# Towards Practical Oblivious Join Processing

Zhao Chang, Dong Xie, Sheng Wang, Feifei Li, and Yulong Shen

**Abstract**—In cloud computing, remote accesses over the cloud 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 band joins and multiway equi-joins. For oblivious join without ORAMs, we extend the existing binary equi-join algorithm to support general band joins obliviously. For oblivious join with ORAMs, we integrate $B$-tree indices into ORAMs for each input table and retrieve blocks through the indices in join processing. The key point is to avoid retrieving tuples that make no contribution to the final join result and bound the number of accesses to each $B$-tree index. 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.

**Index Terms**—Data Privacy, Oblivious RAM, Oblivious Index, Oblivious Join.

## 1 Introduction

Many cloud service providers offer cloud-based database systems such as Amazon RDS and Redshift, Azure SQL, and Google Cloud SQL. Data encryption is a necessary step for keeping sensitive information secure and private on a cloud. To that end, encrypted databases such as Cipherbase [1], [2], CryptDB [3], TrustedDB [4], SDB [5], and Monomi [6], as well as related query execution techniques [7], [8], [9], [10] have been developed. But query access patterns still pose a privacy threat and leak sensitive information [11], [12], [13], [14]. It is possible to analyze the importance of different areas in the database, $e.g.$, by counting the frequency of accessing data items [15], [16], [17], [18]. With certain background knowledge, the server learns a lot about user queries and/or data [11], [19], [20].

Oblivious RAMs (ORAMs) [21], [22], [23] 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 [11]. Recent studies [14], [24], [25], [26], [27], [28] 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, existing solutions to integrating indices into ORAMs leak *the number of accesses to any index* in processing. We will address the security issue in our algorithms in Sections 6 and 7.

Joins are commonly used operations in relational databases. In this work, we consider the problem of computing join functions *in an oblivious way*. Li and Chen [29] first studies oblivious theta-joins, but their algorithms are no better than a Cartesian product. Arasu and Kaushik [13] presents oblivious algorithms for a rich class of database queries including equi-joins. However, Krastnikov $et\ al.$ [30] points out that the details in [13] are incomplete, and no practical implementation is provided to show the empirical results. Opaque [12] and ObliDB [31] are efficient *only* for the special case of one-to-many equi-join, $e.g.$, primary-foreign key join. Krastnikov $et\ al.$ [30] 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 join graph. ObliDB [31] offers an oblivious hash join algorithm to support general equi-joins over multiple tables, but it is equivalent to a Cartesian product.

In summary, prior studies are still unable to address the major challenge in oblivious joins. They are only efficient for foreign key joins [12], [31], or restricted to binary equi-joins [30], or purely theoretical in nature and not leading to practical implementations [13], [29].

Our major objective is to support general band joins and multiway equi-joins obliviously. Band join [34] is defined as a binary join between tables $T_1$ and $T_2$ on attributes $T_1.A$ and $T_2.B$ with the join condition $T_1.A - c_1 \leq T_2.B \leq T_1.A + c_2$. In particular a band join will reduce to a binary equi-join, when $c_1 = c_2 = 0$. First, we extend the binary equi-join algorithm in Krastnikov $et\ al.$ [30] to support general band joins obliviously. Second, we propose two band join algorithms using ORAMs: 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 bound the number of accesses to any index. 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

- Z. Chang and Y. Shen are with Xidian University, China. E-mail: changzhao@xidian.edu.cn, ylshen@mail.xidian.edu.cn.
- D. Xie is with The Pennsylvania State University. E-mail: dongx@psu.edu.
- S. Wang and F. Li are with Alibaba Group. E-mail: {sh.wang, lifeifei}@alibaba-inc.com.

*Manuscript received MON DD, 2022; revised XXX, 2022.*

TABLE 1
Comparison of oblivious join algorithms.

| | Join Type [a] | | | | Algorithm | Complexity Analysis [b] | | |
|---|---|---|---|---|---|---|---|---|
| | BE | BD | ME | | | Computation Overhead [c] | Cloud Storage | Client Storage |
| Li and Chen [29] | ✓ | ✓ | ✓ | BE | A1 | $\Omega(\prod_{j=1}^{\ell}\|T_j\|)$ | $O(\prod_{j=1}^{\ell}\|T_j\|)$ | $O(1)$ |
| | | | | BD | A2 | | $O(\|T_{\text{in}}\|+\|T_{\text{out}}\|)$ | $O(M)$ |
| | | | | ME | A3 | | $\Omega(\|T_{\text{in}}\|+\|T_{\text{out}}\|)$ | $O(M)$ |
| Arasu and Kaushik [13] | ✓ | × | ✓ | BE ME | Equi-Join | $O((\|T_{\text{in}}\|+\|T_{\text{out}}\|)\log^2(\|T_{\text{in}}\|+\|T_{\text{out}}\|))$ | $O(\|T_{\text{in}}\|+\|T_{\text{out}}\|)$ | $O(\log(\|T_{\text{in}}\|+\|T_{\text{out}}\|))$ |
| Opaque [12] | × | × | × | PF | Opaque Join | $O((\|T_{\text{in}}\|+\|T_{\text{out}}\|)\log^2((\|T_{\text{in}}\|+\|T_{\text{out}}\|)/M))$ | $O(\|T_{\text{in}}\|+\|T_{\text{out}}\|)$ | $O(M)$ |
| ObliDB [31] | ✓ | × | ✓ | PF | 0-OM Join | $O((\|T_{\text{in}}\|+\|T_{\text{out}}\|)\log^2(\|T_{\text{in}}\|+\|T_{\text{out}}\|))$ | $O(\|T_{\text{in}}\|+\|T_{\text{out}}\|)$ | $O(1)$ |
| | | | | BE ME | Hash Join | $O(\prod_{j=1}^{\ell}\|T_j\|)$ | $O(\prod_{j=1}^{\ell}\|T_j\|)$ | $O(M)$ |
| Krastnikov *et al.* [30] | ✓ | × | × | BE | Binary Join | $O((\|T_{\text{in}}\|+\|T_{\text{out}}\|)\log^2(\|T_{\text{in}}\|+\|T_{\text{out}}\|))$ | $O(\|T_{\text{in}}\|+\|T_{\text{out}}\|)$ | $O(1)$ |
| Ours [d] | ✓ | ✓ | ✓ | BD | | | | |
| | | | | BE BD | SMJ | $O((\|T_{\text{in}}\|+\|T_{\text{out}}\|)\cdot(\log_M(\|T_{\text{in}}\|+\|T_{\text{out}}\|)+\log\|T_{\text{in}}\|))$ | $O(\|T_{\text{in}}\|+\|T_{\text{out}}\|)$ | $O(\|T_{\text{in}}\|/B+M+\log\|T_{\text{in}}\|)$ |
| | | | | BE BD | INLJ (+Cache) | $O((\|T_1\|+\|T_{\text{out}}\|)\cdot(\log_M(\|T_1\|+\|T_{\text{out}}\|)+\log\|T_1\|+\Delta\log\|T_2\|))$ | $O(\|T_{\text{in}}\|+\|T_{\text{out}}\|)$ | $O(\|T_{\text{in}}\|/B+M+\log\|T_{\text{in}}\|)$ |
| | | | | ME | INLJ (+Cache) | $O((\|T_{\text{in}}\|+\|T_{\text{out}}\|)\cdot(\log_M(\|T_{\text{in}}\|+\|T_{\text{out}}\|)+\log\|T_1\|+\Delta\sum_{j=2}^{\ell}\log\|T_j\|))$ | $O(\|T_{\text{in}}\|+\|T_{\text{out}}\|)$ | $O(\|T_{\text{in}}\|/B+M+\sum_{j=1}^{\ell}\log\|T_j\|)$ |

[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 $|T_{\text{in}}| = \sum_{j=1}^{\ell}|T_j|$ and the real join result size as $|T_{\text{out}}|$. 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 [31]. In our implementation, we allow each block to contain multiple data tuples but have the same block size $B$.

[c]Let $m$ be the trusted storage size. We assume an oblivious sorting needs $O(n\log^2(n/m))$ time cost, as with Table 2 in [31] and Table 1 in [30]. We also assume an oblivious filter needs $O(n\log_m n)$ time cost based on oblivious compaction algorithm, as with Theorem 6 in [32].

[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 as $\Delta$. Note that recursive Path-ORAM [33] or oblivious $B$-tree [31] can reduce the $O(|T_{\text{in}}|/B)$ client storage cost for position map.

and 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.*, [35], [36], [37]) rather than Path-ORAM [33] to improve the performance. We can also leverage other types of indices (*e.g.*, Oblix [26]) 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 extend the binary equi-join algorithm in Krastnikov *et al.* [30] to support general band joins obliviously in Section 5. Note that existing studies (except [29]) do not work for any non-equi joins.
- We also propose two band join algorithms using ORAMs: sort-merge join and index nested-loop join in Section 6.1 and 6.2. The key point is to bound the number of accesses to each $B$-tree index.
- We support acyclic equi-joins over multiple tables obliviously using the index nested-loop join in Section 7. We 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 10. 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 [11]. We adopt Path-ORAM [33] due to good performance and simplicity. It can be replaced with some novel ORAMs (*e.g.*, [35], [36], [37]). A few advanced ORAMs [38], [39], [40], [41], [42], [43], [44] work on file systems, multiple clients, parallelization, asynchronicity and distributed data stores. We may leverage them as our secure ORAM storage, since we treat ORAM as a blackbox.

**Oblivious query processing.** A list of studies [12], [13], [29], [30], [31] works on oblivious joins (as discussed in Section 1). We follow the same definition as theirs (see Definition 1), which is different from the differentially oblivious join [45].

A series of studies focus on oblivious query processing. Xie *et al.* [46] proposes ORAM solutions to shortest path computation. ZeroTrace [47] supports oblivious get/put/insert operations over set/dictionary/list interfaces. Obladi [48] provides ACID transactions while hiding access patterns. OCQ [49] performs oblivious coopetitive analytics in a decentralized manner. Snoopy [50] designs an oblivious storage based on oblivious load balancer and subORAMs.

Note that existing solutions [12], [30], [31] rely on Trusted Execution Environments (TEE). However, TEE is orthogonal to oblivious algorithms and has no advantage to the obliviousness. If the server keeps secure enclaves (*e.g.*, Intel SGX [51], [52]), our client can be moved and co-located in server.

**Oblivious data structures.** Prior studies [14], [24], [25], [26], [27], [28] build oblivious indices or tree structures, but they do not protect *how many accesses to the data structure*. In our method, we integrate $B$-tree indices into ORAMs and address the security issue above. Some other indices (*e.g.*, Oblix [26]) also work for our method, as long as they support both point and range queries obliviously.

**Secure multi-party computation.** Secure multi-party computation (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 [49], [53], [54], [55], [56], [57] have a different problem setting from our cloud database setting.

**Differential privacy.** Differential privacy (DP) protects against attacks with guaranteed probabilistic accuracy. They build index [58] and key-value data collection [59], and support general SQL queries [60], [61], [62]. However, DP-based solutions [58], [59], [60], [61], [62], [63], [64], [65] 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 and encrypts these data blocks. If using ORAM, the client builds an ORAM data structure ($e.g.$, Path-ORAM) over the encrypted blocks and integrates some $B$-tree indices $I$ into the ORAM data structure using ORAM+$B$-tree or oblivious $B$-tree (see Section 4.2). Then, the client uploads the encrypted blocks or the ORAM data structure to the cloud storage, and keeps the encryption keys and other metadata ($e.g.$, ORAM stash and position map in Path-ORAM) at her side.

In online processing, the client issues join queries against the server. In oblivious join algorithms that will be described later, the client performs a series of oblivious operations or ORAM operations, which reads/writes blocks from/to the server and generates the query results.

### 3.2 Problem Definition

We follow the definition in Opaque [12] and ObliDB [31]. Let $D$ be a relational database (where some $B$-tree indices $I$ may also be integrated) and $Q$ be a join query. Let $\mathsf{Size}(D)$ 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 $\mathsf{Sch}(D)$ including table and column names in $D$ can be easily hidden using encryption. Let $\mathsf{IOSize}(D, Q)$ 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 $\mathsf{IOSize}(D, Q)$ and must be protected against the adversary. Let $\mathsf{Trace}$ be the trace of server location accesses and network traffic patterns while running $Q$ over $D$.

**Definition 1.** Oblivious Join [12]. *For any two relational databases $D$ and $D'$ and two join queries $Q$ and $Q'$, where $\mathsf{Size}(D)$ = $\mathsf{Size}(D')$, $\mathsf{Sch}(D)$ = $\mathsf{Sch}(D')$ and $\mathsf{IOSize}(D, Q)$ = $\mathsf{IOSize}(D', Q')$, we denote the access patterns produced by the join algorithm $\mathsf{OJoin}$ running $Q$ and $Q'$ over $D$ and $D'$ as $\mathsf{Trace}(\mathsf{OJoin}(D, Q))$ and $\mathsf{Trace}(\mathsf{OJoin}(D', Q'))$. $\mathsf{OJoin}$ is an oblivious join algorithm, if*

*1) $\mathsf{OJoin}$ ensures the confidentiality; and*

*2) access patterns $\mathsf{Trace}(\mathsf{OJoin}(D, Q))$ and $\mathsf{Trace}(\mathsf{OJoin}$ $(D', Q'))$ 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). 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 9 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 [66], [67], [68] can help to alleviate such leakage.

## 4 Preliminaries

### 4.1 ORAM and Oblivious Sorting

**ORAM.** ORAM [21], [22], [23] allows the client to access encrypted data in the server while hiding her access patterns. 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. ORAM hides 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.** Path-ORAM [33] organizes the ORAM data structure as a full binary tree where each node is a bucket with a fixed number of encrypted blocks. It 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* 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 $O(\log N)$ cost per query. A recursive Path-ORAM needs $O(\log N)$ client memory size and $O(\log_B N \cdot \log N)$ cost per query.

**Oblivious sorting.** A set of items can be sorted by accessing the items in a *fixed, predefined order*. Bitonic sort [69] needs $O(n \log^2 n)$ time cost but with small constant factor. Some advanced algorithms [70], [71], [72] achieve $O(n \log n)$ time cost. However, they may fail with a small probability [71], or lead to large constant factors [70] and non-trivial implementation [72]. Hence, most prior studies [11], [12], [29], [30], [31] adopt bitonic sort [69] and also extend it to an oblivious external sorting using a relatively large client memory [12], [31]. The time complexity is $O(n \log^2(n/m))$, where $m$ is trusted storage size.

**Oblivious filtering.** Dummy records can be removed by oblivious filtering. Prior studies [12], [13], [29] and the conference version [73] adopt an oblivious sorting to filter out dummy records. Actually, it can be done by oblivious compaction. OptORAMa [37] achieves this in $O(n)$ time but needs some non-trivial techniques. In our method, we adopt a simple oblivious compaction algorithm [32] with $O(n \log_m n)$ time cost, where $m$ is trusted storage size.

### 4.2 Integrate $B$-Tree Indices into ORAM

**ORAM+$B$-tree.** $B$-tree indices can be introduced to speed up the oblivious query processing [31], [74]. The client *ignores* the semantic difference of (encrypted) index and data blocks and stores all the blocks into an 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 through the ORAM.

**Oblivious $B$-tree.** To avoid storing the position map in the client, we can explore the idea of oblivious $B$-tree [31], [74]. 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.

**Index caching.** Index caching is a popular tree-based ORAM optimization [35], [47], [75]. We let the client cache *one specific level* of $B$-tree index to speed up the query performance. Due to large fanout in $B$-tree index, this overhead to the client storage is far less than storing the entire index.

**Segmenting ORAM.** We separate one single ORAM into multiple smaller ORAMs to reduce the cost of each ORAM access as in ObliDB [31]. For each input table, we build an ORAM for data blocks and another smaller one for index blocks. The comparison in Table 1 is based on this design. ObliDB [31] even suggests using a separate ORAM for each level of any $B$-tree index.

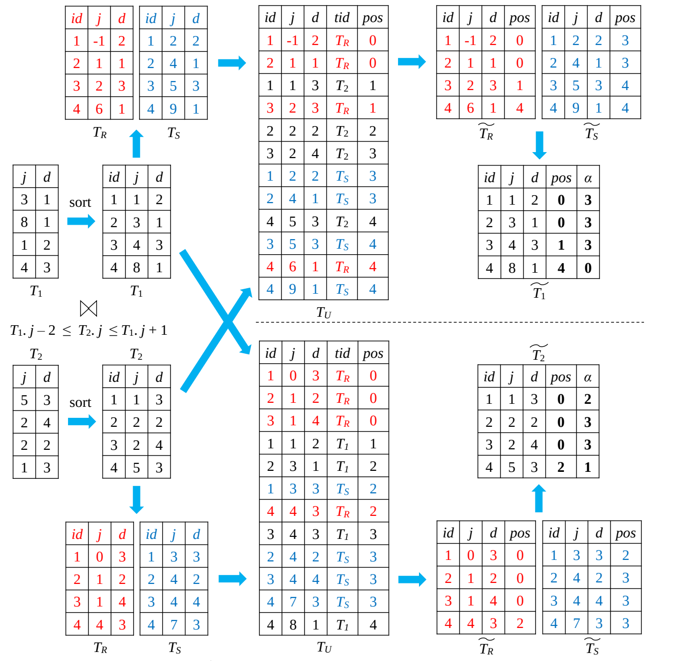
## 5 Oblivious Band Join without ORAM

Krastnikov *et al.* [30] is restricted to binary equi-join. In this section, we extend their sort-merge join algorithm to support general band join obliviously. First, we obliviously compute the degree information in the join graph, *i.e.*, how many times any input tuple will appear in the final join output. Second, we obliviously make copies for each tuple according to the join degree and perform an oblivious one-to-one mapping operation to generate the final join output. Note that existing studies (except [29]) do not work for any non-equi joins. In particular, we regard any binary equi-join as a special case of band joins.

### 5.1 Join Degree Computation

**Algorithm 1:** Join Degree Computation

```
Require: Input: two tables $T_1(j,d)$ and $T_2(j,d)$ with join
    condition $T_1.j - c_1 \leq T_2.j \leq T_1.j + c_2$.
    Output: $\tilde{T}_1(id, j, d, pos, \alpha)$ and $\tilde{T}_2(id, j, d, pos, \alpha)$.
 1: for $i \leftarrow 1$ to 2 do
 2:     $T_i \leftarrow$ OSort$(T_i)\langle j \uparrow, d \uparrow \rangle$;
 3:     $T_i(id, j, d) \leftarrow T_i(id \leftarrow$ ID$, j, d)$;        $\triangleright$ add $id$ column
 4: end for
 5: for $i \leftarrow 1$ to 2 do
 6:     $T_R(id, j, d) \leftarrow T_i(id, j - c_i, d)$;
 7:     $T_S(id, j, d) \leftarrow T_i(id, j + c_{3-i}, d)$;
 8:     $T_U(id, j, d, tid) \leftarrow T_R \cup T_S \cup T_{3-i}$;   $\triangleright$ add $tid$ column
 9:     $T_U \leftarrow$ OSort$(T_U)\langle j \uparrow, tid : T_R < T_{3-i} < T_S \rangle$;
10:     $T_U(id, j, d, tid, pos) \leftarrow$ Fill-Pos$(T_U)$;
11:     $T_U \leftarrow$ OSort$(T_U)\langle tid : T_R < T_S < T_{3-i}, id \uparrow \rangle$;
12:     $\tilde{T}_R \leftarrow \pi_{id,j,d,pos}(T_U[1 \ldots |T_i|])$;
13:     $\tilde{T}_S \leftarrow \pi_{id,j,d,pos}(T_U[|T_i| + 1 \ldots 2|T_i|])$;
14:     $\tilde{T}_i \leftarrow T_i(id, j, d, pos \leftarrow \tilde{T}_R.pos, \alpha \leftarrow \tilde{T}_S.pos - \tilde{T}_R.pos)$;
15: end for
16: return $\tilde{T}_1$ and $\tilde{T}_2$;
```



$T_1$

| $j$ | $d$ |
|---|---|
| 3 | 1 |
| 8 | 1 |
| 1 | 2 |
| 4 | 3 |

$T_1$

| $id$ | $j$ | $d$ |
|---|---|---|
| 1 | 1 | 2 |
| 2 | 3 | 1 |
| 3 | 4 | 3 |
| 4 | 8 | 1 |

$T_R$

| $id$ | $j$ | $d$ |
|---|---|---|
| 1 | -1 | 2 |
| 2 | 1 | 1 |
| 3 | 2 | 3 |
| 4 | 6 | 1 |

$T_S$

| $id$ | $j$ | $d$ |
|---|---|---|
| 1 | 2 | 2 |
| 2 | 4 | 1 |
| 3 | 5 | 3 |
| 4 | 9 | 1 |

$T_U$

| $id$ | $j$ | $d$ | $tid$ | $pos$ |
|---|---|---|---|---|
| 1 | -1 | 2 | $T_R$ | 0 |
| 2 | 1 | 1 | $T_R$ | 0 |
| 1 | 1 | 3 | $T_2$ | 1 |
| 3 | 2 | 3 | $T_R$ | 1 |
| 2 | 2 | 2 | $T_2$ | 2 |
| 3 | 2 | 4 | $T_2$ | 3 |
| 1 | 2 | 2 | $T_S$ | 3 |
| 2 | 4 | 1 | $T_S$ | 3 |
| 4 | 5 | 3 | $T_2$ | 4 |
| 3 | 5 | 3 | $T_S$ | 4 |
| 4 | 6 | 1 | $T_R$ | 4 |
| 4 | 9 | 1 | $T_S$ | 4 |

$\widetilde{T_R}$

| $id$ | $j$ | $d$ | $pos$ |
|---|---|---|---|
| 1 | -1 | 2 | 0 |
| 2 | 1 | 1 | 0 |
| 3 | 2 | 3 | 1 |
| 4 | 6 | 1 | 4 |

$\widetilde{T_S}$

| $id$ | $j$ | $d$ | $pos$ |
|---|---|---|---|
| 1 | 2 | 2 | 3 |
| 2 | 4 | 1 | 3 |
| 3 | 5 | 3 | 4 |
| 4 | 9 | 1 | 4 |

$\widetilde{T_1}$

| $id$ | $j$ | $d$ | $pos$ | $\alpha$ |
|---|---|---|---|---|
| 1 | 1 | 2 | **0** | **3** |
| 2 | 3 | 1 | **0** | **3** |
| 3 | 4 | 3 | **1** | **3** |
| 4 | 8 | 1 | **4** | **0** |

$T_2$

| $j$ | $d$ |
|---|---|
| 5 | 3 |
| 2 | 4 |
| 2 | 2 |
| 1 | 3 |

$T_2$

| $id$ | $j$ | $d$ |
|---|---|---|
| 1 | 1 | 3 |
| 2 | 2 | 2 |
| 3 | 2 | 4 |
| 4 | 5 | 3 |

$T_R$

| $id$ | $j$ | $d$ |
|---|---|---|
| 1 | 0 | 3 |
| 2 | 1 | 2 |
| 3 | 1 | 4 |
| 4 | 4 | 3 |

$T_S$

| $id$ | $j$ | $d$ |
|---|---|---|
| 1 | 3 | 3 |
| 2 | 4 | 2 |
| 3 | 4 | 4 |
| 4 | 7 | 3 |

$T_U$

| $id$ | $j$ | $d$ | $tid$ | $pos$ |
|---|---|---|---|---|
| 1 | 0 | 3 | $T_R$ | 0 |
| 2 | 1 | 2 | $T_R$ | 0 |
| 3 | 1 | 4 | $T_R$ | 0 |
| 1 | 1 | 2 | $T_1$ | 1 |
| 2 | 3 | 1 | $T_1$ | 2 |
| 1 | 3 | 3 | $T_S$ | 2 |
| 4 | 4 | 3 | $T_R$ | 2 |
| 3 | 4 | 3 | $T_1$ | 3 |
| 2 | 4 | 2 | $T_S$ | 3 |
| 3 | 4 | 4 | $T_S$ | 3 |
| 4 | 7 | 3 | $T_S$ | 3 |
| 4 | 8 | 1 | $T_1$ | 4 |

$\widetilde{T_R}$

| $id$ | $j$ | $d$ | $pos$ |
|---|---|---|---|
| 1 | 0 | 3 | 0 |
| 2 | 1 | 2 | 0 |
| 3 | 1 | 4 | 0 |
| 4 | 4 | 3 | 2 |

$\widetilde{T_S}$

| $id$ | $j$ | $d$ | $pos$ |
|---|---|---|---|
| 1 | 3 | 3 | 2 |
| 2 | 4 | 2 | 3 |
| 3 | 4 | 4 | 3 |
| 4 | 7 | 3 | 3 |

$\widetilde{T_2}$

| $id$ | $j$ | $d$ | $pos$ | $\alpha$ |
|---|---|---|---|---|
| 1 | 1 | 3 | **0** | **2** |
| 2 | 2 | 2 | **0** | **3** |
| 3 | 2 | 4 | **0** | **3** |
| 4 | 5 | 3 | **2** | **1** |

Fig. 1. An example of join degree computation.

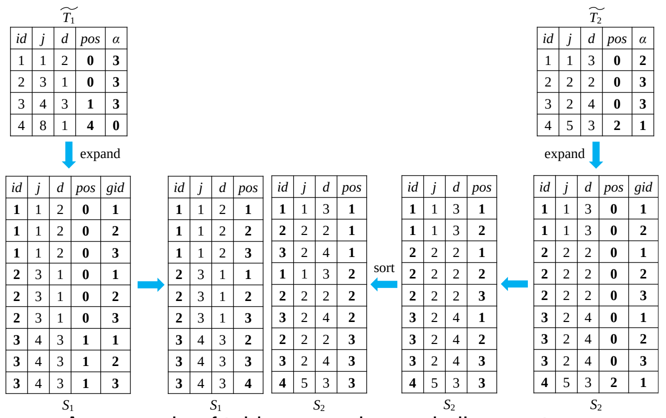
**Example 1.** *Algorithm 1 shows the details of join degree computation. An example is given in Figure 1. For each input table $T_i(j,d)$, we denote the join key as $j$ and the remaining attributes as $d$. We mainly focus on the join degree computation on table $T_1$, and that on $T_2$ goes in a similar way.*

*First, we obliviously sort $T_1$ and $T_2$ lexicographically by $(j,d)$, and add a unique identifier $id$ to each tuple (Line 1-4). In Algorithm 1, we parameterize any oblivious sorting with a lexicographic ordering on chosen attributes. For example, OSort$(T_i)\langle j \uparrow, d \uparrow \rangle$ (Line 2) will sort the tuples in $T_i$ by increasing $j$ attribute, followed by increasing $d$ attributes.*

*Then, we aim to generate augmented tables $\tilde{T}_1$ and $\tilde{T}_2$ with join degree $\alpha$ and position $pos$, such that each tuple $\tilde{t}_1 \in \tilde{T}_1$ (e.g., $(3,4,3,1,3) \in \tilde{T}_1$) matches $\tilde{t}_1.\alpha$ tuples in $\tilde{T}_2$, where each matched tuple $\tilde{t}_2$ has a unique $\tilde{t}_2.id \in [\tilde{t}_1.pos + 1, \tilde{t}_1.pos + \tilde{t}_1.\alpha]$ (e.g., $(2,2,2,0,3)$, $(3,2,4,0,3)$ and $(4,5,3,2,1)$ with $id \in [2,4]$) (Line 5-15).*

*Specifically, we generate two auxiliary tables $T_R$ and $T_S$ for $T_1$, where $T_R.j \leftarrow T_1.j - c_1$ and $T_S.j \leftarrow T_1.j + c_2$ (Line 6-7). Suppose tuple $t_1 \in T_1$ (e.g., $(3,4,3) \in T_1$) corresponds to tuples $t_R \in T_R$ (e.g., $(3,2,3) \in T_R$) and $t_S \in T_S$ (e.g., $(3,5,3) \in T_S$). According to the join condition, any tuple $t_2 \in T_2$ with $t_2.j \in [t_R.j, t_S.j]$ will match $t_1 \in T_1$ (e.g., $(2,2,2)$, $(3,2,4)$ and $(4,5,3) \in T_2$ with $j \in [2,5]$ are 3 matches).*

*Then, we compute a union $T_U$ of tables $T_R$, $T_S$ and $T_2$ (Line 8). In particular, we add a tid attribute to indicate which table each tuple is originally from (e.g., $(3,2,3,T_R)$, $(3,5,3,T_S)$, $(2,2,2,T_2)$). We obliviously sort $T_U$ lexicographically by $(j, tid)$ (Line 9). In particular, for tuples with the same join key $j$, their ordering is based on $tid : T_R < T_2 < T_S$ (e.g., $(3,2,3,T_R) < (2,2,2,T_2)/(3,2,4,T_2) < (1,2,2,T_S)$). Hence, tuple $t_2 \in T_2$ matches tuple $t_1 \in T_1$, if and only if $t_2$ ranks between $t_1$'s*



$\widetilde{T_1}$

| id | j | d | pos | α |
|---|---|---|---|---|
| 1 | 1 | 2 | 0 | 3 |
| 2 | 3 | 1 | 0 | 3 |
| 3 | 4 | 3 | 1 | 3 |
| 4 | 8 | 1 | 4 | 0 |

$\widetilde{T_2}$

| id | j | d | pos | α |
|---|---|---|---|---|
| 1 | 1 | 3 | 0 | 2 |
| 2 | 2 | 2 | 0 | 3 |
| 3 | 2 | 4 | 0 | 3 |
| 4 | 5 | 3 | 2 | 1 |

$S_1$

| id | j | d | pos | gid |
|---|---|---|---|---|
| 1 | 1 | 2 | 0 | 1 |
| 1 | 1 | 2 | 0 | 2 |
| 1 | 1 | 2 | 0 | 3 |
| 2 | 3 | 1 | 0 | 1 |
| 2 | 3 | 1 | 0 | 2 |
| 2 | 3 | 1 | 0 | 3 |
| 3 | 4 | 3 | 1 | 1 |
| 3 | 4 | 3 | 1 | 2 |
| 3 | 4 | 3 | 1 | 3 |

$S_1$

| id | j | d | pos |
|---|---|---|---|
| 1 | 1 | 2 | 1 |
| 1 | 1 | 2 | 2 |
| 1 | 1 | 2 | 3 |
| 2 | 3 | 1 | 1 |
| 2 | 3 | 1 | 2 |
| 2 | 3 | 1 | 3 |
| 3 | 4 | 3 | 2 |
| 3 | 4 | 3 | 3 |
| 3 | 4 | 3 | 4 |

$S_2$

| id | j | d | pos |
|---|---|---|---|
| 1 | 1 | 3 | 1 |
| 2 | 2 | 2 | 1 |
| 3 | 2 | 4 | 1 |
| 1 | 1 | 3 | 2 |
| 2 | 2 | 2 | 2 |
| 3 | 2 | 4 | 2 |
| 2 | 2 | 2 | 3 |
| 3 | 2 | 4 | 3 |
| 4 | 5 | 3 | 3 |

$S_2$

| id | j | d | pos |
|---|---|---|---|
| 1 | 1 | 3 | 1 |
| 1 | 1 | 3 | 2 |
| 2 | 2 | 2 | 1 |
| 2 | 2 | 2 | 2 |
| 2 | 2 | 2 | 3 |
| 3 | 2 | 4 | 1 |
| 3 | 2 | 4 | 2 |
| 3 | 2 | 4 | 3 |
| 4 | 5 | 3 | 3 |

$S_2$

| id | j | d | pos | gid |
|---|---|---|---|---|
| 1 | 1 | 3 | 0 | 1 |
| 1 | 1 | 3 | 0 | 2 |
| 2 | 2 | 2 | 0 | 1 |
| 2 | 2 | 2 | 0 | 2 |
| 2 | 2 | 2 | 0 | 3 |
| 3 | 2 | 4 | 0 | 1 |
| 3 | 2 | 4 | 0 | 2 |
| 3 | 2 | 4 | 0 | 3 |
| 4 | 5 | 3 | 2 | 1 |

Fig. 2. An example of table expansion and alignment.

**Algorithm 2:** Table Expansion and Alignment

```
Require: Input: two tables $\tilde{T}_1(id, j, d, pos, \alpha)$ and
    $\tilde{T}_2(id, j, d, pos, \alpha)$ with band join parameters $c_1$ and $c_2$.
    Output: join result table $T_{\text{out}}(j_1, d_1, j_2, d_2)$.
1: for $i \leftarrow 1$ to 2 do
2:     $S_i(id, j, d, pos) \leftarrow$ OExpand$(\tilde{T}_i, \alpha)$;
3:     $S_i(id, j, d, pos, gid) \leftarrow S_i(id, j, d, pos, gid \leftarrow \text{ID}_{id})$;
4:     $S_i(id, j, d, pos) \leftarrow S_i(id, j, d, pos \leftarrow pos + gid)$;
5: end for
6: $S_2 \leftarrow$ OSort$\langle pos \uparrow, id \uparrow\rangle(S_2)$;
7: $T_{\text{out}}(j_1, d_1, j_2, d_2) \leftarrow (S_1.j, S_1.d, S_2.j, S_2.d)$;
8: return $T_{\text{out}}$;
```

*corresponding tuples $t_R \in T_R$ and $t_S \in T_S$ (e.g., $(2,2,2,T_2)$, $(3,2,4,T_2)$ and $(4,5,3,T_2)$ are 3 matches of $(3,4,3) \in T_1$, and they all rank between $(3,2,3,T_R)$ and $(3,5,3,T_S)$ in $T_U$).*

*Now, we add a pos attribute in $T_U$ by invoking* Fill-Pos$(T_U)$ *(Line 10). It iterates over each tuple $t_U \in T_U$, counts the tuples with $tid = T_2$, and assigns the current number to $t_U.pos$.*

*After that, we extract the augmented $T_R$ and $T_S$ (denoted as $\tilde{T}_R$ and $\tilde{T}_S$) from $T_U$. To accomplish this, we re-sort $T_U$ lexicographically by $(tid, id)$, where the ordering on attribute tid is $T_R < T_S < T_2$ (Line 11). After re-sorting, we ensure that the first $|T_1|$ rows of $T_U$ correspond to $\tilde{T}_R$ (augmented and sorted by id), and the second $|T_1|$ rows correspond to $\tilde{T}_S$ (Line 12-13).*

*Finally, we generate the augmented $T_1$ (denoted as $\tilde{T}_1$) from $\tilde{T}_R$ and $\tilde{T}_S$, where the position $\tilde{T}_1.pos \leftarrow \tilde{T}_R.pos$ and the join degree $\alpha \leftarrow \tilde{T}_S.pos - \tilde{T}_R.pos$ (Line 14).*

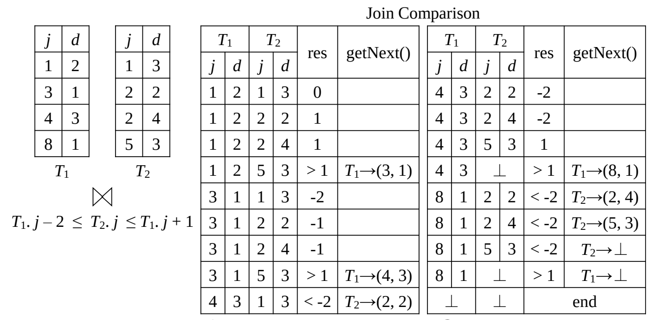
*The join degree computation on table $T_2$ goes in a similar way.*

Algorithm 1 takes $O((|T_1| + |T_2|)\log^2(|T_1| + |T_2|))$ time cost, since oblivious sorting dominates the time complexity.

### 5.2 Table Expansion and Alignment

**Example 2.** *Algorithm 2 shows the details of table expansion and alignment. An example is given in Figure 2. After obtaining the join degree $\alpha$, we need to make copies for each tuple based on the join degree (aka* table expansion*) and perform an one-to-one matching between two expanded tables (aka* table alignment*). For example, in Figure 2, for tuple $\tilde{t}_1 = (3,4,3,1) \in \tilde{T}_1(id, j, d, pos)$, we need to make $\tilde{t}_1.\alpha = 3$ copies and match the $\tilde{t}_1.\alpha = 3$ tuples in $\tilde{T}_2(id, j, d, pos)$ with $\tilde{T}_2.id \in [\tilde{t}_1.pos + 1, \tilde{t}_1.pos + \tilde{t}_1.\alpha] = [2,4]$ (i.e., $(2,2,2,0)$, $(3,2,4,0)$ and $(4,5,3,2)$ in $\tilde{T}_2$).*

*First, we obliviously expand each $\tilde{T}_i$ into table $S_i$, i.e., $S_i$ consists of $\alpha$ (contiguous) copies of each tuple $(id, j, d, pos) \in \tilde{T}_i$ (Line 2). The oblivious expansion algorithm is the same as Algorithm 4 in [30].*


Join Comparison
$T_1.j - 2 \leq T_2.j \leq T_1.j + 1$
$\bowtie$

$T_1$

| j | d |
|---|---|
| 1 | 2 |
| 3 | 1 |
| 4 | 3 |
| 8 | 1 |

$T_2$

| j | d |
|---|---|
| 1 | 3 |
| 2 | 2 |
| 2 | 4 |
| 5 | 3 |

| $T_1$ | | $T_2$ | | res | getNext() |
|---|---|---|---|---|---|
| j | d | j | d | | |
| 1 | 2 | 1 | 3 | 0 | |
| 1 | 2 | 2 | 2 | 1 | |
| 1 | 2 | 2 | 4 | 1 | |
| 1 | 2 | 5 | 3 | > 1 | $T_1 \rightarrow (3, 1)$ |
| 3 | 1 | 1 | 3 | -2 | |
| 3 | 1 | 2 | 2 | -1 | |
| 3 | 1 | 2 | 4 | -1 | |
| 3 | 1 | 5 | 3 | > 1 | $T_1 \rightarrow (4, 3)$ |
| 4 | 3 | 1 | 3 | < -2 | $T_2 \rightarrow (2, 2)$ |

| $T_1$ | | $T_2$ | | res | getNext() |
|---|---|---|---|---|---|
| j | d | j | d | | |
| 4 | 3 | 2 | 2 | -2 | |
| 4 | 3 | 2 | 4 | -2 | |
| 4 | 3 | 5 | 3 | 1 | |
| 4 | 3 | $\perp$ | | > 1 | $T_1 \rightarrow (8, 1)$ |
| 8 | 1 | 2 | 2 | < -2 | $T_2 \rightarrow (2, 4)$ |
| 8 | 1 | 2 | 4 | < -2 | $T_2 \rightarrow (5, 3)$ |
| 8 | 1 | 5 | 3 | < -2 | $T_2 \rightarrow \perp$ |
| 8 | 1 | $\perp$ | | > 1 | $T_1 \rightarrow \perp$ |
| $\perp$ | | $\perp$ | | end | |

Fig. 3. An example of sort-merge join with ORAMs.

*Then, we obliviously align $S_2$ with $S_1$ so that each join record corresponds to a row of $S_1$ and a row of $S_2$ with matching index.*

*1) We perform a grouping identity operation by scanning each $S_i$ (Line 3). All tuples in $S_i$ with the same id (i.e., originated from the same $\tilde{T}_i$ tuple) belong to the same group, and each gets a different identifier $gid \leftarrow \text{ID}_{id}$.*

*2) We update pos attribute as $pos \leftarrow pos + gid$ in table $S_i$ (Line 4). After that, we will let any tuple $s_2 \in S_2$ match the only one tuple $s_1 \in S_1$, where $s_1.id = s_2.pos$ and $s_1.pos = s_2.id$.*

*3) After 2), $S_1$ has been permutated lexicographically by $(id, pos)$. Hence, we obliviously sort table $S_2$ lexicographically by $(pos, id)$ to achieve the table alignment (Line 6).*

*4) Finally, we generate the join output table $T_{out}$ by simply concatenating $(j, d)$ attributes in $S_1$ and $S_2$ (Line 7).*

Algorithm 2 consists of two parts: table expansion and table alignment. For oblivious table expansion, the time cost is $O((|T_1| + |T_2|)\log^2(|T_1| + |T_2|) + |R_{\text{real}}|\log|R_{\text{real}}|)$ as with [30]. For oblivious table alignment, the time cost is $O(|R_{\text{real}}|\log^2|R_{\text{real}}|)$, since oblivious sorting dominates the time complexity. Hence, the total time cost is $O((|T_1| + |T_2|)\log^2(|T_1| + |T_2|) + |R_{\text{real}}|\log^2|R_{\text{real}}|)$.

## 6 Oblivious Band Join with ORAMs

We support two types of oblivious band joins with ORAMs: sort-merge join and index nested-loop join.

### 6.1 Oblivious Sort-Merge Join

Our algorithm is similar to the traditional sort merge join but with some differences. In 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.

**Example 3.** *Algorithm 3 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.*

**Algorithm 3:** Oblivious Sort-Merge Band Join

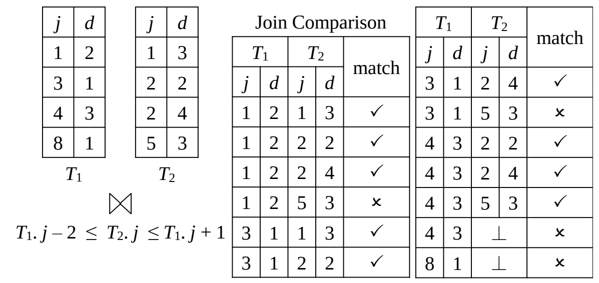
```
Require: Input: two tables $T_1(j, d)$ and $T_2(j, d)$ with join
    condition $T_1.j - c_1 \le T_2.j \le T_1.j + c_2$.
    Output: join result table $T_{\text{out}}$.
1: Initialize $T_{\text{out}} \leftarrow \emptyset$.
2: Initialize $t_1, t_2 \leftarrow \emptyset$.
3: for $i \leftarrow 1$ to 2 do
4:     $t_i \leftarrow T_i$.getFirst();
5: end for
6: while $t_1 \neq \perp$ or $t_2 \neq \perp$ do
7:     res $\leftarrow t_2.j - t_1.j$;
8:     if $-c_1 \le$ res $\le c_2$ then
9:         begin $\leftarrow t_2$;
10:        while $-c_1 \le$ res $\le c_2$ do
11:            $T_{\text{out}}$.put(Join($t_1, t_2$));
12:            $T_1$.getDummy(); $t_2 \leftarrow T_2$.getNext();
13:            res $\leftarrow t_2.j - t_1.j$;
14:        end while
15:        $T_{\text{out}}$.put($\perp$);
16:        $t_2 \leftarrow$ begin;
17:        $t_1 \leftarrow T_1$.getNext(); $T_2$.getDummy();
18:    else
19:        $T_{\text{out}}$.put($\perp$);
20:        if res $> c_2$ then
21:            $t_1 \leftarrow T_1$.getNext(); $T_2$.getDummy();
22:        else
23:            $T_1$.getDummy(); $t_2 \leftarrow T_2$.getNext();
24:        end if
25:    end if
26: end while
27: $T_{\text{out}} \leftarrow$ OFilter($T_{\text{out}}$);
28: return $T_{\text{out}}$;
```

*An example is given in Figure 3. First, Algorithm 3 initializes $T_{\text{out}} := \emptyset$ (Line 1). Then, we retrieve the first two tuples (e.g., $T_1(1, 2)$ and $T_2(1, 3)$) from $T_1$ and $T_2$ as $t_1$ and $t_2$ (Line 2-5). While either $t_1$ or $t_2$ is real, we compute the join comparison result "res" between them (Line 6-7). We keep the* invariant *above that we always pull tuples from $T_1$ and $T_2$ alternatively for either of two possible cases:*

*1) $t_1$ matches $t_2$ (e.g., $T_1(1, 2)$ and $T_2(1, 3)$ (Line 8)). First, we save the current $t_2$ (e.g., $T_2(1, 3)$) to a temporary tuple "begin" (Line 9). We keep writing out the join record (i.e., Join($t_1, t_2$) to $T_{\text{out}}$, and retrieving the next tuple (e.g., $T_2(2, 2)$) from $T_2$ as $t_2$, until the newly retrieved $t_2$ does not match $t_1$ (e.g., $T_2(5, 3)$ does not match $T_1(1, 2)$) (Line 10-14). Whenever we invoke a getNext() from $T_2$, we also pull a dummy tuple from $T_1$ to ensure the obliviousness (Line 12). Once they do not match, we write out a dummy record and assign "begin" (e.g., $T_2(1, 3)$) back to $t_2$ (Line 15-16). Then, we will retrieve the next tuple (e.g., $T_1(3, 1)$) from $T_1$, and also pull a dummy tuple from $T_2$ (Line 17). Finally, we move to the next iteration (Line 6-7).*

*2) $t_1$ does not match $t_2$ (e.g., $T_1(4, 3)$ and $T_2(1, 3)$ (Line 18)). Since they do not match, we first write out a dummy record (Line 19). If res $> c_2$, we retrieve the next tuple from $T_1$ (Line 20-21). Otherwise (i.e., res $< -c_1$ for $T_1(4, 3)$ and $T_2(1, 3)$), we retrieve the next tuple (e.g., $T_2(2, 2)$) from $T_2$ (Line 22-23). Note that in either branch we also pull a dummy tuple from the other table. Finally, we move to the next iteration (Line 6-7).*

*Note that once a cursor moves to the end of table $T_1$ or $T_2$, we will retrieve a dummy tuple $\perp$ from the table and logically let $\perp.j = +\infty$. For example, when the cursor on $T_1$ moves to tuple $T_1(4, 3)$ and that on $T_2$ reaches the end of $T_2$, we will retrieve a dummy tuple $\perp$ from $T_2$ (Line 12) and let the join comparison result res $= +\infty > c_2$ (Line 13). The rest still goes in the same way as stated above.*



| $j$ | $d$ |
|---|---|
| 1 | 2 |
| 3 | 1 |
| 4 | 3 |
| 8 | 1 |

$T_1$

| $j$ | $d$ |
|---|---|
| 1 | 3 |
| 2 | 2 |
| 2 | 4 |
| 5 | 3 |

$T_2$

$\bowtie$

$T_1.j - 2 \le T_2.j \le T_1.j + 1$

Join Comparison

| $T_1$ | | $T_2$ | | match |
|---|---|---|---|---|
| $j$ | $d$ | $j$ | $d$ | |
| 1 | 2 | 1 | 3 | ✓ |
| 1 | 2 | 2 | 2 | ✓ |
| 1 | 2 | 2 | 4 | ✓ |
| 1 | 2 | 5 | 3 | × |
| 3 | 1 | 1 | 3 | ✓ |
| 3 | 1 | 2 | 2 | ✓ |

| $T_1$ | | $T_2$ | | match |
|---|---|---|---|---|
| $j$ | $d$ | $j$ | $d$ | |
| 3 | 1 | 2 | 4 | ✓ |
| 3 | 1 | 5 | 3 | × |
| 4 | 3 | 2 | 2 | ✓ |
| 4 | 3 | 2 | 4 | ✓ |
| 4 | 3 | 5 | 3 | ✓ |
| 4 | 3 | $\perp$ | | × |
| 8 | 1 | $\perp$ | | × |

Fig. 4. An example of index nested-loop join with ORAMs.

**Algorithm 4:** Oblivious Index Nested-Loop Band Join

```
Require: Input: two tables $T_1(j, d)$ and $T_2(j, d)$ with join
    condition $T_1.j - c_1 \le T_2.j \le T_1.j + c_2$.
    Output: join result table $T_{\text{out}}$.
1: Initialize $T_{\text{out}} \leftarrow \emptyset$.
2: Initialize $t_1, t_2 \leftarrow \emptyset$.
3: for $i \leftarrow 1$ to $|T_1|$ do
4:     $t_1 \leftarrow T_1$.getNext();
5:     $t_2 \leftarrow T_2$.getFirst($t_1.j - c_1$);
6:     while $t_1.j - c_1 \le t_2.j \le t_1.j + c_2$ do
7:         $T_{\text{out}}$.put(Join($t_1, t_2$));
8:         $T_1$.getDummy();
9:         $t_2 \leftarrow T_2$.getNext();
10:    end while
11:    $T_{\text{out}}$.put($\perp$);
12: end for
13: $T_{\text{out}} \leftarrow$ OFilter($T_{\text{out}}$);
14: return $T_{\text{out}}$;
```

*After both cursors reach the end of tables $T_1$ and $T_2$, the final step is to obliviously filter out dummy records from $T_{out}$ (see "Oblivious filtering" in Section 4.1) and only keep real join records (Line 27).*

Note that $B$-tree indices are not required for Algorithm 3. If each tuple keeps the pointer to the next tuple, succeeding tuples can be retrieved when needed through ORAM using the pointers.

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

In Figure 3, two input table sizes $|T_1| = 4$ and $|T_2| = 4$, the real join size $|R_{\text{real}}| = 9$, and the number of tuple retrievals from each input table $\text{Num}_{\text{tr}} = |T_1| + |T_2| + |R_{\text{real}}| + 1 = 18$.

## 6.2 Oblivious Index Nested-Loop Join

In our index nested-loop join, we integrate $B$-tree indices into ORAMs for each input table and retrieve tuples by querying the indices through ORAMs. 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 $T_2$ through

1. Due to space limit, proofs of theorems, complexity analyses on our algorithms, and implementation details of multiway equi-join algorithm are given in the full version [76].

$B$-tree index. In each join step, we ensure the *invariant* that we retrieve the tuple needed from each input table *alternatively*. A dummy tuple is retrieved from table $T_1$ as necessary. 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 out the join record; otherwise, a dummy record is output as necessary.

**Example 4.** *Algorithm 4 shows the details of joining two tables $T_1$ and $T_2$. An example is given in Figure 4. Algorithm 4 begins with initializing an empty output table $T_{\text{out}}$ (Line 1). The outer loop is to iterate over each tuple in table $T_1$ (Line 3). Each time we retrieve a new tuple $t_1$ (e.g., $T_1(1,2)$) from $T_1$ (Line 4), we first retrieve a tuple $t_2$ (e.g., $T_2(1,3)$) from $T_2$, which is the first tuple satisfying $t_2.j \geq t_1.j - c_1$ (Line 5). If those two tuples can match (i.e., $t_1.j - 2 \leq t_2.j \leq t_1.j + 1$), we write the join record (e.g., Join$(t_1, t_2)$) to the output table $T_{\text{out}}$ (Line 7) and retrieve the next tuple (e.g., $T_2(2,2)$) from $T_2$ as $t_2$ (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 $t_2$ does not match the current $t_1$ (e.g., $T_2(5,3)$ does not match $T_1(1,2)$). Once they do not match, we write out a dummy record (Line 11) and step into the next iteration (e.g., processing the next tuple $T_1(3,1)$ 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 (e.g., the last two rows in Join Comparison in Figure 4). 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 13).*

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 $R_{\text{real}}$, let $\text{Num}_{\text{tr}}$ 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 $\text{Num}_{\text{tr}} = f(|T_1|, |T_2|, |R_{\text{real}}|) = |T_1| + |R_{\text{real}}|$.*

In Figure 4, the first input table size $|T_1| = 4$, the real join size $|R_{\text{real}}| = 9$, and the number of tuple retrievals from each input table $\text{Num}_{\text{tr}} = |T_1| + |R_{\text{real}}| = 4 + 9 = 13$.

## 7 Oblivious Multiway Equi-Join

Recent work [12], [30], [31] 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). Arasu and Kaushik [13] 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.* [30] points out that details in [13] are incomplete, and no practical implementation is provided to show the empirical results.

In this work, we extend our Algorithm 4 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.

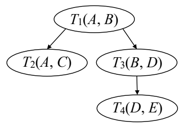


Table $T_1$

| $A$ | $B$ |
|---|---|
| 1 | 1 |
| 2 | 1 |
| 2 | 2 |
| 2 | 3 |

Table $T_2$

| $A$ | $C$ |
|---|---|
| 1 | 1 |
| 2 | 1 |
| 2 | 2 |
| 3 | 1 |

$\bowtie$

Table $T_3$

| $B$ | $D$ |
|---|---|
| ~~1~~ | ~~4~~ |
| 2 | 1 |
| ~~2~~ | ~~3~~ |

Table $T_4$

| $D$ | $E$ |
|---|---|
| 1 | 2 |
| 2 | 1 |
| 2 | 3 |

Join Comparison

| $T_1$ | | $T_2$ | | $T_3$ | | $T_4$ | | match |
|---|---|---|---|---|---|---|---|---|
| $A$ | $B$ | $A$ | $C$ | $B$ | $D$ | $D$ | $E$ | |
| 1 | 1 | 1 | 1 | 1 | 4 | $\perp$ | | false |
| | | $\perp$ | | ~~1~~ | ~~4~~ | $\perp$ | | disable |
| 2 | 1 | 2 | 1 | 2 | 1 | $\perp$ | | false |
| 2 | 2 | 2 | 1 | 2 | 1 | 1 | 2 | true |
| | | 2 | 1 | 2 | 3 | $\perp$ | | false |
| | | $\perp$ | | ~~2~~ | ~~3~~ | $\perp$ | | disable |
| | | 2 | 2 | 2 | 1 | 1 | 2 | true |
| 2 | 3 | 2 | 1 | $\perp$ | | $\perp$ | | false |
| padding dummy operations | | | | | | | | |

Fig. 5. An example of oblivious multiway equi-join.

**Example 5.** *Figure 5 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 5. 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 [77]. 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, 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 5). 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 $T_{p(j)}$,* tuple$[p(j)]$ *in $T_{p(j)}$ makes no contribution to the final join result, if no tuple in $T_j$ matches* tuple$[p(j)]$*. Then,* 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.*

*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 disabled entries during searching over $B$-tree index.*

**Observation 2.** *For any non-root table $T_j$ and its parent table $T_{p(j)}$,* tuple$[p(j)]$ *in $T_{p(j)}$ makes no contribution to the final join result, if each tuple in $T_j$ that matches* tuple$[p(j)]$ *has been* disabled*. Then,* 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 5) in Theorem 3 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 3.** *For any $\ell$ ($\ell \geq 2$) input tables $T_1, \cdots, T_\ell$ and the real join result $R_{\text{real}}$, let $\text{Num}_{\text{tr}}$ be the number of tuple retrievals over each input table. It is a function of $|T_1|, \cdots, |T_\ell|$ and $|R_{\text{real}}|$.*

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

## 8 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 [31]. 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 $B$-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 1-3. Details are given in the full version of our paper [76].

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.

## 9 Security Analysis

We provide an (informal) security theorem for our method, as with Opaque [12] and ObliDB [31]. Our security is guaranteed by the existence of simulator $\mathsf{SIM}$: 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 $\mathsf{SIM}$. $\mathsf{SIM}$ 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 $\Delta$). Hence, the adversary cannot learn any additional information in oblivious join processing, since simulator $\mathsf{SIM}$ only sees the above information. Note that $\mathsf{SIM}$ 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 4 with the same notations in Definition 1.

**Theorem 4.** *For any relational database $D$, schema $\mathsf{Sch}(D)$, join query $Q$, oblivious join algorithm $\mathsf{OJoin}$, and security parameter $\lambda$, there is a polynomial-time simulator $\mathsf{SIM}$ such that for any PPT adversary $\mathcal{A}$,*

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

Theorem 4 guarantees our security in the sense of Definition 1. For binary joins, our security guarantee is the same as Krastnikov *et al.* [30] and oblivious mode in Opaque [12] and ObliDB [31]. For multiway joins, our security guarantee is the same as Arasu and Kaushik [13].

Last, we may also introduce a padding mode, as in Opaque [12] and ObliDB [31]. 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 [61], or padding the result size to the closest power of a constant $x$ (*e.g.*, 2 or 4) [78], [79], [80], 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 $\mathsf{SIM}$' for padded mode behaves analogously to $\mathsf{SIM}$. In padded mode, the security theorem replaces the final join output size with an upper bound size as a public parameter in simulator $\mathsf{SIM}$, which indicates the padded output size.

## 10 Experimental Results

### 10.1 Experimental Setup and Datasets

We make the evaluation for ObliDB [31], Krastnikov *et al.* [30] (denoted as ODBJ) and our ORAM based method. For ObliDB, we set Hash Select as the oblivious filter algorithm (see Table 2 in [31]). For ODBJ, we extend its implementation [81] to support general band joins over disk-based relational tables. For our ORAM based method, we have two settings: 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 6 and 7). By default, we evaluate all algorithms in non-padded mode.

We also compare our method with an insecure baseline (Raw Index(+Cache)). It builds $B$-tree indices over data blocks and stores 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 in ORAM based method).

**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. All methods are implemented in C++.

**Default parameter values.** We set encrypted block size $B$ = 4 KB, as in [11], [40], [46]). We set the additional trusted memory size $M = 2B$ ($B$ is the block size) in ODBJ and ORAM based 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 evaluate the methods on the following 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. We refer to [82] and explore general many-to-many join queries as follows. Appendix 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.

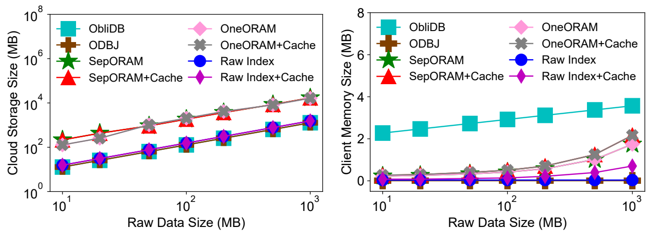
**Social graph.** Social graph [82], [83] contains three twitter user tables "popular-user", "inactive-user" and "normal-user". Each record is a friendship link with a source ID and a destination ID. We randomly sample 5,000 to 200,000 users (with raw data size from 1.3 MB to 58 MB) in our experiments. The default user number is 20,000 (with raw data size 4.5 MB). Appendix 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. It is confirmed by our experimental results (see Figures 8-17). For simplicity, we mainly focus on experimental results for query cost.

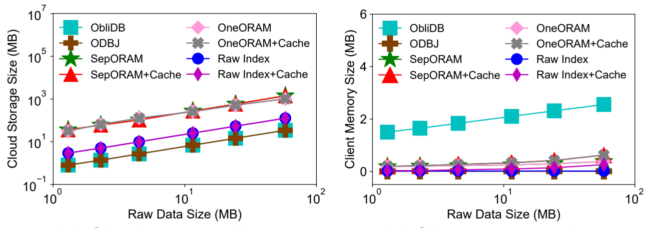
### 10.2 Cloud and Client Storage Costs

Figures 6a and 7a show cloud storage cost on two datasets. ObliDB and ODBJ achieve the minimum cloud storage



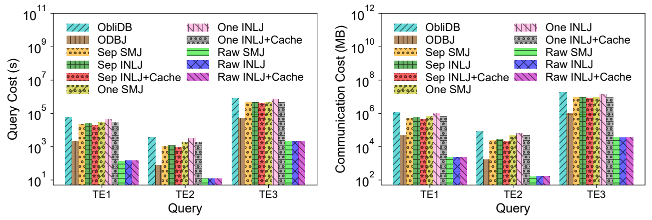
(a) Cloud storage size. (b) Client memory size.

Fig. 6. Storage cost against raw data size on TPC-H.



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

Fig. 7. Storage cost against raw data size on social graph.



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

Fig. 8. Performance of binary equi-join on TPC-H.

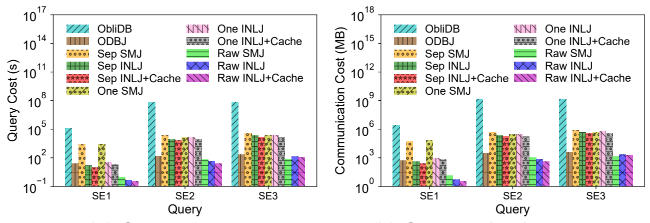
cost, since they only store encrypted data blocks. Raw Index(+Cache) needs a little more cost for storing index blocks. ORAM based method has roughly 10X larger cost than Raw Index(+Cache), due to building ORAM data structure. When we scale up both datasets, our largest cloud storage cost comes to 16.6 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 6b and 7b 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. For ORAM based method, the 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, this 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 adopt oblivious index.

### 10.3 Performance of Binary Equi-Join

#### 10.3.1 Default Setting

Figures 8a and 9a 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*



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

Fig. 9. Performance of binary equi-join on social graph.

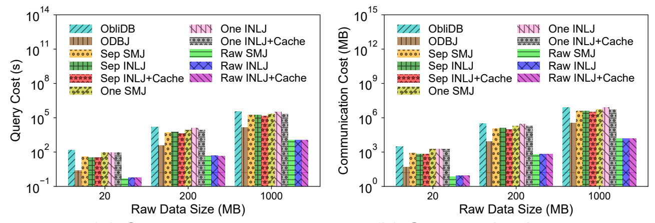
*with input size* on social graph. Our method takes advantage of this, since our query cost depends on input and output sizes *linearly*.

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 performance improvement if using some novel ORAM schemes [35], [36], [37]. 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 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 ORAM based method, Sep INLJ achieves 1.2X-2.6X better performance than One INLJ. As explained in Section 8, 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 8. Last, the index caching brings 1.2X-1.6X speedup ratio in both settings.

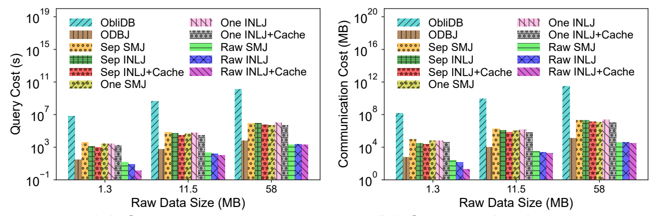
#### *10.3.2 Scalability*

Figures 10a and 11a 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 10a. For Query SE2, we demonstrate the experimental results with 5,000, 50,000, 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 11a. Our SepORAM(+Cache) achieves 2X-4X and $1.6 \times 10^3$X-



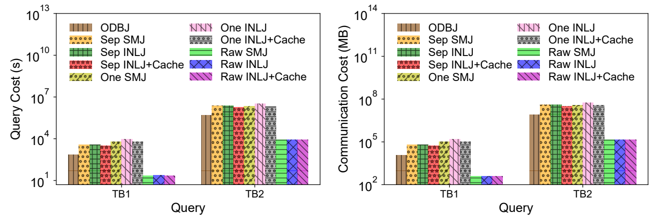
(a) Query cost. (b) Communication cost.

Fig. 10. Performance of Query TE2 against raw data size.



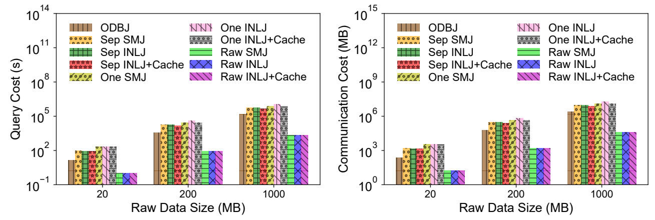
(a) Query cost. (b) Communication cost.

Fig. 11. Performance of Query SE2 against raw data size.



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

Fig. 12. Performance of band join on TPC-H.



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

Fig. 13. Performance of Query TB1 against raw data size.

$1.6 \times 10^4$X 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 10.3.1. 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 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 10.3.1. 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 10.3.1. As in Section 10.3.1, 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.

### 10.4 Performance of Band Join

Figure 12a shows query cost for band join on TPC-H in default setting. In comparison with Raw INLJ(+Cache), our

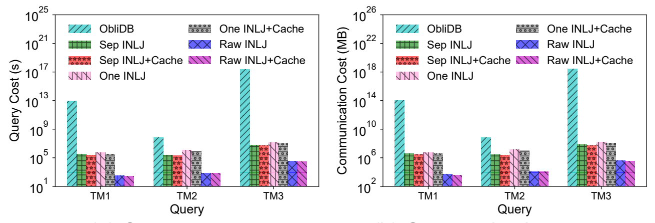

(a) Query cost. (b) Communication cost.

Fig. 14. Performance of multiway equi-join on TPC-H.

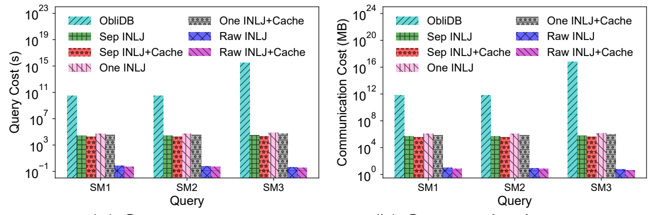

(a) Query cost. (b) Communication cost.

Fig. 15. Performance of multiway equi-join on social graph.

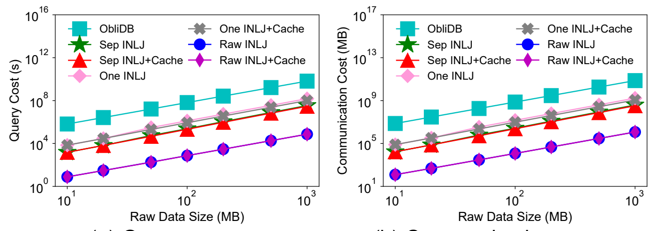

(a) Query cost. (b) Communication cost.

Fig. 16. Performance of Query TM2 against raw data size.

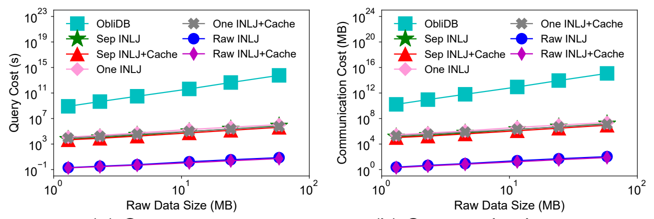

(a) Query cost. (b) Communication cost.

Fig. 17. Performance of Query SM2 against raw data size.

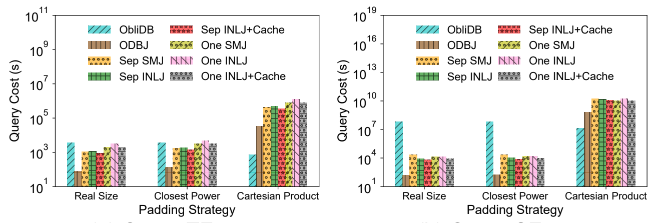

(a) Query TE2. (b) Query SE2.

Fig. 18. Padded vs. non-padded mode (binary equi-join).

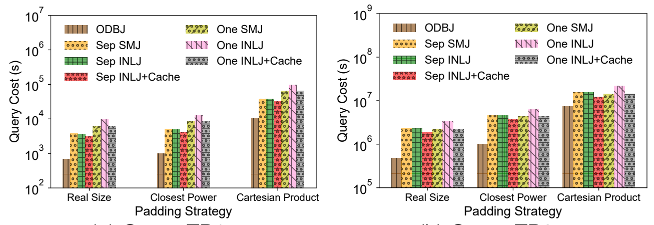

(a) Query TB1. (b) Query TB2.

Fig. 19. Padded vs. non-padded mode (band join).

extended ODBJ algorithm brings 30X and 58X blowup of query cost on Query TB1 and TB2, and Sep INLJ(+Cache) brings 164X and 288X blowup of query cost on Query TB1 and TB2. For ORAM based method, Sep INLJ achieves 1.4X-2.5X better performance than One INLJ, as explained in Section 10.3. The index caching brings 1.2X-1.5X better performance in both settings. Figure 13a shows query cost on Query TB1 against raw data size. When raw data size increases from 20 MB to 1 GB, our extended ODBJ algorithm and Sep INLJ(+Cache) bring 15X-66X and 73X-264X blowup of query cost on Query TB1 compared with Raw INLJ(+Cache). For ORAM based 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.

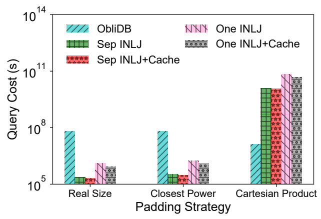
## 10.5 Performance of Multiway Equi-Join

### 10.5.1 Default Setting

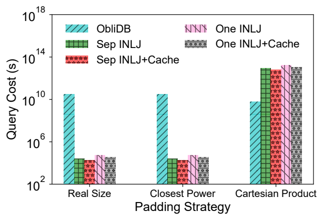
Figures 14a and 15a show query cost for multiway equi-join on two datasets in default setting. Our Sep INLJ(+Cache) achieves $10^6$X-$10^{11}$X better performance than ObliDB on all queries except Query TM2. The reason is that our query cost is roughly *linear* with input and output sizes, but ObliDB 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.

### 10.5.2 Scalability

Figures 16a and 17a 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 $10^5$X-$10^8$X, due to far less join result size. Compared with Raw INLJ(+Cache), Sep INLJ(+Cache) brings 194X-469X and 28000X-91000X blowup 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 10.5.1. 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 10.5.1. Last, the index caching brings 1.1X-2.0X speedup ratio in both settings.



(a) Query TM2.



(b) Query SM2.

Fig. 20. Padded vs. non-padded mode (multiway equi-join).

### 10.6 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 [78], [79], [80]; (3) padding to the Cartesian product (denoted as Cartesian Product). Details of the padding techniques are provided in the second last paragraph in Section 9.

Figures 18-20 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 ORAM based 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 ORAM based 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. ORAM based 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 6b and 7b). 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.

## 11 Conclusion

This paper focuses on oblivious joins over a cloud database. Our method supports general band joins and multiway equi-joins obliviously. For oblivious join without ORAMs, we extend the binary equi-join algorithm in Krastnikov *et al.* [30] to support general band joins. For oblivious join with ORAMs, we integrate $B$-tree indices into ORAMs and propose two types of oblivious algorithms including sort-merge join and index nested-loop join, supporting general band joins and multiway equi-joins. Extensive experimental evaluation has demonstrated the superior efficiency and scalability. Our current design does not address challenges associated with ad-hoc updates, which is a future direction to explore.

## Appendix

### 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 the difference of account balances within $[-100.00, 1000.00]$.

```
SELECT s1.s_suppkey, s2.s_suppkey,
       s1.s_acctbal, s2.s_acctbal
FROM supplier s1, supplier s2
WHERE s1.s_acctbal − 100.00 ≤ s2.s_acctbal
  AND s2.s_acctbal ≤ s1.s_acctbal + 1000.00;
```

Query TB2: Parts joined with other parts with the difference of retail prices within $[-50.00, 40.00]$.

```
SELECT p1.p_partkey, p2.p_partkey,
       p1.p_retailprice, p2.p_retailprice
FROM part p1, part p2
WHERE p1.p_retailprice − 50.00 ≤ p2.p_retailprice
  AND p2.p_retailprice ≤ p1.p_retailprice + 40.00;
```

**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 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;
```

## References

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

[2] A. Arasu, S. Blanas, K. Eguro, M. Joglekar, R. Kaushik, D. Kossmann, R. Ramamurthy, P. Upadhyaya, and R. Venkatesan, "Secure database-as-a-service with Cipherbase," in *SIGMOD*, 2013, pp. 1033–1036.

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

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

[5] Z. He, W. K. Wong, B. Kao, D. W. Cheung, R. Li, S. Yiu, and E. Lo, "SDB: A secure query processing system with data interoperability," *PVLDB*, vol. 8, no. 12, pp. 1876–1879, 2015.

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

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

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

[9] B. Yao, F. Li, and X. Xiao, "Secure nearest neighbor revisited," in *ICDE*, 2013, pp. 733–744.

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

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

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

[13] A. Arasu and R. Kaushik, "Oblivious query processing," in *ICDT*, 2014, pp. 26–37.

[14] T. Hoang, C. D. Ozkaptan, G. Hackebeil, and A. A. Yavuz, "Efficient oblivious data structures for database services on the cloud," *IEEE Trans. Cloud Computing*, 2018.

[15] M. Naveed, S. Kamara, and C. V. Wright, "Inference attacks on property-preserving encrypted databases," in *CCS*, 2015, pp. 644–655.

[16] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart, "Leakage-abuse attacks against searchable encryption," in *CCS*, 2015, pp. 668–679.

[17] G. Kellaris, G. Kollios, K. Nissim, and A. O'Neill, "Generic attacks on secure outsourced databases," in *CCS*, 2016, pp. 1329–1340.

[18] V. Bindschaedler, P. Grubbs, D. Cash, T. Ristenpart, and V. Shmatikov, "The tao of inference in privacy-protected databases," *PVLDB*, vol. 11, no. 11, pp. 1715–1728, 2018.

[19] B. Pinkas and T. Reinman, "Oblivious RAM revisited," in *CRYPTO*, 2010, pp. 502–519.

[20] M. S. Islam, M. Kuzu, and M. Kantarcioglu, "Access pattern disclosure on searchable encryption: Ramification, attack and mitigation," in *NDSS*, 2012.

[21] O. Goldreich, "Towards a theory of software protection and simulation by oblivious RAMs," in *STOC*, 1987, pp. 182–194.

[22] R. Ostrovsky, "Efficient computation on oblivious RAMs," in *STOC*, 1990, pp. 514–523.

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

[24] M. Keller and P. Scholl, "Efficient, oblivious data structures for MPC," in *ASIACRYPT, Part II*, 2014, pp. 506–525.

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

[26] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa, "Oblix: An efficient oblivious search index," in *S&P*, 2018, pp. 279–296.

[27] T. Hoang, M. O. Ozmen, Y. Jang, and A. A. Yavuz, "Hardware-supported ORAM in effect: Practical oblivious search and update on very large dataset," *PoPETs*, vol. 2019, no. 1, pp. 172–191, 2019.

[28] E. Shi, "Path oblivious heap: Optimal and practical oblivious priority queue," in *S&P*, 2020, pp. 842–858.

[29] Y. Li and M. Chen, "Privacy preserving joins," in *ICDE*, 2008, pp. 1352–1354.

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

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

[32] M. T. Goodrich, "Data-oblivious external-memory algorithms for the compaction, selection, and sorting of outsourced data," in *SPAA*, 2011, pp. 379–388.

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

[34] D. J. DeWitt, J. F. Naughton, and D. A. Schneider, "An evaluation of non-equijoin algorithms," in *VLDB*, 1991, pp. 443–452.

[35] L. Ren, C. W. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. van Dijk, and S. Devadas, "Constants count: Practical improvements to oblivious RAM," in *USENIX Security*, 2015, pp. 415–430.

[36] H. Chen, I. Chillotti, and L. Ren, "Onion ring ORAM: Efficient constant bandwidth oblivious RAM from (leveled) TFHE," in *CCS*, 2019, pp. 345–360.

[37] G. Asharov, I. Komargodski, W. Lin, K. Nayak, E. Peserico, and E. Shi, "OptORAMa: Optimal oblivious RAM," in *EUROCRYPT, Part II*, 2020, pp. 403–432.

[38] P. Williams, R. Sion, and A. Tomescu, "PrivateFS: A parallel oblivious file system," in *CCS*, 2012, pp. 977–988.

[39] J. R. Lorch, B. Parno, J. W. Mickens, M. Raykova, and J. Schiffman, "Shroud: Ensuring private access to large-scale data in the data center," in *FAST*, 2013, pp. 199–214.

[40] E. Stefanov and E. Shi, "ObliviStore: High performance oblivious cloud storage," in *S&P*, 2013, pp. 253–267.

[41] V. Bindschaedler, M. Naveed, X. Pan, X. Wang, and Y. Huang, "Practicing oblivious access on cloud storage: the gap, the fallacy, and the new way forward," in *CCS*, 2015, pp. 837–849.

[42] C. Sahin, V. Zakhary, A. E. Abbadi, H. Lin, and S. Tessaro, "TaoStore: Overcoming asynchronicity in oblivious data storage," in *S&P*, 2016, pp. 198–217.

[43] A. Chakraborti and R. Sion, "ConcurORAM: High-throughput stateless parallel multi-client ORAM," in *NDSS*, 2019.

[44] S. Maiyya, S. Ibrahim, C. Scarberry, D. Agrawal, A. E. Abbadi, H. Lin, S. Tessaro, and V. Zakhary, "QuORAM: A quorum-replicated fault tolerant ORAM datastore," in *USENIX Security*, 2022, pp. 3665–3682.

[45] S. Chu, D. Zhuo, E. Shi, and T. H. Chan, "Differentially oblivious database joins: Overcoming the worst-case curse of fully oblivious algorithms," in *ITC 2021*, 2021, pp. 19:1–19:24.

[46] D. Xie, G. Li, B. Yao, X. Wei, X. Xiao, Y. Gao, and M. Guo, "Practical private shortest path computation based on oblivious storage," in *ICDE*, 2016, pp. 361–372.

[47] S. Sasy, S. Gorbunov, and C. W. Fletcher, "ZeroTrace: Oblivious memory primitives from intel SGX," in *NDSS*, 2018.

[48] N. Crooks, M. Burke, E. Cecchetti, S. Harel, R. Agarwal, and L. Alvisi, "Obladi: Oblivious serializable transactions in the cloud," in *OSDI*, 2018, pp. 727–743.

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

[50] E. Dauterman, V. Fang, I. Demertzis, N. Crooks, and R. A. Popa, "Snoopy: Surpassing the scalability bottleneck of oblivious storage," in *SOSP*, 2021, pp. 655–671.

[51] V. Costan and S. Devadas, "Intel SGX explained," *IACR Cryptology ePrint Archive*, vol. 2016, p. 86, 2016.

[52] T. Kim, Z. Lin, and C. Tsai, "CCS'17 Tutorial Abstract: SGX security and privacy," in *CCS*, 2017, pp. 2613–2614.

[53] X. S. Wang, Y. Huang, T.-H. H. Chan, A. Shelat, and E. Shi, "SCORAM: Oblivious RAM for secure computation," in *CCS*, 2014, pp. 191–202.

[54] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi, "ObliVM: A programming framework for secure computation," in *S&P*, 2015, pp. 359–376.

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

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

[57] Y. Wang and K. Yi, "Secure yannakakis: Join-aggregate queries over private data," in *SIGMOD*, 2021.

[58] C. Sahin, T. Allard, R. Akbarinia, A. E. Abbadi, and E. Pacitti, "A differentially private index for range query processing in clouds," in *ICDE*, 2018, pp. 857–868.

[59] Q. Ye, H. Hu, X. Meng, and H. Zheng, "PrivKV: Key-value data collection with local differential privacy," in *S&P*, 2019, pp. 317–331.

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

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

[62] I. Kotsogiannis, Y. Tao, X. He, M. Fanaeepour, A. Machanavajjhala, M. Hay, and G. Miklau, "PrivateSQL: A differentially private SQL query engine," *PVLDB*, vol. 12, no. 11, pp. 1371–1384, 2019.

[63] N. Wang, X. Xiao, Y. Yang, J. Zhao, S. C. Hui, H. Shin, J. Shin, and G. Yu, "Collecting and analyzing multidimensional data with local differential privacy," in *ICDE*, 2019, pp. 638–649.

[64] T. Wang, B. Ding, J. Zhou, C. Hong, Z. Huang, N. Li, and S. Jha, "Answering multi-dimensional analytical queries under local differential privacy," in *SIGMOD*, 2019, pp. 159–176.

[65] J. Yang, T. Wang, N. Li, X. Cheng, and S. Su, "Answering multi-dimensional range queries under local differential privacy," *PVLDB*, vol. 14, no. 3, pp. 378–390, 2020.

[66] C. W. Fletcher, L. Ren, X. Yu, M. van Dijk, O. Khan, and S. Devadas, "Suppressing the oblivious RAM timing channel while making information leakage and program efficiency trade-offs," in *HPCA*, 2014, pp. 213–224.

[67] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang, "Detecting privileged side-channel attacks in shielded execution with déjà vu," in *AsiaCCS*, 2017, pp. 7–18.

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

[69] K. E. Batcher, "Sorting networks and their applications," in *AFIPS Spring Joint Computing Conference*, 1968, pp. 307–314.

[70] M. Ajtai, J. Komlós, and E. Szemerédi, "An $O(n \log n)$ sorting network," in *STOC*, 1983, pp. 1–9.

[71] M. T. Goodrich, "Randomized Shellsort: A simple oblivious sorting algorithm," in *SODA*, 2010, pp. 1262–1277.

[72] ——, "Zig-zag sort: A simple deterministic data-oblivious sorting algorithm running in $O(n \log n)$ time," in *STOC*, 2014, pp. 684–693.

[73] Z. Chang, D. Xie, S. Wang, and F. Li, "Towards practical oblivious join," in *SIGMOD*, 2022, pp. 803–817.

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

[75] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song, "PHANTOM: Practical oblivious computation in a secure processor," in *CCS*, 2013, pp. 311–324.

[76] Z. Chang, D. Xie, S. Wang, F. Li, and Y. Shen, "Towards practical oblivious join processing," 2022. [Online]. Available: https://zhao-chang.github.io/paper/ojoin.pdf

[77] C. T. Yu and M. Z. Ozsoyoglu, "An algorithm for tree-query membership of a distributed query," in *COMPSAC*, 1979, pp. 306–312.

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

[79] G. Asharov, T. H. Chan, K. Nayak, R. Pass, L. Ren, and E. Shi, "Locality-preserving oblivious RAM," in *EUROCRYPT, Part II*, 2019, pp. 214–243.

[80] I. Demertzis, D. Papadopoulos, C. Papamanthou, and S. Shintre, "SEAL: attack mitigation for encrypted databases via adjustable leakage," in *USENIX Security*, 2020.

[81] "Oblivious database join algorithm," 2020. [Online]. Available: https://git.uwaterloo.ca/skrastni/obliv-join-impl

[82] Z. Zhao, R. Christensen, F. Li, X. Hu, and K. Yi, "Random sampling over joins revisited," in *SIGMOD*, 2018, pp. 1525–1539.

[83] M. Cha, H. Haddadi, F. Benevenuto, and P. K. Gummadi, "Measuring user influence in twitter: The million follower fallacy," in *ICWSM*, 2010.

**Zhao Chang** received a BS degree in computer science and technology from Peking University in 2013, and a PhD degree in computing from University of Utah in 2021. He currently works as an associate professor in the School of Computer Science and Technology at Xidian University. His research interests focus on security and privacy issues in large-scale data management.

**Dong Xie** received his Ph.D. degree in Computer Science from University of Utah in 2020. He currently works as an assistant professor at the CSE department of Penn State University. His research interests span the stack of data systems, including distributed databases, main-memory databases, stream processing systems, spatio-temporal data processing, approximate query processing, data privacy, and system security.

**Sheng Wang** holds a Ph.D. in Computer Science from National University of Singapore. He is a Research Scientist and director at Alibaba DAMO Academy. He has published more than 40 papers in top conferences and journals in database area, such as SIGMOD, VLDB and ICDE. His research interests are mainly in data management and data security, including cloud-native databases, encrypted databases, privacy-preserving computation, data storage, and blockchains.

**Feifei Li** received the BS degree in computer engineering from Nanyang Technological University, in 2002, and the PhD degree in computer science from Boston University, in 2007. He is currently a Vice President of Alibaba Group, ACM Distinguished Scientist, President of the Database Products Business Unit of Alibaba Cloud Intelligence, and Director of the Database and Storage Lab of DAMO academy.

**Yulong Shen** received the B.S. and M.S. degrees in computer science and the Ph.D. degree in cryptography from Xidian University, Xi'an, China, in 2002, 2005, and 2008, respectively. He is currently a Professor at the School of Computer Science and Technology, Xidian University. He is also the Associate Director of the Shaanxi Key Laboratory of Network and System Security, Xidian University. His research interests include wireless network security and cloud computing security.

## SUPPLEMENTAL MATERIAL

### A. Security Proof of Theorem 4

*Proof.* (Informal Sketch) In this proof, we show the existence of simulator SIM, and argue that access pattern of SIM is distributed indistinguishable from Trace(OJoin$(D, Q)$) (generated from algorithm OJoin$(D, Q)$). SIM reads algorithm OJoin$(D, Q)$ to determine which operations to simulate.

**For Oblivious Join without ORAMs:**

The security proof is similar to that of Krastnikov *et al.* [30] and Arasu and Kaushik [13]. First, the process of our join algorithm OJoin$(D, Q)$ guarantees that each intermediate table size only pertains to the input and output sizes IOSize$(D, Q)$. Then, we consider how SIM simulates the access patterns for the operations in OJoin$(D, Q)$ as follows.

- Oblivious Sorting or Linear Scan: Since SIM has the access to schema Sch$(D)$, sizing information Size$(D)$ and input and output sizes IOSize$(D, Q)$, SIM can simulate the access pattern, which is indistinguishable from that of the original oblivious sorting or linear scan operations.
- Table Augmentation: When the derived attribute can be computed from existing attributes (*e.g.*, $\tilde{T}_i \leftarrow T_i(id, j, d, pos \leftarrow \tilde{T}_R.pos, \alpha \leftarrow \tilde{T}_S.pos - \tilde{T}_R.pos)$, our join algorithm scans the input tuples, adds the derived attribute, and writes out the output tuple. Given the access to schema Sch$(D)$, sizing information Size$(D)$ and input and output sizes IOSize$(D, Q)$, SIM can simulate the corresponding access patterns.
- Union of Tables: To unify some input tables (*e.g.*, $T_U \leftarrow T_R \cup T_S \cup T_{3-i}$), our join algorithm scans the tuples in each input table one by one and writes out the output tuples. Given the access to schema Sch$(D)$, sizing information Size$(D)$ and input and output sizes IOSize$(D, Q)$, SIM can simulate the corresponding access patterns.
- Filling Positions: To perform Fill-Pos$(T_U)$ operation, our join algorithm scans the input tuples while maintaining a counter in the private client. The counter will be incremented when the algorithm meets any tuple with $tid = T_{3-i}$. For each input tuple, we append a new attribute $pos$ equal to the current value of the counter. Given the access to schema Sch$(D)$, sizing information Size$(D)$ and input and output sizes IOSize$(D, Q)$, SIM can simulate the corresponding access patterns.
- Table Expansion: Our join algorithm makes copies of tuples in each table obliviously based on the join degree information. We directly perform the oblivious expansion algorithm in [30] (*i.e.*, Algorithm 4 in [30]), and the security analysis is the same as that in [30].
- Table Alignment: Our join algorithm generates the join output table by simply concatenating the join key and remaining attributes (*i.e.*, $(j, d)$ attributes) in expanded input tables. Specifically, our join algorithm scans the input tuples, concatenates the attributes, and writes out the output tuple. Given the access to schema Sch$(D)$, sizing information Size$(D)$ and input and output sizes IOSize$(D, Q)$, SIM can simulate the corresponding access patterns.

**For Oblivious Join with ORAMs:**

Specifically, SIM needs to simulate access patterns for ORAM operations and oblivious filter operations (including oblivious compaction or oblivious sorting, and a few linear scans) in OJoin$(D, Q)$.

This proof is covered by Arguments 1-4. We mainly focus on separate ORAMs setting (denoted as SepORAM) in Arguments 1-3. For one ORAM setting (denoted as OneORAM), the proof relies on Argument 4: OneORAM does not introduce any more privacy leakage than SepORAM.

**Argument 1.** *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 Sch$(D)$ and sizing information Size$(D)$, the access pattern simulation for each of such operations is indistinguishable from 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.

**Argument 2.** *We ensure the number of join steps only pertains to the input and output sizes.*

In SepORAM, Theorems 1-3 guarantee that the number of tuple retrievals from each input table (*i.e.*, number of join steps) in algorithm OJoin$(D, Q)$ only pertains to the input and output sizes IOSize$(D, Q)$. Since SIM has the access to IOSize$(D, Q)$, SIM will know the number of join steps based on IOSize$(D, Q)$, and perform the corresponding number of join step simulations.

**Argument 3.** *Arguments 1 and 2 ensure the simulated access pattern is indistinguishable from* Trace(OJoin$(D, Q)$) *in the whole process (i.e., the obliviousness in SepORAM).*

**Argument 4.** *OneORAM does not introduce any more privacy leakage than SepORAM.*

For each step in OneORAM, algorithm 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 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 $\mathsf{SIM}$ still has the access to the background knowledge. □

## B. Proof of Theorem 1

*Proof.* We divide the process of Algorithm 3 into two parts and compute the number of tuple retrievals in each part.

**Part I:** The process except for Line 10-14 in Algorithm 3.

In Part I, at the beginning, we invoke $\text{getFirst}()$ once for $T_1$ 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 $\text{res} > c_2$, the cursor on $T_1$ advances (Line 17 and Line 21); otherwise, the comparison result is $\text{res} < -c_1$, and the cursor on $T_2$ advances (Line 23). In total, the number of invoking $\text{getNext}()$ for $T_1$ 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 10-14 in Algorithm 3.

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 $\text{Num}_{\text{tr}} = |T_1| + |T_2| + |R_{\text{real}}| + 1$. □

## C. Proof of Theorem 2

In Algorithm 4, 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 $T_{\text{out}}$ (Line 7); otherwise, we write out a dummy record to $T_{\text{out}}$ (Line 11). By observing the process of Algorithm 4, the outer loop performs exactly $|T_1|$ iterations, and each iteration leads to exactly one dummy output record (Line 11). 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, $\text{Num}_{\text{tr}} = |T_1| + |R_{\text{real}}|$.

## D. Details of Oblivious Multiway Equi-Join Algorithm

Algorithm 5 shows the details of joining $\ell$ tables $T_1, \cdots, T_\ell$. The main function begins with initializing an empty output table $T_{\text{out}}$ (Line 1). The outer for loop is to iterate over each tuple in table $T_1$ (Line 3). Each time we retrieve a new tuple from $T_1$ (Line 4), we call the subfuntion JOIN() to search the matched tuples from $T_2, \cdots, T_\ell$ (Line 5). After the join processing, we pad tuple retrievals from each input table and dummy output records to an upper bound to ensure the obliviousness (Line 6). Then, we obliviously filter out dummy records from $T_{\text{out}}$ and only keep real join records (Line 7). The last step is to go over all index blocks and reset boolean tags in each entry (Line 8).

The subfunction JOIN($j$) is a recursive function. The input parameter $j$, varying from 2 to $\ell$, indicates that we are currently searching over table $T_j$ to match an array of specific tuples $\text{tuple}[1], \cdots, \text{tuple}[j-1]$. The returned value is always a pair of $\{\text{res}, \text{index}\}$. If $\text{tuple}[p(j)]$ should be disabled, the process will return $\text{res} = \text{false}$ and backtrack to JOIN($\text{index} = p(j)$) (Line 23-24). Otherwise, the process returns $\text{res} = \text{true}$ and backtracks to JOIN($\text{index} = j - 1$) (Line 25-26).

JOIN($j$) begins with initializing $\text{res} = \text{false}$ (Line 2) and retrieving the first tuple $\text{tuple}[j]$ from $T_j$ whose join key is larger than or equal to $\text{tuple}[p(j)]$'s (Line 3). If $\text{tuple}[p(j)]$ and $\text{tuple}[j]$ do not match, we can conclude that no available tuple in $T_j$ can match $\text{tuple}[p(j)]$ and directly output a dummy record (Line 4-5). Otherwise, we leverage a while loop to perform the join processing (Line 7-22). If $j = \ell$ (*i.e.*, we have reached the last table $T_\ell$), we set $\text{res} = \text{true}$ and write out the real join record (Line 8-10). Otherwise, we

**Algorithm 5:** Oblivious Index Nested-Loop Multiway Equi-Join

```
Require: Input: $\ell$ tables $T_1, \cdots, T_\ell$.
    Output: join result table $T_{\text{out}} = T_1 \bowtie \cdots \bowtie T_\ell$.
 1: Initialize $T_{\text{out}} := \emptyset$.
 2: Initialize tuple$[1 \dots \ell] := \emptyset$.
 3: for $i := 1$ to $|T_1|$ do
 4:   tuple[1] := $T_1$.getNext();
 5:   JOIN(2);
 6: end for
 7: Pad tuple retrievals and dummy output records to an
    upper bound.
 8: Obliviously filter out dummy records from $T_{\text{out}}$.
 9: Go over all index blocks and reset boolean tags in each
    entry.
10: return $T_{\text{out}}$;

 1: function JOIN($j$)
 2: res := false;
 3: tuple[$j$] := RETRIEVEFIRSTTUPLE($j$, tuple[$p(j)$].key);
 4: if match(tuple[$p(j)$], tuple[$j$]) = false then
 5:   OUTPUTDUMMYRECORD($j$);
 6: else
 7:   while true do
 8:     if $j = \ell$ then
 9:       res := true;
10:       OUTPUTREALJOINRECORD();
11:     else
12:       {flag, index} := JOIN($j + 1$);
13:       if $j \neq$ index then
14:         return {flag, index};
15:       end if
16:       if flag = false then
17:         DISABLECURRENTTUPLE($j$);
18:       else
19:         res := true;
20:       end if
21:     end if
22:     if HASNEXTMATCHEDTUPLE($j$) = true then
23:       tuple[$j$] := RETRIEVENEXTTUPLE($j$);
24:     else
25:       break;
26:     end if
27:   end while
28: end if
29: if res = false then
30:   return {false, $p(j)$};
31: else
32:   return {true, $j - 1$};
33: end if
34: end function
```

search over the succeeding tables by calling JOIN($j+1$) and gain the returned value $\{\text{flag}, \text{index}\}$ (Line 12). If $j \neq$ index, the process should backtrack to JOIN(index) (Line 13-14). If the returned value flag $=$ false, we can safely disable the current tuple tuple$[j]$ (Line 15-16). Otherwise, we set res $=$ true, which indicates that tuple$[p(j)]$ should still be enabled (Line 17-18). Then, if there exists the next matched tuple in $T_j$, we retrieve it from $T_j$ (Line 19-20); otherwise, the search over $T_j$ can be safely terminated (Line 21-22). The final step is to return the pair of $\{\text{res}, \text{index}\}$ (Line 23-26). The implementation details of the subfunctions are given in Supplemental Material E. The correctness proof of Algorithm 5 is provided in Supplemental Material F.

## E. Details of Subfunctions in Algorithm 5

```
function OutputDummyRecord(index)
for $i$ := index + 1 to $\ell$ do
    for $j$ := 1 to $T_i$.btreeHeight do
        $T_i$.getIndexBlock($\perp$);
    end for$T_i$.getDataBlock($\perp$);
end for
$T_{\text{out}}$.put($\perp$);
end function
```

```
function OutputRealJoinRecord()
$T_{\text{out}}$.put(tuple[1] $\bowtie \cdots \bowtie$ tuple[$\ell$]);
end function
```

```
function RetrieveFirstTuple(index, key)
height := $T_{\text{index}}$.btreeHeight;
// blocks along the $B$-tree path
block[1 ... height] := $\emptyset$;
// block IDs along the $B$-tree path
blockID[1 ... height] := $\emptyset$;
// position of current entry in index block
entryPos[1 ... height] := $\emptyset$;
// if current entry is the last enabled one in index block
isLastEnabled[1 ... height] := $\emptyset$;
// the preceding leaf block
preLeaf := $\perp$;
// the preceding leaf block ID
preLeafID := $\perp$;

curTuple := $\perp$;
curID := $T_{\text{index}}$.btreeRootID;
for $h$ := 1 to height do
    blockID[$h$] := curID;
    block[$h$] := $T_{\text{index}}$.getIndexBlock(curID);
    find := false;
    for $i$ := 1 to block[$h$].entryNum do
        $e$ := block[$h$].entry[$i$];
        if $e$.key $\geq$ key and $e$.enabled = true then
            find := true;
            entryPos[$h$] := $i$;
            isLastEnabled[$h$] := true;
            for $j$ := $i$ + 1 to block[$h$].entryNum do
                if block[$h$].entry[$j$].enabled = true then
                    isLastEnabled[$h$] := false;
                    break;
                end if
            end for
            curID := $e$.blockID;
            if $h$ = height then
                $b$ := $T_{\text{index}}$.getDataBlock(curID);
                curTuple := getDataTuple($b$, $e$);
            end if
            break;
        end if
    end for
    if find = false then
        for $j$ := $h$ + 1 to height do
            $T_{\text{index}}$.getIndexBlock($\perp$);
        end for
        $T_{\text{index}}$.getDataBlock($\perp$);
        break;
    end if
end for
return curTuple;
end function
```

```
function DisableCurrentTuple(index)
height := $T_{\text{index}}$.btreeHeight;
// the following variables have been defined in RetrieveFirstTuple()
// block[1 ... height];
// blockID[1 ... height];
// entryPos[1 ... height];
// preLeaf;
// preLeafID;

for $i$ := 1 to index − 1 do
    if $i > 1$ then
        for $j$ := 1 to $T_i$.btreeHeight do
            $T_i$.getIndexBlock($\perp$);
        end for
    end if
    $T_i$.getDataBlock($\perp$);
end for

// indicate if preLeaf has been updated
updated := false;
curPos := entryPos[height];
block[height].entry[curPos].enabled := false;
$e$ := block[height].entry[curPos];
if $e$.next = false then
    // update the "next" flag in last enabled leaf entry
    done := false;
    for $i$ := curPos − 1 to 1 do
        $e'$ := block[height].entry[$i$];
        if $e'$.key = $e$.key and $e'$.enabled = true then
            block[height].entry[$i$].next := false;
            done := true;
            break;
        else if $e'$.key $\neq$ $e$.key then
            done := true;
            break;
        end if
    end for
    if preLeaf $\neq \perp$ and done = false then
        for $i$ := preLeaf.entryNum to 1 do
            $e'$ := preLeaf.entry[$i$];
            if $e'$.key = $e$.key and $e'$.enabled = true then
            preLeaf.entry[$i$].next := false;
            updated := true;
            break;
          else if $e'$.key $\neq$ $e$.key then
            break;
          end if
        end for
      end if
    end if

    // check if any entry in the block is still enabled
    for $h$ := height to 2 do
      find := false;
      for $i$ := 1 to block[$h$].entryNum do
        if block[h].entry[$i$].enabled = true then
          find := true;
          break;
        end if
      end for
      if find = true then
        break;
      else
        curPos := entryPos[$h - 1$];
        block[$h - 1$].entry[curPos].enabled := false;
      end if
    end for

    for $h$ := 1 to height do
      $T_{\text{index}}$.putIndexBlock(blockID[$h$], block[$h$]);
    end for
    $T_{\text{index}}$.getDataBlock($\perp$);
    OutputDummyRecord(index);

    if updated = true then
      UpdatePreLeaf(index);
    end if
    end function
```

```
    function UpdatePreLeaf(index)
    height := $T_{\text{index}}$.btreeHeight;
    // the following variables have been defined in RetrieveFirstTuple()
    // preLeaf;
    // preLeafID;

    for $i$ := 1 to index $- 1$ do
      if $i > 1$ then
        for $j$ := 1 to $T_i$.btreeHeight do
          $T_i$.getIndexBlock($\perp$);
        end for
      end if
      $T_i$.getDataBlock($\perp$);
    end for

    for $h$ := 1 to height $- 1$ do
      $T_{\text{index}}$.getIndexBlock($\perp$);
    end for
    $T_{\text{index}}$.putIndexBlock(preLeafID, preLeaf);
    $T_{\text{index}}$.getDataBlock($\perp$);
    OutputDummyRecord(index);
    end function
```

```
    function HasNextMatchedTuple(index)
    height := $T_{\text{index}}$.btreeHeight;
    // the following variable has been defined in RetrieveFirstTuple()
    // block[1 . . . height];
    // entryPos[1 . . . height];

    curPos := entryPos[height];
    $e$ := block[height].entry[curPos];
    if $e$.next = true then
      return true;
    else
      return false;
    end if
    end function
```

```
    function RetrieveNextTuple(index)
    height := $T_{\text{index}}$.btreeHeight;
    // the following variables have been defined in RetrieveFirstTuple()
    // block[1 . . . height];
    // blockID[1 . . . height];
    // entryPos[1 . . . height];
    // isLastEnabled[1 . . . height];
    // preLeaf;
    // preLeafID;

    for $i$ := 1 to index $- 1$ do
      if $i > 1$ then
        for $j$ := 1 to $T_i$.btreeHeight do
          $T_i$.getIndexBlock($\perp$);
        end for
      end if
      $T_i$.getDataBlock($\perp$);
    end for

    curTuple := $\perp$;
    curID := $T_{\text{index}}$.btreeRootID;
    for $h$ := height to 1 do
      if isLastEnabled[$h$] = false then
        nextH := $h$;
        break;
      end if
    end for
    for $h$ := 1 to height do
      if $h$ = height and curID $\neq$ blockID[$h$] then
        key := tuple[index].key;
        find := false;
        for $i$ := block[$h$].entryNum to 1 do
          $e$ := block[$h$].entry[$i$];
          if $e$.key = key and $e$.enabled = true then
            find := true;
            break;
          else if $e$.key $\neq$ key then
            break;
          end if
        end for
        if find = true then
          preLeaf := block[$h$];
          preLeafID := blockID[$h$];
        end if
    end if
    blockID[$h$] := curID;
    block[$h$] := $T_{\text{index}}$.getIndexBlock(curID);
    if $h <$ nextH then
      begin := entryPos[$h$];
    else if $h =$ nextH then
      begin := entryPos[$h$] + 1;
    else
      begin := 1;
    end if
    for $i$ := begin to block[$h$].entryNum do
      $e$ := block[$h$].entry[$i$];
      if $e$.enabled = true then
        if $h <$ nextH then
          assert($i$ = entryPos[$h$]);
        else
          entryPos[$h$] := $i$;
          isLastEnabled[$h$] := true;
          for $j$ := $i + 1$ to block[$h$].entryNum do
            if block[$h$].entry[$j$].enabled = true then
              isLastEnabled[$h$] := false;
              break;
            end if
          end for
        end if
        curID := $e$.blockID;
        if $h$ = height then
          $b$ := $T_{\text{index}}$.getDataBlock(curID);
          curTuple := getDataTuple($b, e$);
        end if
        break;
      end if
    end for
  end for
  return curTuple;
end function
```

### F. Correctness Proof of Algorithm 5

**Observation 4.** *Given the structure of the join tree, any tuple in any input table will be disabled in Algorithm 5, if and only if either*

*1) no matched tuple can be found from any of its children node tables; or*

*2) all matched tuples from any of its children node tables have been disabled.*

**Lemma 1.** *No tuple from any leaf node table in the join tree will be disabled.*

*Proof.* Based on Observation 4, a tuple will be disabled, only if the current table has a child node table in the join tree. Obviously, we have no opportunity to disable any tuple from any leaf node table in the join tree. □

**Lemma 2.** *Any disabled tuple cannot make any contribution to the final join result, $i.e.$ tuple disabling does not incur any false negatives.*

*Proof.* We define the level number of each node in the join tree as follows. We let each leaf node in the join tree be in level 0. Then we recursively define the level numbers of non-leaf nodes. For each non-leaf node, if the minimum level number of its children nodes is $k$, then this node is in level $k + 1$.

(Proof by induction) We perform an induction over the level number $k$.

1°. The base case is $k = 0$. Lemma 1 shows that no tuple from any table in level 0 will be disabled. Therefore, the conclusion in $k = 0$ is correct.

2°. (Induction hypothesis) Given $k$ ($k \geq 0$), suppose that the conclusion is correct for each table whose level number is no larger than $k$. Then we consider the case of $k + 1$.

(Proof by contradiction) We assume the conclusion is wrong for the case of $k+1$, $i.e.$, there exists a disabled tuple $\text{tuple}[j]$ in table $T_j$ with level number $k + 1$, which can lead to a real join record. Since $T_j$ is in level $k+1$, there must exist a child node table $T_i$ in level $k$. Since we assume that $\text{tuple}[j]$ leads to a real join record, there must exist a tuple $\text{tuple}[i]$ in $T_i$ that matches $\text{tuple}[j]$ and leads to the same join record. According to the induction hypothesis, $\text{tuple}[i]$ will not be disabled. Then, according to Observation 4, $\text{tuple}[j]$ will also not be disabled. But this is contradictory to our assumption. The contradiction shows that our previous assumption does not hold. Hence, the case of $k + 1$ is proven.

3°. Based on 1°and 2°, the conclusion is proven. □

**Theorem 5.** *Algorithm 5 generates an accurate join result for each query, $i.e.$, no false positives and no false negatives.*

*Proof.* Algorithm 5 is still based on the traditional index nested-loop join algorithm that generates an accurate join result for each query. The major differences between our algorithm and the traditional one are as follows:

1) we perform an oblivious algorithm, $i.e.$, adding dummy tuple retrievals and writing out dummy records if necessary;

2) we disable the tuples that cannot make any contribution to the final join result;

3) we will not retrieve the next tuple that has a different join key from the current tuple.

First, 1) does not introduce any false positives, since we will obliviously filter out dummy records at the end of Algorithm 5. Second, Lemma 2 guarantees that 2) will not lead to any false negatives. Last, 3) is based on the property of equi-joins. If the current tuple $\text{tuple}[j]$ in $T_j$ matches the given tuple $\text{tuple}[p(j)]$ in $T_{p(j)}$ but the succeeding tuple in $T_j$ has a different join key from $\text{tuple}[j]$'s, we can conclude that the succeeding tuple cannot match $\text{tuple}[p(j)]$ and we do not need to retrieve it from $T_j$. Therefore, Algorithm 5 still generates an accurate join result for each query. □

### G. Proof of Theorem 3

**Observation 5.** *We access the $\ell$ input tables in a round-robin way. After each tuple retrieval from each input table, we will output exactly one (real or dummy) join record.*

**Lemma 3.** *For any tuple in any input table, if it is enabled after being processed at least once, it will be enabled all the time during processing the current query.*

*Proof.* We follow the definition of the level number in the join tree in the proof of Lemma 2.

(Proof by induction) We perform an induction over the level number $k$.

1°. The base case is $k = 0$. Lemma 1 shows that no tuple from any table in level 0 will be disabled. Therefore, the conclusion in $k = 0$ is correct.

2°. (Induction hypothesis) Given $k$ ($k \geq 0$), suppose that the conclusion for each table whose level number is no larger than $k$ is correct. Then we consider the case of $k + 1$.

(Proof by contradiction) We assume the conclusion is wrong for the case of $k + 1$, *i.e.*, there exists a tuple $\text{tuple}[j]$ in table $T_j$ with level number $k + 1$, which is enabled after being processed at least once, but is finally disabled while processing the current query. According to Observation 4, since $T_j$ is in level $k + 1$ and $\text{tuple}[j]$ is enabled after processed at least once by the algorithm, there must exist a tuple $\text{tuple}[i]$ in a child node table $T_i$ in level $k$, where $\text{tuple}[i]$ can match $\text{tuple}[j]$ and is enabled at that time. Hence, $\text{tuple}[i]$ is also a tuple that is enabled after being processed at least once by the algorithm. According to the induction hypothesis, $\text{tuple}[i]$ will be enabled all the time during the process. However, according to Observation 4, since $\text{tuple}[j]$ is finally disabled in the algorithm, all the matched tuples of $\text{tuple}[j]$ from any children node tables of $T_j$, including $\text{tuple}[i]$ in Table $T_i$, will finally be disabled in the algorithm. This is contradictory to the deduction that $\text{tuple}[i]$ will be enabled all the time. The contradiction shows that our previous assumption does not hold. Hence, the case of $k + 1$ is proven.

3°. Based on 1°and 2°, the conclusion is proven. □

**Lemma 4.** *The total number of calling subfunctions* DISABLECURRENTTUPLE() *and* UPDATEPRELEAF() *is bounded by* $\sum_{j=2}^{\ell} |T_j|$.

*Proof.* The proof is based on a series of sub-conclusions.

1°. The number of leaf entries that can be processed by DISABLECURRENTTUPLE() or UPDATEPRELEAF() is bounded by $\sum_{j=2}^{\ell} |T_j|$.

Note that according to the process of Algorithm 5, both DISABLECURRENTTUPLE() and UPDATEPRELEAF() will not process any leaf entry for the root node table $T_1$.

2°. Any leaf entry for any input table will be processed by DISABLECURRENTTUPLE() at most once.

Note that after disabling any tuple, the leaf entry that points to the disabled tuple will be marked as disabled. The subfunctions RETRIEVEFIRSTTUPLE() and RETRIEVENEXTTUPLE() (see Supplemental Material E) ensure that we will skip all the disabled leaf entries during searching over available tuples. Hence, we guarantee that each leaf entry can be disabled at most once.

3°. Any leaf entry for any input table will be processed by UPDATEPRELEAF() at most once.

Note that Algorithm 5 calls UPDATEPRELEAF(), if and only if the "next" tag of the last enabled leaf entry in block "preLeaf" has changed from "true" to "false" (Line 36-39 in DISABLECURRENTTUPLE()). This change only occurs at most once for that leaf entry. Hence, we guarantee that UPDATEPRELEAF() can process each leaf entry at most once.

4°. No leaf entry will be processed by both DISABLECURRENTTUPLE() and UPDATEPRELEAF().

1) Any leaf entry that has been disabled will not be processed by UPDATEPRELEAF().

Line 36-39 in DISABLECURRENTTUPLE() ensures that only enabled leaf entry can be updated and will be processed by UPDATEPRELEAF().

2) Any leaf entry that has been processed by UPDATEPRELEAF() will not be disabled.

This point is guaranteed by Lemma 3.

5°. Based on 1°, 2°, 3°, and 4°, the conclusion is proven. □

**Lemma 5.** *The total number of dummy output records is bounded by* $|T_1| + 2\sum_{j=2}^{\ell} |T_j|$.

*Proof.* According to the process of Algorithm 5, one dummy record will be output, if and only if one of the following cases occurs:

1) the first retrieved tuple $\text{tuple}[j]$ from $T_j$ does not match $\text{tuple}[p(j)]$ in its parent node $T_{p(j)}$ (Line 3-5 in JOIN($j$)); or

2) the current tuple $\text{tuple}[j]$ in $T_j$ is being disabled (Line 16 in JOIN($j$)); or

3) the last enabled leaf entry in block "preLeaf" has been updated (Line 36-39 in DISABLECURRENTTUPLE()), and is being processed by UPDATEPRELEAF() (Line 62 in DISABLECURRENTTUPLE()).

According to Lemma 4, the total occurrence number of Case 2) and Case 3) is bounded by $\sum_{j=2}^{\ell} |T_j|$.

In Case 1), the algorithm will return false and backtrack to JOIN($p(j)$) (Line 23-24 in JOIN($j$)). After that, we have either

A) if table $T_j$ is a child node table of $T_1$, the join processing over the current $\text{tuple}[1]$ will be terminated, and the next tuple from $T_1$ will be retrieved; or

B) otherwise, the current $\text{tuple}[p(j)]$ in the parent node $T_{p(j)}$ will be disabled.

Obviously, the occurrence number of Case A) is bounded by $|T_1|$. According to Lemma 4, the occurrence number of Case B) is no larger than $\sum_{j=2}^{\ell} |T_j|$.

Therefore, the total occurrence number of Case 1), Case 2) and Case 3) is bounded by $|T_1| + 2\sum_{j=2}^{\ell} |T_j|$. □

**Proof of Theorem 3.**

*Proof.* Based on Observation 5, the number of tuple retrievals from each input table is equal to the total number of real and dummy output records. According to Lemma 5, the total number of dummy output records is bounded by $|T_1| + 2\sum_{j=2}^{\ell} |T_j|$. Since the number of real join records is $|R_{\text{real}}|$, the total number of output records will be $|T_1| + 2\sum_{j=2}^{\ell} |T_j| + |R_{\text{real}}|$. Therefore, $\text{Num}_{\text{tr}} = |T_1| + 2\sum_{j=2}^{\ell} |T_j| + |R_{\text{real}}|$. □

## H. Discussion on Oblivious Cyclic Join

For cyclic join query, the join graph is not a tree structure. Hence, there may exist one child node table that has multiple join attributes and multiple parent node tables. For each tuple in this child node table, we must keep multiple "next" flags, one flag for each join attribute. When some of these flags are true and the others are false, our join algorithm will not know what to do in the next step. If we simply delete some edges from the join graph and make the join graph into some tree structure, there will be an unbounded number of false positives generated in the join processing.

As discussed in Arasu and Kaushik [13], we have reason to believe that there may not exist efficient oblivious cyclic join algorithms. The existence of such algorithms would imply more efficient algorithms for variants of 3SUM problem. For the detailed analysis on the hardness of oblivious cyclic joins, please refer to Section 6 in Arasu and Kaushik [13].

## I. Complexity Analyses

**For Oblivious Join without ORAMs:**

**Sort-Merge Join.** According to the analysis on Algorithm 1 and Algorithm 2, the total time complexity is $O((|T_1|+|T_2|)\log^2(|T_1|+|T_2|)+|R_{\text{real}}|\log^2|R_{\text{real}}|)$. Algorithm 1 takes $O((|T_1|+|T_2|)\log^2(|T_1|+|T_2|))$ time cost, since the oblivious sorting takes $O((|T_1|+|T_2|)\log^2(|T_1|+|T_2|))$ time cost. Algorithm 2 consists of two parts: table expansion and table alignment. For oblivious table expansion, the time complexity is $O((|T_1|+|T_2|)\log^2(|T_1|+|T_2|)+|R_{\text{real}}|\log|R_{\text{real}}|)$ as with [30]. For oblivious table alignment, the time complexity is $O(|R_{\text{real}}|\log^2|R_{\text{real}}|)$, since the oblivious sorting takes $O(|R_{\text{real}}|\log^2|R_{\text{real}}|)$ time cost. Hence, the total time complexity is $O((|T_1|+|T_2|)\log^2(|T_1|+|T_2|)+|R_{\text{real}}|\log^2|R_{\text{real}}|)$.

**For Oblivious Join with ORAMs:**

**Sort-Merge Join.** According to the analysis on Algorithm 3, the total number of tuple retrievals is $O(|T_1|+|T_2|+|R_{\text{real}}|)$. In our implementation, each tuple is retrieved through the $B$-tree leaf level pointers, which leads to two Path-ORAM accesses (one for the leaf block, the other for the data block) over each input table. Thus, this time cost is $O((|T_1|+|T_2|+|R_{\text{real}}|)\cdot(\log|T_1|+\log|T_2|))$. We denote the trusted storage size as $M$. Then, the final oblivious filter needs $O((|T_1|+|T_2|+|R_{\text{real}}|)\log_M(|T_1|+|T_2|+|R_{\text{real}}|))$ cost. Therefore, the total time complexity is $O((|T_1|+|T_2|+|R_{\text{real}}|)\cdot(\log|T_1|+\log|T_2|+\log_M(|T_1|+|T_2|+|R_{\text{real}}|)))$. When $|R_{\text{real}}|\ll|T_1|\cdot|T_2|$, this cost is far less than performing a Cartesian product. Note that index caching (see Section 4.2) cannot optimize the performance of our sort-merge join, unless we bluntly cache the whole $B$-tree leaf level index, since each tuple is directly retrieved through the leaf level pointers.

**Index Nested-Loop Join.** According to the analysis on Algorithm 4, the total number of tuple retrievals is $O(|T_1|+|R_{\text{real}}|)$. In our implementation, each tuple retrieval from $T_1$ is performed through Path-ORAM protocol, which leads to one Path-ORAM access (over the data block). However, each tuple needed from $T_2$ is retrieved by searching over a whole $B$-tree path, which leads to $O(\log_B|T_2|)$ Path-ORAM accesses, when the $B$-tree fanout is $\Theta(B)$. Thus, this time cost is $O((|T_1|+|R_{\text{real}}|)\cdot(\log|T_1|+\log_B|T_2|\cdot\log|T_2|))$. We denote the trusted storage size as $M$. Then, the final oblivious filter needs $O((|T_1|+|R_{\text{real}}|)\log_M(|T_1|+|R_{\text{real}}|))$ cost. Therefore, the total time complexity is $O((|T_1|+|R_{\text{real}}|)\cdot(\log|T_1|+\log_B|T_2|\cdot\log|T_2|+\log_M(|T_1|+|R_{\text{real}}|)))$. When introducing the index caching, we denote the number of outsourced levels in each $B$-tree index over each input table as $\Delta$, and each tuple retrieval from $T_2$ only leads to $(\Delta+1)$ Path-ORAM accesses ($\Delta$ for index blocks, one for the data block). Hence, the total time complexity will be $O((|T_1|+|R_{\text{real}}|)\cdot(\log|T_1|+\Delta\log|T_2|+\log_M(|T_1|+|R_{\text{real}}|)))$. When $|R_{\text{real}}|\ll|T_1|\cdot|T_2|$, this cost is still far less than performing a Cartesian product. Especially, the time complexity increases with $|T_2|$ logarithmically rather than linearly. Hence, Algorithm 4 achieves good performance when $|T_1|\ll|T_2|$.

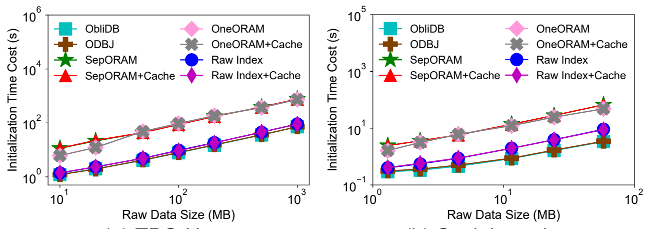
The complexity analysis on Algorithm 5 is similar to that on Algorithm 4. In Algorithm 5, the total number of tuple retrievals from each input table is $O(\sum_{j=1}^{\ell}|T_j|+|R_{\text{real}}|)$. In our implementation, each tuple retrieval from $T_1$ is performed through Path-ORAM protocol, which leads to one Path-ORAM access (over the data block). However, each tuple retrieval from any other table $T_j$ ($j\neq 1$) will search over a whole $B$-tree path, which leads to $O(\log_B|T_j|)$ Path-ORAM accesses, when the $B$-tree fanout is $\Theta(B)$. Thus, this time cost is $O((\sum_{j=1}^{\ell}|T_j|+|R_{\text{real}}|)\cdot(\log|T_1|+\sum_{j=2}^{\ell}\log_B|T_j|\cdot\log|T_j|))$. We denote the trusted storage size as $M$. Then, the final oblivious filter needs $O((\sum_{j=1}^{\ell}|T_j|+|R_{\text{real}}|)\log_M(\sum_{j=1}^{\ell}|T_j|+|R_{\text{real}}|))$ cost. Lastly, we reset boolean tags of each entry in all index blocks, which needs $O(\sum_{j=1}^{\ell}|T_j|/B\cdot\log|T_j|)$ cost. Therefore, the total time complexity is $O((\sum_{j=1}^{\ell}|T_j|+|R_{\text{real}}|)\cdot(\log|T_1|+\sum_{j=2}^{\ell}\log_B|T_j|\cdot\log|T_j|+\log_M(\sum_{j=1}^{\ell}|T_j|+|R_{\text{real}}|)))$. When introducing the index caching, we denote the number of outsourced levels in each $B$-tree index over each input table as $\Delta$, and each tuple retrieval from $T_2,\cdots,T_\ell$ only leads to $(\Delta+1)$ Path-ORAM accesses ($\Delta$ for index blocks, one for the data block). Hence, the total time complexity will be $O((\sum_{j=1}^{\ell}|T_j|+|R_{\text{real}}|)\cdot(\log|T_1|+\sum_{j=2}^{\ell}\Delta\log|T_j|+\log_M(\sum_{j=1}^{\ell}|T_j|+|R_{\text{real}}|)))$. When $|R_{\text{real}}|\ll\prod_{j=1}^{\ell}|T_j|$, this cost will be far less than performing a Cartesian product.

## J. Optimization in One ORAM Setting

**Sort-merge join.** Recall that in Algorithm 3, whenever we perform a $\text{getNext}()$ over one input table ($T_1$ or $T_2$), we also perform a dummy operation $\text{get}(\perp)$ over the other table ($T_2$ or $T_1$) to ensure the obliviousness. But in one ORAM setting, we do not need to perform those dummy operations. Since each tuple retrieval from any input table retrieves an index block and then a data block through ORAM protocol, the adversary cannot distinguish which input table we are currently accessing.

**Index nested-loop join.** For binary join (*e.g.*, Algorithm 4) in separate ORAMs setting, we keep the invariant that we retrieve the tuple needed from each input table alternatively, and add dummy tuple retrievals from $T_1$ as necessary. In one ORAM setting, we can remove those dummy tuple retrievals from $T_1$. However, recall that we retrieve each tuple from $T_1$ using one ORAM access, while we retrieve each tuple from $T_2$ by searching over an outsourced $B$-tree path. To prevent the adversary from distinguishing whether the current tuple retrieval comes from $T_1$ or $T_2$, we must pad the number of ORAM accesses for each tuple retrieval from $T_1$ to the length of the outsourced $B$-tree path in $T_2$.

The optimization in multiway join becomes complicated. First, to prevent the adversary from distinguishing which input table we are currently accessing, we must pad the number of ORAM accesses for each tuple retrieval from any input table to the maximum length of all the retrieved $B$-tree paths. Then, we must find the worst-case upper bound of the total number of tuple retrievals, which only pertains to the sizing information of input tables and real



(a) TPC-H. (b) Social graph.

Fig. 21. Overall initialization time cost.

join result. In most steps, we still need to retrieve tuples from each input table in a round-robin way and add dummy retrievals as necessary. But when we disable any tuple from any input table, we do not need to add dummy retrievals from the other input tables, since we can safely bound the total number of disabled tuples.

## K. Overall Initialization Time Cost

Initializing the original Path-ORAM [33] is very expensive, since each real block insertion pays a Path-ORAM write operation with $O(\log N)$ cost. To avoid the high initialization cost, we pre-build the ORAM data structure in trusted storage and then upload it to the cloud using bulk loading.

In our bulk loading based initialization, the communication overhead and I/O cost of the whole data structure dominate the overall initialization cost, which is roughly proportional to cloud storage cost. Figure 21 shows the overall initialization time cost of different methods. ObliDB has the minimum cost, since it simply stores the encrypted data blocks to the cloud. Raw Index(+Cache) needs a little more cost, since it also builds indices over the data blocks. Our method has the largest cost (still roughly 10X larger than Raw Index(+Cache)), due to building the Path-ORAM data structure. When the raw data size increases from 10 MB to 1 GB on TPC-H, our initialization cost increases from 9 seconds to 605 seconds. When the raw data size increases from 1.3 MB to 58 MB on TPC-H, our initialization cost increases from 2 seconds to 53 seconds.