Adore: Differentially Oblivious Relational Database Operators

Lianke Qin
UC Santa Barbara
lianke@ucsb.edu

Rajesh Jayaram
Carnegie Mellon University
rkjayara@cs.cmu.edu

Elaine Shi
Carnegie Mellon University
runting@cs.cmu.edu

Zhao Song
Adobe Research
zsong@adobe.com

Danyang Zhuo
Duke University
danyang@cs.duke.edu

Shumo Chu
p0x labs
chushumo@cs.washington.edu

ABSTRACT

There has been a recent effort in applying differential privacy on memory access patterns to enhance data privacy. This is called differential obliviousness. Differential obliviousness is a promising direction because it provides a principled trade-off between performance and desired level of privacy. To date, it is still an open question whether differential obliviousness can speed up database processing with respect to full obliviousness. In this paper, we present the design and implementation of **Adore**: **A** set of **D**ifferentially **O**blivious **RE**lational database operators. Adore includes selection with projection, grouping with aggregation, and foreign key join. We prove that they satisfy the notion of differential obliviousness. Our differentially oblivious operators have reduced cache complexity, runtime complexity, and output size compared to their state-of-the-art fully oblivious counterparts. We also demonstrate that our implementation of these differentially oblivious operators can outperform their state-of-the-art fully oblivious counterparts by up to 7.4×.

PVLDB Reference Format:
Lianke Qin, Rajesh Jayaram, Elaine Shi, Zhao Song, Danyang Zhuo, and Shumo Chu. Adore: Differentially Oblivious Relational Database Operators. PVLDB, 16(4): 842 - 855, 2022.
doi:10.14778/3574245.3574267

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/brucechin/dodb.

1 INTRODUCTION

Moving data and computation to the cloud is the most dominant trend in the industry today. Cloud databases [30, 46, 59, 77] collect and analyze a vast amount of user data, including sensitive information such as health data, financial records, and social interactions. These databases allow developers to run complex queries using a SQL interface, the de facto standard for data analytics. Because of these developments, cloud data is often the central target of attacks [18, 22, 32, 33, 41], protecting sensitive data in cloud databases has become more important than ever.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 16, No. 4 ISSN 2150-8097.
doi:10.14778/3574245.3574267

A promising direction is to use hardware enclaves, such as Intel SGX [57], and RISC-V Sanctum [28], to provide secure data processing inside the cloud. These enclaves are protected regions in CPUs, where a remotely attested piece of code can run without interference from a potentially adversarial hypervisor and OS. Major processor vendors have all equipped their new generation of CPUs with hardware enclaves. Cloud providers like Microsoft and Alibaba provide enclave support in their public cloud offerings [1, 4]. Some cloud databases [7, 60] have already used Intel SGX to protect user data, and it is also an area of active research [8, 67].

Unfortunately, the Achilles' heel of using hardware enclaves is that enclaves alone do not protect the access patterns of encrypted data outside the enclave's memory. For applications like big data analytics that require managing a large amount of data, an enclave has to fetch encrypted data residing outside the enclave (e.g., a server's main memory, disks). This leads to *access pattern* attacks [49, 63]. A long list of practical access pattern attacks of this form [2, 40, 45, 52, 53, 79] have been discovered for encrypted databases such as CryptDB [66] and TrustedDB [11].

One approach to address this vulnerability is to make the memory access patterns of enclave-based database systems *oblivious*, which means that the access patterns of the system are indistinguishable for different input data. This notion of obliviousness was first proposed by Goldreich and Ostrovsky [43]. However, making the database systems fully oblivious incurs a huge performance penalty. For example, any query output including intermediate results must be padded with filler tuples to the worse-case size, which is usually much larger than the actual result size. In recent enclave-based databases (e.g., Opaque [83], ObliDB [39]), their fully oblivious modes[1] are significantly slower than their partially oblivious or non-oblivious counterparts. While their partially oblivious or non-oblivious mode either does not protect memory access patterns at all or has arbitrary leakage, such as leaking the sizes of intermediate results and outputs. The ramifications of such leakages are not understood and may likely lead to new attacks.

Recently, there are rising interests in adopting differential privacy to protect access pattern leakage. To apply this idea to databases, instead of making access patterns indistinguishable between all inputs, *we make the access patterns satisfy differential privacy* [36, 37], a privacy model that only requires indistinguishability among neighboring databases. This notion is called *differential obliviousness*, and was introduced by Chan et al. [23]. This relaxation from full obliviousness opens up new design spaces for more efficient algorithms, yet still provides provable privacy guarantees for each database record. Differential oblivious algorithms only add dummy

[1] ObliDB calls its fully oblivious mode padding mode.

Table 1: Cache complexity/Private Memory Size/Output size comparisons of our system with ObliDB [39]. N denotes input size, R denotes output size, B denotes block size, M denotes the size of private memory. Let FO denote Full obliviousness. Let DO denote Differential obliviousness. Let * denote the result from our interpretation of their algorithms, the original work didn't explicit state the result. Let ✓denote the best choice. ObliDB's hash-based grouping with aggregation fails when the total number of distinct groups exceeds the enclave private memory capacity M. Cache complexity measures the total numbers of data blocks movement between private memory and untrusted memory, which is the dominant overhead. Runtime complexity measures the total number of CPU instructions executed on the decrypted data within enclave.

Operator	Algorithms	Privacy	Private Mem. Size	Cache Complexity	Runtime Complexity	Output Size
Selection with projection	ObliDB	FO	1 ✓	$2N^*$	$O(N)^*$	N
Selection with projection	**Alg. 1**	DO	$\text{poly}\log(N)$	$(N+R)/B$ ✓	$O(N+R)$	$R+\text{poly}\log(N)$ ✓
Grouping with aggregation	ObliDB	FO	$M(M > R)$	$2N^*$	$O(N)^*$	M^*
Grouping with aggregation	**Alg. 2**	DO	$M\,(M \geq O(\epsilon^{-1}\log^2(1/\delta)))$	$N/B + 11NR/9MB$ ✓	$O(NR/M)$	$\frac{11}{9}R$
Foreign key join	ObliDB	FO	$\text{poly}\log(N)$ ✓	$N \cdot \log^2(N)^*$	$O(N\log^2(N))^*$	N
Foreign key join	**Alg. 3**	DO	$\text{poly}\log(N)$✓	$6(N/B)\cdot\log(N/B) + N/B + R/B + \text{poly}\log(N)/B$ ✓	$O(N\log(N))$ ✓	$R+\text{poly}\log(N)$ ✓

reads/writes during execution to obfuscate the memory access pattern, and they do not change the query results (except additional dummy tuples). The query accuracy is therefore not affected.

This raises two salient questions: (1) How to design differentially oblivious database operators? (2) Can differentially oblivious database operators outperform their state-of-the-art fully oblivious operators?

In this paper, we present **Adore**: **A** set of **D**ifferentially **O**blivious **RE**lational database operators, including selection with projection, grouping with aggregation, and foreign key join. We pick these operators because they are sufficient to support important database workloads, such as big data benchmark (BDB) [6]. We use three key theoretical performance metrics to guide our design: (1) cache complexity, (2) runtime complexity, and (3) output size. Cache complexity measures the total numbers of blocks read from untrusted memory to enclave memory (a.k.a. *private memory*), and written from private memory to untrusted memory. In this scenario, the enclave memory is the "cache" and each page is a "block" (i.e., the atomic unit being swapped in and out). Cache complexity is a dominant source of query latency because moving data between trusted and untrusted memory requires memory copying, encryption, decryption and the overhead from SGX ECALLS and OCALLS. Using this metric is further justified by our microbenchmark results in §5.2: in most queries, the memory copy, encryption, and decryption together constitute more than 80% of total query completion time. Second, we consider runtime complexity, which is the total number of CPU instructions executed on the decrypted data within enclave to implement the filter, aggregation and join operators, but they are not the dominant overhead compared with data movement between enclave private memory and untrusted memory which is measured by cache complexity metric. Output size is also an important metric: the output size decides how much data needs to transfer from trusted to untrusted memory to generate the output. In Table 1, we provide detailed comparison between our differentially oblivious database operators and ObliDB. We can reduce the cache complexity and output size over ObliDB asymptotically.

We implement these differentially oblivious database operators. Our prototype is developed on top of Intel SGX, because of its availability. We acknowledge that SGX will be deprecated starting from the 11th generation of Intel CPU Core CPUs, but it will continue on Intel Xeon CPUs for cloud usage, which is our target deployment scenario. Choosing SGX also means our prototype is susceptible to known SGX vulnerabilities. These vulnerabilities have known solutions, and patching our prototypes for these vulnerabilities is out of scope of our paper. (See §7.)

We evaluate our implementation using workloads from BDB. We show that our operators can substantially outperforms the fully oblivious operators in ObliDB. Overall, our operators provide up to 7.4× performance improvement over ObliDB. Our operators also allow scaling to larger data compared with the existing oblivious operators: our operators can process input tables containing 30 million tuples in groupby in BDB, while ObliDB fails because the total number of distinct groups is larger than M. Our source code and scripts for running the evaluation are available anonymously at https://github.com/brucechin/dodb.

Our paper makes the following contributions:

- We apply the notion of differential obliviousness to database operators to enhance data privacy by designing three new differentially oblivious database operators.
- We formally prove that these operators satisfy the notion of differential obliviousness and have reduced cache complexity and output size.
- We demonstrate empirically these differential oblivious operators' performance gain compared to their state-of-the-art fully oblivious counterparts.

Roadmap. We first introduce our threat model and background knowledge in §2. We present our differentially oblivious operators including filter, grouping with aggregation and foreign key join in §3. We present a differentially private distinct count algorithm in §4 and use it in differentially oblivious grouping with aggregation. We evaluate the performance improvement of our algorithm in §5. We discuss the related work in §6. We discuss the potential future work in §7. We conclude our paper in §8.

2 BACKGROUND

In this section, we first describe our threat model (§2.1). Then, we formally define differential obliviousness and compare it with full obliviousness (§2.2).

2.1 Threat Model

We use Intel SGX as an example to discuss the threat model. Intel SGX provides confidentiality and integrity of its enclave memory (i.e., *private memory*), which is located in a preconfigured part of DRAM called the Processor Reserved Memory (PRM). The content

in the enclave memory is encrypted. The enclave memory also guarantees integrity: only the code residing inside the enclave can modify the enclave memory after the enclave is created. The enclave memory size has an upper bound (i.e., 128 MB). An SGX enclave has a predefined entry point, so a user process or the OS cannot invoke the enclave to run at arbitrary memory addresses. SGX provides *remote attestation* to allow a remote system to verify what code is loaded into an enclave, and set up a secure communication channel to the enclave.

These SGX features allow us to trust the code running inside the enclave. Untrusted processes and the operating system cannot tamper with the database source code inside the enclave. The execution and memory accesses for the private memory are also invisible to the untrusted processes and the operating system.

However, the database requires an untrusted component for I/O. For a trusted data owner to use the database, the data owner sends an encrypted query to the untrusted component, and the untrusted component forwards the query to the enclave. The enclave decrypts the query and asks the untrusted component to load encrypted data from the public memory into the enclave. The enclave then decrypts the input data, processes the data, and returns the encrypted result to the untrusted component. The untrusted component forwards the result back to the trusted data owner, who has a decryption key to see the query result. During the query processing, the enclave can also send encrypted intermediate results to the untrusted memory and later load them back. This is often needed because enclaves have limited memory. The enclave checks the MACs of the input data and the intermediate results to prevent the cloud server from modifying them.

Unfortunately, the access patterns in the public memory are exposed to the untrusted cloud server. This means an attacker can watch how the enclave reads the encrypted data, writes the encrypted output, and reads/writes the intermediate result. Data access pattern leakage is sufficient for the attacker to extract secrets and data from many encrypted systems [16, 54, 64]. Our threat model is the same with ObliDB [39].

2.2 Differential Obliviousness

Differential obliviousness. The notion of differential obliviousness was proposed by Chan et al. [23]. It essentially requires that the memory traces of an algorithm satisfy differential privacy [36, 37]. To provide some background, differential privacy was introduced in the seminal work by Dwork et. al [36, 37], which is a framework for adding noise to data so that the published result would not harm any individual user's privacy. Over the years, differential privacy has become the de facto standard for privacy, with growing acceptance in the industry. In the differential privacy literature, we typically assume that the data curator is fully trusted, and thus we care about adding noise to the computation result. However, in our setting, the data curator (i.e., the cloud provider) is untrusted. Our goals therefore depart from the standard differential privacy literature. Instead of requiring the outputs of the computation to be differentially private, we require that the database system's observable runtime behavior, namely, the access patterns, be differentially private. As mentioned, Chan et al. [23] formulated this notion as differential obliviousness.

With differential obliviousness, the untrusted cloud provider cannot extract private information for each individual by observing memory access patterns. A differentially oblivious system is resilient to the attacks mentioned in §2.1. We formally define differential obliviousness in Definition 2.1.

Henceforth, we may view a database as an ordered sequence of records. We say that two databases D_1 and D_2 are *neighboring*, iff they are of the same length, and moreover, they differ in exactly one record.

Definition 2.1 (Differential Obliviousness [23])**.** An algorithm $\mathcal{A}$ is (ϵ, δ)-differentially oblivious if for any two **neighboring databases** D_1, D_2, and any subset of memory access patterns S:

$$\Pr[\mathcal{M}(\mathcal{A}, D_1) \in S] \leq e^{\epsilon} \cdot \Pr[\mathcal{M}(\mathcal{A}, D_2) \in S] + \delta.$$

Here, we use $\mathcal{M}(\mathcal{A}, D)$ to denote the distribution of memory access patterns when we apply the algorithm $\mathcal{A}$ on D. The ϵ parameter is a metric of privacy loss. It also controls the privacy-utility trade-off. The δ parameter accounts for a negligible probability on which the upper bound ϵ does not hold. The memory access pattern is a sequence of memory operations, including the address of each operation and the type of operation (read or write). Since the data contents are encrypted, we may assume that the adversary observes only the addresses and types of the operations but not the contents.

It is important to note that in Definition 2.1 above, we allow the databases D_1 and D_2 to contain two types of records, *real* records and *filler* records. We allow the filler records so that deleting one entry from the database can be accomplished by replacing the entry with a filler.

Differential obliviousness perfectly captures the threat model enclave-based database systems face: as we discussed in §2.1, for an enclave-based database system, the data and code execution within the enclave can be considered secure, and the data stored outside the enclave is encrypted but accesses to it leak information. Specifically, in our SGX-based scenario, each memory access observable by the adversary is a page swap event: whenever the SGX enclave wants to swap in or out a new (encrypted) memory page, it needs to contact the untrusted OS for help.

Comparison with full obliviousness. It is also instructive to compare the notion of differential obliviousness with the more classical, *full obliviousness* notion first proposed by Goldreich [42]. We formally define full obliviousness below in Definition 2.2.

Definition 2.2 (Full Obliviousness)**.** An algorithm $\mathcal{A}$ is oblivious if for any two databases D_1, D_2 of the same size and any subset of possible memory access patterns S:

$$\Pr[\mathcal{M}(\mathcal{A}, D_1) \in S] \leq \Pr[\mathcal{M}(\mathcal{A}, D_2) \in S] + \delta$$

Differential obliviousness is a relaxation of full obliviousness in the following senses: (1) differential obliviousness only requires the memory access patterns over *neighboring* databases to be indistinguishable; (2) the definition of indistinguishability is also relaxed in differential obliviousness, in the sense that we additionally allow a multiplicative e^{ϵ} factor when measuring the distance between the two access pattern distributions.

These relaxations make designing more I/O efficient algorithms possible. For example, in a fully oblivious model, a database system

has to add filler tuples to the result until it reaches the worst-case size. In database queries, this worst-case size could be orders of magnitude worse than the average-case size. However, with differential obliviousness, it suffices for the database system to add a small, random number of fillers so that the output size is indistinguishable for two neighboring databases.

3 DIFFERENTIALLY OBLIVIOUS OPERATORS

In this section, we propose a series of differentially oblivious algorithms that implement major relational operators, including selection with projection, grouping with aggregation, and foreign key join.

Overview of our differentially oblivious operators. We propose a differentially oblivious algorithm for selection with projection with optimal cache complexity. The main technique of this algorithm is inspired by a theoretical result on differentially oblivious compaction [23]: using a differentially private prefix-sum sub-routine to guide the memory access of filtering (§3.1). Next, we propose a differentially oblivious algorithm for grouping with aggregation. Notably, to develop this algorithm, we propose a novel, practical differentially private distinct count algorithm (Algorithm 5). This is the *first* practical differentially private streaming algorithm for distinct count with provable approximation guarantees to the best of our knowledge! We use this algorithm to estimate the number of groups produced and then use a pseudorandom function to partition the input database into smaller partitions such that the groups generated in each partition can fit into the private memory with high probability. Last, we present our differentially oblivious foreign key join algorithm based on oblivious sort (§3.3). We summarize the notations in Table 2.

Table 2: Notations used in this section

Notation	Description
Π	Projection operator
σ_ϕ	Filter operator with filtering predicate ϕ
ϵ	The multiplicative factor in differential obliviousness
δ	The additive factor in differential obliviousness
I	Input table of size N
P	A FIFO buffer in the private memory
c	Read counter in I
$\widetilde{Y}_c$	Differentially private prefix sum in first c elements
t	A single tuple from I
L	Consisting of grouping attributes and aggregation operators
M	Private enclave memory size
h	Hash function in DoGroup$_h$
$\widetilde{G}$	Estimated number of distinct elements in data stream

3.1 Selection with Projection (σ, Π)

A selection operator takes a relation and outputs a subset of the relation according to a filtering predicate. Such an operation is denoted $\sigma_\phi(R)$, where ϕ is the filtering predicate and R is the input table. Intuitively, selection operators act like filtering operations in functional programming languages. A projection operator transforms one relation into another, possibly with a different schema: it is written $(\Pi_{a_1,\ldots,a_n}(R))$ where $a_1, \ldots, a_n$ is a set of attribute names. The result of such a projection keeps components of the tuple defined by the set of projected attributes and discards the other attributes. In many database systems, projection is usually inlined in selection. We follow this tradition. Now, we give the differentially oblivious algorithm for $\sigma_\phi(\Pi(R))$, where ϕ is the filtering predicate and R is the input table. To better understand our algorithm, we start with a naïve non-oblivious algorithm:

Naïve non-oblivious algorithm. It is clear that a non-private filtering algorithm can achieve linear time, by reading each input tuple t once and writing it when $\phi(t) =$ TRUE. However, this naïve algorithm is not differentially oblivious. This is because after reading a tuple from input, whether or not another tuple is written to the output leaks whether the previous tuple from input evaluated to TRUE or FALSE. Intuitively, one can visualize the memory access pattern of this algorithm using two pointers, a read pointer and a write pointer. The attacker observes how fast these two pointers move in each step.

Thus, the main idea of our differentially oblivious filtering algorithm, DoFilter, is to obfuscate how fast each pointer advances *just enough* to achieve differential obliviousness. DoFilter is inspired by the theoretical result of differentially oblivious stable compaction from [23]. To determine how much noise to add on memory access at each step, we query a differentially private oracle for computing prefix sum in data streams.

Differentially private prefix-sum. For a data stream D that consists of only 0s and 1s with length $|D| = n$, $D \in \{0,1\}^n$, the prefix-sum Y_c is the count of how many 1s appear in the first c elements of data stream D. Now, suppose we have a $(\epsilon, \delta)-$differentially private prefix sum algorithm that can answer up to n queries, and each answer $\widetilde{Y}_c \in [Y_c - s, Y_c + s]$ with high probability. To make the traces of the write pointer differentially private, we can always move the output pointer to $\widetilde{Y}_c - s$, and keep the scanned but not yet output tuples in the private buffer. Most importantly, we only need a $2s$ sized buffer in private memory and the algorithm would not encounter errors with high probability.

We use the binary mechanism of Chan et al. [25] as our DP prefix-sum oracle. This mechanism essentially builds a binary interval tree to store noisy partial sums for the optimal approximation-privacy trade-off. For each $c \in [n]$, the estimated prefix-sum $\widetilde{Y}_c$ from the binary mechanism preserves ϵ-differential privacy while has $O(\epsilon^{-1} \cdot (\log T) \cdot \sqrt{\log t} \cdot \log(1/\delta)$ error with at least $1 - \delta$ probability [25, Theorem 3.5, 3.6].

Differentially Oblivious Filtering. We present the detailed DoFilter in Algorithm 1. Let I be the input table of length $|I| = N$. Let s be the approximation error (with probability at least $1 - \delta$ for each query) of the DP prefix-sum oracle (line 2). We create a FIFO buffer P in private memory with size $2s$, the output table I' outside the private memory, and a counter c to indicate the number of tuples read so far (line 3-5). Then we repeat the following until reaching the end of I: we read the next s tuples, and update the counter c (line 7-8). For each tuple t, we push it to P only if the predicate evaluates to TRUE (line 9-13). We then pop P to fill the output table I' till it reaches size $\widetilde{Y}_c - s$ (line 14). After we reach the end of I, we pop all the tuples in P and add filler tuples if necessary to append on I' till it reaches size $\widetilde{Y}_N$ where $N = |I|$ (line 16).

Algorithm 1 DoFilter: Differentially Oblivious Filtering

1: **procedure** DoFilter($I, \Pi, \phi, \epsilon, \delta, s$) ▷ Theorem 3.1.
2: $P \leftarrow \emptyset$ ▷ a FIFO buffer in private memory of size $2s$
3: $I' \leftarrow \emptyset$ ▷ output table
4: $c \leftarrow 0$ ▷ current read counter in I
5: $\widetilde{Y} \leftarrow$ DPPrefixSum(ϵ, δ) ▷ $\widetilde{Y}_c \in [Y_c - s, Y_c + s]$ with high probability
6: **while** $c < |I|$ **do**
7: $T \leftarrow \{I_c, I_{c+1}, \ldots I_{c+s-1}\}$ ▷ read the next s tuples
8: $c \leftarrow c + s$ ▷ update the read counter
9: **for** $t \in T$ **do**
10: **if** $\phi(t) =$ TRUE **then**
11: $P \leftarrow P$.push($\Pi(t)$)
12: **end if**
13: **end for**
14: Pop P to write I' until $|I'| = \widetilde{Y}_c - s$
15: **end while**
16: write all tuples from P and filler tuples to I' s.t. $|I'| = \widetilde{Y}_N + s$ ($N = |I|$)
17: **end procedure**

Correctness failures to privacy failures. Algorithm 1 is designed to have correctness failure (i.e. the algorithm does not return the correct result) of probability at most δ. This means that Algorithm 1 can fail when the DP prefix-sum oracle's estimation is off by more than s. When this happens, the size P private memory could either overflow at line 11 or underflow (i.e. has nothing to pop) at line 14. In practice, we do not need to worry about this for two reasons. First, δ is negligible (usually set to $2^{-20} - 2^{-40}$). Second, in case that users want perfect correctness, we can use the standard technique to convert the correctness failures to privacy failures: Instead of failing, if overflow is about to happen at line 11, we can simply write the overflowed tuple to the output. If underflow happens at line 14, we can write a filler tuple to the output. Applying these approaches converts the at most δ probability correctness error to at most δ probability privacy error, which is negligible.

Theorem 3.1 (Main result for filter)**.** *For any $\epsilon \in (0, 1)$, $\delta \in (0, 1)$ and input I with N tuples , there is an (ϵ, δ)-differentially oblivious filtering algorithm (DoFilter in Algorithm 1) that uses $O(\log(1/\epsilon) \cdot \log^{1.5} N \cdot \log(N/\delta))$ private memory and $(N+R)/B$ cache complexity, and its output size is $R + \text{poly}\log(N)$.*

Proof. First, we prove Algorithm 1 is $(\epsilon, 0)$-differential oblivious if P has infinite capacity: For two neighboring input I, I', assuming I, I', we only leaks $\widetilde{Y}_k$ ($k = s, 2s, \ldots, n$). This leakage is bounded by leaking all $\widetilde{Y}_i$ ($i \in [N]$). From the DP guarantee provided by the DP prefix sum oracle [24], all writes have at most $(\epsilon, 0)$-DP leakage.

Second, we prove Algorithm 1 has at most δ probability of privacy failure with $O(\log(1/\epsilon) \cdot \log^{1.5} N \cdot \log(N/\delta))$ private memory. Let $s = O(\log(1/\epsilon) \cdot \log^{1.5} N \cdot \log(N/\delta))$. We set $P = 2s$. Let Y_c denote number of actual filtered tuples generated so far (at line 14). From the DP guarantee provided by the DP prefix sum oracle, we know that for each c, $Y_c - s \le \widetilde{Y}_c \le Y_c + s$ with $1 - \frac{\delta}{N}$ probability. By the union bound, we know that for all rounds of batched read, with at least $1 - \delta$ probability, P over-flows or P under-flows:

$$\Pr[Y_c - (\widetilde{Y}_c - s) > 2s \vee Y_c < \widetilde{Y}_c - s] \ge 1 - \delta$$

Now we can conclude that the failure probability of Algorithm 1 is at most δ. In Algorithm 1, the number of tuples we read is N, and the number of tuples we write to the output is $\widetilde{Y}_N + s$. From the utility-privacy bound from the DP oracle [24], we know $\widetilde{Y}_N + s = R + \text{poly}\log(N)$. Therefore, the output size of the algorithm is $R + \text{poly}\log(N)$. In addition, all the read and write in Algorithm 1 are batched, with the batch size s. Thus, the cache complexity of Algorithm 1 is $(N + R)/B$ as long as $s \ge B$. □

Remark 3.2. Remark: Note that $\Omega((N + R)/B)$ cache complexity is a trivial lower bound. Thus, our cache complexity is optimal.

We adapted the technique in [23], which presents a DO stable compaction algorithm. Stable compaction is a different problem since it keeps all the elements in the input. Also, in [23], they don't have a notion of the cache complexity and therefore don't provide any bound for that.

3.2 Grouping with Aggregation (γ, α)

A grouping operator groups a relation and/or aggregates some columns. It usually denoted $\gamma_L(R)$ where L is the list which consists of two kinds of elements: grouping attributes, namely attributes of R by which R will be grouped, and aggregation operators applied to attributes of R. For example, $\gamma_{c_1,c_2,\alpha_1(c_3)}(R)$ partitions the tuples in R into groups according to attributes $\{c_1, c_2\}$, and outputs the aggregation value of α_3 on c_3 for each group.

We propose a non-oblivious hash based grouping based on randomized partitioning on grouping attributes. This can be done by applying a pseudorandom function (PRF) on the grouping attributes L. One key challenge is to make each partition fit into the private memory (size M). To obtain the correct parameter s for the randomized partitioning algorithm, we apply a preprocessing step. Specifically, we use a randomized streaming distinct count algorithm to get the estimated number of groups produced by this query $\widetilde{G}$. As a result, roughly we need $k = \lceil \widetilde{G}/M \rceil$ sequential scans to find all the groups. To make this algorithm differentially oblivious is yet another challenge. One first observation is: this algorithm is "almost" oblivious if we pad the output in each round to M, except that the number of sequential scans following the preprocessing step leaks information. As a result, we need to use a differentially private distinct count algorithm. Additionally, we need to bound the failure probability of the randomized partitioning algorithm, such that the size of each partition will not overflow M.

Unfortunately, despite few theoretical results [15, 26, 47], to the best of our knowledge there is no practical differentially private distinct count streaming algorithm. To remedy this, we propose a differentially private distinct count algorithm based on the classical distinct count estimator of Bar-Yossef et al. [12]. The core technique we leverage here is to use properties of uniform order statistics to bound the concentration of both the approximation error and the sensitivity at the same time. The general version of this differentially private distinct count algorithm and detailed proofs are in §4. And we use it to design our differentially oblivious grouping with aggregation algorithm.

Our differentially private distinct count algorithm (Algorithm 5) first creates a priority-queue P of size t in the private memory (line 2). Then, for each element x_i in the stream, our algorithm applies

Algorithm 2 DoGroup$_h$: Differentially Oblivious Grouping

1: **procedure** DoGroup$_h(I, L, \epsilon, \delta)$ ▷ Theorem 3.3
2: $\widetilde{G} \leftarrow$ DPDistinctCount$(I, \epsilon, 0.1, \delta/2)$ ▷ Algorithm 5 with approximation parameter $\eta = 0.1$
3: $k \leftarrow \lceil \widetilde{G}/0.9M \rceil$ ▷ M: size of the private memory
4: Verify that $\sqrt{0.5\widetilde{G}\log(2k/\delta)} \leq 0.1M$
5: $R \leftarrow \emptyset$ ▷ output table
6: **for** $i \in 0, \ldots, k-1$ **do**
7: $\mathcal{H} \leftarrow \emptyset$ ▷ Hash table for grouping
8: **for** $t \in I$ **do**
9: **if** $h(t.L) \in [i/k, (i+1)/k)$ **then**
10: ▷ $h : [m \times \ldots \times m] \leftarrow [0, 1)$, is a PRF
11: **if** $\mathcal{H}$.hasKey$(t.L)$ **then**
12: update $\mathcal{H}(t.L)$'s aggregate values using t
13: **else**
14: $\mathcal{H}(t.L) \leftarrow t$
15: **end if**
16: **end if**
17: **end for**
18: write all tuples in $\mathcal{H}$ and filler tuples to R, s.t. R's size increased by M
19: **end for**
20: **end procedure**

a PRF h to obtain a hash value of the element ($h(x_i) \in [0, 1)$). Our algorithm uses the priority-queue to keep the t smallest hash values of the stream (line 4 - 11). In the end, we pop P to get the t-th smallest hash value v (line 13). Finally, we output the estimated value of distinct count in line 14. The unbiased estimation should be t/v, as stated in [12]. Here, since the estimated value is used to calculate the number of partitions needed, we can only over estimate. We need to add proper noise to make the algorithm differentially private as well. As a result, the algorithm outputs the noisy count as shown in line 14. We will use this algorithm with approximation parameter $\eta = 0.1$ to design our differentially oblivious grouping algorithm.

In Algorithm 2, we present our differentially oblivious grouping algorithm. Let I be the input table. Let L be the list which consists of two kinds of elements: grouping attributes and aggregation operators. The algorithm first computes the 1.1-approximate differentially private distinct count $\widetilde{G}$ (line 2), and use $\widetilde{G}$ to calculate the number of partitions $k = \lceil \widetilde{G}/0.9M \rceil$ (line 3). We set this k value to overestimate the number of iterations we need in order to obtain a negligible failure probability of out-of-enclave memory error among k iterations. To ensure the size of each partition is less than M with at least $1 - \delta$ probability, we verify that $\sqrt{0.5\widetilde{G}\log(2k/\delta)} \leq 0.1M$ (line 4). Next, the algorithm sequentially scans I a total of k times. In i-th scan, the algorithm creates an empty hash table $\mathcal{H}$ (line 7). Then, for each tuple t, the algorithm applies a PRF h on the list of grouping attributes of t. Here $h(t.L)$. $h(t.L)$ falling into $[i/k, (i+1)/k)$ means this group is within the partitioned groups of current sequential scan. In this case, the algorithm either update the aggregate values if $\mathcal{H}$ already contains t, or create a new entry for t in $\mathcal{H}$ (line 9 - 16). In the end of each sequential scan, we output all groups in $\mathcal{H}$ and filler tuples so that size M is written to the output (line 18).

Theorem 3.3 (Main result for grouping). *For any $\epsilon \in (0, 1)$, $\delta \in (0, 1)$ and input I with size N and $O(\epsilon^{-1}\log^2(1/\delta))$ private memory M, there is an (ϵ, δ)-differentially oblivious and distance preserving grouping algorithm (DoGroup$_h$ in Algorithm 2) that uses M private memory, has $N/B + 11NR/9MB$ cache complexity and its output size is $\frac{11}{9}R$.*

Proof. Because at the end of each scan, we write to the output table until its size is increased by M and the enclave private memory size M is public, the only information that Algorithm 2 leaks is k, the number of sequential scans of I. This only leaks $\widetilde{G}$ though. Moreover, $\widetilde{G}$ is $(\epsilon, \delta/2)$-differentially private. And the differentially private distinct count algorithm is oblivious. Thus, Algorithm 2 is $(\epsilon, 0)$-differentially oblivious if we ignore the failure case.

Next, we prove that the failure probability of Algorithm 2 is at most δ. Since $\sqrt{0.5\widetilde{G}\log(2k/\delta)} \leq 0.1M$ (line 4) and the expected number of group generated in each sequential scan is $0.9M$, let G_i be the number of groups i-th sequential scan generated. From the properties of binomial distribution (detailed lemmas can be found in Appendix C), we have $\Pr[G_i \leq M] \leq \delta/2k$. Applying a union bound over k scans, we can bound the failure probability of the randomized partitioning by at most $\delta/2$. Applying union bound again with the (ϵ, δ)-differentially private $\widetilde{G}$, we can bound the failure probability of Algorithm 2 to at most δ.

Finally, Algorithm 2 requires a sequential scan of input in preprocessing, whose cache complexity is N/B. In the following steps, the cache complexity is $k(M+N)/B$, where $k \leq 1.1R/0.9M$, which is no more than $\frac{11R(M+N)}{9MB}$. Here M is absorbed by N. Thus, the overall cache complexity of Algorithm 2 is $N/B + 11NR/9MB$. Because we need to write M groups to the output for k passes, the output size is $\frac{11}{9}R$.

□

3.3 Foreign Key Join (⋈)

Foreign key join is the most widely used join operator in data analytics. We write $R \bowtie S$[2] to represent a join on the primary key and foreign key pairs of the relations R and S

A typical oblivious join algorithm first pads the tuples from two joined tables to the same size, and add a "mark" column to every tuple to mark which table is this tuple from. Then, it performs an oblivious sort on the concatenation of both joined tables. This oblivious sort routes the tuples to be joined from both tables to the same group. Next, the algorithm makes a sequential scan of the sorted table to generate the result table. This can be done obliviously since for each tuple read from the primary key table, the algorithm will output a filler tuple instead. Last, the algorithm uses another oblivious sort to remove all the filler tuples. This algorithm is first implemented in Opaque [83] and then followed by ObliDB [39].

We develop DoJoin, a differentially oblivious foreign key join algorithm. In DoJoin, the neighboring databases D_1 and D_2 should contain the same primary key table and their foreign-key tables have the same length but differ in one record. DoJoin improves the standard oblivious foreign key join in three aspects. First, we use the more efficient bucket oblivious sort [10] replacing the bitonic sort used in ObliDB. Compared with bitonic sort, bucket oblivious

[2] ⋈ is normally used for natural join, we abuse the notion here.

sort has better asymptotic complexity ($O(n \log n)$ compared with $O(n \log^2 n)$) and still relatively small constant 6. Second, our algorithm only sorts the input once, removing filler tuples is done by our differential oblivious filtering algorithm (Algorithm 1, §3.1). Lastly, our algorithm pads filler tuples in the output to the size of differential obliviousness requirement, rather than to the worse case size, which could be much smaller in practice.

Algorithm 3 DoJoin: DO Foreign Key Join

1: **procedure** DoJoin($R, S, k_R, k_S, \epsilon, s$) ▷ Theorem 3.4
2: ▷ k_R is the PK of R, k_S is a FK in S referring k_R
3: pad the size of each row of R and S to the greater row size of R and S
4: $R' \leftarrow \rho(\texttt{mark}(R, \text{'r'}), k_R \rightarrow k)$
5: $S' \leftarrow \rho(\texttt{mark}(S, \text{'s'}), k_S \rightarrow k)$
6: $I \leftarrow$ BucketObliviousSort($R'||S', k||mark$)
7: $t \leftarrow \perp$ ▷ Current tuple from R to be joined
8: $I' \leftarrow \emptyset$
9: **for** $x_i \in I$ **do**
10: **if** $x_i.mark =$ 'r' **then**
11: $t \leftarrow x_i$
12: $I' \leftarrow I'||\perp$ ▷ $\perp$ means filler tuple
13: **else** ▷ $x_i.mark =$'s'
14: $I' \leftarrow I'||(x_i \cup t)$
15: **end if**
16: **end for**
17: $T \leftarrow$ DoFilter(I', ID, $\lambda t.t \neq \perp, \epsilon, \delta$) ▷ remove filler tuples, Algorithm 1.
18: **return**
19: **end procedure**

We present DoJoin in Algorithm 3. Let R and S be the primary key and foreign key tables. k_R is the PK of R, k_S is a FK in S referring k_R. DoJoin first pads tuples from R and S to the same size and adds an additional "mark column" to each tuple to mark which relation it comes from. This result in R' and S' (line 3 - 5). Next, DoJoin concatenates R' and S' ($R'||S'$) and then sort the result first by the key column and then by the mark column. For tuples with the same key, the tuple from R' will always be read first (if it exists). Now, DoJoin sequentially scans the sorted table: if a tuple from R' is scanned, assign it to the working tuple, and output a filler tuple to the output (line 9 - 11); if a tuple from S' is scanned, we join it with the working tuple and write the joined tuple to the output (line 13). Lastly, we call DoFilter to remove the filler tuples.

Theorem 3.4 (Main result for join)**.** *For any $\epsilon \in (0,1)$, $\delta \in (0,1)$, input I with size N, private memory of size M and result size R, there is an (ϵ, δ)-differentially oblivious and distance preserving foreign key join algorithm (DoJoin in Algorithm 3) that uses $O(\log(1/\epsilon) \cdot \log^{1.5} N \cdot \log(N/\delta))$ private memory and has $6(N/B)\log(N/B) + (N+R)/B$ cache complexity, and its output size is $R + \text{poly}\log(N)$.*

Proof. DoJoin is (ϵ, δ)-differentially oblivious follows that BucketObliviousSort is fully oblivious and DoFilter is (ϵ, δ)-differentially oblivious with $O(\log(1/\epsilon) \cdot \log^{1.5} N \cdot \log(N/\delta))$ private memory.

DoJoin requires sorting $R'||S'$ obliviously once. It uses BucketObliviousSort, which has cache complexity of $6(N/B)\log N/B$. Additionally, the DoJoin algorithm uses the DoFilter algorithm which has a cache complexity $(N+R)/B$. Thus, the cache complexity of DoJoin is $6(N/B)\log(N/B) + (N+R)/B$. From the output size of DoFilter algorithm, we know the output size of DoJoin is $R + \text{poly}\log(N)$. □

4 DIFFERENTIALLY PRIVATE DISTINCT COUNT

In this section, we describe a differentially private distinct count algorithm based on [12]. We first prove the main technical lemmas about order statistics properties of random sampling in §4.1. Next, we define (ϵ, δ)-sensitivity and introduce the Laplacian mechanism for (ϵ, δ)-sensitivity in §4.2. This follows by our analysis of [12]: its (ϵ, δ)-sensitivity and its approximation ratio concentration. Last, we develop a differentially private distinct count algorithm (Algorithm 5) based on [12], which is used to implement DoGroup$_h$ (Algorithm 2) with approximation parameter $\eta = 0.1$.

4.1 Order Statistics Properties of Random Sampling

For any integer n, we use $[n]$ to denote the set $\{1, 2, \cdots, n\}$. Let $x_1, x_2, \ldots, x_n \sim [0,1]$ be independently and uniformly sampled, and let $x_{(1)} \leq x_{(2)} \leq \cdots \leq x_{(n)}$ be the order statistics of the samples $\{x_i\}_{i=1}^n$. For simplicity, we write $y_i = x_{(i)}$ to align with the notation above, so that y_i is the i-th smallest value in $\{x_i\}_{i=1}^n$. Fix any $1 \leq t \leq n/2$. Our goal is to prove that $|\frac{1}{y_t} - \frac{1}{y_{t+1}}| \leq O(\frac{n}{t^2})$ with large constant probability. We begin with the following simple claim which lower bounds y_t. We note that the bound improves for larger t, so one can use whichever of the two bounds is better for a given value of t. We delay the proof of Claim 4.1 to Section B.1.

Claim 4.1. *Let $t \in [n]$, and fix any $0 < \delta < 1/2$. Then we have the following two bounds:*

(1) $\Pr[y_t > \delta \frac{t}{n}] \geq 1 - \delta$.
(2) $\Pr[y_t > \frac{t}{2n}] \geq 1 - \exp(-t/6)$.

We now must lower bound y_{t+1}, which we do in the following claim. We delay the proof to Section B.1.

Claim 4.2. *Fix any $4 < \alpha < n/2$, and $1 \leq t \leq n/2$. Then we have*

$$\Pr[y_{t+1} < y_t + \alpha/n] \geq 1 - \exp(-\alpha/4).$$

We then want to bound $|\frac{1}{y_t} - \frac{1}{y_{t+1}}|$ in the following lemma and delay the proof to Section B.1.

Lemma 4.3. *Fix any $0 < \beta \leq 1/2$, $1 \leq t \leq n/2$, and α such that $4 < \alpha < \beta t/2$. Then we have the following two bounds:*

(1) $\Pr[|\frac{1}{y_t} - \frac{1}{y_{t+1}}| < \frac{\alpha}{\beta^2}\frac{n}{t^2}] \geq 1 - \beta - \exp(-\alpha/4)$.
(2) $\Pr[|\frac{1}{y_t} - \frac{1}{y_{t+1}}| < 4\alpha\frac{n}{t^2}] \geq 1 - \exp(-t/6) - \exp(-\alpha/4)$.

Remark 4.4. Notice that the above lemma is only useful when t is larger than some constant, otherwise the bounds $4 < \alpha < \delta t/2$ for $0 < \delta < 1/2$ will not be possible. Note that if we wanted bounds on $|\frac{1}{y_t} - \frac{1}{y_{t+1}}|$ for t smaller than some constant, such as $t = 1, 2$, ect. then one can simply bound $|\frac{1}{y_t} - \frac{1}{y_{t+1}}| < \frac{1}{y_t}$ and apply the results of Claim 4.1, which will be tight up to a (small) constant.

Algorithm 4 Distinct Count [12]

procedure DistinctCount(I, t) ▷ Lemma 4.9
  $d \leftarrow \emptyset$ ▷ d is a priority-queue of size t
  for $x_i \in I$ **do**
    $y \leftarrow h(x_i)$ ▷ h: $[m] \to [0,1]$, is a PRF
    if $|d| < t$ **then**
      d.push(y)
    else if $y < d$.top() $\wedge$ $y \notin d$ **then**
      d.pop()
      d.push(y)
    end if
  end for
  $v \leftarrow d$.top()
  return t/v
end procedure

Algorithm 5 DPDistinctCount: Differentially Private Distinct Count

procedure DPDistinctCount($I, \epsilon, \eta, \delta$) ▷ Theorem 4.11
  pQueue $\leftarrow \emptyset$ ▷ pQueue is a priority-queue of size t ▷ $t \geq \max\left(3(1+\eta/4)(\eta/4)^{-2}\log(6/\delta),\ 20\epsilon^{-1}(\eta/4)^{-1} \cdot \log(24(1+e^{-\epsilon})/\delta) \cdot \log(3/\delta)\right)$
  for $x_i \in I$ **do**
    $y \leftarrow h(x_i)$ ▷ h: $[m] \to [0,1]$, is a PRF
    if |pQueue| $< t$ **then**
      pQueue.push(y)
    else if $y <$ pQueue.top() $\wedge\ y \notin$ pQueue **then**
      pQueue.pop()
      pQueue.push(y)
    end if
  end for
  $v \leftarrow$ pQueue.top()
  $\widetilde{G} \leftarrow (1+\frac{3}{4}\eta)\frac{t}{v} + \mathrm{Lap}(20\epsilon^{-1}\frac{n}{t}\log(24(1+e^{-\epsilon})/\delta))$
  return $\widetilde{G}$
end procedure

4.2 (ϵ, δ)-Sensitivity

In what follows, let $\mathcal{X}$ be the set of databases, and say that two databases $X, X' \in \mathcal{X}$ are neighbors if $\|X - X'\|_1 \leq 1$.

Definition 4.5 (ℓ-sensitivity [37])**.** Let $f : \mathcal{X} \to \mathbb{R}$ be a function. We say that f is ℓ-sensitivity if for every two neighboring databases $X, X' \in \mathcal{X}$, we have $|f(X) - f(X')| \leq \ell$.

Theorem 4.6 (The Laplace Mechanism [36])**.** *Let $f : \mathcal{X} \to \mathbb{R}$ be a function that is ℓ-sensitive. Then the algorithm A that on input X outputs $A(X) = f(X) + Lap(0, \ell/\epsilon)$ preserves $(\epsilon, 0)$-differential privacy.*

In other words, we have $\Pr[A(X) \in S] = (1 \pm \epsilon)\Pr[A(X') \in S]$ for any subset S of outputs and neighboring data-sets $X, X' \in \mathcal{X}$. We now introduce a small generalization of pure sensitivity (Definition 4.5), that allows the algorithm to *not* be sensitive with a very small probability δ. The difference between ℓ-sensitivity and (ℓ, δ)-sensitivity is precisely analogous to the difference between ϵ-differential privacy and (ϵ, δ)-differential privacy, where in the latter we only require the guarantee to hold on a $1 - \delta$ fraction of the probability space. Thus, to achieve (ϵ, δ)-differential privacy (as is our goal), one only needs the weaker (ℓ, δ) sensitivity bounds.

Definition 4.7 ((ℓ, δ)-sensitive)**.** Fix a randomized algorithm $\mathcal{A} : \mathcal{X} \times R \to \mathbb{R}$ which takes a database $X \in \mathcal{X}$ and a random string $r \in R$, where $R = \{0,1\}^m$ and m is the number of random bits used. We say that $\mathcal{A}$ is (ℓ, δ)-sensitive if for every $X \in \mathcal{X}$ there is a subset $R_X \subset R$ with $|R_X| > (1-\delta)|R|$ such that for any neighboring datasets $X, X' \in \mathcal{X}$ and any $r \in R_X$ we have $|\mathcal{A}(X, r) - \mathcal{A}(X', r)| \leq \ell$

Notice that our algorithm for count-distinct is $(O(\alpha\frac{n}{t}), O(e^{-t} + e^{-\alpha}))$-sensitive, following from the technical lemmas proved above. We now show that this property is enough to satisfy (ϵ, δ)-differential privacy after using the Laplacian mechanism.

Lemma 4.8. *Fix a randomized algorithm $\mathcal{A} : \mathcal{X} \times R \to \mathbb{R}$ that is (ℓ, δ)-sensitive. Then consider the randomized laplace mechanism $\overline{\mathcal{A}}$ which on input X outputs $\mathcal{A}(X, r) + Lap(0, \ell/\epsilon)$ where $r \sim R$ is uniformly random string. Then the algorithm $\overline{\mathcal{A}}$ is $(\epsilon, 2(1+e^{\epsilon})\delta)$-differentially private.*

The proof is delayed to Section B.2.

4.3 Analysis of Distinct Count

In this section, we thoroughly analyze the properties of Distinct Count [12]. We first describe the algorithm in Algorithm 4. Then we prove its (ℓ, δ)-sensitivity in Lemma 4.9 and a tighter (ϵ, δ)-approximation result in Lemma 4.10 (compared with the approximation result in [12]).

Sensitivity of distinct count. By analyzing the Distinct Count Algorithm 4, we show that it is $(20\log(4/\delta)\frac{n}{t}, \delta)$-sensitive in Lemma 4.9. We will use its sensitivity to design our differential private distinct count algorithm 5.

Lemma 4.9 (Sensitivity of DistinctCount)**.** *Assume $r \in R$ is the source of randomness of the PRF in DistinctCount (Algorithm 4), where $R \in \{0,1\}^m$, n is the number of distinct element of the input, for any $16 < t < n/2$, DistinctCount is $(20\log(4/\delta)\frac{n}{t}, \delta)$-sensitive.*

The proof is delayed to Section B.3.

Lemma for approximation guarantees. Then we show the approximation guarantees for the Distinct Count with high probability in Lemma 4.10.

Lemma 4.10. *Let $x_1, x_2, \ldots, x_n \sim [0,1]$ be uniform random variables, and let $y_1, y_2, \ldots, y_n$ be their order statistics; namely, y_i is the i-th smallest value in $\{x_j\}_{j=1}^n$. Fix $\eta \in (0, 1/2), \delta \in (0, 1/2)$. Then if $t > 3(1+\eta)\eta^{-2}\log(2/\delta)$, with probability $1 - \delta$ we have*

$$(1-\eta)\cdot n \leq \frac{t}{y_t} \leq (1+\eta)\cdot n.$$

The proof is delayed to Section B.3.

4.4 Differentially Private Distinct Count

We present our main result for differentially private distinct count algorithm below:

Theorem 4.11 (main result)**.** *For any $0 < \epsilon < 1$, $0 < \eta < 1/2$, $0 < \delta < 1/2$, there is an distinct count algorithm (Algorithm 5) such that:*

(1) *The algorithm is* (ϵ, δ)*-differentially private.*
(2) *With probability at least* $1-\delta$*, the estimated distinct count* $\widetilde{A}$ *satisfies:*

$$n \le \widetilde{G} \le (1+\eta)\cdot n,$$

where n *is the number of distinct elements in the data stream.*

The space used by the distinct count algorithm is

$$O\Big((\eta^{-2}+\epsilon^{-1}\eta^{-1}\log(1/\delta))\cdot\log(1/\delta)\cdot\log n\Big)$$

bits.

The proof is delayed to Section B.4.

Claim 4.12. *For any* $0<\delta\le 10^{-3}$, $0.1\le\eta<1$ *and* $0<\epsilon<1$, *then we have*

$$3(1+\eta/4)\cdot(\eta/4)^{-2}\cdot\log(6/\delta)$$
$$\le 25\epsilon^{-1}(\eta/4)^{-1}\cdot\log(24(1+e^{-\epsilon})/\delta)\cdot\log(3/\delta).$$

The proof is delayed to Section B.4.

Lemma 4.13. *For any* $0<\epsilon<1$, $0<\delta\le 10^{-3}$, *there is an distinct count algorithm (Algorithm 5) such that:*

(1) *The algorithm is* (ϵ, δ)*-differentially private.*
(2) *With probability at least* $1-\delta$*, the estimated distinct count* $\widetilde{G}$ *satisfies:*

$$n \le \widetilde{G} \le 1.1n,$$

where n *is the number of distinct elements in the data stream.*

The space used by the distinct count algorithm is

$$O\Big((100+10\epsilon^{-1}\log(1/\delta))\cdot\log(1/\delta)\cdot\log n\Big)$$

bits.

The proof is delayed to Section B.4.

5 MEASURING EMPIRICAL SPEEDUP

We evaluate our DO operators on Big Data Benchmark [6]. We compare our performance with ObliDB [39], and Spark SQL [9]. We run our experiments on a machine with Intel Core-i7 9700 (8 cores @ 3.00GHz, 12 MB cache). The machine has SGX hardware and 64GB DDR4 RAM, and it runs Ubuntu 18.04 with SGX Driver version 2.6, SGX PSW version 2.9, and SGX SDK version 2.9. We set the SGX max heap size as 224 MB and EPC page swapping will be triggered during processing large tables. We fill data from the Big Data Benchmark [6]. We evaluate the performance under three tiers of input table sizes: For filter operator benchmark, *Rankings* table contains small (100K), medium (1M), large (10M) rows and each *Rankings* row is 308 bytes. For groupby operator benchmark, *UserVisits* table contains small (300K), medium (3M) and large (30M) rows and each *UserVisits* row is 529 bytes. For foreign key join benchmark, *Rankings* and *UserVisits* contain small (100K, 300K), medium (300K, 900K), large (1M, 3M) rows. All codebases are compiled and run under SGX prerelease and hardware mode. We do not compare our operators with those in Opaque [83]. This is because Opaque's open-sourced version does not pad the result of an operator to the worse-case length, which means Opaque's open-sourced version does not satisfy the notion of full obliviousness.

Setting Privacy Parameters We set ϵ to 1 and δ to 2^{-30}, which is negligible small (e.g. $\delta < 1/N$, where N is the size of the data). These settings follows the standard privacy settings in differentially private systems such as PINQ [58], Vuvuzela [76], and RAPPOR [38]. To further optimize the privacy parameters, we use numeric simulation to calculate a tighter bound of the differentially private mechanism when possible. For example, for the binary mechanism [24] that we used as a DP oracle in Algorithm 1, we can simulate its approximation error by repeating random trials of sum of Laplace noises. Figure 4 shows the simulation result. We can observe that the sum of independently sampled Laplace noises grows linearly as the $\frac{1}{\delta}$ grows exponentially. We can estimate the error by assuming linear growth of $|Y|$ over $\frac{1}{\delta}$'s exponentially growth when δ is too small to simulate.

5.1 Comparison to Prior Work

We now evaluate our three DO operators: selection with projection, grouping with aggregation, and foreign key join. The Big Data Benchmark [6] directly contains benchmarks to evaluate selection with projection and grouping with aggregation. Our DO operators only insert dummy tuples to the results to ensure that our memory access pattern is differentially private and do not add noise to the query results. Hence the accuracy of returned results is not affected.

Benchmark #1: Selection with projection:

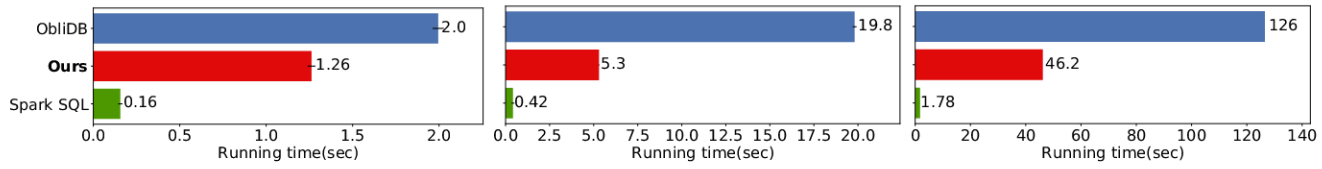
```
SELECT pageURL, pageRank
FROM rankings
WHERE pageRank > 1000
```

The first benchmark performs a selection with projection on *rankings* table. We compare the performance of ours with Spark SQL, ObliDB in Figure 1, Figure 2, and Figure 3. Compared with non-encrypted and non-oblivious spark SQL, for moderately large size datasets, ours exhibits 7.8 – 26.0x overhead. As shown in §5.2, our overhead mostly comes from encryption and decryption when moving data in and out of SGX enclave memory. The performance gain comes from the batched read and write implemented in our system. Compared with oblivious systems, we are 1.5 – 3.7× faster than ObliDB in benchmark 1. This performance gain comes from more efficient algorithm and the less padding size brought by the differential obliviousness.

Benchmark #2: Grouping with aggregation:

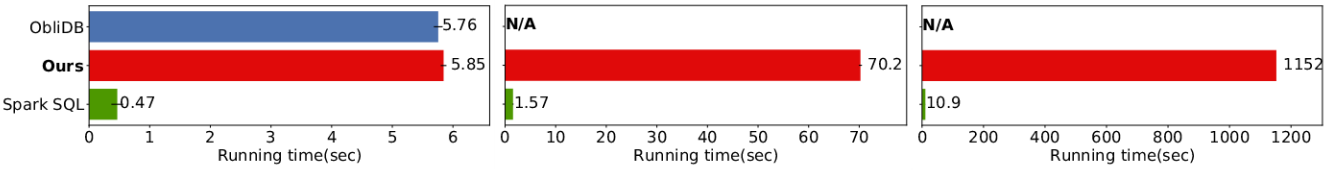
```
SELECT SUBSTR(sourceIP, 1, 8), SUM(adRevenue)
FROM uservisits
GROUP BY SUBSTR(sourceIP, 1, 8)
```

The second benchmark aggregates the sum of *adRevenue* based on their *sourceIP* column over *UserVisits* table. Compared with non-encrypted and non-oblivious spark SQL, for moderately large size datasets(300K - 30M), ours exhibits 12.4 – 105.7x overhead. For grouping with aggregation over *UserVisits* table of 300K rows, ours has similar performance with ObliDB. However, ObliDB's grouping operator assumes that the aggregation statistics of all the distinct groups (up to 400,000 under current SGX enclave memory capacity) can fit in enclave memory so that it can calculate the aggregation results in just one pass, but this assumption does not hold for *UserVisits* table of 3 million rows and more. ObliDB fails to run grouping with aggregation over *UserVisits* table of 30 million rows



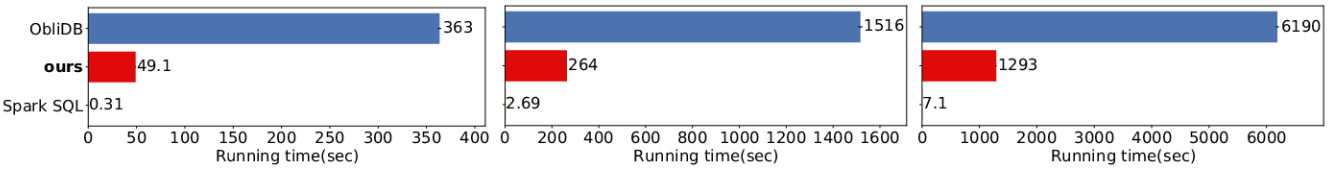
(a) Selection with projection-100K (b) Selection with projection-1M (c) Selection with projection-10M

Figure 1: Selection with projection operator performance under different input sizes. Error bars show the standard deviations.



(a) Grouping with aggregation-300K (b) Grouping with aggregation-3M (c) Grouping with aggregation-30M

Figure 2: Grouping with aggregation operator performance under different input sizes. Error bars show the standard deviations.



(a) Foreign key join-300K (b) Foreign key join-900K (c) Foreign key join-3M

Figure 3: Foreign key join operator performance under different input table sizes. Error bars show the standard deviations.

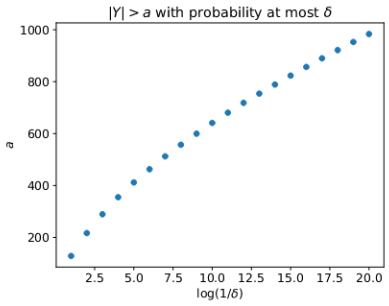


Figure 4: Simulated Binary Mechanism Concentration ($\epsilon = 1$, $N = 10^9$, each data point uses $10^4/\delta$ trials)

under our hardware settings too. As the number of distinct groups grows, ours has to process aggregation query in more passes, which is another source of overhead to achieve differentially oblivious grouping with aggregation.

BDB does not contain a benchmark that directly evaluate foreign key join. BDB has a complex benchmark that requires composing a series of database operators. Composing DO operators is beyond the scope of this paper. Here, we use a simplified benchmark to evaluate foreign key join.

Benchmark #3: Foreign key join:

```
SELECT *
FROM Rankings AS R, UserVisits AS UV
WHERE R.pageURL = UV.destURL
```

The third benchmark is to do foreign key join between *Rankings* and *UserVisits*. Ours exhibits 98.1 – 182.1x overhead over Spark SQL, but it is 4.8 – 7.4× faster than ObliDB in benchmark 3. As stated before, this performance gain mainly comes from less dummy writes to achieve differential obliviousness compared to full obliviousness. Bucket oblivious sort achieves $O((N/B)\log_{M/B}(N/B))$ number of page swaps and bitonic sort requires $O(N\log^2 N)$ page swaps if implemented naively. The practical speedup we see is $5-7\times$ partly because bitonic has a smaller constant in the big-O.

5.2 Latency Breakdown

It is interesting to understand where are the key performance bottlenecks in our differentially oblivious operators. We break down each of our basic operator's completion time into six categories: (1) decryption within enclave; (2) encryption within enclave; (3)

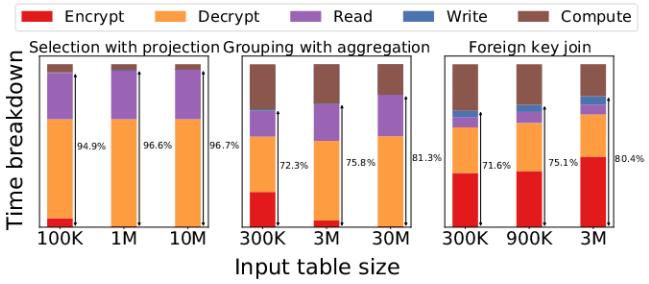


Figure 5: DO operators performance breakdown

reading from untrusted memory to enclave buffer; (4) writing from enclave buffer to untrusted memory; and (5) computation within enclave. We use Read Time-Stamp Counter (RDTSC) to profile the time spent on each category. Figure 5 shows the performance breakdown results. Encryption, decryption, memory copy between untrusted memory and enclave memory are the major overheads in our differentially oblivious operators. Under the largest input table size scenario, the real query computation time only accounts for 2% of the total execution time of the filter operator. Memory copy between untrusted memory and enclave memory accounts for 30% of the total time and encryption plus decryption take up the rest 68%. Because applying hash function to distinguish different groups in grouping with aggregation operator and oblivious sorting in foreign key join operator are more expensive than the simple comparison in filter operator, the computation constitutes a larger fraction in the operator execution time.

For the encryption time portion, we find that it decreases as the input table size increases in DoGroup$_h$. This is because the number of groups increases sublinearly as the size of input table increases when we group by the first 8 bytes of IP address in BDB2 benchmark. The encryption time portion increases as the input table size increases in DoJoin, because bucket oblivious sort is the dominant overhead and its cache complexity is $O((N/B)\log_{M/B}(N/B))$ and the overhead from compute part grows linearly to the size of primary and foreign key tables.

Moving data between the enclave and untrusted memory incurs overhead from both SGX ECALL/OCALL and encryption and decryption within the enclave. These results validate that the data movement between the trusted and untrusted components is the key bottleneck for our operators, which justify the usage of cache complexity and output size as the key theoretical performance metrics for the analysis of oblivious operators.

Table 3: Time percentage spent on inserting padding tuples.

Dataset Size	DoFilter	DoGroup$_h$	DoJoin
Small	51.0%	28.5%	26.3%
Medium	12.5%	2.9%	40.1%
Large	1.7%	0.4%	47.5%

Cost of tuple padding procedure. We further profile how much percentage of overall execution time is spent for inserting tuple padding. As shown in Table 3, for DoFilter and DoGroup$_h$ operators, the time portion spent on tuple padding decreases when the size of input table grows, and the time portion is smaller than 2% when the input table size is large. The reason is that DoFilter and DoGroup$_h$ do not need to do worst-case padding like full oblivious guarantee and the ratio of number of padding tuples and size of input decreases when the size of input table becomes larger. For DoJoin, the percentage of tuple padding procedure grows when when the size of input table increases, because in the bucket oblivious sort at the end of each iteration of oblivious random bucket assignment, we need to pad each bucket with dummy tuples until full. The number of padding tuples needed in oblivious random bucket assignment grows faster than linear.

6 RELATED WORK

Encrypted Databases. There are a series of encrypted database systems uses standard or customized encryption schemes. For example, CryptDB [66] uses a multi-layer encryption scheme to allow user to set different security levels for different columns. Arx [65] uses strong encryption and applies special data structures to enable search. Other systems [19, 20, 34**?**] build on searchable encryption techniques. All these systems only encrypt data, not access patterns. As a result, they are all vulnerable to access pattern attacks. Recently, there are many new database systems based on hardware enclaves, such as TrustedDB [11], Cipherbase [8], EnclaveDB [67], VC3 [71], VeriDB [85] and StealthDB[44]. These systems all leave data outside enclaves encrypted. However, these systems either only support data that can fit into very limited enclave memory (128MB in case of Intel SGX), such as EnclaveDB, or vulnerable to memory access pattern attacks.

Oblivious Databases. To address the vulnerability to access pattern attacks, recent data analytic systems like Opaque [83] and ObliDB [39] proposed and implemented a few database query processing algorithms that are fully oblivious. However, there are significant performance penalties of their oblivious modes compared to the non-oblivious or partial-oblivious (but encrypted) counterparts. Obladi [29] focuses on providing ACID transactions; federated oblivious database systems [13, 14, 31, 78] provide cooperative data analytics for untrusted parties (semi-honest or malicious). Oblix [61] is an oblivious search index whose internal memory access is also oblivious. Shrinkwrap [14] uses fully oblivious operators but padding with DP guarantees, and this greatly reduces its intermediate query results sizes. We are the first work to demonstrate the theoretical and empirical performance of differentially oblivious database operators.

ORAM and Oblivious Algorithms. Oblivious RAM and oblivious computation were proposed in the seminar work by Goldreich [42]. Since then, various ORAM schemes and hardware implementations were proposed, such as Path ORAM [74], Ring ORAM [68], and PrORAM [80]. Despite these exciting advances, ORAM still suffers from a $\log(N)$ factor slow down. For database that potentially has billions of tuples, this overhead is significant. In addition, using ORAM while leaking the runtime or result length does not provide full obliviousness. GhostRider [56] provides an FPGA-based implementation to ensure memory-trace obliviousness by employing ORAM. ZeroTrace [70] is a library of oblivious memory primitives for SGX enclave against side-channel attacks. Obfuscuro [3] leverage ORAM

operations to perform secure code execution and data access, and ensures that the program always runs for a pre-configured time interval Apart from ORAM, many other oblivious data structures have been proposed, such as oblivious priority queues [50, 72]. Apart from differential obliviousness [23], Allen et al. [5] proposed a security model, ODP, which combines differential obliviousness and differential privacy. This model is useful when both the published result and the memory access pattern need to be protected.

Other Ways of Mitigating SGX Side-channel Vulnerability. DR.SGX [21] designs and implements a compiler-based tool that instruments the enclave code, permuting data locations at fine granularity. By periodically re-randomizing all enclave data, DR.SGX can prevent correlation of repeated memory accesses. T-SGX [73] ensures that no page fault sequence will be leaked to attackers via Intel Transactional Synchronization Extensions (TSX) in order to mitigate the memory side channel attacks.

Differential Privacy. Another related development is differential privacy. Since its introduction [36], differential privacy has become the de facto standard for protecting user privacy. Many differential privacy data analytics systems have been developed, such as PINQ [58], FLEX [51], GUPT [62], PrivateSQL [55]. In this paper, we use a differentially private prefix-sum algorithm [24] as a building block of our differentially oblivious filtering algorithm. Additionally, we uses two established theoretical results in differential privacy, the group privacy theorem [75] and the basic composition [37].

7 DISCUSSION

Our paper presents the first step towards using different obliviousness in databases. Although several theoretical papers have already been moving in this direction [23, 27], our paper is the first one that have designed and implemented database operators and show their empirical speedup against fully oblivious operators. The result is promising: we show that differentially oblivious operators can deliver up to 7.4 × performance improvement. Now, one interesting question is how far away we are from an end-to-end differentially oblivious database. This is admittedly our original goal for the project, however, we have encountered substantial challenges. We want to leave them as future works for the research community.

Operator Composition. A complex SQL query needs to combine multiple operators. DO operators are defined on two neighboring databases. Let's imagine we want to apply a differentially oblivious operator $\mathcal{M}_2$ to the outcome of another differentially oblivious operator $\mathcal{M}_1$. To ensure differential obliviousness end-to-end, we need to make sure $\mathcal{M}_1$ is distance-preserving. We say that an operator is distance preserving, iff when applying the operator to two neighboring databases, the two output databases are still neighboring databases. This is required because $\mathcal{M}_2$'s obliviousness guarantee depends on the inputs to $\mathcal{M}_2$ to be neighboring databases. It is unclear how to build database operators that are both distance preserving and DO. Operators such as join are particularly challenging because join can increase distance. To date, there is only a theoretical work [84] that is able to compose DO database operators but there is a long way towards practical DO composability.

Query Optimization. When we have more differentially oblivious operators in the future (e.g,. sort-based grouping with aggregation, hash-based join) and need to run multiple operators to serve one SQL query, we need to choose which operator to use to accelerate query execution. For example, how do we choose sort-based grouping with aggregation or hash-based grouping with aggregation for a given query, and how to generate the optimal query execution plan will be another interesting problem to solve.

Access Patterns for Private Memory. Our design patches the side channel of the access pattern leakage for the public memory. The hardware we implement our algorithms on, Intel SGX, has known vulnerabilities for the access pattern leakage for the private memory, and this can also leak sensitive information. Specifically, popular commodity processors (even the ones with secure enclaves such as Intel SGX) allow time-sharing of the same on-chip cache among different processes. This leads to a series practical cache-timing attacks [16, 35, 69, 81, 82]. Fortunately, we can harden our implementation against cache-timing attacks without dramatic changes. The recipe is to make the algorithms and data structures within private memory oblivious as well. For example, we can change our implementation of bucket oblivious sort (in DoGroup$_s$ and DoJoin) so that it is oblivious within private memory. We can also use oblivious priority queues such as [72] to implement the priority queue in Algorithm 5. Our comparison with ObliDB is fair: both ObliDB and our implementation do not consider private enclave memory access pattern leakage.

8 CONCLUSION

Preventing data leakage in cloud databases has become a critical problem. Leveraging secure execution in hardware enclaves, such as Intel SGX, is not enough to prevent an attacker from breaking data confidentiality by observing the access patterns of encrypted data. Ensuring oblivious access patterns can lead to substantial performance overheads. In this paper, we study how to incorporate into databases one new notion of obliviousness, *differential obliviousness*, a novel obliviousness property which ensures that memory access patterns satisfy differential privacy. We design and implement **Adore**: **A** set of **D**ifferentially **O**blivious **RE**lational database operators, and we formally prove that they satisfy the notion of differential obliviousness. Our evaluations show that our differentially oblivious operators outperform the state-of-the-art fully oblivious databases by up to 7.4× on Big Data Benchmark dataset with the same hardware configuration.

ACKNOWLEGMENTS

The authors would like to thank Bolin Ding, Cong Yan, Derek Leung for helpful discussions and the valuable suggestions from anonymous reviewers.

REFERENCES

[1] [n.d.]. Azure confidential computing. https://azure.microsoft.com/en-us/solutions/confidential-compute/. Accessed: 2020-09-10.

[2] Mohamed Ahmed Abdelraheem, Tobias Andersson, and Christian Gehrmann. 2017. Inference and Record-Injection Attacks on Searchable Encrypted Relational Databases. *IACR Cryptol. ePrint Arch.* 2017 (2017), 24.

[3] Adil Ahmad, Byunggill Joe, Yuan Xiao, Yinqian Zhang, Insik Shin, and Byoungyoung Lee. 2019. Obfuscuro: A commodity obfuscation engine on intel sgx. In *Network and Distributed System Security Symposium*.

[4] Alibaba. [n.d.]. Alibaba ECS baremetal instance document. https://www.alibabacloud.com/help/doc-detail/108507.htm. Accessed: 2020-09-10.

[5] Joshua Allen, Bolin Ding, Janardhan Kulkarni, Harsha Nori, Olga Ohrimenko, and Sergey Yekhanin. 2019. An Algorithmic Framework For Differentially Private Data Analysis on Trusted Processors. In *NeurIPS*. 13635–13646.

[6] UC Berkeley AMP Lab. [n.d.]. Big Data Benchmark. https://amplab.cs.berkeley.edu/benchmark/.

[7] Panagiotis Antonopoulos, Arvind Arasu, Kunal D. Singh, Ken Eguro, Nitish Gupta, Rajat Jain, Raghav Kaushik, Hanuma Kodavalla, Donald Kossmann, Nikolas Ogg, Ravi Ramamurthy, Jakub Szymaszek, Jeffrey Trimmer, Kapil Vaswani, Ramarathnam Venkatesan, and Mike Zwilling. 2020. Azure SQL Database Always Encrypted. In *SIGMOD*. 1511–1525.

[8] Arvind Arasu, Spyros Blanas, Ken Eguro, Manas Joglekar, Raghav Kaushik, Donald Kossmann, Ravishankar Ramamurthy, Prasang Upadhyaya, and Ramarathnam Venkatesan. 2013. Secure database-as-a-service with Cipherbase. In *SIGMOD*. 1033–1036.

[9] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *SIGMOD*. 1383–1394.

[10] Gilad Asharov, T.-H. Hubert Chan, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. 2020. Bucket Oblivious Sort: An Extremely Simple Oblivious Sort. In *SOSA@SODA*. 8–14.

[11] Sumeet Bajaj and Radu Sion. 2013. TrustedDB: A trusted hardware-based database with privacy and data confidentiality. *IEEE Transactions on Knowledge and Data Engineering* 26, 3 (2013), 752–765.

[12] Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, D. Sivakumar, and Luca Trevisan. 2002. Counting Distinct Elements in a Data Stream. In *RANDOM*. 1–10.

[13] Johes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel Kho, and Jennie Rogers. 2017. SMCQL: secure querying for federated databases. *VLDB* (2017), 673–684.

[14] Johes Bater, Xi He, William Ehrich, Ashwin Machanavajjhala, and Jennie Rogers. 2018. Shrinkwrap: Efficient SQL Query Processing in Differentially Private Data Federations. *VLDB* 12, 3 (2018), 307–320.

[15] Omri Ben-Eliezer, Rajesh Jayaram, David P. Woodruff, and Eylon Yogev. 2020. A Framework for Adversarially Robust Streaming Algorithms. In *PODS*. 63–80.

[16] Daniel J Bernstein. 2005. Cache-timing attacks on AES. https://cr.yp.to/antiforgery/cachetiming-20050414.pdf. (2005).

[17] Sergei Bernstein. 1924. On a modification of Chebyshev's inequality and of the error formula of Laplace. *Ann. Sci. Inst. Sav. Ukraine, Sect. Math* 1, 4 (1924), 38–49.

[18] Chris Bing. [n.d.]. Atos, IT provider for Winter Olympics, hacked months before Opening Ceremony cyberattack. https://www.cyberscoop.com/atos-olympics-hack-olympic-destroyer-malwa\re-peyongchang/. Accessed: 2020-09-10.

[19] Raphael Bost. 2016. $\sum o\varphi o\varsigma$: Forward Secure Searchable Encryption. In *CCS*. 1143–1154.

[20] Raphaël Bost, Brice Minaud, and Olga Ohrimenko. 2017. Forward and Backward Private Searchable Encryption from Constrained Cryptographic Primitives. In *CCS*. 1465–1482.

[21] Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, and Ahmad-Reza Sadeghi. 2019. DR.SGX: Automated and Adjustable Side-Channel Protection for SGX Using Data Location Randomization. In *Proceedings of the 35th Annual Computer Security Applications Conference* (San Juan, Puerto Rico, USA) *(ACSAC '19)*. Association for Computing Machinery, New York, NY, USA, 788–800.

[22] Brandon Butler. [n.d.]. NSA spying fiasco sending customers overseas. https://www.computerworld.com/article/2484894/nsa-spying-fiasco-sending-customers-overseas.html. Accessed: 2020-09-10.

[23] T.-H. Hubert Chan, Kai-Min Chung, Bruce M. Maggs, and Elaine Shi. 2019. Foundations of Differentially Oblivious Algorithms. In *SODA*. 2448–2467.

[24] T.-H. Hubert Chan, Elaine Shi, and Dawn Song. 2010. Private and Continual Release of Statistics. In *ICALP*. 405–417.

[25] T.-H. Hubert Chan, Elaine Shi, and Dawn Song. 2011. Private and Continual Release of Statistics. *ACM Trans. Inf. Syst. Secur.* 14, 3 (2011), 26:1–26:24.

[26] Lijie Chen, Badih Ghazi, Ravi Kumar, and Pasin Manurangsi. 2020. On Distributed Differential Privacy and Counting Distinct Elements. (2020).

[27] Shumo Chu, Danyang Zhuo, Elaine Shi, and T-H. Hubert Chan. 2021. Differentially Oblivious Database Joins: Overcoming the Worst-Case Curse of Fully Oblivious Algorithms. In *The Second Information-Theoretic Cryptography (ITC) Conference*.

[28] Victor Costan, Ilia A. Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *USENIX Security*. 857–874.

[29] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. 2018. Obladi: Oblivious Serializable Transactions in the Cloud. In *OSDI*. USENIX Association, 727–743.

[30] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *SIGMOD*. 215–226.

[31] Ankur Dave, Chester Leung, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. 2020. Oblivious coopetitive analytics using hardware enclaves. In *EuroSys*. 39:1–39:17.

[32] Jessica Davis. [n.d.]. Inadequate Security, Policies Led to LifeLabs Data Breach of 15M Patients. https://healthitsecurity.com/news/inadequate-security-policies-led-to-lifela\bs-data-breach-of-15m-patients. Accessed: 2020-09-10.

[33] Jessica Davis. [n.d.]. Magellan Health Data Breach Victim Tally Reaches 365K Patients. https://healthitsecurity.com/news/magellan-health-data-breach-victim-tally-\reaches-365k-patients. Accessed: 2020-09-10.

[34] Ioannis Demertzis, Stavros Papadopoulos, Odysseas Papapetrou, Antonios Deligiannakis, and Minos N. Garofalakis. 2016. Practical Private Range Search Revisited. In *SIGMOD*. 185–198.

[35] John Demme, Robert Martin, Adam Waksman, and Simha Sethumadhavan. 2012. Side-channel vulnerability factor: A metric for measuring information leakage. In *ISCA*. 106–117.

[36] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam D. Smith. 2006. Calibrating Noise to Sensitivity in Private Data Analysis. In *TCC*. 265–284.

[37] Cynthia Dwork and Aaron Roth. 2014. The Algorithmic Foundations of Differential Privacy. *Foundations and Trends in Theoretical Computer Science* 9, 3-4 (2014), 211–407.

[38] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. 2014. RAPPOR: Randomized Aggregatable Privacy-Preserving Ordinal Response. In *CCS*. 1054–1067.

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

[40] Matthieu Giraud, Alexandre Anzala-Yamajako, Olivier Bernard, and Pascal Lafourcade. 2017. Practical Passive Leakage-abuse Attacks Against Symmetric Searchable Encryption. In *SECRYPT*. 200–211.

[41] Vindu Goel and Nicole Perlroth. [n.d.]. Yahoo Says 1 Billion User Accounts Were Hacked. https://www.nytimes.com/2016/12/14/technology/yahoo-hack.html. Accessed: 2020-09-10.

[42] Oded Goldreich. 1987. Towards a Theory of Software Protection and Simulation by Oblivious RAMs. In *STOC*. 182–194.

[43] Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *J. ACM* 43, 3 (1996), 431–473.

[44] Alexey Gribov, Dhinakaran Vinayagamurthy, and Sergey Gorbunov. 2017. Stealthdb: a scalable encrypted database with full sql query support. *arXiv preprint arXiv:1711.02279* (2017).

[45] Paul Grubbs, Richard McPherson, Muhammad Naveed, Thomas Ristenpart, and Vitaly Shmatikov. 2016. Breaking Web Applications Built On Top of Encrypted Data. In *CCS*. 1353–1364.

[46] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon Redshift and the Case for Simpler Data Warehouses. In *SIGMOD*. 1917–1923.

[47] Avinatan Hassidim, Haim Kaplan, Yishay Mansour, Yossi Matias, and Uri Stemmer. 2020. Adversarially Robust Streaming Algorithms via Differential Privacy. *arXiv preprint arXiv:2004.05975* (2020).

[48] Wassily Hoeffding. 1963. Probability Inequalities for Sums of Bounded Random Variables. *J. Amer. Statist. Assoc.* 58, 301 (1963), 13–30.

[49] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation. In *NDSS*.

[50] Zahra Jafargholi, Kasper Green Larsen, and Mark Simkin. 2019. Optimal Oblivious Priority Queues and Offline Oblivious RAM. *IACR Cryptol. ePrint Arch.* 2019 (2019), 237.

[51] Noah Johnson, Joseph P. Near, and Dawn Song. 2018. Towards Practical Differential Privacy for SQL Queries. *VLDB* (2018), 526–539.

[52] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. 2016. Generic Attacks on Secure Outsourced Databases. In *CCS*. 1329–1340.

[53] Deokjin Kim, DaeHee Jang, Minjoon Park, Yunjong Jeong, Jonghwan Kim, Seokjin Choi, and Brent ByungHoon Kang. 2019. SGX-LEGO: Fine-grained SGX controlled-channel attack and its countermeasure. *Comput. Secur.* 82 (2019), 118–139.

[54] Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO*. 104–113.

[55] Ios Kotsogiannis, Yuchao Tao, Xi He, Maryam Fanaeepour, Ashwin Machanavajjhala, Michael Hay, and Gerome Miklau. 2019. PrivateSQL: a differentially private sql query engine. *VLDB* (2019), 1371–1384.

[56] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. 2015. Ghostrider: A hardware-software system for memory trace oblivious computation. *ACM SIGPLAN Notices* 50, 4 (2015), 87–101.

[57] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. Innovative instructions and software model for isolated execution. In *The Second Workshop on Hardware and Architectural Support for Security and Privacy 2013*. 10.

[58] Frank D. McSherry. 2009. Privacy Integrated Queries: An Extensible Platform for Privacy-Preserving Data Analysis. In *SIGMOD*. 19–30.

[59] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, Mosha Pasumansky, and Jeff Shute. 2020. Dremel: A Decade of Interactive SQL Analysis at Web Scale. *VLDB* (2020), 3461–3472.

[60] Microsoft. [n.d.]. Always Encrypted with Secure Enclaves. https://techcommunity.microsoft.com/t5/azure-sql-database/always-encrypted-with-secure-enclaves\-try-it-now-in-sql-server/ba-p/386249. Accessed: 2020-09-10.

[61] 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.

[62] Prashanth Mohan, Abhradeep Thakurta, Elaine Shi, Dawn Song, and David Culler. 2012. GUPT: Privacy Preserving Data Analysis Made Easy. In *SIGMOD*. 349–360.

[63] Olga Ohrimenko, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Markulf Kohlweiss, and Divya Sharma. 2015. Observing and Preventing Leakage in MapReduce. In *CCS*. 1570–1581.

[64] Dan Page. 2002. Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel. *IACR Cryptol. ePrint Arch.* 2002 (2002), 169.

[65] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. 2019. Arx: An Encrypted Database Using Semantically Secure Encryption. *VLDB* 12, 11 (2019), 1664–1678.

[66] Raluca Ada Popa, Catherine MS Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: protecting confidentiality with encrypted query processing. In *SOSP*. 85–100.

[67] Christian Priebe, Kapil Vaswani, and Manuel Costa. 2018. Enclavedb: A secure database using SGX. In *SP (Oakland)*. 264–278.

[68] Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. 2014. Ring ORAM: Closing the Gap Between Small and Large Client Storage Oblivious RAM. *IACR Cryptol. ePrint Arch.* 2014 (2014), 997.

[69] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *CCS*. 199–212.

[70] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. 2018. ZeroTrace : Oblivious Memory Primitives from Intel SGX. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society.

[71] 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.

[72] Elaine Shi. 2020. Path Oblivious Heap: Optimal and Practical Oblivious Priority Queue. In *IEEE Symposium on Security and Privacy*. 842–858.

[73] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs.. In *NDSS*.

[74] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: an extremely simple oblivious RAM protocol. In *CCS*. 299–310.

[75] Salil P. Vadhan. 2017. The Complexity of Differential Privacy. In *Tutorials on the Foundations of Cryptography*. 347–450.

[76] Jelle van den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. 2015. Vuvuzela: scalable private messaging resistant to traffic analysis. In *SOSP*. 137–152.

[77] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *SIGMOD*. 1041–1052.

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

[79] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *SP (Oakland)*. 640–656.

[80] Xiangyao Yu, Syed Kamran Haider, Ling Ren, Christopher W. Fletcher, Albert Kwon, Marten van Dijk, and Srinivas Devadas. 2015. PrORAM: dynamic prefetcher for oblivious RAM. In *ISCA*. 616–628.

[81] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2012. Cross-VM side channels and their use to extract private keys. In *CCS*. 305–316.

[82] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2014. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *CCS*. 990–1003.

[83] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. 2017. Opaque: An oblivious and encrypted distributed analytics platform. In *NSDI*. 283–298.

[84] Mingxun Zhou, Elaine Shi, TH Hubert Chan, and Shir Maimon. 2022. A Theory of Composition for Differential Obliviousness. *Cryptology ePrint Archive* (2022).

[85] Wenchao Zhou, Yifan Cai, Yanqing Peng, Sheng Wang, Ke Ma, and Feifei Li. 2021. Veridb: An sgx-based verifiable database. In *Proceedings of the 2021 International Conference on Management of Data*. 2182–2194.