikov, Florian Kerschbaum, and Douglas Stebila. Efficient oblivious database joins. arXiv preprint arXiv:2003.09481, 2020.
- [94] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in) security of hash-based oblivious ram and a new balancing scheme. In Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms, pages 143–156. SIAM, 2012.
- [95] Eyal Kushilevitz and Rafail Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In Proceedings 38th annual symposium on foundations of computer science, pages 364–373. IEEE, 1997.
- [96] Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. Improved reconstruction attacks on encrypted data using range query leakage. In 2018 IEEE Symposium on Security and Privacy (SP), pages 297–314. IEEE, 2018.
- [97] Marie-Sarah Lacharité and Kenneth G Paterson. A note on the optimality of frequency analysis vs. ℓ_p -optimization. Cryptology ePrint Archive, 2015.
- [98] Marie-Sarah Lacharité and Kenneth G Paterson. Frequency-smoothing encryption: preventing snapshot attacks on deterministically encrypted data. Cryptology ePrint Archive, 2017.
- [99] Shangqi Lai, Xingliang Yuan, Shi-Feng Sun, Joseph K Liu, Yuhong Liu, and Dongxi Liu. Graphse²: An encrypted graph database for privacy-preserving social search. In Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security, pages 41–54, 2019.

- [100] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. Commun. ACM, 21(7):558–565, 1978.
- [101] Leslie Lamport. The part-time parliament. In Transactions on Computer Systems, pages 133–169. ACM, 1998.
- [102] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. ACM Transactions on Programming Languages and Systems (TOPLAS), 4(3):382–401, 1982.
- [103] Kasper Green Larsen, Tal Malkin, Omri Weinstein, and Kevin Yeo. Lower bounds for oblivious near-neighbor search. In Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, pages 1116–1134. SIAM, 2020.
- [104] Kasper Green Larsen and Jesper Buus Nielsen. Yes, there is an oblivious RAM lower bound! In Advances in Cryptology–CRYPTO 2018: 38th Annual International Cryptology Conference, pages 523–542, 2018.
- [105] Jingwei Li, Chuan Qin, Patrick PC Lee, and Xiaosong Zhang. Information leakage in encrypted deduplication via frequency analysis. In 2017 47th Annual IEEE/IFIP international conference on dependable systems and networks (DSN), pages 1–12. IEEE, 2017.
- [106] Rui Li, Alex X Liu, Ann L Wang, and Bezawada Bruhadeshwar. Fast range query processing with strong privacy protection for cloud computing. Proceedings of the VLDB Endowment, 7(14):1953–1964, 2014.
- [107] Helger Lipmaa and Bingsheng Zhang. Two new efficient pir-writing protocols. In International Conference on Applied Cryptography and Network Security, pages 438–455. Springer, 2010.
- [108] Gang Liu, Kenli Li, Zheng Xiao, and Rujia Wang. Ehap-oram: Efficient hardware-assisted persistent oram system for non-volatile memory. arXiv preprint arXiv:2011.03669, 2020.
- [109] Zheli Liu, Bo Li, Yanyu Huang, Jin Li, Yang Xiang, and Witold Pedrycz. Newmcos: towards a practical multi-cloud oblivious storage scheme. IEEE Transactions on Knowledge and Data Engineering (TKDE), 32(4):714–727, 2019.
- [110] Lookout. <https://www.lookout.com>.

- [111] Jacob R. Lorch, Bryan Parno, James Mickens, Mariana Raykova, and Joshua Schiffman. Shroud: Ensuring private access to Large-Scale data in the data center. In 11th USENIX Conference on File and Storage Technologies (FAST 13), pages 199–213, San Jose, CA, February 2013. USENIX Association.
- [112] Steve Lu and Rafail Ostrovsky. Distributed oblivious ram for secure two-party computation. In Theory of Cryptography Conference (TCC), pages 377–396. Springer, 2013.
- [113] N. A. Lynch and A. A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS '97), FTCS '97, pages 272–, Washington, DC, USA, 1997. IEEE Computer Society.
- [114] Mamoru Maekawa. An algorithm for mutual exclusion in decentralized systems. ACM Transactions on Computer Systems (TOCS), 3(2):145–159, 1985.
- [115] Hatem Mahmoud, Faisal Nawab, Alexander Pucher, Divyakant Agrawal, and Amr El Abbadi. Low-latency multi-datacenter databases using replicated commit. Proceedings of the VLDB Endowment, 6(9):661–672, 2013.
- [116] Sujaya Maiyya, I. Ahmad, D. Agrawal, and A. El Abbadi. Samya: Geo-distributed data system for high contention data aggregates. International Conference on Data Engineering (ICDE), 2021.
- [117] Sujaya Maiyya, P. Ananth, D. Agrawal, and A. El Abbadi. ORTOA: One Round Trip Oblivious Access. https://sites.cs.ucsb.edu/~sujaya_maiyya/assets/papers/ORTOA.pdf. Accessed: 2021-11-11.
- [118] Sujaya Maiyya, DBH. Cho, D. Agrawal, and A. El Abbadi. Fides: Managing data on untrusted infrastructure. International Conference on Distributed Computing Systems (ICDCS), 2020.
- [119] Sujaya Maiyya, Seif Ibrahim, Caitlin Scarberry, Divyakant Agrawal, Amr El Abbadi, Huijia Lin, Stefano Tessaro, and Victor Zakhary. QuORAM: A quorum-replicated fault tolerant ORAM datastore. In 31st USENIX Security Symposium, pages 3665–3682, 2022.
- [120] Sujaya Maiyya, F. Nawab, D. Agrawal, and A. El Abbadi. Unifying consensus and atomic commitment for effective cloud data management. Proceedings of the VLDB Endowment, 12(5):611–623, 2019.

- [121] Sujaya Maiyya, Sharath Chandra Vemula, Divyakant Agrawal, Amr El Abbadi, and Florian Kerschbaum. Waffle: An online oblivious datastore for protecting data access patterns. Proceedings of the ACM on Management of Data (SIGMOD), 1(4):1–25, 2023.
- [122] Sujaya Maiyya, V. Zakhary, D. Agrawal, and A. El Abbadi. Database and distributed computing fundamentals for scalable, fault-tolerant, and consistent maintenance of blockchains. Proceedings of the VLDB Endowment, 11(12):2098–2101, 2018.
- [123] Sujaya Maiyya, V. Zakhary, MJ. Amiri, D. Agrawal, and A. El Abbadi. Database and distributed computing foundations of blockchains. In SIGMOD, 2019.
- [124] Evangelia Anna Markatou, Francesca Falzon, Zachary Espiritu, and Roberto Tamassia. Attacks on encrypted response-hiding range search schemes in multiple dimensions. Proceedings on Privacy Enhancing Technologies, 2023.
- [125] Evangelia Anna Markatou and Roberto Tamassia. Full database reconstruction with access and search pattern leakage. In International Conference on Information Security, pages 25–43. Springer, 2019.
- [126] Jean-Philippe Martin, Lorenzo Alvisi, and Michael Dahlin. Minimal byzantine storage. In Proceedings of the 16th International Conference on Distributed Computing, DISC '02, pages 311–325, London, UK, UK, 2002. Springer-Verlag.
- [127] Charalampos Mavroforakis, Nathan Chenette, Adam O’Neill, George Kollios, and Ran Canetti. Modular order-preserving encryption, revisited. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pages 763–777, 2015.
- [128] Apostolos Mavrogiannakis, Xian Wang, Ioannis Demertzis, Dimitrios Papadopoulos, and Minos Garofalakis. Oblivator: Oblivious parallel joins and other operators in shared memory environments. Cryptology ePrint Archive, 2025.
- [129] Sharad Mehrotra, Shantanu Sharma, Jeffrey D. Ullman, Dhrubajyoti Ghosh, Peeyush Gupta, and Anurag Mishra. PANDA: partitioned data security on outsourced sensitive and non-sensitive data. ACM Trans. Manag. Inf. Syst., 11(4):23:1–23:41, 2020.
- [130] Xianrui Meng, Seny Kamara, Kobbi Nissim, and George Kollios. GreCs: Graph encryption for approximate shortest distance queries. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pages 504–517, 2015.

- [131] Microsoft Azure. <https://azure.microsoft.com/en-ca>.
- [132] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. Consolidating concurrency control and consensus for commits under conflicts. In 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), pages 517–532, 2016.
- [133] Moni Naor and Udi Wieder. Scalable and dynamic quorum systems. Distributed Computing, 17(4):311–322, 2005.
- [134] Navajo Systems. <https://tinyurl.com/yc4z5nyf>.
- [135] Muhammad Naveed, Seny Kamara, and Charles V Wright. Inference attacks on property-preserving encrypted databases. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pages 644–655, 2015.
- [136] Kartik Nayak and Jonathan Katz. An oblivious parallel RAM with $O(\log^2 N)$ parallel runtime blowup. Cryptology ePrint Archive, page 1141, 2016.
- [137] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. Graphsc: Parallel secure computation made easy. In 2015 IEEE Symposium on Security and Privacy, pages 377–394. IEEE, 2015.
- [138] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In 2014 USENIX Annual Technical Conference (Usenix ATC 14), pages 305–319, 2014.
- [139] R. Ostrovsky. Efficient computation on oblivious RAMs. In Proceedings of the 22nd Annual ACM Symposium on Theory of Computing (STOC), pages 514–523, 1990.
- [140] Simon Oya and Florian Kerschbaum. Hiding the access pattern is not enough: Exploiting search pattern leakage in searchable encryption. In USENIX Security Symposium, pages 127–142, 2021.
- [141] Simon Oya and Florian Kerschbaum. IHOP: Improved statistical query recovery against searchable symmetric encryption through quadratic optimization. In 31st USENIX Security Symposium, pages 2407–2424, 2022.
- [142] Antonis Papadimitriou, Ranjita Bhagwan, Nishanth Chandran, Ramachandran Ramjee, Andreas Haeberlen, Harmeet Singh, Abhishek Modi, and Saikrishna Badrinarayanan. Big data analytics over encrypted datasets with seabed. In OSDI, volume 16, pages 587–602, 2016.

- [143] Sarvar Patel, Giuseppe Persiano, Mariana Raykova, and Kevin Yeo. PanORAMa: Oblivious RAM with logarithmic overhead. In IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS), pages 871–882, 2018.
- [144] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. What storage access privacy is achievable with small overhead? In Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, pages 182–199, 2019.
- [145] Sarvar Patel, Giuseppe Persiano, Kevin Yeo, and Moti Yung. 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, pages 79–93, 2019.
- [146] Giuseppe Persiano and Kevin Yeo. Lower bounds for differentially private RAMs. In Advances in Cryptology–EUROCRYPT 2019: 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, pages 404–434, 2019.
- [147] Perspecsys. <https://tinyurl.com/45ubwnef>.
- [148] Casper Petersen, Jakob Grue Simonsen, and Christina Lioma. Power law distributions in information retrieval. ACM Transactions on Information Systems (TOIS), 34(2):1–37, 2016.
- [149] Benny Pinkas and Tzachy Reinman. Oblivious RAM revisited. In Advances in Cryptology–CRYPTO 2010: 30th Annual Cryptology Conference, pages 502–519, 2010.
- [150] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. Arx: an encrypted database using semantically secure encryption. Cryptology ePrint Archive, 2016.
- [151] Rishabh Poddar, Stephanie Wang, Jianan Lu, and Raluca Ada Popa. Practical volume-based attacks on encrypted databases. In 2020 IEEE European Symposium on Security and Privacy (EuroS&P), pages 354–369. IEEE, 2020.
- [152] Geong Sen Poh, Moesfa Soeheila Mohamad, and Muhammad Reza Z’aba. Structured encryption for conceptual graphs. In International Workshop on Security, pages 105–122. Springer, 2012.
- [153] Raluca Ada Popa, Catherine MS Redfield, Nikolai Zeldovich, and Hari Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In Proceedings

- of the twenty-third ACM Symposium on Operating Systems Principles (SOSP), pages 85–100, 2011.
- [154] Christian Priebe, Kapil Vaswani, and Manuel Costa. Enclavedb: A secure database using sgx. In 2018 IEEE Symposium on Security and Privacy (SP), pages 264–278, 2018.
 - [155] Redis. <https://redis.io/>.
 - [156] Leonie Reichert, Gowri R Chandran, Phillipp Schoppmann, Thomas Schneider, and Björn Scheuermann. Menhir: An oblivious database with protection against access and volume pattern leakage. In Proceedings of the 19th ACM Asia Conference on Computer and Communications Security, ASIA CCS '24, page 1675–1690, New York, NY, USA, 2024. Association for Computing Machinery.
 - [157] Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Constants count: Practical improvements to oblivious RAM. In USENIX Security Symposium, pages 415–430. USENIX Association, 2015.
 - [158] Cetin Sahin, Victor Zakhary, Amr El Abbadi, Huijia Lin, and Stefano Tessaro. TaoStore: Overcoming asynchronicity in oblivious data storage. In 2016 IEEE Symposium on Security and Privacy (SP), pages 198–217, 2016.
 - [159] Scalability in MySQL. <https://www.mysql.com/products/cluster/scalability.html>.
 - [160] Maryam Sepehri and Florian Kerschbaum. Low-cost hiding of the query pattern. In Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security, pages 593–603, 2021.
 - [161] Amin Setayesh, Cheran Mahalingam, Emily Chen, and Sujaya Maiyya. Treebeard: A scalable and fault tolerant oram datastore. 2024. <https://bit.ly/3ziSH0c>.
 - [162] Sagar Sharma, James Powers, and Keke Chen. Privategraph: Privacy-preserving spectral analysis of encrypted graphs in the cloud. IEEE Transactions on Knowledge and Data Engineering, 31(5):981–995, 2018.
 - [163] Meng Shen, Baoli Ma, Liehuang Zhu, Rashid Mijumbi, Xiaojiang Du, and Jiankun Hu. Cloud-based approximate constrained shortest distance queries over encrypted graphs with privacy protection. IEEE Transactions on Information Forensics and Security, 13(4):940–953, 2017.

- [164] Elaine Shi, T-H Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((\log n)^3)$ worst-case cost. In Advances in Cryptology–ASIACRYPT 2011: 17th International Conference on the Theory and Application of Cryptology and Information Security, pages 197–214, 2011.
- [165] Skyhigh Networks. <https://www.skyhighsecurity.com>.
- [166] SQLite4. <https://sqlite.org/src4/doc/trunk/www/design.wiki>.
- [167] Emil Stefanov and Elaine Shi. Multi-cloud oblivious storage. In Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, pages 247–258, 2013.
- [168] Emil Stefanov and Elaine Shi. Oblivistore: High performance oblivious cloud storage. In 2013 IEEE Symposium on Security and Privacy (SP), pages 253–267, 2013.
- [169] Emil Stefanov, Elaine Shi, and Dawn Xiaodong Song. Towards practical oblivious RAM. In 19th Annual Network and Distributed System Security Symposium (NDSS), 2012.
- [170] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS), pages 299–310, 2013.
- [171] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. Cockroachdb: The resilient geo-distributed sql database. In Proceedings of the 2020 ACM International Conference on Management of Data (SIGMOD), pages 1493–1509, 2020.
- [172] Jeff Terrace and Michael J Freedman. Object storage on craq: High-throughput chain replication for read-mostly workloads. In USENIX Annual Technical Conference, number June, pages 1–16. San Diego, CA, 2009.
- [173] The Infrastructure Behind Twitter: Scale. <https://bit.ly/3KJR7J1>. Accessed Feb 10, 2023.
- [174] Robert H Thomas. A majority consensus approach to concurrency control for multiple copy databases. ACM Transactions on Database Systems (TODS), 4(2):180–209, 1979.

- [175] TiDB. <https://www.pingcap.com>.
- [176] Shruti Tople, Yaoqi Jia, and Prateek Saxena. Pro-oram: Practical read-only oblivious ram. In 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019), pages 197–211, 2019.
- [177] Robbert Van Renesse and Fred B Schneider. Chain replication for supporting high throughput and availability. In OSDI, volume 4, pages 91–104, 2004.
- [178] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. Conclave: secure multi-party computation on big data. In Proceedings of the Fourteenth EuroSys Conference 2019, pages 1–18, 2019.
- [179] Midhul Vuppalapati, Kushal Babel, Anurag Khandelwal, and Rachit Agarwal. SHORTSTACK: distributed, fault-tolerant, oblivious data accesss. In 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI), pages 719–734, 2022.
- [180] Jianfeng Wang, Shi-Feng Sun, Tianci Li, Saiyu Qi, and Xiaofeng Chen. Practical volume-hiding encrypted multi-maps with optimal overhead and beyond. In Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, pages 2825–2839, 2022.
- [181] Xiao Wang, Hubert Chan, and Elaine Shi. Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pages 850–861, 2015.
- [182] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, CCS ’14, page 215–226, New York, NY, USA, 2014. Association for Computing Machinery.
- [183] Mor Weiss and Daniel Wichs. Is there an oblivious ram lower bound for online reads? Journal of Cryptology, 34(3):18, 2021.
- [184] Peter Williams, Radu Sion, and Alin Tomescu. PrivateFS: A parallel oblivious file system. In Proceedings of the 2012 ACM SIGSAC Conference on Computer and Communications Security (CCS), pages 977–988, 2012.
- [185] David J Wu, Joe Zimmerman, J  r  my Planul, and John C Mitchell. Privacy-preserving shortest path computation. arXiv preprint arXiv:1601.02281, 2016.

- [186] Xinan Yan, Linguan Yang, Hongbo Zhang, Xiayue Charles Lin, Bernard Wong, Kenneth Salem, and Tim Brecht. Carousel: Low-latency transaction processing for globally-distributed data. In Proceedings of the 2018 International Conference on Management of Data (SIGMOD), pages 231–243, 2018.
- [187] Sergey Yekhanin. Private information retrieval. In Locally Decodable Codes and Private Information Retrieval Schemes, pages 61–74. Springer, 2010.
- [188] YugaByte: Key-value data model. <https://docs.yugabyte.com/preview/develop/data-modeling/common-patterns/keyvalue/>. Accessed Feb 28, 2025.
- [189] V. Zakhary, MJ Amiri, Sujaya Maiyya, D. Agrawal, and A. El Abbadi. Towards global asset management in blockchain systems. BCDL co-located with VLDB, 2019.
- [190] Victor Zakhary, Cetin Sahin, Amr El Abbadi, Huijia Lin, and Stefano Tessaro. Pharos: Privacy hazards of replicating oram stores. In Proceedings of the 21th International Conference on Extending Database Technology, EDBT 2018, 2018.
- [191] Irene Zhang, Naveen Kr Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan RK Ports. Building consistent transactions with inconsistent replication. ACM Transactions on Computer Systems (TOCS), 35(4):1–37, 2018.
- [192] Jinsheng Zhang, Qiumao Ma, Wensheng Zhang, and Daji Qiao. Tskt-oram: A two-server k-ary tree oram for access pattern protection in cloud storage. In MILCOM 2016-2016 IEEE Military Communications Conference, pages 527–532. IEEE, 2016.
- [193] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In USENIX Security Symposium, volume 2016, pages 707–720, 2016.
- [194] F. Zhao, Sujaya Maiyya, R. Wiener, D. Agrawal, and A. El Abbadi. Kll[±]: Approximate quantile sketches over dynamic datasets. Proceedings of the VLDB Endowment, 2021.
- [195] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), pages 283–298, 2017.