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

ABSTRACT

Most cloud service providers offer limited data privacy guarantees, discouraging clients from using them for managing their sensitive data. Cloud providers may use servers with Trusted Execution Environments (TEEs) to protect outsourced data, while supporting remote querying. However, TEEs may leak access patterns and allow communication volume attacks, enabling an honest-but-curious cloud provider to learn sensitive information. Oblivious algorithms can be used to completely hide data access patterns, but their high overhead could render them impractical. To alleviate the latter, the notion of Differential Obliviousness (DO) has been recently proposed. DO applies differential privacy (DP) on access patterns while hiding the communication volume of intermediate and final results; it does so by trading some level of privacy for efficiency.

We present Doquet: **D**ifferentially **O**blivious Range and Join **Que**ries with Private Data Struc**t**ures, a framework for DO outsourced database systems. Doquet is the first approach that supports private data structures, indices, selection, foreign key join, many-to-many join, and their composition select-join in a *realistic* TEE setting, even when the accesses to the private memory can be eavesdropped on by the adversary. We prove that the algorithms in Doquet satisfy differential obliviousness. Furthermore, we implemented Doquet and tested it on a machine having a second generation of Intel SGX (TEE); the results show that Doquet offers up to an order of magnitude speedup in comparison with other fully oblivious and differentially oblivious approaches.

PVLDB Reference Format:

Lina Qiu, Georgios Kellaris, Nikos Mamoulis, Kobbi Nissim, and George Kollios. Doquet: Differentially Oblivious Range and Join Queries with Private Data Structures. PVLDB, 16(13): 4160 - 4173, 2023.
doi:10.14778/3625054.3625055

PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at https://github.com/linaqiu22/DOQP.

Proceedings of the VLDB Endowment, Vol. 16, No. 13 ISSN 2150-8097.
doi:10.14778/3625054.3625055

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 has accessed [10, 35, 49, 54], 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 (ϵ, δ)-differential obliviousness (DO) [11], which requires that the memory access patterns of a program and the lengths of intermediate results satisfy (ϵ, δ)-differential privacy (DP). If a query processing algorithm is

(ϵ, δ)-DO and the sensitive data are encrypted in all states (data-in-use, data-at-rest, and data-in-transit) on the server, the information revealed to honest-but-curious adversaries satisfies (ϵ, δ)-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 $\mathcal{D}$ consists of a collection of tables. Each table T in $\mathcal{D}$ can be abstracted as an array of n records $T = \{r_1, \ldots, r_n\}$. Each record r_j is in the form of $(rid_j, \mathbf{v}_j)$, where rid_j is the unique identifier of the record, and $\mathbf{v}_j = \{v_j(a_1), \ldots, v_j(a_m)\}$ is a list of values corresponding to the attributes $\{a_1, \ldots, a_m\}$ that T is defined on. We use $Domain(a_s)$ to denote the domain of attribute a_s, which is a superset of distinct values of a_s in table T.

2.1 Differential Privacy

Two databases $\mathcal{D}_1, \mathcal{D}_2$ over domain $\mathcal{X}$ are called *neighboring* (denoted by $\mathcal{D}_1 \sim \mathcal{D}_2$) if one of them can be obtained from the other through one insertion or one substitution of an element of $\mathcal{X}$.

Definition 1. *Let X, Y be two random variables over the same support Ω. We write $X \underset{\epsilon,\delta}{\approx} Y$ if for all $S \subseteq \Omega$,*

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

Definition 2 ((ϵ, δ)-DP [17, 18]). *Let $\epsilon > 0$ and $\delta \in [0, 1)$. A randomized mechanism $\mathcal{M}$ is (ϵ, δ)-differentially private, (ϵ, δ)-DP for short, if $\mathcal{M}(\mathcal{D}_1) \underset{\epsilon,\delta}{\approx} \mathcal{M}(\mathcal{D}_2)$ for all $\mathcal{D}_1 \sim \mathcal{D}_2$.*

Definition 3 (Sensitivity [18]). *For a query q that maps a dataset to a vector of real numbers, the global sensitivity Δ_q of q and the local sensitivity $\Delta_q(\mathcal{D}_1)$ of q at database $\mathcal{D}_1$ are as follows:*

$$\Delta_q = \max_{\mathcal{D}_1 \sim \mathcal{D}_2} \|q(\mathcal{D}_1) - q(\mathcal{D}_2)\|_1$$

$$\Delta_q(\mathcal{D}_1) = \max_{\mathcal{D}_2 : \mathcal{D}_1 \sim \mathcal{D}_2} \|q(\mathcal{D}_1) - q(\mathcal{D}_2)\|_1$$

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

$$\mathcal{G}(\epsilon, \delta, \Delta, p) = \min\{\max\{p - \frac{U}{2}, p + Geom(e^{\frac{\epsilon}{\Delta}})\}, p + \frac{U}{2}\}$$

The mechanism that on input $\mathcal{D}$ outputs $y = q(\mathcal{D}) + \mathcal{G}(\epsilon, \delta, \Delta_q, p)$ is (ϵ, δ)-differentially private.

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

Proposition 1 (Post-processing [19]). *If a randomized mechanism $\mathcal{M} : \mathcal{X} \to R$ is (ϵ, δ)-DP. Let $f : R \to R'$ be an arbitrary mapping. We have $f \circ \mathcal{M} : \mathcal{X} \to R'$ also satisfies (ϵ, δ)-DP.*

Theorem 1 (Composition [19, 20]). *Suppose there is a set of randomized mechanisms $\mathcal{M} = \{\mathcal{M}_1, \ldots, \mathcal{M}_m\}$, each $\mathcal{M}_i$ is (ϵ_i, δ_i)-DP. If every $\mathcal{M}_i$ in the set is performed on a disjoint subset of the*

Table 1: Notation table

Query	
N	The number of input records
r	The actual output size
R	The noisy output size (communication volume)
$r(q)$	The actual response of query q
$R(q)$	The noisy response of q including dummy records
Differential Privacy	
(ϵ, δ)	The total privacy budget
(ϵ', δ')	The privacy budget to create *PDS*
	$U = \max \mathcal{G}(\epsilon, \delta)$; $U' = \max \mathcal{G}(\epsilon', \delta')$
(ϵ_h, δ_h)	The privacy budget to create DP histogram
(ϵ_b, δ_b)	The privacy budget for bucketization
(ϵ_c, δ_c)	The privacy budget for join output compaction
$\Delta_{\text{algo}}(T)$	The local sensitivity of the algorithm w.r.t. input T
	Privacy budgets subject to: $(\epsilon', \delta') = (\epsilon_h, \delta_h) + (\epsilon_b, \delta_b)$ and $(\epsilon, \delta) = (\epsilon', \delta') + (\epsilon_c, \delta_c)$
HTree	
h	The tree height
k_b	The tree branching factor
$\widehat{c}(v), c(v), \widehat{n}(v)$	The noisy count (capacity), the real count and the number of dummy records added to tree node/interval v, subject to $\widehat{c}(v) = c(v) + \widehat{n}(v)$
PDS	
B	The number of buckets in *PDS*
θ	The threshold used to decide bucket domains
$Domain(a_s)$	The domain of attribute a_s, $[x_1, \cdots, x_D]$, with size D
$\widehat{c}(b_i), c(b_i), \widehat{n}(b_i)$	The noisy count (capacity), the real count and the number of dummy records added to bucket b_i, subject to $\widehat{c}(b_i) = c(b_i) + \widehat{n}(b_i)$
$\mathcal{L}$(domains)	A list $\{Domain(b_i),$ for all buckets $b_i\}$
$\mathcal{L}$(capacities)	A list $\{\widehat{c}(b_i),$ for all buckets $b_i\}$

entire dataset, $\mathcal{M}$ provides $(\max(\epsilon_1, \ldots, \epsilon_m), \max(\delta_1, \ldots, \delta_m))$-DP. If $\mathcal{M}$ is performed sequentially on the same dataset, $\mathcal{M}$ provides $(\sum_{i=1}^{m} \epsilon_i, \sum_{i=1}^{m} \delta_i)$-DP.

2.2 Differential Obliviousness

Definition 5 ((ϵ, δ)-DO [11]). *A randomized mechanism $\mathcal{M}$ is (ϵ, δ)-differentially oblivious, or (ϵ, δ)-DO for short, if $View^{\mathcal{M}}(\mathcal{D}_1) \underset{\epsilon,\delta}{\approx} View^{\mathcal{M}}(\mathcal{D}_2)$ for all $\mathcal{D}_1 \sim \mathcal{D}_2$, where $View^{\mathcal{M}}(\mathcal{D}_1)$ is the sequence of memory addresses (access pattern) generated by the random execution of $\mathcal{M}$ on input $\mathcal{D}_1$.*

Definition 6 (δ-implementation [11]). *A mechanism $\mathcal{M}$ δ-obliviously implements a functionality $\mathcal{F}$ with leakage $\mathcal{L}$ if there exists a simulator SIM that produces a simulated access pattern, such that for any security parameter λ and any input $\mathcal{D}$, the executions*

Ideal: Let $O_{ideal} \leftarrow \mathcal{F}(\lambda, \mathcal{D}, \rho)$ and $Leak_{ideal} \leftarrow \mathcal{L}(\lambda, \mathcal{D}, \rho)$, where ρ is the random bits needed by $\mathcal{F}$.

Real: Let $(O_{real}, Leak_{real}, View) \leftarrow \mathcal{M}(\lambda, \mathcal{D})$, where View is the access pattern of $\mathcal{M}$

have $\delta(\lambda)$-statistically close distributions:

$$(O_{ideal}, Leak_{ideal}, SIM(\lambda, Leak_{ideal})) \overset{\delta(\lambda)}{\equiv} (O_{real}, Leak_{real}, View)$$

Lemma 1 ([13]). *If a mechanism $\mathcal{M}$ δ'-obliviously implements a functionality $\mathcal{F}$ with leakage $\mathcal{L}$, where $\mathcal{L}$ is (ϵ, δ)-DP with respect to the input, then $\mathcal{M}$ is $(\epsilon, \delta + \delta')$-DO.*

2.3 Fully Oblivious Building Blocks

We review three FO building blocks used throughout Doquet.

2.3.1 ObliSort. In our work, we use bitonic sorter [5], which is one of the most used algorithms, having $O(N \log^2 N)$ time complexity. Although there exist alternative $O(N \log N)$-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 sorter given a reasonable input size [1, 3]. ObliSort(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 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 $f : X \longrightarrow [n]$, 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 $(f(x_i), f(x_{i+1}))$ 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 (ϵ, δ).[1]

Algorithm 1: HTree$(C, k_b, (\epsilon_h, \delta_h))$

Initialize $L_1 = C, h = \lceil \log_{k_b} D \rceil + 1$;
for $l \in [2, h]$ **do**
  foreach *node/interval* $v \in L_{l-1}$ **do**
    Independently sample noise $\widehat{n}(v) = \mathcal{G}(\epsilon_h, \delta_h, h, 0)$;
    Augment the current entry $c(v)$ of L_{l-1} to be $(c(v), \widehat{n}(v))$;
  end
  Let $s = \lceil |L_{l-1}| / k_b \rceil$;
  Create a new array $L_l = \{c(v_0), \cdots, c(v_{s-1})\}$, such that $c(v_i) = \sum_{j=i \cdot k_b}^{(i+1) \cdot k_b - 1} L_{l-1}[j].c(v)$;
end
Apply Constrained Inference (CI);

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_b children corresponding to k_b equally sized subintervals. Suppose that *Tree* is built for attribute a_s of table T and its domain is $Domain(a_s) = [x_1, x_D]$, the unit-length intervals $[x_1], \cdots, [x_D]$ are the leaves and the root is the entire domain. The privacy budget (ϵ_h, δ_h) 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. $C = \{c(x_1), \cdots, c(x_D)\}$ is the real counts for unit-length intervals $[x_1], \cdots, [x_D]$.

Because the noise of each node $\widehat{n}(v)$ 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 $H(T) = \{(c(v), \widehat{n}(v)) : \text{for every unit-length interval } v \in [x_1, x_D]\}$. We define the leakage of HTree as $\mathcal{L}(\text{HTree}) = \{\widehat{c}(v) = c(v) + \widehat{n}(v) : \text{for } v \in [x_1, x_D]\}$, since $c(v)$ and $\widehat{n}(v)$ are private and will not be used directly. The leakage is (ϵ_h, δ_h)-DP [27].

Proposition 2 ([27]). *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 $O(h^2 U)$.*

3 PROBLEM DEFINITION

In this section, we present the DO database outsourcing model, adapted from [30] whereon 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

[1]The 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.

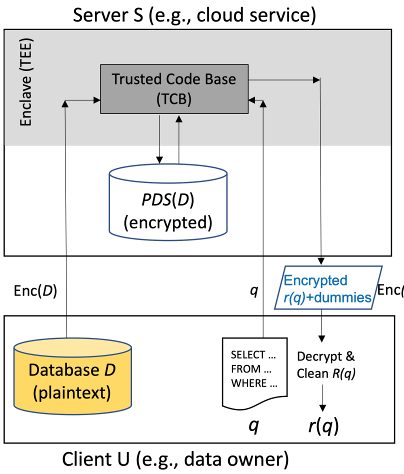


Figure 1: Overview of DO database outsourcing

Algorithm 2: Histogram$(T, a_s, k_b, (\epsilon_h, \delta_h))$

ObliSort(T, a_s);
In the sorted table T, the last element of each distinct value x_j of the attribute a_s is the representative of x_j;
Initialize *counter* = 1, array C, array A and array F;
For each element in T, if it is a representative, let $c(x_j)$ = *counter* and reset *counter* = 1. Insert $(x_j, c(x_j))$ to A, where $c(x_j)$ is the number of x_j in T. Otherwise, insert a dummy entry $(\bot, \bot)$ to A and increase *counter* by 1. If $|T| < D$, continue adding dummy entry $(\bot, \bot)$ to A until $|A| = D$;
A = ObliCompact(A, D);
for $i \in [1, D]$ **do**
  if $A[i]$ *is* $(\bot, \bot)$ **then**
    Insert $\bot$ to array F;
  else
    Insert $A[i][0] - x_1 + 1$ to F;
  end
end
C = ObliExpand(A, F, D);
In one linear scan of C, replace entry $(x_j, c(x_j))$ with $c(x_j)$, or replace dummy entry $(\bot, \bot)$ at position j with 0 to represent that the key $x_j = x_1 + j - 1$ has no elements in T;
HTree$(C, k_b, (\epsilon_h, \delta_h))$;

(server) $\mathcal{S}$. Client $\mathcal{U}$ wants to outsource data management to $\mathcal{S}$ but also requires that the information leakage to unauthorized parties (e.g., server $\mathcal{S}$) is bounded by (ϵ, δ)-DP. The server is divided into trusted (grey-shaded) and untrusted parts. Inside the enclave (TEE), there is a trusted code base (TCB) containing DO algorithms, in addition to the encryption keys used to decrypt/encrypt data swapped in/out of the enclave. The majority of outsourced data and its indices are encrypted and stored in untrusted storage. All traffic between trusted and untrusted parts is monitored by the enclave. Database outsourcing is performed in two phases as follows:

Setup Phase: Client $\mathcal{U}$ ships encrypted database $Enc(\mathcal{D})$ to server $\mathcal{S}$. Given $Enc(\mathcal{D})$, $\mathcal{S}$ computes a set of private data structures $PDS(\mathcal{D})$ for the database in the enclave and then stores them.

Query Phase: Client $\mathcal{U}$ submits queries to server $\mathcal{S}$. To process a query q, $\mathcal{S}$ may load some encrypted data into its TEE memory, where they are decrypted to obtain a response $R(q)$ based on $PDS(\mathcal{D})$; the enclave then encrypts the response and $Enc(R(q))$ is sent to $\mathcal{U}$. The response $R(q)$ includes the records $r(q)$ which are selected by the query and possibly some dummy records. As such, $r(q) \subseteq R(q)$. Dummy records are indistinguishable from real records by the untrusted part of $\mathcal{S}$ after encryption. To complete query processing, $\mathcal{U}$ decrypts $Enc(R(q))$ and removes the dummy records to get the correct answer $r(q)$.

The above model only considers static data: there are no updates to $\mathcal{D}$ after the initial setup, and no updates to $PDS(\mathcal{D})$ are made by queries. We leave the management of updates as future work.

3.2 Threat Model and Security Requirements

We model the server $\mathcal{S}$ to be an honest-but-curious adversary, i.e., $\mathcal{S}$ 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 $\mathcal{S}$. $\mathcal{S}$ 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 receieved from and sent to the client $\mathcal{U}$ 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 $Q = \{q_1, q_2, \dots\}$ submitted by $\mathcal{U}$, we define the server view $\text{View}_Q(\mathcal{D})$ to be the random variable corresponding to the sequences of memory accesses and messages between $\mathcal{U}$ and the TEE during the execution of Q on database $\mathcal{D}$. We require that $\text{View}_Q(\mathcal{D}_1) \underset{\epsilon,\delta}{\approx} \text{View}_Q(\mathcal{D}_2)$ for all Q and all $\mathcal{D}_1 \sim \mathcal{D}_2$.[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 inital 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 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. CreatePDS consists of two steps: 1) Histogram obliviously computes a DP histogram and 2) 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 slighlty 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 $q_1, \dots, q_{i-1}$.

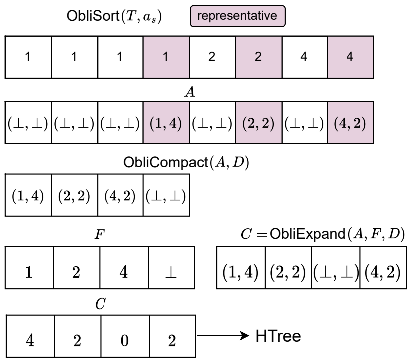


Figure 2: An example of Histogram with domain $[1,4]$

4.1.1 Histogram. Algorithm 2 takes as input a table T, an attribute a_s of T, a tree branching factor k_b, and a privacy budget (ϵ_h, δ_h). It first obliviously computes the multiplicity of each distinct value x_j (representative) of a_s in T (lines 1-4). Then, it creates an array A that stores for each x_j its multiplicity. In particular, 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 $Domain(a_s)$. 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 $C = \{c(x_1), \cdots, c(x_D)\}$ of HTree, i.e., an array of exact counts for each value in the domain. Specifically, 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 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 $Domain(a_s) = [1,4]$. The representatives are highlighted in the figure.

4.1.2 CreateBuckets. A bucket b in $PDS(T, a_s)$ corresponds to an interval $Domain(b) \subseteq Domain(a_s)$. 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 (ϵ_h, δ_h)-DP histogram (i.e. the HTree), the number of buckets B, and a privacy budget (ϵ_b, δ_b) as input. The average number of records per bucket is $\theta = (\sum H(T))/B$, where $\sum H(T) = \sum_{v=x_1}^{v=x_D} \widehat{c}(v)$ is the sum of noisy counts of HTree leaves. Notice that the algorithm can run at different levels of the tree, not only the leaf level. If the cumulative noisy count $\sum_{v=x_s}^{v=x_e} \widehat{c}(v) < \theta$, we keep merging the next interval to the current bucket b_i (lines 2-9). Otherwise, the bucket's domain is chosen, and we keep both the real and noisy count $c(b_i)$ and $\widehat{c}(b_i)$ along with the domain. The capacity of the resulting private data structure is $|PDS(T)| = \sum \widehat{c}(b_i)$. 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 $\sum \widehat{c}(b_i)$. 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 $PDS(T) = \{b_i, \text{ for all buckets}\}$.

Algorithm 3: CreateBuckets$(T, H(T), B, (\epsilon_b, \delta_b))$

1 Initialize $\theta = (\sum H(T))/B, s = 1, e = 1, i = 1$;
2 **while** $e \leq D$ **do**
3   **if** $\sum_{v=x_s}^{v=x_e} \widehat{c}(v) \geq \theta$ **then**
4   Create bucket $b_i = (Domain(b_i), c(b_i), \widehat{c}(b_i))$, where $Domain(b_i) = [x_s, x_e], c(b_i) = \sum_{v=x_s}^{v=x_e} c(v), \widehat{c}(b_i) = c(b_i) + \widehat{n}(b_i), \widehat{n}(b_i) = \mathcal{G}(\epsilon_b, \delta_b)$;
5     $s = e + 1, e = e + 1, i = i + 1$;
6   **else**
7     $e = e + 1$;
8   **end**
9 **end**
10 Compute an array A of length B such that $A[i] = A[i-1] + \widehat{c}(b_{i-1})$ for $i \in [2, B]$. $A[1] = 1$;
11 Compute an array F of length B such that $F[i] = F[i-1] + c(b_{i-1})$ for $i \in [2, B]$. $F[1] = 1$;
12 $V =$ ObliExpand$(A, F, |T|)$; In one forward scan of V, fill dummy position $V[j] = V[j-1] + 1$;
13 ObliExpand$(T, V, \sum \widehat{c}(b_i))$;
14 Let $prefix = 1$;
15 **for** $i \in [1, B]$ **do**
16   Insert $T[prefix], \cdots, T[prefix + \widehat{c}(b_i) - 1]$ to bucket b_i;
17   $prefix = prefix + \widehat{c}(b_i)$
18 **end**

Algorithm 4: CreatePDS (T, a_s)

$H(T) \leftarrow$ Histogram$(T, a_s, k_b, (\epsilon_h, \delta_h))$;
$PDS(T, a_s) \leftarrow$ CreateBuckets$(T, H(T), B, (\epsilon_b, \delta_b))$;

4.1.3 CreatePDS. Algorithm 4 summarizes the entire CreatePDS process for $PDS(T, a_s)$. Parameters $\{k_b, (\epsilon_h, \delta_h), (\epsilon_b, \delta_b), B\}$ are omitted for readability. Suppose that the privacy budget follows $\epsilon' = \epsilon_h + \epsilon_b$ and $\delta' = \delta_h + \delta_b$, CreatePDS is (ϵ', δ')-DO. Fig. 3 gives an example flow of CreatePDS. Assume that $Domain(a_s)$ is $[x_1 = 1, x_4 = 4]$, $H(T)$ produced by Histogram has four bins consisting of the real count and added noise. If the number of buckets $B = 2$, we have $\sum H(T) = 10$ and the threshold $\theta = 5$. In CreateBuckets, the domains of the two buckets $Domain(b_1) = [1,1]$ and $Domain(b_2) = [2,4]$ are decided by the threshold θ and the noisy counts of histogram bins (lines 2-9 in Algo. 3). $V[i]$ stores the position that record $T[i]$ should be obliviously distributed to in the expanded table $PDS(T)$ (line 12 in Algo. 3). Lastly, records in T are moved to corresponding positions via ObliExpand.

4.1.4 Proof of Differential Obliviousness. We will use the following leakage functions throughout our analysis: $\mathcal{L}$(HTree) is the leakage produced by the call to procedure HTree in line 15 of Algo. 2. Let $\mathcal{L}$(domains) and $\mathcal{L}$(capacities) denote the collection of bucket domains and the bucket capacities created in Algo. 3, respectively. Our proof uses DP composition Theorem 1 and Lemma 1 to reason about the composition of DO mechanisms. The new and more intuitive DO composition theorem [57] is an alternative to achieve the same conclusion.

Histogram (Algo. 2): Note that the execution of lines 1-14 where the array C is computed is FO and leaks nothing other than the public information: input table size $|T|$ and the domain size D of a_s. This is due to the use of ObliSort, ObliCompact, ObliExpand, and HTree and the fact that memory accesses during the for loops in lines 4 and 6-12 as well as the linear scan in line 14 are independent of the input. Finally, neighboring inputs result in neighboring arrays C (each insertion or removal of a record increases or decreases a

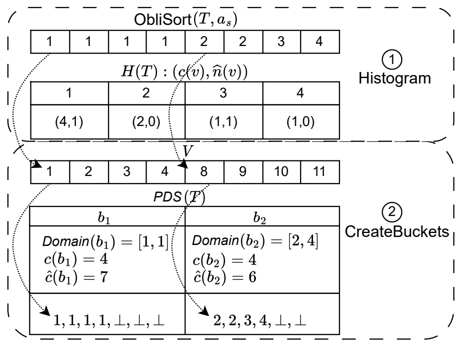


Figure 3: Overall flow of CreatePDS

single entry of the array C by 1). Hence, the application of HTree (i.e., $\mathcal{L}$(HTree)) preserves (ϵ_h, δ_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 0-oblivious with leakage $\mathcal{L}$(HTree), which is (ϵ_h, δ_h)-DP. Hence, by Lemma 1, the merging procedure is (ϵ_h, δ_h)-DO. The sensitivity of the real counts of buckets $\{c(b_i),$ for all buckets $b_i\}$ is 1, hence $\mathcal{L}$(capacities) $= \{\widehat{c}(b_i),$ for all $b_i\}$ is (ϵ_b, δ_b)-DP given that $\widehat{c}(b_i) = c(b_i) + \mathcal{G}(\epsilon_b, \delta_b)$. The access pattern of moving data to buckets (lines 10-18 in Algo. 3) is oblivious w.r.t. $\mathcal{L}$(capacities), so it is (ϵ_b, δ_b)-DO by Lemma 1 again. Based on Theorem 1, we hence get CreateBuckets is (ϵ', δ')-DO where $\epsilon' = \epsilon_h + \epsilon_b$ and $\delta' = \delta_h + \delta_b$. Note that the bucket domains are constructed from $\mathcal{L}$(HTree) only, so $\mathcal{L}$(domains) is a post-processing of $\mathcal{L}$(HTree) and also (ϵ_h, δ_h)-DP by Proposition 1.

CreatePDS (Algo. 4) is (ϵ', δ')-DO given that Histogram is FO and CreateBuckets is (ϵ', δ')-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 $\mathcal{L}$(PrivTree) be the leakage of PrivTree; $\mathcal{L}$(PrivTree) $= \{(Domain(b_i), \widehat{c}(b_i)),$ for all buckets $b_i\}$. Given privacy budget (ϵ', δ'), the leakage of PrivTree is (ϵ', δ')-DP [55]. If HTree is replaced by PrivTree in our framework, Histogram is oblivious w.r.t. $\mathcal{L}$(PrivTree). CreateBuckets, again, moves data to the corresponding buckets and pads each bucket to its capacity $\widehat{c}(b_i)$, which should also be oblivious w.r.t. $\mathcal{L}$(PrivTree). Overall, CreatePDS is (ϵ', δ')-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 (ϵ', δ')-DO. Let $Domain(b_i) = [x_s, x_e]$ 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 $\{(x_s, b_i)$: for all $b_i\}$, and the other for $\{(x_e, b_i)$: for all $b_i\}$. 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 $\mathcal{L}$(domains). In general, any index built at the granularity of buckets can be easily shown to satisfy (ϵ', δ')-DO.

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

Initialize $B = cN/U'$;
if *PDS*(T, a_s) *does not exist* **then**
  PDS$(T, a_s) \leftarrow$ CreatePDS(T, a_s);
end
Search *PDS*(T, a_s) using indices or through one scan, add all buckets whose domains overlap with $[q_s, q_e]$ to $R(q)$;

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: $R(q) = \{2, 2, 3, 4, \perp, \perp\}$.

Analysis We now assess the storage complexity and communication volume given the number of buckets $B = cN/U'$ (c is a constant and $U' = \max \mathcal{G}(\epsilon', \delta')$) chosen for selection queries. Each bucket contains approximately N/B real records and $\mathcal{G}(\epsilon_b, \delta_b) = O(U')$ 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.

	Private Memory	Degree of Obliviousness	Hide Communication Volume	Operators	Select Runtime	Join Runtime
Doquet	×	DO	✓	select, join, select-join,	$O(\log N + R)$	$\boldsymbol{O(N\log^2 N + (r+NU)\log(r+NU))}$
Adore [42]	✓	DO	✓	select, groupby, foreign key join	$O(N + R)$	$O(N\log N)$
CZSC21 [13]	×	DO	✓	join		$\boldsymbol{*O(N\log^2 N + (r+NU)\log(r+NU))}$
Opaque [56]	×	FO	∘, support DP padding	select, groupby, foreign key join	$*O(N\log N)$	$O(N\log^2 N)$
ObliDB [21]	✓	FO	∘, support DP padding	select, groupby, foreign key join, insert, update, delete	$O(N^2/S)$	$O(N\log^2 N)$
ODBJ [34]	×	FO	×	join		$\boldsymbol{*O((r+NU)\log^2(r+NU))}$
Shrinkwrap [6]	×	FO	∘, support DP padding	select, groupby, join	$O(N\log^2 N)$	$\boldsymbol{O(N^4\log^2 N^2)}$

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 $|PDS(T)| \le N + B * U' = O(N)$.

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.

$PDS(T), [q_s, q_e] = [3, 4]$

b_1 skipped	b_2 overlapped
$Domain(b_1) = [1, 1]$	$Domain(b_2) = [2, 4]$
$1, 1, 1, 1, \perp, \perp, \perp$	$2, 2, 3, 4, \perp, \perp$

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 $T_1.a_s = T_2.a_s$". 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 (ϵ, δ)-DO. Another side contribution of this work is a practical implementation of CZSC21 [13] with a runtime-efficient $O(N\log^2 N)$ bitonic sorter and $O(N\log N)$ oblivious compaction [47]. Both ODBJ and CZSC21 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 $R = r + \mathcal{G}(\epsilon, \delta, \max(|T_1|, |T_2|))$, 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 $U = \max \mathcal{G}(\epsilon, \delta)$, then $\mathcal{G}(\epsilon, \delta, \max(|T_1|, |T_2|))$ 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 partiton-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_s, and a privacy budget (ϵ_c, δ_c) for compaction. If we use Ind-DO-join, $PDS(T_1)$ and $PDS(T_2)$ are constructed independently: $Domain(T_1, b_i)$ may be different from $Domain(T_2, b_i)$, where $Domain(T_t, b_i)$ is the domain of the i^{th} bucket in $PDS(T_t)$ for $t \in [1, 2]$. For each bucket b_i^1 in $PDS(T_1)$, we need to check for a list of consequent buckets in $PDS(T_2)$ 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 concatination (T_1, T_2), so that the output $PDS(T_1)$ and $PDS(T_2)$ share the same bucket structure. Since $Domain(T_1, b_i)$ and $Domain(T_2, b_i)$ 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 CreateBuckets Algo. 3 with the following changes: 1) the threshold $\theta = (\sum H(T_1) + \sum H(T_2))/B$ in line 1; 2) the tuple count $\widehat{c}(v)$ in line 3 is replaced with $\widehat{c}_1(v) + \widehat{c}_2(v)$, where $\widehat{c}_t(v)$ is the noisy count of node v in $H(T_t)$; 3) if the condition in line 3 is true, two buckets b_i^1 and b_i^2 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_t, with b_i replaced by b_i^t.

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_1^2), (b_2^1, b_1^2) and (b_2^1, b_2^2). For Uni-DO-join, $\tilde{R}(q)$ contains 86 records produced by two bucket-wise Cartesian products (b_1^1, b_1^2) 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_1 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 $(T_1, T_2, a_s, (\epsilon_c, \delta_c))$

1 Initialize $B = chN/U, i = 1, j = 1$;
2 **if** *Ind-DO-join* **then**
3     $PDS(T_1) \leftarrow$ CreatePDS(T_1, a_s);
4     $PDS(T_2) \leftarrow$ CreatePDS(T_2, a_s);
5 **else**
    `// Uni-DO-join`
6     $PDS(T_1), PDS(T_2) \leftarrow$ CreatePDS$((T_1, T_2), a_s)$;
7 **end**
8 **foreach** *bucket* b_i^1 *in* $PDS(T_1)$ **do**
9   **while** b_j^2 *in* $PDS(T_2)$ *overlaps with* b_i^1 **do**
10   Compute the Cartesian product of b_i^1 and b_j^2;
11   Add the result to array $\tilde{R}(q)$;
12         $j = j + 1$;
13   **end**
14     $i = i + 1$;
15   **if** *Ind-DO-join* **then** $j = j - 1$;
16 **end**
17 $R(q) =$ ObliCompact$(\tilde{R}(q), R)$, where $R = r + \mathcal{G}(\epsilon_c, \delta_c, \Delta_{\text{join}}(T_1, T_2))$;

bucketization can be done before the execution of the join, since it is independent of the join query. Therefore, we avoid computing the bucketization on-the-fly with the trade-off to producing larger results for $|\tilde{R}(q)|$. In our evaluation, we assume that $PDS(T_1)$ and $PDS(T_2)$ are pre-computed in Ind-DO-join, but not in Uni-DO-join. In other words, the query runtime measurement for Ind-DO-join starts from the bucket-wise Cartesian product join.

When the output of join is the input of another query operator, shrinking the noisy join output with a (ϵ_c, δ_c)-DO compaction [13] will effectively accelerate the whole query execution. On the other hand, if join is the last operator before returning to the client, the server can simply send overlapped bucket-pairs back to the client. Upon receiving a pair, the client computes a subset of the join result using any efficient local join algorithm.

Analysis. The choice of B, i.e., the number of buckets, affects the runtime performance of DO-join. Let $U = \max \mathcal{G}(\epsilon, \delta)$, we have $\mathcal{G}(\epsilon_b, \delta_b) = O(U)$ and the independent noise added to HTree's nodes is $\mathcal{G}(\epsilon_h, \delta_h, h, 0) = O(hU)$. For the extreme cases $B = 1$ and $B = D$, the sum of bucket-wise Cartesian product sizes $|\tilde{R}(q)|$ is $O(N^2)$ and $O(r + DU^2)$ respectively. By choosing a value of B between 1 and D, it is possible to produce a significantly smaller $|\tilde{R}(q)|$. A smaller $|\tilde{R}(q)|$ implies a more efficient bucket-wise Cartesian product join and ObliCompact, which is the performance bottleneck of DO-join.

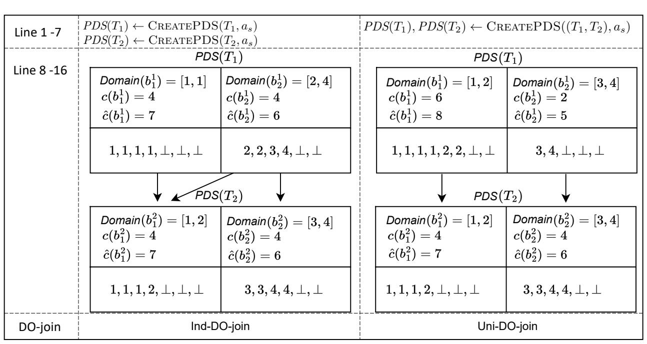


Figure 5: An example of Ind-DO-join and Uni-DO-join

Algorithm 7: DO-PF-join (T_1, T_2, a_s)

1 $PDS(T_2) \leftarrow$ CreatePDS(T_2, a_s);
  `//` T_1 `is stored in the order of its primary key`
2 ObliExpand(T_1, F, D), such that record $r_i = T_1[i]$ associated with key x_j is at position $F[i] = x_j - x_1 + 1$;
3 **foreach** *bucket* b_i^2 *in* $PDS(T_2)$ **do**
4   If $Domain(b_i^2) = [x_s, x_e]$, let bucket $b_i^1 = \{T_1[x_s - x_1 + 1], \cdots, T_1[x_e - x_1 + 1]\}$, whose domain is also $[x_s, x_e]$;
5   Add the result of ObliSortMerge(b_i^1, b_i^2) to $R(q)$;
6 **end**

In our approach, we set $B = chN/U$ (c is a constant). By employing constrained inference, we get an average noise of $\frac{h^2U}{D}$ 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 $c(b_i) = O(\theta + \frac{h^2U}{D} \cdot \frac{D}{B}) = O(U)$ real records in the created bucket. The noisy bucket count is $\widehat{c}(b_i) = c(b_i) + \mathcal{G}(\epsilon_b, \delta_b) = O(U)$. Therefore, the sum of bucket-wise Cartesian product sizes is $|\tilde{R}(q)| \leq 2B \cdot \max \widehat{c}(b_i)^2 = O(r + hNU)$. When the domain size of the join attribute is proportional to the input size (i.e., $D = O(N)$), the grouping technique using $B = chN/U$ reduces the amount of noise in $\tilde{R}(q)$ by a factor of $U/h = \log_2 k_b$, in comparison to the extreme case $B = D$. Actually, if the domain size is a constant and independent of N, $|\tilde{R}(q)| = O(r + NU)$, 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 $|PDS(T_t)| = N + B * O(U) = O(hN)$ 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] $\Delta_{\text{join}}(T_1, T_2) = \max \widehat{c}(b_i) = O(U)$. Furthermore, the compacted join output size is $R = r + \mathcal{G}(\epsilon_c, \delta_c, \Delta_{\text{join}}(T_1, T_2)) = O(r + U^2)$, where the privacy budget (ϵ_c, δ_c) is a constant fraction of the total privacy budget subjecting to $\epsilon = \epsilon' + \epsilon_c$ and $\delta = \delta' + \delta_c$. Finally, the communication volume is $R = O(r + U^2)$.

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 T_2 into several partitions using CreatePDS. It then expands the primary-key table T_1 to the domain size D, such that a record with join key $x_j \in [x_1, \cdots, x_D]$ is at position $x_j - x_1 + 1$, where x_1 is the smallest value of $Domain(a_s) = [x_1, \cdots, x_D]$. ObliSortMerge is invoked for each partition covering $Domain(b_i^2)$. In the example depicted in Fig. 6, T_2 is divided to two partitions with domains $[1, 2]$ and $[3, 4]$ respectively. The primary key table T_1 is expanded to the domain size $D = 4$. The i^{th} bucket b_i^1 in T_1 is constructed based on the corresponding b_i^2 bucket in T_2 such that they share the same domain on the join attribute (line 4 in Algo. 7).

5.2.4 DO-select-join. DO-join naturally supports selection predicates on the join attributes. Suppose that there is a set of selection predicates on the join attribute a_s, DO-select-join can first follow

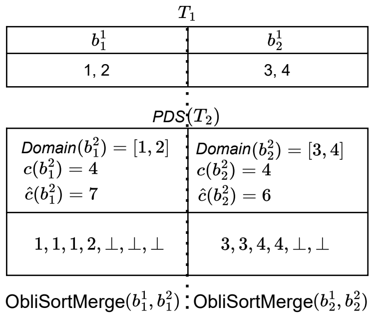


Figure 6: An example of DO-PF-join

step 1-7 of Algo. 6 to get $PDS(T_t)$ $t \in \{1, 2\}$. 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` $T_1.a_s = T_2.a_s$ `AND` $T_2.f_s > v$", in a more efficient way than sending back to the client the complete set of join output of $T_1.a_s = T_2.a_s$ without applying any selection predicate, $PDS(T_2)$ should support filtering operations for attribute f_s and join operations for attribute a_s. In other words, $PDS(T_2)$ should be multi-dimensional. Suppose that $Domain(a_s) = Domain(f_s) = [0, 10]$ 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 $PDS(T_2, (a_s, f_s))$ as depicted in Fig. 7, and a one-dimensional $PDS(T_1, a_s)$. Then DO-select is invoked to obtain the set of buckets in $PDS(T_2, (a_s, f_s))$ that satisfy the predicate $T_2.f_s > v$, i.e., $L_2 = \{b_2^2, b_3^2\}$. For each of the qualified buckets, it looks for the buckets in $PDS(T_1, a_s)$ that have domain overlaps on the join attribute a_s with the qualified bucket (line 6 in Algo. 8). Cartesian product join is performed on each overlapped bucket pair: (b_1^1, b_2^2), (b_2^1, b_2^2), and (b_2^1, b_3^2). 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 $\mathcal{L}$(domains) when looking for overlapped bucket pairs, and reveals bucket capacities $\mathcal{L}$(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 (ϵ_c, δ_c)-DO join output compaction. Since the union of the leakages $\mathcal{L}$(domains), $\mathcal{L}$(capacities), and (ϵ_c, δ_c)-DP join output size is (ϵ, δ)-DP, and the fact that DO-join and DO-select-join are oblivious w.r.t. the (ϵ, δ)-DP leakage, we can conclude that DO-join and DO-select-join are (ϵ, δ)-DO by Lemma 1.

DO-PF-join (Algo. 7) without (ϵ_c, δ_c)-DO join output compaction is (ϵ', δ')-DO: the partitioning strategy CreatePDS is (ϵ', δ')-DO; ObliExpand is fully oblivous with input size $|T_1|$ and output size D; fully oblivious ObliSortMerge is applied to each partition.

Algorithm 8: DO-select-join (T_1, T_2, a_s, f_s, v)

1 $PDS(T_1, a_s) \leftarrow$ CreatePDS(T_1, a_s);
2 $PDS(T_2, (a_s, f_s)) \leftarrow$ CreatePDS$(T_2, (a_s, f_s))$;
3 $L_2 \leftarrow$ DO-select$(T_2, f_s, v, \text{inf})$;
4 **foreach** *bucket* b_i *in* L_2 **do**
5   Let $Domain(b_i, a_s) = [x_s, x_e]$ be b_i's domain of a_s;
6     $L_1 \leftarrow$ DO-select(T_1, a_s, x_s, x_e);
7   **foreach** *bucket* b_j *in* L_1 **do**
8   Compute the Cartesian product of b_i and b_j;
9   Add the result to $\tilde{R}(q)$;
10   **end**
11 **end**
12 $R(q) =$ ObliCompact$(\tilde{R}(q), R)$, where
    $R = r + \mathcal{G}(\epsilon_c, \delta_c, \Delta_{\text{select-join}}(T_1, T_2))$;

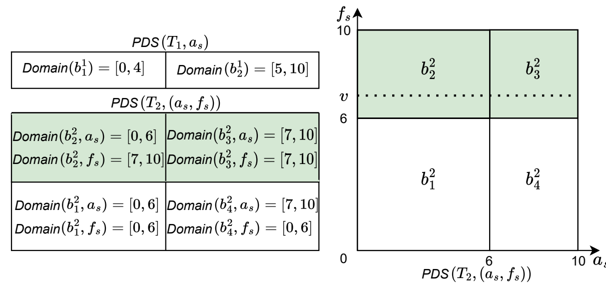


Figure 7: An example of 2D DO-select-join

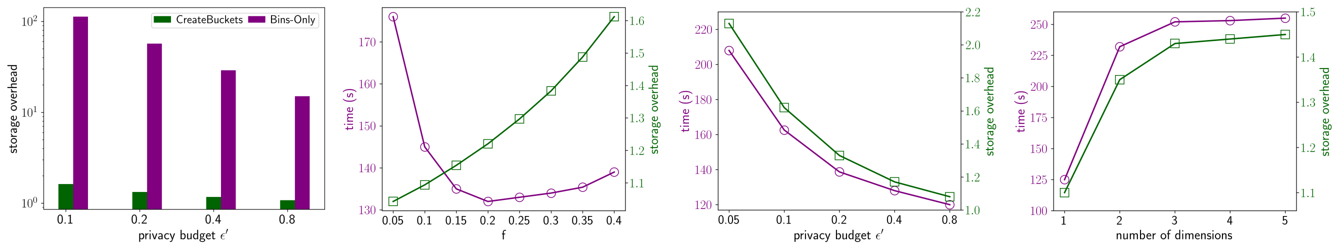
6 EVALUATION

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
```



(a) Bucketization efficiency **(b) Budget (ϵ', δ') division** **(c) Effect of ϵ'** **(d) Usability of m-d *PDS***

Figure 8: *PDS* costs

```
     WHERE ws_sold_date_sk = ss_sold_date_sk
TE6: SELECT inv_warehouse_sk
     FROM web_sales, inventory
     WHERE ws_item_sk = inv_item_sk
```

Baselines. We compare Doquet with the in-memory and no-EPC-swapping implementations of other baselines, as follows. Reported runtimes are averages of 5 measurements.

- DO-select vs. Adore [42], Opaque [56], and Epsolute [7]. We run Opaque's filtering algorithm and Epsolute with DP padding on the output, which makes them satisfy DO. Epsolute is set up following the instructions in [7] with 64 ORAMs running on 8 ORAM servers and 8 Redis servers, each of which has 8 Redis services (64 Redis services in total). We compare with Adore by allowing their assumption of private memory.
- DO-join vs. ODBJ [34] and CZSC21 [13]. The adapted (ϵ, δ)-DO ODBJ is described in Section 5.2.1. CZSC21 is an (ϵ, δ)-DO many-to-many join algorithm, described in Section 7. We also compare DO-PF-join with Adore [42].
- DO-select and DO-join vs. Shrinkwrap [6] in a circuit model. We build a prototype for range and join queries using the EMP toolkit [53], based on the code Shrinkwrap authors kindly shared with us. Shrinkwrap supports FO query operators and DP post-processing (i.e., shrinking the output size from the worst-case to a DP value) using secure multi-party computation (MPC).

Default settings. We set the encrypted block size to 512 bytes. The default privacy budget is $\epsilon = 0.3$ and $\delta = 2 * (1/N)^{1.3} < 2^{-20}$ for $N \geq 10^5$. We split the budget to $(\epsilon', \delta') = (0.28, (1/N)^{1.3})$ and $(\epsilon_c, \delta_c) = (0.02, (1/N)^{1.3})$ for join queries. The tree fanout $k_b = 16$ for the HTree method, as suggested in [41].

Performance measures. Besides runtime, we measure storage and communication overhead w.r.t. their optimal values, i.e., $|PDS(T)|/N$ and R/r , to assess the storage efficiency of *PDS* and the communication efficiency of queries, respectively.

6.1 *PDS* Costs

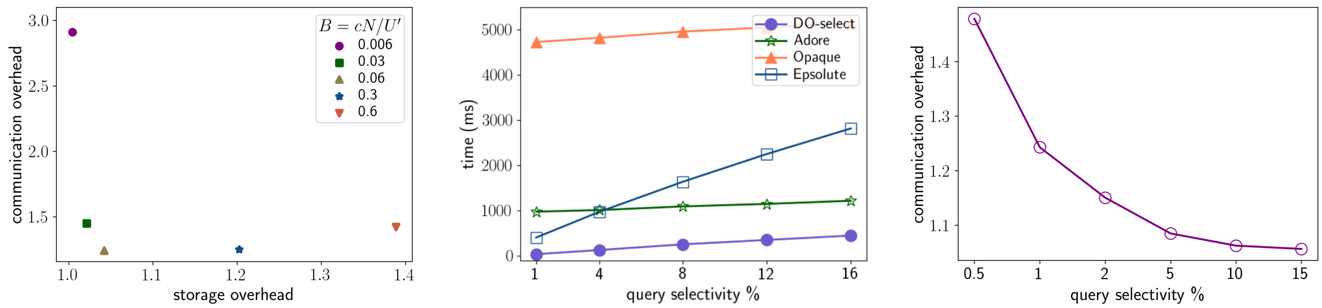
We evaluate HTree-based *PDS* with the default synthetic dataset of 10^6 records, by setting one of the attributes as the key attribute a_s and other attributes as a value. We first show the positive effect of bucketization on storage in Fig. 8a. As a competitor, we used a Bins-Only method, which uses 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 (ϵ', δ') to (ϵ_h, δ_h) and (ϵ_b, δ_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 (ϵ_h, δ_h) is simply used to generate noisy bin counts to merge bins to equal-size buckets, while (ϵ_b, δ_b) directly affects the amount of noise added to each bucket. Fig. 8c shows the impact of the privacy budget (ϵ', δ') on the runtime of CreatePDS and on the storage overhead. Both of them decrease as ϵ' 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.

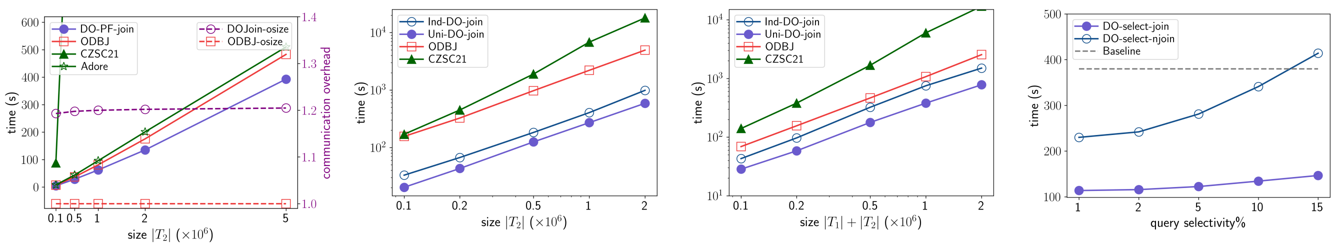
Select. We evaluate DO-select with 1D range queries. We first test how the number of buckets $B = cN/U'$ affects the storage and communication overhead for different values of c (Fig. 9a). The communication overhead is measured for a query workload with selectivity 1%. The point lying closer to the origin 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 Epsolute in Fig. 9b. We assume that the supported data structures for selection queries already exist (i.e., *PDS* of DO-select and ORAM of Epsolute have been built before the first query).[4] 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 Shrinkwrap 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

[4]The preparation time, i.e., the one-time cost spent in the construction of the supported data structures, of Adore and Opaque is negligible, and the preparation time of DO-select and Epsolute (with parallelization enabled) is 125s and 27s respectively.



(a) Number of buckets and efficiency **(b) Performance against query selectivity** **(c) Query communication overhead**

Figure 9: DO-select



(a) PF-join, $|T_2|$=100$|T_1|$ **(b) General join, $|T_2|$=100$|T_1|$** **(c) General join, $|T_2|$=$|T_1|$** **(d) DO-select-join, $|T_2|$=$|T_1|$, $N = 10^6$**

Figure 10: DO-join

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 $|T_2| = 5 * 10^6$, 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., $|T_2| = 100|T_1|$, 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

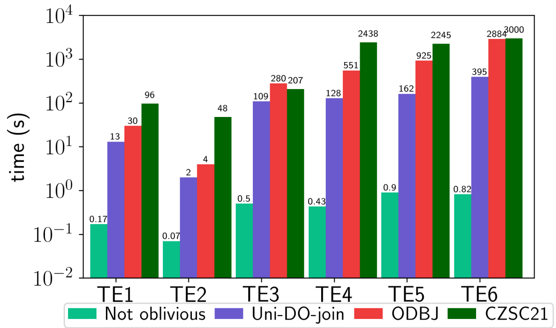


Figure 11: TE1-TE6 join query costs

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

	TE1	TE2	TE3	TE4	TE5	TE6
DO-join	**1.68**	2.3	**1.05**	1.36	1.06	1.13
ODBJ	2.04	2.28	1.07	9.31	1.8	9.33
CZSC21	**1.68**	**2.22**	**1.05**	**1.04**	**1.03**	**1.03**

7 RELATED WORK

DP Histograms There is a batch of algorithms to publish DP histograms [27, 32, 33, 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 oblivous 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 (ϵ, δ)-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 oblvious 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 (HFRI) under the “2nd Call for HFRI Research Projects to support Faculty Members Researchers” (Project No. 2757).

REFERENCES

[1] Miklós Ajtai, János Komlós, and Endre Szemerédi. 1983. An 0 (n log n) sorting network. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*. 1–9.

[2] Arvind Arasu and Raghav Kaushik. 2013. Oblivious query processing. *arXiv preprint arXiv:1312.4012* (2013).

[3] Gilad Asharov, TH Hubert Chan, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. 2020. Bucket oblivious sort: An extremely simple oblivious sort. In *Symposium on Simplicity in Algorithms*. SIAM, 8–14.

[4] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Enoch Peserico, and Elaine Shi. 2020. Oblivious parallel tight compaction. *Cryptology ePrint Archive* (2020).

[5] Kenneth E Batcher. 1968. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*. 307–314.

[6] Johes Bater, Xi He, William Ehrich, Ashwin Machanavajjhala, and Jennie Rogers. 2018. Shrinkwrap: efficient sql query processing in differentially private data federations. *Proceedings of the VLDB Endowment* 12, 3 (2018).

[7] Dmytro Bogatov. 2021. *Epsolute*. https://github.com/epsolute/epsolute

[8] Dmytro Bogatov, Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. 2021. εpsolute: Efficiently Querying Databases While Providing Differential Privacy. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2262–2276.

[9] Elette Boyle and Moni Naor. 2016. Is There an Oblivious RAM Lower Bound?. In *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science, Cambridge, MA, USA, January 14-16, 2016*, Madhu Sudan (Ed.). ACM, 357–368. https://doi.org/10.1145/2840728.2840761

[10] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software grand exposure:{SGX} cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*.

[11] TH Hubert Chan, Kai-Min Chung, Bruce M Maggs, and Elaine Shi. 2019. Foundations of differentially oblivious algorithms. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2448–2467.

[12] Zhao Chang, Dong Xie, Sheng Wang, and Feifei Li. 2022. Towards Practical Oblivious Join. (2022).

[13] Shumo Chu, Danyang Zhuo, Elaine Shi, and TH Chan. 2021. Differentially Oblivious Database Joins: Overcoming the Worst-Case Curse of Fully Oblivious Algorithms. In *2nd Conference on Information-Theoretic Cryptography (ITC 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[14] Victor Costan and Srinivas Devadas. 2016. Intel SGX explained. *Cryptology ePrint Archive* (2016).

[15] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. 2018. Obladi: Oblivious serializable transactions in the cloud. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 727–743.

[16] Tien Tuan Anh Dinh, Prateek Saxena, Ee-Chien Chang, Beng Chin Ooi, and Chunwang Zhang. 2015. M2r: Enabling stronger privacy in mapreduce computation. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 447–462.

[17] Cynthia Dwork. 2006. Differential Privacy. In *33rd International Colloquium on Automata, Languages and Programming, part II (ICALP 2006)*, Vol. 4052. 1–12.

[18] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. 2006. Calibrating noise to sensitivity in private data analysis. In *Theory of cryptography conference*. Springer, 265–284.

[19] Cynthia Dwork, Aaron Roth, et al. 2014. The algorithmic foundations of differential privacy. *Foundations and Trends® in Theoretical Computer Science* 9, 3–4 (2014), 211–407.

[20] Cynthia Dwork, Guy N Rothblum, and Salil Vadhan. 2010. Boosting and differential privacy. In *2010 IEEE 51st Annual Symposium on Foundations of Computer Science*. IEEE, 51–60.

[21] Saba Eskandarian and Matei Zaharia. 2019. ObliDB: Oblivious Query Processing for Secure Databases. *Proc. VLDB Endow.* 13, 2 (2019), 169–183.

[22] Oded Goldreich. 1987. Towards a theory of software protection and simulation by oblivious RAMs. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. 182–194.

[23] Oded Goldreich and Rafail Ostrovsky. 1996. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)* 43, 3 (1996), 431–473.

[24] Michael T Goodrich. 2014. Zig-zag sort: A simple deterministic data-oblivious sorting algorithm running in o (n log n) time. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*. 684–693.

[25] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. 2018. Pump up the volume: Practical database reconstruction from volume leakage on range queries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 315–331.

[26] Hakan Hacigümüs, Sharad Mehrotra, and Balakrishna R. Iyer. 2002. Providing Database as a Service. In *Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, USA, February 26 - March 1, 2002*, Rakesh Agrawal and Klaus R. Dittrich (Eds.). IEEE Computer Society, 29–38.

[27] Michael Hay, Vibhor Rastogi, Gerome Miklau, and Dan Suciu. 2009. Boosting the accuracy of differentially-private histograms through consistency. *arXiv preprint arXiv:0904.0942* (2009).

[28] Thang Hoang, Muslum Ozgur Ozmen, Yeongjin Jang, and Attila A Yavuz. 2019. Hardware-supported ORAM in effect: Practical oblivious search and update on very large dataset. *Proceedings on Privacy Enhancing Technologies* 2019, 1 (2019).

[29] Intel. 2021. *3rd Gen Intel Xeon Scalable Processors Brief.* https://www.intel.com/content/www/us/en/products/docs/processors/xeon/3rd-gen-xeon-scalable-processors-brief.html

[30] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'neill. 2016. Generic attacks on secure outsourced databases. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 1329–1340.

[31] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. 2017. Accessing data while preserving privacy. *CoRR, abs/1706.01552* 5 (2017).

[32] Georgios Kellaris and Stavros Papadopoulos. 2013. Practical differential privacy via grouping and smoothing. *Proceedings of the VLDB Endowment* 6, 5 (2013), 301–312.

[33] Georgios Kellaris, Stavros Papadopoulos, and Dimitris Papadias. 2018. Engineering methods for differentially private histograms: Efficiency beyond utility. *IEEE Transactions on Knowledge and Data Engineering* 31, 2 (2018), 315–328.

[34] Simeon Krastnikov, Florian Kerschbaum, and Douglas Stebila. 2020. Efficient Oblivious Database Joins. *Proc. VLDB Endow.* 13, 12 (2020), 2132–2145.

[35] Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. 2018. Improved reconstruction attacks on encrypted data using range query leakage. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 297–314.

[36] Kasper Green Larsen and Jesper Buus Nielsen. 2018. Yes, There is an Oblivious RAM Lower Bound!. In *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part II (Lecture Notes in Computer Science)*, Hovav Shacham and Alexandra Boldyreva (Eds.), Vol. 10992. Springer, 523–542. https://doi.org/10.1007/978-3-319-96881-0_18

[37] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. 2020. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[38] Chao Li, Michael Hay, Gerome Miklau, and Yue Wang. 2014. A data-and workload-aware algorithm for range queries under differential privacy. *arXiv preprint arXiv:1410.0265* (2014).

[39] Ilya Mironov, Omkant Pandey, Omer Reingold, and Salil Vadhan. 2009. Computational differential privacy. In *Advances in Cryptology-CRYPTO 2009: 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2009. Proceedings*. Springer, 126–142.

[40] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. 2018. Oblix: An efficient oblivious search index. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 279–296.

[41] Wahbeh Qardaji, Weining Yang, and Ninghui Li. 2013. Understanding hierarchical methods for differentially private histograms. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1954–1965.

[42] Lianke Qin, Rajesh Jayaram, Elaine Shi, Zhao Song, Danyang Zhuo, and Shumo Chu. 2022. Differentially Oblivious Relational Database Operators. *Proc. VLDB Endow.* 16, 4 (2022), 842–855.

[43] Lina Qiu. 2023. *Doquet*. https://github.com/linaqiu22/DOQP.git

[44] Vijaya Ramachandran and Elaine Shi. 2021. Data oblivious algorithms for multicores. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*. 373–384.

[45] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. 2015. Trusted execution environment: what it is, and what it is not. In *2015 IEEE Trustcom/BigDataSE/ISPA*, Vol. 1. IEEE, 57–64.

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

[47] Sajin Sasy, Aaron Johnson, and Ian Goldberg. 2022. Fast Fully Oblivious Compaction and Shuffling. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2565–2579.

[48] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy data analytics in the cloud using SGX. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 38–54.

[49] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2017. Malware guard extension: Using SGX to conceal cache attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 3–24.

[50] AMD SEV-SNP. 2020. Strengthening VM isolation with integrity protection and more. *White Paper, January* (2020).

[51] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *Proceedings of the 26th USENIX Security Symposium*. USENIX Association, 1041–1056.

[52] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. 2017. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2421–2434.

[53] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. 2016. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit.

[54] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 640–656.

[55] Jun Zhang, Xiaokui Xiao, and Xing Xie. 2016. Privtree: A differentially private algorithm for hierarchical decompositions. In *Proceedings of the 2016 International Conference on Management of Data*. 155–170.

[56] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. 2017. Opaque: An oblivious and encrypted distributed analytics platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 283–298.

[57] Mingxun Zhou, Elaine Shi, T-H Hubert Chan, and Shir Maimon. 2023. A theory of composition for differential obliviousness. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 3–34.