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 background knowledge, the server may learn 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 5 and 6.

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 the algorithm to join multiple tables obliviously. A series of oblivious binary joins will disclose the intermediate table sizes, which may leak some sensitive information, e.g., data distribution or 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. Table 1 shows the comparison of oblivious join algorithms.

Example 1. Figure 1 shows that Opaque Join [12] and 0-OM Join [31] do not work for many-to-many join, due to leaking some sensitive information (e.g., join degree).

Given two input tables $T_1$ and $T_2$, they first put tuples from both input tables into one single table $T$, and obliviously sort $T$ according to the join key. Next, they perform a linear scan over the single sorted table $T$, and join each tuple originally from $T_1$ with the corresponding tuples originally from $T_2$. In the original setting, they need to ensure the invariant that after accessing every input tuple in $T$, they write out exactly one real or dummy join record. But for many-to-many join, they cannot keep the invariant above. For example, after accessing tuple...
T_2(2, 1), which can match two tuples \( T_1(2, 1) \) and \( T_2(2, 2) \), they must write out two join records \( T_1(2, 1) \bowtie T_2(2, 1) \) and \( T_2(2, 2) \bowtie T_2(2, 1) \) before the next access over \( T \), i.e., the number of output records between two accesses over \( T \) leaks the join degree. Processing tuples \( T_2(2, 2) \) and \( T_2(2, 3) \) brings the same security issue.

In summary, prior studies are still unable to address the major challenge in oblivious join. They are only efficient for foreign key join [12], [31], or restricted to binary join [30], or not leading to practical implementation [13], [29].

Our major objective is to support general band joins and multiway equi-joins obliviously. Band join [33] is a binary join between tables \( T_1 \) and \( T_2 \) on numeric attributes \( T_1.A \) and \( T_2.B \) with the join condition \( T_1.A - c_1 \leq T_2.B \leq T_1.A + c_2 \), where \( c_1 \) and \( c_2 \) are numeric values satisfying \( c_1 \geq 0 \) and \( c_2 \geq 0 \). 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-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 and bound the total number of block accesses. Note that ORAM can be viewed as a blackbox, providing read and write interface, while hiding access patterns. We can introduce some novel ORAMs (e.g., [34], [35], [36]) 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 4. 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 5.1 and 5.2. 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 index nested-loop join in Section 6. We avoid retrieving tuples that make no contribution to the final join result and bound the total number of block accesses.
- We conduct extensive experiments on real-world datasets in Section 9. The results demonstrate a superior performance gain (orders of magnitude speedup for oblivious multiway equi-joins) over baseline algorithms.

2 BACKGROUND AND RELATED WORK

2.1 Generic ORAMs and Path-ORAM

Generic ORAMs. 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 and hides the access patterns with the same length of 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 to the ORAM data structure. A few advanced ORAMs [37], [38], [39], [40], [41], [42], [43] 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.

Path-ORAM. In this work, we adopt Path-ORAM [44] due to good performance and simplicity. It 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. To store \( N \) blocks of size \( B \), a basic Path-ORAM requires \( O(\log N + N/B) \) client memory and \( O(\log N) \) cost per query.

2.2 Oblivious Sorting and Filtering

Oblivious sorting. Items can be sorted by accessing in a fixed, predefined order. Bitonic sort [45] needs \( O(N \log^2 N) \) time cost but with small constant factor. It can be extended to an oblivious external sort with \( O(N \log^2 (N/M)) \) time cost using client memory size \( M \) [12], [31]. A few algorithms [46], [47], [48] achieve \( O(N \log N) \) time cost but may fail with a small probability [47], or lead to large constant factors [46] and non-trivial implementation [48]. Recently, Shi [28] proposes an oblivious heap sort with \( O(N \log N) \) time cost, which works better in memory but is not IO-efficient.

Oblivious filtering. Dummy records can be removed by oblivious filtering. Prior studies [12], [13], [29] and the conference version [49] adopt an oblivious sorting to filter out dummy records. Actually, it can be done by oblivious compaction. OptORAM [36] achieves this in \( O(N) \) time but needs some non-trivial techniques. In this work, we adopt a simple oblivious compaction algorithm [32] with \( O(N \log N) \) time cost, where \( M \) is trusted memory size.

2.3 Oblivious Data Structure and Index

Prior studies [14], [24], [25], [26], [27], [28] build oblivious tree structures or indices. For certain data structure whose access pattern exhibits some predictability, they make the structure “oblivious” to improve the performance rather than blindly storing blocks from the structure into ORAM. ORAM+B-tree. B-tree indices can be introduced to speed up the oblivious query processing [31], [50]. The client ignores the semantic difference of (encrypted) index and data blocks and stores them into ORAM. When answering any query, the client starts with retrieving the root block (of the index) from the server and then traverse down the tree. Intuitively, the client queries the index by running the same algorithm as that over a standard B-tree. The only difference is that each index or data block is retrieved through ORAM.

Oblivious B-tree. Oblivious B-tree [31], [50] is designed to avoid storing the position map in client. The main idea is that each index node keeps block IDs and position tags of its children nodes. When retrieving any node through ORAM, we have acquired the position tags of 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
TABLE 1
Comparison of oblivious join algorithms.

<table>
<thead>
<tr>
<th>Join Type</th>
<th>Algorithm</th>
<th>Complexity Analysis</th>
<th>Cloud Storage</th>
<th>Client Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Li and Chen [29]</td>
<td>BE A1</td>
<td>$\Omega(\prod_{j=1}^{\ell}</td>
<td>T_j</td>
<td>)$</td>
</tr>
<tr>
<td></td>
<td>BD A2</td>
<td>$O((</td>
<td>T_{in}</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>ME A3</td>
<td>$\prod_{j=1}^{\ell}</td>
<td>T_j</td>
<td>\log(</td>
</tr>
<tr>
<td>Arasu and Kaushik [13]</td>
<td>BE Equi-Join</td>
<td>$O((</td>
<td>T_{in}</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>PF Opaque Join</td>
<td>$O((</td>
<td>T_{in}</td>
<td>+</td>
</tr>
<tr>
<td>Opaque [12]</td>
<td>PF SMJ</td>
<td>$O(</td>
<td>T_{in}</td>
<td>+</td>
</tr>
<tr>
<td>ObliDB [31]</td>
<td>ME Hash Join</td>
<td>$O(\prod_{j=1}^{\ell}</td>
<td>T_j</td>
<td>)$</td>
</tr>
<tr>
<td>ODB [30]</td>
<td>BE Binary Join</td>
<td>$O((</td>
<td>T_{in}</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>BD SMJ</td>
<td>$O((</td>
<td>T_{in}</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>BD INLJ</td>
<td>$O(</td>
<td>T_{in}</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>BD (Cache)</td>
<td>$O(</td>
<td>T_{in}</td>
<td>+</td>
</tr>
<tr>
<td>Ours d</td>
<td>BD (Cache)</td>
<td>$O(</td>
<td>T_{in}</td>
<td>+</td>
</tr>
</tbody>
</table>

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

We denote the total size of all input tables as $|T_{in}| = \sum_{j=1}^{\ell} |T_j|$ and the real join result size as $|T_{out}|$.

We assume an oblivious sorting needs $O(n \log n)$ time cost [28], and an oblivious external 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 filtering needs $O(n \log_{log_{max}} n)$ time cost, as well as Theorem 6 in [32].

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 as $\Delta$.

<table>
<thead>
<tr>
<th>TABLE 2 Notations.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D, D'$</td>
</tr>
<tr>
<td>$Q, Q'$</td>
</tr>
<tr>
<td>$c_1, c_2$</td>
</tr>
<tr>
<td>$N$</td>
</tr>
<tr>
<td>$B$</td>
</tr>
<tr>
<td>$M$</td>
</tr>
<tr>
<td>$T_{in}, \ldots, T_{\ell}$</td>
</tr>
<tr>
<td>$T_{out}(H_{real})$</td>
</tr>
<tr>
<td>$T_{out}(I_{real})$</td>
</tr>
<tr>
<td>$T_{out}(O_{real})$</td>
</tr>
<tr>
<td>$\Delta$</td>
</tr>
<tr>
<td>SMJ</td>
</tr>
<tr>
<td>INLJ (+Cache)</td>
</tr>
<tr>
<td>Size($D$)</td>
</tr>
<tr>
<td>Sch($D$)</td>
</tr>
<tr>
<td>IQSize($Q, D$)</td>
</tr>
<tr>
<td>OJoin($D, Q$)</td>
</tr>
<tr>
<td>Trace($\gamma$)</td>
</tr>
</tbody>
</table>

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 [34], [51], [52]. The client can 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.

Note that the techniques above do not protect how many accesses to the data structure. In our method, we integrate indices into ORAMs and address the security issue in the scenario of oblivious join, as long as the indices support both point and range queries obliviously.

2.4 Oblivious Query Processing

Xie et al. [53] proposes ORAM solutions to shortest path computation. ZeroTrace [52] supports oblivious get/put/insert operations over set/dictionary/list interfaces. Obladi [54] provides ACID transactions while hiding access patterns. OCQ [55] performs oblivious cooperative analytics in a decentralized manner. Snoopy [56] designs an oblivious storage based on oblivious load balancer and subORAMs. Chu et al. [57] focuses on differentially oblivious join whose problem definition is different from our work.

Note that existing solutions [12], [30], [31] rely on Trusted Execution Environments (TEE) (e.g., Intel SGX [58], [59]). However, TEE is orthogonal to oblivious algorithms and has no advantage to the obliviousness.

2.5 Other Related Work

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 [55], [60], [61], [62], [63], [64] 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 [65] and key-value data collection [66], and support general SQL queries [67], [68], [69]. However, DP-based solutions [65], [66], [67], [68], [69], [70], [71], [72] provide differential privacy for query results, while we provide the obliviousness in query processing.

3 Problem Definition and Overview

The formulation includes a client and a cloud server. The client, who has a small and secure memory, stores her data into the large but untrusted cloud storage. In online processing, the client issues join queries against the server.

We follow the definition in Opaque [12] and ObliDB [31]. Let $D$ be the relational database (where some $B$-tree indices may be integrated) in the cloud and $Q$ be a join query. Let $\text{Size}(D)$ be the sizing information of database $D$, which includes numbers and sizes of tables, rows, columns, and attributes in $D$, but does not include any attribute values. Let $\text{Sch}(D)$ be the schema information of database $D$, which
includes table and column names in $D$ (easily hidden using encryption). Let $\text{IOSize}(D, Q)$ be the input/output sizes of running $Q$ over $D$. Note that $\text{IOSize}(D, Q)$ does not include the sizes of all intermediate join tables for any join query $Q$ over multiple tables in $D$, which must be protected against the adversary. Let $\text{Trace}(\cdot)$ be the trace of server location accesses and network traffic patterns in query processing. Table 2 lists the notations used in this paper.

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

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

We support two oblivious join approaches including non-ORAM approach (see Section 4) and ORAM approach (see Sections 5 and 6). In preprocessing stage, the client partitions the data into blocks and encrypts these data blocks. In particular for ORAM approach, the client builds an ORAM data structure (e.g., Path-ORAM) over the encrypted blocks and integrates some $B$-tree indices into the ORAM data structure using ORAM+$B$-tree or oblivious $B$-tree (see Section 2.3). 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 runs the oblivious join algorithms by performing a series of oblivious operations or ORAM operations, which reads/writes blocks from/to the server and generates the query results.

**Segmenting ORAM.** In ORAM approach, we separate one single ORAM into multiple smaller ORAMs (denoted as SepORAM) 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 ORAM for index blocks. The comparison in Table 1 is based on this setting. We also consider one single ORAM setting (denoted as OneORAM) and make the related discussion in Section 7.

**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.

By default, we follow the security guarantee in Definition 1 in both non-ORAM approach and ORAM approach (including both SepORAM and OneORAM settings). We provide the security analysis and proof in Section 8.

We also introduce a padding mode to ease the volume leakage in final output size, 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, we may introduce some novel padding techniques. For example, explore differential privacy rather than full obliviousness to reduce the padding size [68], or pad the result size to the closest power of a constant $x$ (e.g., 2 or 4) [73], [74], [75], leading to at most $\log_x |R_{\text{worst}}|$ distinct result sizes, where $|R_{\text{worst}}|$ is the Cartesian product size in join scenario.

Note that our approaches do not consider privacy leakage through any side-channel attack (like time taken for each operation). Prior orthogonal studies [76], [77], [78] can help to alleviate such leakage.

### 4 Oblivious Band Join without ORAM

We extend the binary equi-join algorithm in Krasnikov et al. [30] to support general band join obliviously. First, we obliviously compute the degree information in the join graph. 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.

#### 4.1 Join Degree Computation

Algorithm 1 shows the details of join degree computation. For each input table $T_j(d, j)$, we denote the join key as $j$ and the remaining attributes as $d$. We mainly focus on table $T_1$, and the computation 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 $id$ to each tuple (Line 1-4). We parameterize oblivious sorting with a lexicographic ordering on chosen attributes, e.g., $\text{OSort}(T_i(j \uparrow, d \uparrow))$ sorts $T_i$ by increasing $j$ attribute, followed by increasing $d$ attributes.

Then, we aim to generate augmented tables $T_{i1}$ and $T_{i2}$ with join degree $\alpha$ and position $\text{pos}$, such that each tuple $t_1 \in T_{i1}$ matches $t_1.\alpha$ tuples in $T_{i2}$, where each matched tuple $t_2$ has a unique $t_2.\text{id} \in (t_1.\text{pos}, t_1.\text{pos} + t_1.\alpha)$ (Line 5-15).

---

Algorithm 1: Join Degree Computation

**Require:** Input: two tables $T_1(j, d)$ and $T_2(j, d)$ with join condition $T_1(j) = T_2(j)$ and position map $\text{pos}$.

**Output:** $T_1(id, j, d, \text{pos}, \alpha)$ and $T_2(id, j, d, \text{pos}, \alpha)$.

1. for $i \leftarrow 1$ to 2 do
2. $T_i \leftarrow \text{OSort}(T_i(j \uparrow, d \uparrow))$
3. $T_i(id, j, d) \leftarrow T_i(id \leftarrow ID, j, d); \quad \triangleright \text{add id column}$
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. $T_S(id, j, d) \leftarrow T_i(id, j + c_i, d)$
8. $T_U(id, j, d, \text{tid}) \leftarrow T_R \cup T_S \cup T_{3 \cdots i}; \quad \triangleright \text{add tid column}$
9. $T_U(id, j, d, \text{tid}, \text{pos}) \leftarrow \text{Fill-Pos}(T_U)$
10. $T_R(id, j, d, \text{tid}) \leftarrow \text{OSort}(T_U(j \uparrow, \text{tid} < T_R < T_S < T_{3 \cdots i} \uparrow));$
11. $T_R(id, j, d, \text{tid}, \text{pos}) \leftarrow \text{Fill-Pos}(T_R)$
12. $T_R(id, j, d, \text{pos}) \leftarrow \sum T_R(id, j, d, \text{pos})$
13. $T_R(id, j, d, \text{pos}) \leftarrow T_R{id, j, d, \text{pos}} - T_U{\text{id, j, d, \text{pos}}}$
14. $T_i \leftarrow T_i(id, j, d, \text{pos} \leftarrow T_R{id, j, d, \text{pos}}, \alpha \leftarrow T_S{id, j, d, \text{pos}} - T_R{id, j, d, \text{pos}})$
15. end for
16. return $T_1$ and $T_2$.
Specifically, we generate two auxiliary tables $T_R$ and $T_S$ for $T_I$, where $T_R . j \leftarrow T_I . j - c_1$ and $T_S . j \leftarrow T_I . j + c_2$ (Line 6-7). Suppose tuple $t_1 \in T_I$ corresponds to tuples $t_{R} \in T_R$ and $t_{S} \in T_S$. According to the join condition, any tuple $t_2 \in T_R$ with $t_{R} . j \in [t_{R} . j, t_{S} . j]$ will match $t_1 \in T_I$. Then, we generate augmented tables $T_{R^3}$ and $T_{S^3}$ with position pos, such that any tuple $t_3 \in T_{R^3}$ with $t_{R^3} . id \in (t_{R} . pos, t_{S} . pos)$ will match $t_1 \in T_I$, with the help of a union $T_U$ of tables $T_R . j$, $T_S . j$ and $T_{R^3} . j$ (Line 8-13). Finally, we generate $T_1$ from $T_R . j$ and $T_S . j$, where the position $t_1 . pos \leftarrow t_{R^3} . pos$ and the join degree $\alpha \leftarrow t_{R^3} . pos - t_{R} . pos$ (Line 14).

Algorithm 1 takes $O((|T_1| + |T_2|) \log(|T_1| + |T_2|))$ time cost, when an oblivious sort needs $O(n \log n)$ time cost [28].

Example 2. An example is given in Figure 2. We mainly focus on table $T_I$. First, we sort $T_R . j$ and $T_S . j$ lexicographically by $(j, d)$ (Line 1-4). Then, we generate two auxiliary tables $T_R . j$ and $T_S . j$, where $T_R . j \leftarrow T_I . j - 2$ and $T_S . j \leftarrow T_I . j + 2$ (Line 6-7). Note that $t_1 \in (3, 4, 3) \in T_1$ corresponds to $t_2 \in (3, 2, 3) \in T_R$ and $(3, 5, 3) \in T_S$. According to the join condition, $(2, 2, 2), (3, 2, 4)$ and $(4, 5, 3) \in T_{R^3}$ with $j \in [2, 5]$ are 3 matches of $t_1 \in T_I$.

Now, we compute a union $T_U$ of tables $T_R . j$, $T_S . j$ and $T_{R^3}$, and sort $T_U$ lexicographically by $(j, tid)$: $T_R . j < T_S . j < T_{R^3} . j$ (Line 8-9). Note that $t_2 \in (3, 2, 4, 2) \in T_{R^3}$ with $j \in [2, 5]$ are 3 matches of $(3, 4, 3) \in T_1$ and they all rank between $(3, 2, 3)$ and $(3, 5, 3)$ in $T_{R^3} . j$. We scan $T_{R^3}$ and assign the current number of tuples with $tid = 2$ to $T_{R^3} . pos$ (Fill-Pos($T_U$) in Line 10).

After that, we extract $T_R . j$ and $T_S . j$ from $T_U$ by re-sorting $T_U$ (Line 11-13). Finally, we generate $T_1$ from $T_R . j$ and $T_S . j$, where $\tilde{T}_1 . pos \leftarrow T_{R^3} . pos$ and $\tilde{T}_1 . \alpha \leftarrow T_{R^3} . pos - T_{R} . pos$ (Line 14).

4.2 Table Expansion and Alignment

Algorithm 2 shows the details of table expansion and alignment. After obtaining the join degree $\alpha$, we need to make copies for each tuple based on $\alpha$ (aka table expansion). We obliviously expand each $T_I$ into table $S_I$ using Algorithm 4 in [30], where $S_I$ consists of $\alpha$ (contiguous) copies of each tuple $(id, j, d, pos) \in T_I$ (Line 2). Then, we obliviously align $S_2$ with $S_1$ (aka table alignment), so that each join record corresponds to a row of $S_1$ and a row of $S_2$ with matching index (Line 3-6). Finally, we generate the join output table $T_{out}$ by concatenating $(j, d)$ attributes in $S_1$ and $S_2$ (Line 7).

Example 3. An example is given in Figure 3. First, we obliviously expand each $T_I$ into table $S_I$ (Line 2). For example, for tuple $t_1 = (3, 4, 3) \in T_1 \in T_I . (j, d, pos)$, we make $t_1 . \alpha = 3$ copies of $t_1$ to match $(2, 2, 2), (3, 2, 4)$ and $(4, 5, 3, 2)$ in $T_2 . (j, d, pos)$ with $T_{2 \alpha} . id \in (t_1 . pos, t_1 . pos + t_1 . \alpha) = [1, 4]$.

Then, we obliviously sort $S_2$ with $S_1$ (Line 3-6).

1) We perform a grouping identity operation by scanning $S_1$ (Line 3). For example, $t_1 \in 3$ copies $(3, 4, 3, 1), (3, 4, 3, 2)$ and $(3, 4, 3, 3)$ belong to the same group, and each gets a different gid = 1, 2 and 3.

2) We update pos attribute as $pos \leftarrow pos + gid$ in table $S_I$ (Line 4). After that, $t_1 . \alpha$’s 3 copies in $S_1$ will be $(3, 4, 3), (3, 4, 3, 2)$ and $(3, 4, 3, 3)$, and $t_1 . \alpha$’s 3 matches in $S_2$ will be $(2, 2, 2), (3, 2, 4)$ and $(4, 5, 3, 2)$. Now, any tuple $t_2 \in S_2$ matches the only tuple $t_1 \in S_1$, where $s_1 . id = s_2 . pos$ and $s_1 . pos = s_2 . id$.

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

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. We assume an oblivious sorting need $O(n \log n)$ time cost [28]. For oblivious table expansion, the time cost is $O((|T_1| + |T_2|) \log(|T_1| + |T_2|) + |R_{real}| \log |R_{real}|)$. For oblivious table alignment, the time cost is $O(|R_{real}| \log |R_{real}|)$. Hence, the total time cost is $O((|T_1| + |T_2|) \log(|T_1| + |T_2|) + |R_{real}| \log |R_{real}|)$.

5 OBLIVIOUS BAND JOIN WITH ORAM

5.1 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 are sorted as per the attribute. For each input table, we build an ORAM structure for data blocks and another smaller one for index blocks.

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.

Algorithm 3 joins two tables $T_1$ and $T_2$. Whenever we perform a getNext() over one input table ($T_1$ or $T_2$), we also
Algorithm 3: Oblivious Sort-Merge Band Join

Require: Input: two tables $T_1(j,d)$ and $T_2(j,d)$ with join condition $T_1.j - c_1 \leq T_2.j \leq T_1.j + c_2$.
Output: join result table $T_{out}$.
1: Initialize $T_{out} \leftarrow \emptyset$.
2: Initialize $t_1, t_2 \leftarrow \emptyset$.
3: for $i \leftarrow 1$ to 2 do
4: $t_i \leftarrow T_i$.
5: end for
6: while $t_1 \neq \emptyset$ or $t_2 \neq \emptyset$ do
7: if $c_1 \leq res \leq c_2$ then
8: begin $t_2$.
9: while $c_1 \leq res \leq c_2$ do
10: $T_{out}$.put(res); $t_2 \leftarrow T_2$.getFirst();
11: end while
12: end if
13: return $T_{out}$;

perform a dummy operation $getDummy(\cdot)$ over the other table ($T_2$ or $T_1$) to ensure the obliviousness.

First, Algorithm 3 initializes $T_{out} \leftarrow \emptyset$ (Line 1). Then, we retrieve the first two tuples 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 $\text{"res"}$ 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 $t_2$. First, we save the current $t_2$ to a temporary tuple $\text{"begin"}$ (Line 9). We keep writing out the join result join(t1, t2) to $T_{out}$, and retrieving the next tuple from $T_2$ as $t_2$, until the newly retrieved $t_2$ does not match $t_1$ (Line 10-14). Once they do not match, we write out a dummy record and assign $\text{"begin"}$ to back to $t_2$ (Line 15-16). Finally, we retrieve the next tuple from $T_1$ (Line 17) and move to the next iteration (Line 6-7).

2) $t_1$ does not match $t_2$. 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, we retrieve the next tuple from $T_2$ (Line 22-23). Finally, we move to the next iteration (Line 6-7).

After both cursors reach the end of tables $T_1$ and $T_2$, the final step is to obliviously filter out dummy records from $T_{out}$ (Line 27).

Example 4. An example is given in Figure 4. First, we retrieve $t_1 \leftarrow T_1(1,2)$ and $t_2 \leftarrow T_2(1,3)$ from $T_1$ and $T_2$ (Line 2-5). Since the join comparison result $res = 0 \in [-c_1, c_2]$ (Line 7), we can conclude $t_1$ matches $t_2$ (Line 8). Then, we assign $t_2 = T_2(1,3)$ to "begin" (Line 9). We keep writing out join(t1, t2) and retrieving the preceding tuples $T_2(2,2)$, $T_2(4,2)$ and $T_2(3,3)$ from $T_2$ as $t_2$, until $t_2 = T_2(5,3)$ does not match $t_1 = T_1(1,2)$ (Line 10-14). Once they do not match, we write out a dummy record and assign "begin" $= T_2(1,3)$ back to $t_2$ (Line 15-16). Finally, we retrieve the next tuple $T_1(3,1)$ from $T_1$ (Line 17) and move to the next iteration (Line 6-7).

Then, consider $t_1 \leftarrow T_1(4,3)$ and $t_2 \leftarrow T_2(1,3)$. Since the join comparison result $res < -c_1$ (Line 7), we can conclude $t_1$ does not match $t_2$ (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 $T_1(1,3)$), we retrieve the next tuple $T_2(2,2)$ from $T_2$ (Line 22-23). Finally, we move to the next iteration (Line 6-7).

In particular, when the cursor on $T_1$ moves to $T_1(4,3)$ and that on $T_2$ reaches the end of $T_2$, we will retrieve a dummy tuple from $T_2$ (Line 12) and let $res = +\infty > c_2$ (Line 13). The rest still goes in the same way as stated above.

After both cursors reach the end of tables $T_1$ and $T_2$, the final step is to obliviously filter out dummy records from $T_{out}$. 

Theorem 1. For any two input tables $T_1$ and $T_2$ and the real join result $R_{real}$, let $Num_{js}$ be the number of join steps from each input table. It is a function of $|T_1|$, $|T_2|$ and $|R_{real}|$. We have

$$Num_{js} = f(|T_1|, |T_2|, |R_{real}|) = |T_1| + |T_2| + |R_{real}| + 1.$$

Proof. We divide the process of Algorithm 3 into two parts and compute the number of join steps in each part.

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

In the first step, we invoke $getFirst()$ once for $T_1$ and $T_2$ (Line 2-4). Note that each join step leads to one join comparison. In Part I, each join comparison leads to writing out one dummy record. If the comparison result is $res > c_2$, the cursor on $T_1$ advances (Lines 17 and 21); otherwise, the comparison result is $res < -c_1$, and the cursor on $T_2$ advances (Line 23). The process above will end when both cursors reach the end of $T_1$ and $T_2$. Hence, we will invoke $getNext()$ $|T_1| + |T_2|$ times. Therefore, the total number of join steps in Part I is $|T_1| + |T_2| + 1$.

Part II: The process in Line 10-14 in Algorithm 3.

Note that each join step leads to one join comparison. In Part II, each join comparison leads to writing out one real join record. Since the number of real join records is $|R_{real}|$, the number of join steps in Part II is also $|R_{real}|$.

Based on Part I and II, $Num_{js} = |T_1| + |T_2| + |R_{real}| + 1$.

5.2 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

1. Due to space limit, proofs of theorems, complexity analyses and implementation details of our algorithms are given in full version [79].
Algorithm 4: Oblivious Index Nested-Loop Band Join

Require: Input: two tables $T_1(j, d)$ and $T_2(j, d)$ with join condition $t_1.j - c_1 \leq t_2.j \leq t_1.j + c_2$. Output: join result table $T_{out}$.

1: Initialize $T_{out} \leftarrow \emptyset$.
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 $t_1.j - c_1 \leq t_2.j \leq t_1.j + c_2$ do
7:      $T_{out}.put(Join(t_1, t_2))$;
8:      $T_1.getDummy();$
9:      $t_2 \leftarrow T_2.getNext();$
10: end while
11: $T_{out}.put(\perp)$;
12: end for
13: $T_{out} \leftarrow OFilter(T_{out})$;
14: return $T_{out}$;

querying the indices through ORAMs. In detail, the outer loop is to scan table $T_1$. When accessing each tuple in $T_1$, the algorithm retrieves matched tuples from table $T_2$ through $B$-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.

Algorithm 4 joins two tables $T_1$ and $T_2$. Algorithm 4 begins with initializing an empty output table $T_{out}$ (Line 1). The outer loop is to iterate over each tuple in table $T_1$ (Line 3). Each time we retrieve a new tuple $t_1$ from $T_1$ (Line 4), we first retrieve a tuple $t_2$ 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, we write the join record $Join(t_1, t_2)$ to the output table $T_{out}$ (Line 7) and retrieve the next tuple 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 $t_1$. Once they do not match, we write out a dummy record (Line 11) and step into the next iteration. The final step is to obliviously filter out dummy records from $T_{out}$ and only keep real join records (Line 13).

Example 5. An example is given in Figure 5. When we retrieve tuple $t_1 \leftarrow T_1(1, 2)$ from $T_1$ (Line 4), we first retrieve tuple $t_2 \leftarrow T_2(1, 3)$ from $T_2$, which is the first tuple satisfying $t_2.j \geq t_1.j - c_1$ (Line 5). When $t_1$ can match $t_2$, we keep writing out the join record $Join(t_1, t_2)$ (Line 7) and retrieving the succeeding tuples $T_2(2, 2)$ and $T_2(2, 4)$ from $T_2$ as the new $t_2$ (Line 9). Once the newly retrieved $t_2 \leftarrow T_2(5, 3)$ does not match $t_1 \leftarrow T_1(1, 2)$, we step into the next iteration and process the next tuple $t_1 \leftarrow T_1(3, 1)$ from $T_1$. In particular, 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 5). The final step is to obliviously filter out dummy records from $T_{out}$ (Line 13).

Theorem 2. For any two input tables $T_1$ and $T_2$ and the real join result $R_{real}$, let $Num_{js}$ be the number of join steps. It is a function of $|T_1|, |T_2|$ and $|R_{real}|$. Specifically, we have

$$Num_{js} = f(|T_1|, |T_2|, |R_{real}|) = |T_1| + |R_{real}|.$$

6 OBLIVIOUS MULTIWAY EQUI-JOIN

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.

Example 6. Figure 6 shows an example of acyclic multiway equi-join over four tables $T_1 \cdot T_4$. Due to the acyclicity, each input table can be arranged as a node in a join tree. In this tree, for any different tables $T_1, T_2, T_3$, if $T_k$ is on the path from $T_i$ to $T_j$, we must have $Attr(T_i) \cap Attr(T_j) \subseteq Attr(T_k)$ for their attribute sets. The algorithm of building a join tree is presented in [80]. 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 $P(i)$.

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., $T_2(1, 1), T_3(1, 4), \ldots$) from $T_2, \ldots, T_4$. To ensure the obliviousness, we retrieve the tuple needed from each input table in a round-robin way and add dummy retrievals as necessary (e.g., retrieve $\perp$ from $T_4$, due to no tuple with join key $D \geq 4$ for matching $T_3(1, 4)$, as highlighted in yellow in Figure 6). In each step, if there is a match (e.g., in 4th and 7th join step), we output the join record; otherwise, we output a dummy record.

To bound the total number of join steps, we make the following observations to avoid retrieving unnecessary tuples.

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 tuple $T_3(3, 4)$ in parent table $T_3$, we find no tuple in $T_3$ matches $T_3(3, 4)$ (in 1st join step). Hence, $T_3(3, 4)$ makes no contribution to the final join result, and can be safely disabled in an additional 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 its leaf entry as disabled using an additional boolean tag. 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 tuple $T_1(1,1)$ in parent table $T_1$, $T_3(1,4)$ is the only tuple in $T_3$ that matches $T_1(1,1)$. However, since $T_1(1,4)$ 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 another 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 $T_{p(j)}$, tuple $p[j]$ in $T_{p(j)}$ will have no more matches, if the current tuple tuple $p[j]$ in $T_j$ matches tuple $p[j]$ but the succeeding tuple in $T_j$ has a different join key from tuple $p[j]$’s.

Observation 3 is based on the property of equi-joins. For example, for table $T_3$ and tuple $T_1(1,1)$ in parent table $T_1$, we find that $T_3(1,4)$ 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 the normal join process, we pad the number of join steps to the upper bound (e.g., the last step in Figure 6) 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.

In brief, tuple disabling operations will introduce some additional dummy join steps, but we can still bound the total number of join steps in Theorem 3. Besides, tuple disabling operations also bring the overhead of resetting the boolean tags after answering each join query. However, the total time complexity is dominated by regular join steps and final oblivious filtering. Hence, the time cost of resetting the boolean tags is relatively small in oblivious join processing.

Theorem 3. For any $\ell (\ell \geq 2)$ input tables $T_1, \ldots, T_\ell$ and the real join result $R_{real}$, let $Num_{js}$ be the number of join steps. It is a function of $|T_1|, \ldots, |T_\ell|$ and $|R_{real}|$. Specifically, we have

$$Num_{js} = f(|T_1|, \ldots, |T_\ell|, |R_{real}|) = |T_1| + 2 \sum_{j=2}^{\ell} |T_j| + |R_{real}|.$$

7 Discussion on One ORAM Setting

In this work, we separate one single ORAM into multiple smaller ORAMs (aka SepORAM setting). Now, we reconsider the optimization in OneORAM setting.

Since we retrieve all the tuples through one single ORAM, an optimization in OneORAM is to safely remove some dummy tuple retrievals to speed up join processing. To ensure the obliviousness, we must write out a real or dummy join record after each tuple retrieval in OneORAM (rather than after each join step in SepORAM). Then, we must pay the same number of ORAM accesses between writing out any two join records. In other words, we must pad the number of ORAM accesses to the maximum height of B-tree indices in OneORAM. Note that each tuple retrieval from any input table will be indistinguishable for the adversary, although he knows the total number of tuple retrievals. We can bound the total number of tuple retrievals in OneORAM, as long as it only pertains to the input and output sizes, and no additional information will be leaked.

However, there is a major drawback in OneORAM setting. Suppose there are multiple tables in the whole dataset, but only a few binary joins will be processed online. In this scenario, we must put all input tables into one single ORAM in advance, since we do not know the online workload. Hence, we have to pay much larger cost for accessing the large single ORAM rather than smaller separate ORAMs.

8 Security Analysis

We formalize our security guarantee in Theorem 4 with the same notations in Definition 1. As with Opaque [12] and ObliDB [31], our security is guaranteed by the existence of simulator SIM such that for any probabilistic polynomial-time (PPT) adversary $A$, $A$ cannot distinguish between the real server location trace from our method and the simulated trace from simulator SIM. Since SIM only sees what we want to leak, $A$ cannot learn any additional information. A brief description on specifics of simulated traces from SIM is given in the proof of Theorem 4.

In our setting, 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 $\Delta$). Note that SIM has no access to the sizes of all intermediate join tables, since we protect this sensitive information against the adversary.

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

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

Proof. (Informal Sketch) In this proof, we show the existence of simulator SIM, and argue that access pattern of SIM is distributed indistinguishable from 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 and Linear Scan**: These two operations access the blocks in a fixed, predefined order. Hence, SIM can simulate the access patterns, given the access to Sch(D), Size(D) and IOSize(D, Q).
- **Table Augmentation**: For each iteration in Table Augmentation, we read an input tuple, compute and add derived attributes, and write out the output tuple. As with linear scan, SIM can simulate the access patterns, given Sch(D), Size(D) and IOSize(D, Q).
• Union of Tables: For each iteration in Union of Tables, we read a tuple from one input table and write it out to the output table. As with linear scan, SIM can simulate the access patterns, given Sch(D), Size(D) and IOSize(D, Q).
• Filling Position: Filling Position (Fill-Pos(·) in Algorithm 1) operation scans the input tuples while maintaining a counter in private client. The counter will be incremented once we meet specific tuples. For each input tuple, we assign the counter to a new attribute pos and write out the updated tuple. Hence, the access pattern simulation can be reduced to Table Augmentation.
• Table Expansion: We adopt Algorithm 4 in [30] to support Table Expansion operation. SIM can simulate the access patterns as in [30].
• Table Alignment: Table Alignment sorts and scans the expanded tables. For each iteration in the scan, we read two input tuples, concatenate their (j, d) attributes and write one join record. Hence, the access pattern simulation can be reduced to oblivious sorting, Linear Scan, and Table Augmentation.

For Oblivious Join with ORAMs:

SIM needs to simulate access patterns for ORAM operations and oblivious filtering operations (including oblivious compaction/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 the same as that in the original ORAM scheme, or that for original oblivious compaction/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 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 IOSize(D, Q). Then, after each tuple retrieval in OneORAM (rather than after each join step in SepORAM), we ensure to write out a real or dummy join record to the output table, to protect the join degree information and ensure the full obliviousness. The simulation is similar to that in SepORAM, since SIM still has the access to the background knowledge.

Theorem 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].

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

9 EXPERIMENTAL RESULTS

9.1 Experimental Setup and Datasets

We make the evaluation for ObliDB [31], ODBJ [30] and our ORAM approach. For ODBJ, we extend its implementation [81] to support general band joins. We adopt two oblivious sorting algorithms including oblivious external bitonic sorting [45] (denoted as ODBJ (Bitonic)) and oblivious heap sorting [28] (denoted as ODBJ (Heap)). For our ORAM approach, 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., Δ = 1 (see Table 2).

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 encryption.

Setup. The client is an Ubuntu 18.04 machine with 18 GB memory. The server is an Ubuntu 18.04 machine with 256 GB memory and 2 TB hard disk. The bandwidth is 1 Gbps.

Default parameter values. We set block size B = 4 KB, as in [11], [39], [53]. We set trusted memory size M = 2B (B is block size) in ODBJ and our method, but set M = 50 log N 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 TPC-H benchmark. Query TE1-TE3 and Query TM1-TM3 come from the conference version [49]. Appendix A shows Query TB1-TB2 in SQL.

• Query TE1-TE3: general equi-joins over 2 tables.
• Query TM1-TM3: general multiway joins over 3-5 tables.

Social graph. Social graph [82], [83] contains twitter friendship links. We set default user number to 20,000 (with raw data size 4.5 MB) and vary user numbers from 5,000 to 200,000 (with raw data size from 1.3 MB to 58 MB). The following queries come from the conference version [49].

- Query SE1-SE3: general equi-joins over 2 tables.
- Query SM1-SM3: general multiway joins over 3-4 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 9-16). For simplicity, we mainly focus on experimental results for query cost.

9.2 Cloud and Client Storage Costs

Figures 7a and 8a show cloud storage cost on two datasets. ObliDB and ODBJ achieve the minimum cloud storage cost, since they only store encrypted data blocks. Raw Index(+Cache) needs a little more cost for storing index blocks. ORAM based method has roughly 10X larger cost than Raw Index(+Cache), due to building ORAM data structure.

Figures 7b and 8b show 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 retrieved B-tree paths and may cache some index blocks. For ObliDB, we set trusted memory size \( M = 50 \log N \) and make it finish as soon as possible. For ORAM based method, the client memory cost grows (roughly) linearly with raw data size, due to \( O(N/B) \) blocks in the position map.

9.3 Performance of Binary Equi-Join

9.3.1 Default Setting

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

Our SepORAM(+Cache) brings 90X-450X larger query cost than Raw Index(+Cache) except for Query SE1, and also brings 7X-15X and 40X-160X larger query cost on TPC-H and social graph than ODBJ (Bitonic) except for Query SE1. The major reason is that data tuple size is much less than block size. For index based methods (Raw Index(+Cache) and ours), only one index entry or data tuple in each retrieved block contributes to the join processing. For ODBJ method, ODBJ (Heap) brings 4X-9X and 25X-37X larger query cost on TPC-H and social graph than ODBJ (Bitonic), since oblivious heap sort [28] works better in memory but does not achieve good IO performance. In contrast, oblivious external bitonic sort [45] is more IO-efficient. Note that data tuple size is much less than block size on both datasets, even if the trusted memory contains only two blocks, the trusted memory actually holds decades of tuples.

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 cost than Raw Index(+Cache) algorithms. Sep INLJ(+Cache) even achieves 17X-27X better performance than ODBJ (Bitonic). 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 7, One INLJ(+Cache) has to pad the number of ORAM accesses for each tuple retrieval to the maximum length of outsourced B-tree paths, although this problem can be alleviated by index caching. One SMJ does not need padding, since the client always accesses an index block and then a data block for each tuple retrieval through ORAM. One SMJ even achieves 1.6X better performance than Sep SMJ on Query SE2 and SE3, due to less number of tuple retrievals based on the optimization in Section 7. Last, the index caching brings 1.2X-1.6X speedup ratio.

9.3.2 Scalability

Figures 11a and 12a show query cost for Query TE2 and SE2 against raw data size. Our SepORAM(+Cache) achieves 2X-4X and 1600X-16000X better performances than ObliDB for Query TE2 and SE2, when raw data size increases from the minimum to the maximum. The speedup difference for two queries is still on account of the join result size, as explained in Section 9.3.1. Compared with Raw Index(+Cache), Se-
### 9.4 Performance of Band Join

Figure 13a shows query cost for band join on TPC-H in default setting. Compared with Raw INLJ(+Cache), our extended ODBJ (Bitonic), SepORAM(+Cache) brings 75X-157X and 161X-409X larger query cost on Query TE2 and SE2. Compared with ODBJ (Bitonic), SepORAM(+Cache) brings 10X-20X and 30X-140X larger query cost on Query TE2 and SE2 than ODBJ (Bitonic), since oblivious heap sort [28] is suitable in memory but not IO-efficient. For our method, Sep INLJ achieves 1.1X-3.4X better performance than One SMJ, as explained in Section 9.3.1. In Section 9.3.1, our Sep INLJ(+Cache) 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.

#### Fig. 11. Performance of Query TE2 against raw data size.

<table>
<thead>
<tr>
<th>Raw Data Size (MB)</th>
<th>Query Cost (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10^{-1}</td>
<td>Sep INLJ+Cache</td>
</tr>
<tr>
<td>10^{2}</td>
<td>Sep INLJ+Cache</td>
</tr>
<tr>
<td>10^{5}</td>
<td>Sep INLJ+Cache</td>
</tr>
<tr>
<td>10^{8}</td>
<td>Sep INLJ+Cache</td>
</tr>
<tr>
<td>10^{11}</td>
<td>Sep INLJ+Cache</td>
</tr>
<tr>
<td>10^{14}</td>
<td>Sep INLJ+Cache</td>
</tr>
</tbody>
</table>

#### Fig. 12. Performance of Query SE2 against raw data size.

<table>
<thead>
<tr>
<th>Raw Data Size (MB)</th>
<th>Query Cost (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10^{-1}</td>
<td>Sep INLJ+Cache</td>
</tr>
<tr>
<td>10^{3}</td>
<td>Sep INLJ+Cache</td>
</tr>
<tr>
<td>10^{7}</td>
<td>Sep INLJ+Cache</td>
</tr>
<tr>
<td>10^{11}</td>
<td>Sep INLJ+Cache</td>
</tr>
<tr>
<td>10^{15}</td>
<td>Sep INLJ+Cache</td>
</tr>
<tr>
<td>10^{19}</td>
<td>Sep INLJ+Cache</td>
</tr>
</tbody>
</table>

#### Fig. 13. Performance of band join on TPC-H.

pORAM(+Cache) brings 75X-157X and 161X-409X larger query cost on Query TE2 and SE2. Compared with ODBJ (Bitonic), SepORAM(+Cache) brings 10X-20X and 30X-140X larger query cost on Query TE2 and SE2. The major reason is still that data tuple size is much less than the block size, as explained in Section 9.3.1. For ODBJ method, ODBJ (Heap) brings 5X-15X and 16X-37X larger query cost on Query TE2 and SE2 than ODBJ (Bitonic), since oblivious heap sort [28] is suitable in memory but not IO-efficient. For our method, Sep INLJ achieves 1.1X-3.4X better performance than One SMJ, as explained in Section 9.3.1. In Section 9.3.1, our Sep SMJ 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.

#### 9.5 Performance of Multiway Equi-Join

##### 9.5.1 Default Setting

Figures 15a and 16a show query cost for multiway equi-join on two datasets in default setting. Our Sep INLJ(+Cache) achieves $10^8 \times 10^{11}$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, the speedup ratio goes down to 280X, since the join result size is roughly proportional to Cartesian product size. Compared with Raw INLJ(+Cache), Sep INLJ(+Cache) brings 185X-985X and 3700X-7000X larger query cost on TPC-H and social graph, due to ensuring the obliviousness. For our method, Sep INLJ achieves 1.6X-5.5X better performance than One SMJ, since One SMJ has to access the large single ORAM. Last, index caching brings 1.1X-1.5X speedup ratio.

#### 9.6 Padded Mode vs. Non-Padded Mode

We also make the comparison between padded mode and non-padded mode for all secured methods. We discuss three padding strategies for join result size: (1) no padding
entry level, and data tuple level. Specifically, horizontal axis means the discretized time. For $T_1-T_4$, vertical axis means the index level in $B$-tree for $T_1-T_4$; each light bar means an ORAM read, and each dark bar means an ORAM write. For $T_{out}$, vertical axis means the record index; each light bar means a record read, and each dark bar means a record write. We have verified that given the specific input and output sizes ranging from 10 to 10,000, the tests for different input tuples produce the same logs of access patterns.

10 Conclusion
This work supports general band joins and multiway equi-joins obliviously based on non-ORAM approach [30] and ORAM approach [49]. Non-ORAM approach stores input tables in flat storage and achieves better performance in join processing, but needs some delicate design of oblivious operations. ORAM approach builds oblivious indices over input tables directly, but usually brings larger computation overhead in join processing. As with ObliDB [31], accessing a few rows in any table should use the indexed storage, while the flat storage performs better for accessing large segments. Hence, to design the query optimizer for different approaches is a crucial point in building encrypted or oblivious databases. Note that our current design does not address challenges associated with ad-hoc updates, which is a future direction. Last, how to support query concurrency in an efficient manner using ORAM is still a major challenge.

APPENDIX A
Query TB1: Suppliers joined with other suppliers with the difference of account balances within $[-100.00, 100.00]$.

```
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;
```

ACKNOWLEDGMENTS
This work was supported by National Key R&D Program of China (2021YFB3101100) and the National Natural Science Foundation of China under Grant 62220106004, 61972308, 62302368. This work was partially done when Zhao Chang worked as a visiting scholar at Alibaba DAMO Academy.

REFERENCES
Fig. 20. Access pattern of our oblivious band join algorithm in ODBJ (Bitonic).

Fig. 21. Access pattern of our oblivious multway equi-join algorithm.


