Efficient Oblivious Query Processing for Range and kNN Queries

Zhao Chang, Dong Xie, Feifei Li, Jeff M. Phillips, and Rajeev Balasubramonian

Abstract—Increasingly, individuals and companies adopt a cloud service provider as a primary data and IT infrastructure platform. The remote access of the data inevitably brings the issue of trust. Data encryption is necessary to keep sensitive information secure and private on the cloud. Yet adversaries can still learn valuable information regarding encrypted data by observing data access patterns. To solve such problem, Oblivious RAMs (ORAMs) are proposed to completely hide access patterns. However, most ORAM constructions are expensive and not suitable to deploy in a database for supporting query processing over large data. Furthermore, an ORAM processes queries *synchronously*, hence, does not provide high throughput for *concurrent query processing*. In this article, we design a practical *oblivious query processing framework* to enable efficient query processing over a cloud database. In particular, we focus on processing multiple range and kNN queries *asynchronously and concurrently with high throughput*. The key idea is to integrate indices into ORAM which leverages a suite of optimization techniques (e.g., oblivious batch processing and caching). The effectiveness and efficiency of our oblivious query processing framework is demonstrated through extensive evaluations over large datasets. Our construction shows an order of magnitude speedup in comparison with other baselines.

Index Terms—Data privacy, oblivious RAM, oblivious query processing, range and kNN query

1 Introduction

Increasingly, individuals and companies choose to move their data and IT operations to a cloud service provider (e.g., Azure, AWS) and use the cloud as their primary infrastructure platform. While utilizing a cloud infrastructure offers many attractive features and is a cost-effective solution in many cases, the potential risk of compromising sensitive information poses a serious threat.

A necessary step for keeping sensitive information secure and private on the cloud is to encrypt the data. To that end, encrypted databases such as Cipherbase [1], [2], CryptDB [3], TrustedDB [4], SDB [5], and Monomi [6], as well as various query execution techniques over encrypted databases [7], [8], [9], [10] have been developed. But query access patterns over an encrypted database can still pose a threat to data privacy and leak sensitive information, even if the data is encrypted before uploading to the cloud and a secure query processing method over encrypted data is used [11], [12], [13], [14]. Islam *et al.* [15] demonstrate that an attacker can identify as much as 80 percent of email search queries by observing the access pattern of an encrypted email repository alone. Moreover, by counting the frequency of accessing data items from the clients, the server is able to analyze the importance of different areas in the database. With certain background knowledge, the server can learn a great deal about client queries and/or data. For example, knowing that the database stores spatial POIs from NYC, the most frequently accessed records are probably from Manhattan area [11]. The recent Spectre attack [16] shows that potentially vulnerable code patterns can be exploited easily by engaging speculation features in processors. At its heart, the attack takes advantage of the fact that internal program secrets are betrayed by the program's access pattern. It thus highlights the importance of ORAM primitives in protecting an application's access pattern and its sensitive data.

The examples above highlight the necessity of hiding the access patterns of clients' operations on a cloud and protect against the sensitive information leakage. To that end, Oblivious RAM (ORAM) is proposed by Goldreich [17] and Ostrovsky [18] to protect the client's access patterns from the cloud. It allows a client to access encrypted data on a server without revealing her access patterns to the server.

However, most existing practical ORAM constructions are still very expensive, and not suitable for deployment in a database engine to support query processing over large data [11]. Furthermore, an ORAM by itself does not support *query-level concurrency*, i.e., an ORAM processes incoming queries synchronously: a new query request is not processed until a prior ongoing query has been completed. This creates a serious bottleneck under concurrent loads in a database setting with multiple clients. Many ORAM constructions [17], [19], [20], [21], [22], [23] do not even support *operation-level concurrency*, i.e., these ORAMs handle operations (each operation is to read or write a block) *synchronously*. Recent studies have addressed this issue and proposed various *parallel ORAMs at the storage level* that can handle operations *asynchronously*, e.g., PrivateFS [24], Shroud [25], ObliviStore [26], CURIOUS [27], and TaoStore [28], hence, achieving *operation-level concurrency at the storage level*.

Since each query (e.g., a range or a kNN query) consists of a sequence of read operations (read a block, which will

• *The authors are with the School of Computing, University of Utah, Salt Lake City, UT 84112 USA. E-mail: {zchang, dongx, lifeifei, jeffp, rajeev}@cs.utah.edu.*

Manuscript received 5 Oct. 2019; revised 18 Oct. 2020; accepted 17 Feb. 2021. Date of publication 0 . 0000; date of current version 0 . 0000.
(Corresponding author: Zhao Chang.)
Recommended for acceptance by P. Pietzuch.
Digital Object Identifier no. 10.1109/TKDE.2021.3060757

also result in write operations when operating over an ORAM structure), parallel ORAMs with their support for *operation-level concurrency* are useful in *reducing query latency*, which will improve system throughput indirectly, but they are *not designed for improving system throughput*. For example, a single expensive query that consists of many operations can still seriously hurt system throughput even if its latency has been reduced. In short, operation-level concurrency using a parallel ORAM storage engine does not lead to query-level concurrency in a query engine.

Just to clarify, our query-level concurrency works in a *batched manner*. It means that if any query q_1 (in the last batch) is currently executed and a new query q_2 arrives in the meantime, we will not run q_2 instantly. Query q_2 will not start until all the queries in the last batch (including q_1) are completed. If another query q_3 arrives before the last batch (containing q_1) is completed, the execution of q_2 and q_3 can be performed concurrently in the next batch. The details will be demonstrated in Sections 4.4 and 4.5.

Prior efforts mainly focus on designing efficient query processing protocols for specific types of queries, e.g., join [29], [30] and shortest path [19], [31]. Some studies focus on providing theoretical treatment for SQL queries [13], but are of only theoretical interest. There are also investigations working on designing *oblivious data structures* [14], [32] that help improve the efficiency of certain queries (e.g., binary search) compared to processing these queries using a standard ORAM construction. The idea is that for some query algorithms which exhibit a degree of predictability in their access patterns, it will be beneficial to have customized and more efficient ORAM constructions [32].

To the best of our knowledge, Opaque [12] and ObliDB [33] are the state-of-the-art studies concerning *generic oblivious analytical processing*. However, to support kNN or range queries, Opaque needs to perform expensive scan-based operations (see Baseline part in Section 3). ObliDB [33] exploits indexed storage method and builds oblivious B+ trees to support point and range queries. In their implementation, data is fixed to one record per block. But in our implementation of oblivious B-tree in Section 4.2, each block contains B bytes, and the number of records that fit in each data block is $\Theta(B)$ rather than one. Hence, our design is more suitable for hard disk storage and reduces the number of disk seeks in query processing. We also leverage a suite of optimization techniques including batch processing and ORAM caching. Extensive experimental evaluation shows that our design with those optimizations achieves an order of magnitude speedup in terms of query throughput, in comparison with Opaque method (without the distributed storage) and the basic oblivious index baseline (similar to ObliDB).

Our Contributions. We propose a general oblivious query processing framework (OQF) for cloud databases, which is efficient and practical (easy to implement and deploy) and supports concurrent query processing (i.e., concurrency within a query's processing) with high throughput. This work focuses on (one and multi-dimensional) range and kNN queries, and explores the design of OQF that is much more efficient than baseline approaches. The proposed framework can be extended to handle other query types (e.g., joins), which is an active ongoing work. In particular,

- We formalize the definition of an oblivious query processing framework (OQF) and review the background of oblivious query processing in Section 2.
- We describe the architecture of our OQF design in Section 3, and a present baseline instantiation based on a standard ORAM protocol.
- We present our design of an OQF in Section 4 that achieves concurrent query processing with high throughput using the idea of integrating an (oblivious) index into ORAM and also leveraging a suite of optimization techniques (e.g., oblivious batch processing and caching).
- We conduct extensive experiments in Section 5 using our oblivious query processing framework on large datasets. The results demonstrate a superior performance gain (at least one order of magnitude) achieved by our design over baseline constructions.

The paper is concluded in Section 7 with a review of related work in Section 6.

2 Preliminaries

2.1 Problem Definition and Security Model

Consider a client who would like to store her data D on a remote server (e.g., cloud) and ask other clients (including herself) to issue queries (such as range and k nearest neighbor queries). A trusted coordinator collects queries from different clients and answers them by interacting with the server. The communication between clients and the coordinator are secured and not observed by the server. Index structures such as B-tree and R-tree are often built to enable efficient query processing. Suppose the query sequence to the server for queries collectively issued by all clients is $\{(\mathrm{op}_1, \arg_1), \ldots, (\mathrm{op}_m, \arg_m)\}$, where op_i is a query type (which may be range or kNN in our context) and $\arg_i$ provides the arguments for the ith query q_i. Our goal is to protect the privacy of clients by preventing the server from inferring knowledge about the queries themselves, the returned results, and the database D.

While traditional encryption schemes can provide confidentiality, they do not hide data access patterns. This enables the server to infer the query behavior of clients by observing access locality from the index structure and the data itself. Formally, our problem can be defined as below:

Definition 1. ObliviousQueryProcessing. *Given an input query sequence $\vec{q} = \{(\mathrm{op}_1, \arg_1), (\mathrm{op}_2, \arg_2), \ldots, (\mathrm{op}_m, \arg_m)\}$, an oblivious query processing protocol P should interact with an index structure I built on the server over the encrypted database D to answer all queries in $\vec{q}$ such that all contents of D and I stored on the server and messages involved between the coordinator and the server should be confidential. Denote the access pattern produced by P for $\vec{q}$ as $P(\vec{q})$. In addition to confidentiality, for any other query sequence $\vec{q}_*$ so that the access patterns $P(\vec{q})$ and $P(\vec{q}_*)$ have the same length, they should be computationally indistinguishable for anyone but the coordinator and clients.*

Security Model. Note that multiple clients may exist and retrieve the data as long as they are trusted by the client who is the original data owner and follow the same client side protocol. In this paper, we consider an "honest-but-curious" server, which is consistent with most existing work in the literature. To ensure confidentiality, the client needs to store the secret keys of a symmetric encryption scheme. The encryption should be done with a *semantically secure* encryption scheme, and therefore two encrypted copies of the same data block look different [34]. The client should re-encrypt a block before storing it back to the cloud and decrypt a block after retrieving it. Since these encryption/decryption operations are independent of the design of an OQF, we may omit them while describing an OQF.

Data is encrypted, retrieved, and stored in *atomic units* (i.e., blocks), same as in a database system. We must make all blocks of the same size; otherwise, the cloud can easily distinguish these blocks by observing the differences in size. We use N to denote the number of real data blocks in the database. Each block in the cloud or client storage contains B bytes (note that the number of entries that fit in a block is $\Theta(B)$ and the constants will vary depending on the entry types, e.g., encrypted record versus encrypted index entry).

Definition 1 implies that we must make different access types (read and write operations) indistinguishable. This is achieved by performing read-then-write (potentially a dummy write) operations, which is commonly used in existing ORAMs. Our security definition requires indistinguishability only for query sequences inducing access patterns of the same length. We will discuss how to protect against volume leakage from range query by introducing padding techniques in Section 4.6.

Definition 1 does not consider privacy leakage through any side-channel attack like time taken for each operation (timing attack). Existing work [35] actually offers mechanisms for bounding ORAM timing channel leakage to a user-controllable limit. Oblix [36] also considers any side-channel leakage as out of scope. Orthogonal solutions [37], [38] in Oblix also work for our setting.

Remarks. Note that our setting is that *multiple clients submit queries at any time* instead of the scenario where one unique client makes a large number of queries. The coordinator runs the oblivious query algorithms acting as a trusted middle layer between multiple clients and the untrusted cloud (the same setting in TaoStore [28]). The coordinator and the clients are *in a closed and private internal network*. Analogously, ObliviStore [26] hosts the trusted components *in a small-scale private cloud*, while outsourcing the untrusted storage to a remote cloud. If the cloud has a secure hardware that comes with trusted private memory regions, e.g., the enclave from SGX [39], we can make it co-located on the cloud, serving as the trusted coordinator.

2.2 ORAM and Oblivious Data Structure

Oblivious RAM. Oblivious RAM (ORAM) is first proposed by Goldreich and Ostrovsky where the key motivation is to offer software protection from an adversary who can observe the memory access patterns. In the ORAM model, the client, who has a small but private memory, wants to store and retrieve her data using the large but untrusted server storage, while preserving data privacy. Generally, ORAM is modeled similar as a key-value store. Data is encrypted, retrieved, and stored in atomic units (i.e., blocks) annotated by unique keys. An ORAM construction will hide access patterns of block operations (i.e., `get()` and `put()`) to make them computationally indistinguishable to server.

An ORAM construction consists of two components: anORAMdatastructure and anORAMqueryprotocol, where a part of the ORAM data structure is stored on the server side, and another (small) part of the ORAM data structure is stored on the client side. Client and server then run the ORAM query protocol to read and write any data blocks.

Path-ORAM. Path-ORAM is a key representative among proposed ORAM constructions due to its good performance and simplicity [11], [23]. It organizes the server side ORAM structure as a full binary tree where each node is a bucket that holds up to a fixed number of encrypted blocks (from the client's database), while the client hosts a small amount of local data in a *stash*. Path-ORAM maintains *the invariant*that at any time, each block b is mapped to a leaf node chosen uniformly at random in the binary tree, and is always placed in some bucket along the path to the leaf node that b is mapped to. The private stash stores a small set of blocks that have not been written back to the server.

When block b is requested by the client, Path-ORAM protocol will retrieve an entire path, with the leaf node that b is mapped to, from the server into the client's stash. Then, the requested block b is re-mapped to another leaf node, and the entire path that was just accessed is written back to the server. When a path is written back, additional blocks may be evicted from the stash if the above invariant is preserved and there is free space in some bucket along that path.

In this construction, the client has to keep a *position map* to keep track of the mapping between blocks and leaf node IDs, which brings a linear space cost to the client; note that even though it is linear with N, the number of blocks in the database, the mapping information is much smaller than the actual database size. We may choose to recursively build Path-ORAMs to store position maps until the final level position map is small enough to fit in client memory.

To store N blocks of size B, a basic Path-ORAM protocol requires $O(\log N + N/B)$ client side blocks and can process each request at a cost of $O(\log N)$. In a recursive Path-ORAM, the client needs a memory size of $O(\log N)$ and each request can be processed in $O(\log_B N \cdot \log N)$ cost.

Oblivious Data Structure. For certain data structure (such as map and queue) whose data access pattern exhibits some degree of predictability, one may improve the performance of oblivious access by making these data structures "oblivious" (in the memory hierarchy sense), rather than simply storing (data and index) blocks from such a data structure bluntly into a generic ORAM construction. Wang *et al.* [32] design oblivious data structures and algorithms for some standard data structures. In particular, they propose the methodology to build oblivious data structures for AVL tree. The main idea is that each node keeps the position map information of its children nodes together with their page IDs. When retrieving a node from this oblivious data structure, we acquire the position map for its children simultaneously. Note that most query algorithms over tree indices traverse the tree from the root to the leaf. As a result, the client only needs to remember the position tag for the root node block, and all other position map information can be fetched on the fly from the oblivious data structure stored on the server.

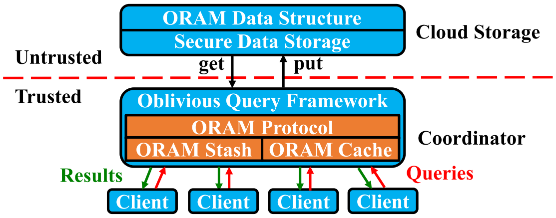


Fig. 1. Oblivious query framework.

3 Framework

Our proposed OQF consists of four parties: the data owner, clients (data owner can be a client), a trusted coordinator, and the server. The trusted coordinator has limited storage, and answers queries from different clients by interacting with the server while ensuring the security in Definition 1.

In a pre-processing step, the data owner partitions records in the database D into blocks, encrypts these data blocks, and builds an ORAM data structure (e.g., Path-ORAM) over these data blocks. She then uploads both encrypted data blocks and the ORAM data structure to the server. She shares the encryption/decryption keys and other metadata (e.g., position map in Path-ORAM), which are needed to execute an ORAM protocol, with the coordinator. The server stores the encrypted data blocks and the ORAM data structure into a secure cloud data storage.

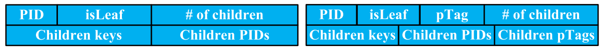
Subsequently, clients may issue (range and kNN) queries against the cloud server through the coordinator. Using an oblivious query algorithm that will be described later in details, the coordinator reads/writes blocks from/to the server based on an ORAM protocol and returns query results to the clients. The clients and the coordinator are trusted. The communication between them are secured and not observed by the cloud. The oblivious query framework is shown in Fig. 1.

Note that *an ORAM protocol refers to steps taken in order to read or write a single data block securely and obliviously* with the help of the ORAM metadata on the coordinator and the ORAM data structure on the server. The oblivious query algorithm is constructed based on this ORAM protocol to answer a range or kNN query securely and obliviously.

Baseline. The most straightforward solution is to encrypt each data block from the database D, store these encrypted blocks to the server, and process queries obliviously by scanning through all the encrypted blocks over the coordinator.

Specifically, the coordinator can answer a range query simply by retrieving each encrypted data block from the server, decrypting it and checking all records in the block against the query range. For a kNN query, the coordinator will scan through all encrypted data blocks as well, calculate the distance from each data point to the query, and maintain a bounded priority queue to figure out the global kNN result. Note that the coordinator has to retrieve *every encrypted block in a fixed order* to process each query. From the server's perspective, the access pattern from the coordinator is always the same, thus no information can be inferred by observing access patterns. As a result, simple encryption is enough and ORAM is not required.

This baseline is clearly very expensive, but simple to implement. This is essentially the solution explored by the recent work known as Opaque [12]. Opaque uses the above baseline with a distributed cloud storage.



(a) Standard B-tree. (b) Oblivious B-tree.

Fig. 2. B-tree internal node layout.

4 Efficient OQF

4.1 Integrate an Index into ORAM

A better solution is to add an index (e.g., B-tree or R-tree) over the database D before uploading data to the cloud. It takes some care to utilize the index obliviously though.

The key idea is to ignore the semantic difference of the (encrypted) index and data blocks from the data owner, and store all the blocks into an ORAM construction, say Path-ORAM. Take B-tree as an example: each node in a B-tree can be organized in a disk page as shown in Fig. 2a; the pointers to its children nodes in the tree are page IDs. Hence, we can treat such pages as ORAM blocks uniquely identified by their page IDs (i.e., ORAM block IDs).

In this case, the ORAM data structure on the server is the Path-ORAM data structure over both encrypted index and data blocks. The ORAM protocol is simply the read and write (a single block) operations through Path-ORAM.

When answering a query, we follow the range or kNN query algorithm in a B-tree or R-tree, and start with retrieving the root block (of the index) from the server. We then traverse down the tree to answer the incoming query. Whenever we need a tree node that does not reside in the coordinator memory, we retrieve the block by looking up its block ID through the ORAM protocol. Intuitively, we query the index structure by running the same algorithm as that over a standard B-tree or R-tree index. The only difference is that we are retrieving index and data blocks through an ORAM protocol with the help of the ORAM data structure.

Suppose we exploit the basic Path-ORAM protocol as the underlying ORAM protocol. Retrieving a block has $O(\log N)$ overhead in both communication and computation, where N is the total number of data blocks. The fanout for index blocks is $\Theta(B)$, where B is the block size in bytes. Now take a B-tree point query as an example. Each point query would cost $O(\log_B N \cdot \log N)$, where the height of B-tree is $O(\log_B N)$. Recall that the basic Path-ORAM protocol requires $O(\log N + N/B)$ client side memory to record the position map, which may be not practical for a coordinator over a large dataset. To address this problem, we can adopt recursive Path-ORAM protocol which only requires $O(\log N)$ memory in the coordinator but increases the cost of retrieving one block to $O(\log_B N \cdot \log N)$. Hence, the above B-tree query algorithm will cost $O(\log_B^2 N \cdot \log N)$.

One can easily generalize this query algorithm to range and kNN queries using the corresponding range and kNN query algorithms for a B-tree or an R-tree.

4.2 Oblivious B-Tree and R-Tree

Another approach is to explore the idea of building an oblivious data structure [14], [32], which will eliminate the need of storing any position map at the coordinator. In particular, Wang *et al.* [32] leverage pointer-based technique to build an oblivious AVL tree. In our design, we simply replace a standard B-tree or R-tree in Section 4.1 with an oblivious B-tree

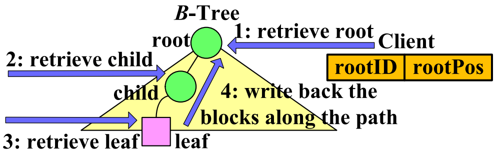


Fig. 3. An example of querying an oblivious B-tree.

or R-tree. Note that B-tree/R-tree has much larger fanout in index levels than AVL tree and then achieves a lower tree height. Suppose N is the number of real data blocks and B is the block size in bytes. In B-tree/R-tree, the fanout is $\Theta(B)$ and the tree height is $O(\log_B N)$; but in AVL tree, the fanout is only two, which leads to $O(\log N)$ tree height. Since the cost of searching over a tree index is related to the tree height, oblivious B-tree/R-tree achieves higher query performance than oblivious AVL tree.

The main idea of building oblivious tree structures is that each node in the index keeps the position map information of its children nodes together with their block IDs. Fig. 2b shows the new B-tree node for *an oblivious B-tree*. When retrieving a node from the server using the ORAM protocol, we have acquired the position map for its children nodes simultaneously. Note that most query algorithms over tree indices traverse the tree from the root to leaf nodes. As a result, the coordinator only needs to remember the position tag of the root node, and all other position map information can be fetched on the fly as part of the query algorithm.

As before, the Path-ORAM structure on the cloud stores both index and data blocks and makes no distinction between these two types of blocks. We illustrate how to answer a query obliviously in this case, using again B-tree point query as an example (see Fig. 3 for an illustration):

1) The coordinator retrieves the root node block from the cloud through the Path-ORAM protocol by using its position map, and then assigns the root node block to a random leaf node ID in the Path-ORAM tree by altering its position map.
2) By observing key values in the retrieved node b, the coordinator decides which child node to retrieve next and acquires its position map information directly from the parent node b.
3) The coordinator retrieves the child node using the position map acquired in the last step and assigns a new random leaf node ID to the child node block by altering the position map stored in its parent node.
4) Repeat Step 2 and 3 until the coordinator reaches a leaf node. The record(s) that matches the point query search key will be found.

Note that when retrieving any node b other than the root node, we need to alter the position tag of its parent node to store the fact that b is assigned to the path with a new random leaf node in the Path-ORAM tree. Thus, we need to modify the Path-ORAM protocol slightly, to prevent the protocol from writing an index block back to the cloud while we are still traversing its descendants.

In summary, by integrating the position map information to the block content of a tree node, we can avoid saving the full position map in coordinator memory or using the expensive recursive Path-ORAM construction. Specifically, this new method requires $O(\log N)$ coordinator memory, which includes the Path-ORAM stash (with $O(\log N)$ size) and the memory needed (with $O(\log_B N)$ size) to store the traversed path for updating the position map information recursively. Its query cost for each B-tree point query is $O(\log_B N \cdot \log N)$, the same as that of using the original Path-ORAM construction with a standard index.

Lastly, a similar design and analysis can be carried out for constructing an oblivious R-tree from a standard R-tree; we omit the details in the interest of space.

4.3 A Comparison of Different Designs

Table 1 compares Baseline (Opaque) in Section 3 (essentially Opaque method [12] without distributed storage in cloud), ORAM+Index in Section 4.1 (ORAM with a standard index) and Oblivious Index in Section 4.2. The comparison is based on B-tree point query in terms of cloud storage, coordinator storage, number of communication rounds per query, and computation overhead per query. Recall that for all the designs, per query, number of accessed blocks in the cloud, communication overhead in bytes, and computation cost in the coordinator have the same Big-O complexity. Hence, we use the computation overhead to denote the Big-O complexity of those metrics. Note that Oblivious Index saves the coordinator memory size, but involves $O(1)$ times more computation overhead and communication rounds than ORAM+Index to recursively update the position map information to the server. Therefore, Oblivious Index may be suitable when the coordinator only has limited memory.

4.4 Optimizations

In most practical database applications with multiple clients, a critical objective is to improve the overall query throughput. A useful optimization technique is to *process queries in batches*. This allows the coordinator to retrieve index and data blocks from the cloud in batch.

Batch processing brings the benefit of ORAM caching. The coordinator can leverage a good caching strategy that takes advantage of the access pattern for queries in the same batch. In detail, the coordinator introduces an ORAM buffer of a given size on her side, and the ORAM buffer stores a set of blocks from the Path-ORAM structure on the cloud that she has previously retrieved. If there is a buffer

TABLE 1
Comparison of Different Designs

Design	Computation Overhead	Cloud Storage	Communication Round	Coordinator Storage
Baseline (Opaque)	$O(N)$	$O(N)$	$O(N)$	$O(1)$
ORAM+Index	$O(\log_B N \cdot \log N)$	$O(N)$	$O(\log_B N)$	$O(\log N + N/B)$
Oblivious Index	$O(\log_B N \cdot \log N)$	$O(N)$	$O(\log_B N)$	$O(\log N)$

hit for a subsequent block request, the coordinator does not need to retrieve that block from the cloud again using the expensive ORAM protocol. Note that each of these blocks can be either an index or a data block from the original database with an index (e.g., a B-tree/R-tree or an oblivious B-tree/R-tree).

An important and interesting challenge arises from this discussion, which is how to design a good caching strategy for the coordinator to improve the overall performance of the proposed oblivious query processing framework.

4.4.1 ORAM Caching at the Coordinator

Formally, given a buffer size τ (number of data blocks that can be stored in the coordinator's buffer) and a query batch size g (g queries in one query batch), our objective is to design a good ORAM caching strategy to reduce the cost of processing a sequence of query batches obliviously and improve the overall query throughput of the proposed OQF, where the system query throughput is simply defined as the number of queries processed per minute.

To illustrate the key idea of our design, we *assume for now* that given a query batch with g queries $\{q_1, \ldots, q_g\}$, the coordinator is able to infer the set of blocks (index and data blocks) to be retrieved by each query, i.e., there is a mapping function h that takes a query q and outputs the set of block IDs that refers to blocks to be accessed while processing q. We will discuss how to design h in Section 4.5.

The following analysis assumes the basic Path-ORAM protocol, where the coordinator would traverse a whole path (read-and-then-write) from the Path-ORAM structure stored on the cloud server through Path-ORAM protocol, when a cache miss happens for reading a particular block b. Formally, the problem is reduced to the followings.

Given a query sequence of s query batches: $\{(q_{1,1}, \ldots, q_{1,g}), \ldots, (q_{s,1}, \ldots, q_{s,g})\}$, the ith batch needs to retrieve a set of m_i blocks with IDs $\{\mathrm{id}_{i,1}, \ldots, \mathrm{id}_{i,m_i}\}$ that will be accessed by $(q_{i,1}, \ldots, q_{i,g})$. We also let $m = \min\{m_1, \ldots, m_s\}$. When the context is clear, we drop the subscript for a batch i. Our objective is to design a good ORAM caching strategy to minimize the number of cache misses over the s batches, with the following *constraint*: queries within a batch can be processed in arbitrary order, but queries across different batches cannot be reordered. Hence, we can bound and adjust the query latency for each query by tuning the query batch size g.

OfflineOptimalStrategy. In offline setting, the coordinator knows block IDs from all (future) query batches. We denote the optimal strategy for a given query sequence as opt.

OnlineStrategy. In online setting, the coordinator knows *only* block IDs from the current query batch. The goal is to find a strategy that enjoys a good competitive ratio [40]. Specifically, suppose $\mathcal{I}$ represents the class of all valid inputs (each input in $\mathcal{I}$ is a sequence of batches of queries), $\mathcal{A}$ represents the class of all valid online algorithms for the ORAM caching problem, and $\mathrm{cost}(A, I)$ represents the cost of running algorithm $A \in \mathcal{A}$ over an input $I \in \mathcal{I}$. Then the competitive ratio of A is

$$\rho(A) = \max_{I \in \mathcal{I}} \frac{\mathrm{cost}(A, I)}{\mathrm{cost}(\mathrm{opt}, I)}, \quad (572)$$

where $\mathrm{cost}(A, I)$ (or $\mathrm{cost}(\mathrm{opt}, I)$) is proportional to the number of retrieved blocks from the cloud through ORAM.

ORAM Caching Strategy. We are given a query sequence $Q = \{(q_{1,1}, \ldots, q_{1,g}), \ldots, (q_{s,1}, \ldots, q_{s,g})\}$ that will access a block sequence $Q_b = \{(\mathrm{id}_{1,1}, \ldots, \mathrm{id}_{1,m_1}), \ldots, (\mathrm{id}_{s,1}, \ldots, \mathrm{id}_{s,m_s})\}$. Whenever the coordinator needs to replace a cached block, she evicts the block in her cache that is not accessed until Furthest-In-Future (FIF) with regard to Q_b. The evicted block is then re-mapped to a new leaf node ID in Path-ORAM data structure, before being placed into the private stash with the new mapping information.

Recall that in Path-ORAM protocol, when reading a block b, an entire path (which contains b) will be retrieved from the cloud. Here, we assume the coordinator only caches the block b in her buffer and places other real blocks along that path into the stash as that in the original protocol.

Under this setting, each cache miss (caused by the request to access a block) leads to the same cost, which is to read a block from the ORAM data structure in the cloud using the ORAM protocol. Recall that the coordinator reorders the queries within a batch. After that, the ordering of queries is fixed. This setting leads to the following result. The proof is fairly straightforward, and hence omitted.

Theorem 1. *For a query sequence with fixed ordering of queries, the optimal offline method for our ORAM caching problem is the FIF caching strategy.*

The offline optimal method inspires us to design the following online strategy. In online setting, the coordinator can only see $Q_{b,i} = \{(\mathrm{id}_{i,1}, \ldots, \mathrm{id}_{i,m_i})\}$ for query batch $Q_i = \{(q_{i,1}, \ldots, q_{i,g})\}$. After processing the jth query from Q_i, there are two classes of blocks in the ORAM cache: classa : those who *will* appear in $\{(q_{i,j+1}, \ldots, q_{i,g})\}$; classb : those who *will not* appear in $\{(q_{i,j+1}, \ldots, q_{i,g})\}$. A key observation is that *if the coordinator was to see the entire future query batches as in offline setting*, each block from classb should be evicted first before evicting any block from classa. Each block in classb is guaranteed to be referenced only further-in-the-future than any block in classa, and in the offline optimal method, evicted first.

This observation leads to the following online strategy. At any point while processing a query batch, we perform FIF for any blocks in the ORAM cache that belong to classa as defined above at this point, and we use Least Recently Used (LRU) for the remaining blocks in the ORAM cache that belong to classb as also defined above. We always evict a block from classb before evicting any block in classa, and only start evicting blocks from classa if classb is already empty. An evicted block is re-mapped to a randomly chosen leaf node ID in Path-ORAM data structure and placed into the private stash, waiting to be written back to the server. We denote this algorithm as batch-FIF.

Theorem 2. [1] *If there are duplicate block IDs within any batch, $\rho(\text{batch-FIF}) \le \tau$ (τ is the buffer size); otherwise,*

A) *If $\tau \le m$, the competitive ratio $\rho(\text{batch-FIF}) \le 2$;*
B) *Otherwise, the competitive ratio $\rho(\text{batch-FIF}) \le \tau$.*

1. Due to the space limit, all proofs of lemmas and theorems are given in the supplemental material, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TKDE.2021.3060757

4.4.2 Other Optimizations

Query Locality. The coordinator can re-order queries within each query batch to improve their locality, which will lead to better ORAM caching performance regardless of which caching strategy is to be used. For one-dimensional queries, this is easily done by sorting (based on the query point if it is a kNN query or the left-range point if it is a range query). For two-dimensional queries, we can leverage a space-filling curve, and use the z values of the query point for a kNN query and the centroid of a range query box for sorting and re-ordering queries in a batch.

Batch Writing. In the original protocol, for each read operation the coordinator needs to retrieve the entire path and then write the same path back to the cloud. Details are represented in "Path-ORAM" part in Section 2.2. Instead of immediately writing each path back to the cloud, we can also introduce a batch concept to wait for retrieving λ paths and then write all the λ paths back to the cloud at once. Batch writing to tree-based ORAM is also leveraged in prior studies [28], [36], [41]. Specifically, the coordinator can keep the set $\mathcal{H}$ that stores the leaf node indices of the retrieved paths, where $\max|\mathcal{H}| = \lambda$. During batch writing, she writes the λ paths in $\mathcal{H}$ back to the cloud from the bottom level to the top level, which ensures that blocks in her cache and stash can be pushed as deep down into the tree as possible.

Given a leaf node index x, let $P(x, \ell)$ denote the bucket in level ℓ of path $P(x)$. Now for any given block b, for each leaf node index x in $\mathcal{H}$, if b is mapped to x (according to the position map information), and the bucket $P(x, \ell)$ still has space to hold more blocks, the coordinator pushes b into $P(x, \ell)$ and removes b from her cache or stash. The coordinator repeats this process until no more blocks from her cache or stash can be written back to one of the λ paths.

Finally, for each leaf node index x in $\mathcal{H}$ and each level ℓ of path $P(x)$, if bucket location $P(x, \ell)$ still holds blocks with the number less than the maximum capacity of a bucket, the coordinator appends some randomly generated dummy blocks to $P(x, \ell)$ to fulfill its maximum capacity. Finally, she writes all λ paths in $\mathcal{H}$ back to the cloud and clears $\mathcal{H}$.

In our implementation, queries need to be blocked temporarily while writing the λ paths back to the cloud. As in TaoStore [28], we can also keep an additional subtree structure for saving these paths in coordinator and asynchronously write back the λ paths in the background.

Partial Path Retrieval. In the original Path-ORAM protocol, for each block access operation, the coordinator needs to retrieve a whole path from the cloud. With the ORAM caching mechanism and batch writing optimization that we have introduced, for each block access operation, the coordinator only needs to retrieve a partial path, which is not kept in her cache and stash, rather than a whole path in the original Path-ORAM protocol. To be clear, this partial path operation is only performed as part of a batch retrieval, where the part of the path not retrieved in this sub-operation is still retrieved in a larger batch retrieval operation.

An example is shown in Fig. 4. Suppose that blocks along the red-colored paths have already been retrieved and cached by the coordinator. Now the coordinator needs to retrieve the blue-colored path $P(x)$ for a block b, which is mapped to the leaf level node with node ID x. Here, she only needs to retrieve the leaf bucket, since all the remaining buckets (the dotted blue-colored part in path $P(x)$) have already been retrieved.

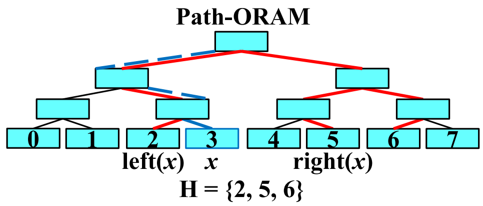


Fig. 4. Partial path retrieval.

To decide which part of $P(x)$ to retrieve, the coordinator builds a set $\mathcal{H}$ to store the leaf node indices of retrieved paths. Given a path to be retrieved by the current operation, identified by the leaf node ID x, she finds

$$\begin{aligned}\text{left}(x) &= \text{argmax}_{y \in \mathcal{H}} y < x,\\ \text{right}(x) &= \text{argmin}_{y \in \mathcal{H}} y > x.\end{aligned}$$

The coordinator checks which part of $P(x)$ is not covered by $P(\text{left}(x)) \cup P(\text{right}(x))$ and only retrieves the blocks from the partial path. Furthermore, and more importantly, the coordinator can check this without the access to the Path-ORAM's binary tree structure.

Theorem 3. *Under partial path retrieval, for any path $P(x)$, each block is either retrieved or already in the stash.*

Block Sorting. If the coordinator has the function h that maps queries to block IDs to be accessed, she can further sort the block IDs in the current batch based on their position tags in Path-ORAM. This improves the performance when combined with batch writing and partial path retrieval optimizations. For those optimizations to make sense, there must be some blocks that reside in the overlap part of the λ paths. Sorting blocks based on their position tags aims to increase the number of overlapping blocks. Intuitively, paths in Path-ORAM that share more overlapping blocks will be put close to each other in the block access sequence after sorting, due to Path-ORAM's full binary tree structure. Then, more overlapping blocks along paths lead to less communication and computation overhead in Path-ORAM. Besides, block sorting also improves the performance of ORAM caching. It makes duplicate block accesses occur in a sequential way, and the coordinator only needs to retrieve each block once rather than multiple times.

4.5 Query to Block ID Mapping

Lastly, in order to apply our ORAM caching algorithm, a mapping function h that maps a query to a set of block IDs is needed. These block IDs represent the index and data blocks that the coordinator needs to retrieve from the cloud.

Intuitively, the coordinator caches *only one specific level* of B-tree or R-tree index in her storage, which is a popular tree-based ORAM optimization [41], [42], [43]. Since the fan-out is large in a B-tree or R-tree index (see the analysis in Section 4.1), this overhead to the coordinator's storage is still far less than storing the entire index. Given any query, the coordinator first finds *which set of blocks that she may need to access* by performing a local search algorithm on the cached level of the index. More specifically, for every node u that is

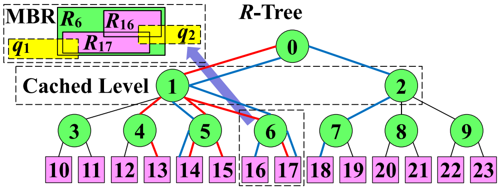


Fig. 5. An example of the set cover-based technique.

cached at the coordinator, we remember the set of index and data blocks from the subtree of u. Henceforth, the local search will return the super set of index and data blocks a query will need to access. This super set allows us to infer the set covers of block IDs to access for all queries in a query batch, and our caching decision will be made based on these set covers of block IDs, instead of the exact set of block IDs.

We take range query in R-tree as an example, as shown in Fig. 5. This R-tree index has three index levels and one leaf level with data blocks. Each (index or leaf) node in the R-tree is shown with its block ID. Suppose that we have a query batch $Q = \{(q_1, q_2)\}$, and results of q_1 and q_2 reside in data blocks $(13, 14, 15, 17)$ and $(14, 16, 17, 18)$ respectively. Thus, the coordinator needs to access blocks (0, 1, 4, 5, 6, 13, 14, 15, 17) to answer q_1 (highlighted in red), and blocks (0, 1, 2, 5, 6, 7, 14, 16, 17, 18) to answer q_2 (highlighted in blue). Assume that the coordinator caches the *second level* of the R-tree index, which contains the minimum bounding rectangle (MBRs) of blocks in the third level of the index, as well as the set of all block IDs from the subtree of node 1 and node 2 respectively. She will know the results of q_1 reside in the MBRs of blocks (4, 5, 6) and those of q_2 reside in the MBRs of blocks (5, 6, 7). Thus, the block sequence to be accessed should be $Q_b = \{(4, 5, 6, [12, 13], [14, 15], [16, 17], 5, 6, 7, [14, 15], [16, 17], [18, 19])\}$, where $[\mathrm{id}_{x_1}, \mathrm{id}_{x_2}, \ldots]$ means that the coordinator may access *one or more blocks* that reside in that set of blocks. In our ORAM caching strategy, to find the furthest reference to a given block in the current query batch, we look for either the exact block ID or a set that covers that block ID. The rest of the caching strategy remains the same as that in Section 4.4.

A similar procedure can be developed for kNN queries by maintaining the priority queue using the MBRs for the children nodes of the cached level.

4.6 Security Analysis

The security of the oblivious index structure (oblivious B-tree and R-tree) and the query protocol as proposed in Section 4.2 follows directly from the same security guarantee and analysis as that in the design of oblivious data structure [32]. The security of the ORAM caching introduced in Section 4.4 relies on the two critical facts. One is that the clients and the coordinator are trusted. The other is that the communication between them is secured and not observed by the cloud server. From the server's point of view, he still receives a sequence of requests to read one block at a time and those blocks being read are written back to a randomly chosen path from the Path-ORAM's binary tree structure. In other words, the Path-ORAM protocol is still followed while accessing a sequence of seemingly random blocks.

For batch writing optimization together with partial path retrieval optimization in Section 4.4.2, from the perspective of the cloud, the coordinator still first retrieves λ uniform random paths and then writes these λ paths back to the cloud. The security guarantee and analysis are similar to those for write-back operation in TaoStore [28]. TaoStore also writes in batches of λ paths, and leaks no additional information to normal Path-ORAM, except for value of λ which only pertains to the implementation, not the actual data or queries. Hence, it still satisfies Definition 1.

For security analysis in ORAM caching, the additional sensitive information leaked is only that each ORAM retrieval corresponds to a cache miss in trusted coordinator. But since we do not consider timing attack (see "Security model" part in Section 2.1), as most existing ORAM constructions, such leakage is not a major concern in our setting. Introducing ORAM caching still follows Definition 1.

To be honest, there does exist some security issue regarding query correlation. Suppose we build 5 levels of B-tree index for a sequence of data blocks. If batch 1 makes exact 5 Path-ORAM accesses and batch 2 makes 5X more ORAM accesses than a specific number, the adversary does learn some query correlation information across batches.

Last, since volume leakage from range query may facilitate reconstruction attacks over encrypted databases [44], we also introduce a padding mode, similar to that in Opaque [12] and ObliDB [33], to protect against such volume leakage. A basic approach is to pad the total number of Path-ORAM accesses for queries in each batch to the worst-case number by issuing dummy block requests, which leaks nothing with regard to the queries. Furthermore, some novel padding techniques can be introduced, e.g., exploring differential privacy rather than full obliviousness to reduce the padding number [45], or padding the number of Path-ORAM accesses in each batch to the closest power of a constant x (e.g., 2 or 4) [46], [47], [48], leading to at most $\log_x |R_{\max}|$ distinct numbers, where $|R_{\max}|$ is the worst-case number of Path-ORAM accesses in each batch.

5 Experimental Evaluation

5.1 Datasets and Setup

Basically, we evaluate our method (OQF+Optimization), Baseline (Opaque) in Section 3, ORAM+Index in Section 4.1, and Oblivious Index in Section 4.2. Note that our method uses either ORAM+Index or Oblivious Index. The costs of the two instantiations under (OQF+Optimization) are similar while Oblivious Index needs less coordinator memory.

Shared Scan is an improved approach over Baseline (Opaque). Shared Scan answers each batch of queries all together by leveraging only one single scan operation. During query processing, it keeps the states of all queries in a batch at the same time and shares the retrieved blocks from the scan operation across the queries within that batch.

For one-dimensional range query, we also make an evaluation of disk-based Oblivious AVL Tree. In our implementation, we put consecutive nodes in each level of the original oblivious AVL tree into blocks and make each block still contain B bytes. Our implementation reduces the number of disk seeks, since retrieving one block can help us access $\Theta(B)$ nodes, although the fanout of the tree is still two.

Lastly, we also compare our method with Raw Index. Raw Index builds a B-tree/R-tree index over data blocks and stores all index and data blocks to the cloud *without*

TABLE 2
Datasets

Dataset	# of Points	Raw Data Size
USA	23,947,347	681 MB
Twitter	247,032,130	7.1 GB
OSM_40M	40,000,000	1.1 GB
OSM_200M	200,000,000	5.6 GB
OSM_400M	400,000,000	12 GB
OSM_800M	800,000,000	23 GB
OSM_1600M	1,600,000,000	46 GB

[a] *OSM_XXM is a random sample of the full OSM dataset.*

using any encryption or any ORAM protocol. During query processing, the coordinator performs batch query processing and caching with the *same cache size* as that in our method. The caching strategy is LRU.

We compare these methods on three datasets in our experiments. Statistics on the datasets are given in Table 2.

USA. USA is from the 9th DIMACS Implementation Challenge (Shortest Paths), which contains points on road networks in USA.

Twitter. Twitter dataset is sampled from the geo-locations in tweets collected by us from October to December in 2017.

OSM. OSM (short for OpenStreetMap) is a collaborative project to create a free editable map of the world. The full OSM data contains 2,682,401,763 points in 78 GB.

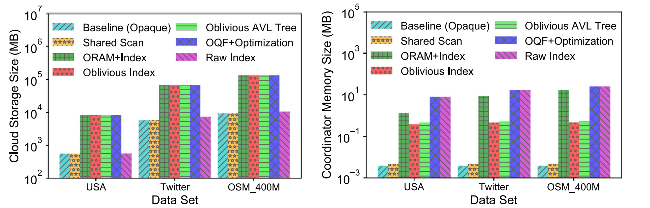
SETUP. We use a Ubuntu 14.04 machine with Intel Core i7 CPU (8 cores, 3.60 GHz) and 18 GB main memory as the coordinator. The cloud server is a Ubuntu 14.04 machine with Intel Xeon E5-2609 CPU (8 cores, 2.40 GHz), 256 GB main memory and 2 TB hard disk. The bandwidth is 1 Gbps.

In our experiments, the cloud server hosts a MongoDB instance as the outsourced storage. We also implement a MongoDB connector class, which supports insertion, deletion and update operations on blocks inside the MongoDB engine. The cloud server supports read and write operations from the coordinator through the basic operations on blocks.

All methods are implemented in C++. AES/CFB from Crypto++ library is adopted as our encryption function in all methods. The key length of AES encryption is 128 bits.

Default Parameter Values. The default values for key parameters are as follows. We set the size of each encrypted block to 4 KB (the same as [11], [19], [26]). We set the number of blocks in each bucket of Path-ORAM to 4 (the same as [11], [23]). We set default cache reserved factor c to 50, which means the threshold of cache size $\tau = c \cdot \log N$ (N is the number of blocks in database). We set default query batch size g (see Section 4.4.1) to 50. We set default batch-write size λ (see "Batch writing" part in Section 4.4.2) to 10.

Query Generation. We generate 2,000 queries for each query type, where each query batch contains g queries. For R-tree query, given the center point of each query batch, a new query point is generated by adding a random offset (no larger than a given *batch locality parameter*) over each dimension of the center point. The default batch locality parameter is 0.05 (for both longitude and latitude dimensions). By default, the range size for each R-tree range query is 0.05×0.05 (longitude dimension$\times$latitude dimension), and $k = 10$ for each R-tree kNN query. A similar procedure works for B-tree range query generation. The only difference is we set the default result size of each B-tree range query to be 1,000.



(a) Cloud storage size. (b) Coordinator memory size.

Fig. 6. Cloud and coordinator storage costs.

Remarks. Ideally, if the coordinator accesses the same number of blocks in the cloud for answering each query, the communication cost between the cloud and the coordinator should be *roughly inversely proportional to* the query throughput for each method. It is confirmed by our experimental results (see Figs. 8, 9, and 10 and Figs. 12, 13, 14, 15, and 16) to some extent. For simplicity, we mainly focus on experimental results for query throughput while brushing lightly over those for communication cost in the following sections.

5.2 Cloud and Coordinator Storage Costs

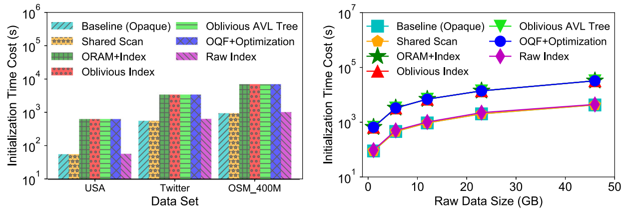
Fig. 6a shows the cloud storage cost in default setting. Baseline (Opaque) and Shared Scan achieve the same and minimum cost, since they only store all encrypted data blocks to the cloud. Raw Index needs a little more cost, since it also builds an index over the data blocks. The other four methods have a similar storage overhead (roughly 10X larger than Baseline (Opaque), Shared Scan and Raw Index), since they all require Path-ORAM data structure on cloud.

Fig. 6b shows the coordinator storage cost. Baseline (Opaque) has the minimum cost, since the coordinator only keeps a constant number of blocks during scan-based operations. Shared Scan needs a little more cost, since it also keeps the parameters and states of all queries in a batch during query processing. Oblivious AVL Tree and Oblivious Index achieve less cost than ORAM+Index, since they integrate position map information into tree nodes to reduce the coordinator memory size. Especially, Oblivious AVL Tree needs a little more private memory than Oblivious Index, since Oblivious AVL Tree has a larger tree height and needs $O(\log N)$ (rather than $O(\log_B N)$) memory to store a traversed tree path. Raw Index and our method have larger private memory sizes (which are set to be the same) than ORAM+Index, since the coordinator keeps an additional ORAM cache with the threshold $c \cdot \log N$.

5.3 Overall Initialization Time Cost

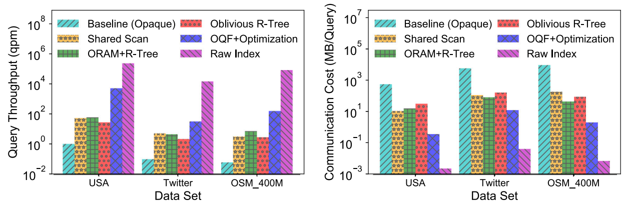
Initializing the original Path-ORAM [23] is very expensive, since each real block insertion pays a Path-ORAM write operation with $O(\log N)$ cost. To avoid the high initialization cost, we pre-build the ORAM data structure in trusted storage and then upload it to the cloud using bulk loading.

In our bulk loading based initialization, the communication overhead and I/O cost of the whole data structure dominate the overall initialization cost, which is roughly proportional to cloud storage cost. Fig. 7 shows the overall initialization time cost of different methods. Baseline (Opaque)



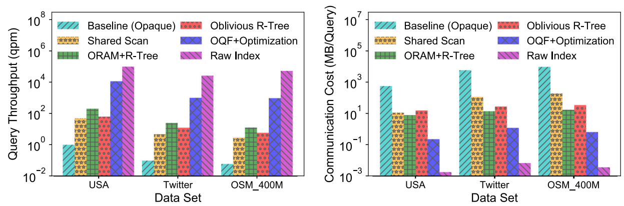
(a) Against different datasets. (b) Against raw data size.

Fig. 7. Overall initialization time cost.



(a) Query throughput. (b) Communication cost.

Fig. 8. Performance of R-tree range query.



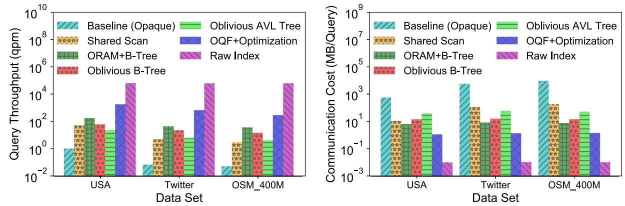
(a) Query throughput. (b) Communication cost.

Fig. 9. Performance of R-tree kNN query.

and Shared Scan have the minimum cost, since they simply store the encrypted data blocks to the cloud. Raw Index needs a little more cost, since it also builds an index over the data blocks. All other four methods have a similar cost (still roughly 10X larger than Baseline (Opaque), Shared Scan and Raw Index), due to building the Path-ORAM data structure. When the raw data size increases from 1.1 to 46 GB, their initialization cost increases from 656 to 32,451 seconds.

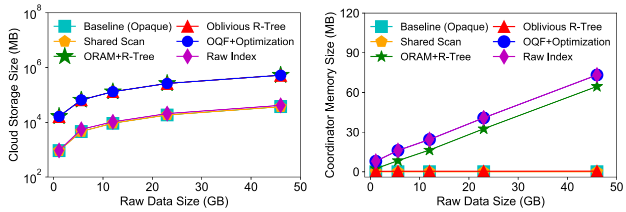
5.4 Query Performance in Default Setting

Fig. 8a shows query throughput for R-tree range query in default setting. The label on y-axis "qpm" is short for "queries per minute". Not surprisingly, Baseline (Opaque) has the lowest query throughput, and Raw Index achieves the largest one. Shared Scan achieves around 50X larger query throughput than Baseline (Opaque). The reason is that Shared Scan leverages only one single scan to answer each batch of queries, while Baseline (Opaque) must scan all the blocks once for each query in the batch. ORAM +Index has roughly 2X larger query throughput than Oblivious Index, since in ORAM+Index the coordinator only performs a `get()` operation through Path-ORAM protocol for each block access, while in Oblivious Index she also performs a `put()` operation for each block access (see Step 4 in Fig. 3). In general, Shared Scan, ORAM+Index and Oblivious Index have comparable performances in terms of query throughput. Our method achieves much larger query throughput than those three methods (by almost one to two orders of magnitude), due to the ORAM caching and other optimizations that we have introduced. Fig. 8b shows the communication cost for R-tree range query in default setting. For each method, the communication cost is roughly inversely proportional to the query throughput.



(a) Query throughput. (b) Communication cost.

Fig. 10. Performance of B-tree range query.



(a) Cloud storage size. (b) Coordinator memory size.

Fig. 11. Storage cost against raw data size.

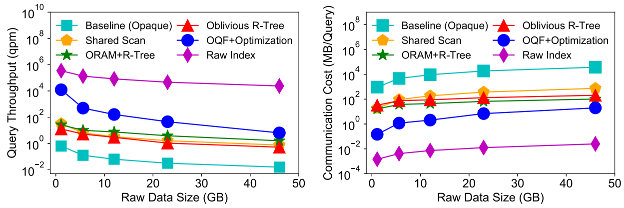
The performances of R-tree kNN query and B-tree range query are shown in Figs. 9 and 10. The trends are similar to those for R-tree range query in Fig. 8. Especially, for B-tree range query (aka one-dimensional range query), Fig. 10 shows that Oblivious Index achieves 2X-4X larger query throughput and less communication cost than Oblivious AVL Tree, due to higher fanout and lower tree height.

5.5 Scalability

We focus on R-tree range query on OSM dataset to report the experimental results regarding scalability.

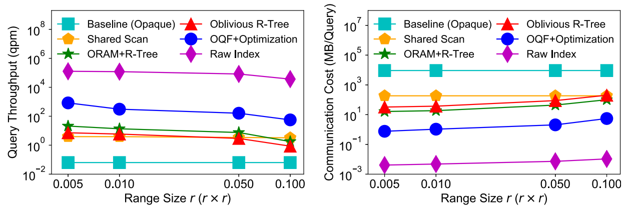
Fig. 11a shows the cloud storage cost against raw data size. Baseline (Opaque) and Shared Scan have the minimum cost, since they simply store all encrypted data blocks to the cloud. Raw Index needs a little more cost, since it also builds an index over the data blocks. When raw data size increases from 1.1 to 46 GB, all other three methods have a similar storage cost, increasing from 16 to 512 GB. Fig. 11b shows the coordinator memory size against raw data size. Baseline (Opaque) still has the minimum cost. Shared Scan has a little more cost, due to storing parameters and states of all queries in each batch. Oblivious Index still has much less cost than ORAM+Index, Raw Index and our method, which increases with the data size *logarithmically, not linearly*. For the other three methods, the cost grows (roughly) linearly with the data size. The reason is that $O(N/B)$ blocks in the position map dominate the coordinator storage when the number of blocks is large. However, since position map entries are small in size, our coordinator storage size only increases from 8 to 73 MB, when raw data size increases from 1.1 to 46 GB. It can be further mitigated if we instantiate our method with oblivious index.

Fig. 12 shows query performance against raw data size. Baseline (Opaque) has the lowest performance, while Raw Index still achieves the best. Our method still achieves 4X-405X larger query throughput and 5X-106X less communication cost than Shared Scan, ORAM+Index



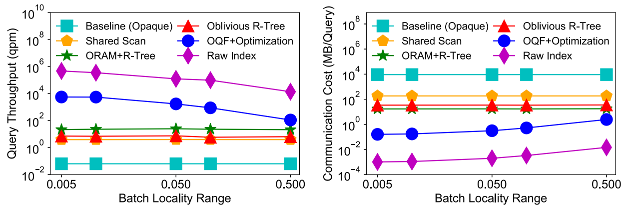
(a) Query throughput. (b) Communication cost.

Fig. 12. Query performance against raw data size.



(a) Query throughput. (b) Communication cost.

Fig. 13. Influence of query selectivity.



(a) Query throughput. (b) Communication cost.

Fig. 14. Influence of query locality.

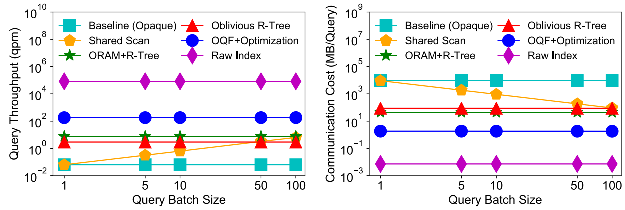
and Oblivious Index, when raw data size varies from 1.1 to 46 GB.

5.6 Selectivity, Locality, Batching, and Caching

We focus on R-tree range query on OSM_400M to report the experimental results regarding selectivity, locality, query batch size g and caching strategy. We also focus on R-tree range query to report results regarding batch-write size λ.

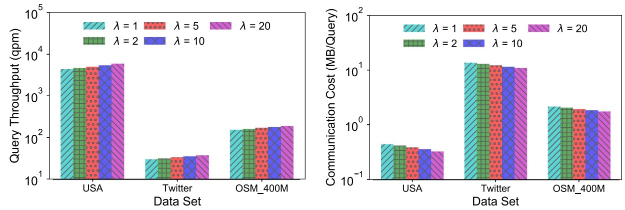
Query Selectivity. Fig. 13 shows query performance against query range size. Baseline (Opaque) has the lowest but stable query performance due to scan-based operations. Shared Scan also has a stable query performance, around 50X better than Baseline (Opaque). Raw Index still achieves the best performance. When range size is small ($\leq 0.01 \times 0.01$), ORAM+Index and Oblivious Index achieve better performance than Shared Scan, due to index searching. When range size varies from 0.005×0.005 to 0.1×0.1, our method achieves 18X-40X larger query throughput and around 20X less communication cost than Shared Scan, ORAM+Index and Oblivious Index.

Query Locality. Fig. 14 shows query performance against batch locality parameter. Baseline (Opaque), Shared Scan, ORAM+Index, and Oblivious Index have a stable query performance, since the coordinator does not perform ORAM caching and cannot take advantage of any locality information. For our method and Raw Index, when the parameter increases, query points in a batch will be distributed more sparsely, which leads to less locality, $<$ *nbw* $>$ *i.e.* $<$ */nbw* $>$, less cache hit rate and less query throughput. When the parameter varies from 0.005 to 0.5, our method achieves 5X-243X larger query



(a) Query throughput. (b) Communication cost.

Fig. 15. Influence of query batch size g.



(a) Query throughput. (b) Communication cost.

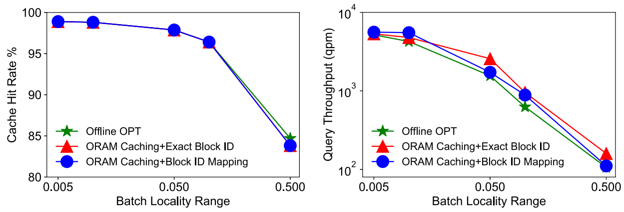
Fig. 16. Influence of batch-write size λ.

throughput and 7X-106X less communication cost than Shared Scan, ORAM+Index and Oblivious Index.

Query Batching. Fig. 15 shows query performance against query batch size g. Baseline (Opaque), ORAM+Index and Oblivious Index have a stable query performance, since these methods do not introduce any optimization in batch processing. Shared Scan achieves roughly g times performance improvement than Baseline (Opaque) when g increases. Raw Index also has a stable query performance, since LRU caching strategy does not benefit from any information in future block accesses, no matter how large g grows. For our method, the performance improvement is very limited when g grows, since the cache size is relatively large in our setting. Hence, a basic LRU caching strategy has achieved very high cache hit rate, and batch-FIF only obtains limited advantage from future block accesses. Fig. 16 shows query performance against batch-write size λ in default setting. When λ increases from 1 to 20, our method achieves 23-35 percent larger query throughput and 19-26 percent less communication cost, due to batch writing and partial path retrieval optimizations.

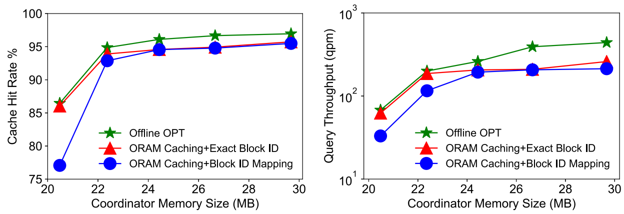
ORAM Caching. Here, we compare the performance of three ORAM caching strategies. Offline OPT is the offline optimal caching strategy ($<$ *nbw* $>$ *i.e.* $<$ */nbw* $>$, FIF algorithm in Section 4.4). ORAM Caching+Exact Block ID is our online algorithm when given the exact block IDs to access in a query batch ($<$ *nbw* $>$ *i.e.* $<$ */nbw* $>$, the online batch-FIF algorithm in Section 4.4), which shows the ideal case of our ORAM caching strategy. ORAM Caching+Block ID Mapping is the same online ORAM caching strategy but now working with query to block ID mapping as described in Section 4.5. In all three caching strategies, the coordinator keeps an cache with the same threshold of cache size.

Fig. 17 shows query performance against query locality with default cache size threshold. The three caching strategies have comparable cache hit rate and query throughput in our block access sequence. When locality parameter is below 0.1, the cache hit rate is above 96 percent and query throughput is above 620 qpm for all caching strategies. Fig. 18 shows query performance against cache size. Both cache hit rate and query throughput have Offline OPT $>$ ORAM Caching



(a) Cache hit rate. (b) Query throughput.

Fig. 17. ORAM caching strategy against query locality.



(a) Cache hit rate. (b) Query throughput.

Fig. 18. ORAM caching strategy against cache size.

+Exact Block ID $>$ ORAM Caching+Block ID Mapping. When private memory size is below 22 MB, ORAM Caching +Block ID Mapping only has 1.6X-1.8X less query throughput than ORAM Caching+Exact Block ID, which demonstrates the effectiveness of our query to block ID mapping strategy under a small cache size. When private memory size is up to 24 MB, the cache hit rate is above 94 percent and query throughput is above 190 qpm for ORAM Caching+Block ID Mapping. Fig. 19a shows the communication cost against cache size, which has Offline OPT $<$ ORAM Caching+Exact Block ID $<$ ORAM Caching+Block ID Mapping. When private memory size is up to 22 MB, the communication cost of ORAM Caching+Block ID Mapping is below 3 MB/Query.

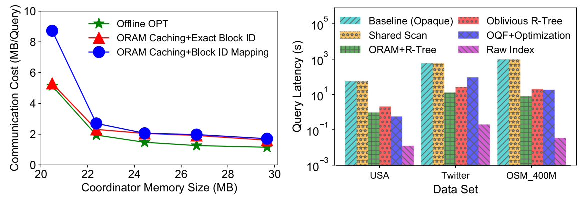
5.7 Query Latency

Lastly, Fig. 19b shows query latency for R-tree range query in default setting. For Baseline (Opaque), ORAM+Index and Oblivious Index, the query latency is roughly proportional to communication cost, since they all process incoming queries synchronously and sequentially. Shared Scan has roughly the same query latency with Baseline (Opaque), since the query results of each query in a batch are not fully generated until the scan operation for that batch is completed. For our method and Raw Index, the coordinator needs to re-order the queries in a batch to improve query throughput, which in fact hurts query latency to some extent. But our method still has comparable query latency with ORAM+Index and Oblivious Index.

6 Related Work

Generic ORAMs. ORAMs allow the client to access encrypted data in a remote server while hiding her access patterns. For detailed analysis on various ORAM constructions, please refer to recent work [11]. However, most ORAM constructions are not suitable for the multi-user scenario, since they handle operation requests *synchronously in a sequential fashion*. Hence, the system throughput is seriously limited.

Range ORAMs [46], [47] are well-designed ORAMs to specifically support range queries. To minimize the number of disk seeks, they take advantage of data locality information and access ranges of sequentially logical blocks.



(a) (b)

Fig. 19. (a) Communication cost against cache size. (b) R-tree range query latency.

However, range ORAMs need much larger cloud storage cost, since they must deploy $O(\log N)$ separate sub-ORAMs. They also bring much more bandwidth overhead and I/O cost in bytes, although they achieve a less number of disk seeks. Besides, they are only suitable for key-value stores but do not work for relational tables with multiple columns.

There exist more advanced ORAM constructions, such as PrivateFS [24], Shroud [25], ObliviStore [26], CURIOUS [27] and TaoStore [28]. They focus on building oblivious file systems, supporting multiple clients, enabling parallelization, supporting asynchronous operations and building distributed ORAM data stores. In other words, those constructions above focus on achieving *operation-level* parallelism or asynchronicity. In contrast, our OQF focuses on improving *query-level* throughput where each query consists of multiple operations in a sequence. Hence, those constructions are orthogonal to our study. OQF can use such a construction (e.g., TaoStore) as the secure ORAM storage on the cloud.

Recent studies also investigate how to support the ORAM primitive more efficiently inside the architecture design of new memory technologies (e.g., [49]). Our design of OQF can benefit from these hardware implementations.

Oblivious Query Processing. Oblivious query processing techniques for specific types of queries have also been explored. Li *et al.* [29] study how to compute theta-joins obliviously. Arasu *et al.* [13] design oblivious algorithms in theory for a rich class of SQL queries, and Krastnikov *et al.* [30] improve their oblivious binary equi-join algorithm. Xie *et al.* [19] propose ORAM based solutions to perform shortest path computation and achieve performance improvement on private information retrieval (PIR) based solutions [50], [51]. ZeroTrace [43] is a new library of oblivious memory primitives, combining ORAM techniques with SGX. However, it only performs basic get/put/insert operations over Set/Dictionary/List interfaces. Obladi [52] is the first system to provide oblivious ACID transactions. The contribution is orthogonal to our study.

To the best of our knowledge, Opaque [12] and ObliDB [33] are the state-of-the-art studies concerning *generic oblivious analytical processing*. We have compared with Opaque (without the distributed storage) and ObliDB (similar to Oblivious Index baseline) in Section 5 and achieved an order of magnitude speedup in query throughput. Lastly, as we point out in "Remarks" part of Section 2.1, the coordinator in OQF can be replaced with an enclave from SGX [39] on cloud, which eliminates the need for a trusted coordinator.

Oblivious Data Structures. Prior studies [14], [32], [53] also design oblivious data structures. Wang *et al.* [32] apply pointer-based and locality-based techniques to some commonly-used data structures (e.g., binary search trees). In this work, we extend their construction and propose oblivious B-tree and oblivious R-tree. Hoang *et al.* [14] propose some new oblivious data structures including Oblivious Tree Structure (OTREE). However, OTREE only works for *binary tree structures* but cannot be extended for larger fanout (e.g., in B-tree and R-tree). Oblix [36] builds an oblivious sorted multimap (OSM) based on oblivious AVL tree [32] and supports queries over $\langle$key, sorted list of values $\rangle$ pairs. ObliDB [33] exploits indexed storage method and builds oblivious B+ trees to support point and range queries. In their implementation, data is fixed to one record per block. But in our implementation of oblivious B-tree in Section 4.2, each block contains B bytes, and the number of records that fit in each data block is $\Theta(B)$ rather than one. Hence, our design is more suitable for hard disk storage and reduces the number of disk seeks in query processing.

Private Index. Existing work [9], [54], [55], [56] also designs *specialized private index* to support some specific types of queries including secure nearest neighbor query and kNN query. Hu *et al.* [57] devise secure protocols for point query on B-tree and R-tree. However, their method works for two-party model where the client owns the query and the cloud server owns the data, which is different from our model.

A number of searchable indices [58], [59], [60], [61], [62], [63] are also proposed to support range query over encrypted data using searchable encryptions. However, those searchable indices cannot protect query access patterns.

Secure Multi-Party Computation. Some recent work explores building an ORAM for secure multi-party computation (MPC) [64], [65]. MPC is a powerful cryptographic primitive that allows multiple parties to perform rich data analytics over their private data, while no party can learn the data from another party. Hence, MPC-based solutions [64], [65], [66], [67], [68] have a different problem setting from our cloud database setting and we do not evaluate them in our study.

Differential Privacy. Differential privacy (DP) is an effective model to protect against unknown attacks with guaranteed probabilistic accuracy. Existing DP-based solutions build key-value data collection [69], build index for range query [70] or support general SQL queries [45], [71]. In brief, DP-based solutions [45], [69], [70], [71], [72], [73], [74], [75], [76] provide *differential privacy for query results*, while our setting is to answer queries *exactly*.

7 Conclusion

This paper proposes an oblivious query framework (OQF). We investigate different instantiations of an OQF and demonstrate a design that is practical, efficient, and scalable. Our design introduces ORAM caching and other optimizations and integrates these optimizations with oblivious indices like oblivious B-tree and oblivious R-tree. Extensive experimental evaluation has demonstrated the superior efficiency and scalability of the proposed design when being compared against other alternatives and state-of-the-art baselines that exist in the literature. Our investigation focuses on range and kNN queries, however, the proposed framework is generic enough and can be extended to handle other query types (e.g., joins), which is an active ongoing work. The current design does not address challenges associated with ad-hoc updates, which is another future direction to explore.

Acknowledgments

Thanks for NSF CCF-1350888, ACI-1443046, CNS-1514520, CNS-1564287, CNS-1718834, IIS-1816149, CDS&E-1953350, and the support from Visa Research and Microsoft Research PhD Fellowship.

References

[1] A. Arasu, K. Eguro, M. Joglekar, R. Kaushik, D. Kossmann, and R. Ramamurthy, "Transaction processing on confidential data using Cipherbase," in *Proc. IEEE Int. Conf. Data Eng.*, 2015, pp. 435–446.

[2] A. Arasu *et al.*, "Secure database-as-a-service with Cipherbase," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 1033–1036.

[3] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan, "CryptDB: Protecting confidentiality with encrypted query processing," in *Proc. ACM Symp. Operating Syst. Princ.*, 2011, pp. 85–100.

[4] S. Bajaj and R. Sion, "TrustedDB: A trusted hardware-based database with privacy and data confidentiality," *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 3, pp. 752–765, Mar. 2014.

[5] Z. He *et al.*, "SDB: A secure query processing system with data interoperability," *Proc. VLDB Endowment*, vol. 8, no. 12, pp. 1876–1879, 2015.

[6] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich, "Processing analytical queries over encrypted data," *Proc. VLDB Endowment*, vol. 6, no. 5, pp. 289–300, 2013.

[7] A. Arasu, K. Eguro, R. Kaushik, and R. Ramamurthy, "Querying encrypted data," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2014, pp. 1259–1261.

[8] H. Hacigümüs, B. R. Iyer, C. Li, and S. Mehrotra, "Executing SQL over encrypted data in the database-service-provider model," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2002, pp. 216–227.

[9] B. Yao, F. Li, and X. Xiao, "Secure nearest neighbor revisited," in *Proc. IEEE 29th Int. Conf. Data Eng.*, 2013, pp. 733–744.

[10] W. K. Wong, B. Kao, D. W. Cheung, R. Li, and S. Yiu, "Secure query processing with data interoperability in a cloud database environment," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2014, pp. 1395–1406.

[11] Z. Chang, D. Xie, and F. Li, "Oblivious RAM: A dissection and experimental evaluation," *Proc. VLDB Endowment*, vol. 9, no. 12, pp. 1113–1124, 2016.

[12] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Opaque: An oblivious and encrypted distributed analytics platform," in *Proc. USENIX Conf. Netw. Syst. Des. Implementation*, 2017, pp. 283–298.

[13] A. Arasu and R. Kaushik, "Oblivious query processing," in *Proc. Int. Conf. Database Theory*, 2014, pp. 26–37.

[14] T. Hoang, C. D. Ozkaptan, G. A. Hackebeil, and A. A. Yavuz, "Efficient oblivious data structures for database services on the cloud," *IACR Cryptol. ePrint Arch.*, vol. 2017, pp. 1238, 2017. [Online]. Available: http://eprint.iacr.org/2017/1238

[15] M. S. Islam, M. Kuzu, and M. Kantarcioglu, "Access pattern disclosure on searchable encryption: Ramification, attack and mitigation," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2012.

[16] P. Kocher *et al.*, "Spectre attacks: Exploiting speculative execution," in *Proc. IEEE Symp. Secur. Privacy*, 2019, pp. 1–19.

[17] O. Goldreich, "Towards a theory of software protection and simulation by oblivious RAMs," in *Proc. Annu. ACM Symp. Theory Comput.*, 1987, pp. 182–194.

[18] R. Ostrovsky, "Efficient computation on oblivious RAMs," in *Proc. Annu. ACM Symp. Theory Comput.*, 1990, pp. 514–523.

[19] D. Xie *et al.*, "Practical private shortest path computation based on oblivious storage," in *Proc. IEEE 32nd Int. Conf. Data Eng.*, 2016, pp. 361–372.

[20] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious RAMs," *J. ACM*, vol. 43, no. 3, pp. 431–473, 1996.

[21] E. Stefanov, E. Shi, and D. X. Song, "Towards practical oblivious RAM," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2012.

[22] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li, "Oblivious RAM with $O((\log N)^3)$ worst-case cost," in *Proc. Int. Conf. Theory Appl. Cryptol. Inf. Secur.*, 2011, pp. 197–214.

[23] E. Stefanov *et al.*, "Path ORAM: An extremely simple oblivious RAM protocol," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2013, pp. 299–310.

[24] P. Williams, R. Sion, and A. Tomescu, "PrivateFS: A parallel oblivious file system," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2012, pp. 977–988.

[25] J. R. Lorch, B. Parno, J. W. Mickens, M. Raykova, and J. Schiffman, "Shroud: Ensuring private access to large-scale data in the data center," in *Proc. USENIX Conf. File Storage Technol.*, 2013, pp. 199–214.

[26] E. Stefanov and E. Shi, "ObliviStore: High performance oblivious cloud storage," in *Proc. IEEE Symp. Secur. Privacy*, 2013, pp. 253–267.

[27] V. Bindschaedler, M. Naveed, X. Pan, X. Wang, and Y. Huang, "Practicing oblivious access on cloud storage: The gap, the fallacy, and the new way forward," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2015, pp. 837–849.

[28] C. Sahin, V. Zakhary, A. E. Abbadi, H. Lin, and S. Tessaro, "TaoStore: Overcoming asynchronicity in oblivious data storage," in *Proc. IEEE Symp. Secur. Privacy*, 2016, pp. 198–217.

[29] Y. Li and M. Chen, "Privacy preserving joins," in *Proc. IEEE 24th Int. Conf. Data Eng.*, 2008, pp. 1352–1354.

[30] S. Krastnikov, F. Kerschbaum, and D. Stebila, "Efficient oblivious database joins," *Proc. VLDB Endowment*, vol. 13, no. 11, pp. 2132–2145, 2020.

[31] K. Mouratidis and M. L. Yiu, "Shortest path computation with no information leakage," *Proc. VLDB Endowment*, vol. 5, no. 8, pp. 692–703, 2012.

[32] X. S. Wang *et al.*, "Oblivious data structures," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2014, pp. 215–226.

[33] S. Eskandarian and M. Zaharia, "ObliDB: Oblivious query processing for secure databases," *Proc. VLDB Endowment*, vol. 13, no. 2, pp. 169–183, 2019.

[34] B. Pinkas and T. Reinman, "Oblivious RAM revisited," in *Proc. Annu. Cryptol. Conf.*, 2010, pp. 502–519.

[35] C. W. Fletcher, L. Ren, X. Yu, M. van Dijk, O. Khan, and S. Devadas, "Suppressing the oblivious RAM timing channel while making information leakage and program efficiency trade-offs," in *Proc. IEEE 20th Int. Symp. High Perform. Comput. Archit.*, 2014, pp. 213–224.

[36] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa, "Oblix: An efficient oblivious search index," in *Proc. IEEE Symp. Secur. Privacy*, 2018, pp. 279–296.

[37] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang, "Detecting privileged side-channel attacks in shielded execution with Déjà Vu," in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, 2017, pp. 7–18.

[38] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa, "Strong and efficient cache side-channel protection using hardware transactional memory," in *Proc. USENIX Conf. Secur. Symp.*, 2017, pp. 217–233.

[39] T. Kim, Z. Lin, and C. Tsai, "CCS'17 Tutorial Abstract: SGX security and privacy," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 2613–2614.

[40] R. Motwani and P. Raghavan, *Randomized Algorithms*. Cambridge, U.K.: Cambridge Univ. Press, 1995.

[41] L. Ren *et al.*, "Constants count: Practical improvements to oblivious RAM," in *Proc. USENIX Conf. Secur. Symp.*, 2015, pp. 415–430.

[42] M. Maas *et al.*, "PHANTOM: Practical oblivious computation in a secure processor," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2013, pp. 311–324.

[43] S. Sasy, S. Gorbunov, and C. W. Fletcher, "ZeroTrace: Oblivious memory primitives from Intel SGX," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2018.

[44] P. Grubbs, M. Lacharité, B. Minaud, and K. G. Paterson, "Pump up the volume: Practical database reconstruction from volume leakage on range queries," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 315–331.

[45] J. Bater, X. He, W. Ehrich, A. Machanavajjhala, and J. Rogers, "Shrinkwrap: Efficient SQL query processing in differentially private data federations," *Proc. VLDB Endowment*, vol. 12, no. 3, pp. 307–320, 2018.

[46] A. Chakraborti, A. J. Aviv, S. G. Choi, T. Mayberry, D. S. Roche, and R. Sion, "rORAM: Efficient range ORAM with $O(\log^2 N)$ locality," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2019.

[47] G. Asharov, T. H. Chan, K. Nayak, R. Pass, L. Ren, and E. Shi, "Locality-preserving oblivious RAM," in *Proc. Annu. Int. Conf. Theory Appl. Cryptogr. Techn.*, 2019, pp. 214–243.

[48] I. Demertzis, D. Papadopoulos, C. Papamanthou, and S. Shintre, "SEAL: Attack mitigation for encrypted databases via adjustable leakage," in *Proc. USENIX Secur. Symp.*, 2020, pp. 2433–2450.

[49] A. Shafiee, R. Balasubramonian, M. Tiwari, and F. Li, "Secure DIMM: Moving ORAM primitives closer to memory," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2018, pp. 428–440.

[50] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan, "Private information retrieval," *J. ACM*, vol. 45, no. 6, pp. 965–981, 1998.

[51] P. Williams and R. Sion, "Usable PIR," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2008.

[52] N. Crooks, M. Burke, E. Cecchetti, S. Harel, R. Agarwal, and L. Alvisi, "Obladi: Oblivious serializable transactions in the cloud," in *Proc. 13th USENIX Conf. Operating Syst. Des. Implementation*, 2018, pp. 727–743.

[53] M. Keller and P. Scholl, "Efficient, oblivious data structures for MPC," in *Proc. Int. Conf. Theory Appl. Cryptol. Inf. Secur.*, 2014, pp. 506–525.

[54] S. Papadopoulos, S. Bakiras, and D. Papadias, "Nearest neighbor search with strong location privacy," *Proc. VLDB Endowment*, vol. 3, no. 1, pp. 619–629, 2010.

[55] X. Yi, R. Paulet, E. Bertino, and V. Varadharajan, "Practical k nearest neighbor queries with location privacy," in *Proc. IEEE 30th Int. Conf. Data Eng.*, 2014, pp. 640–651.

[56] Y. Elmehdwi, B. K. Samanthula, and W. Jiang, "Secure k-nearest neighbor query over encrypted data in outsourced environments," in *Proc. IEEE 30th Int. Conf. Data Eng.*, 2014, pp. 664–675.

[57] H. Hu, J. Xu, X. Xu, K. Pei, B. Choi, and S. Zhou, "Private search on key-value stores with hierarchical indexes," in *Proc. IEEE 30th Int. Conf. Data Eng.*, 2014, pp. 628–639.

[58] R. Li, A. X. Liu, A. L. Wang, and B. Bruhadeshwar, "Fast range query processing with strong privacy protection for cloud computing," *Proc. VLDB Endowment*, vol. 7, no. 14, pp. 1953–1964, 2014.

[59] R. Li and A. X. Liu, "Adaptively secure conjunctive query processing over encrypted data for cloud computing," in *Proc. IEEE 33rd Int. Conf. Data Eng.*, 2017, pp. 697–708.

[60] I. Demertzis, S. Papadopoulos, O. Papapetrou, A. Deligiannakis, and M. N. Garofalakis, "Practical private range search revisited," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2016, pp. 185–198.

[61] P. Karras, A. Nikitin, M. Saad, R. Bhatt, D. Antyukhov, and S. Idreos, "Adaptive indexing over encrypted numeric data," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2016, pp. 171–183.

[62] C. Horst, R. Kikuchi, and K. Xagawa, "Cryptanalysis of comparable encryption in SIGMOD'16," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2017, pp. 1069–1084.

[63] I. Demertzis and C. Papamanthou, "Fast searchable encryption with tunable locality," in *Proc. ACM Int. Conf. Manage. Data*, 2017, pp. 1053–1067.

[64] X. S. Wang, Y. Huang, T.-H. H. Chan, A. Shelat, and E. Shi, "SCORAM: Oblivious RAM for secure computation," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2014, pp. 191–202.

[65] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi, "ObliVM: A programming framework for secure computation," in *Proc. IEEE Symp. Secur. Privacy*, 2015, pp. 359–376.

[66] J. Bater, G. Elliott, C. Eggen, S. Goel, A. N. Kho, and J. Rogers, "SMCQL: Secure query processing for private data networks," *Proc. VLDB Endowment*, vol. 10, no. 6, pp. 673–684, 2017.

[67] N. Volgushev, M. Schwarzkopf, B. Getchell, M. Varia, A. Lapets, and A. Bestavros, "Conclave: Secure multi-party computation on big data," in *Proc. Eur. Conf. Comput. Syst.*, 2019, pp. 3:1–3:18.

[68] A. Dave, C. Leung, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Oblivious coopetitive analytics using hardware enclaves," in *Proc. Eur. Conf. Comput. Syst.*, 2020, pp. 39:1–39:17.

[69] Q. Ye, H. Hu, X. Meng, and H. Zheng, "PrivKV: Key-value data collection with local differential privacy," in *Proc. IEEE Symp. Secur. Privacy*, 2019, pp. 317–331.

[70] C. Sahin, T. Allard, R. Akbarinia, A. E. Abbadi, and E. Pacitti, "A differentially private index for range query processing in clouds," in *Proc. IEEE 34th Int. Conf. Data Eng.*, 2018, pp. 857–868.

[71] N. M. Johnson, J. P. Near, and D. Song, "Towards practical differential privacy for SQL queries," *Proc. VLDB Endowment*, vol. 11, no. 5, pp. 526–539, 2018.

[72] R. Chen, H. Li, A. K. Qin, S. P. Kasiviswanathan, and H. Jin, "Private spatial data aggregation in the local setting," in *Proc. IEEE 32nd Int. Conf. Data Eng.*, 2016, pp. 289–300.

[73] T. Wang, J. Blocki, N. Li, and S. Jha, "Locally differentially private protocols for frequency estimation," in *Proc. USENIX Conf. Secur. Symp.*, 2017, pp. 729–745.

[74] G. Cormode, S. Jha, T. Kulkarni, N. Li, D. Srivastava, and T. Wang, "Privacy at scale: Local differential privacy in practice," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2018, pp. 1655–1658.

[75] N. Wang *et al.*, "Collecting and analyzing multidimensional data with local differential privacy," in *Proc. IEEE 35th Int. Conf. Data Eng.*, 2019, pp. 638–649.

[76] T. Wang *et al.*, "Answering multi-dimensional analytical queries under local differential privacy," in *Proc. Int. Conf. Manage. Data*, 2019, pp. 159–176.