Towards Practical Oblivious Join

Zhao Chang
Xidian University
changzhao@xidian.edu.cn

Dong Xie
The Pennsylvania State University
dongx@psu.edu

Sheng Wang, Feifei Li
Alibaba Group
sh.wang@alibaba-inc.com
lifeifei@alibaba-inc.com

ABSTRACT

Many individuals and companies choose the public cloud as their data and IT infrastructure platform. But remote accesses over the data inevitably bring the issue of trust. Despite strong encryption schemes, adversaries can still learn sensitive information from encrypted data by observing data access patterns. Oblivious RAMs (ORAMs) are proposed to protect against access pattern attacks. However, directly deploying ORAM constructions in an encrypted database brings large computational overhead.

In this work, we focus on oblivious joins over a cloud database. Existing studies in the literature are restricted to either primary-foreign key joins or binary equi-joins. Our major contribution is to support general binary and multiway equi-joins. We integrate B-tree indices into ORAMs for each input table and retrieve blocks through the indices in join processing. The key points are to address the security issue (*i.e.*, leaking the number of accesses to any index) in the extended existing solutions and bound the total number of block accesses. Our index nested-loop join algorithm can also support some types of band joins obliviously. The effectiveness and efficiency of our algorithms are demonstrated through extensive evaluations over real-world datasets. Our method shows orders of magnitude speedup for oblivious multiway equi-joins in comparison with baseline algorithms.

CCS CONCEPTS

• **Security and privacy → Management and querying of encrypted data**.

KEYWORDS

oblivious RAM; oblivious index; binary join; multiway join

ACM Reference Format:
Zhao Chang, Dong Xie, and Sheng Wang, Feifei Li. 2022. Towards Practical Oblivious Join. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3514221.3517868

1 INTRODUCTION

Many cloud service providers offer cloud-based database systems such as Amazon RDS and Redshift, Azure SQL, and Google Cloud SQL. A necessary step for keeping sensitive information secure and private on a cloud is to encrypt the data. To that end, encrypted databases such as Cipherbase [8, 9], CryptDB [62], TrustedDB [14], SDB [43], and Monomi [70], as well as related query execution techniques [10, 42, 80, 83] have been developed. But query access patterns still pose a privacy threat and leak sensitive information [11, 23, 44, 87]. It is possible to analyze the importance of different areas in the database, *e.g.*, by counting the frequency of accessing data items [18, 20, 48, 59]. With certain background knowledge, the server learns a lot about user queries and/or data [23, 46, 61].

Oblivious RAMs (ORAMs) [37, 38, 60] allow the client to access encrypted data on a server without revealing her access patterns. However, most ORAM constructions are still too expensive to be deployed in a large database [23]. Recent studies [44, 45, 49, 58, 67, 76] also explore building oblivious data structures or indices over encrypted data, but none of them support complex queries (*e.g.*, joins). The key point is that ORAM does not protect *the number of block accesses* inherently for a general query operator. Hence, all extended existing solutions to integrating indices into ORAMs do leak *the number of accesses to any index* in processing. We will address the security issue in our algorithms in Sections 5 and 6.

Joins are commonly used operations in relational databases. The following two applications motivate the research of privacy preserving joins. One application is to check if any airline passengers are on the watch list of a federal agency for national security [51]. Privacy preserving joins help to find only those passengers who are on the list, without obtaining personal information about any other passengers from the airline or revealing the watch list [5]. The other application is to find out the correlation between a reaction to a drug and some DNA sequence for medical research [5]. It joins DNA information from a gene bank with patient records from various hospitals, and requires accessing only the matching sequences from the gene bank and not disclosing the patient information [6].

In this work, we consider the problem of computing join functions *in an oblivious way*. Li and Chen [54] first studies how to compute theta-joins obliviously, but all of their algorithms are no better than a Cartesian product. Arasu and Kaushik [11] presents oblivious query processing algorithms for a rich class of database queries including joins. However, Krastnikov *et al*. [53] points out that the details in [11] are incomplete, and no practical implementation is provided to show the empirical results. Opaque [87] and ObliDB [35] are efficient *only* for the special case of one-to-many equi-join, *e.g.*, primary-foreign key join. It is still unknown how to extend the Opaque join algorithm [35, 87] to support general oblivious many-to-many equi-join. Krastnikov *et al*. [53] proposes a novel oblivious algorithm for general binary equi-joins. However, it is non-trivial to extend their algorithm to join over multiple tables obliviously. A series of oblivious binary joins will disclose the intermediate table sizes, which may leak some sensitive information, *e.g.*, the data distribution or the sparseness of the intermediate

SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA.

ACM ISBN 978-1-4503-9249-5/22/06…$15.00
https://doi.org/10.1145/3514221.3517868

join graph. ObliDB [35] offers an oblivious hash join algorithm to support general equi-joins over multiple tables, but it is equivalent to a Cartesian product and not a practical solution.

In summary, prior studies are still unable to address the major challenge in oblivious joins. They are only efficient for primary-foreign key joins [35, 87], or restricted to binary equi-joins [53], or purely theoretical in nature and not leading to practical implementations [11, 54].

Our major objective is to support general binary and multiway equi-joins efficiently. First, we propose two oblivious algorithms for general binary equi-joins: sort-merge join and index nested-loop join. We integrate B-tree indices into ORAMs for input tables and retrieve blocks through indices obliviously to perform our algorithms. The key point is to address the security issue (*i.e.*, leaking the number of accesses to any index) in extended existing solutions. Our index nested-loop join algorithm can also support some types of band joins obliviously. Furthermore, we extend the index nested-loop join to support multiway equi-joins obliviously. The key idea is to avoid retrieving tuples that make no contribution to the final join result to bound the total number of block accesses. Note that ORAM scheme can be viewed as a blackbox, providing read and write interface, while hiding access patterns. We can introduce some novel ORAM schemes (*e.g.*, [13, 25, 63]) rather than Path-ORAM [69] to improve the performance. We can also leverage other types of indices (*e.g.*, Oblix [58]) rather than B-tree to perform our algorithms, as long as they can support both point and range queries obliviously. Our major contributions are listed as follows.

- We propose two oblivious algorithms for general binary equi-joins: oblivious sort-merge join and oblivious index nested-loop join in Section 5.1 and 5.2. The key point is to bound the number of accesses to each B-tree index and address the security issue in extended existing solutions.
- We support some types of band joins (*e.g.*, "<" and ">") obliviously by extending our index nested-loop join in Section 5.3. Note that existing studies (except [54]) do not work for any non-equi joins.
- We support acyclic equi-joins over multiple tables obliviously using the index nested-loop join in Section 6. The key idea is to avoid retrieving tuples that cannot make any contribution to the final join result, which helps to bound the total number of block accesses.
- We conduct extensive experiments on real-world datasets in Section 9. The results demonstrate a superior performance gain (orders of magnitude speedup for oblivious multiway equi-joins) achieved by our method over baseline algorithms.

2 RELATED WORK

Generic ORAMs. ORAMs allow the client to access encrypted data remotely while hiding access patterns. A detailed survey is given in [23]. We adopt Path-ORAM [69] in our method, due to good performance and simplicity. It can be replaced with some novel ORAM schemes (*e.g.*, [13, 25, 63]) as a blackbox. There exist more advanced ORAM constructions, such as PrivateFS [79], Shroud [56], ObliviStore [68], CURIOUS [19] and TaoStore [65], working on file systems, multiple clients, parallelization, asynchronous operations and distributed data stores. We may leverage them as our secure ORAM storage, since we treat ORAM as a blackbox.

Oblivious query processing. There exists a list of studies [11, 35, 53, 54, 87] working on oblivious joins (see the discussion in Section 1). We follow the same definition as them (see Definition 1), which allows the leakage of input and output sizes. It is different from the fully oblivious join in [29], which only allows the leakage of input sizes. Padding techniques in Section 8 may ease the issue.

A series of studies also pay attention to oblivious query processing. Xie *et al*. [81] proposes ORAM solutions to shortest path computation, for which earlier work explores private information retrieval (PIR) solutions [28, 78]. ZeroTrace [66] provides a new library of oblivious get/put/insert operations over set/dictionary/list interfaces. Obladi [32] is the first system to provide ACID transactions while hiding access patterns. It processes operations in batches but does not support indices. OCQ [33] is a general framework for oblivious coopetitive analytics, which builds on Opaque [87] to execute coopetitive queries in a decentralized manner.

Note that existing solutions [35, 53, 87] rely on Trusted Execution Environments (TEE). However, TEE is orthogonal to oblivious algorithms and has no advantage to the full obliviousness in the untrusted storage. If the server keeps secure enclaves (*e.g.*, Intel SGX [31, 50]), our client can be moved and co-located in the server.

Oblivious data structures. Some prior studies [44, 49, 67, 76] build oblivious tree structures. POSUP [45] explores hardware-supported oblivious indices. Oblix [58] builds an oblivious search index to store multiple values for the same key. However, none of them explore join query over oblivious indices, since they do not protect *how many accesses to the data structure* inherently. In our method, we integrate B-tree indices into ORAMs for input tables and address the security issue above. Some other indices (*e.g.*, Oblix [58]) also work for our method, as long as they support both point and range queries obliviously.

Secure multi-party computation. Recent work also explores protecting access patterns for secure multi-party computation (MPC) [55, 75]. MPC allows multiple parties to perform data analytics over their private data, while no party learns the data from another party. Hence, MPC-based solutions [16, 33, 55, 71, 75, 77] have a different problem setting from our cloud database setting.

Differential privacy. Differential privacy (DP) solutions protect against attacks with guaranteed probabilistic accuracy. They build index for range query [64] and key-value data collection [84], support multi-dimensional analytical queries [72, 74, 82] and general SQL queries [17, 47, 52]. However, DP-based solutions [17, 26, 30, 47, 52, 64, 72–74, 82, 84] provide *differential privacy for query results*, while we provide the *obliviousness in query processing*.

3 PROBLEM OVERVIEW AND DEFINITION

3.1 Overview

The formulation includes a client and a cloud server. The client, who has a small and secure memory, wants to store and later retrieve her data using the large but untrusted cloud storage. In a preprocessing step, the client partitions records in database D into blocks, encrypts these data blocks, and builds an ORAM data structure (*e.g.*, Path-ORAM) over them. Some B-tree indices I can be integrated into the ORAM data structure using ORAM+B-tree or oblivious B-tree (see Section 4.2). Then, the client uploads the ORAM data structure to the secure data storage in the server, and keeps the encryption keys and other metadata (*e.g.*, ORAM stash and position map in

	Join Type [a]				Algorithm	Complexity Analysis [b]		
	BE	BD	ME			Computation Overhead [c]	Cloud Storage	Client Storage
Li and Chen [54]	✓	✓	✓	BE	A1	$\Omega(\prod_{j=1}^{\ell} \lvert T_j \rvert)$	$O(\prod_{j=1}^{\ell} \lvert T_j \rvert)$	$O(1)$
				BD	A2		$O(\lvert T_{\text{in}} \rvert + \lvert T_{\text{out}} \rvert)$	$O(M)$
				ME	A3		$\Omega(\lvert T_{\text{in}} \rvert + \lvert T_{\text{out}} \rvert)$	$O(M)$
Arasu and Kaushik [11]	✓	×	✓	BE ME	Equi-Join	$O((\lvert T_{\text{in}} \rvert + \lvert T_{\text{out}} \rvert) \cdot \log^2(\lvert T_{\text{in}} \rvert + \lvert T_{\text{out}} \rvert))$	$O(\lvert T_{\text{in}} \rvert + \lvert T_{\text{out}} \rvert)$	$O(\log(\lvert T_{\text{in}} \rvert + \lvert T_{\text{out}} \rvert))$
Opaque [87]	×	×	×	PF	Opaque Join	$O((\lvert T_{\text{in}} \rvert + \lvert T_{\text{out}} \rvert) \cdot \log^2((\lvert T_{\text{in}} \rvert + \lvert T_{\text{out}} \rvert)/M))$	$O(\lvert T_{\text{in}} \rvert + \lvert T_{\text{out}} \rvert)$	$O(M)$
ObliDB [35]	✓	×	✓	PF	0-OM Join	$O((\lvert T_{\text{in}} \rvert + \lvert T_{\text{out}} \rvert) \cdot \log^2(\lvert T_{\text{in}} \rvert + \lvert T_{\text{out}} \rvert))$	$O(\lvert T_{\text{in}} \rvert + \lvert T_{\text{out}} \rvert)$	$O(1)$
				BE ME	Hash Join	$O(\prod_{j=1}^{\ell} \lvert T_j \rvert)$	$O(\prod_{j=1}^{\ell} \lvert T_j \rvert)$	$O(M)$
Krastnikov *et al.* [53]	✓	×	×	BE	Binary Join	$O((\lvert T_{\text{in}} \rvert + \lvert T_{\text{out}} \rvert) \cdot \log^2(\lvert T_{\text{in}} \rvert + \lvert T_{\text{out}} \rvert))$	$O(\lvert T_{\text{in}} \rvert + \lvert T_{\text{out}} \rvert)$	$O(1)$
Ours [d]	✓	✓	✓	BE	SMJ	$O((\lvert T_{\text{in}} \rvert + \lvert T_{\text{out}} \rvert) \cdot (\log \lvert T_{\text{in}} \rvert + \log^2((\lvert T_{\text{in}} \rvert + \lvert T_{\text{out}} \rvert)/M)))$	$O(\lvert T_{\text{in}} \rvert + \lvert T_{\text{out}} \rvert)$	$O(\log \lvert T_{\text{in}} \rvert + M + \lvert T_{\text{in}} \rvert / B)$
				BE BD	INLJ(+Cache)	$O((\lvert T_1 \rvert + \lvert T_{\text{out}} \rvert) \cdot (\log \lvert T_1 \rvert + \Delta \log \lvert T_2 \rvert + \log^2((\lvert T_1 \rvert + \lvert T_{\text{out}} \rvert)/M)))$	$O(\lvert T_{\text{in}} \rvert + \lvert T_{\text{out}} \rvert)$	$O(\log \lvert T_{\text{in}} \rvert + M + \lvert T_{\text{in}} \rvert / B)$
				ME	INLJ(+Cache)	$O((\lvert T_{\text{in}} \rvert + \lvert T_{\text{out}} \rvert) \cdot (\log \lvert T_1 \rvert + \Delta \sum_{j=2}^{\ell} \log \lvert T_j \rvert + \log^2((\lvert T_{\text{in}} \rvert + \lvert T_{\text{out}} \rvert)/M)))$	$O(\lvert T_{\text{in}} \rvert + \lvert T_{\text{out}} \rvert)$	$O(\sum_{j=1}^{\ell} \log \lvert T_j \rvert + M + \lvert T_{\text{in}} \rvert / B)$

Table 1: Comparison of oblivious join algorithms.

[a] We denote binary equi-join as BE, band join as BD, acyclic multiway equi-join as ME, primary-foreign key join as PF.

[b] We denote the total size of all input tables as $\lvert T_{\text{in}} \rvert = \sum_{j=1}^{\ell} \lvert T_j \rvert$ and the real join result size as $\lvert T_{\text{out}} \rvert$. For the convenience of complexity analysis, we assume that each block contains $O(1)$ data tuples but $\Theta(B)$ index entries, which is consistent with ObliDB [35]. In our implementation, we allow each block to contain multiple data tuples but have the same block size B.

[c] We assume an oblivious sorting needs $O(n \log^2(n/m))$ time cost, where m is the trusted storage size, as with Table 2 in [35] and Table 1 in [53].

[d] We denote sort-merge join as SMJ, and index nested-loop join as INLJ. We denote the number of outsourced levels in each B-tree index over each input table as Δ. We may leverage recursive Path-ORAM [23, 69] or oblivious B-tree [35] to reduce the $O(\lvert T_{\text{in}} \rvert / B)$ storage cost for position map in the client.

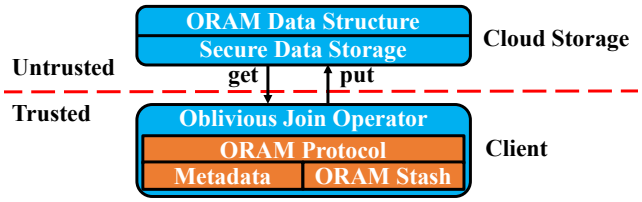


Figure 1: An overview of our oblivious join process.

Path-ORAM) at her side. Note that the client can maintain indices locally, but she may only cache one level of B-tree index to save the memory.

In online processing, the client issues join queries against the server. Using an oblivious join algorithm that will be described later, the client reads/writes blocks from/to the server through an ORAM protocol and generates the query results. The ORAM protocol refers to steps taken to read or write blocks obliviously with the help of ORAM metadata in client and ORAM data structure in server. Our oblivious join algorithm is designed based on the ORAM protocol. An overview of the oblivious join process is shown in Figure 1.

3.2 Problem Definition

We follow the definition in Opaque [87] and ObliDB [35]. Let D be a relational database (where some B-tree indices I may also be integrated) and Q be a join query. Let $\text{Size}(D)$ be the sizing information of D, which includes the size of each table, row, column, attribute, the number of rows and columns, but does not include the value of each attribute. Note that the schema information $\text{Sch}(D)$ including table and column names in D can be easily hidden using encryption. Let $\text{IOSize}(D, Q)$ be the input/output size of running Q over D. Note that for any join query Q over multiple tables in D, the sizes of all intermediate join tables are not included in $\text{IOSize}(D, Q)$ and must be protected against the adversary. Let Trace be the trace of server location accesses and network traffic patterns while running Q over D.

Definition 1. Oblivious Join [87]. *For any two relational databases D and D' and two join queries Q and Q', where $\text{Size}(D) = \text{Size}(D')$, $\text{Sch}(D) = \text{Sch}(D')$ and $\text{IOSize}(D, Q) = \text{IOSize}(D', Q')$, we denote the access patterns produced by the join algorithm* OJoin *running Q and Q' over D and D' as* $\text{Trace}(\text{OJoin}(D, Q))$ *and* $\text{Trace}(\text{OJoin}(D', Q'))$. OJoin *is an oblivious join algorithm, if*

1) OJoin *ensures the confidentiality; and*

2) access patterns $\text{Trace}(\text{OJoin}(D, Q))$ *and* $\text{Trace}(\text{OJoin}(D', Q'))$ *have the same length and computationally indistinguishable for anyone but the client.*

Security model. We consider a "honest-but-curious" server. Data is encrypted, retrieved, and stored in *atomic units* (*i.e.*, blocks). The encryption should be *semantically secure*, and two encrypted copies of the same data block look different. All blocks are of the same size and are indistinguishable for the server. We use N to denote the number of real data blocks in the database, and each encrypted block 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 types of entries, *e.g.*, encrypted index entry, encrypted attribute value, and position tag in ORAM.

Definition 1 does not consider the volume leakage in final output size. Padding techniques in Section 8 may ease the leakage. Definition 1 also does not consider privacy leakage through any side-channel attack (like time taken for each operation). Prior orthogonal solutions [27, 36, 41] can help to alleviate such leakage.

4 PRELIMINARIES

4.1 ORAM and Oblivious Sorting

ORAM. ORAM [37, 38, 60] allows the client to access encrypted data in a remote server while hiding her access patterns. 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. A valid ORAM construction will hide the access patterns with the same length of block operations (*i.e.*, `get()` and `put()`) to make them computationally indistinguishable to the server. It consists of two components: an ORAM data structure and an ORAM query protocol. The client and server run the ORAM query protocol to read and write any data blocks to the ORAM data structure.

Path-ORAM. There are two advantages in Path-ORAM [69]: good performance and simplicity [23]. It organizes the ORAM data structure as a full binary tree where each node is a bucket with a fixed number of encrypted blocks. Path-ORAM maintains *the invariant* that at any time, each block b is always placed in some bucket along the path to the leaf node that b is mapped to. The *stash* in the client stores a few blocks that have not been written back to the binary tree in server. The *position map* keeps track of the mapping between blocks and leaf node IDs, which brings a linear space cost to the client. We may 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 memory size and $O(\log N)$ per query cost. In a recursive Path-ORAM, the client memory size is reduced to $O(\log N)$ and each request can be processed in $O(\log_B N \cdot \log N)$.

Oblivious sorting. A set of items can be sorted by accessing the items in a *fixed, predefined order*. Bitonic sort [15] needs $O(n \log^2 n)$ time cost but with small constant factor. Some advanced algorithms [7, 39, 40] achieves $O(n \log n)$ time cost. However, they may fail with a small probability [39], or lead to large constant factors [7] and non-trivial implementation [40]. Hence, most prior studies [23, 35, 53, 54, 87] still adopt bitonic sort [15]. In particular we extends it to an oblivious external sorting by leveraging a relatively large client memory, as in [35, 87]. The time complexity is $O(n \log^2(n/m))$, where m is trusted storage size.

4.2 Integrate B-Tree Indices into ORAM

ORAM+B-tree. A commonly used optimization in relational database is to build B-tree indices to speed up the query processing. The similar optimization can be introduced in oblivious query processing as that in [24, 35]. The key idea is to *ignore* the semantic difference of the (encrypted) index and data blocks from the client over the database D, and store all these blocks into an ORAM construction, say Path-ORAM. When answering an incoming query, we start with retrieving the root block (of the index) from the server and then traverse down the tree. Intuitively, we query the index structure by running the same algorithm as that over a standard B-tree index. The only difference is that we are retrieving index and data blocks by looking up their block IDs through the ORAM.

Oblivious B-tree. To avoid storing the position map in the client, we can explore the idea of oblivious data structures [44, 58, 76] and replace a standard B-tree in ORAM+B-tree with an oblivious B-tree [24, 35]. The main idea is that each index node keeps the block IDs and position tags of its children nodes. When retrieving any node from the server through the ORAM, we have acquired the position tags 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 client 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.

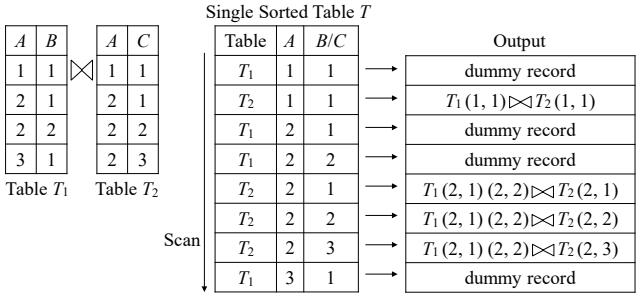


Figure 2: Strawman solution to oblivious sort-merge equi-join.

Index caching. Index caching is a popular tree-based ORAM optimization [57, 63, 66]. We let the client cache *one specific level* of B-tree index to speed up the query performance. Since the fanout is usually large in a B-tree index, this overhead to the client storage is far less than storing the entire index. Given any query, the client can find *which block(s) that she may need to access* by performing a local search on the cached level of the B-tree index.

Segmenting ORAM. We separate one single ORAM into multiple smaller ORAMs to reduce the cost of each ORAM access as in ObliDB [35]. For each input table, we build a Path-ORAM data structure for data blocks and another smaller one for index blocks. The comparison in Table 1 is based on this design. ObliDB [35] even suggests using a separate ORAM for each level of a B-tree index, but does not introduce this into their implementation.

5 OBLIVIOUS BINARY JOIN

We support two types of oblivious binary joins: sort-merge join and index nested-loop join. We integrate B-tree indices into ORAMs for input tables, as shown in Section 4.2. Note that the extended existing solutions cannot support many-to-many joins obliviously. As we show below, they may leak some sensitive information including the join degree. Hence, our goal is to address the security issue and propose new algorithms to ensure the obliviousness.

5.1 Oblivious Sort-Merge Equi-Join

Strawman solution. For foreign key joins, ObliDB [35] supports two sort-merge join algorithms: 1) a re-implementation of Opaque join [87] and 2) 0-OM Join. The only difference between them is that 0-OM Join only requires $O(1)$ trusted memory but pays larger time complexity. However, Example 1 shows that this strawman solution does not work for many-to-many join, due to leaking some sensitive information (*e.g.*, the join degree).

Example 1. *An example is shown in Figure 2. Given two input tables T_1 and T_2, Opaque join [87] and 0-OM Join [35] first put tuples from both input tables into one single table T, and obliviously sort T according to the join key. Next, they perform a linear scan over the single sorted table T. For primary-foreign key join, they perform a linear scan over T and join each primary key tuple (originally from T_1) with the corresponding foreign key tuples (originally from T_2). They ensure the* invariant *that after accessing every input tuple in T, they write out exactly one real or dummy join record.*

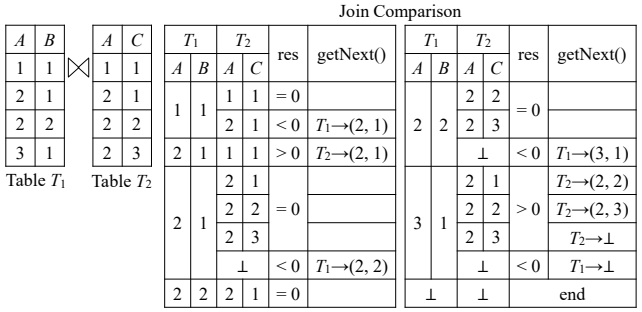


Figure 3: An example of oblivious sort-merge equi-join.

Algorithm 1: Oblivious Binary Sort-Merge Equi-Join

Require: Input: two tables T_1 and T_2.
Output: join result table $T_{\text{out}} = T_1 \bowtie T_2$.

```
1: Initialize T_out := ∅.
2: Initialize tuple[1, 2] := ∅.
3: for i := 1 to 2 do
4:     tuple[i] := T_i.getFirst();
5: while tuple[1] ≠⊥ or tuple[2] ≠⊥ do
6:     res := compare(tuple[1], tuple[2]);
7:     if res = 0 then
8:         begin := tuple[2];
9:         while res = 0 do
10:            T_out.put(tuple[1] ⋈ tuple[2]);
11:            T_1.getDummy(); tuple[2] := T_2.getNext();
12:            res := compare(tuple[1], tuple[2]);
13:        T_out.put(⊥);
14:        tuple[2] := begin;
15:        tuple[1] := T_1.getNext(); T_2.getDummy();
16:    else
17:        T_out.put(⊥);
18:        if res < 0 then
19:            tuple[1] := T_1.getNext(); T_2.getDummy();
20:        else
21:            T_1.getDummy(); tuple[2] := T_2.getNext();
22: Obliviously filter out dummy records from T_out.
23: return T_out;
```

But for many-to-many join, they will not have such an invariant. For example, after accessing the tuple $T_2(2,1)$ (originally from T_2), which can match two tuples $T_1(2,1)$ and $T_1(2,2)$ (originally from T_1), they must output two join records $T_1(2,1) \bowtie T_2(2,1)$ and $T_1(2,2) \bowtie T_2(2,1)$ before the next access over T, i.e., the number of output records between two accesses over T leaks the join degree. Processing tuples $T_2(2,2)$ and $T_2(2,3)$ brings the same security issue.

Our algorithm. Our algorithm is similar to the traditional sort-merge join but with some differences. During preprocessing, we integrate non-clustered B-tree indices into ORAMs for each input table in advance, where each leaf index entry keeps a pointer to the data tuple. Leaf index entries (rather than data tuples) are sorted as per the attribute. Succeeding data tuples from any input table can be retrieved through the pointers in succeeding leaf index entries. For each input table, we build an ORAM structure for data blocks and another smaller one for index blocks (see "Segmenting ORAM" in Section 4.2).

In each join step, we keep the *invariant* that we retrieve the tuple needed from each input table *alternatively*. A dummy tuple is retrieved as necessary. It ensures the full obliviousness, since each tuple retrieval needs the same number of ORAM accesses for each input table. Then, we perform a join comparison in each step. If there is a match, we write out a join record; otherwise, we write out a dummy record as necessary. Then the adversary cannot find any join degree information by observing the access patterns.

Example 2. *Algorithm 1 shows the details of joining two tables T_1 and T_2. Note that whenever we perform a* getNext() *over one input table (T_1 or T_2), we also perform a dummy operation* getDummy() *over the other table (T_2 or T_1) to ensure the obliviousness.*

An example is given in Figure 3. First, Algorithm 1 initializes $T_{\text{out}} := \emptyset$ (Line 1). Then, we retrieve the first two tuples (e.g., $T_1(1,1)$ and $T_2(1,1)$) from T_1 and T_2 as tuple[1] *and* tuple[2] *(Line 2-4). While any tuple is real, we perform a join comparison between them (Line 5-6). We keep the* invariant *above that we always pull tuples from T_1 and T_2 alternatively for either of two possible cases:*

1) tuple[1] *matches* tuple[2] *(e.g., $T_1(1,1)$ and $T_2(1,1)$ (Line 7)). First, we save the current* tuple[2] *(e.g., $T_2(1,1)$) to a temporary tuple "*begin*" (Line 8). We keep writing out the join record (e.g., $T_1(1,1) \bowtie T_2(1,1)$) to T_{out}, and retrieving the next tuple (e.g., $T_2(2,1)$) from T_2 as* tuple[2]*, until the newly retrieved* tuple[2] *does not match* tuple[1] *(e.g., $T_2(2,1)$ does not match $T_1(1,1)$) (Line 9-12). Whenever we invoke a getNext() from T_2, we also pull a dummy tuple from T_1 to ensure the obliviousness (Line 11). Once they do not match, we write out a dummy record and assign "*begin*" (e.g., $T_2(1,1)$) back to* tuple[2] *(Line 13-14). Then, we will retrieve the next tuple (e.g., $T_1(2,1)$) from T_1, and also pull a dummy tuple from T_2 (Line 15). Finally, we move to the next iteration (Line 5-6).*

2) tuple[1] *does not match* tuple[2] *(e.g., for now, $T_1(2,1)$ and $T_2(1,1)$ (Line 7)). Since they do not match, we first write out a dummy record (Line 17). If* tuple[1] *ranks ahead of* tuple[2]*, we retrieve the next tuple from T_1 (Line 18-19). Otherwise (e.g., $T_1(2,1)$ ranks behind of $T_2(1,1)$), we retrieve the next tuple from T_2 (Line 20-21). Note that in either branch we also pull a dummy tuple from the other table. Finally, we move to the next iteration (Line 5-6).*

Note that once a cursor moves to the end of table T_1 or T_2, a dummy tuple $\perp$ will be retrieved from that table. We logically let it rank behind of any real tuple in join comparison. For example, when the cursor on T_1 moves to tuple $T_1(2,2)$ and that on T_2 reaches the end of T_2, we will retrieve a dummy tuple $\perp$ from T_2 (Line 11) and let the join comparison result be res < 0 *(logically $T_1(2,2).A < \perp.A$) (Line 12). The rest still goes in the same way as stated above.*

After both cursors reach the end of tables T_1 and T_2, the final step is to filter out dummy records from T_{out} using oblivious sorting and only keep real join records (Line 22).

Note that B-tree indices are not required for Algorithm 1. If each tuple keeps the pointer to the next tuple, succeeding tuples can be retrieved when needed through ORAM using the pointers.

Last, we consider whether the number of tuple retrievals from each input table leaks any sensitive information. Theorem 1 shows that this number will be a function of the sizes of input tables and real join result, *i.e.*, no additional information is leaked except for the sizing information of input and output tables.

Theorem 1. *For any two input tables T_1 and T_2 and the real join result R_{real}, let Num_{tr} be the number of tuple retrievals from each input table. It is a function of $|T_1|$, $|T_2|$ and $|R_{\text{real}}|$. Specifically, we have $\text{Num}_{\text{tr}} = f_{\text{osmj}}(|T_1|, |T_2|, |R_{\text{real}}|) = |T_1| + |T_2| + |R_{\text{real}}| + 1$.*

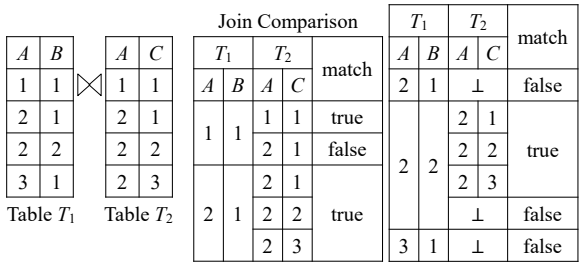


Figure 4: An example of oblivious index nested-loop equi-join.

Proof. We divide the process of Algorithm 1 into two parts and compute the number of tuple retrievals in each part.

Part I: The process except for Line 9-12 in Algorithm 1.

In Part I, at the beginning, we make one call of getFirst() over T_1 and T_2 (Line 3-4). Recall that after each pair of tuple retrievals from T_1 and T_2, we perform exactly one join comparison. In Part I, each join comparison leads to exactly one dummy output record. If the comparison result is res > 0, the cursor on T_2 advances (Line 21); otherwise, the cursor on T_1 advances (Line 15 and Line 19). In total, the number of calling getNext() over T_1 and T_2 is $|T_1|$ and $|T_2|$ respectively. Therefore, the total number of tuple retrievals from each input table in Part I is $|T_1| + |T_2| + 1$.

Part II: The process in Line 9-12 in Algorithm 1.

Recall that after each pair of tuple retrievals from T_1 and T_2, we perform exactly one join comparison. In Part II, each join comparison leads to exactly one real join record. Since the number of real join records is $|R_{\text{real}}|$, the number of tuple retrievals from each input table in Part II is also $|R_{\text{real}}|$.

Based on the analysis above, we will have $\text{Num}_{\text{tr}} = f_{\text{osmj}}(|T_1|, |T_2|, |R_{\text{real}}|) = |T_1| + |T_2| + |R_{\text{real}}| + 1$. □

In Figure 3, the sizes of two input tables are $|T_1| = 4$ and $|T_2| = 4$, the real join size is $|R_{\text{real}}| = 7$, and the total number of tuple retrievals from each input table is $\text{Num}_{\text{tr}} = |T_1|+|T_2|+|R_{\text{real}}|+1 = 16$.

5.2 Oblivious Index Nested-Loop Equi-Join

Strawman solution. A strawman solution is similar to the traditional index nested-loop join. The only difference is that this solution integrates B-tree indices into ORAMs for the input tables and retrieves tuples by querying the index structure through an ORAM protocol. In detail, the outer loop is to scan table T_1. While accessing each tuple in T_1, the algorithm retrieves matched tuples from table T_2 one by one through B-tree index. If no matched tuple is found, it outputs a dummy record; otherwise, it outputs one join record for each match. However, this strawman solution leaks the join degree in a similar way. The number of output records between two tuple retrievals from T_1 leaks the join degree.

Our algorithm. We integrate B-tree indices into ORAMs for each input table in preprocessing. The index structure is the same as that in oblivious sort-merge join. To address the security issue in the strawman solution, we add dummy tuple retrievals from table T_1. We ensure the *invariant* that we retrieve the tuple needed from each input table *alternatively*. The difference on two tables is that we retrieve tuples from T_1 one by one according to sequential block IDs, while for table T_2 we retrieve the tuple that we need by searching over a whole B-tree path. After each pair of tuple retrievals, we make a join comparison of the current two tuples. If there is a match, we write a join record to the output table; otherwise, a dummy record is output as necessary. Then the adversary cannot find any join degree information by observing the access patterns.

Algorithm 2: Oblivious Index Nested-Loop Binary Equi-Join

Require: Input: two tables T_1 and T_2.
Output: join result table $T_{\text{out}} = T_1 \bowtie T_2$.
1: Initialize $T_{\text{out}} := \emptyset$.
2: Initialize tuple$[1, 2] := \emptyset$.
3: **for** $i := 1$ to $|T_1|$ **do**
4: tuple$[1] := T_1$.getNext();
5: tuple$[2] := T_2$.getFirst(tuple$[1]$.key);
6: **while** match(tuple$[1]$, tuple$[2]$) = **true do**
7: T_{out}.put(tuple$[1] \bowtie$ tuple$[2]$);
8: T_1.getDummy();
9: tuple$[2] := T_2$.getNext();
10: T_{out}.put($\perp$);
11: Obliviously filter out dummy records from T_{out}.
12: **return** T_{out};

Example 3. *Algorithm 2 shows the details of joining two tables T_1 and T_2. An example is given in Figure 4. Algorithm 2 begins with initializing an empty output table T_{out} (Line 1). The outer loop is to iterate over each tuple in table T_1 (Line 3). Each time we retrieve a new tuple* tuple$[1]$ *(e.g., $T_1(1, 1)$) from T_1 (Line 4), we first retrieve a tuple* tuple$[2]$ *(e.g., $T_2(1, 1)$) from T_2, which is the first tuple whose join key is no less than* tuple$[1]$*'s (e.g., $T_2(1, 1).A \geq T_1(1, 1).A$) (Line 5). If those two tuples can match (e.g., $T_1(1, 1).A = T_2(1, 1).A$), we write a join record (e.g., $T_1(1, 1) \bowtie T_2(1, 1)$) to the output table T_{out} (Line 7) and retrieve the next tuple (e.g., $T_2(2, 1)$) from T_2 as* tuple$[2]$ *(Line 9). To ensure the obliviousness, we also perform a dummy retrieval from T_1 (Line 8). We repeat the process above until the newly retrieved* tuple$[2]$ *does not match the current* tuple$[1]$ *(e.g., $T_2(2, 1)$ does not match $T_1(1, 1)$). Once they do not match, we write out a dummy record (Line 10) and step into the next iteration (e.g., processing the next tuple $T_1(2, 1)$ from T_1).*

During the process above, once we cannot find any tuple needed from T_2, we retrieve a dummy tuple $\perp$ from T_2 and logically let the matching result be false. For example, when processing tuple $T_1(3, 1)$ from T_1, since we cannot find any tuple from T_2 with join key $A \geq 3$, we simply retrieve a dummy tuple $\perp$ from T_2 and let the matching result be false. The rest still goes in the same way as stated above. The final step is to obliviously filter out dummy records from T_{out} and only keep real join records (Line 11).

In a similar way, Theorem 2 shows that the number of tuple retrievals from each input table leaks no more sensitive information except for the sizing information of input and output tables.

Theorem 2. *For any two input tables T_1 and T_2 and the real join result R_{real}, let Num_{tr} be the number of tuple retrievals over each input table. It is a function of $|T_1|$, $|T_2|$ and $|R_{\text{real}}|$. Specifically, we have $\text{Num}_{\text{tr}} = f_{\text{obej}}(|T_1|, |T_2|, |R_{\text{real}}|) = |T_1| + |R_{\text{real}}|$.*

Proof. In Algorithm 2, after each pair of tuple retrievals from T_1 and T_2, we perform exactly one join comparison. If the current two tuples can match, we write out the join record to T_{out} (Line 7); otherwise, we write out a dummy record to T_{out} (Line 10). By observing the process of Algorithm 2, the outer loop performs exactly $|T_1|$ iterations, and each iteration leads to exactly one dummy output record

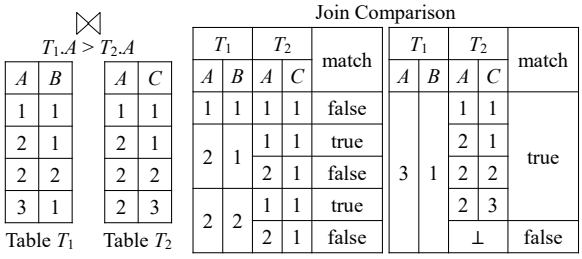


Figure 5: An example of oblivious index nested-loop band join.

(Line 10). Thus, the number of dummy output records is $|T_1|$. Since the number of real join records is $|R_{\text{real}}|$, the total number of output records will be $|T_1| + |R_{\text{real}}|$. Therefore, $\text{Num}_{\text{tr}} = |T_1| + |R_{\text{real}}|$. □

In Figure 4, the first input table size is $|T_1| = 4$, the real join size is $|R_{\text{real}}| = 7$, and the total number of tuple retrievals from each input table is $\text{Num}_{\text{tr}} = |T_1| + |R_{\text{real}}| = 4 + 7 = 11$.

5.3 Oblivious Index Nested-Loop Band Join

Prior studies [11, 35, 53, 87] do not support oblivious band joins, except for a Cartesian product solution [54]. We do support some types of band joins (*e.g.*, "$>$", "$\geq$", "$<$" and "$\leq$") by extending Algorithm 2 and leveraging B-tree indices to speed up the process.

Example 4. *Our band join algorithm is similar to Algorithm 2. To join two input tables T_1 and T_2, the outer loop is to iterate over each tuple in table T_1. For each iteration, we retrieve a new tuple from T_1. If join type is "$>$" or "$\geq$", we begin with retrieving the first tuple from T_2 and keep retrieving succeeding tuples until they cannot match. If join type is "$<$" or "$\leq$", we begin with retrieving the last tuple from T_2 and keep retrieving preceding tuples until they cannot match.*

An example is given in Figure 5. When processing tuple $T_1(2,1)$ from T_1, since join type is "$>$", we begin with retrieving the first tuple $T_2(1,1)$ from T_2. Since two tuples can match ($T_1(2,1).A > T_2(1,1).A$), we keep retrieving succeeding tuples from T_2, until the newly retrieved tuple (e.g., $T_2(2,1)$) from T_2 does not match tuple $T_1(2,1)$.

In each step, we also perform a dummy retrieval from T_1 as necessary. We ensure the invariant *that we retrieve the tuple needed from each input table* alternatively. *If there is a match, we output the join record; otherwise, we output a dummy record. The final step is to obliviously filter T_{out} and only keep real join records.*

Theorem 3. *For any two input tables T_1 and T_2 and the real join result R_{real}, let Num_{tr} be the number of tuple retrievals over each input table. It is a function of $|T_1|$, $|T_2|$ and $|R_{\text{real}}|$. Specifically, we have $\text{Num}_{\text{tr}} = f_{\text{obdj}}(|T_1|, |T_2|, |R_{\text{real}}|) = |T_1| + |R_{\text{real}}|$.*

Proof. The proof is similar to that of Theorem 2. □

In Figure 5, the first input table size is $|T_1| = 4$, the real join size is $|R_{\text{real}}| = 6$, and the total number of tuple retrievals from each input table is $\text{Num}_{\text{tr}} = |T_1| + |R_{\text{real}}| = 4 + 6 = 10$.

6 OBLIVIOUS MULTIWAY EQUI-JOIN

Recent work [35, 53, 87] supports foreign key joins or binary equi-joins obliviously. However, extending these algorithms to oblivious equi-joins over multiple tables will leak the intermediate table sizes, which may pertain to some sensitive information (*e.g.*, the data distribution or the sparseness of the intermediate join graph).

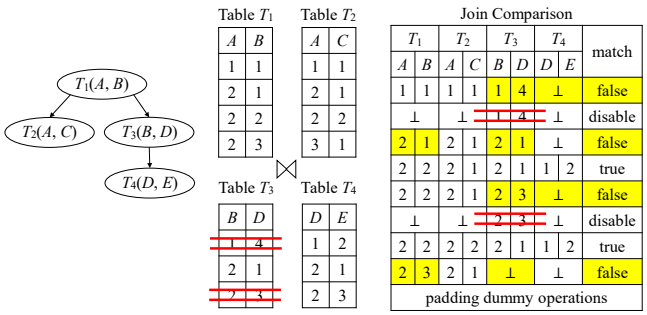


Figure 6: An example of oblivious multiway equi-join.

Arasu and Kaushik [11] supports oblivious multiway equi-joins, for the case where the join graph is *acyclic*, while hiding the sizes of intermediate tables. However, Krastnikov *et al*. [53] points out that details in [11] are incomplete, and no practical implementation is provided to show the empirical results.

In this work, we extend our Algorithm 2 to support *acyclic multiway equi-joins* obliviously. The key idea is to avoid retrieving tuples that make no contribution to the final join result to bound the total number of block accesses.

Example 5. *Figure 6 shows an example of acyclic multiway equi-join over four tables T_1-T_4. Due to the acyclicity, each input table can be arranged as a node in a join tree, which is given in the left part of Figure 6. In this tree, for any different tables T_i, T_j, T_k, if T_k is on the path from T_i to T_j, we must have $\text{Attr}(T_i) \cap \text{Attr}(T_j) \subseteq \text{Attr}(T_k)$ for their attribute sets. The algorithm of building a join tree is presented in [85]. Without loss of generality, we number input tables in a pre-order traversal of the join tree. It ensures $i < j$, if T_i is an ancestor table of T_j. We also denote the parent table of T_i in the join tree as $T_{p(i)}$.*

In our index nested-loop join algorithm [1], the outer loop is to iterate over each tuple in root table T_1. Each time we retrieve a new tuple (e.g., $T_1(1,1)$) from T_1, we search matched tuples (e.g., $T_2(1,1)$, $T_3(1,4)$, $\cdots$) from $T_2, \cdots, T_\ell$. To ensure the obliviousness, we retrieve the tuple needed from each input table in a round-robin way *and add dummy retrievals as necessary (e.g., retrieve $\perp$ from T_4, due to no tuple with join key $D \geq 4$ for matching $T_3(1,4)$, as highlighted in yellow in Figure 6). In each step, if there is a match (e.g., in 4th and 7th join step), we output the join record; otherwise, we output a dummy record.*

To bound the total number of join steps, we make the following observations to avoid retrieving unnecessary tuples that makes no contribution to the final join result.

Observation 1. *For any non-root table T_j and its parent table $T_{p(j)}$, $\text{tuple}[p(j)]$ in $T_{p(j)}$ makes no contribution to the final join result, if no tuple in T_j matches $\text{tuple}[p(j)]$. Then, $\text{tuple}[p(j)]$ can be safely* disabled *(i.e., will not be accessed in the future).*

For example, for table T_4 and its parent table T_3, given the parent tuple $T_3(1,4)$, we find no tuple in T_4 matches tuple $T_3(1,4)$ (in 1st join step). Hence, we know that $T_3(1,4)$ makes no contribution to the final join result. Then, we safely disable *tuple $T_3(1,4)$ by adding a dummy join step (in 2nd join step). In this dummy step, we perform a dummy tuple retrieval from each input table except T_3. For T_3, we perform a tuple disabling operation, which is indistinguishable from a tuple retrieval based on the access patterns.*

[1] Due to space limit, implementation details of multiway equi-join algorithm, proof of Theorem 4 and complexity analyses on our algorithms are given in full version [4].

When disabling any tuple, we mark the corresponding leaf entry as disabled using an additional boolean tag rather than actually delete any entry or tuple. If all entries in any B-tree leaf block have been marked as disabled, the parent entry in the B-tree parent block will also be marked as disabled. This can recursively go up to B-tree root block. Since the recursion goes up along a B-tree path, we can still finish each disabling operation using some additional B-tree path access through ORAM (i.e., adding some dummy join step). When retrieving a new tuple from any input table, we skip the disabled entries during searching over the B-tree index.

Observation 2. *For any non-root table T_j and its parent table $T_{p(j)}$,* $\text{tuple}[p(j)]$ *in $T_{p(j)}$ makes no contribution to the final join result, if each tuple in T_j that matches* $\text{tuple}[p(j)]$ *has been* disabled. *Then,* $\text{tuple}[p(j)]$ *can also be safely* disabled.

For example, for table T_3 and its parent table T_1, given the parent tuple $T_1(1,1)$, $T_3(1,4)$ is the only tuple in T_3 that matches $T_1(1,1)$. However, since $T_3(1,4)$ has been disabled (in 2nd join step), we know that $T_1(1,1)$ makes no contribution to the final join result. If the parent tuple is in a non-root table, we will disable it by adding some dummy join step as above. Otherwise, we do not physically disable any tuple in root table T_1, since the outer loop in our algorithm iterates over each tuple in root table T_1, and will not access any previous tuple in T_1 in the future.

Observation 3. *For any non-root table T_j and its parent table $T_{p(j)}$,* $\text{tuple}[p(j)]$ *in $T_{p(j)}$ will have no more matches, if the current tuple* $\text{tuple}[j]$ *in T_j matches* $\text{tuple}[p(j)]$ *but the succeeding tuple in T_j has a different join key from* $\text{tuple}[j]$*'s.*

Observation 3 is based on the property of equi-joins. For example, for table T_3 and its parent table T_1, given the parent tuple $T_1(1,1)$, we find that the current tuple $T_3(1,4)$ can match $T_1(1,1)$ (in 1st join step). But since the succeeding tuple $T_3(2,1)$ has a different join key from $T_3(1,4)$, we can conclude that $T_3(2,1)$ does not match $T_1(1,1)$ in equi-join scenario. Hence, $T_1(1,1)$ will have no more matches.

To perform this optimization, we attach another boolean tag to each leaf entry, which indicates whether the next leaf entry in T_j has the same key with the current entry in T_j. If not, we do not retrieve the next tuple from the child table T_j. After answering each join query, we simply go over all index blocks and reset all boolean tags.

After the normal join process, we pad the number of join steps to the upper bound (e.g., the last step in Join Comparison in Figure 6) in Theorem 4 to ensure the obliviousness. Finally, we obliviously filter out dummy records and only keep real join records. The last step is to go over all index blocks and reset boolean tags in each entry.

Theorem 4. *For any ℓ ($\ell \geq 2$) input tables $T_1, \cdots, T_\ell$ and the real join result R_{real}, let Num_{tr} be the number of tuple retrievals over each input table. It is a function of $|T_1|, \cdots, |T_\ell|$ and $|R_{\text{real}}|$. Specifically,*

$$\text{Num}_{\text{tr}} = f_{\text{omj}}(|T_1|, \cdots, |T_\ell|, |R_{\text{real}}|) = |T_1| + 2\sum_{j=2}^{\ell} |T_j| + |R_{\text{real}}|.$$

7 DISCUSSION ON ONE ORAM SETTING

In this work, we separate one single ORAM into multiple smaller ORAMs (denoted as SepORAM) to reduce the cost of each ORAM access, as in ObliDB [35]. Now, we reconsider join optimizations in one ORAM setting (denoted as OneORAM). The key observation is that we retrieve any tuple from any input table through one single ORAM. If we pay the same cost for each tuple retrieval from each input table, *e.g.*, padding the number of ORAM accesses to the maximum height of all B-tree indices, each tuple retrieval from any input table will be indistinguishable for the adversary, although he knows the total number of tuple retrievals.

A major optimization in OneORAM is to safely remove some dummy retrievals to speed up the join processing. Note that after each tuple retrieval from any input table in OneORAM (rather than after each join step in SepORAM), we writes out a real join record or a dummy record to the output table, to protect the join degree information and ensure the full obliviousness. As long as the total number of tuple retrievals only pertains to the input and output sizes, no additional information will be leaked. The last thing is to bound the total number of tuple retrievals in OneORAM, similar to Theorems 1-4. Details are given in the full version of our paper [4].

However, there is a major drawback in OneORAM setting. Suppose there are multiple tables in the whole dataset (*e.g.*, 8 tables in TPC-H dataset), but only a few binary joins will be processed online. In this scenario, we need to put all input tables into one single ORAM in advance, since we do not know which two tables will be joined online. In online processing, we have to pay much larger cost for accessing the large single ORAM rather than smaller separate ORAMs. Mainly due to this drawback, algorithms in OneORAM setting perform no better than those in SepORAM setting in most cases, which is confirmed by our experimental results.

8 SECURITY ANALYSIS

We provide an (informal) security theorem for our method, as with Opaque [87] and ObliDB [35]. Our security is guaranteed by the existence of simulator SIM: any probabilistic polynomial-time (PPT) adversary $\mathcal{A}$ cannot distinguish between the real server location trace from our method and the simulated trace from simulator SIM. SIM only has the access to the schema and sizing information of input and output tables, the oblivious join operator, and some specific public constants (*e.g.*, the number of outsourced levels in each B-tree index, denoted as Δ). Hence, the adversary cannot learn any additional information in oblivious join processing, since simulator SIM only sees the above information. Note that SIM has no access to the sizes of all intermediate join tables, since we protect this sensitive information against the adversary. We formalize our security guarantee in Theorem 5 with the same notations in Definition 1.

Theorem 5. *For any relational database D, schema $\text{Sch}(D)$, join query Q, oblivious join algorithm* OJoin*, and security parameter λ, there is a polynomial-time simulator* SIM *such that for any PPT adversary $\mathcal{A}$,*

$$|\Pr[\mathcal{A}(\text{SIM}(\text{Size}(D), \text{Sch}(D), \text{IOSize}(D,Q), \text{OJoin}(D,Q))) \Rightarrow 1] - \Pr[\mathcal{A}(\text{Trace}(\text{OJoin}(D,Q))) \Rightarrow 1]| \leq negl(\lambda).$$

Proof. (Informal Sketch) In this proof, we show the existence of simulator SIM, and argue that access pattern of SIM is distributed indistinguishable from $\text{Trace}(\text{OJoin}(D,Q))$ (generated from algorithm $\text{OJoin}(D,Q)$). SIM reads algorithm $\text{OJoin}(D,Q)$ to determine which operations to simulate. Specifically, SIM needs to simulate access patterns for ORAM operations and oblivious filter operations (including oblivious sorting and a few linear scans) in $\text{OJoin}(D,Q)$.

This proof is covered by Arguments A1-A4. We mainly focus on separate ORAMs setting (denoted as SepORAM) in Arguments

A1-A3. For one ORAM setting (denoted as OneORAM), the proof relies on Argument A4: OneORAM does not introduce any more privacy leakage than SepORAM.

A1: We ensure the obliviousness in each join step.

First, we argue that SIM can simulate each ORAM or oblivious filtering operation. Since SIM has the access to schema $\mathsf{Sch}(D)$ and sizing information $\mathsf{Size}(D)$, the access pattern simulation for each of such operations is the same as that in the original ORAM scheme, or that for original oblivious sorting and linear scan operations.

Then, we argue that SIM can simulate each join step. In SepORAM, we keep the *invariant* that we always retrieve the tuples needed from each input table in a round-robin way in each join step. Even if we do not need to retrieve any new tuple, we still retrieve a dummy tuple to ensure the obliviousness. At the end of each join step, if there is a match, we write out a join record to the output table; otherwise, we write out a dummy record as necessary. Specifically, each tuple retrieval for any input table leads to the same number of ORAM accesses, which only pertains to the height of the outsourced B-tree index. In each join step, since SIM has the access to specific public constants (*e.g.*, the number of outsourced levels in each B-tree index), SIM can perform the corresponding number of ORAM operation simulations for each input table in a round-robin way and output a (randomized encrypted) join record.

A2: We ensure the number of join steps only pertains to the input and output sizes.

In SepORAM, Theorems 1-4 guarantee that the number of tuple retrievals from each input table (*i.e.*, number of join steps) in algorithm $\mathsf{OJoin}(D, Q)$ only pertains to the input and output sizes $\mathsf{IOSize}(D, Q)$. Since SIM has the access to $\mathsf{IOSize}(D, Q)$, SIM will know the number of join steps based on $\mathsf{IOSize}(D, Q)$, and perform the corresponding number of join step simulations.

A3: A1 and A2 ensure the simulated access pattern is distributed indistinguishably from $\mathsf{Trace}(\mathsf{OJoin}(D, Q))$ in the whole process (*i.e.*, the obliviousness in SepORAM).

A4: OneORAM does not introduce any more privacy leakage than SepORAM.

For each step in OneORAM, algorithm $\mathsf{OJoin}(D, Q)$ retrieves the tuple needed from an input table through the single ORAM, and pads the number of ORAM accesses to the maximum length of all retrieved B-tree paths. It ensures that each tuple retrieval from any input table will be indistinguishable for the adversary. Note that $\mathsf{OJoin}(D, Q)$ may remove some dummy tuple retrievals, as long as total number of tuple retrievals only pertains to the input and output sizes $\mathsf{IOSize}(D, Q)$. Then, after each tuple retrieval from any input table in OneORAM (rather than after each join step in SepORAM), we ensure to write out a real or dummy join record to the output table, to protect the join degree information and ensure the full obliviousness. The simulation is similar to that in SepORAM, since SIM still has the access to the background knowledge. □

Theorem 5 guarantees the security of our algorithms in the sense of Definition 1. For binary joins, our security guarantee is the same as Krastnikov *et al*. [53] and oblivious mode in Opaque [87] and ObliDB [35]. For multiway joins, our security guarantee is the same as Arasu and Kaushik [11].

Last, we may also introduce a padding mode, as in Opaque [87] and ObliDB [35]. The join result size will be padded to an upper bound size, which leaks nothing regarding the join query but the upper bound size. Besides, some novel padding techniques can be introduced, *e.g.*, exploring differential privacy rather than full obliviousness to reduce the padding size [17], or padding the result size to the closest power of a constant x (*e.g.*, 2 or 4) [12, 22, 34], leading to at most $\log_x |R_{\text{worst}}|$ distinct result sizes, where $|R_{\text{worst}}|$ is the Cartesian product size in join scenario.

The simulator SIM' for padded mode behaves analogously to SIM. In padded mode, the security theorem replaces the final join output size with an upper bound size as a public parameter in simulator SIM, which indicates the padded output size.

9 EXPERIMENTAL RESULTS

9.1 Experimental Setup and Datasets

We make the evaluation for Krastnikov *et al*. [53] (denoted as ODBJ), ObliDB [35] and our method. We extend ODBJ implementation [3] to disk-based relational tables. For ObliDB, we set Hash Select as the oblivious filter algorithm (see Table 2 in [35]). There are two settings in our method: SepORAM and OneORAM. Each setting includes three algorithms: SMJ, INLJ and INLJ+Cache (see Table 1). In "+Cache" mode, the client caches all index blocks *above the leaf level*, *i.e.*, the number of outsourced index levels $\Delta = 1$. Due to large fanout in B-tree indices, this overhead to the client storage is very light (see Figures 7 and 8).

We also compare our method with an insure baseline (Raw Index(+Cache)). It builds B-tree indices over data blocks and store them in the cloud *without using any encryption and ORAM protocol*. It also includes three algorithms: Raw SMJ, Raw INLJ and Raw INLJ+Cache (with the same index caching setting as ours).

By default, we make the evaluation for all the algorithms in non-padded mode. In Section 9.6, we briefly discuss the padding techniques by evaluating all secured algorithms in padded mode.

Setup. The client is an Ubuntu 18.04 machine with Intel Core i7 CPU (8 cores, 3.60 GHz) and 18 GB memory. The server is an Ubuntu 18.04 machine with Intel Xeon E5-2609 CPU (8 cores, 2.40 GHz), 256 GB memory and 2 TB hard disk. The bandwidth is 1 Gbps.

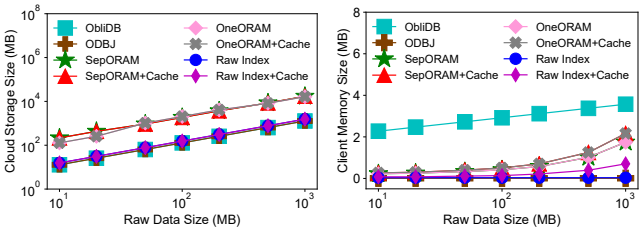
We compare all the methods under the same system setting. Especially, any algorithms that rely on building ORAMs have been implemented solely with ORAMs, although some baseline solutions including ObliDB rely on secure enclave in their original implementation. The commonly used ORAM repository is SEAL-ORAM [2], where the server hosts a MongoDB instance as the outsourced storage. They implement a MongoDB connector class to support insertion, deletion and update operations on blocks inside MongoDB. Specifically, MongoDB only serves as the backend storage but does not provide any other computations or optimizations.

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. We set encrypted block size $B = 4$ KB, as in [23, 68, 81]). We set the number of blocks in each bucket of Path-ORAM to $Z = 4$, as in [23, 69]). For additional trusted memory size M, we set $M = 2B$ (B is the block size) in ODBJ and our method, but set $M = 50 \log N$ (N is the number of data blocks) in ObliDB to make it finish in a reasonable period.

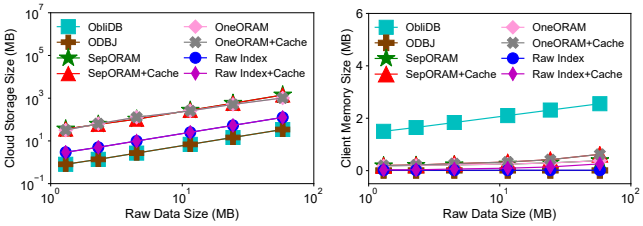
We make the evaluation of all the methods on two datasets.

TPC-H. We set default data size to 100 MB and vary data sizes from 10 MB to 1 GB in standard TPC-H benchmark. It has a comparable



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

Figure 7: Storage cost against raw data size on TPC-H.



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

Figure 8: Storage cost against raw data size on social graph.

size with Big Data Benchmark [1] in ObliDB (see Table 3 in [35]). But note that we focus on many-to-many joins, which generate much more join records and become much more time-consuming. We refer to [86] and explore general many-to-many join queries as follows. Section A shows these queries in SQL.

- Query TE1-TE3: general equi-joins over two tables.
- Query TB1-TB2: band joins over two tables.
- Query TM1-TM3: general multiway joins over three, four and five tables.

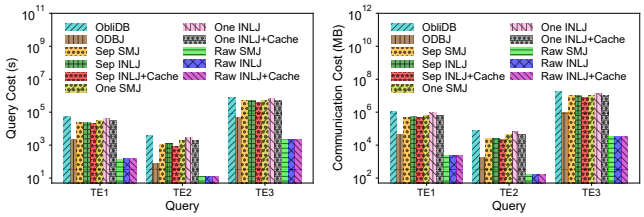
Social graph. Social graph [21, 86] contains three twitter user tables "popular-user", "inactive-user" and "normal-user" with 1 billion friendship records in 19 GB. Each record is a friendship link with a source ID and a destination ID. We randomly sample 20,000 twitter users as our default dataset (*i.e.*, with raw data size 4.5 MB), and vary the number of sampled users from 5,000 to 200,000 (*i.e.*, with raw data size from 1.3 MB to 58 MB) in our experiments. We perform the following join queries on this dataset. Section B shows the queries in SQL.

- Query SE1-SE3: general equi-joins over two tables.
- Query SM1-SM3: general multiway joins over three and four tables.

Remarks. The query cost for each method should be roughly proportional to the communication cost between the cloud and client, as with the computation overhead in Table 1. It is confirmed by our experimental results (see Figures 9-18) to some extent. For simplicity, we mainly focus on experimental results for query cost.

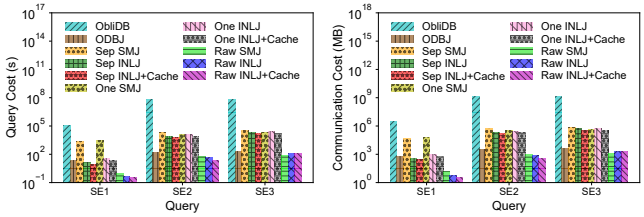
9.2 Cloud and Client Storage Costs

Figures 7a and 8a show the cloud storage cost on two datasets. ObliDB and ODBJ achieve the minimum cloud storage cost, since they only store encrypted data blocks. Raw Index(+Cache) needs a little more cost for storing index blocks. This difference is relatively large on social graph, since leaf level entries have comparable sizes with data records on social graph. Our method has roughly 10X larger cost than Raw Index(+Cache), due to building Path-ORAM data structure. When we scale up both datasets, our largest cloud



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

Figure 9: Performance of binary equi-join on TPC-H.



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

Figure 10: Performance of binary equi-join on social graph.

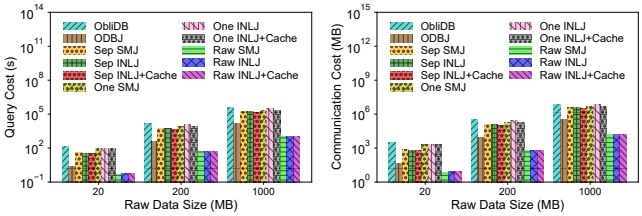
storage cost comes to 16.6 GB on TPC-H and 1.4 GB on social graph, while that of Raw Index(+Cache) is 1.5 GB on TPC-H and 126 MB on social graph.

Figures 7b and 8b show the client memory size on two datasets. ODBJ achieves the minimum cost, since the client always keeps a constant number of blocks. For Raw Index(+Cache), the client also keeps a few more blocks along currently retrieved B-tree paths and may cache the index blocks above the leaf level in "+Cache" mode. For ObliDB, we set the trusted memory size to the largest ($M = 50 \log N$) and make it finish as soon as possible. Our client memory cost grows (roughly) linearly with raw data size, since $O(N/B)$ blocks in the position map dominate the client storage when the number of blocks is large. However, since position map entries are small in size, our client memory size is no larger than 2.2 MB on TPC-H and 0.6 MB on social graph. It can be further mitigated if we instantiate our method with oblivious index.

9.3 Performance of Binary Equi-Join

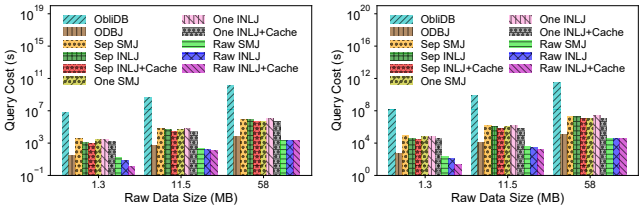
9.3.1 Default Setting. Figures 9a and 10a show query cost for binary equi-join on two datasets in default setting. Our SepORAM(+Cache) algorithms achieve 2X-3X and 50X-3000X better performances than ObliDB on TPC-H and social graph. The speedup difference is mainly due to the join result size, which grows with *square of input size* on TPC-H but is almost *comparable with input size* on social graph. Our method takes advantage of this, since our query cost depends on input and output sizes *linearly*.

Our SepORAM(+Cache) brings 90X-450X larger blowup of query cost than Raw Index(+Cache) except for Query SE1, and also brings 7X-15X and 40X-160X larger blowup of query cost on TPC-H and social graph than ODBJ except for Query SE1. The major reason is that data tuples only contain 100-200 bytes on TPC-H and 2 integers on social graph, much less than 4 KB block size. For index based methods (Raw Index(+Cache) and ours), only one index entry or data tuple in each retrieved index or data block can contribute to the join processing. Based on this, the performance differences will be reduced if we leverage small and suitable block sizes. Another reason is that Path-ORAM brings a relatively large constant factor in Big-O complexity ($2Z \log N$ with $Z = 4$). We may achieve further



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

Figure 11: Performance of Query TE2 against raw data size.



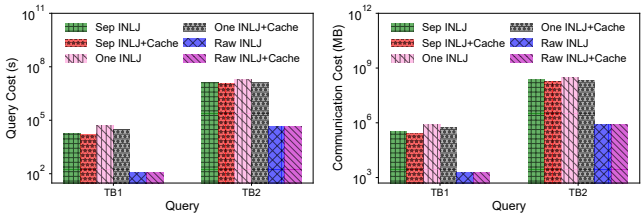
(a) Query cost. (b) Communication cost.

Figure 12: Performance of Query SE2 against raw data size.

performance improvement if using some novel ORAM schemes [13, 25, 63]. In particular Query SE1 joins a small table with a large one but generates few join records. Sep SMJ and Sep INLJ(+Cache) bring 2400X and 30X larger blowup of query cost than Raw Index(+Cache) algorithms. Sep INLJ(+Cache) even achieves 1.7X-2.7X better performance than ODBJ. The reason is that the query cost of Sep INLJ(+Cache) increases with large table size *logarithmically*, while that of Sep SMJ and ODBJ increases with large table size *linearly* (see Table 1).

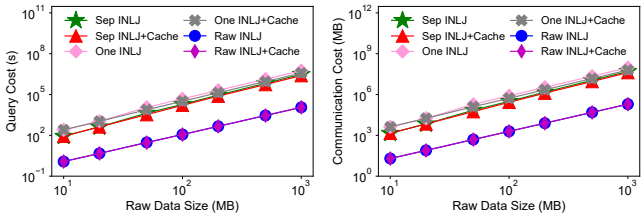
For our method, Sep INLJ achieves 1.2X-2.6X better performance than One INLJ. As explained in Section 7, OneORAM setting has to pay much larger cost for accessing longer paths through the large single Path-ORAM. Besides, One INLJ(+Cache) has to pad the number of ORAM accesses for each tuple retrieval to the maximum length of outsourced B-tree paths to be retrieved, although this problem can be alleviated by index caching. One SMJ does not have this padding problem, since the client always accesses an index block and then a data block for each tuple retrieval through Path-ORAM. One SMJ even achieves 1.6X better performance than Sep SMJ on Query SE2 and SE3, due to less number of tuple retrievals based on the optimization in Section 7. Last, the index caching brings 1.2X-1.6X speedup ratio in both settings.

9.3.2 Scalability. Figures 11a and 12a show query cost for Query TE2 and SE2 against raw data size. For Query TE2, we demonstrate the experimental results on TPC-H dataset with raw data sizes 20 MB, 200 MB, 1,000 MB in Figure 11a. For Query SE2, we demonstrate the experimental results with 5,000, 5,0000, and 200,000 sampled twitter users (*i.e.*, with raw data sizes 1.3 MB, 11.5 MB, and 58 MB) on social graph dataset in Figure 12a. Our SepORAM(+Cache) achieves 2X-4X and 1.6×10^3X-1.6×10^4X better performances than ObliDB for Query TE2 and SE2, when raw data size increases from the minimum to the maximum. The speedup difference for two queries is still on account of the join result size, as explained in Section 9.3.1. Compared with Raw Index(+Cache), SepORAM(+Cache) brings 75X-157X and 161X-409X blowup of query cost on Queries TE2 and SE2. Compared with ODBJ, SepORAM(+Cache) brings



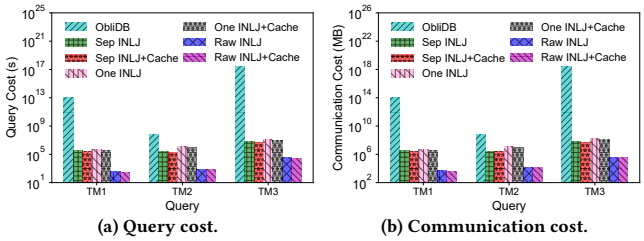
(a) Query cost. (b) Communication cost.

Figure 13: Performance of band join on TPC-H.



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

Figure 14: Performance of Query TB1 against raw data size.



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

Figure 15: Performance of multiway equi-join on TPC-H.

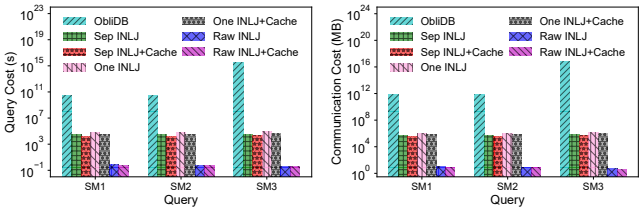
10X-20X and 30X-140X blowup of query cost on Query TE2 and SE2. The major reason of this blowup is still that data tuple sizes are much less than the block size, as explained in Section 9.3.1. Replacing Path-ORAM with some advanced ORAMs can help to improve our query performance. For our method, Sep INLJ achieves 1.1X-3.4X better performance than One INLJ, as explained in Section 9.3.1. As in Section 9.3.1, One SMJ even achieves 1.4X-1.7X better performance than Sep SMJ on Query SE2 due to less number of tuple retrievals. Last, the index caching brings 1.2X-2.0X speedup ratio on two queries in both settings.

9.4 Performance of Band Join

Figure 13a shows query cost for band join on TPC-H in default setting. In comparison with Raw INLJ(+Cache), our Sep INLJ(+Cache) brings 164X and 288X blowup of query cost on Query TB1 and TB2. For our method, Sep INLJ achieves 1.4X-2.5X better performance than One INLJ, as explained in Section 9.3. The index caching brings 1.2X-1.5X better performance in both settings. Figure 14a shows query cost on Query TB1 against raw data size. When raw data size increases from 10 MB to 1 GB, Sep INLJ(+Cache) brings 73X-264X blowup of query cost on Query TB1 compared with Raw INLJ(+Cache). For our method, Sep INLJ achieves 1.9X-2.9X better performance than One INLJ, and the index caching achieves 1.2X-1.5X better performance in both settings.

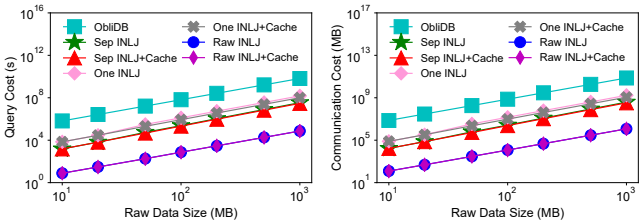
9.5 Performance of Multiway Equi-Join

9.5.1 Default Setting. Figures 15a and 16a show query cost for multiway equi-join on two datasets in default setting. Our Sep INLJ(+Cache) achieves 10^6X-10^{11}X better performance than ObliDB



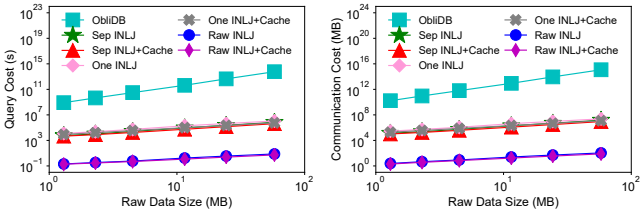
(a) Query cost. (b) Communication cost.

Figure 16: Performance of multiway equi-join on social graph.



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

Figure 17: Performance of Query TM2 against raw data size.

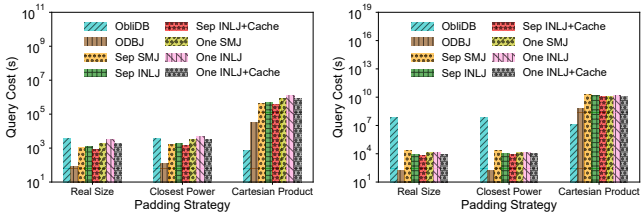


(a) Query cost. (b) Communication cost.

Figure 18: Performance of Query SM2 against raw data size.

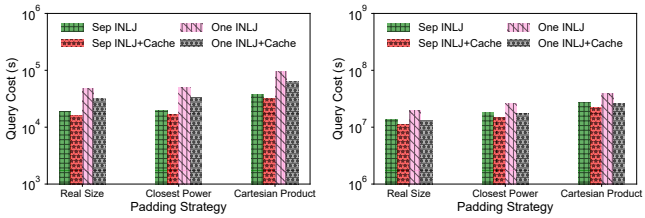
on all queries except Query TM2. The reason is that our query cost is roughly *linear* with input and output sizes, but ObliDB has to perform a Cartesian product. For Query TM2, this speedup ratio goes down to 280X. The reason is that the join result size grows with *square of input size*, consistent with the Cartesian product of four tables (including two nation tables with a static size). In comparison with Raw INLJ(+Cache), Sep INLJ(+Cache) brings 185X-985X and 37000X-70000X blowup of query cost on TPC-H and social graph. The blowup difference on two datasets is due to different join result sizes. Raw INLJ(+Cache) leverages the index filtering well when real join size is small, but our method must pad the number of operations to the upper bound to ensure the obliviousness. For our method, Sep INLJ achieves 1.6X-2.4X better performance than One INLJ on all queries except Query TM2. For Query TM2, this performance difference goes up to 5.5X. The reason is that Sep INLJ has to keep accessing the large single Path-ORAM that contains the largest table `lineitem`, although `lineitem` is not covered in Query TM2. Last, the index caching brings 1.1X-1.5X speedup ratio in both settings.

9.5.2 Scalability. Figures 17a and 18a show query cost for multiway equi-joins Query TM2 and SM2 against raw data size. For Query TM2, our Sep INLJ(+Cache) achieves 190X-430X better performance than ObliDB. The speedup ratio is roughly stable, since the join result size is roughly proportional to the Cartesian product size. For Query SM2, this speedup ratio increases to 10^5X-10^8X, due to far less join result size. Compared with Raw INLJ(+Cache), Sep INLJ(+Cache) brings 194X-469X and 28000X-91000X blowup



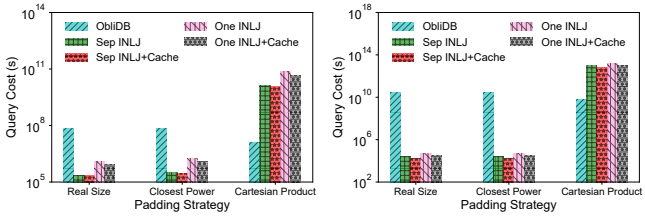
(a) Query TE2. (b) Query SE2.

Figure 19: Padded vs. non-padded mode (binary equi-join).



(a) Query TB1. (b) Query TB2.

Figure 20: Padded vs. non-padded mode (band join).



(a) Query TM2. (b) Query SM2.

Figure 21: Padded vs. non-padded mode (multiway equi-join).

of query cost on Query TM2 and SM2. The blowup difference on two queries is still due to the different join result sizes, as explained in Section 9.5.1. For our method, Sep INLJ achieves 4.4X-5.5X and 1.6X-2.3X better performances than One INLJ on Query TM2 and SM2. For Query TM2, the reason of this performance difference is still that the largest table `lineitem` on TPC-H is not covered in Query TM2, as explained in Section 9.5.1. Last, the index caching brings 1.1X-2.0X speedup ratio in both settings.

9.6 Padded Mode vs. Non-Padded Mode

We also make the comparison between padded mode and non-padded mode for all secured methods. We discuss three padding strategies for the join result size: (1) no padding (denoted as Real Size); (2) padding to the closest power of a constant $x = 2$ (denoted as Closest Power) as in [12, 22, 34]; (3) padding to the Cartesian product (denoted as Cartesian Product). Details of the padding techniques are provided in the second last paragraph in Section 8.

Figures 19-21 show query cost for binary equi-joins, band joins, and multiway equi-joins against different padding strategies in default datasets. Note that in all three padding strategies, we set the additional trusted memory size $M = 2B$ (B is the block size) in ODBJ and our method, but set $M = 50 \log N$ (N is the number of data blocks) in ObliDB as above. For ObliDB, Cartesian Product even achieves around 5X less query cost than Real Size and Closest Power. The reason is that Real Size and Closest Power need to additionally perform an oblivious filtering over the join output with Cartesian product size in ObliDB. For ODBJ and our method, the blowup of query cost in different padding strategies is roughly proportional

to different ratios of padded join result sizes to real join sizes. For example, Closest Power introduces within 2X larger query cost than Real Size, due to padding the join result size to the closest power of $x = 2$. In Cartesian Product, ODBJ needs 40X-50X larger query cost than ObliDB, since ODBJ only has $O(1)$ client memory size. Our method brings 500X-1700X and 900X-5300X larger query cost on binary and multiway equi-joins than ObliDB. The first reason is still that we have much less trusted memory size than ObliDB (as shown in Figures 7b and 8b). The second reason is that we perform a few tuple retrievals through B-tree searches over ORAMs in each join step, and each ORAM operation introduces $O(\log N)$ cost.

10 CONCLUSION

This paper focuses on oblivious joins over a cloud database. Our design integrates B-tree indices into ORAMs to speed up the query processing. We address the security issues in extended existing solutions and propose two types of oblivious algorithms including sort-merge join and index nested-loop join. Our method supports general binary and multiway equi-joins and some types of band joins. Extensive experimental evaluation has demonstrated the superior efficiency and scalability, when being compared against other alternatives and state-of-the-art baselines in the literature. Our current design does not address challenges associated with ad-hoc updates, which is a future direction to explore.

A TPC-H QUERIES

Binary Equi-Join.

Query TE1: Suppliers and customers in the same nations.

```
SELECT s_suppkey, c_custkey, s_nationkey
FROM supplier, customer
WHERE s_nationkey = c_nationkey;
```

Query TE2: Suppliers in the same nations.

```
SELECT s1.s_suppkey, s2.s_suppkey, s1.s_nationkey
FROM supplier s1, supplier s2
WHERE s1.s_nationkey = s2.s_nationkey;
```

Query TE3: Customers in the same nations.

```
SELECT c1.c_custkey, c2.c_custkey, c1.c_nationkey
FROM customer c1, customer c2
WHERE c1.c_nationkey = c2.c_nationkey;
```

Band Join.

Query TB1: Suppliers joined with other suppliers with higher account balance.

```
SELECT s1.s_suppkey, s2.s_suppkey,
       s1.s_acctbal, s2.s_acctbal
FROM supplier s1, supplier s2
WHERE s1.s_acctbal < s2.s_acctbal;
```

Query TB2: Parts joined with other parts with higher retail price.

```
SELECT p1.p_partkey, p2.p_partkey,
       p1.p_retailprice, p2.p_retailprice
FROM part p1, part p2
WHERE p1.p_retailprice < p2.p_retailprice;
```

Multiway Equi-Join.

Query TM1: Lineitems joined with the orders they associated with and the customers who placed the orders.

```
SELECT c_custkey, o_orderkey, l_linenumber
FROM customer, orders, lineitem
WHERE   c_custkey = o_custkey
    AND l_orderkey = o_orderkey;
```

Query TM2: Suppliers and customers in the same regions.

```
SELECT s_suppkey, c_custkey, n1.n_nationkey,
       n2.n_nationkey, n1.n_regionkey
FROM supplier, customer, nation n1, nation n2
WHERE   s_nationkey = n1.n_nationkey
    AND c_nationkey = n2.n_nationkey
    AND n1.n_regionkey = n2.n_regionkey;
```

Query TM3: Suppliers and customers in the same nations with the perchase history of the customers.

```
SELECT n_nationkey, s_suppkey, c_custkey,
       o_orderkey, l_linenumber
FROM nation, supplier, customer, orders, lineitem
WHERE   n_nationkey = s_nationkey
    AND s_nationkey = c_nationkey
    AND c_custkey = o_custkey
    AND o_orderkey = l_orderkey;
```

B SOCIAL GRAPH QUERIES

Binary Equi-Join.

Query SE1: A popular user followed by an inactive user.

```
SELECT * FROM
popular-user p, inactive-user i
WHERE   p.dst = i.src;
```

Query SE2: A popular user followed by a normal user.

```
SELECT * FROM
popular-user p, normal-user n
WHERE   p.dst = n.src;
```

Query SE3: A normal user followed by a popular user.

```
SELECT * FROM
popular-user p, normal-user n
WHERE   p.src = n.dst;
```

Multiway Equi-Join.

Query SM1: A popular user who is followed by a normal user followed by an inactive user.

```
SELECT *
FROM popular-user p, normal-user n, inactive-user i
WHERE p.dst = n.src AND n.dst = i.src;
```

Query SM2: A popular user and a normal user who are followed by an inactive user.

```
SELECT *
FROM popular-user p, normal-user n, inactive-user i
WHERE p.dst = i.src AND n.dst = i.src;
```

Query SM3: An inactive user who is followed by a popular user, a normal user, and another inactive user.

```
SELECT *
FROM popular-user p, normal-user n,
     inactive-user i1, inactive-user i2
WHERE   i1.dst = p.src
    AND i1.dst = n.src
    AND i1.dst = i2.src;
```


ACKNOWLEDGMENTS

We would like to thank the anonymous SIGMOD reviewers for several helpful comments and suggestions. This work was supported by National Key R&D Program of China (2021YFB3101100) and the National Natural Science Foundation of China (61972308, U1736216). This work was partially done when Zhao Chang worked as an intern in the Database and Storage Lab at Alibaba DAMO Academy.

REFERENCES

[1] 2014. Big Data Benchmark. https://amplab.cs.berkeley.edu/benchmark/. (2014).
[2] 2016. SEAL-ORAM. https://github.com/InitialDLab/SEAL-ORAM. (2016).
[3] 2020. Oblivious Database Join Algorithm. https://git.uwaterloo.ca/skrastni/obliv-join-impl. (2020).
[4] 2021. Towards Practical Oblivious Join. https://anonymous.4open.science/r/Towards-Practical-Oblivious-Join-3477/ojoin.pdf. (2021).
[5] Rakesh Agrawal, Dmitri Asonov, Murat Kantarcioglu, and Yaping Li. 2006. Sovereign Joins. In *ICDE*. 26.
[6] Rakesh Agrawal, Alexandre V. Evfimievski, and Ramakrishnan Srikant. 2003. Information Sharing Across Private Databases. In *SIGMOD*. 86–97.
[7] Miklós Ajtai, János Komlós, and Endre Szemerédi. 1983. An $O(n \log n)$ Sorting Network. In *STOC*. 1–9.
[8] Arvind Arasu, Spyros Blanas, Ken Eguro, Manas Joglekar, Raghav Kaushik, Donald Kossmann, Ravishankar Ramamurthy, Prasang Upadhyaya, and Ramarathnam Venkatesan. 2013. Secure database-as-a-service with Cipherbase. In *SIGMOD*. 1033–1036.
[9] Arvind Arasu, Ken Eguro, Manas Joglekar, Raghav Kaushik, Donald Kossmann, and Ravi Ramamurthy. 2015. Transaction processing on confidential data using Cipherbase. In *ICDE*. 435–446.
[10] Arvind Arasu, Ken Eguro, Raghav Kaushik, and Ravishankar Ramamurthy. 2014. Querying encrypted data. In *SIGMOD*. 1259–1261.
[11] Arvind Arasu and Raghav Kaushik. 2014. Oblivious Query Processing. In *ICDT*. 26–37.
[12] Gilad Asharov, T.-H. Hubert Chan, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. 2019. Locality-Preserving Oblivious RAM. In *EUROCRYPT, Part II*. 214–243.
[13] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. 2020. OptORAMa: Optimal Oblivious RAM. In *EUROCRYPT, Part II*. 403–432.
[14] Sumeet Bajaj and Radu Sion. 2014. TrustedDB: A Trusted Hardware-Based Database with Privacy and Data Confidentiality. *TKDE* 26, 3 (2014), 752–765.
[15] Kenneth E. Batcher. 1968. Sorting Networks and Their Applications. In *AFIPS Spring Joint Computing Conference*. 307–314.
[16] Johes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel N. Kho, and Jennie Rogers. 2017. SMCQL: Secure Query Processing for Private Data Networks. *PVLDB* 10, 6 (2017), 673–684.
[17] Johes Bater, Xi He, William Ehrich, Ashwin Machanavajjhala, and Jennie Rogers. 2018. Shrinkwrap: Efficient SQL Query Processing in Differentially Private Data Federations. *PVLDB* 12, 3 (2018), 307–320.
[18] Vincent Bindschaedler, Paul Grubbs, David Cash, Thomas Ristenpart, and Vitaly Shmatikov. 2018. The Tao of Inference in Privacy-Protected Databases. *PVLDB* 11, 11 (2018), 1715–1728.
[19] Vincent Bindschaedler, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, and Yan Huang. 2015. Practicing Oblivious Access on Cloud Storage: the Gap, the Fallacy, and the New Way Forward. In *CCS*. 837–849.
[20] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. 2015. Leakage-Abuse Attacks Against Searchable Encryption. In *CCS*. 668–679.
[21] Meeyoung Cha, Hamed Haddadi, Fabrício Benevenuto, and P. Krishna Gummadi. 2010. Measuring User Influence in Twitter: The Million Follower Fallacy. In *ICWSM*.
[22] Anrin Chakraborti, Adam J. Aviv, Seung Geol Choi, Travis Mayberry, Daniel S. Roche, and Radu Sion. 2019. rORAM: Efficient Range ORAM with $O(\log^2 N)$ Locality. In *NDSS*.
[23] Zhao Chang, Dong Xie, and Feifei Li. 2016. Oblivious RAM: A Dissection and Experimental Evaluation. *PVLDB* 9, 12 (2016), 1113–1124.
[24] Zhao Chang, Dong Xie, Feifei Li, Jeff M. Phillips, and Rajeev Balasubramonian. 2021. Efficient Oblivious Query Processing for Range and kNN Queries. *IEEE TKDE* (2021).
[25] Hao Chen, Ilaria Chillotti, and Ling Ren. 2019. Onion Ring ORAM: Efficient Constant Bandwidth Oblivious RAM from (Leveled) TFHE. In *CCS*. 345–360.
[26] Rui Chen, Haoran Li, A. K. Qin, Shiva Prasad Kasiviswanathan, and Hongxia Jin. 2016. Private spatial data aggregation in the local setting. In *ICDE*. 289–300.
[27] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. 2017. Detecting Privileged Side-Channel Attacks in Shielded Execution with Déjà Vu. In *AsiaCCS*. 7–18.
[28] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. 1998. Private Information Retrieval. *J. ACM* 45, 6 (1998), 965–981.
[29] Shumo Chu, Danyang Zhuo, Elaine Shi, and T.-H. Hubert Chan. 2021. Differentially Oblivious Database Joins: Overcoming the Worst-Case Curse of Fully Oblivious Algorithms. In *ITC 2021*. 19:1–19:24.
[30] Graham Cormode, Somesh Jha, Tejas Kulkarni, Ninghui Li, Divesh Srivastava, and Tianhao Wang. 2018. Privacy at Scale: Local Differential Privacy in Practice. In *SIGMOD*. 1655–1658.
[31] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* 2016 (2016), 86.
[32] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. 2018. Obladi: Oblivious Serializable Transactions in the Cloud. In *OSDI*. 727–743.
[33] Ankur Dave, Chester Leung, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. 2020. Oblivious coopetitive analytics using hardware enclaves. In *EuroSys*. 39:1–39:17.
[34] Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Saurabh Shintre. 2020. SEAL: Attack Mitigation for Encrypted Databases via Adjustable Leakage. In *USENIX Security*.
[35] Saba Eskandarian and Matei Zaharia. 2019. ObliDB: Oblivious Query Processing for Secure Databases. *PVLDB* 13, 2 (2019), 169–183.
[36] Christopher W. Fletcher, Ling Ren, Xiangyao Yu, Marten van Dijk, Omer Khan, and Srinivas Devadas. 2014. Suppressing the Oblivious RAM timing channel while making information leakage and program efficiency trade-offs. In *HPCA*. 213–224.
[37] Oded Goldreich. 1987. Towards a Theory of Software Protection and Simulation by Oblivious RAMs. In *STOC*. 182–194.
[38] Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *J. ACM* 43, 3 (1996), 431–473.
[39] Michael T. Goodrich. 2010. Randomized Shellsort: A Simple Oblivious Sorting Algorithm. In *SODA*. 1262–1277.
[40] Michael T. Goodrich. 2014. Zig-zag sort: A simple deterministic data-oblivious sorting algorithm running in $O(n \log n)$ time. In *STOC*. 684–693.
[41] Daniel Gruss, Julian Lettner, Felix Schuster, Olga Ohrimenko, István Haller, and Manuel Costa. 2017. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *USENIX Security*. 217–233.
[42] Hakan Hacigümüs, Balakrishna R. Iyer, Chen Li, and Sharad Mehrotra. 2002. Executing SQL over encrypted data in the database-service-provider model. In *SIGMOD*. 216–227.
[43] Zhian He, Wai Kit Wong, Ben Kao, David Wai-Lok Cheung, Rongbin Li, Siu-Ming Yiu, and Eric Lo. 2015. SDB: A Secure Query Processing System with Data Interoperability. *PVLDB* 8, 12 (2015), 1876–1879.
[44] Thang Hoang, Ceyhun D. Ozkaptan, Gabriel Hackebeil, and Attila A. Yavuz. 2018. Efficient Oblivious Data Structures for Database Services on the Cloud. *IEEE Trans. Cloud Computing* (2018).
[45] Thang Hoang, Muslum Ozgur Ozmen, Yeongjin Jang, and Attila A. Yavuz. 2019. Hardware-Supported ORAM in Effect: Practical Oblivious Search and Update on Very Large Dataset. *PoPETs* 2019, 1 (2019), 172–191.
[46] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation. In *NDSS*.
[47] Noah M. Johnson, Joseph P. Near, and Dawn Song. 2018. Towards Practical Differential Privacy for SQL Queries. *PVLDB* 11, 5 (2018), 526–539.
[48] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. 2016. Generic Attacks on Secure Outsourced Databases. In *CCS*. 1329–1340.
[49] Marcel Keller and Peter Scholl. 2014. Efficient, Oblivious Data Structures for MPC. In *ASIACRYPT, Part II*. 506–525.
[50] Taesoo Kim, Zhiqiang Lin, and Chia-che Tsai. 2017. CCS'17 Tutorial Abstract: SGX Security and Privacy. In *CCS*. 2613–2614.
[51] Tony Kontzer. 2004. Airlines and Hotels Face Customer Concerns Arising from Anti-Terrorism Efforts. *https://www.informationweek.com/privacy-pressure/d/d-id/1023945* (2004).
[52] Ios Kotsogiannis, Yuchao Tao, Xi He, Maryam Fanaeepour, Ashwin Machanavajjhala, Michael Hay, and Gerome Miklau. 2019. PrivateSQL: A Differentially Private SQL Query Engine. *PVLDB* 12, 11 (2019), 1371–1384.
[53] Simeon Krastnikov, Florian Kerschbaum, and Douglas Stebila. 2020. Efficient Oblivious Database Joins. *PVLDB* 13, 11 (2020), 2132–2145.
[54] Yaping Li and Minghua Chen. 2008. Privacy Preserving Joins. In *ICDE*. 1352–1354.
[55] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. 2015. ObliVM: A Programming Framework for Secure Computation. In *S&P*. 359–376.
[56] Jacob R. Lorch, Bryan Parno, James W. Mickens, Mariana Raykova, and Joshua Schiffman. 2013. Shroud: Ensuring private access to large-scale data in the data center. In *FAST*. 199–214.
[57] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. 2013. PHANTOM: Practical oblivious computation in a secure processor. In *CCS*. 311–324.
[58] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. 2018. Oblix: An Efficient Oblivious Search Index. In *S&P*. 279–296.
[59] Muhammad Naveed, Seny Kamara, and Charles V. Wright. 2015. Inference Attacks on Property-Preserving Encrypted Databases. In *CCS*. 644–655.
[60] Rafail Ostrovsky. 1990. Efficient Computation on Oblivious RAMs. In *STOC*. 514–523.
[61] Benny Pinkas and Tzachy Reinman. 2010. Oblivious RAM Revisited. In *CRYPTO*. 502–519.
[62] Raluca A. Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: Protecting confidentiality with encrypted query processing. In *SOSP*. 85–100.

[63] Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. 2015. Constants Count: Practical Improvements to Oblivious RAM. In *USENIX Security*. 415–430.

[64] Cetin Sahin, Tristan Allard, Reza Akbarinia, Amr El Abbadi, and Esther Pacitti. 2018. A Differentially Private Index for Range Query Processing in Clouds. In *ICDE*. 857–868.

[65] Cetin Sahin, Victor Zakhary, Amr El Abbadi, Huijia Lin, and Stefano Tessaro. 2016. TaoStore: Overcoming Asynchronicity in Oblivious Data Storage. In *S&P*. 198–217.

[66] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. 2018. ZeroTrace: Oblivious Memory Primitives from Intel SGX. In *NDSS*.

[67] Elaine Shi. 2020. Path Oblivious Heap: Optimal and Practical Oblivious Priority Queue. In *S&P*. 842–858.

[68] Emil Stefanov and Elaine Shi. 2013. ObliviStore: High Performance Oblivious Cloud Storage. In *S&P*. 253–267.

[69] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In *CCS*. 299–310.

[70] Stephen Tu, M. Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. 2013. Processing Analytical Queries over Encrypted Data. *PVLDB* 6, 5 (2013), 289–300.

[71] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. 2019. Conclave: Secure multi-party computation on big data. In *EuroSys*. 3:1–3:18.

[72] Ning Wang, Xiaokui Xiao, Yin Yang, Jun Zhao, Siu Cheung Hui, Hyejin Shin, Junbum Shin, and Ge Yu. 2019. Collecting and Analyzing Multidimensional Data with Local Differential Privacy. In *ICDE*. 638–649.

[73] Tianhao Wang, Jeremiah Blocki, Ninghui Li, and Somesh Jha. 2017. Locally Differentially Private Protocols for Frequency Estimation. In *USENIX Security*. 729–745.

[74] Tianhao Wang, Bolin Ding, Jingren Zhou, Cheng Hong, Zhicong Huang, Ninghui Li, and Somesh Jha. 2019. Answering Multi-Dimensional Analytical Queries under Local Differential Privacy. In *SIGMOD*. 159–176.

[75] Xiao Shaun Wang, Yan Huang, T.-H. Hubert Chan, Abhi Shelat, and Elaine Shi. 2014. SCORAM: Oblivious RAM for Secure Computation. In *CCS*. 191–202.

[76] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. 2014. Oblivious Data Structures. In *CCS*. 215–226.

[77] Yilei Wang and Ke Yi. 2021. Secure Yannakakis: Join-Aggregate Queries over Private Data. In *SIGMOD*.

[78] Peter Williams and Radu Sion. 2008. Usable PIR. In *NDSS*.

[79] Peter Williams, Radu Sion, and Alin Tomescu. 2012. PrivateFS: A parallel oblivious file system. In *CCS*. 977–988.

[80] Wai Kit Wong, Ben Kao, David Wai-Lok Cheung, Rongbin Li, and Siu-Ming Yiu. 2014. Secure query processing with data interoperability in a cloud database environment. In *SIGMOD*. 1395–1406.

[81] Dong Xie, Guanru Li, Bin Yao, Xuan Wei, Xiaokui Xiao, Yunjun Gao, and Minyi Guo. 2016. Practical Private Shortest Path Computation Based on Oblivious Storage. In *ICDE*. 361–372.

[82] Jianyu Yang, Tianhao Wang, Ninghui Li, Xiang Cheng, and Sen Su. 2020. Answering Multi-Dimensional Range Queries under Local Differential Privacy. *PVLDB* 14, 3 (2020), 378–390.

[83] Bin Yao, Feifei Li, and Xiaokui Xiao. 2013. Secure Nearest Neighbor Revisited. In *ICDE*. 733–744.

[84] Qingqing Ye, Haibo Hu, Xiaofeng Meng, and Huadi Zheng. 2019. PrivKV: Key-Value Data Collection with Local Differential Privacy. In *S&P*. 317–331.

[85] C. T. Yu and M. Z. Ozsoyoglu. 1979. An algorithm for tree-query membership of a distributed query. In *COMPSAC*. 306–312.

[86] Zhuoyue Zhao, Robert Christensen, Feifei Li, Xiao Hu, and Ke Yi. 2018. Random Sampling over Joins Revisited. In *SIGMOD*. 1525–1539.

[87] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. 2017. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *NSDI*. 283–298.