Efficient and Oblivious Query Processing for Range and kNN Queries

(Extended Abstract)

Zhao Chang[1], Dong Xie[2], Feifei Li[3], Jeff M. Phillips[4], Rajeev Balasubramonian[4]
[1]*Xidian University, China,* [2]*The Pennsylvania State University, USA,*
[3]*Alibaba Group, China,* [4]*University of Utah, USA*
changzhao@xidian.edu.cn, dongx@psu.edu, lifeifei@alibaba-inc.com, jeffp@cs.utah.edu, rajeev@cs.utah.edu

***Abstract*—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. In this work, 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). Our construction shows an order of magnitude speedup in comparison with other baselines over large datasets.**

I. Introduction

A necessary step for keeping sensitive information secure and private on the cloud is to encrypt the data. But query access patterns over an encrypted database can still pose a threat to data privacy and leak sensitive information. Oblivious RAM (ORAM) is proposed to protect the client's access patterns from the cloud. However, most ORAMs are very expensive and not suitable for deployment in a database. Some advanced ORAMs handle operations (to read or write blocks) *asynchronously* and achieve *operation-level concurrency* at the storage level. However, they do not support *query-level concurrency*, since each query (*e.g.*, a range or a kNN query) consists of a sequence of operations, and any incoming query request is not processed until a prior ongoing query has been completed. It creates a serious bottleneck under concurrent workload in handling multiple clients.

Opaque [1] and ObliDB [2] are two novel studies concerning *generic oblivious analytical processing*. However, Opaque [1] needs to perform expensive scan-based operations to support kNN or range queries. ObliDB [2] exploits indexed storage method to support range queries without any optimizations. In this work, we propose a general oblivious query processing framework (OQF) for cloud databases, which supports concurrent query processing (*i.e.*, concurrency within a query's processing) with high throughput. We focus on one and multi-dimensional range and kNN queries. The key idea is to integrate indices into ORAM and also leverage a suite of optimization techniques including batch processing and ORAM caching. In comparison with baseline methods, our design achieves an order of magnitude speedup in terms of query throughput. Please refer to [3] for the details.

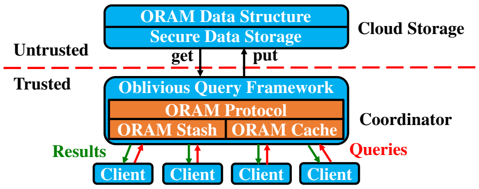


Fig. 1: Oblivious query framework.

II. Problem Formulation and Framework

Consider a client (aka data owner) who stores her data on a remote cloud server and asks other clients (including herself) to issue (range and kNN) queries. A trusted coordinator collects queries from different clients. 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. Our goal is to prevent the server from inferring the query behavior of clients by observing access locality from the index structure and the data itself. Our problem is defined in Definition 1.

Definition 1. Oblivious Query Processing. *Given an input query sequence* $\vec{q} = \{(\mathrm{op}_1, \mathrm{arg}_1), (\mathrm{op}_2, \mathrm{arg}_2), \cdots, (\mathrm{op}_m, \mathrm{arg}_m)\}$*, 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.*

The oblivious query framework is shown in Figure 1. In a preprocessing step, the data owner partitions records into blocks, encrypts these data blocks, and builds an ORAM data structure (*e.g.*, Path-ORAM [4]) over them. She then uploads the ORAM data structure to the cloud storage and shares the encryption keys and other metadata (*e.g.*, position map in Path-ORAM) with the coordinator. Subsequently, clients may issue (range and kNN) queries against the server through the coordinator. The coordinator reads/writes blocks from/to the server based on the ORAM protocol and generates query results for the clients using an oblivious query algorithm.

2375-026X/22/$31.00 ©2022 IEEE
DOI 10.1109/ICDE53745.2022.00120

III. Efficient OQF

We integrate an index ($e.g.$, B-tree or R-tree) into the ORAM before uploading the ORAM data structure to the cloud (denoted as ORAM+Index). 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.

Another approach is to build an oblivious data structure [5], which eliminates the need of storing the position map at the coordinator. The main idea is that each node in the index keeps the position tags and block IDs of its children nodes. When retrieving a node through ORAM, we have acquired its children position tags simultaneously. In our design, we replace the standard B-tree or R-tree above with an oblivious B-tree or R-tree (denoted as Oblivious Index).

To improve the overall query throughput, the coordinator retrieves blocks from the cloud by leveraging batch processing and ORAM caching. In detail, the coordinator keeps an ORAM cache with at most τ blocks. If there is a buffer hit for a subsequent block request, the coordinator does not need to retrieve that block from the cloud again through ORAM.

Given s query batches $\{(q_{1,1}, \cdots, q_{1,g}), \cdots, (q_{s,1}, \cdots, q_{s,g})\}$, the ith batch needs to retrieve a set of m_i blocks with IDs $\{\mathrm{id}_{i,1}, \cdots, \mathrm{id}_{i,m_i}\}$ that will be accessed by $(q_{i,1}, \cdots, q_{i,g})$. We also let $m = \min\{m_1, \cdots, m_s\}$. Our objective is to minimize the number of cache misses over the s batches.

Offline optimal strategy. In offline setting, the coordinator knows block IDs from all (future) query batches. The offline optimal algorithm is Farthest In Future (FIF). It means when there is a cache miss, the coordinator will evict the block in the cache that will not be accessed until *farthest in future*.

Online strategy. 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 ρ. Our online strategy is called batch-FIF. When there is a cache miss, the coordinator first evicts the block that will not be accessed within the current batch using LRU strategy. If such a block is not found, the coordinator will evict the block that will not be accessed until *farthest in future within the current batch*.

Theorem 1. *If there are duplicate block IDs within any batch, ρ(batch-FIF) $\leq \tau$ (τ is the buffer size); otherwise,*

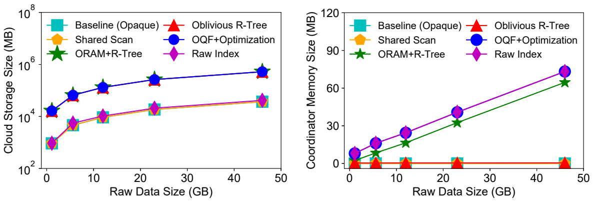
A) If $\tau \leq m$, the competitive ratio ρ(batch-FIF) ≤ 2;

B) Otherwise, the competitive ratio ρ(batch-FIF) $\leq \tau$.

IV. Experimental Evaluation

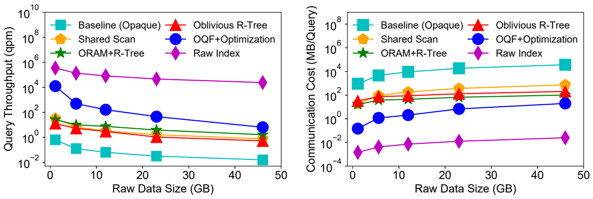
We evaluate our method (OQF+Optimization), Baseline (Opaque) (Opaque [1] without distributed storage), Shared Scan (answering each batch of queries together using one single scan), ORAM+Index, and Oblivious Index (similar to ObliDB [2]). We also compare our method with Raw Index, which builds a B-tree/R-tree index over data blocks on cloud *without using any encryption or any ORAM protocol.*

We focus on R-tree range query on OSM dataset (available at https://www.openstreetmap.org/) to report the experimental results in scalability tests.



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

Fig. 2: Storage cost against raw data size.



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

Fig. 3: Query performance against raw data size.

Figure 2a shows the cloud storage cost. Baseline (Opaque) and Shared Scan simply store encrypted data blocks to the cloud. Raw Index needs a little more cost, due to building an index over the data. The other three methods need roughly 10X larger cost, since they all require Path-ORAM data structure. Figure 2b shows the coordinator memory size. Baseline (Opaque) and Shared Scan only keep a constant number of blocks during scan-based operations. Oblivious Index achieves less coordinator memory size than ORAM+Index, due to integrating position tags into tree nodes. Raw Index and our method have larger private memory sizes (which are set to be the same) than ORAM+Index, since we let the coordinator keep an additional cache.

Figure 3a shows the query throughput. The label on y-axis "qpm" is short for "queries per minute". Baseline (Opaque) has the lowest query throughput, and Raw Index achieves the largest one. In general, Shared Scan, ORAM+Index and Oblivious Index have comparable performances in terms of query throughput. Our method achieves 4X-405X larger query throughput than the three methods, when raw data size varies from 1.1 GB to 46 GB, due to batch processing and ORAM caching. Figure 3b shows communication cost, which is *roughly inversely proportional to* query throughput for each method.

Acknowledgment

Thanks for National Key R&D Program of China (2021YFB3101100), NSFC under grants No. 61972308 and U1736216, and NSF grants 1564287, 1718834, 1816149, and 1953350.

References

[1] 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 *NSDI*, 2017, pp. 283–298.

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

[3] Z. Chang, D. Xie, F. Li, J. M. Phillips, and R. Balasubramonian, "Efficient oblivious query processing for range and knn queries," *IEEE TKDE*, To appear, 2021.

[4] E. Stefanov, M. van Dijk, E. Shi, C. W. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: An extremely simple oblivious RAM protocol," in *CCS*, 2013, pp. 299–310.

[5] X. S. Wang, K. Nayak, C. Liu, T.-H. H. Chan, E. Shi, E. Stefanov, and Y. Huang, "Oblivious data structures," in *CCS*, 2014, pp. 215–226.