ADORE: A Differentially Oblivious Relational Database System

Paper #222

ABSTRACT

As data analytic workloads move to the cloud, preventing data leakage has become a critical problem in data management. Today, an effective approach is to leverage secure execution provided by hardware enclaves, such as Intel SGX, to ensure data confidentiality and integrity. Unfortunately, even when data analytics are protected by hardware enclaves, an attacker can still break data confidentiality by observing the access patterns of encrypted data.

We design and implement Adore, the first database system that satisfies *differential obliviousness*, a novel obliviousness property which ensures that memory access patterns satisfy differential privacy. We explore this new notion of obliviousness, and investigate the principled trade-offs which arise between performance and privacy. Compared to *full obliviousness*, which requires that memory access patterns leak no information whatsoever, differential obliviousness is a relaxation that still preserves provable privacy for each individual, while providing sufficient power to design more efficient algorithms.

We design a series of differentially oblivious algorithms for fundamental relational database operators, all of which have improved cache complexity compared to the state-of-art fully oblivious ones. Adore encrypts and decrypts data in parallel to enable high-performance database I/O. Our evaluations show that Adore outperforms the state-of-the-art fully oblivious algorithms by up to 17x on Big Data Benchmarks, and can scale to run on 10x larger database with the same hardware configuration.

1 INTRODUCTION

Moving data and computation to the cloud is the most dominant trend in the industry today. Cloud databases [30, 47, 59, 76] collect and analyze 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, 21, 32, 33, 41], protecting sensitive data in cloud databases has become more important than ever.

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 region 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 of 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 from outside the enclave (e.g., a server's main memory, disks). This leads to *access pattern* attacks [50, 63]. A long list of practical access pattern attacks of this form have been discovered for encrypted databases such as CryptDB and TrustedDB [2, 40, 46, 53, 54, 78].

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 database output including intermediate result needs to 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 [82], 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 don't protect memory access pattern at all or has arbitrary leakage, such as leaking the sizes of intermediate results and outputs, it is unclear whether these leakages would cause further privacy issues.

To address the performance issues of existing oblivious database systems, we design and build Adore, the first differentially oblivious database system. Instead of making access patterns indistinguishable between inputs, Adore enforces that the access patterns satisfy differential privacy [36, 37], a principled privacy guarantee for individuals. This notion is formally defined as *differential obliviousness*, and was introduced by Chan et al. [22]. This relaxation from full obliviousness opens up new design spaces for more efficient algorithms, yet still provides provable privacy guarantees for each database record.

We find that the key metric for the performance of enclave-based (differentially) oblivious database systems is **cache complexity** [3] (also called I/O complexity in some literatures). 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 encryption and decryption. Using this metric is further justified by our microbenchmark results in §7.1: in most queries the memory copy, encryption, and decryption together constitute more than 80% of total query completion time.

We propose a series of differentially oblivious algorithms for major relational operators: selection with projection, grouping with aggregation, and foreign key join. Designing these algorithms is challenging since differential obliviousness requires differential privacy guarantee on the entire memory access traces: simply adding differentially private paddings to the end result does not suffice. We show that our differentially oblivious algorithms require very little private memory (only polylogarithmic in N, where N is the

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

Systems/Ops	Privacy Model	Private Mem. Size	Cache Complexity	Output Size
ObliDB-Sel.	FO	1 ✓	$2N^*$	N
Opaque-Sel.	FO	1 ✓	$2N^*$	N
Adore-Sel.	DO	$\text{poly}\log(N)$	$(N+R)/B$ ✓	$R+ \text{poly}\log(N)$ ✓
ObliDB-FKHashJoin	FO	M	$N+O(N^2/M)^*$	$O(N^2/S)^*$
Opaque-FKSortJoin	FO	$\text{poly}\log(N)$ ✓	$N \cdot \log^2(N)^*$	N
Adore-FKSortJoin	DO	$\text{poly}\log(N)$✓	$6(N/B)\cdot\log(N/B)+N/B+R/B+\text{poly}\log(N)/B$ ✓	$R+\text{poly}\log(N)$ ✓
ObliDB-Grp.HashAgg.	FO	$M(M>R)$	$2N^*$	M^*
Opaque-Grp.SortAgg.	FO	$\text{poly}\log(N)$ ✓	$N \cdot \log^2(N)^*$	N
Adore-Grp.HashAgg.	DO	$M^*\ (M \geq O(\epsilon^{-1}\log^2(1/\delta)))$	$N/B+11NR/9MB$ ✓	$\frac{11}{9}R$
Adore-Grp.SortAgg.	DO	$\text{poly}\log(N)$ ✓	$6(N/B)\cdot\log(N/B)+N/B+R/B+\text{poly}\log(N)/B$ ✓	$R+\text{poly}\log(N)$ ✓

Table 1: Cache complexity/Private Memory Size/Output size comparisons of our system with ObliDB [39] and Opaque [82]. N denotes input size [2], 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.

input size), have significantly improved cache complexity, and have smaller output sizes with paddings when compared to their full oblivious alternatives. We show the comparison of our differentially oblivious relational operators with their fully oblivious alternatives in ObliDB and Opaque in Table 1.

We implement these differentially oblivious relational operators in Adore. To reduce the encryption and decryption latencies during query processing, we use a thread pool to enable the enclave encrypt and decrypt data in parallel. Our microbenchmarks show that it can shorten encryption and decryption time by 61% with 4 threads. We evaluate Adore using big data benchmark (BDB) [6]. Adore outperforms ObliDB, Opaque oblivious mode, and Opaque encryption mode (non-secure) in every BDB query and for every input size. Adore's parallel mode provides up to 17x performance improvement over existing oblivious database systems. Adore can also scale to larger data compared with existing oblivious database systems: Adore is the only system that can process input tables containing 30 million tuples in BDB Q2 and Q3, while the other two systems fail. In addition, Adore does not have any arbitrary leakage, such as the query plan leakage in ObliDB.

To summarize our contributions:

- We develop the methodology of differentially oblivious data analytics, using differential privacy to defend against access pattern leakage in enclave based encrypted database systems.
- We propose a series of differentially oblivious relational operators that have better cache complexity, require less private memory, and output less padding compared with existing fully oblivious ones.
- We design and implement Adore, the first differentially oblivious database system, which provides provable privacy guarantees, outperforms existing oblivious database systems up to 17x, and scales to larger input data.

Fundamentally, there is an inherent tension between achieving fully oblivious data processing and high performance. Adore's differentially oblivious approach establishes a principled trade-off between privacy and performance: each user's privacy is still enforced by differential privacy, and the relaxation on full obliviousness provides a new territory for more efficient data analytics algorithms and implementations.

[2]For join operator, we assume the total size of left table and right table is N

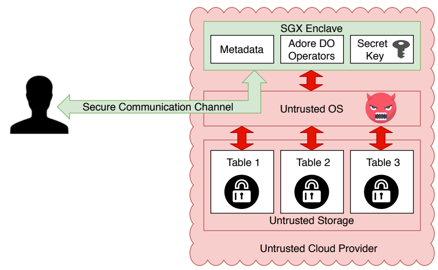


Figure 1: Adore's threat model. The data owner (who issues queries) and the database running inside SGX enclave are trusted. The operating system and the cloud provider is not trusted. Data is stored encrypted in untrusted storage.

2 BACKGROUND

In this section, we first describe our threat model (§2.1). We then formally define differential obliviousness and compare it with other privacy models (§2.2).

2.1 Threat Model

Figure 1 shows our thread model. We consider a database that runs in a trusted hardware enclave, i.e., Intel SGX [57], on an untrusted cloud server. The data that the database operates on is stored encrypted on the untrusted server's memory (i.e., *public memory*). All the processes and the operating systems are not trusted.

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 enclave can modify the enclave memory after the enclave is created. The enclave memory has an upper bound on its size (i.e., 128 MB). A SGX enclave has a predefined entry point, so a user process or the OS cannot invoke the enclave to run at arbitrary memory addresses. In addition, SGX provides *remote attestation* to allow a remote system to verify what software is loaded into an enclave and set up a secure communication channel to the enclave. This prevents the attacker from tempering with the enclave code and data during enclave creation.

These SGX features allow us to trust the code running inside the enclave. Untrusted processes and the operating system cannot temper 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.

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 back 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 integrity of the input data and the intermediate results so that the cloud server cannot modify them.

The database depends on the untrusted component to provide liveness, i.e., a malicious untrusted component can block the database from making any progress. However, the database does not depend on the untrusted component to provide data confidentiality and integrity. The untrusted component only loads data and query into the enclave, and the enclave can check whether the untrusted component's behavior is correct in terms of what data is loaded and how much data is loaded.

Unfortunately, the access patterns in the public memory are exposed to the untrusted cloud server. This means an attacker can watch how the enclave read the encrypted data, write the encrypted output, and read/write the intermediate result. Data access pattern leakage is sufficient for the attacker to extract secrets and data in many secure systems [17, 55, 64].

2.2 Differential Obliviousness

A good privacy model needs to capture both present and future attacks, and enable efficient algorithm designs and system implementations. We choose differential obliviousness since it offers provable privacy for individuals and at the same time allows more efficient algorithms.

To formally define differential obliviousness, we first define differential privacy in Definition 2.1.

Definition 2.1 (Differential Privacy)**.** A randomized algorithm $\mathcal{A}$ is $(\epsilon, \delta)-$differential private if for any two **neighboring databases** D_1, D_2 and any subset of possible outputs O of the algorithm:

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

Here, the probability is taken over the randomized coin flips of the algorithm $\mathcal{A}$, and the neighboring databases is defined as two databases that only differed by one tuple.

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 reveal any individual user's identity. Over the years, differential privacy has become the de facto standard for privacy, with growing acceptance in industry.

The notion of differential obliviousness is first proposed by Chan et al. [22]. It essentially requires the memory traces of an algorithm satisfy differential privacy. As a result, attackers cannot extract private information for each individual from observing memory access patterns. A system that is differentially oblivious is resilient to the attacks mentioned in §2.1. We formally define differential obliviousness in Definition 2.2.

Definition 2.2 (Differential Obliviousness)**.** A algorithm $\mathcal{A}$ is $(\epsilon, \delta)-$ 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 memory access pattern by applying algorithm $\mathcal{A}$ on D. We assume memory access pattern is a sequence of memory operations. The address of each operation and the type of operation (read or write) is known. The amount of data each read and write access must be the same. However, the content accessed is encrypted.

Differential obliviousness perfectly captures the threat model of enclave based database systems faces: as we discussed in §2.1, for a enclave based database systems, the data and code execution with enclave can be considered secure, and the data stored outside enclave is encrypted but the access patterns to them leak privacy. To fully appreciate the beauty of differential obliviousness model, we can compare it with another model that can also guard systems against access pattern leakage. For example, Goldreich defined a more strict oblivious notion in [42]. To differentiate it with differential obliviousness, we call it *full obliviousness*. We formally define full obliviousness in Definition 2.3.

Definition 2.3 (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:

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

We can see differential obliviousness has two vital relaxations on full obliviousness: (1) differential obliviousness requires the memory access patterns to be ϵ-indistinguishable rather than fully indistinguishable. (2) differential obliviousness only requires the memory access patterns over neighboring databases to be indistinguishable.

These relaxations makes designing more I/O efficient algorithms possible. For example, in full obliviousness model, the 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, in differential obliviousness model, the database system can add random amount filler tuples so that the output size is indistinguishable for two neighboring databases. Classic differential privacy results show that the average amount of filler tuples added is only poly logarithmic to the input size.

Databases with Filler Tuples. In differential oblivious setting, one technicality is that the input size could leak sensitive information. For example, if D' is the database that we delete 1 tuple from D, apparently D and D' are neighboring databases. However, the size difference of D and D' itself leaks what is the input data. As a result, in this paper, we assume that all the input, as well as intermediate result and output, contains ploy logarithmic to the overall size filler tuples. If we delete one tuple from the database, we simply mark this tuple as a filler tuple ($\perp$).

Privacy Parameters. If an algorithm is (ϵ, δ)-differentially oblivious, ϵ and δ are called privacy parameters of the algorithm. In Adore, we need to provide (ϵ, δ)-differential obliviousness guarantee to the entire query, which may consists of more than one differentially oblivious operators. So we need to set each operator's own ϵ_i and δ_i so that the over all query is (ϵ, δ)-differentially oblivious. Since the composition rule (we prove in §5) says the overall privacy parameter, ϵ, δ is additive of the security parameters of each operator. We refer the overall security parameter ϵ and δ as "privacy budgets". These budgets need to be distributed among operators.

3 OVERVIEW

Adore is an SGX-based database that enforces data privacy. Data is stored encrypted in the public memory, and query processing happens inside an enclave. We require Adore to (1) satisfy *differential obliviousness*, i.e., access patterns of the public memory satisfy differential privacy; and (2) *have high performance*, i.e., the database sustain high throughput and can scale to contain large datasets.

To realize our design goals, we first need to design a set of differentially oblivious algorithms for basic operators in relational databases. This is challenging because we need to make sure that the access patterns (including reading/writing intermediate tables) during these basic database operations satisfy differential privacy—simply padding the result of each operation is not enough. To this end, we have designed four differentially oblivious basic operators: (1) selection and projection, (2) sort-based groupby and aggregation, (3) hash-based groupby and aggregation, and (4) foreign key join.

Although these four algorithms are completely different, our guiding principle is the same: we either use a differentially private algorithm to guide our accesses of public memory or to break down a query into smaller ones for which we can directly use fully oblivious algorithms. Note that fully obliviousness is strictly stronger than differential obliviousness, so we can use them without leaking access patterns. Taking selection as an example. Selection is to filter database tuples with a given predicate. If we only output a tuple when the predicate evaluates to TRUE, this will leak the information about what the predicate evaluates to for every tuple in the database. Instead, we consult a differentially private prefix sum oracle, i.e., a streaming algorithm that outputs the sum for all the elements it has currently seen while maintaining differential privacy. We ask the oracle to process a stream of elements, where each element is either 0 or 1, depending on whether the predicate evaluates to FALSE or TRUE for each database tuple. We output the tuple only when the prefix sum oracle increments the current sum, otherwise we buffer the tuple inside the enclave memory. Because the output pattern becomes differentially private, the selection operator is differentially oblivious (§4.1). We also design our own differentially private algorithms as building blocks when there is no known practical differentially private algorithm for the problem. For example, we propose a new differentially private distinct element count algorithm and use it in our hash-based grouping algorithm (§4.2). We use fully oblivious sort as a building block in our sort-based grouping (§4.2) and foreign key join (§4.3) algorithms.

Ensuring differential obliviousness at a per-operator level is not enough for Adore. Real-world queries usually combine multiple operators, and we need to make sure that the entire query, rather than each operator, is differentially oblivious. We develop the composition rule for differentially oblivious operators: essentially both the indistinguishable factor ϵ and the failure probability δ is additive among different operators and have a multiplicative scale of sensitivity [3]. In addition, we also propose optimizations to reduce multi-query leakage by reusing randomness for shared computation (§5).

Our differentially oblivious query processing allows Adore to achieve higher performance than fully oblivious alternatives, but the cost of encryption and decryption is still a significant bottleneck. To address this issue, we leverage the multi-thread support in Intel SGX enclaves. Adore instantiates a separate thread pool, and the enclave can use the threads in it to encrypt and decrypt data in parallel to speed up accesses to the encrypted data in the public memory (§6).

4 DIFFERENTIALLY OBLIVIOUS RELATIONAL 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.

We propose a differentially oblivious algorithm for selection with projection whose cache complexity matches the lower bound. The main technique of this algorithm is inspired by a theoretical result on differentially oblivious compaction [22]: using a differentially private prefix-sum oracle to guide the memory access of filtering (§4.1). Next, we propose two differentially oblivious algorithms for grouping with aggregation, one is hash based, the other is sort based (§4.2). The hash based algorithm is more efficient when number of groups generated is small. Notably, to develop a hash based grouping algorithm, we propose a novel, practical differentially private distinct count algorithm (Algorithm 2). This is the *first* practical differentially private distinct count algorithm with bounded approximation concentration to the best of our knowledge! We use this algorithm to estimate the number of group produced and then use a pseudo-random function to partition the input database to 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 (§4.3).

One invariant that all our differentially oblivious algorithm carefully maintains is that they are all *distance preserving*. This requires that if they take any neighboring databases as input, their output must be neighboring database as well. For example, this invariant is not guaranteed many database systems since most SQL query doesn't enforce an explicit order on the output. Distance preserving allows the differential obliviousness guarantees composed more easily, as we will discuss in detail in §5.

[3]Sensitivity is a standard differential privacy notion. In our context, it can be defined as: if the input database is differed by one tuple, what the the maximum difference of output in terms of hamming distance.

4.1 Selection and 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. Intuitively, selection operators act like a filtering operation in functional programming languages. A projection operator transforms one relation into another, possibly with a different schema: it is written $(\Pi_{a_1,...,a_n}(R))$ where $a_1, ..., a_n$ is a set of attribute names. The result of such a projection is defined as the set obtained from the components of the tuple; it discards (or excludes) the other attributes. In many database systems, projection is usually inlined in selection. ADORE follows this tradition. In ADORE, projection is inlined with selection and is done within private memory. As a result, ADORE in fact does not require a stand-alone DO algorithm for projection.

Now, we give the DO algorithm for $\sigma_\phi(R)$, where ϕ is the filtering predicate and R is the input table. To better understand our algorithm, we start from 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 naive 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 can observe 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 [22]. To determine how much noise to add on memory access at each step, we can query a differentially private oracle for computing prefix sum in data streams.

Differentially private prefix-sum. For a data stream that consists of only 0s and 1s with length T, $I \in \{0,1\}^T$, the prefix-sum Y_t is the count of how many 1s appears in the first t elements. Now, suppose we have a $(\epsilon, \delta)-$differentially private prefix sum algorithm that answers can answer up to T queries, and each answer $\widetilde{Y}_t \in [Y_t - s, Y_t + s]$ with high probability. To make the traces of the write pointer differentially private, We can always move the output pointer to $\widetilde{Y}_t - s$, and keep the scanned but not yet outputted tuples in the private buffer. And most importantly, we only need $2s$ sized buffer in private memory and the algorithm would not need to output filler tuples except in the and with high probability.

We use the binary mechanism of Chan et al. [25] for our DP prefix-sum oracle. This mechanism essentially builds a binary interval tree to store noisy partial sums for the optimal approximaty-privacy [4] trade-off. For each $t \in [T]$, the estimated prefix-sum $\widetilde{Y}_t$ from the binary mechanism preserves ϵ-differential privacy while $(O(\epsilon^{-1} \cdot (\log T) \cdot \sqrt{\log t} \cdot \log(1/\delta)), \delta)$-useful [25, Theorem 3.5, 3.6].

[4]The approximaty also called the utility of prefix-sum: a counting mechanism is (λ, δ)-useful at time t, if with probability (over the randomness of the mechanism) at least $1 - \delta$, we have the counting error less than λ. Note λ may be a function of δ and t.

Algorithm 1 DOFILTER: Differentially Oblivious Filtering

1: **procedure** DOFILTER($I, \Pi, \phi, \epsilon, \delta$) ▷ Theorem 4.1.
2: $s \leftarrow$ DPORACLEUTILITY($\epsilon, \delta, |I|$)
3: $P \leftarrow \emptyset$ ▷ a FIFO buffer in private memory of size $2s$
4: $I' \leftarrow \emptyset$ ▷ output table
5: $c \leftarrow 0$ ▷ current read counter in I
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**

Differentially Oblivious Filtering. We present the detailed DOFILTER in Algorithm 1. Let s be the approximation error (with probability at least $1 - \delta$ for each query) of 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).

Correctness failures to privacy failures. Algorithm 1 is designed to have at most δ failure probability. Although δ is negligible (usually set to $2^{-20} - 2^{-40}$), in case of that users want perfect correctness, we can use the standard technique to convert the correctness failures to privacy failures. The only case it could fail is that the DP prefix-sum oracle's estimation is off by more than s. P will could either blow up at line 11 or have nothing to pop at line 14. Instead of fail and re-run the algorithm, we we could simply write t to the output if P's capacity blows up and simply write a filler tuple to the output if there is nothing to pop at line 14. Thus, we convert correctness failures to privacy failures.

Theorem 4.1 (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 $O(\log(1/\epsilon) \cdot \log^{1.5} N \cdot \log(1/\delta))$ private memory and $(N+R)/B$ cache complexity*[5].

Remark 4.2. Remark: Note that $\Omega((N + R)/B)$ cache complexity is trivial lower bound. Thus, our cache complexity is optimal.

We adapted the technique in [22], which present a DO stable compaction algorithm. Stable compaction is different problem since it keeps all the elements in the input. Also, in [22], they don't have

[5]All proofs in this section can be found in appendix C.

notion of the cache complexity, and therefore don't provide any bound for that.

4.2 Grouping and 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.

Hash Based Grouping. In many real world scenarios, the number of groups generated is much less than the size of the input. As a result, they may well fit into private memory, or be only few times larger than the size of private memory. In this case, we first develop a hash based grouping algorithm for when the number of groups is small.

First, we propose a non-oblivious hash based grouping based on randomized partitioning on grouping attributes. This can be done by applying a pseudo-random 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 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 [16, 26, 48], 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 concentration of both the approximation error and the sensitivity at the same time. We present a 1.1-approximate version in Algorithm 2 [6].

Our differentially private distinct count algorithm (Algorithm 2) 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 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

[6]The more general version of this algorithm and detailed proofs can be found in Appendix A

Algorithm 2 1.1-APPROX. DPDISTINCTCOUNT

1: **procedure** 1.1-APPROX. DPDC(I, ϵ, δ) ▷ Lemma 4.3
2: pQUEUE $\leftarrow \emptyset$ ▷ pQUEUE is a priority-queue of size t
3: ▷ $t = 10^3\epsilon^{-1}\log(24(1+e^{-\epsilon})/\delta)\log(3/\delta))$
4: **for** $x_i \in I$ **do**
5: $y \leftarrow h(x_i)$ ▷ h: $[m] \rightarrow [0, 1)$, is a PRF
6: **if** $|$pQUEUE$| < t$ **then**
7: pQUEUE.PUSH(y)
8: **else if** $y <$ pQUEUE.TOP() $\wedge\, y \notin$ pQUEUE **then**
9: pQUEUE.POP()
10: pQUEUE.PUSH(y)
11: **end if**
12: **end for**
13: $v \leftarrow$ pQUEUE.TOP()
14: ct $\leftarrow 1.075\frac{t}{v} + \mathrm{Lap}(0.02n/\log(3/\delta))$
15: **return** ct
16: **end procedure**

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.

Lemma 4.3. *For any $0 < \epsilon < 1$, $0 < \delta \leq 10^{-3}$, there is an distinct count algorithm (Algorithm 2) such that:*

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

$$n \leq \widetilde{A} \leq 1.1n,$$

where n is the number of distinct elements in the data stream. The space used by the distinct count algorithm is $O(\epsilon^{-1} \cdot \log^2(1/\delta) \cdot \log n)$ bits.

In Algorithm 3, we present our differentially oblivious hash based grouping algorithm. 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). 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 a 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 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 4.4 (Main result for group, group via hashing)**.** *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 3) that uses M private memory and has $N/B + 11NR/9MB$ cache complexity.*

Sort Based Grouping. Next, we present a sort based grouping algorithm for the query that produces a large number of groups.

Algorithm 3 DOGroup$_h$: DO Hash Based Grouping

procedure DOGROUP$_h$(I, L, ϵ, δ) ▷ Theorem 4.4
  $\widetilde{G} \leftarrow$ 1.1-APPROX. DPDC($I, \epsilon, \delta/2$) ▷ Algorithm 2
  $k \leftarrow \lceil \widetilde{G}/0.9M \rceil$ ▷ M: size of the private memory
  Verify that $\sqrt{0.5\widetilde{G}\log(2k/\delta)} \leq 0.1M$
  $R \leftarrow \emptyset$ ▷ output table
  for $i \in 0, \ldots, k-1$ **do**
    $\mathcal{H} \leftarrow \emptyset$ ▷ Hash table for grouping
    for $t \in I$ **do**
      if $h(t.L) \in [i/k, (i+1)/k)$ **then**
        ▷ $h : [m \times \ldots \times m] \leftarrow [0, 1)$, is a PRF
        if $\mathcal{H}$.HASKEY($t.L$) **then**
          update $\mathcal{H}(t.L)$'s aggregate values using t
        else
          $\mathcal{H}(t.L) \leftarrow t$
        end if
      end if
    end for
    write all tuples in $\mathcal{H}$ and filler tuples to R, s.t. R's size increased by M
  end for
end procedure

We start from a naïve, non-oblivious sort based grouping algorithm. First, sort R according to the grouping attributes. Then, run a linear scan over the sorted R while accumulating aggregation results. By the time the first tuple from each new group is scanned, output the tuple representing the previous group. We note that this naïve algorithm is not oblivious, even if we replace the sorting algorithm with oblivious sort. To see this, Figure 2 demonstrates the write pattern of this naïve grouping algorithm. We can observe that a new tuple is written at the first tuple of a new group (except for the first group) as well as at the end of the input. This clearly leaks how many tuples are in each group.

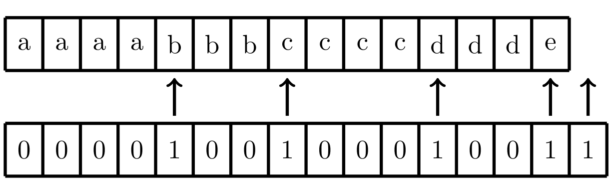


Figure 2: Write Pattern of Grouping

One observation is that if we assign key "1" to the first element of each group (except the first group) and add an element with key 1 to the end, now the progress of the write pointer is exactly equal to the prefix sum of the "1" keys. Thus, we can develop an algorithm similar to DOFILTER (Algorithm 1) for differentially obliviously grouping. Additionally, we need to inline the accumulation of aggregation value in the algorithm.

We present our differentially oblivious sort based grouping algorithm, DOGROUP$_s$, in Algorithm 4. DOGROUP$_s$ first sorts input I according to the list of grouping attributes L (line 2). Here, BUCKETOBLIVIOUSSORT [10] is used since it has good asymptotic complexity ($O(n\log(n)$, compare with bitonic sort's $O(n\log^2(n))$ and a constant factor of 6, which is relatively small. Similar to DOFILTER, we create a FIFO buffer P of size s in private memory, where s is the utility of of the DP prefix-sum oracle (line 3 - 4). We also initiate the output table R in public memory, a counter c, and the working tuple a in private memory (line 5 - 7). The DOGROUP$_s$ repeats the following until reaching the end of I': read the next x tuples, update the counter c (line 9 -10); for each tuple t read, if its grouping attributes' value is the same as the working tuple a's, update a's aggregate values using t (line 13), otherwise, push a to P if a is not $\perp$ (initial working tuple), and assign t as the new a (line 15 - 18). After processing all tuples in this batch, pop P to write R until $|R| = \widetilde{Y}_c - s$, where $\widetilde{Y}_c$ is the DP oracle's estimation of the prefix sum till c (line 21). In the end, push the last working tuple a to P (line 23), pop all tuples in P and add filler tuples if necessary so that $|R| = \widetilde{Y}_N + s$, where N is the size of I (line 24).

Algorithm 4 DOGROUP$_s$: DO Sort Based Grouping

1: **procedure** DOGROUP$_s$(I, L, ϵ, δ) ▷ Theorem 4.5
2:   $I' \leftarrow$ BUCKETOBLIVIOUSSORT(I, L)
3:   $s \leftarrow$ DPORACLEUTILITY($\epsilon, \delta, |I|$)
4:   $P \leftarrow \emptyset$ ▷ a FIFO buffer in private memory of size $2s$
5:   $R \leftarrow \emptyset$ ▷ output table
6:   $c \leftarrow 0$ ▷ current read counter in I'
7:   $a \leftarrow \perp$ ▷ the working tuple
8:   **while** $c < |I'|$ **do**
9:     $T \leftarrow \{I_c, I_{c+1}, \ldots I_{c+s-1}\}$ ▷ read the next s tuples
10:     $c \leftarrow c + s$ ▷ update the read counter
11:     **for** $t \in T$ **do**
12:       **if** $t.L = a.L$ **then**
13:         update a's aggregate values using t
14:       **else**
15:         **if** $a \neq \perp$ **then**
16:           $P \leftarrow P$.PUSH(a)
17:         **end if**
18:         $a \leftarrow t$
19:       **end if**
20:     **end for**
21:     Pop P to write R until $|R| = \widetilde{Y}_c - s$
22:   **end while**
23:   $P \leftarrow P$.PUSH(a)
24:   write all tuples from P and filler tuples to R s.t. $|R| = \widetilde{Y}_N + s$ ($N = |I|$)
25: **end procedure**

Theorem 4.5 (Main result for group, group via sorting). *For any $\epsilon \in (0, 1), \delta \in (0, 1)$ and input of size N, there is an (ϵ, δ)-differentially oblivious and distance preserving grouping algorithm (DOGROUP$_s$ in Algorithm 4) that uses $O(\log(1/\epsilon) \cdot \log^{1.5} N \cdot \log(1/\delta))$ private memory and $6(N/B)\log_B(N/B) + (N+R)/B$ cache complexity.*

4.3 Foreign Key Join (⋈)

Foreign key join is the most widely used join operator in data analytics. We write $R \bowtie S$[7] to represent a join on the primary key and foreign key pairs of the relations R and S

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

The standard oblivious foreign key join works similarly to sort based grouping algorithm. It first pad 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. The oblivious sort routes the tuples to be joined from both tables to the same group. Next, the algorithm make a sequential scan of 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 [82] and followed by ObliDB [39].

We develop DoJoin, a differentially oblivious foreign key join algorithm. 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 Opaque. Compared with bitonic sort, bucket oblivious 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, §4.1). 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 5 DoJoin: DO Foreign Key Join

1: **procedure** DoJoin($R, S, k_R, k_S, \epsilon, s$) ▷ Theorem 4.6
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: $R' \leftarrow \rho(\mathtt{mark}(R, \text{‘r’}), k_R \rightarrow k)$
5: $S' \leftarrow \rho(\mathtt{mark}(S, \text{‘s’}), k_S \rightarrow k)$
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** $x_i.mark = \text{‘r’}$ **then**
11: $t \leftarrow x_i$
12: $I' \leftarrow I'||\perp$ ▷ $\perp$ means filler tuple
13: **else** ▷ $x_i.mark = \text{‘s’}$
14: $I' \leftarrow I'||(x_i \cup t)$
15: **end if**
16: **end for**
17: $T \leftarrow$ DoFilter$(I', \mathrm{ID}, \lambda t.t \neq \perp, \epsilon, \delta)$ ▷ remove filler tuples, Algorithm 1.
18: **return**
19: **end procedure**

We present DoJoin in Algorithm 5. 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 4.6 (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 5) that uses $O(\log(1/\epsilon) \cdot \log^{1.5} N \cdot \log(1/\delta))$ private memory and has $6(N/B)\log(N/B) + (N+R)/B$ cache complexity.*

5 COMPOSING DIFFERENTIAL OBLIVIOUSNESS

Having differentially oblivious operators in §4 is only the first step, overall, we need have differential obliviousness guarantee on the entire database query, which consists of the composition of these operators. In this section, we discuss how to compose these differentially oblivious operators to obtain the overall differential obliviousness of a database query and among multiple queries. We first present the composition rule of differential obliviousness, and then discuss how to reduce leakage in multiple queries by reusing randomness.

The Composition Rule. We present the composition rule for differential obliviousness in Lemma 5.1. Intuitively, the overall leakage is additive among different operators and is multiplicative with respect to the sensitivity.

Lemma 5.1 (The composition rule of DO)**.** *If $\mathcal{M}_1 : \mathcal{X} \rightarrow \mathcal{Y}$ is (ϵ_1, δ_1)-differentially oblivious mechanism and has sensitivity k and mechanism $\mathcal{M}_2 : \mathcal{Y} \rightarrow \mathcal{Z}$ is (ϵ_2, δ_2)-differentially oblivious, then $\mathcal{M}_1 \circ \mathcal{M}_2 : \mathcal{X} \rightarrow \mathcal{Z}$ is $(\epsilon_1 + k\epsilon_2,\ \delta_1 + ke^{k\epsilon_2}\delta_2)$-differentially oblivious.*

Proof. From the group privacy lemma [74, Lemma 2.2], and sensitivity of $\mathcal{M}_1$, we know that $\mathcal{M}_2(\mathcal{M}_1(X))$ alone is $(k\epsilon_2, ke^{k\epsilon_2}\delta_2)$-differentially oblivious with regard to neighboring $X, X' \in \mathcal{X}$. Using the basic composition [37, Theorem 3.16], we know that $\mathcal{M}_1 \circ \mathcal{M}_2$ is $(\epsilon_1 + k\epsilon_2,\ \delta_1 + ke^{k\epsilon_2}\delta_2)$-differentially oblivious. □

Since all the differentially oblivious operators we developed in §4 are distance preserving, they all have sensitivity 1. As a result, the composition of privacy parameters is mostly additive. we can view the privacy parameters for a query, ϵ and δ, as "privacy budgets". These budgets need to be distributed among different operators.

For example, if the query we evaluate is $Q_1 = \sigma_\phi(R_1 \bowtie R_2)$. We can call DoJoin with $\epsilon = \epsilon_0/2$ and $\delta = \delta_0/2$, and DoFilter with $\epsilon = \epsilon_0/2$ and $\delta = \delta_0/2e^{\epsilon_0}$. As a result, the entire query is (ϵ_0, δ_0)-differentially oblivious.

How to distribute privacy budget optimally also opens new research problems. There are two knobs to tune here: the first is how to formulate the query. The second knob is how to distribute privacy budget over different operators. Further privacy aware query optimization is an open problem but out of the scope of this paper.

Reducing Privacy Leakage of Multiple Queries. A typical data analytic system processes many queries per day. We find there are two scenarios where we can reduce the privacy leakage brought by multiple queries: re-run of the same query and multiple queries sharing same computation.

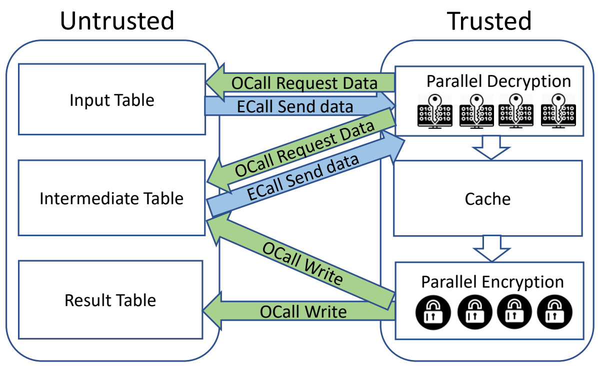


Figure 3: ADORE's workflow. Green arrows are ocalls, and blue arrows are ecalls.

In the case of re-running the same query over the same data, even we don't cache the query result, we can still prevent new privacy leakage in each re-run of the query by reusing the same random seed for generating random noise. In the case of running a new query that shares some part of computation with previous queries, we could cache the random seed in the operator level, and replay the random seed whenever the same operator over the same input data is called. There were intensive studies in database research community on how to identify the part of computation that is shared among multiple queries, such as reusing materialized views [15, 27, 44, 61, 72].

6 IMPLEMENTATION

ADORE is implemented using ~4500 lines of C++. We encrypt each row with authenticated encryption AES-GCM and store the MAC of each row into its header to ensure the integrity and confidentiality of each row. We use Intel's SGX SDK (version 2.9) to develop ADORE. We use thread pool implementations in Boost (version 1.71).

ADORE consists of two components: an untrusted component in a standard Linux program that manages data outside the enclave and a trusted component inside the enclave for query processing. The untrusted component holds the entire encrypted data inside DRAM. Similar as other SGX-based applications, the untrusted component and the trusted component communicate through ecalls and ocalls. The untrusted component uses ecalls to trigger functions inside the trusted component, and the trusted component uses ocalls to request data.

Figure 3 shows ADORE's workflow. When untrusted component receives a query from the database user, it forwards the query to the trusted component inside the enclave. This triggers the start of the query processing. The untrusted component, through having access to the entire encrypted data, does not know what fraction of them are the input of the query. During the query processing, the trusted components can issue four types of commands to the untrusted component: (1) read input data, (2) write intermediate result (because memory inside the enclave is limited), (3) read intermediate result, and (4) write output result. At the end of the query processing, the encypted result is written in the untrusted component's memory.

Accessing encrypted in the public memory is a major bottleneck because encryption and decryption are CPU-intensive. To alleviate this bottleneck, we use multiple dedicated threads to decrypt and encrypt data. The trusted component cannot create threads because it cannot directly use system calls. Instead, we create a thread pool in the untrusted component, when the trusted component issues a request to read encrypted data from the public memory and write encrypted data to the public memory, the untrusted component let all the threads in the thread pool to ecall into the trusted component to let the trusted component encrypt and decrypt data. This allows the trusted component to encrypt and decrypt data in parallel.

7 EVALUATION

In this section, we first break down and analyze the latency of ADORE's basic operators, such as filter, grouping and join. We then evaluate ADORE on Big Data Benchmark [6]. We compare ADORE with Opaque encrypted (non-oblivious) and non-padded oblivious mode[8] [82], 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 fill in ADORE with data from the Big Data Benchmark [6]. We evaluate ADORE under three tiers of input table sizes: *Rankings* table contains 100K, 1M, 10M rows, and *UserVisits* table has triple size of *Rankings*: 300K, 3M and 30M rows. By default, ADORE allocates 4 threads for parallel encryption and decryption with batch size 65,536. All systems are compiled and run under SGX prerelease and hardware mode.

Setting Privacy Parameters In ADORE, we set ϵ to 1 and δ to 2^{-20}, 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 [75], and RAPPOR [38]. When processing queries that consists of more than one operators, these privacy budget distributed evenly among different operators. 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 differentially private oracle in Algorithm 1, we can simulate its usefulness by repeating random trials of sum of Laplace noises. Figure 5 shows the simulation result. We can observe that the sum of independently sampled Laplace noises grows sub-linearly as the δ grows exponentially. We can confidently over estimate the usefulness by assuming linear growth of $|Y|$ over δ's exponentially growth when δ is too small to simulate.

7.1 Microbenchmark

We break down each of ADORE's basic operator's completion time into six categories: (1) decryption within enclave; (2) encryption within enclave; (3) reading from untrusted memory to enclave buffer; (4) writing from enclave buffer to untrusted memory; and (5) computation within enclave. Figure 4 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

[8] Opaque does not maintain its oblivious mode anymore. We did our benchmark using a previous branch provided by Opaque authors : https://github.com/mc2-project/opaque/tree/c42fe1bb758a93. In this branch, Opaque's oblivious mode does not pad query result to maximum possible size, so it leaks the size of the output and is thus not fully oblivious.

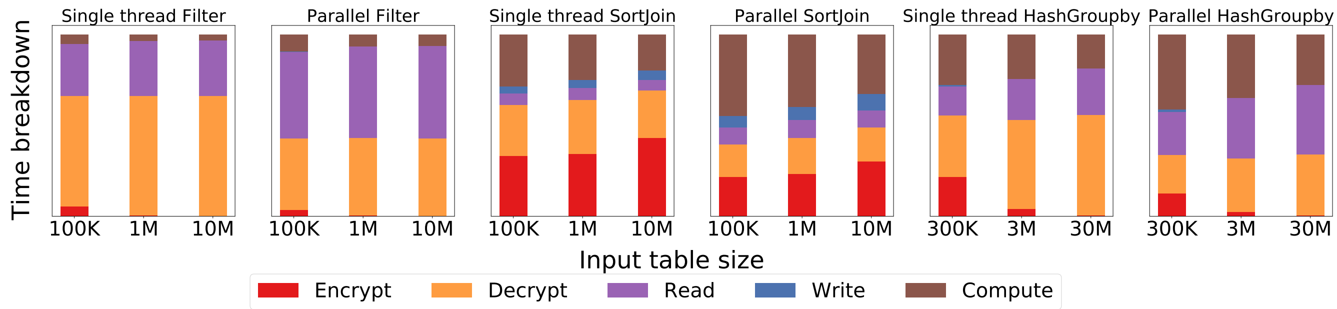


Figure 4: Differentially oblivious operator performance breakdown

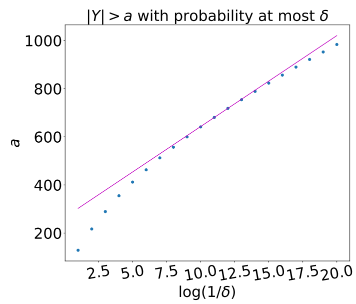


Figure 5: Simulated Binary Mechanism Concentration ($\epsilon = 1$, $N = 10^9$, each data point uses $10^4/\delta$ trials)

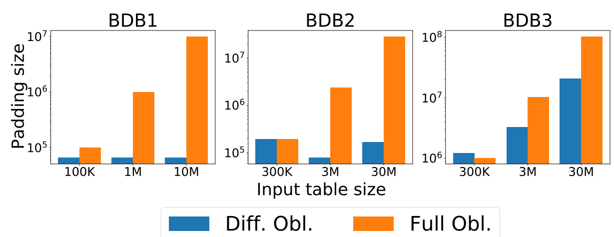


Figure 6: Padding size comparison between differential obliviousness and full obliviousness

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%. After applying parallel encryption and decryption technique, the encryption and decryption time reduce by 60% and the proportion of encryption and decryption time consumption drops from 68% to 48%. Parallel encryption and decryption technique also works for other differentially oblivious operators (i.e., sort-based join, hash-based groupby). Because applying hash function to distinguish different groups in hash-based groupby operator and oblivious sorting in sort-based join operator are more expensive than simple comparison in filter operator, the computation constitutes a larger fraction in the query completion time.

Figure 6 compares the number of paddings rows Adore requires in running Big Data Benchmark queries under different input table sizes. For comparison, we calculate the maximum possible output size based on BDB queries for fully oblivious database, and the minimum padding size for any fully oblivious database is then the maximum possible output size minus the real output size. On the largest input table size, using Adore reduces the padding sizes by 99.3%, 99.4%, and 79.8%, respectively. Our hash-based groupby requires the same padding size with ObliDB hash-based groupby when all of the distinct groups can be processed within one pass. This is because our hash-based groupby requires the enclave to pad output to the maximum number of distinct groups aggregation statistics SGX enclave memory can hold(400000 under current SGX enclave memory capacity) at the end of each pass. When the number of distinct groups can not fit in enclave memory, Opaque and ObliDB use sort-based groupby whose padding size is much larger than Adore's hash-based groupby. On BDB3, Adore still requires significant amount of paddings, because our sort-based join requires a random bucket assignment phase, which incurs additional paddings in intermediate tables.

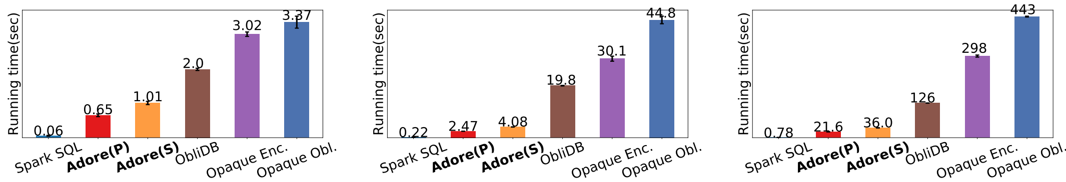
7.2 Comparison to Prior Work

We now evaluate Adore on the three standard queries in the Big Data Benchmark [6], a widely used benchmark for big data analytics. The three queries cover filter, aggregation, join and orderby operators

BDB 1:

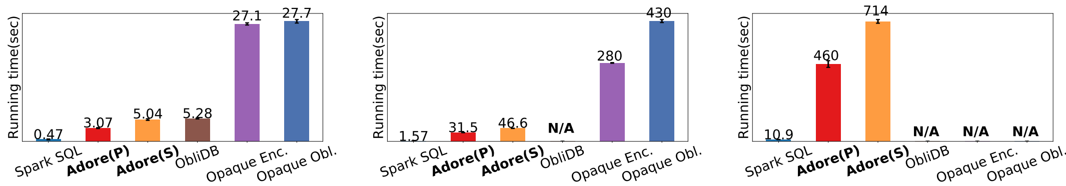
```
SELECT pageURL, pageRank
FROM rankings
WHERE pageRank > 1000
```

The first query performs a filter on *rankings* table, then project pageURL and pageRank columns to output. We compare the performance of Adore (both the single-threaded and the parallel version) with Spark SQL, ObliDB, Opaque encrypted mode and Opaque non-padding oblivious mode in Figure 7. Compared with non-encrypted and non-oblivious spark SQL, for moderately large sized datasets, the single-threaded Adore exhibits 15.8−57.5x overhead. As shown in §7.1, Adore's system overhead mostly comes from encryption and decryption when moving data in and out of SGX enclave memory. Adore's single-threaded execution time is only 12.7%−33.4% of Opaque encrypted mode, a encrypted but not oblivious system, under dataset sizes(100K, 1M and 10M). The performance gain comes from the batched read and write implemented in our system. Compared with oblivious systems, Adore's single-threaded execution time is 20.6% − 50.5% of ObliDB and 8.1% − 29.8% of Opaque non-padded oblivious mode (it still leaks output size). This performance gain comes from more efficient algorithm and the less padding size brought by the differential obliviousness.



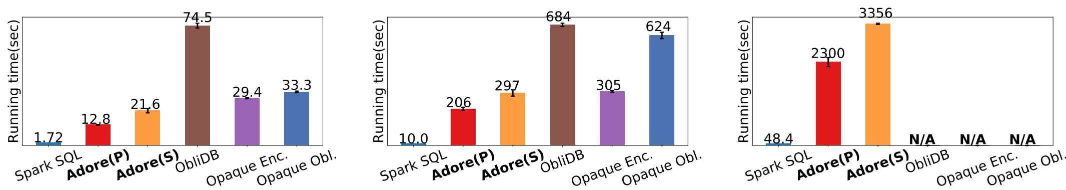
(a) BDB1-100k **(b) BDB1-1million** **(c) BDB1-10million**

Figure 7: BDB1 performance under different Rankings table size. Error bars show the standard deviations.



(a) BDB2-300k **(b) BDB2-3million** **(c) BDB2-30million**

Figure 8: BDB2 performance under different UserVisits table size. Error bars show the standard deviations.



(a) BDB3-300k **(b) BDB3-3million** **(c) BDB3-30million**

Figure 9: BDB3 performance under different UserVisits table size. Error bars show the standard deviations.

Adore's parallel version is 1.51 – 1.66x faster than the single-threaded version. This further performance improvement comes from parallel encryption and decryption when moving data in and out of enclave memory. Although the overhead of spawning threads is negligible in our implementation because of our SGX threading pool design, there are still ocall and ecall cost per thread per batch. As a result, there is a balance between the run time gain by parallelism and the extra ecall and ocall cost. We find that 4 threads are the optimal setting in our current setup. In the future, if the next generation enclave has larger memory size, our parallel mode would have even large performance gain since we could use a larger batch size.

BDB 2:

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

The second query 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 sized datasets(300K - 30M), the single-threaded Adore exhibits 10.8 – 64.5x overhead. Adore's single-threaded execution time is only 16.6% – 18.8% of Opaque encrypted mode. As mentioned before, the performance gain comes from the batched read and write implemented in our system. Compared with oblivious systems, Adore's single-threaded execution time is 10.8% – 18.3% of Opaque non-padded oblivious mode. For query 2 over *UserVisits* table of 300K rows, single-thread Adore has similar performance with ObliDB. However, ObliDB's hash-based groupby operator assumes that all of distinct groups can fit in enclave memory so that it can calculate all the aggregation statistics(each entry is 4 bytes) in just one pass, but this assumption does not hold for *UserVisits* table of 3 million rows and more. ObliDB, Opaque encrypted and oblivious mode all fail to run query 2 over *UserVisits* table of 30 million rows under our hardware settings too. As the number of distinct groups grows, Adore has to process aggregation query in more passes, which is another source of overhead to achieve differentially oblivious hash-based groupby.

Adore's parallel version further improves the single-threaded version performance on query 2 by 1.47 – 1.81X. Parallel decryption and encryption contribute to this performance speedup.

BDB 3:

```
SELECT sourceIP, totalRevenue, avgPageRank
FROM (SELECT sourceIP,
        AVG(pageRank) AS avgPageRank,
        SUM(adRevenue) AS totalRevenue
    FROM Rankings AS R, UserVisits AS UV
    WHERE R.pageURL = UV.destURL
```

```
      AND UV.visitDate
        BETWEEN Date('1980-01-01')
        AND Date('1983-01-01')
  GROUP BY UV.sourceIP)
  ORDER BY totalRevenue DESC
```

The third query first performs a filter on *visitDate* column of *UserVisits* table. Then it joins *Rankings* and *UserVisits* tables on *pageURL* column. Finally, it does an aggregation on *sourceIP* column to calculate the sum of *adRevenue* and average of *pageRank* and output them ordered by the sum of *adRevenue*. Query 3 is a relatively more complex analytical query. In Opaque and ObliDB benchmark section, they set the *visitDate* range as 3 months (1980-01-01 to 1980-04-01) which is a quite short period for analytical queries. We extend this range to 3 years for benchmarking all the four systems. Similar to query 2, ObliDB and Opaque both fail to execute query 3 over the largest dataset with 64GB memory. On moderately large datasets, the single-threaded Adore exhibits 12.2 – 69.4x overhead over Spark SQL. Its execution time is 71.3% – 94.2% of Opaque encrypted mode, 46.2% – 63.0% of Opaque non-padded oblivious mode and 32.5% – 42.1% of ObliDB. As stated before, this performance gain under single-thread execution mainly comes from less dummy writes to achieve differential obliviousness and batch processing to reduce number of ecall and ocall switch overhead.

Adore's parallel version is 1.45 – 1.64x faster than its single-thread version on query 3. This speedup mainly comes from less encryption and decryption overhead by applying parallel encryption and decryption technique.

8 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 [70], and StealthDB[45]. 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 [82] 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. Other oblivious systems focus on different kind of workload or have different security guarantee. For example, Obladi [29] focuses on providing ACID transactions instead of analytical workloads; federated oblivious database systems [13, 14, 31, 77] provide cooperative data analytics for untrusted parties (semi-honest or malicious). They are out of the scope of this work.

ORAM and Oblivious Algorithms Oblivious RAM and oblivious computation was first pioneered in the seminar work by Goldreich [42]. Since then, many ORAM schemes and hardware implementations emerges, such as Path ORAM [73], Ring ORAM [68], and PrORAM [79]. Despite many works, including the above ones, makes ORAM more efficient. Using ORAM still pays a $\log(N)$ factor slow down. For database that could potentially have billions of tuples, this overhead is still significant. In addition, using ORAM doesn't automatically make the algorithm itself oblivious. Apart from ORAM, many other oblivious data structures have been proposed, such as oblivious priority queues [51, 71]. Apart from differential obliviousness [22], Allen et al. [5] proposed a security model, ODP, which combines differential obliviousness and differential privacy. This model is useful when both published result and memory access pattern need to be protected.

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 [52], GUPT [62], PrivateSQL [56]. In this paper, we uses 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 [74] and the basic composition [37].

Cache-Timing Attacks. Timing side channel, which is out of the scope of the definition of differential obliviousness, could leak sensitive information, too. Specifically, many 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 [17, 35, 69, 80, 81]. Fortunately, we can harden Adore against cache-timing attacks without dramatic changes. The overall idea is to make the the algorithms and data structures within private memory oblivious as well. For example, we can use cache oblivious sort such as [23] to replace the bucket oblivious sort used in Algorithm 4 and Algorithm 5. We can also use oblivious priority queues such as [71] to implement the priority queue in Algorithm 3. As a result, we could harden Adore against cache-timing attacks at the cost of a slight performance overhead.

9 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. We design and implement Adore, the first database system that satisfies *differential obliviousness*, a novel obliviousness property which ensures that memory access patterns satisfy differential privacy. Adore consists of a series of differentially oblivious algorithms for fundamental relational database operators, which have significantly lower query completion time compared to state-of-art fully oblivious alternatives. Adore also encrypts and decrypts data in parallel to enable high-performance database I/O. Our evaluations show that Adore outperforms the state-of-the-art fully oblivious databases by up to 17x on Big Data Benchmarks, and can scale to run on 10x larger database with the same hardware configuration. Adore's source code will be publicly available.

REFERENCES

[1] 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. Inference and record-injection attacks on searchable encrypted relational databases. *IACR Cryptol. ePrint Arch.*, 2017:24, 2017.

[3] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

[4] Alibaba. 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. An algorithmic framework for differentially private data analysis on trusted processors. In *NeurIPS*, pages 13635–13646, 2019.

[6] UC Berkeley AMP Lab. 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. Azure SQL database always encrypted. In *SIGMOD*, pages 1511–1525, 2020.

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

[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. Spark SQL: relational data processing in spark. In *SIGMOD*, pages 1383–1394, 2015.

[10] Gilad Asharov, T.-H. Hubert Chan, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. Bucket oblivious sort: An extremely simple oblivious sort. In *SOSA@SODA*, pages 8–14, 2020.

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

[12] Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, D. Sivakumar, and Luca Trevisan. Counting distinct elements in a data stream. In *RANDOM*, pages 1–10, 2002.

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

[14] Johes Bater, Xi He, William Ehrich, Ashwin Machanavajjhala, and Jennie Rogers. Shrinkwrap: Efficient sql query processing in differentially private data federations. *VLDB*, 12(3):307–320, 2018.

[15] Randall G. Bello, Karl Dias, Alan Downing, James J. Feenan Jr., James L. Finnerty, William D. Norcott, Harry Sun, Andrew Witkowski, and Mohamed Ziauddin. Materialized views in oracle. In *VLDB*, pages 659–664, 1998.

[16] Omri Ben-Eliezer, Rajesh Jayaram, David P. Woodruff, and Eylon Yogev. A framework for adversarially robust streaming algorithms. In *PODS*, pages 63–80, 2020.

[17] Daniel J Bernstein. Cache-timing attacks on aes. 2005.

[18] Chris Bing. Atos, it provider for winter olympics, hacked months before opening ceremony cyberattack. https://www.cyberscoop.com/atos-olympics-hack-olympic-destroyer-malware-peyongchang/. Accessed: 2020-09-10.

[19] Raphael Bost. $\sum o\varphi o\varsigma$: Forward secure searchable encryption. In *CCS*, pages 1143–1154, 2016.

[20] Raphaël Bost, Brice Minaud, and Olga Ohrimenko. Forward and backward private searchable encryption from constrained cryptographic primitives. In *CCS*, pages 1465–1482, 2017.

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

[22] T.-H. Hubert Chan, Kai-Min Chung, Bruce M. Maggs, and Elaine Shi. Foundations of differentially oblivious algorithms. In *SODA*, pages 2448–2467, 2019.

[23] T.-H. Hubert Chan, Yue Guo, Wei-Kai Lin, and Elaine Shi. Cache-oblivious and data-oblivious sorting and applications. In *SODA*, pages 2201–2220, 2018.

[24] T.-H. Hubert Chan, Elaine Shi, and Dawn Song. Private and continual release of statistics. In *ICALP*, pages 405–417, 2010.

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

[26] Lijie Chen, Badih Ghazi, Ravi Kumar, and Pasin Manurangsi. On distributed differential privacy and counting distinct elements. 2020.

[27] Sara Cohen, Werner Nutt, and Alexander Serebrenik. Rewriting aggregate queries using views. In *PODS*, pages 155–166, 1999.

[28] Victor Costan, Ilia A. Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security*, pages 857–874, 2016.

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

[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. The snowflake elastic data warehouse. In *SIGMOD*, pages 215–226, 2016.

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

[32] Jessica Davis. Inadequate security, policies led to lifelabs data breach of 15m patients. https://healthitsecurity.com/news/inadequate-security-policies-led-to-lifelabs-data-breach-of-15m-patients. Accessed: 2020-09-10.

[33] Jessica Davis. 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. Practical private range search revisited. In *SIGMOD*, pages 185–198, 2016.

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

[36] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam D. Smith. Calibrating noise to sensitivity in private data analysis. In *TCC*, pages 265–284, 2006.

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

[38] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. Rappor: Randomized aggregatable privacy-preserving ordinal response. In *CCS*, pages 1054–1067, 2014.

[39] Saba Eskandarian and Matei Zaharia. Oblidb: Oblivious query processing for secure databases. *VLDB*, page 169–183, 2019.

[40] Matthieu Giraud, Alexandre Anzala-Yamajako, Olivier Bernard, and Pascal Lafourcade. Practical passive leakage-abuse attacks against symmetric searchable encryption. In *SECRYPT*, pages 200–211, 2017.

[41] Vindu Goel and Nicole Perlroth. 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. Towards a theory of software protection and simulation by oblivious rams. In *STOC*, pages 182–194, 1987.

[43] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.

[44] Jonathan Goldstein and Per-Åke Larson. Optimizing queries using materialized views: A practical, scalable solution. In *SIGMOD*, pages 331–342, 2001.

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

[46] Paul Grubbs, Richard McPherson, Muhammad Naveed, Thomas Ristenpart, and Vitaly Shmatikov. Breaking web applications built on top of encrypted data. In *CCS*, pages 1353–1364, 2016.

[47] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. Amazon redshift and the case for simpler data warehouses. In *SIGMOD*, pages 1917–1923, 2015.

[48] Avinatan Hassidim, Haim Kaplan, Yishay Mansour, Yossi Matias, and Uri Stemmer. Adversarially robust streaming algorithms via differential privacy. *arXiv preprint arXiv:2004.05975*, 2020.

[49] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963.

[50] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS*, 2012.

[51] Zahra Jafargholi, Kasper Green Larsen, and Mark Simkin. Optimal oblivious priority queues and offline oblivious RAM. *IACR Cryptol. ePrint Arch.*, 2019:237, 2019.

[52] Noah Johnson, Joseph P. Near, and Dawn Song. Towards practical differential privacy for sql queries. *VLDB*, page 526–539, 2018.

[53] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. Generic attacks on secure outsourced databases. In *CCS*, pages 1329–1340, 2016.

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

[55] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *CRYPTO*, pages 104–113, 1996.

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

[57] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions

and software model for isolated execution. In *The Second Workshop on Hardware and Architectural Support for Security and Privacy 2013*, page 10, 2013.
[58] Frank D. McSherry. Privacy integrated queries: An extensible platform for privacy-preserving data analysis. In *SIGMOD*, pages 19–30, 2009.
[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. Dremel: A decade of interactive SQL analysis at web scale. *VLDB*, pages 3461–3472, 2020.
[60] Microsoft. 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] Hoshi Mistry, Prasan Roy, S. Sudarshan, and Krithi Ramamritham. Materialized view selection and maintenance using multi-query optimization. In *SIGMOD*, pages 307–318, 2001.
[62] Prashanth Mohan, Abhradeep Thakurta, Elaine Shi, Dawn Song, and David Culler. Gupt: Privacy preserving data analysis made easy. In *SIGMOD*, pages 349–360, 2012.
[63] Olga Ohrimenko, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Markulf Kohlweiss, and Divya Sharma. Observing and preventing leakage in mapreduce. In *CCS*, pages 1570–1581, 2015.
[64] Dan Page. Theoretical use of cache memory as a cryptanalytic side-channel. *IACR Cryptol. ePrint Arch.*, 2002:169, 2002.
[65] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. Arx: An encrypted database using semantically secure encryption. *VLDB*, 12(11):1664–1678, 2019.
[66] Raluca Ada Popa, Catherine MS Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *SOSP*, pages 85–100, 2011.
[67] Christian Priebe, Kapil Vaswani, and Manuel Costa. Enclavedb: A secure database using sgx. In *SP (Oakland)*, pages 264–278, 2018.
[68] Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Ring ORAM: closing the gap between small and large client storage oblivious RAM. *IACR Cryptol. ePrint Arch.*, 2014:997, 2014.
[69] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *CCS*, pages 199–212, 2009.
[70] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. Vc3: Trustworthy data analytics in the cloud using sgx. In *2015 IEEE Symposium on Security and Privacy*, pages 38–54. IEEE, 2015.
[71] Elaine Shi. Path oblivious heap: Optimal and practical oblivious priority queue. In *IEEE Symposium on Security and Privacy*, pages 842–858, 2020.
[72] Martin Staudt and Matthias Jarke. Incremental maintenance of externally materialized views. In *VLDB*, pages 75–86, 1996.
[73] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *CCS*, pages 299–310, 2013.
[74] Salil P. Vadhan. The complexity of differential privacy. In *Tutorials on the Foundations of Cryptography*, pages 347–450. 2017.
[75] Jelle van den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: scalable private messaging resistant to traffic analysis. In *SOSP*, pages 137–152, 2015.
[76] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *SIGMOD*, pages 1041–1052, 2017.
[77] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. Conclave: secure multi-party computation on big data. In *EuroSys*, pages 3:1–3:18, 2019.
[78] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *SP (Oakland)*, pages 640–656, 2015.
[79] Xiangyao Yu, Syed Kamran Haider, Ling Ren, Christopher W. Fletcher, Albert Kwon, Marten van Dijk, and Srinivas Devadas. Proram: dynamic prefetcher for oblivious RAM. In *ISCA*, pages 616–628, 2015.
[80] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-vm side channels and their use to extract private keys. In *CCS*, pages 305–316, 2012.
[81] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in paas clouds. In *CCS*, pages 990–1003, 2014.
[82] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *NSDI*, pages 283–298, 2017.

A 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 §A.1. Next, we define (ϵ, δ)-sensitivity and introduce the Laplacian mechanism for (ϵ, δ)-sensitivity in §A.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 based on [12] (Algorithm 7) and also demonstrate a simpler 1.1 approximation version (Algorithm 8).

A.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 $x_{(1)} \le x_{(2)} \le \cdots \le x_{(n)}$ 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 $1 \le t \le n/2$. Our goal is to prove that $|\frac{1}{y_t} - \frac{1}{y_{t+1}}| \le O(\frac{n}{t^2})$ 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.

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

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

Proof. **Part 1.** Consider the interval $[0, \delta \frac{t}{n}]$. We have

$$\mathbb{E}[|\{i \in [n] : x_i < \delta t/n\}|] = \delta t$$

namely, the expected number of points x_i will fall in this interval is exactly δt. Then by Markov's inequality, we have

$$\Pr[|\{i \in [n] : x_i < \delta t/n\}| \ge t] \le \delta.$$

So with probability $1 - \delta$, we have $|\{i : x_i < \delta t/n\}| < t$, and conditioned on this we must have $y_t > \delta t/n$ by definition, which yields the first inequality.

Part 2. For the second inequality, note that we can write

$$|\{i \in [n] : x_i < t/(2n)\}| = \sum_{i=1}^{n} z_i$$

where $z_i \in \{0, 1\}$ is an random variable that indicates the event that $x_i < t/2n$. Moreover, $\mathbb{E}[\sum_{i=1}^n z_i] = t/2$. Applying Chernoff bounds, we have

$$\Pr[|\{i \in [n] : x_i < t/(2n)\}| \ge t] \le \exp(-t/6)$$

Which proves the second inequality. □

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

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

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

Proof. Note that we can first condition on any realization of the values $y_1, y_2, \ldots, y_t$ one by one. Now that these values are fixed, the remaining distribution of the $(n-t)$ uniform variables is the same as drawing $(n-t)$ uniform random variables independently from the interval $[y_t, 1]$. Now observed that for any of the remaining $n-t$ uniform variables x_i, the probability that $x_i \in [y_t, y_t + \alpha/n]$ is at least $\frac{\alpha}{n}$, which follows from the fact that x_i is drawn uniformly from $[y_t, 1]$. Thus,

$$\begin{aligned}
\Pr[|\{i \in S : x_i \in [y_t, y_t + \alpha/n]\}| = 0] &\leq (1-\alpha/n)^{n-t} \\
&\leq (1-\alpha/n)^{n/2} \\
&= \exp(\frac{n}{2}\log(1-\alpha/n)) \\
&< \exp(\frac{n}{2}(-\alpha/n + 2(\alpha/n)^2)) \\
&< \exp(-\frac{n}{2}\frac{\alpha}{2n}) \\
&= \exp(-\alpha/4).
\end{aligned}$$

Thus $|\{i \in S : x_i \in [y_t, y_t + \alpha/n]\}| \geq 1$ with probability at least $1 - e^{-\alpha/4}$. Conditioned on this, we must have $y_{t+1} < y_t + \alpha/n$, as desired.

□

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

Proof. **Part 1.** For the first statement, we condition on $y_t > \beta\frac{t}{n}$ and, $y_{t+1} < y_t + \frac{\alpha}{n}$, which by a union bound hold together with probability $1 - \beta - e^{-\frac{\alpha}{4}}$ by Claims A.1 and A.2. Define the value t' such that $y_t = \frac{t'}{n}$. By the above conditioning, we know that $t' > \beta t$. Conditioned on this, we have

$$\begin{aligned}
\left|\frac{1}{y_t} - \frac{1}{y_{t+1}}\right| &< \frac{n}{t'} - \frac{1}{t'/n + \alpha/n} \\
&= \frac{n}{t'} - \frac{n}{t' + \alpha} \\
&= \frac{n}{t'}\left(1 - \frac{1}{1 + \alpha/t'}\right) \\
&< \frac{n}{t'}\left(1 - (1 - \alpha/t')\right) \\
&\leq \frac{\alpha n}{(t')^2} \\
&\leq \frac{\alpha n}{\beta^2 t^2}
\end{aligned} \tag{1}$$

Where we used that $\alpha/t' < \alpha/(\beta t) < 1/2$, and the fact that $1/(1+x) > 1 - x$ for any $x \in (0,1)$.

Part 2. For the second part, we condition on $y_t > \frac{t}{2n}$ and, $y_{t+1} < y_t + \frac{\alpha}{n}$, which by a union bound hold together with probability $1 - e^{-\frac{t}{6}} - e^{-\frac{\alpha}{4}}$ by Claims A.1 and A.2. Then from Lemma 1:

$$\begin{aligned}
\left|\frac{1}{y_t} - \frac{1}{y_{t+1}}\right| &\leq \frac{\alpha n}{\beta^2 t^2} \\
&= 4\alpha\frac{n}{t^2}
\end{aligned}$$

In this case, the same inequality goes throguh above with the setting $\beta = 1/2$, which finishes the proof. □

Remark A.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 A.1, which will be tight up to a (small) constant.

A.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 A.5. Let $f : \mathcal{X} \to \mathbb{R}$ be a function. We say that f has sensitivity ℓ if for every two neighboring databases $X, X' \in \mathcal{X}$, we have $|f(X) - f(X')| \leq \ell$.

Theorem A.6 (The Laplace Mechanism [?]). *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 $X, X' \in \mathcal{X}$. Now consider the following definition.

Definition A.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 A.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.*

Proof. Fix any neighboring datasets $X, X' \in \mathcal{X}$. Let $R^* = R_X \cap R_{X'}$ where $R_X, R_{X'}$ are in Definition A.7. Since$|R_X| > (1-\delta)|R|$ and $|R_{X'}| > (1-\delta)|R|$, we have $|R_X \cap R_{X'}| > (1 - 2\delta)|R|$. Now fix any $r \in R^*$. By Definition A.7, we know that $|\mathcal{A}(X, r) - \mathcal{A}(X', r)| < \ell$.

From here, we follow the standard proof of correctness of the Laplacian mechanism by bounding the ratio

$$\frac{\Pr[\mathcal{A}(X, r) + \mathrm{Lap}(0, \frac{\ell}{\epsilon}) = z]}{\Pr[\mathcal{A}(X', r) + \mathrm{Lap}(0, \frac{\ell}{\epsilon}) = z]}$$

for any $z \in \mathbb{R}$.

In what follows, set $b = \frac{\ell}{\epsilon}$

$$
\begin{aligned}
&\frac{\Pr[\mathcal{A}(X,r) + \mathrm{Lap}(0,b) = z]}{\Pr[\mathcal{A}(X',r) + \mathrm{Lap}(0,b) = z]} \\
&= \frac{\Pr[\mathrm{Lap}(0,b) = z - \mathcal{A}(X,r)]}{\Pr[\mathrm{Lap}(0,b) = z - \mathcal{A}(X',r)]} \\
&= \frac{\frac{1}{2b}\exp(-|z - \mathcal{A}(X,r)|/b)}{\frac{1}{2b}\exp(-|z - \mathcal{A}(X',r)|/b)} \\
&= \exp\left((|z - \mathcal{A}(X',r)| - |z - \mathcal{A}(X,r)|)/b\right) \\
&\leq \exp\left(|\mathcal{A}(X,r) - \mathcal{A}(X',r)|/b\right) \\
&\leq \exp(\ell/b) \\
&\leq e^{\epsilon},
\end{aligned}
$$

where the forth step follows from triangle inequality $|x| - |y| \leq |x - y|$, the last step follows from $\ell/b = \epsilon$.

It follows that for any set $S \subset \mathbb{R}$ and any $r \in R^*$, we have

$$\Pr\left[\mathcal{A}(X,r) + \mathrm{Lap}(0,b) \in S\right] \leq e^{\epsilon} \cdot \Pr\left[\mathcal{A}(X',r) + \mathrm{Lap}(0,b) \in S\right],$$

where the randomness is taken over the generation of the Laplacian random variable $\mathrm{Lap}(0,b)$. Since this holds for all $r \in R^*$, in particular it holds for a random choice of $r \in R^*$, thus we have

$$
\begin{aligned}
&\Pr_{Z\sim\mathrm{Lap}(0,b), r\sim R^*}\left[\mathcal{A}(X,r) + Z \in S\right] \\
&\leq e^{\epsilon} \cdot \Pr_{Z\sim\mathrm{Lap}(0,b), r\sim R^*}\left[\mathcal{A}(X',r) + Z \in S\right] \qquad (2)
\end{aligned}
$$

Now since $|R^*| \geq (1 - 2\delta)|R|$, by the law of total probability we have

$$
\begin{aligned}
&\Pr_{Z\sim\mathrm{Lap}(0,b), r\sim R}\left[\mathcal{A}(X,r) + Z \in S\right] \\
=& \Pr_{Z\sim\mathrm{Lap}(0,b), r\sim R^*}\left[\mathcal{A}(X,r) + Z \in S\right] \cdot \Pr[r \in R^*] \\
&+ \Pr_{Z\sim\mathrm{Lap}(0,b), r\sim R\backslash R^*}\left[\mathcal{A}(X,r) + Z \in S\right] \cdot \Pr[r \notin R^*] \\
<& \Pr_{Z\sim\mathrm{Lap}(0,b), r\sim R^*}\left[\mathcal{A}(X,r) + Z \in S\right] \\
&+ \Pr_{Z\sim\mathrm{Lap}(0,b), r\sim R\backslash R^*}\left[\mathcal{A}(X,r) + Z \in S\right] \cdot 2\delta \\
\leq& \Pr_{Z\sim\mathrm{Lap}(0,b), r\sim R^*}\left[\mathcal{A}(X,r) + Z \in S\right] + 2\delta \qquad (3)
\end{aligned}
$$

Similarly, it follows that

$$
\begin{aligned}
&\Pr_{Z\sim\mathrm{Lap}(0,b), r\sim R}\left[\mathcal{A}(X',r) + Z \in S\right] \\
>& \Pr_{Z\sim\mathrm{Lap}(0,b), r\sim R^*}\left[\mathcal{A}(X',r) + Z \in S\right](1 - 2\delta) \\
\geq& \Pr_{Z\sim\mathrm{Lap}(0,b), r\sim R^*}\left[\mathcal{A}(X',r) + Z \in S\right] - 2\delta \qquad (4)
\end{aligned}
$$

where the last step follows from probability $\Pr[] \leq 1$.

Combining Eq. (2), (3) and (4), we have

$$
\begin{aligned}
&\Pr_{Z\sim\mathrm{Lap}(0,b), r\sim R}\left[\mathcal{A}(X,r) + Z \in S\right] \\
\leq& \Pr_{Z\sim\mathrm{Lap}(0,b), r\sim R^*}\left[\mathcal{A}(X,r) + Z \in S\right] + 2\delta \\
\leq& e^{\epsilon} \cdot \Pr_{Z\sim\mathrm{Lap}(0,b), r\sim R^*}\left[\mathcal{A}(X',r) + Z \in S\right] + 2\delta \\
\leq& e^{\epsilon} \cdot \left(\Pr_{Z\sim\mathrm{Lap}(0,b), r\sim R}\left[\mathcal{A}(X',r) + Z \in S\right] + 2\delta\right) + 2\delta
\end{aligned}
$$

where the first step follows from Eq. (3), the second step follows Eq. (2), and the last step follows from Eq. (4).

Now recall that for the actual laplacian mechanism algorithm $\overline{\mathcal{A}}$, for any database X we have

$$\Pr[\overline{\mathcal{A}}(X) \in S] = \Pr_{Z\sim\mathrm{Lap}(0,b), r\sim R}\left[\mathcal{A}(X,r) + Z \in S\right],$$

which completes the proof that $\overline{\mathcal{A}}$ is $(\epsilon, 2(1 + e^{\epsilon})\delta)$-differentially private.

□

A.3 Analysis of Distinct Count ([12])

In this section, we thoroughly analyze the properties of Distinct Count [12]. We first describe the algorithm in Algorithm 6. Then we prove its (ℓ, δ)-sensitivity and a tighter (ϵ, δ)-approximation result (compared with the approximation result in [12]).

Algorithm 6 Distinct Count [12]

```
procedure DistinctCount(I, t)                       ▷ Lemma A.9
    d ← ∅                               ▷ d is a priority-queue of size t
    for x_i ∈ I do
        y ← h(x_i)                      ▷ h: [m] → [0, 1], is a PRF
        if |d| < t then
            d.push(y)
        else if y < d.top() ∧ y ∉ d then
            d.pop()
            d.push(y)
        end if
    end for
    v ← d.top()
    return t/v
end procedure
```

Sensitivity of distinct count

Lemma A.9 (Sensitivity of DistinctCount)**.** *Assume $r \in R$ is the source of randomness of the PRF in DistinctCount (Algorithm 6), 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.*

Proof. We denote DistinctCount (Algorithm 6) $F : \mathcal{X} \times R \to \mathbb{R}$, and define the same y_t as Section A.1. Thus, for two neighboring database $X, X' \in \mathcal{X}$ ($\|X\|_0 = n$):

$$|F(X,r) - F(X',r)| \leq \max\left\{\left|\frac{t}{y_t} - \frac{t}{y_{t-1}}\right|, \left|\frac{t}{y_t} - \frac{t}{y_{t+1}}\right|\right\}.$$

Part 1. From second inequality of Lemma A.3 (the case $\beta = 1/2$), for any $5 < t \leq n/2$ and $4 < \alpha < t/4$, we have:

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

It follows that

$$\begin{aligned}
\Pr\left[\left|\frac{t}{y_t}-\frac{t}{y_{t+1}}\right| \le 5\alpha\frac{n}{t}\right] &> \Pr\left[\left|\frac{t}{y_t}-\frac{t}{y_{t+1}}\right| \le 4\alpha\frac{n}{t}\right] \\
&= \Pr\left[\left|\frac{1}{y_t}-\frac{1}{y_{t+1}}\right| \le 4\alpha\frac{n}{t^2}\right] \\
&\ge 1-\exp(-t/6)-\exp(-\alpha/4) \\
&\ge 1-\exp(-(t/4)\cdot(2/3))-\exp(-\alpha/4) \\
&\ge 1-\exp(-2\alpha/3)-\exp(-\alpha/4) \\
&\ge 1-2\exp(-\alpha/4) \qquad (5)
\end{aligned}$$

Set $\alpha = 4\log(4/\delta)$ in Eq. (5):

$$\Pr\left[\left|\frac{t}{y_t}-\frac{t}{y_{t+1}}\right| \le 20\log(4/\delta)\frac{n}{t}\right] \ge 1-\delta/2$$

Part 2. Similarly, from the the second inequality of Lemma A.3 (the case $\beta = 1/2$), for any $10 < t \le n/2$ and $4 < \alpha < t/4$, we have:

$$\Pr\left[\left|\frac{1}{y_{t-1}}-\frac{1}{y_t}\right| \le 4\alpha\frac{n}{(t-1)^2}\right] \ge 1-\exp(-(t-1)/6)-\exp(-\alpha/4)$$

From $t > 16$, we know $0.8t^2 < (t-1)^2$ and $t-1 > 0.75t > 0$. Thus:

$$\begin{aligned}
\Pr\left[\left|\frac{t}{y_t}-\frac{t}{y_{t-1}}\right| \le 5\alpha\frac{n}{t}\right] &= \Pr\left[\left|\frac{t}{y_t}-\frac{t}{y_{t-1}}\right| \le 4\alpha\frac{nt}{0.8t^2}\right] \\
&> \Pr\left[\left|\frac{t}{y_t}-\frac{t}{y_{t-1}}\right| \le 4\alpha\frac{nt}{(t-1)^2}\right] \\
&= \Pr\left[\left|\frac{1}{y_{t-1}}-\frac{1}{y_t}\right| \le 4\alpha\frac{n}{(t-1)^2}\right] \\
&\ge 1-\exp(-(t-1)/6)-\exp(-\alpha/4) \\
&\ge 1-\exp(-0.75t/6)-\exp(-\alpha/4) \\
&\ge 1-\exp(-\alpha/2)-\exp(-\alpha/4) \\
&\ge 1-2\exp(-\alpha/4) \qquad (6)
\end{aligned}$$

Set $\alpha = 4\log(4/\delta)$ in Eq. (6):

$$\Pr\left[\left|\frac{t}{y_t}-\frac{t}{y_{t+1}}\right| \le 20\log(4/\delta)\cdot\frac{n}{t}\right] \ge 1-\delta/2$$

Part 3. Now apply union bound combining the results of **Part 1.** and **Part 2.**. Hence, for any X, $0 < \delta < 1$, $16 < t < n/2$:

$$\Pr\left[|F(X,r)-F(X',r)| \le 20\log(4/\delta)\cdot\frac{n}{t}\right] \le 1-\delta.$$

Now, we proved the sensitivity of Algorithm 6. □

Lemma for approximation guarantees

Lemma A.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*

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

Proof. We define I_1 and I_2 as follows

$$I_1 = \ [0, \frac{t}{n(1+\eta)}],\ I_2 = \ [0, \frac{t}{n(1-\eta)}].$$

First note that if $x \sim [0,1]$, $\Pr[x \in I_1] = \frac{t}{n(1+\eta)}$. Since we have n independent trials, setting $Z = |\{x_i : i \in I_1\}|$ we have $\mathbb{E}[Z] = \frac{t}{(1+\eta)}$.

Then by the upper Chernoff bound, we have

$$\Pr[Z > t] \le \exp\left(-\frac{\eta^2 t}{3(1+\eta)}\right) \le 1-\delta/2.$$

Similarly, setting $Z' = |\{x_i : i \in I_2\}|$, we have $\mathbb{E}[Z'] = \frac{t}{(1-\eta)}$, so by the lower Chernoff bound, we have

$$\Pr[Z' < t] \le \exp(-\eta^2 t/2) \le 1-\delta/2.$$

Thus by a union bound, we have both that $Z < t$ and $Z' > t$ with probability $1-\delta$. Conditioned on these two events, it follows that $y_t \notin I_1$ but $y_t \in I_2$, which implies that $\frac{t}{n(1+\eta)} < y_t < \frac{t}{n(1-\eta)}$, and so we have

$$(1-\eta)n < \frac{t}{y_t} < (1+\eta)n$$

as desired.

□

A.4 Differentially Private Distinct Count

Algorithm 7 DPDistinctCount: Differentially Private Distinct Count

```
procedure DPDistinctCount(I, ϵ, η, δ)                ▷ Theorem A.11
    pQueue ← ∅                ▷ pQueue is a priority-queue of size t
                                                  ▷ t ≥ max (3(1 +
η/4)(η/4)^{-2} log(6/δ), 20ϵ^{-1}(η/4)^{-1} log(24(1 + e^{-ϵ})/δ) log(3/δ))
    for x_i ∈ I do
        y ← h(x_i)                        ▷ h: [m] → [0, 1], is a PRF
        if |pQueue| < t then
            pQueue.push(y)
        else if y < pQueue.top() ∧ y ∉ pQueue then
            pQueue.pop()
            pQueue.push(y)
        end if
    end for
    v ← pQueue.top()
    ct ← (1 + (3/4)η) t/v + Lap(20ϵ^{-1} (n/t) log(24(1 + e^{-ϵ})/δ))
    return ct
end procedure
```

Theorem A.11 (main result)**.** *For any $0 < \epsilon < 1$, $0 < \eta < 1/2$, $0 < \delta < 1/2$, there is an distinct count algorithm (Algorithm 7) such that:*

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

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

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.

Proof. Let $\widetilde{F}_0$ be the result output by the original algorithm (Algorithm 6) with the same t. Our differentially private distinct count algorithm (Algorithm 7) essentially output $\widetilde{A} = (1 + \frac{3}{4}\eta)\widetilde{F}_0 + \mathrm{Lap}(\ell/\epsilon)$, where:

$$\ell = 20\frac{n}{t}\log(24(1+e^{-\epsilon})/\delta)$$
$$t = \max\Big\{3(1+\eta/4)(\eta/4)^{-2}\log(6/\delta),$$
$$20\epsilon^{-1}(\eta/4)^{-1}\cdot\log(24(1+e^{-\epsilon})/\delta)\cdot\log(3/\delta)\Big\}$$

From Lemma A.9, we know that for any $16 < t < n/2$, the original distinct count algorithm (Algorithm 6)

$$\left(20\log(4/\delta)\cdot\frac{n}{t}\ ,\ \delta\right) - \text{sensitive}.$$

After rescaling δ by a constant factor, it can be rewritten as

$$\left(20\log(24(1+e^{-\epsilon})/\delta)\cdot\frac{n}{t}\ ,\ \frac{\delta}{6(1+e^{-\epsilon})}\right) - \text{sensitive}.$$

Then, from Lemma A.8, we know that it becomes $(\epsilon, \delta/3)$-DP by adding $\mathrm{Lap}(\ell/\epsilon)$ noise when outputting the estimated count, where ℓ is as defined above, which completes the proof of the first part of the Theorem.

Next, from Lemma A.10 and the fact that $t \geq 3(1+\eta/4)\cdot(\eta/4)^{-2}\cdot\log(6/\delta)$, we know that $(1-\frac{\eta}{4})n < \widetilde{F}_0 < (1+\frac{\eta}{4})n$ with probability at least $1-\delta/3$.

Since $(1-\frac{1}{4}\eta)(1+\frac{3}{4}\eta) \leq 1 + \frac{1}{4}\eta$ for any $0 < \eta \leq 1$, we get:

$$\Pr\left[(1+\frac{1}{4}\eta)\cdot n < (1+\frac{3}{4}\eta)\widetilde{F}_0 < (1+\frac{1}{2}\eta)\cdot n\right] \geq 1-\delta/3$$

Next, using the exponential $\Theta(e^{-x})$ tails of the Laplace distribution, and the fact that $\ell = \Omega(\log(1/\delta)\frac{n}{t})$ and $t = \Omega(\epsilon^{-1}\eta^{-1}\log^2(1/\delta))$, we have:

$$\Pr\left[|\mathrm{Lap}(\ell/\epsilon)| > \frac{1}{4}\eta n\right] \leq \delta/3.$$

Conditioned on both event that

$$|\mathrm{Lap}(\ell/\epsilon)| < \frac{1}{4}\eta n \text{ and}$$
$$(1+\frac{1}{4}\eta)\cdot n < (1+\frac{3}{4}\eta)\cdot\widetilde{F}_0 < (1+\frac{1}{2}\eta)\cdot n$$

which hold together with probability $1-\delta$ by a union bound, it follows that the estimate $\widetilde{A}$ of the algorithm indeed satisfies $n \leq \widetilde{A} \leq (1+\eta)n$, which completes the proof of the approximation guarantee in the second part of the Theorem. Finally, the space bound follows from the fact that the algorithm need only store the identities of the t smallest hashes in the data stream, which requires $O(t\log n)$ bits of space, yielding the bound as stated in the theorem after plugging in

$$t = \Theta\Big((\eta^{-2} + \epsilon^{-1}\eta^{-1}\log(1/\delta))\cdot\log(1/\delta)\Big).$$

Thus, we complete the proof. □

Claim A.12. *For any $0 < \delta \leq 10^{-3}$, $0.1 \leq \eta < 1$ and $0 < \epsilon < 1$, then we have*

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

Algorithm 8 1.1-APPROX. DPDISTINCTCOUNT

```
procedure 1.1-Approx. DPDC(I, ϵ, δ)                    ▷ Lemma A.13
    pQueue ← ∅               ▷ pQueue is a priority-queue of size t
                      ▷ t = 10^3 ϵ^{-1} log(24(1 + e^{-ϵ})/δ) log(3/δ))
    for x_i ∈ I do
        y ← h(x_i)                     ▷ h: [m] → [0, 1], is a PRF
        if |pQueue| < t then
            pQueue.push(y)
        else if y < pQueue.top() ∧ y ∉ pQueue then
            pQueue.pop()
            pQueue.push(y)
        end if
    end for
    v ← pQueue.top()
    ct ← 1.075 t/v + Lap(0.02n/log(3/δ))
    return ct
end procedure
```

Proof. From $0 < \delta < 1$, we know:

$$\log(6/\delta) \leq 2\log(3/\delta)$$

It follows:

$$\begin{aligned}\text{LHS} &\leq 3\big[(1+\eta/4)\cdot(\eta/4)^{-1}\big]\cdot(\eta/4)^{-1}\cdot 2\log(3/\delta)\\ &\leq 6(4/\eta+1)\cdot(\eta/4)^{-1}\cdot\log(3/\delta)\end{aligned}$$

From $\eta \geq 0.1$, we know $4/\eta + 1 \leq 41$. From $\delta \leq 10^{-3}$, we also know $\log(24/\delta) \geq 10$. Thus:

$$\begin{aligned}LHS &\leq 246(\eta/4)^{-1}\cdot\log(3/\delta)\\ &\leq 25\cdot 10\cdot(\eta/4)^{-1}\cdot\log(3/\delta)\\ &\leq 25\log(24/\delta)\cdot(\eta/4)^{-1}\cdot\log(3/\delta)\\ &\leq 25\log(24(1+e^{-\epsilon})/\delta)\cdot(\eta/4)^{-1}\cdot\log(3/\delta)\\ &\leq 25\epsilon^{-1}(\eta/4)^{-1}\cdot\log(24(1+e^{-\epsilon})/\delta)\cdot\log(3/\delta)\end{aligned}$$

Now we completes the proof. □

Lemma A.13. *For any $0 < \epsilon < 1$, $0 < \delta \leq 10^{-3}$, there is an distinct count algorithm (Algorithm 8) such that:*

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

$$n \leq \widetilde{A} \leq 1.1n,$$

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.

Proof. This lemma directly follows Theorem A.11 by setting $\eta = 0.1$, $0 < \delta < 10^{-3}$ and:

$$\begin{aligned}t &= 25\epsilon^{-1}(\eta/4)^{-1}\cdot\log(24(1+e^{-\epsilon})/\delta)\cdot\log(3/\delta)\\ &= 10^3\epsilon^{-1}\log(24(1+e^{-\epsilon})/\delta)\cdot\log(3/\delta)\end{aligned}$$

Thus, the scale factor in the Lap distribution (line 14 in Algorithm 7) becomes:

$$\begin{aligned}
&20\epsilon^{-1}\frac{n}{t}\log(24(1+e^{-\epsilon})/\delta)\\
&= 20\epsilon^{-1}\frac{n\log(24(1+e^{-\epsilon})/\delta)}{1000\epsilon^{-1}\log(24(1+e^{-\epsilon})/\delta)\cdot\log(3/\delta)}\\
&= 0.02n/\log(3/\delta)
\end{aligned}$$

□

B PROPERTIES OF BINOMIAL DISTRIBUTION

Fact B.1 (Tail bounds of binomial distribution). *If $X \sim B(n, p)$, that is, X is a binomially distributed random variable, where n is the total number of experiment and p is the probability of each experiment getting a successful result, and $k \geq np$, then:*

$$\Pr[X \geq k] \leq \exp(-2n(1-p-(n-k)/n)^2)$$

Proof. For $k \leq np$, from the lower tail of the CDF of binomial distribution $F(k; n, p) = \Pr[X \leq k]$, we use Hoeffding's inequality [49] to get a simple bound:

$$F(k; n, p) \leq \exp(-2n(p-k/n)^2)$$

For $k \geq np$, since $\Pr[X \geq k] = F(n-k; n, 1-p)$, we have:

$$\Pr[X \geq k] \leq \exp(-2n(1-p-(n-k)/n)^2).$$

□

Lemma B.2. *If $X \sim B(n, p)$, that is, X is a binomially distributed random variable, where n is the total number of experiment and p is the probability of each experiment getting a successful result, then*

$$\Pr[X \geq np + \sqrt{0.5n\log(1/\delta)}] \leq \delta$$

Proof. From Fact B.1, let $k = np + \sqrt{0.5n\log(1/\delta)}$, we have:

$$\begin{aligned}
&\Pr[X \geq np + \sqrt{0.5n\log(1/\delta)}]\\
\leq\ &\exp(-2n(1-p-(n-(np+\sqrt{-0.5n\log(1/\delta)}))/n)^2)\\
=\ &\exp(-2n\frac{1}{2n}\log(1/\delta))\\
=\ &\delta.
\end{aligned}$$

□

C DIFFERENTIAL OBLIVIOUSNESS PROOFS

Theorem C.1 (Main result for filter). *(restating Theorem 4.1) For any $\epsilon \in (0,1)$, $\delta \in (0,1)$ and input I with N tuples , there is an (ϵ, δ)-differentially oblivious and distance preserving filtering algorithm (DoFilter in Algorithm 1) that uses $O(\log(1/\epsilon)\cdot\log^{1.5}N\cdot\log(1/\delta))$ private memory and $(N+R)/B$ cache complexity*

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 $O(\log(1/\epsilon)\cdot\log^{1.5}N\cdot\log(1/\delta))$ private memory. Let $s = O(\log(1/\epsilon)\cdot\log^{1.5}N\cdot\log(1/\delta))$. 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, $Y_c - s \leq \widetilde{Y}_c \leq Y_c + s$ with $1-\delta$ probability. This leads to an important fact, for each round of batched read, with at least $1-\delta$ probability, P over-flows or P under-flows:

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

Now we can conclude that the failure probablity of Algorithm 1 is at most δ.

In Algorithm 1, the amount of read is N, the amount of write is $\widetilde{Y}_N + s$, from the utility-privacy bound from the DP oracle [24], it is $R + \mathrm{poly}\log(N)$. In addition, all the read and write in Algorithm 1 is batched, with the batch size s. Thus, the cache complexity of Algorithm 1 is $(N+R)/B$ as long as $s \geq B$. □

Theorem C.2 (Main result for group, group via hashing). *(restating Theorem 4.4) 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 3) that uses M private memory and has $N/B + 11NR/9MB$ cache complexity.*

Proof. The only information that Algorithm 1 leaks is k, the number of sequential scans of I. This only leaks $\widetilde{G}$ though, since M is public. Moreover, $\widetilde{G}$ is $(\epsilon, \delta/2)$-differentially private. Thus, Algorithm 3 is $(\epsilon, 0)$-differentially oblivious if we ignore the failure case.

Next, we prove that the failure probability of Algorithm 3 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 B), 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 3 to at most δ.

Finally, Algorithm 3 requires a sequential scan of input in preprocessing, which takes N/B I/O. In the following steps, it requires $k(M+N)/B$ I/O, 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 3 is $N/B + 11NR/9MB$.

□

Theorem C.3 (Main result for group, group via sorting). *(restating Theorem 4.5) For any $\epsilon \in (0,1)$, $\delta \in (0,1)$ and input of size N, there is an (ϵ, δ)-differentially oblivious grouping algorithm (DoGroup$_s$ in Algorithm 4) that uses $O(\log(1/\epsilon)\cdot\log^{1.5}N\cdot\log(1/\delta))$ private memory and $6(N/B)\log_B(N/B) + (N+R)/B$ cache complexity.*

Proof. The proof of differential obliviousness follows the proof of Theorem 4.1. Note that since BucketOblivousSort is fully oblivious, adding it as a preprocessing step does not change the differential obliviousness result.

The cache complexity of BucketOblivousSort is $6(N/B)\log(N/B)$. For the rest of the algorithm, the total read and write is $N+R$. And since both reads and writes are batched, as long as the batch size is greater than B, the cache complexity of the rest of the algorithm is $(N+R)/B$. Thus, we have demonstrated the cache complexity claim. □

Theorem C.4 (Main result for join)**.** *(restating Theorem 4.6) 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 foreign key join algorithm (DoJoin in Algorithm 5) that uses* $O(\log(1/\epsilon) \cdot \log^{1.5} N \cdot \log(1/\delta))$ *private memory and has* $6(N/B)\log(N/B) + (N+R)/B$ *cache complexity.*

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

DoJoin requires sorting $R'||S'$ obliviously once. It uses BucketOblivousSort, which has cache complexity $6(N/B)\log N/B$. Additionally, the DoFiler algorithm it uses to get rid of filler tuples has cache complexity $(N+R)/B$. Thus, the cache complexity of DoJoin is $6(N/B)\log(N/B) + (N+R)/B$ □