Doquet: Differentially Oblivious Range and Join Queries with Private Data Structures

Lina Qiu  
Boston University  
qlina@bu.edu

Georgios Kellaris  
Lerna AI  
georgiosk@lerna.ai

Nikos Mamoulis  
University of Ioannina  
nikos@cs.uoi.gr

Kobbi Nissim  
Georgetown University  
kobbi.nissim@georgetown.edu

George Kollios  
Boston University  
gkollios@bu.edu

1 INTRODUCTION

Companies and organizations have the opportunity to use cloud platform services for storing and querying their data [26]. However, data management outsourcing bears the risk of leaking private and sensitive information. Even if data are encrypted, an honest-but-curious cloud service provider can infer sensitive data without being detected by data owners [10, 25, 30, 35, 49, 51, 52, 54]. Such an adversary, who would not deviate from the defined protocol, but would intentionally try to learn some or all of the sensitive information, poses a big threat to sensitive data outsourcing.

A Trusted Execution Environment (TEE) [37, 45] is an isolated enclave in which a small amount of trusted code can be securely executed on sensitive data. States and computations internal to the TEE cannot be observed by processes running at higher privilege levels outside the enclave. With dedicated hardware supporting memory encryption, clients can create a TEE on the cloud for their sensitive data and run query workloads with a small overhead on performance. Despite the appealing cryptographic features, the current hardware-supported implementations of TEEs, like Intel SGX [14, 29] and AMD SEV [50], are subject to side-channel attacks, due to leakages of 1) memory access patterns and/or 2) communication volume. Specifically, an honest-but-curious adversary may recover private data 1) by tracking the list of physical addresses that a program accesses, or 2) by observing the lengths of messages between the client and server [25, 30]. Oblivious query evaluation is a recently proposed approach toward preventing side-channel attacks in TEE-based database outsourcing.

Full Obliviousness (FO). Oblivious RAM (ORAM) [22, 23] is a generic approach to hiding access patterns. ORAM ensures full obliviousness for any two equal-length sequences of memory access patterns, but it does not hide the lengths of the sequences, so ORAM is prone to communication volume attacks. To meet the guarantee of FO, ORAM-based query algorithms should pad intermediate query results exhaustively to their worst-case lengths. Further, ORAM exhibits an inherent logarithmic multiplicative overhead [9, 22, 23, 36] and is expensive in practice. To tackle these issues, some methods [12, 21, 34, 56] allow the leakage of output size. These techniques claim themselves as being FO, they are prone to communication volume attacks [25, 30].

Differential Obliviousness (DO). To alleviate the overhead of FO, Chan et al. propose a privacy notion called $\epsilon, \delta$-differential obliviousness (DO) [11], which requires that the memory access patterns of a program and the lengths of intermediate results satisfy $\epsilon, \delta$-differential privacy (DP). If a query processing algorithm is
We present the concepts and constructs that are used or improved in our work that provides a DO guarantee against honest-but-curious adversaries satisfying \((\epsilon, \delta)-DP\). Using this notion, very recently, Qin et al. propose a system called Adore [42] that achieves DO for a few relational operations, including selection, grouping with aggregation, and foreign key join, using SGX enclaves. However, Adore makes the assumption that the enclave memory (private memory) is perfectly secure in the sense that all execution and accesses in the private memory are invisible to the adversary. Therefore, Adore does not use oblivious algorithms inside the enclave.

**Doquet.** We present Doquet: the first data management framework that provides a DO guarantee against honest-but-curious adversaries in a realistic TEE setting. Doquet does not make the assumption that memory access patterns in the TEE are invisible, since the current implementations (e.g., Intel SGX) have known vulnerabilities for access pattern leakage. All query algorithms in Doquet, as well as the construction of private data structures (PDS) and indices are DO. In addition to being more secure, Doquet also outperforms Adore on selections and foreign key joins.

Our contributions can be summarized as follows:

- We describe a framework for DO outsourced database systems, which can answer any number of queries with the same initial privacy budget, while the overall information leaked to honest-but-curious adversaries satisfies differential privacy.
- We introduce novel private data structures (PDS) and indices, whose construction and use are proved to be DO. Based on them, we develop DO algorithms for selections and joins. We present the first practical DO solution to select-join compositions.
- We analyze our DO algorithms in three performance metrics: runtime complexity, storage, and communication volume.
- We evaluate our DO algorithms on Intel SGX (2nd gen.). We empirically show the substantial performance gain of our DO algorithms in comparison with their state-of-the-art counterparts.

## 2 BACKGROUND

We present the concepts and constructs that are used or improved by Doquet. Table 1 summarizes the notation used in the paper. A database \(D\) consists of a collection of tables. Each table \(T\) in \(D\) can be abstracted as an array of \(n\) records \(T = \{r_1,\ldots,r_n\}\). Each record \(r_j\) is in the form of \((\text{rid}_j, v_j)\), where \(\text{rid}_j\) is the unique identifier of the record, and \(v_j = \{v_j(a_1),\ldots,v_j(a_m)\}\) is a list of values corresponding to the attributes \(\{a_1,\ldots,a_m\}\) that \(T\) is defined on. We use Domain\(_K(a_k)\) to denote the domain of attribute \(a_k\), which is a superset of distinct values of \(a_k\) in table \(T\).

### 2.1 Differential Privacy

Two databases \(D_1, D_2\) over domain \(X\) are called neighboring (denoted by \(D_1 \sim D_2\)) if one of them can be obtained from the other through one insertion or one substitution of an element of \(X\).

**Definition 1.** Let \(X, Y\) be two random variables over the same support \(\Omega\). We write \(X \approx Y\) if for all \(S \subseteq \Omega\),

\[
\Pr[X \in S] \leq e^{\epsilon} \Pr[Y \in S] + \delta \quad \text{and} \quad \Pr[Y \in S] \leq e^{\epsilon} \Pr[X \in S] + \delta.
\]

**Definition 2 \((\epsilon, \delta)-\text{DP} [17, 18]\).** Let \(\epsilon > 0\) and \(\delta \in [0, 1)\). A randomized mechanism \(M\) is \((\epsilon, \delta)-\text{differentially private}, (\epsilon, \delta)-\text{DP}\) for short, if \(M(D_1) \approx M(D_2)\) for all \(D_1 \sim D_2\).

**Definition 3 \((\text{Sensitivity}) [18]\).** For a query \(q\) that maps a dataset to a vector of real numbers, the global sensitivity \(\Delta_q\) of \(q\) at database \(D_1\) is as follows:

\[
\Delta_q(D_1) = \max_{D_2 \sim D_1} \|q(D_1) - q(D_2)\|_1
\]

**Definition 4 \((\text{Geometric Mechanism}) [11]\).** The geometric distribution over integers, denoted Geo\((\alpha)\), assigns to \(x \in \mathbb{Z}\) probability \(\frac{1}{\alpha^2} \cdot \alpha^{-\lfloor x \rfloor}\). The shifted and truncated geometric distribution Geo\((\epsilon, \delta, \Delta, p)\) has support in \([p-\Delta/2, p+\Delta/2]\), where \(\Delta = 2(k_0+\Delta-1)\) and \(k_0 = \frac{1}{\epsilon} \ln \frac{2}{2}\). An element of Geo is sampled as follows:

\[
G(\epsilon, \delta, \Delta, p) = \min\{\max\{p - \frac{\Delta}{2}, p + \text{Geo}(\epsilon, \delta)\}, p + \frac{\Delta}{2}\}
\]

The mechanism that on input \(D\) outputs \(q = q(D) + G(\epsilon, \delta, \Delta, p)\) is \((\epsilon, \delta)-\text{differentially private}.

In this work, we use \(G(\epsilon, \delta, \Delta)\) to denote Geo\((\epsilon, \delta, \Delta, 1/2)\), and Geo\((\epsilon, \delta)\) to denote Geo\((\epsilon, \delta, 1, 1/2)\), which is the most frequently used noise notation (i.e., sensitivity \(\Delta = 1\) and the support is in \([0, 1]\)). We assume \(\epsilon = \Theta(1)\) and \(\delta = 1/\mathcal{N}^c\) for a constant \(c > 1\) and a database of size \(N\).

**Proposition 1 \((\text{Post-processing}) [19]\).** If a randomized mechanism \(M : X \rightarrow R\) is \((\epsilon, \delta)-\text{DP}\), then \(G : R \rightarrow R'\) be an arbitrary mapping. We have \(G \circ M : X \rightarrow R'\) also satisfies \((\epsilon, \delta)-\text{DP}.

**Theorem 1 \((\text{Composition}) [19, 20]\).** Suppose there is a set of randomized mechanisms \(M = \{M_1, \ldots, M_m\}\), each \(M_i\) is \((\epsilon_i, \delta_i)\)-DP. If every \(M_i\) in the set is performed on a disjoint subset of the

**Table 1: Notation table**

<table>
<thead>
<tr>
<th>Query</th>
<th>(N)</th>
<th>The number of input records</th>
</tr>
</thead>
<tbody>
<tr>
<td>(r)</td>
<td></td>
<td>The actual output size</td>
</tr>
<tr>
<td>(R)</td>
<td></td>
<td>The noisy output size (communication volume)</td>
</tr>
<tr>
<td>(q)</td>
<td></td>
<td>The actual response of query</td>
</tr>
<tr>
<td>(R(q))</td>
<td></td>
<td>The noisy response of (q) including dummy records</td>
</tr>
</tbody>
</table>

**Differential Privacy**

\((\epsilon, \delta)\) The total privacy budget

\((\epsilon', \delta')\) The privacy budget to create PDS

\(U = \max \{G(\epsilon, \delta), \max \{\text{Geo}(\epsilon', \delta')\}\}\)

\((\epsilon_b, \delta_b)\) The privacy budget to create DP histogram

\((\epsilon_b, \delta_b)\) The privacy budget for bucketization

\((\epsilon_b, \delta_b)\) The privacy budget for join output compaction

\(\Delta_{geom}(T)\) The local sensitivity of the algorithm w.r.t. input \(T\)

\(\text{Privacy budgets subject to: } (\epsilon', \delta') = (\epsilon_b, \delta_b) + (\epsilon_b, \delta_b)\) and \((\epsilon, \delta) = (\epsilon', \delta') + (\epsilon_b, \delta_b)\)

**Tree**

\(h\) The tree height

\(h_b\) The tree branching factor

\(\bar{\tau}(\cdot), \bar{\tau}(\cdot), \bar{\tau}(\cdot)\) The noisy count (capacity), the real count and the number of dummy records added to tree node/interval \(a\), subject to \(\bar{\tau}(\cdot) = \bar{\tau}(\cdot) + b(\cdot)\)

**PDS**

\(B\) The number of buckets in PDS

\(\theta\) The threshold used to decide bucket domains

\((\text{Domain}(a_k))\) The domain of attribute \(a_k\), \(\{x_1, \ldots, x_k\}\), with size \(D\)

\(\bar{\tau}(b_k), \bar{\tau}(b_k), \bar{\tau}(b_k)\) The noisy count (capacity), the real count and the number of dummy records added to bucket \(b_k\), subject to \(\bar{\tau}(b_k) = \bar{\tau}(b_k) + b(\cdot)\)

\(L(\text{domains})\) A list \(\{\text{Domain}(b_k)\}\) for all buckets \(b_k\)

\(L(\text{capacities})\) A list \(\{\bar{\tau}(b_k)\}\) for all buckets \(b_k\)
entire dataset, $M$ provides $(\max(\epsilon_1, \ldots, \epsilon_m), \max(\delta_1, \ldots, \delta_m))$-DP.

If $M$ is performed sequentially on the same dataset, $M$ provides $(\sum_{i=1}^{m} \epsilon_i, \sum_{i=1}^{m} \delta_i)$-DP.

2.2 Differential Obliviousness

Definition 5. A randomized mechanism $M$ is $(\epsilon, \delta)$-differentially oblivious, or $(\epsilon, \delta)$-DO, if $\text{View}^M(D_2) \approx \epsilon, \delta \text{View}^M(D_1)$ for all $D_1 \sim D_2$, where $\text{View}^M(D_i)$ is the sequence of memory addresses (access pattern) generated by the random execution of $M$ on input $D_i$.

Definition 6. A mechanism $M$ $\delta$-obliviously implements a functionality $F$ with leakage $L$, if there exists a simulator $SIM$ that produces a simulated access pattern, such that for any security parameter $\lambda$ and any input $D$, the executions $\text{Ideal}: O_{\text{ideal}} \leftarrow F(\lambda, D, \rho)$ and $\text{Leak}_{\text{ideal}} \leftarrow L(\lambda, D, \rho)$, where $\rho$ is the random bits needed by $F$.

Real: $(O_{\text{real}}, \text{Leak}_{\text{real}})$ $\approx M(\lambda, D)$, where $M$ is the access pattern of $M$ having $\delta(\lambda)$-statistically close distributions:

$$(O_{\text{ideal}}, \text{Leak}_{\text{ideal}}) \approx (O_{\text{real}}, \text{Leak}_{\text{real}})$$

Lemma 1. If a mechanism $M$ $\delta'$-obliviously implements a functionality $F$ with leakage $L$, where $L$ is $(\epsilon, \delta)$-DO with respect to the input, then $M$ is $(\epsilon, \delta + \delta')$-DO.

2.3 Fully Oblivious Building Blocks

We review three FO building blocks used throughout Doquet.

2.3.1 OblisSort. In our work, we use bitonic sort [5], which is one of the most used algorithms, having $O(N \log^2 N)$ time complexity. Although there exist alternatives ($O(N \log N)$) time oblivious sorters [1, 3, 24, 44], these either have a large constant hidden by the big $O$ notation [1, 24], or have a non-trivial implementation [3, 44]; hence, they run slower than bitonic sort when given a reasonable input size [1, 3]. OblisSort($A, a_s$) sorts array $A$ by attribute $a_s$.

2.3.2 ObliCompact. Given an array of size $N$ containing $n_1$ distinguished items and $n_2 = N - n_1$ non-distinguished items, oblivious compaction moves all the $n_1$ distinguished items to the front of the output array, and all the $n_2$ non-distinguished items to the end. In our work, we use the tight order-preserving oblivious compaction scheme of [47] with $O(N \log N)$ runtime complexity. ObliCompact($A, n_1$) compacts array $A$ and resizes it to size $n_1$.

2.3.3 ObliExpand. Given an input $X = \{x_1, \ldots, x_B\}$ and an injective function $f : X \rightarrow [n]$, oblivious expansion distributes $x_i$ to position $f(x_i)$ in the output array of size $n$ [34]. The injective function has the property that $f(x_i) < f(x_{i+1})$ for $i \in [1, B - 1]$. After the expansion, the positions $p$ between $(f(x_1), f(x_{i+1}))$ are filled with dummy elements. ObliExpand is the inverse of ObliCompact, and they have the same runtime complexity. Its obliviousness is defined w.r.t. $B$ and $n$. ObliExpand($X, F, n$) has the injective function defined as $f(x_i) = F[i]$.

2.4 DP Data Structures

A private data structure (PDS) built upon a DP histogram [27, 38, 41, 55] transforms and organizes private data to facilitate DO query processing [31]. PDS is defined w.r.t. a specific set of private data and a subset of the data’s attributes. For example, PDS($T, S$) is defined w.r.t. table $T$ and a subset $S$ of its attributes. For a specific set of private data, the DO database may store multiple PDS for different subsets of attributes to support different query sets. Each of these PDS requires a different copy of the original data and consumes a portion of the overall privacy budget $(\epsilon, \delta)$.

In Doquet we mainly build upon the HTree [27] DP histogram algorithm for differentially obliviously constructing private data structures. An overview of HTree is given in Algorithm 1. A sequence of hierarchical intervals is arranged in a $Tree$. Each node $v \in Tree$ corresponds to an interval, and each node has $k$ children corresponding to $k$ equally sized subintervals. Suppose that $Tree$ is built for attribute $a_s$ of table $T$ and its domain is $\text{Domain}(a_s) = \{x_1, x_2\}$, the unit-length intervals $\{x_1, \ldots, x_2\}$ are the leaves and the root is the entire domain. The privacy budget $(\epsilon_k, \delta_k)$ is reserved for HTree to hide the real count $c(v)$ of each node, i.e., the number of records falling in the node’s interval. $C = \{c(x_1), \ldots, c(x_2)\}$ is the real counts for unit-length intervals $\{x_1, \ldots, x_2\}$.

Because the noise of each node $\tilde{n}(v)$ is generated independently, there may be a parent count that does not equal the sum of its children (inconsistency). Constrained Inference (CI) techniques [27] post-processes the tree to derive a consistent estimate for each node, such that every parent count is equal to the sum of its children. Applying CI only requires two linear scans of the tree. The output of HTree is $H(T) = \{\tilde{n}(v) \in \tilde{n}(v) : v \in \{x_1, x_2\}\}$. We define the leakage of HTree as $\mathcal{L}(HTree) = \sum_{v} \tilde{n}(v) \cdot \tilde{n}(v)$. Because $\tilde{n}(v)$ and $\tilde{n}(v)$ are private and will not be used directly. The leakage is $(\epsilon_k, \delta_k)$-DP [27].

Proposition 2. The independent noise added to each node of the HTree is $O(hU)$, where $h$ is the height of the tree, and the error for answering any range query is $O(h^2U)$.

3 PROBLEM DEFINITION

In this section, we present the DO database outsourcing model, adapted from [30] where our Doquet framework applies. Then, we formally describe the threat model and security guarantees.

3.1 DO Database Outsourcing Model

Fig. 1 is an overview of the DO database outsourcing model. We assume a client $\mathcal{U}$, who is the data owner, and a service provider

---

1The problem of selecting which PDS to create and how much privacy budget to give for each of them is an interesting direction for future research.
We model the server \( S \) to be an honest-but-curious adversary, i.e., it is trusted not to deviate from its prescribed protocol but may try to glean information from what it observes during the execution. TEE is the only trusted component within \( S \). \( S \) cannot view sensitive data or interfere with computations inside the enclave. Data are encrypted before they are swapped out from the enclave and written to the untrusted storage. Similarly, data received from and sent to the client \( U \) are also encrypted. Hence, the server can only observe the memory accesses and the lengths of encrypted messages made by the TEE.\(^2\)

For every query sequence \( Q = \{q_1, q_2, \ldots\} \) submitted by \( U \), we define the server view \( \text{View}_Q(D) \) to be the random variable corresponding to the sequences of memory accesses and messages between \( U \) and the TEE during the execution of \( Q \) on database \( D \). We require that \( \text{View}_Q(D_1) \approx \text{View}_Q(D_2) \) for all \( Q \) and all \( D_1 \sim D_2. \)\(^3\) In other words, we require the DO outsourcing system to provide protection of differential privacy even in face of a semi-honest adversary that 1) can monitor every read/write to memory/disk, 2) has admin privileges and hence can view all internal and external communication, and 3) can even choose the queries \( Q \).

### 4 DATA STRUCTURES AND INDICES

In this section, we present how does Doquet differentially obliviously create private data structures and indices. The construction is done at the server side, after the initial upload of encrypted data to the server, with the help of the TEE (i.e., SGX), and without any interaction with the data owner.

#### 4.1 Private Data Structures

We first present \( \text{CreatePDS}(T, a_s) \), a methodology for differentially obliviously computing a DP histogram \( H(T, a_s) \) and constructing a private data structure \( PDS(T, a_s) \) for attribute \( a_s \) of table \( T \) at the server. When the attribute \( a_s \) is implied by the context, we omit it and write \( H(T) \), \( PDS(T) \) respectively. \( \text{CreatePDS} \) consists of two steps: 1) \( \text{Histogram} \) obliviously computes a DP histogram and 2) \( \text{CreateBuckets} \) moves data to the corresponding buckets based on the DP histogram to form \( PDS(T) \).

\(^2\)Our analysis assumes perfect encryption. Our implementation uses Rijndael AES-GCM encryption with a 128-bits key that provides computational DP [39].

\(^3\)We note that our construction withstands a slightly stronger adversarial model, where the queries may be selected adaptively, i.e., where each query \( q_i \) is selected as a function of the lengths of answers given to the previous queries \( q_1, \ldots, q_{i-1} \).
4.1.1 Histogram. Algorithm 2 takes as input a table \( T \), an attribute \( a_k \) of \( T \), a tree branching factor \( k_b \), and a privacy budget \( (\epsilon_b, \delta_b) \). It first obliviously computes the multiplicity of each distinct value \( x_j \) (representative) of \( a_k \) in \( T \) (lines 1-4). Then, it creates an array \( A \) that stores for each \( x_j \) its multiplicity. In particular, \textsc{ObliCompact} is invoked to move all the representatives to the beginning of array \( A \) (line 5). The number of distinct values is private. Hence, the size of \( A \) is set to the size \( D \) of \text{Domain}(a_k). If the number of distinct values is smaller than \( D \), the rest of the array is filled with dummy entries. After that, with the help of another array \( F \), we obtain the input \( C = \{c(x_1), \ldots, c(x_D)\} \) of \text{HTree}, i.e., an array of exact counts for each value in the domain. Specifically, \textsc{ObliExpand} is invoked to distribute non-dummy entry \((x_j, c(x_j))\), i.e., a domain value \( x_j \) and its real count \( c(x_j) \), to position \( j \) (line 13). Finally, an \text{HTree} is constructed (Algo. 1) from the expanded array \( C \). A detailed example of running Algo. 2 is shown in Fig. 2. In the example, we have \text{Domain}(a_k) = [1, 4]. The representatives are highlighted in the figure.

4.1.2 CreateBuckets. A bucket \( b \) in \text{PDS}(T, a_k) \( \) corresponds to an interval \text{Domain}(b) \( \subseteq \text{Domain}(a_k). \) The range of the interval is determined by the DP histogram, and all records in \( T \) within the interval should be moved to the bucket by CreateBuckets. In addition, the bucket is padded with dummy records in order to hide the exact number of records. Next, we present a differentially oblivious bucketization algorithm to achieve that.

Algo. 3 takes a \((\epsilon_b, \delta_b)\)-DP histogram (i.e. the \text{HTree}), the number of buckets \( B \), and a privacy budget \( (\epsilon_b, \delta_b) \) as input. The average number of records per bucket is \( \theta = \frac{\sum H(T)}{B} \), where \( \sum H(T) = \sum_{v \in X} c(v) \) is the sum of noisy counts of \text{HTree} leaves. Notice that the algorithm can run at different levels of the tree, not only the leaf level. If the cumulative noisy count \( \sum_{v \in X} c(v) < \theta \), we keep merging the next interval to the current bucket \( b_1 \) (lines 2-9). Otherwise, the bucket’s domain is chosen, and we keep both the real and noisy count \( c(b_1) \) and \( \hat{c}(b_1) \) along with the domain. The capacity of the resulting private data structure is \(|PDS(T)| = \sum \hat{c}(b_i)|. After deciding the ranges of buckets, data is moved to the corresponding buckets via an expansion (line 13). The array \( V \) of size \(|T| \) stores a mapping: entry \( T[i] \) in the sorted table \( T \) should be distributed to position \( V[i] \) in the expanded table of size \( \sum \hat{c}(b_i). \) The positions in the expanded table \( T \) without real elements mapped to them are filled with dummy records (see Sec. 2.3.3). The output of CreateBuckets is \( PDS(T) = \{b_i \text{ for all buckets}\}. \)
single entity of the array \( C \) by 1). Hence, the application of HTree (i.e., \( L(\text{HTree}) \)) preserves \((\epsilon_h, \delta_h)\)-DP. The execution of HTree is FO and again only leaks the publicly-known domain size \( D = |C| \), since its memory access involves only linear scans of the array \( C \).

CreateBuckets (Algo. 3): The merging of bins to buckets implemented in lines 1-9 is \( \epsilon \)-oblivious with leakage \( L(\text{HTree}) \), which is \((\epsilon_b, \delta_b)\)-DP. Hence, by Lemma 1, the merging procedure is \((\epsilon_b, \delta_b)\)-DO. The sensitivity of the real counts of buckets \( \{c(b_i)\} \) for all buckets \( b_i \) is 1, hence \( L(\text{capacities}) = \{\tilde{c}(b_i)\} \), which for all \( b_i \) is \((\epsilon_b, \delta_b)\)-DP given that \( \tilde{c}(b_i) = c(b_i) + G(\epsilon_b, \delta_b) \). The access pattern of moving data to buckets (lines 10-18 in Algo. 3) is oblivious w.r.t. \( L(\text{capacities}) \), so it is \((\epsilon_b, \delta_b)\)-DO by Lemma 1 again. Based on Theorem 1, we hence get CreateBuckets is \((\epsilon', \delta')\)-DO where \( \epsilon' = \epsilon_b + \epsilon_h \) and \( \delta' = \delta_h + \delta_b \). Note that the bucket domains are constructed from \( L(\text{HTree}) \) only, so \( L(\text{domains}) \) is a post-processing of \( L(\text{HTree}) \) and also \((\epsilon_b, \delta_b)\)-DP by Proposition 1.

CreatePDS (Algo. 4) is \((\epsilon', \delta')\)-DO given that Histogram is FO and CreateBuckets is \((\epsilon', \delta')\)-DO.

4.1.5 Multidimensional PDS. The balanced hierarchical HTree has low Mean Squared Error (MSE) for single dimensional range queries, but it has been shown to be inefficient for multiple dimensions [41]: the number of unit-length intervals has to be extremely large to outperform the naive flat method (Sec. 3.1 in [41]) and HTree construction soon becomes infeasible because of the huge number of intervals. PrivTree [55] is a DP histogram algorithm for multidimensional queries. It outputs a set of buckets that contain roughly the same number of real records, similar to our bucketization procedure (Sec. 4.1.2) that merges HTree's unit-length intervals into buckets. Let \( L(\text{PrivTree}) \) be the leakage of PrivTree: \( L(\text{PrivTree}) = \{\{\text{Domain}(b_i), \tilde{c}(b_i)\} \mid \forall b_i\} \). Given privacy budget \((\epsilon', \delta')\), the leakage of PrivTree is \((\epsilon', \delta')\)-DP [55]. If HTree is replaced by PrivTree in our framework, Histogram is oblivious w.r.t. \( L(\text{PrivTree}) \). CreateBuckets, again, moves data to the corresponding buckets and pads each bucket to its capacity \( \tilde{c}(b_i) \), which should also be oblivious w.r.t. \( L(\text{PrivTree}) \). Overall, CreatePDS is \((\epsilon', \delta')\)-DO according to Lemma 1. The details of Histogram and CreateBuckets using PrivTree are not included, due to space constraints. We instead provide a code implementation [43] of PrivTree-based multidimensional CreatePDS.

4.2 Indices

Creating indices at the granularity of buckets can speed up query processing without violating DO requirements. We now show how to create \( B^+ \)-tree indices for \( PDS(T) \), which are also \((\epsilon', \delta')\)-DO. Let \( \text{Domain}(b_i) = [x_l, x_r] \) be the domain of bucket \( b_i \). Two \( B^+ \)-trees are defined to support range queries \([q_s, q_e] \): one holding key-value pairs \( \{(x_l, b_i) : \forall b_i\} \) and the other for \( \{(x_r, b_i) : \forall b_i\} \). The first (second) index is used to filter out buckets whose \( x_s > q_e \) \( (x_e < q_s) \). The access patterns of \( B^+ \)-tree indices (insertion and search) are determined by the bucket domains, which were revealed to the adversary in CreatePDS. Therefore, the \( B^+ \)-tree indices do not yield any leakage beyond \( L(\text{domains}) \). In general, any index built at the granularity of buckets can be easily shown to satisfy \((\epsilon', \delta')\)-DO.

5 DO QUERY PROCESSING

In this section, we propose operators DO-select and DO-join, and discuss their composition DO-select-join. For each algorithm, we give a suggested value for the number of buckets \( B \), and show that it leads to a satisfactory trade-off among runtime, storage, and communication volume. Other query operators such as projection and aggregation require at most one scan to compute the results. In other words, projection and aggregation can be combined with any one of our DO algorithms as the last step without incurring extra data shuffling or padding to guarantee DO. We leave grouping and update operations as our future work. The comparison with the state-of-the-art counterparts is summarized in Table 2.

### 5.1 Selections

Selection queries, such as “SELECT * FROM \( T \) WHERE \( a_s \) BETWEEN \( q_s \) AND \( q_e \)”, choose elements from a table that satisfy a given set of predicates. If no index was built, DO-select checks for each bucket in \( PDS(T) \) if its domain intersects with the query range \([q_s, q_e] \) (see Algo. 5). If an intersection was found, the bucket \( b_i \) may contain qualified records, so we add the whole bucket (all records in the bucket including real and dummy records) to the result set \( R(q) \) \( = R(q) \cup b_i \). Otherwise, the bucket can be safely skipped. If indices were created for \( PDS(T) \), they can be used for more efficient bucket searching. The example in Fig. 4 shows the structure of a simple \( PDS(T) \) consisting of two buckets \( b_1 \) and \( b_2 \). For the query range \([3, 4] \), only the domain of bucket \( b_2 \) overlaps with it and \( b_1 \) can be safely skipped. We add the whole bucket \( b_2 \) to the query response: \( R(q) = \{2, 3, 4, \bot, \bot\} \).

**Algorithm 5: DO-select** (\( T, a_s, q_s, q_e \))

1. Initialize \( B = cN/U' \).
2. if \( PDS(T, a_s) \) does not exist then
3. \( PDS(T, a_s) \leftarrow \text{CreatePDS}(T, a_s) \).
4. end
5. Search \( PDS(T, a_s) \) using indices or through one scan, add all buckets whose domains overlap with \([q_s, q_e] \) to \( R(q) \).

**Analysis** We now assess the storage complexity and communication volume given the number of buckets \( B = cN/U' \) \( (c \) is a constant and \( U' = \max G(\epsilon', \delta') \) chosen for selection queries. Each bucket contains approximately \( N/B \) real records and \( G(\epsilon_b, \delta_b) = O(U') \) dummy records. There are at most two bucket domains that intersect the query range \([q_s, q_e] \) without being contained in it. We hence get the communication volume \( O((1 + BU'/N)r + N/B + U') \).
Table 2: DO and FO denote differential and full obliviousness respectively. Opaque and ObliDB by default do not hide output size, but it is possible to add DP padding on the output. Shrinkwrap shrinks the output size from the worst-case to a DP value. Operator join includes many-to-many and foreign key join. \( S \) is the size of private memory. Select and join runtimes of Adore require \( S = \text{poly log } N \). Join runtime in bold is for many-to-many join. Let \( * \) denote the runtime of our implementation of their algorithms.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Private Memory</th>
<th>Degree of Obliviousness</th>
<th>Hide Communication Volume</th>
<th>Select</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>DO</td>
<td>×</td>
<td>DO</td>
<td>✓</td>
<td>select, join, select-join, ( O(\log N + R) )</td>
<td>( O(N \log^2 N + (r + NU) \log(r + NU)) )</td>
</tr>
<tr>
<td>Adore [42]</td>
<td>✓</td>
<td>DO</td>
<td>✓</td>
<td>select groupby, foreign key join</td>
<td>( O(N + N\log N) )</td>
</tr>
<tr>
<td>CZSC2 [13]</td>
<td>×</td>
<td>DO</td>
<td>✓</td>
<td>select, merge</td>
<td>( O(\log N) )</td>
</tr>
<tr>
<td>Opaque [56]</td>
<td>×</td>
<td>FO</td>
<td>✓</td>
<td>support DP padding select, groupby, foreign key join</td>
<td>( \leq (\log N) )</td>
</tr>
<tr>
<td>OBIOLDB [21]</td>
<td>✓</td>
<td>FO</td>
<td>✓</td>
<td>select, groupby, foreign key join, insert, update, delete</td>
<td>( O(N^2/S) )</td>
</tr>
<tr>
<td>Shrinkwrap [6]</td>
<td>×</td>
<td>FO</td>
<td>×</td>
<td>select, groupby, join</td>
<td>( O(N \log^2 N) )</td>
</tr>
</tbody>
</table>

For the chosen \( B = cN/U' \), the communication volume instantiates to \( O(r + U') \). The total number of real and dummy records after bucketization (a.k.a, the storage complexity) instantiates to \( |\text{PDS}(T)| \leq N + B + U' = O(N) \).

5.1.1 Proof of Differential Obliviousness. DO-select (Algo. 5) only accesses the data when it invokes the call to CreatePDS in line 3. The search in line 5 or the search through indices is a post-processing (Proposition 1) of the outcome of CreatePDS and does not introduce any cost in privacy.

\[
\text{PDS}(T_i)[v_1, v_4] = [3, 4]
\]

\[
\begin{align*}
&b_1: \text{skipped} & b_2: \text{overlapped} \\
&\text{Domain}(b_1) = [1, 1] & \text{Domain}(b_2) = [2, 4] \\
&1, 1, 1, 1, 1, 1, 1, 1 & 2, 2, 3, 4, 1, 1, 1
\end{align*}
\]

Figure 4: An example of DO-select

5.2 Joins

We consider equi-join queries between two tables \( T_1 \) and \( T_2 \), such as "SELECT * FROM \( T_1 \) AND \( T_2 \) WHERE \( T_1.a_1 = T_2.a_2 \)". Our key contribution to DO join query processing is a flexible and efficient DO-join operator, which can handle foreign key joins, many-to-many joins, and select-join compositions. Before presenting DO-join, we discuss an adaptation of ODBJ [34] that satisfies \( \epsilon, \delta \)-DO. Another side contribution of this work is a practical implementation of CZSC2 [13] with a runtime-efficient \( O(N \log^2 N) \) bitonic sorter and \( O(N \log N) \) oblivious compaction [47]. Both ODBJ and CZSC2 will be used as baselines of DO-join.

5.2.1 Adapted ODBJ

The algorithm first computes the number of join matches \( r \) by sorting the concatenation of two join tables. Then it expands \( T_1 \) and \( T_2 \) to size \( R = r + G(\epsilon, \delta, \max(|T_1|, |T_2|)) \), since changing a single input record can affect at most \( \max(|T_1|, |T_2|) \) number of output records. In the expanded table, every record in the original table has duplicates equal to the number of join matches that the record contributes to. The last step is to stitch the two expanded tables of size \( R \) to assemble the final join output. Notably, the logic of ODBJ degenerates to sort-merge in foreign key join. Let \( U = \max(G(\epsilon, \delta)) \), then \( G(\epsilon, \delta, \max(|T_1|, |T_2|)) \) has magnitude \( O(NU) \). The communication volume \( R = O(r + NU) \), and the runtime complexity \( O((r + NU) \log^2 (r + NU)) \).

5.2.2 DO-join. Our DO-join algorithm is a partition-based method: first we partition the two tables into buckets using CreatePDS, then we join the intersecting buckets using Cartesian product between the buckets that intersect, and finally we produce the results using a compaction algorithm. Based on whether the buckets are created independently or jointly, we have two join alternatives Ind-DO-join and Uni-DO-join.

DO-join, as shown in Algo. 6, takes as input two tables \( T_1 \) and \( T_2 \), a join attribute \( a_1 \), and a privacy budget \( (\epsilon, \delta) \) for compaction. If we use Ind-DO-join, \( \text{PDS}(T_1) \) and \( \text{PDS}(T_2) \) are constructed independently: \( \text{Domain}(T_1, b_1) \) may be different from \( \text{Domain}(T_2, b_1) \), where \( \text{Domain}(T_i, b_1) \) is the domain of the 1st bucket in \( \text{PDS}(T_i) \) for \( i \in [1, 2] \). For each bucket \( b_1^i \) in \( \text{PDS}(T_i) \), we need to check for a list of consequent buckets in \( \text{PDS}(T_j) \) for potential domain overlaps, and invoke the Cartesian product for every pair of overlapped buckets to guarantee join correctness (lines 8-16). The number of bucket-wise Cartesian products invoked by Ind-DO-join is between \( [B, 2B] \). The other alternative, Uni-DO-join, invokes CreatePDS only once for the concatenation (\( T_1, T_2 \)), so that the output \( \text{PDS}(T_1) \) and \( \text{PDS}(T_2) \) share the same bucket structure. Since \( \text{Domain}(T_1, b_1) \) and \( \text{Domain}(T_2, b_1) \) are equal in Uni-DO-join, there is no need to check neighboring buckets for potential overlaps and the total number of bucket-wise Cartesian products is \( B \). Notice that, for Uni-DO-join, we need to adapt CreatePDS Algo. 3 with the following changes: 1) the threshold \( \delta = (\sum H(T_1) + \sum H(T_2))/B \) in line 1; 2) the tuple count \( \tilde{c}(v) \) in line 3 is replaced with \( \tilde{c_1}(v) + \tilde{c_2}(v) \), where \( \tilde{c_1}(v) \) is the noisy count of node \( v \) in \( H(T_1) \); 3) if the condition in line 3 is true, two buckets \( b_1^i \) and \( b_2^i \) covering the same domain will be created for \( T_1 \) and \( T_2 \) respectively in line 4; 4) having the same bucket structure, lines 10-18 are performed independently for table \( T_i \) with \( b_1 \) replaced by \( b_1^i \).

The example in Fig. 5 visualizes the difference between Ind-DO-join and Uni-DO-join. Assume that \( B = 2 \). For Ind-DO-join, the result set before compaction \( \tilde{R}(q) \) contains 127 records produced by three bucket-wise Cartesian products \( (b_1^1, b_2^1), (b_3^1, b_2^2) \) and \( (b_1^2, b_2^2) \). For Uni-DO-join, \( \tilde{R}(q) \) contains 86 records produced by two bucket-wise Cartesian products \( (b_1^1, b_2^1) \) and \( (b_2^1, b_2^2) \). The advantage of Uni-DO-join is that the bucket boundaries of the two tables are the same and therefore, exactly one bucket for table \( T_2 \) needs to be joined with one bucket in table \( T_2 \). This produces smaller results for \( |\tilde{R}(q)| \) compared to Ind-DO-join, and therefore the final compaction algorithm is faster. On the other hand, in Ind-DO-join, the
Algorithm 6: DO-join (T₁, T₂, aₜ, (ε, δ))
1. Initialize B = chN/U, i = 1, j = 1;
2. if Ind-DO-join then
3. PDS(T₁) ← CREATEPDS(T₁, aₜ);
4. PDS(T₂) ← CREATEPDS(T₂, aₜ);
5. else
6. // Uni-DO-join
7. PDS(T₁), PDS(T₂) ← CREATEPDS((T₁, T₂), aₜ);
8. end
9. foreach bucket b₁ in PDS(T₁) do
10. while b₁ overlaps with b₂ do
11. Compute the Cartesian product of b₁ and b₂;
12. Add the result to array R(q);
13. i = i + 1;
14. if Ind-DO-join then j = j − 1;
15. end
16. R(q) = OBLISORTMERGE(b₁, b₂) to R(q);

In our approach, we set B = chN/U (c is a constant). By employing constrained inference, we get an average noise of b/u per domain value (Proposition 2). We assume that every bucket contains on average D/B domain values (leaf nodes of HTree). When a noisy bucket count exceeds the threshold θ, there are c(b₁) = O(1/(hU · D)) = O(U) real records in the created bucket. The noisy bucket count is O(1/θ) = c(b₁) + O(ε, δ) = O(U). Therefore, the sum of bucket-wise Cartesian product sizes is |R(q)| ≤ 2B · max b₁ = O(r/hN). When the domain size of the join attribute is comparable to the input size (i.e., D = O(N)), the grouping technique using B = chN/U reduces the amount of noise in R(q) by a factor of U/h = log₂ b₁, in comparison to the extreme case B = D. Actually, if the domain size is a constant and independent of N, |R(q)| = O(r + NU), which has been shown in [13] being close to the lower bound of DO join.

For the chosen B = chN/U, the storage complexity becomes |PDS(T₁)| = N + B · O(U) = O(hN) for D = O(N), or O(N) for a constant domain size. The local join query sensitivity is defined as the maximum noisy count of buckets [13] Δjoin(T₁, T₂) = max b₁ = O(U). Furthermore, the compacted join output size is R = r + O(ε, δ, Δjoin(T₁, T₂)) = O(r + U²), where the privacy budget (ε, δ) is a constant fraction of the total privacy budget subjecting to ε = ε + ε and δ = δ + δ. Finally, the communication volume is R = O(r + U²).

5.2.3 DO-PF-join. Existing approaches to fully oblivious foreign key join (PF-join) use a sort-based method; in particular, OBLISORTMERGE [21, 34, 56]. Even, for DO foreign key join, Adore [42] uses a sorting approach combined with their DOFILTER algorithm.

In this work, we consider a partition based approach for PF-join that aims to reduce the cost of the oblivious sorting of previous approaches. DO-PF-join, as shown in Algo. 7, first divides the foreign-key table T₂ into several partitions using CREATEPDS. It then expands the primary-key table T₁ to the domain size D, such that a record with join key xₗ ∈ {x₁, ..., x₄D} is at position xₗ = xₗ − x₄ + 1, where x₄ is the smallest value of Domain(aₜ) = {x₁, ..., x₄D}. OBLISORTMERGE is invoked for each partition covering Domain(bₘ). In the example depicted in Fig. 6, T₂ is divided to two partitions with domains [1, 2] and [3, 4] respectively. The primary key table T₁ is expanded to the domain size D = 4. The ith bucket bᵢ in T₁ is constructed based on the corresponding bₘ bucket in T₂ such that they share the same domain on the join attribute (line 4 in Algo. 7).
step 1-7 of Algo. 6 to get $PDS(T_i)$ $t \in \{1, 2\}$. Then the search strategy of DO-select can be used to obtain the set of buckets that satisfy the predicates. For the qualified buckets, it finds overlapped pairs and computes their Cartesian products, i.e., it continues from line 8 of DO-join. The local sensitivity of select-join $\Delta_{\text{select-join}}(T_1, T_2)$ is the maximum noisy count of the qualified buckets [13].

In order to support queries that contain selection predicates on non-join attributes, e.g., “SELECT * FROM $T_1$ and $T_2$ WHERE $T_1.a_3 = T_2.a_3$ AND $T_2.f_2 > 5$”, in a more efficient way than sending back to the client the complete set of join output of $T_1.a_3 = T_2.a_3$ without applying any selection predicate, $PDS(T_i)$ should support filtering operations for attribute $f_2$ and join operations for attribute $a_3$. In other words, $PDS(T_2)$ should be multi-dimensional. Suppose that $\text{Domain}(a_3) = \text{Domain}(f_2) = [0, 10]$ and $v = 7$, we use the example query to explain DO-select-join with selection predicates on non-join attributes. DO-select-join (Algo. 8) first creates a PrivTree-based two-dimensional $PDS(T_2, (a_3, f_2))$ as depicted in Fig. 7, and a one-dimensional $PDS(T_1, a_3)$. Then DO-select is invoked to obtain the set of buckets in $PDS(T_2, (a_3, f_2))$ that satisfy the predicate $T_2.f_2 < v$, i.e., $L_2 = \{b_2^1, b_2^2\}$. For each of the qualified buckets, it looks for the buckets in $PDS(T_1, a_3)$ that have domain overlaps on the join attribute $a_3$ with the qualified bucket (line 6 in Algo. 8). Cartesian product join is performed on each overlapped bucket pair: $(b_1^1, b_2^1), (b_1^2, b_2^2),$ and $(b_1^3, b_2^3)$. Lastly, DO-select-join computes the compacted noisy select-join result with local sensitivity $\Delta_{\text{select-join}}(T_1, T_2)$ defined as the maximum noisy count of buckets that participated in the bucket-wise Cartesian products.

5.2.5 Proof of Differential Obliviousness. The bucket-wise Cartesian product join in DO-join (Algo. 6 lines 8-16) and in DO-select-join (Algo. 8 lines 4-11) is again a post-processing (Proposition 1) of the outcome of CREATEPDS. The process reveals the bucket domains $\mathcal{L}(\text{domains})$ when looking for overlapping bucket pairs, and reveals bucket capacities $\mathcal{L}(\text{capacities})$ when computing bucket-wise Cartesian products. It does not introduce additional cost in privacy. We refer readers to Lemma 4.6 in [13] for the proof of $(\epsilon, \delta)$-DO join output compaction. Since the union of the leakages $\mathcal{L}(\text{domains}), \mathcal{L}(\text{capacities})$, and $(\epsilon, \delta)$-DP join output size is $(\epsilon, \delta)$-DP, and the fact that DO-join and DO-select-join are oblivious w.r.t. the $(\epsilon, \delta)$-DP leakage, we can conclude that DO-join and DO-select-join are $(\epsilon, \delta)$-DO by Lemma 1.

DO-PF-join (Algo. 7) without $(\epsilon, \delta)$-DO join output compaction is $(\epsilon', \delta')$-DO: the partitioning strategy CREATEPDS is $(\epsilon', \delta')$-DO; OblivExpand is fully oblivious with input size $|T_1|$ and output size $D$; fully oblivious OblivSortMerge is applied to each partition.

Algorithm 8: DO-select-join $(T_1, T_2, a_3, f_2, v)$

```plaintext
1. $PDS(T_1, a_3) \leftarrow$ CREATEPDS($T_1, a_3$);
2. $PDS(T_2, (a_3, f_2)) \leftarrow$ CREATEPDS($T_2, (a_3, f_2)$);
3. $L_2 \leftarrow$ DO-select($T_2, f_2, v, \text{inf}$);
4. foreach bucket $b_2$ in $L_2$ do
5. Let $\text{Domain}(b_2, a_3) = [x_1, x_2]$ be $b_2$’s domain of $a_3$;
6. $L_1 \leftarrow$ DO-select($T_1, a_3, x_1, x_2$);
7. foreach bucket $b_1$ in $L_1$ do
8. Compute the Cartesian product of $b_1$ and $b_2$;
9. Add the result to the Cartesian product $\hat{R}(\hat{q})$;
10. end
11. end
12. $R(q) = \text{OblivCompact}(\hat{R}(\hat{q}), R),$ where
13. $R = r + \mathcal{G}(\epsilon, \delta, \Delta_{\text{select-join}}(T_1, T_2))$;
```

6 EVALUATION

We implement Doquet [43] on the second generation of Intel SGX. While the first generation has enclave size limitations, in the new SGX the enclave size is up to 512 GB per CPU [29]. Hence, the server can keep more data in the enclave and go through fewer context switches between trusted and untrusted environments, boosting the performance compared to using a first-generation SGX.

Setup. As server, we use an Ubuntu 20.04 machine with 2.8GHz Intel Xeon Platinum 8370C CPUs and 384GB of RAM (256 GB of EPC). The SGX SDK version is 2.17. The entire execution is in memory; we set the SGX max heap size to be large enough so that no Enclave Page Cache (EPC) swapping will be triggered. The client is an Ubuntu 20.04 machine with 2.6GHz Intel Xeon Platinum 8171M CPU and 3.5 GB of RAM. The ping time between the client and the server is 6.5ms and the effective bandwidth is 90MB/s.

Datasets. The default synthetic dataset of size $N = 10^6$, which we use in PDS and DO-select evaluation, is multi-dimensional (with 5 attributes) and uniformly distributed. The domain size of each attribute is $D = 10^5$. In the datasets generated for join experiments, the key multiplicities are randomly drawn from a Zipfian distribution. We also evaluate Doquet on TPC-H and TPC-DS. The default size of TPC-H and TPC-DS is set to 100MB ($SF = 0.1$). Both TPC benchmarks do not contain queries that directly evaluate many-to-many joins, so we use simplified queries for evaluation. We use the join queries specified in [12] TE1-TE3 on TPC-H. We also run the following join queries TE4-TE6 on TPC-DS.

TE4: SELECT ss_ticket_number, ws_order_number FROM web_sales, store_sales WHERE ws_item_sk = ss_item_sk
TE5: SELECT ss_ticket_number, ws_order_number FROM web_sales, store_sales

Figure 6: An example of DO-PF-join

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b_1^1$</td>
<td>$b_1^2$</td>
</tr>
<tr>
<td>1.2</td>
<td>3.4</td>
</tr>
</tbody>
</table>

PDS($T_2$)

<table>
<thead>
<tr>
<th>Domain($b_2^1$) = [1, 2]</th>
<th>Domain($b_2^2$) = [3, 4]</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c(b_2^1) = 4$</td>
<td>$c(b_2^2) = 4$</td>
</tr>
<tr>
<td>$\delta(b_2^1) = 7$</td>
<td>$\delta(b_2^2) = 6$</td>
</tr>
<tr>
<td>$3, 3, 4, 4, 1, 1$</td>
<td>$2, 1, 1, 2, 1, 1$</td>
</tr>
</tbody>
</table>

OblivSortMerge($b_1^1, b_1^2$) : OblivSortMerge($b_2^1, b_2^2$)

Figure 7: An example of 2D DO-select-join
We evaluate HTree-based \( PDS \) with the default synthetic dataset of 10^6 records, by setting one of the attributes as the key attribute \( a_k \) and other attributes as a value. We first show the positive effect of bucketization on storage in Fig. 8a. As a competitor, we used a Bins-Only method, which uses the unit-length intervals of \( H(T) \) directly as buckets. With bucketization, the storage overhead becomes smaller than 1.7 for \( \epsilon' \geq 0.1 \), while Bins-Only has a storage overhead larger than 15. Given a predefined number of buckets \( B \), we empirically analyze the best division of \( (\epsilon', \delta') \) to \( (\epsilon_h, \delta_h) \) and \( (\epsilon_b, \delta_b) \); let \( \epsilon_h = f \epsilon' \) and \( \delta_h = f \delta' \) \( (f < 1) \). Fig. 8b shows the runtime of \( CreatePDS \) and the storage overhead of the resulting \( PDS \) against \( f \). We choose \( f = 0.2 \) in our implementation to minimize the runtime while retaining a reasonable storage overhead. The intuition of unequal budget division is that \( (\epsilon_h, \delta_h) \) is simply used to generate noisy bin counts to merge bins to equal-size buckets, while \( (\epsilon_b, \delta_b) \) directly affects the amount of noise added to each bucket. Fig. 8c shows the impact of the privacy budget \( (\epsilon', \delta') \) on the runtime of \( CreatePDS \) and on the storage overhead. Both of them decrease as \( \epsilon' \) increases, i.e., the privacy leakage becomes larger. We evaluate the construction time and the storage overhead of multidimensional \( PDS \) (PrivTree-based) with the same default synthetic dataset, see Fig. 8d. The time and the storage overhead increase sharply and eventually stabilize as the number of dimensions increases. PrivTree is a data-adaptive approach, so the number of buckets it tends to create for a fixed number of records has a saturated value given a specific privacy budget. Both storage and construction time are determined by the number of buckets.

6.2 Performance of DO Query Processing

All reported query runtimes are the time intervals from the query transmission until the receipt of the query result at the client side, unless otherwise stated.

Select. We evaluate DO-select with 1D range queries. We first test the number of buckets \( B = cN/U' \) affects the storage and communication overhead for different values of \( c \) (Fig. 9a). The communication overhead is measured for a query workload with selectivity 1%. The point lying closer to the origin of the axes corresponds to the \( B \) value that offers the better trade-off between storage and communication. We implement DO-select with \( B = 0.06N/U' \). We compare the runtime performance of DO-select with Adore, Opaque, and Epolute in Fig. 9b. We assume that the supported data structures for selection queries already exist (i.e., \( PDS \), \( DO-select \) and ORAM of Epolute have been built before the first query). DO-select has the best runtime performance. Even though Adore has a stronger assumption of a perfectly secure memory, DO-select still runs at least 3X faster, because Adore needs to do a full scan of the table while DO-select retrieves only a subset of \( PDS \) buckets through indices. The performance of Shrinking is dominated by the oblivious sorting in secure MPC, which requires over 725s to sort 10^6 records. Fig. 9c shows the communication efficiency of DO-select against various query selectivities. The smaller the query...
range, the higher the communication overhead, since all the records in a bucket are retrieved in order to answer a range that only needs a portion of them. As the query range increases, the communication volume gets closer to the bound $O(r + U')$ and the communication overhead approaches to 1.

**Join.** We implement DO-join with $B = 0.06hN/U$. The constant 0.06 is again experimentally verified to offer the best trade-off among runtime, storage, and communication efficiency. To put the performance in perspective, we compare DO-join with Adore, ODBJ, and CZSC21 in Fig. 10. $T_2$ is used to denote the larger input table, and the join output size of each test case is $|T_2|$.

Fig. 10a plots the runtime of foreign key join algorithms. CZSC21 is not designed for foreign key join so its performance is expected to be the worst. ODBJ and Adore have comparable performance since they are essentially sort-merge algorithms. Compared to ODBJ and Adore, DO-PF-join has a noticeable performance improvement. When $|T_2| = 5 \times 10^8$, DO-PF-join takes 388s to complete, which is 95s faster than ODBJ (483s). The performance gap continues to increase as the input size increases. In terms of communication overhead, ODBJ returns exactly $|T_2|$ number of records to the client, i.e., its communication efficiency is optimal, while the communication overhead of DO-PF-join is roughly 1.2. This shows that applying DO partitioning before sort-merge indeed alleviates the negative effect of large oblivious sorts at the cost of small storage and communication overhead.

For many-to-many joins, even though CZSC21 has the best theoretical runtime complexity, it is not practically competitive with the other two methods. We experiment on two test cases: $|T_2| = |T_1|$ and $|T_2| = 100|T_1|$. When joining a small table with a large table (e.g., $|T_1| = 100|T_1|$, as in Fig. 10b), Uni-DO-join can achieve an order of magnitude performance improvement than ODBJ. When $|T_2| = |T_1|$ as depicted in Fig. 10c, the superiority of DO-join is not as large as joining a small table with a large table, but is still significant: we observe about 3.5X speedup for Uni-DO-join over ODBJ. For 2 million records, Uni-DO-join takes 780s to run and ODBJ takes 2624s. In both test cases, Uni-DO-join achieves approximately 2X better performance than Ind-DO-join, as it needs to perform fewer bucket-wise joins (see Sec. 5.2.2). The performance of Shrinkwrap is dominated by obliviously sorting the Cartesian product join output of the two input tables. For the test case $|T_1| = 10^3$ and $|T_2| = 10^5$ that produces the smallest Cartesian product join output (i.e., $10^8$) in Fig. 10. Shrinkwrap takes more than 3 hours to complete.

We also evaluate the join methods on datasets TPC-H and TPC-DS (100MB, SF=0.1) with the most efficient but not oblivious hash join algorithm; see Fig. 11 for query runtime and Table 3 for communication overhead. Our Uni-DO-join has a 27X-480X runtime overhead compared to the non-oblivious hash join. CZSC21 is the most time-consuming method (except for TE3), but it is the most communication efficient for TE1-6 as shown in Table 3. Uni-DO-join is up to 7X faster than the second-best performing DO join algorithm (ODBJ), and its communication overhead is meanwhile as low as that of CZSC21. Overall, our DO-join outperforms the other two differentially oblivious join algorithms in many-to-many joins.

**Select-Join.** Fig. 10d shows the runtime of DO select-join in comparison with a Baseline, which first computes the complete set of join matches using Uni-DO-join and then post-processes the selection predicates on the complete set of join matches. The technique used by Baseline for post-processing is similar to Opaque’s filtering algorithm: first scan and mark the qualified records with “0” and other records with “1”, then remove the “1” records through oblivious compaction. DO-select-join in Fig. 10d, which applies selection predicates on the join attribute before computing join matches, is much more efficient than Baseline for very selective queries. Its
cost eventually converges to that of Baseline when the selectivity is 100% and all PDS buckets are qualified. DO-select-njoin shows the performance of applying selection predicates on a non-join attribute before computing join matches. This method is less efficient than DO-select-join, because its private data structures do not share the same bucket structure: for each bucket in $PDS(T_2, (a_s, f_s))$, its domain of the join attribute $a_s$ may overlap with several buckets in $PDS(T_1, a_s)$. Still, DO-select-njoin has a better performance than Baseline for highly selective queries. The runtime improvement of DO-select-join and DO-select-njoin over Baseline indicates the effectiveness of our partitioning technique that supports differentially oblivious selection.

### Table 3: TE1-6 communication overhead

<table>
<thead>
<tr>
<th></th>
<th>TE1</th>
<th>TE2</th>
<th>TE3</th>
<th>TE4</th>
<th>TE5</th>
<th>TE6</th>
</tr>
</thead>
<tbody>
<tr>
<td>DO-join</td>
<td>1.68</td>
<td>2.3</td>
<td>1.05</td>
<td>1.36</td>
<td>1.06</td>
<td>1.13</td>
</tr>
<tr>
<td>ODBJ</td>
<td>2.04</td>
<td>2.28</td>
<td>1.07</td>
<td>9.31</td>
<td>1.8</td>
<td>9.33</td>
</tr>
<tr>
<td>CZSC21</td>
<td>1.68</td>
<td>2.22</td>
<td>1.05</td>
<td>1.04</td>
<td>1.03</td>
<td>1.03</td>
</tr>
</tbody>
</table>

7 RELATED WORK

**DP Histograms** There is a batch of algorithms to publish DP histograms [27, 32, 38, 41, 55]. For example, Hay et al. [27] introduce a hierarchical method for optimizing 1D DP histograms, and constrained inference techniques (CI) to significantly reduce the Mean Squared Error of answering all range queries over the data domain. Qardaji et al. [41] perform an in-depth analysis of the hierarchical method. They extend the analysis to multiple dimensions, but find that the benefit of using hierarchies is limited. PrivTree [55], on the other hand, works well for multidimensional selections. It adaptively decomposes the data universe (root) into disjoint sub-domains (leaves), based on the distribution of data.

**Oblivious Selection** The design of oblivious selection has been studied in several privacy-oriented databases/frameworks. Opaque [56] filters out records through oblivious sorting. ObliDB [21] and Adore [42] select qualifying records through one or more scans of the input. Opaque and ObliDB define oblivious access patterns w.r.t. input and output sizes. We can opt for full or DP padding modes to hide the output size completely or differentially privately, which satisfy FO or DO, respectively. Adore uses the same privacy notion of Definition 5. Epsolute [8] uses ORAM to hide the pattern of retrieving qualifying records, and uses DP to hide its output size.

**Oblivious Join** Opaque [56], ObliDB [21], and Adore [42] propose sort-merge algorithms for foreign key joins. Krastnikov et al. [34] introduce ODBJ, a general join algorithm oblivious w.r.t to input and output sizes. Chang et al. [12] propose general ORAM-based algorithms for binary and multiway equi-joins.

Chu et al. pioneer a $(\epsilon, \delta)$-DO many-to-many join algorithm, denoted as CZSC21 [13]. The algorithm has near-optimal theoretical runtime $O(N \log N + r + NU)$, which is based on a $O(N \log N)$ oblivious sorter [44] and $O(N)$ oblivious compaction [4]. CZSC21 has a mirrored overall structure of DO-join: it first differentially obliviously groups the two join tables by join keys, then does a Cartesian product join for each pair of buckets associated with the same set of keys, and lastly performs a DO compaction. However, their grouping technique makes use of true key multiplicities. As revealing the number of buckets created by grouping violates DP, they need to hide this number by padding it to a user-defined upper bound $CN/U$ for a sufficiently large constant (e.g., $C = 1/2$). Predicting an accurate upper bound for the number of buckets is not easy, and a safe but not accurate guess significantly amplifies the amount of noise, negatively affecting the runtime performance.

**Oblivious Database Systems** A number of works have focused on concealing the access patterns of data processing. Arasu et al. [2] propose specialized oblivious query processing algorithms for selection, join, grouping and aggregation. Shrinkwrap [6] is designed for federated and distributed oblivious query processing. In addition to specialized approaches for different operators, there is also work that uses ORAM to hide general access patterns. ZeroTrace [46] presents efficient oblivious memory primitives based on ORAM with the support of Intel SGX. Oblix [40] constructs an oblivious search index. Obladi [15] is the first to support ACID transactions while also hiding access patterns. Inspired by ZeroTrace, POSUP [28] employs ORAM to enable oblivious keyword search and update operations. In a distributed setting, VC3 [48] and M2R [16] implement distributed secure MapReduce computations, protecting data integrity and obliviousness.

8 CONCLUSION

In this paper, we explore the new privacy notion of differential obliviousness (DO) and its application on outsourced database systems supported by TEE. Our work is among the very first towards building DO outsourced database systems. We integrate DO with private data structures, and present an approach for building DO indices that speed up select and select-join queries. We also present the first practical partition-based oblivious join. Our DO-join algorithms are efficient for both foreign key and many-to-many joins. Finally, we show the potential of the DO privacy notion in the field of databases through an extensive experimental evaluation.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. Lina Qiu and George Kollios were supported by NSF CNS-2001075 Award. Kobbi Nissim was supported by NSF Grant No. 2001041, “Rethinking Access Pattern Privacy: From Theory to Practice”. Nikos Mamoulis was supported by the Hellenic Foundation for Research and Innovation (HFR) under the “2nd Call for HFRI Research Projects to support Faculty Members Researchers” (Project No. 2757).


