Differentially Oblivious Database Joins: Overcoming the Worst-Case Curse of Fully Oblivious Algorithms

Shumo Chu[1] ✉
University of California, Santa Barbara, CA, USA

Danyang Zhuo ✉
Duke University, Durham, NC, USA

Elaine Shi ✉
Carnegie Mellon University, Pittsburgh, PA, USA

T-H. Hubert Chan ✉
The University of Hong Kong, Hong Kong

Abstract

Numerous high-profile works have shown that access patterns to even encrypted databases can leak secret information and sometimes even lead to reconstruction of the entire database. To thwart access pattern leakage, the literature has focused on *oblivious* algorithms, where obliviousness requires that the access patterns leak nothing about the input data.

In this paper, we consider the `Join` operator, an important database primitive that has been extensively studied and optimized. Unfortunately, any *fully oblivious* `Join` algorithm would require *always* padding the result to the *worst-case* length which is *quadratic* in the data size N. In comparison, an insecure baseline incurs only $O(R+N)$ cost where R is the true result length, and in the common case in practice, R is relatively short. As a typical example, when $R = O(N)$, any fully oblivious algorithm must inherently incur a prohibitive, N-fold slowdown relative to the insecure baseline. Indeed, the (non-private) database and algorithms literature invariably focuses on studying the *instance-specific* rather than *worst-case* performance of database algorithms. Unfortunately, the stringent notion of full obliviousness precludes the design of efficient algorithms with non-trivial instance-specific performance.

To overcome this worst-case performance barrier of full obliviousness and enable algorithms with good instance-specific performance, we consider a relaxed notion of access pattern privacy called (ϵ, δ)-differential obliviousness (DO), originally proposed in the seminal work of Chan et al. (SODA'19). Rather than insisting that the access patterns leak no information whatsoever, the relaxed DO notion requires that the access patterns satisfy (ϵ, δ)-differential privacy. We show that by adopting the relaxed DO notion, we can obtain efficient database `Join` mechanisms whose instance-specific performance *approximately matches* the insecure baseline, while still offering a meaningful notion of privacy to individual users. Complementing our upper bound results, we also prove new lower bounds regarding the performance of any DO `Join` algorithm.

Differential obliviousness (DO) is a new notion and is a relatively unexplored territory. Following the pioneering investigations by Chan et al. and others, our work is among the very first to formally explore how DO can help overcome the worst-case performance curse of full obliviousness; moreover, we motivate our work with database applications. Our work shows new evidence why DO might be a promising notion, and opens up several exciting future directions.

2012 ACM Subject Classification Theory of computation; Security and privacy → Cryptography; Information systems → Join algorithms; Theory of computation → Design and analysis of algorithms; Security and privacy → Mathematical foundations of cryptography

Keywords and phrases differentially oblivious, database join, instance-specific performance

Digital Object Identifier 10.4230/LIPIcs.ITC.2021.19

[1] (author ordering is randomized)

2nd Conference on Information-Theoretic Cryptography (ITC 2021).
Editor: Stefano Tessaro; Article No. 19; pp. 19:1–19:24
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Related Version *Full Version*: https://eprint.iacr.org/2021/593

Funding T-H. Hubert Chan is partially supported by the Hong Kong RGC under the grants 17200418 and 17201220. Elaine Shi is partially funded by NSF under award numbers 1601879 and 2128519, a faculty award from JP Morgan, an ONR YIP award, and a Packard Fellowship.

Acknowledgements We gratefully acknowledge helpful discussions and insightful feedback with Zhao Song and Lianke Qin.

1 Introduction

We consider a scenario in which a trusted client (e.g., an Intel SGX enclave or a trusted client laptop) outsources an encrypted database to an untrusted storage provider (e.g., untrusted memory or a cloud server). The client would like to make queries to the database without endangering the privacy of users in the database. Since all data contents are encrypted, the key challenge is to ensure that *access patterns* to the database do not accidentally harm individual users' privacy. Notably, plenty of recent attacks [24, 51, 52, 56, 59, 63, 70] against encrypted database systems (e.g., CryptDB [74], Cipherbase [6], and TrustedDB [14]) showed that when left unprotected, access patterns can leak highly sensitive information, and in some cases, even lead to reconstruction of the entire database. Given the importance of this problem, several high-profile works implemented *oblivious* database systems, including Opaque [90], ObliDB [41], Obladi [35], as well as the work by Arasu and Kaushik [7], where *obliviousness* requires that access patterns leak nothing about the underlying data.

In this paper, we focus on an important database operation, the `Join` operation, which has been studied extensively in the database literature [1, 13, 16, 33, 44, 55, 61, 71, 78, 89]. Given two tables and a specified attribute (henceforth also called the *join key* or *key* for short)[2], the `Join` operation computes, for each possible join key value k, the Cartesian product of the rows in each table with the join key k. For example, if the join key k appears twice in the first table and three times in the second table, then in the join result the join key k will have six occurrences. Some works focus on the special case of foreign-key join where it is promised that in one of the input tables, each key appears only once. In this paper, however, we consider the more general case where a key may have multiple occurrences in both tables.

Unfortunately, existing oblivious database systems [7, 41, 90] do not provide a satisfactory solution for the `Join` operation. A fundamental problem is that any *fully oblivious* algorithm for `Join` must always incur the *worst-case cost* even when the actual join result may be short. To see this, recall that an algorithm is said to be *oblivious* iff its memory access patterns (and runtime[3]) are indistinguishable for any two inputs of the same length [47, 48, 81]. For the case of `Join`, the result size for the worst-case input is $\Theta(N_1 \cdot N_2)$ where N_1 and N_2 denote the sizes of the two input tables, respectively. Thus any fully oblivious algorithm must incur at least $\Omega(N_1 \cdot N_2)$ cost on *any* input instance, even when the input instance has a short join result[4]. This is very expensive in real-world databases, since the join result is typically much smaller than quadratic in the common case.

In this paper, we ask the following natural question:

Can we design join algorithms that provide a meaningful and mathematically rigorous notion of privacy, and moreover, avoid having to pay the worst-case quadratic penalty on every input?

---

[2] A join key can also be a set of attributes, without loss of generality, we assume two tables are joined on single attribute in this paper.

[3] Note that the length of the physical accesses is the same as the program's runtime.

[4] The naïve solution of simulating an insecure join algorithm with Oblivious RAM does not provide full obliviousness unless the runtime is padded to the worst case.

Parametrized algorithm design and instance-specific performance. We follow the well-established paradigm in the algorithms and database literature, and focus on *instance-specific* complexity measures. *Parametraized analysis and instance-specific performance have been widely adopted in the database, algorithms, as well as cryptography literature,* and its importance has been explained in numerous prior works. For example, a line of work in the classical (non-private) database literature focused on optimizing the instance-specific performance for database joins [18, 55, 61, 71, 89] – in this context the output length is often used as an additional parameter to characterize the algorithms' performance. The study of instance-specific performance is also closely related to the prominent line of work on parametrized algorithms design and analysis [31,37,42,69,77,79] (see Roughgarden's textbook for an excellent overview [77].) Last but not the least, notable works in the cryptography literature also consider how to achieve good instance-specific performance (sometimes called "input-specific runtime" in the cryptography literature) in the Turing Machine or the Random Access Machine (RAM) models: for example, the seminal work by Goldwasser et al. [49] is motivated by "overcoming the worst-case curse" in cryptographic constructions; and a similar notion is adopted in subsequent works [5, 58].

Prior approaches introduce arbitrary leakages to achieve good instance-specific performance. As mentioned, full obliviousness *precludes* the design of algorithms with non-trivial instance-specific performance. However, prior oblivious database systems [7, 41, 90] do care about instance-specific performance bounds. To achieve good instance-specific performance, they give up on full obliviousness (even though this line of work is commonly referred to as "oblivious databases"), and introduce arbitrary leakages, e.g., by leaking the multiplicity of keys, the lengths of intermediate arrays, the final output length, and/or the exact runtime of the (non-private) program. The ramifications of such leakages are poorly understood, and can lead to unforeseen privacy breaches. Since numerous prior encrypted database systems allowing arbitrary leakages have been broken [24, 51, 52, 56, 59, 63, 70], our philosophy is to advocate for an approach that provides rigorous mathematical guarantees on the leakage.

Although our work focuses on a privately outsourced database scenario, it is interesting to note that in the cryptography literature, a line of work has focused on general RAM computations on encrypted data [20, 23, 32, 45, 46, 50]. These works also care about plugging access pattern leakage. Thus, to achieve full security, essentially full obliviousness is necessary in these constructions (and indeed these constructions rely on Oblivious RAM as a building block). Typically this line of work either pads the RAM computation to the runtime on the worst-case input, or they allow leakage of the exact runtime and thus violate full obliviousness – the ramification of such leakage is unclear and can lead to severe privacy breaches in some applications.

Overcoming the worst-case curse with differential obliviousness. Since the worst-case performance curse is inherent for full obliviousness, we would need a relaxed (but nonetheless meaningful and rigorous) privacy notion to achieve good instance-specific performance. We therefore turn our attention to the notion of *differential obliviousness* (DO) recently defined by Chan et al. [28]. Simply put, differential obliviousness requires that the access patterns revealed during a program's execution must satisfy (ϵ, δ)-differential privacy [39]. So far, a couple of prior works [17, 28] have shown theoretical separations between DO and full obliviousness, thus providing initial theoretical evidence why DO is worth studying. Besides the few pioneering investigations, the landscape of DO remains much unexplored. Designing DO algorithms, especially for practically motivated applications, is a relatively new territory.

1.1 Our Contributions and Results

We show novel DO database join algorithms that can *approximately match the instance-specific performance of the insecure baseline*, whereas any fully oblivious join algorithm must inherently incur, on common instances with $O(N)$-sized outputs, at least a linear (in database size) blowup relative to the insecure baseline. Our work is among the very first to formally explore how DO can help overcome the worst-case curse of full obliviousness.

Specifically, we present *two main upper bound results*: 1) a DO join algorithm in the standard word-RAM model where the primary performance metrics are the algorithm's runtime and output length, and 2) a DO join algorithm in the *external-memory* model where the primary performance metrics are the algorithm's cache complexity and output length. Both algorithms approximately match the performance of the insecure baselines in the corresponding setting. Both models are important to consider: the standard word-RAM model is the prevalent model in which algorithms are studied; and the external-memory model is the best fit when we rely on secure processors such as Intel's SGX to privately outsource the sensitive database to an untrusted server (as we explain more later).

We also prove *lower bound* results regarding the performance of any DO join algorithm. The lower bounds show that some small slowdown relative to the insecure baseline is necessary. Moreover, our upper bound matches the lower bound when the result size is at least quasi-linear. For other parameter regimes, e.g., when the result size is linear or shorter, there remains a small gap between our upper bounds and lower bounds – and bridging this gap is an interesting direction for future work.

We now present the result statements more formally.

Results for the word-RAM model. Recall the application scenario mentioned at the beginning of the paper: a *trusted* client stores an *encrypted* database on an *untrusted* storage. Data can only be decrypted within the trusted client which also runs the database engine. Anything fetched or written to the storage is encrypted such that the adversary can only observe the access patterns.

In the standard word-RAM model, we assume that the trusted client is a CPU with $O(1)$ private registers, and the cost is measured in terms of the number of memory words transmitted between the CPU and memory (which equates to the runtime of the algorithm). Table 1 summarizes our results for the standard RAM model. For simplicity, the results are stated for the typical parameters[5] $\epsilon = \Theta(1)$ and $\delta = 1/N^c$ for some constant $c \geq 1$ and a more generalize version will be provided in Theorem 1.

As shown in Table 1, our algorithm achieves $O(R + N \log N)$ runtime and $R + O((\mu_{\max} + \log N) \cdot \log N)$ result size where N denotes the total input length, R denotes the true result size (when the insecure algorithm is run), and $\mu_{\max}$ denotes the multiplicity of the most frequent join key in the database. Note that even an insecure join algorithm must incur at least $R + N$ runtime since it has to at least read the input and write down the output. In the common case in practice, the true output size R is small, e.g., $R = O(N)$. In this case, our DO algorithm achieves almost a factor of N performance improvement relative to any fully oblivious solution whose cost is inherently quadratic.

We also compare our algorithm with a naïve DO algorithm that basically simulates the insecure algorithm (described in Section 3.5) using the state-of-the-art *statistically* secure Oblivious RAM [29, 88][6] and then appends an appropriate noise to the result as well as

[5] In the cryptography literature, sometimes we want δ to be a negligible function in N. In this case, the $\log N$ factor in the performance bound is replaced with any super-logarithmic function.

[6] Like Chan et al. [28], we adopt a statistical notion of differential obliviousness that defends against even computationally unbounded adversaries.

Table 1 Our results: stated for the typical parameters when $\epsilon = \Theta(1)$ and $\delta = \frac{1}{N^c}$ for some constant $c \geq 1$. N_1 and N_2 denote the lengths of the two input tables, $N := N_1 + N_2$, R denotes the length of the true join result, and $\mu_{\max}$ denotes the maximum multiplicity of any join key in the two input tables. $\Theta(\cdot)$ means that it is both an upper- and lower-bound.

	Runtime	Result size
Insecure	$\Theta(R+N)$	R
Fully oblivious	$\Theta(N_1 \cdot N_2)$	$\Theta(N_1 \cdot N_2)$
Our differentially oblivious algorithms		
Naïve: Theorem 15	$O((R+N\log N)\log^2 N)$	$R+O(N\log N)$
Main scheme 1: Thm 1	$O(R+N\log N)$	$R+O((\mu_{\max}+\log N)\cdot\log N)$
LB: Thm 2	$\Omega(R+N\log\log N+\mu_{\max}\log N)$	$R+\Omega(\mu_{\max}\log N)$

to the program's runtime (see Section 3.5 for a more detailed description). In comparison, our algorithm is a $\log^2 N$ factor faster than the naïve DO algorithm. Other standard DO techniques such as the work by Komargodski and Shi [62] fail to work in our context – see Section 2.5 for more discussions. In light of our lower bound for any DO join algorithm, our upper bound achieves optimality in terms of result length as long as $\mu_{\max} \geq \log N$, and is optimal in terms of runtime when $\mu_{\max} = \Theta(N)$.

Our main theorems[7] are also informally described below for a broader range of choices for ϵ and δ than Table 1.

▶ **Theorem 1** (Our DO join algorithm). *Let R be the length of the true join result (i.e., without fillers), let $\mu_{\max}$ denote the multiplicity of the most frequent join key in either input array, and let N denote the total input length. There is an (ϵ, δ)-differentially oblivious join algorithm that runs in time $O(R + N(\log N + \frac{1}{\epsilon}\log\frac{1}{\delta}))$ and produces a result whose length is at most $R + O(\frac{1}{\epsilon}\cdot(\mu_{\max} + \frac{1}{\epsilon}\log\frac{1}{\delta})\cdot\log\frac{1}{\delta})$.*

Table 1 also shows our lower bound which states that any DO join algorithm must incur at least $\Omega(R + N\log\log N + \mu_{\max}\log N)$ runtime and must have a result size of at least $R + \Omega(\mu_{\max}\log N)$ with high probability (assuming the same typical choices of ϵ and δ). A formal statement with more general parameters is given below.

▶ **Theorem 2** (Limits of any DO join algorithm (informal)). *Let N be the total input length, then, for most reasonable choices[8] of ϵ and δ,*

1. *any (ϵ, δ)-differentially oblivious join algorithm must produce a result of at least $R + \Omega(\mu_{\max}\cdot\frac{1}{\epsilon}\cdot\log\frac{\epsilon}{\delta})$ with at least δ/ϵ probability.*
2. *any "natural" (ϵ, δ)-differentially oblivious join algorithm must have some input of total length N and whose true join result size is R, such that the algorithm incurs at least $\Omega(R + N\log\log\frac{1}{\delta} + \mu_{\max}\cdot\frac{1}{\epsilon}\cdot\log\frac{\epsilon}{\delta})$ runtime with at least δ/ϵ probability.*

In the above, the lower bound for runtime holds for a broad class of *natural* algorithms that do not perform encoding or computation on the elements' payloads – indeed, most known join algorithms fall into this class. We refer the reader to the online full version [34] for more details.

[7] In Table 1, we assume that $\delta = 1/N^c$ like the standard differentially privacy literature suggests. In some cryptographic application settings where one may desire δ to be a negligible function in N, there will be an extra (arbitrarily small) super-constant factor added to the bounds in Table 1. See also Theorem 1 for the statement with general parameters.

[8] See the formal theorem in our online full version [34] for a more precise characterization of the parameter regime in which the lower bound holds.

Table 2 Our results: cache-agnostic cache complexity. See the caption of Table 1 for the meaning of the notations N_1, N_2, N, and R. Our results need to assume the standard "tall cache" and "wide block" assumptions, i.e., $M \geq B^2$, and $B \geq \log^{0.55} N$ where M is the cache size and B is the block size.

	Cache-oblivious cache complexity
Insecure	$O(\frac{R}{B} + \frac{N}{B} \cdot \log_{\frac{M}{B}} \frac{N}{B})$
Fully oblivious	$\Theta(N_1 \cdot N_2/B)$
Our differentially oblivious algorithms	
Naïve: Theorem 15	$O\left((R + N\log N)\log N \cdot \log_B N\right)$
Main scheme 2: Cor 3	$O\left(\frac{R}{B} + \frac{N}{B} \cdot \log N\right)$

Results for in the cache-agnostic, external-memory model. The external-memory model [3, 43, 86] and cache complexity are important for scenarios where we want to employ secure processors to enable encrypted, differentially oblivious databases. Imagine that the server has a secure processor such as Intel SGX. In this case, the database is stored in an encrypted format on the server, and only the secure processor can decrypt the data and perform computation. In other words, we can think of the *trusted client* as the secure processor, and the rest of the server's software stack is untrusted. Moreover, a remote client can communicate with the secure processor using a secure channel to ask queries and receive answers back.

Interestingly, it turns out that when Intel SGX is used to outsource both the computation and storage to an untrusted server, the major performance metric is the number of pages the SGX enclave needs to fetch. Each enclave page swap is a heavy-weight operation that involves communication with the untrusted operating system, and moreover, the enclave must decrypt (or encrypt) the memory page being swapped in (or out). In this scenario, the trusted enclave memory can be viewed as a *cache* whose size is henceforth denoted M, and each page is a *block* (i.e., the atomic unit being swapped in and out) whose size is henceforth denoted B. Further, the rest of the storage outside the trusted enclave memory is the *external memory*. An algorithm's *cache complexity* is defined as the number of blocks transmitted between the cache and the external memory during the algorithm's execution. In the online full version [34], we provide additional background on the external-memory model which is a well-accepted model in the algorithms literature – it is very interesting to observe that the line of work on external-memory algorithms [3, 43, 86] is a perfect fit for studying the performance of algorithms running on commodity secure processors.

We propose a variant of our algorithm optimized for cache complexity. Our algorithm is *cache agnostic*, i.e., the algorithm is unaware of the cache's parameters, namely, M and B. Cache-agnostic was also commonly referred to as "cache-oblivious" in the algorithms literature [38, 43]. In our paper, we use the term "cache-agnostic" instead to disambiguate from our usage of the term "obliviousness". The importance and advantages of cache-agnostic algorithms have been extensively discussed in the algorithms literature [38, 43]. First, a cache-agnostic algorithm is "universal" and the performance bounds hold no matter what the system parameters (including M and B) are. Not only so, when deployed on a multi-level memory hierarchy, an optimal cache-agnostic algorithm would give optimal IO performance between any two adjacent levels of the hierarchy [38, 43].

Table 2 summarizes our cache complexity results. The insecure baseline and the naïve DO algorithm in the table are described in Section 3.5. As shown in the table, our cache complexity is quite close to that of the insecure baseline, and outperforms the naïve DO algorithm by a $(B/\log B) \cdot \log^2 N$ factor. In a typical scenario, $B = \mathsf{poly}\log N$; in this case, our improvement over the naïve DO algorithm is polylogarithmic.

Last but not the least, our cache efficient instantiation has relatively small constants in the big-O notation, and therefore an interesting future direction is to implement our algorithm and measure its concrete efficiency. We summarize our cache-complexity results in the following corollary:

▶ **Corollary 3** (Our DO join algorithm: cache complexity). *There is an (ϵ, δ)-DO database join algorithm that incurs cache complexity upper bounded by* $\frac{1}{B} \cdot O\left(N\left(\log_{\frac{M}{B}} \frac{N}{B} + \frac{1}{\epsilon}\log\frac{1}{\delta}\right) + R + \left(\frac{1}{\epsilon}\log\frac{1}{\delta}\right)^2\right)$, *assuming the standard tall cache assumption $M = \Omega(B^2)$ and the wide block assumption $B = \Omega(\log^{0.55} N)$.*

Both the tall cache assumption and the wide block assumption are standard assumptions adopted commonly in the external-memory algorithms line of work [3, 9, 38, 43, 86].

Technical highlight. Inspired by the original work of Chan et al. [28], we adopt the following design paradigm for devising DO algorithms. At a very high level, we decompose the task of designing a DO algorithm into the following: 1) identify a set of intermediate ideal functionalities with *differentially private leakage*; and 2) leverage oblivious algorithms building blocks to obliviously realize these ideal functionalities, such that the access patterns leak only the stated differentially private leakage, and nothing else.

Although the design paradigm is simple to state, the non-trivial challenge is to identify appropriate intermediate functionalities that not only lend to solving our problem, but also being cognizant that the computational tasks they embody must have efficient oblivious realizations. We defer the algorithmic details to subsequent formal sections, and we hope that our algorithmic techniques can inspire the design of DO algorithms for new applications. We believe that our work provides further evidence on top of the early-stage explorations of Chan et al. [28] and Beimel et al. [17] that differential obliviousness is a useful notion that deserves attention.

2 Technical Roadmap

For convenience, henceforth we call the two input tables *arrays*, denoted $\mathbf{I}_1$ and $\mathbf{I}_2$ respectively. Each element in $\mathbf{I}_1$ and $\mathbf{I}_2$ is either a real element of the form (k, v) or a filler element of the form $(\perp, \perp)$. For a real element, k is called the join key (or key for short) and v is called the payload. The database join operation wants to compute, for each unique join key k, the Cartesian product of the elements contained in both arrays with join key k. All results are concatenated and output, and moreover, the output is allowed to contain an arbitrary number of filler elements that may be needed for privacy.

▶ Remark 4 (Simplifying assumption for the roadmap). Throughout our informal technical roadmap, we will assume the typical parameters $\epsilon = \Theta(1)$ and $\delta = 1/N^c$ for an arbitrary constant $c \geq 1$. In this case, $\frac{1}{\epsilon}\log\frac{1}{\delta} = \Theta(\log N)$, and we thus use two expressions interchangeably – but our formal sections later will differentiate the two to be more general. Specifically, jumping ahead, we often need to add noises of magnitude roughly $\frac{1}{\epsilon}\log\frac{1}{\delta} = \Theta(\log N)$.

Our differential oblivious join algorithm will make use of standard obivious algorithm building blocks including oblivious sorting [4, 10, 76], and oblivious compaction [11, 67]. We review these building blocks in more detail in Section 3.6.

2.1 Warmup Algorithm

We first present a warmup that achieves $O(R + N\log^2 N)$ runtime – the warmup algorithm does not achieve the bounds stated earlier; but it is conceptually simpler and helps our understanding. Later in Section 2.3, we describe additional techniques to improve the algorithm's asymptotical performance, and achieve the bounds stated earlier.

A strawman idea. To understand our algorithm, let us first consider a flawed strawman – we sketch the high-level idea, and for the time being, omit the details on how to oblivious sorts to implement the relevant steps.

1. *Compute the bin load array L.* First, using a constant number of oblivious sorts, write down a list L of length $N := |\mathbf{I}_1| + |\mathbf{I}_2|$. Each element in L is of the form $(k, \widehat{n}_k^{(1)}, \widehat{n}_k^{(2)})$ where $\widehat{n}_k^{(b)}$ denotes the noisy count of the join key k in table $b \in \{1, 2\}$. The noisy count is obtained by adding an appropriate, independently sampled noise to the actual multiplicity of join key k in the corresponding array. To maintain correctness, the noise must be non-negative. So rather than adding a Laplacian noise with standard deviation $1/\epsilon$, we shift the Laplacian to the right to be centered at $U/2 = \Theta(\frac{1}{\epsilon} \cdot \log \frac{1}{\delta}) = \Theta(\log N)$. In this way, the noise lies within the range $[0, U]$ except with δ probability[9].
The list L should contain all join keys that appear in at least one input array, padded with fillers of the form $(\star_1, \widehat{n}_{\star_1}^{(1)}, \widehat{n}_{\star_1}^{(2)})$, $(\star_2, \widehat{n}_{\star_2}^{(1)}, \widehat{n}_{\star_2}^{(2)})$, …, to a length of N. The filler join keys $\star_1, \star_2, \ldots$ have an actual multiplicity of 0 in both input arrays, and thus their noisy counts are the shifted Laplacian noise in the range $[0, U]$. The list L is *sorted by the join key k, and all filler join keys appear at the end.*
2. *Binning.* Now, we have $2N$ bins each indexed by a pair (b, i) where $b \in \{1, 2\}$ and $i \in [N]$. Let k_i denote the i-th smallest join key. Then, the bin indexed (b, i) has capacity $\widehat{n}_{k_i}^{(b)}$, and all elements in $\mathbf{I}_b$ with the join key k_i are destined for this bin. Using a constant number of oblivious sorts, route all elements in either array to their respective destined bins, and pad each bin with fillers to its intended capacity. Note that the bins corresponding to the filler join keys $\star_1, \star_2, \ldots$ have no real elements in them and are full of fillers.
3. *Bin-wise Cartesian product.* Now, take every pair of bins $(1, i)$ and $(2, i)$ for $i \in [N]$, and compute the Cartesian product of elements in the two bins – if the two elements being joined have the same real join key, add the joined tuple to the output; otherwise, add a filler element to the output.

A flaw that violates differential obliviousness. The above algorithm is natural and conceptually simple; it almost works, except for a critical flaw that violates differential obliviousness, which is illustrated in Figure 1 and explained in detail below. Observe that the array L is sorted by the join key during Step 1. Now, consider an input $\mathcal{I} := (\mathbf{I}_1, \mathbf{I}_2)$ in which 8th smallest join key k_8 appears only 1 time in $\mathbf{I}_1$ and does not appear in $\mathbf{I}_2$, and all other join

[9] In our formal sections later, we actually use a shifted geometric distribution which is the discrete counterpart of the real-valued Laplacian, and moreover we simply truncate the δ-probability mass outside the range $[0, U]$ which allows us to get deterministic bounds on the algorithm's runtime.

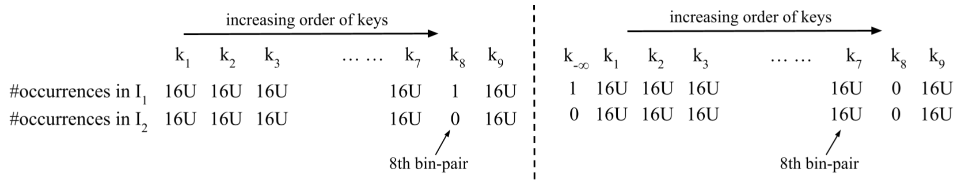

Figure 1 Strawman scheme: a flaw that violates differential obliviousness. See the main body for more explanation.

keys that appear in $\mathcal{I}$ appear more than $16U$ times in both arrays. Consider a 2-neighboring input where the only occurrence of k_8 is replaced with a join key $k_{-\infty}$ that 1) does not exist in $\mathcal{I}$, and 2) is smaller than all other join keys in $\mathcal{I}$. In this case, an adversary observing the access patterns of the program can easily tell which input is used. Recall that the bin pairs are sorted in the order of the join keys: in the case of $\mathcal{I}$, the 8-th bin pair has small capacities and the first bin pair has large capacities (where small means between at most $U + 1$ and large means at least $16U$). In the case of $\mathcal{I}'$, however, the first bin pair, now corresponding to the join key $k_{-\infty}$, has small capacities.

A remedy. It turns out that a simple fix can address the above flaw: instead of ordering the array L using the join key k, we can order it lexicographically based on the $(\widehat{n}_k^{(1)}, \widehat{n}_k^{(2)})$ fields. Intuitively, this can avoid accidental information leakage through the ordering. More specifically, in the above steps 2 and 3, the capacities of all bins are leaked to the adversary. Therefore, if we use the fields $(\widehat{n}_k^{(1)}, \widehat{n}_k^{(2)})$ to order the array L and the corresponding bins, then the only information leaked is the *multiset* of $(\widehat{n}_k^{(1)}, \widehat{n}_k^{(2)})$ pairs. Given the *multiset* of the $(\widehat{n}_k^{(1)}, \widehat{n}_k^{(2)})$ pairs, the access patterns of the above algorithm are fully determined.

Therefore, it suffices to prove that the leakage, i.e., the *multiset* of the $(\widehat{n}_k^{(1)}, \widehat{n}_k^{(2)})$ pairs, satisfies $(O(\epsilon), O(\delta))$-differential privacy. In our formal technical sections later, we shall prove that this is indeed the case as long as the noises are chosen from the shifted and truncated geometric distribution defined in Section 3.4.

Final step: compaction of the join result. The above modification fixes the security flaw, but at this moment, the result output by our algorithm may have length $O(R + N \log^2 N)$ – see Section 2.2 for a more detailed analysis. Specifically, when the true result length R is small, the additive term $N \log^2 N$ is dominant.

We would like our algorithm to output a result that is as short as possible specific to the instance. Clearly, for the result to be correct, it cannot be shorter than R. In the online full version [34], we prove that all (ϵ, δ)-DO algorithms must have result length at least $R + \Omega(\mu_{\max} \cdot \frac{1}{\epsilon} \log \frac{1}{\delta})$ where $\mu_{\max}$ is the maximum multiplicity of any join key in either array.

Our idea is to obliviously compact the result output by the above algorithm to length $R + \mathsf{noise}$ where noise is sampled from an appropriate distribution. The most natural idea is to sample a shifted Laplacian noise proportional to $\Delta_{\text{global}}/\epsilon$ where Δ_{global} is the global sensitivity of the exact result length (i.e., how much the length of the exact result would change in the *worst case* when we change one position in the input). However, Δ_{global} can be as large as $\Theta(N)$. Instead, we would like to achieve an *instance-optimal* bound on the result length (for almost all parameter regimes). To do so, we add noise proportional to the local sensitivity (or instance-specific sensitivity) which is equal to $\mu_{\max}$. To make the idea work, however, we have to first obtain a noisy version $\widehat{\mu}_{\max}$ to the local sensitivity $\mu_{\max}$, and then add noise proportional to $\widehat{\mu}_{\max}$ to the result length. In this way, our algorithm achieves result length $R + (\mu_{\max} + \log N)\log N$ which is instance-optimal in light of the $R + \Omega(\mu_{\max} \log N)$ lower bound in almost all parameter regimes. We defer a detailed description of the scheme to the formal technical sections.

2.2 Performance of the Warmup Algorithm

As mentioned, the warmup algorithm, obtained by changing the way the array L is ordered in the strawman solution, achieves runtime $O(R + N \log^2 N)$. To understand the techniques in Section 2.3 that improves the performance to $O(R + N \log N)$, let us first understand the performance breakdown.

1. The first step, which computes the bin load array L, performs a constant number of oblivious sorts on arrays of length at most $N := |\mathbf{I}_1| + |\mathbf{I}_2|$, and thus takes $N \log N$ time.
2. The second step, which places the elements into bins, performs a constant number of oblivious sorts, and the length of the arrays sorted is upper bounded by the sum of the bin capacities. In the worst case, there can be $\Theta(N)$ sparsely loaded bins each with only $O(1)$ number of real elements. The noise added to each bin's capacity is roughly of magnitude $\frac{1}{\epsilon} \log \frac{1}{\delta}$ which is $O(\log N)$ under typical parameters (see Remark 4). Therefore, the length of the array sorted is at most $O(N \log N)$, and the second step takes time $O((N \log N) \log(N \log N)) = N \log^2 N$.
3. The third step computes the Cartesian product of pairs of bins: the runtime of this step is the actual result length R when there is no noise, plus the number of fillers. The number of fillers is maximized when there are $\Theta(N)$ bins each with $O(1)$ number of real elements and $O(\log N)$ fillers. In this case, the total number of fillers after the Cartesian product is $O(N \log^2 N)$. Therefore, the third step takes time $O(R + N \log^2 N)$.

Summarizing the above, our warmup algorithm achieves $O(R + N \log^2 N)$ runtime.

2.3 Final Algorithm

Section 2.2 reveals that the $N \log^2 N$ additive term in the performance bound is incurred because in the worst-case scenario, there can be $\Theta(N)$ sparsely loaded bin-pairs each with $O(1)$ real elements, and padded with $\Theta(\log N)$ fillers[10]. This introduces the $N \log^2 N$ additive term in two ways: 1) the binning step requires sorting arrays of length $O(N \log N)$ which takes $O(N \log^2 N)$ time; and 2) the bin-wise Cartesian product introduces $O(N \log^2 N)$ fillers.

Imprecisely speaking, having many sparsely loaded bin-pairs cause a small "signal to noise" ratio, i.e., the ratio of fillers is high. To improve the performance bound to $O(R + N \log N)$, our idea is to reduce the number of bin-pairs to $O(N/\log N)$, thereby improving the "signal to noise" ratio. We say that a join key k is *sparse* iff its noise counts $\widehat{n}_k^{(1)}$ and $\widehat{n}_k^{(2)}$ are both upper bounded by $2U$, where recall that $U = \Theta(\frac{1}{\epsilon} \log \frac{1}{\delta})$. A join key that is not sparse is said to be *dense*. Recall that $N := |\mathbf{I}_1| + |\mathbf{I}_2|$.

As mentioned, our noise distribution is upper bounded by U except with probability δ. This means that if a bin-pair contains a dense join key, then at least one of the bins in the pair has at least U real elements in it except with δ probability. Thus there can be

[10] For example, this can happen if there are many elements with $O(1)$ occurrences in both input arrays, or with $O(1)$ occurrences in one array but not appearing in the other. In the latter case, essentially there are many elements in the symmetric difference of the two input arrays that do not contribute to the true joined result.

no more than $O(N/\log N)$ bin-pairs for dense-keys. Our focus therefore is to consolidate multiple sparse join keys into the same bin-pair such that each bin-pair contains at least $\Theta(U)$ elements (including elements from both input arrays). To achieve this, we perform the following.

Compute key-to-bin mapping. Recall that earlier we computed the bin load array L which contains tuples of the form $(k, \widehat{n}_k^{(1)}, \widehat{n}_k^{(2)})$ sorted according to lexicographical ordering on $(\widehat{n}_k^{(1)}, \widehat{n}_k^{(2)})$. It is not too hard to extend the algorithm for computing L such that the bin load array L also stores the actual counts, i.e., L now contains entries of the form $(k, n_k^{(1)}, \widehat{n}_k^{(1)}, n_k^{(2)}, \widehat{n}_k^{(2)})$. where $n_k^{(1)}$ and $n_k^{(2)}$ denote the actual multiplicity of the join key k in $\mathbf{I}_1$ and $\mathbf{I}_2$, respectively.

Now, we can classify L into a part L_s corresponding to sparse keys, i.e, $L_s := \{(k, n_k^{(1)}, \widehat{n}_k^{(1)}, n_k^{(2)}, \widehat{n}_k^{(2)}) \in L : \widehat{n}_k^{(1)} \leq 2U, \widehat{n}_k^{(2)} \leq 2U\}$; and a part $L_d := L \backslash L_s$ corresponding to dense join keys. All of L, L_s, and L_d are sorted according to lexicographical ordering on $(\widehat{n}_k^{(1)}, \widehat{n}_k^{(2)})$; and L_s and L_d can be constructed in $O(N)$ time if we allow the access patterns to reveal the noisy counts contained in L.

Our goal now is to construct an array called BinMap that maps join keys to bin pairs (note by constructing BinMap, we have not moved the elements into their bins yet – the actual moving will be done in the subsequent binning step). Each entry of BinMap is of the form (k, i), meaning that the join key k should be mapped to the i-th bin-pair. To construct BinMap, we first scan through L_d: for each $i \in \{1, 2, \ldots, |L_d|\}$, if the i-th entry in L_d has the join key k, add the tuple (k, i) to the array BinMap. At this moment, BinMap stores the mapping from each dense join key to its destined bin-pair index. The capacities of these bin-pairs (for dense join keys) are determined by the noisy counts in L_d.

Next, we will add to BinMap the mapping from sparse join keys to their bins. Specifically, sparse join keys are mapped to additional bin-pairs numbered $\{(1, j), (2, j) : j = |L_d| + 1, |L_d| + 2, \ldots, |L_d| + O(N/U)\}$ where j is also called the bin-pair index. All of these bins (for sparse join keys) will have capacity *exactly* $4U$, and here we allow multiple join keys to be mapped to the same bin pair. The invariant we want is that for each bin-pair $(1, j), (2, j)$ where $j \in \{|L_d| + 1, |L_d| + 2, \ldots, |L_d| + O(N/U)\}$, a total of at least $2U$ elements will be mapped to the bin pair (summing across both input arrays). In this way, all the sparse join keys altogether will not consume more than $N/2U$ bins.

To achieve this, we can scan linearly through L_s. For each entry $(k, n_k^{(1)}, \widehat{n}_k^{(1)}, n_k^{(2)}, \widehat{n}_k^{(2)}) \in L_s$ encountered during the scan, we append to BinMap a tuple (k, j) that indicates that join key k is mapped to the j-th bin-pair. Here j is the current bin-pair counter whose starting value is $|L_d| + 1$, and j is incremented whenever one of the current bins is about to exceed its capacity. To achieve this, the algorithm additionally maintains two counters that remember how many cumulative elements have been mapped to the current bins $(1, j)$ and $(2, j)$ so far. Whenever one of the bins $(1, j)$ or $(2, j)$ is about to exceed its capacity $4U$, we increment the bin-pair counter j and reset both counters to be 0 again, i.e., we start mapping join keys to the next bin-pair. We stress that the access pattern of this step is fixed and depends only on $|L_s|$ because we append a single entry to BinMap whenever we visit an entry of L_s.

Binning. Next, we perform a binning step and move elements into their desired bins – henceforth a bin designated for a single dense join key is called a D-bin, and a bin designated for possibly multiple sparse join keys is called an S-bin. To perform the binning, we need BinMap which provides the mapping between join keys and their bin indices. D-bins have capacities determined by the corresponding noisy counts contained in L_d and there are $|L_d|$

D-bins. S-bins have capacities exactly $4U$, and the number of S-bins is an a-priori fixed upper bound CN/U for a sufficiently large constant C. We can now use a constant number of oblivious sorts to move elements into their desired bins, padding each bin with fillers to its intended capacity as defined above. Note that it is possible that some S-bins do not receive any element.

Remainder of the algorithm. The remainder would be similar to the warmup algorithm. We perform bin-wise Cartesian product; and finally we obliviously compact the result adding an appropriate noise to the final result length. Since now, multiple join keys can share the same bin-pair, during the Cartesian product step, whenever we try to pairwise-join two elements with different join keys, we append a filler to the output array.

2.4 Lower Bound Results

As mentioned, we prove new lower bounds on the result length and runtime of any DO join algorithm. The result length lower bound is proven using the definition of differential obliviousness. Our lower bound on runtime is obtained by taking the maximum of two lower bounds: 1) the aforementioned lower bound on the result length, and 2) a lower bound that stems from a privacy-preserving and complexity-preserving reduction from sorting to database join. Such a reduction shows that any DO join algorithm must suffer from the same lower bound for DO sorting proven recently by Chan et al. [28]. We refer the reader to the online full version [34] for the detailed statements and proofs.

2.5 Additional Related Work

In this paper, we adopt the differential obliviousness notion defined by Chan et al. [28]. Besides Chan et al. [28], several other works also considered related but somewhat incomparable notions [60, 68, 87].

Besides the aforementioned works on *oblivious* databases [7, 35, 41, 90], a related but incomparable line of work [2, 14, 25, 26, 36, 57, 73, 74, 80, 82, 83] focuses on encrypted databases or searchable encryption systems. Typically, this line of works either give up on hiding access patterns [14, 25, 26, 36, 74, 80, 82, 83], or hide the access pattern by performing a linear scan through the entire database upon every update or query [2]. The cryptographic techniques for computation on encrypted data developed in this line of work is somewhat orthogonal and complementary to our techniques for obfuscating the access patterns.

Following the classical differential privacy (DP) literature, another line of work that focuses on differentially private database queries [30, 53, 54, 65, 66]. These works assume that the database curator is trusted and only aims to guarantee that the result is DP – they are not concerned about information leakage through the runtime behavior of the database engine. Notably, the techniques we use to release the noisy counts for the multiplicity of join keys may be remotely reminiscent of differentially private histogram mechanisms [2, 15, 19, 22, 64, 84]. We stress, however, that our work and techniques are of a different nature from classical DP mechanisms, including classical DP algorithms for releasing histograms. While prior DP mechanisms introduce noise to the statistics released, in our context, we introduce noise to the algorithm's access patterns (and not its output) – importantly, we need to do so without affecting the algorithm's correctness. It would be very interesting, however, to apply our DO techniques to classical DP algorithms – in this way, both the runtime behavior of the database as well as the released statistics would guarantee DP, i.e., we get "end-to-end" privacy.

Mazloom and Gordon [68] proposed new techniques that guarantee differential obliviousness for tasks that can be performed in a graph-parallel framework. They rely on shuffling to guarantee that the access patterns of these algorithms reveal only differentially private histograms. While seemingly related to our techniques, we do not know any straightforward way to apply their techniques to the join problem and get our asymptotical bounds: partly, our techniques are non-trivial because *we avoid suffering too much overhead for join keys that are in the symmetric difference of the two input arrays* – these join keys do not contribute to the true joined result. Imprecisely speaking, doing such "pruning" *privately* introduces non-trivial algorithmic challenges.

Komargodski and Shi [62] suggest how to compile any Turing Machine (TM) to a differentially oblivious TM. A strawman idea is to apply their compiler to the (insecure) TM that computes the database join problem. Unfortunately, this completely fails because their work defines neighboring on the *operational sequences* of two TMs; whereas we define neighboring on the *inputs*. For two inputs that are neighboring (i.e., Hamming distance 1), applying the insecure TM that computes database join over these inputs may result in operational sequences that are far apart. This is also partly why our problem is challenging.

2.6 Open Problems

Our work is among the first to explore DO algorithms motivated by practical database systems. Our work reveals that this is a promising direction with many intriguing open questions. For example, can we bridge the gap between our upper- and lower-bounds for a broader range of parameters?

Another exciting direction is to explore DO algorithms for other common database queries. In this paper, we considered two-way joins. In the classical, non-private database literature, however, *multi-way join* [1, 13, 16, 33, 44, 55, 61, 71, 78, 89] received significantly more attention because instance-optimal two-way join is long known to be a solved problem. Our paper reveals that with the extra privacy requirements, even two-way join raises non-trivial algorithmic challenges. Of course, it also makes sense to ask whether one can design efficient DO algorithms for multi-way joins as well, especially, whether we can (approximately) match the performance of the best known insecure algorithms. Recent works in the database literature also considered join algorithms for the special case when the input tables are already sorted based on the join key (or more generally, preprocessed in some way). In this case, it may not be necessary to read the entire input, and sublinear (non-private) algorithms are known [61, 71]. Therefore, an open question is whether we can achieve sublinear DO algorithms for sorted inputs. Besides joins, more general class of database queries such as conjunctive queries [1] are also interesting to consider.

Last but not the least, as mentioned, the cache-efficient variants of our algorithms are potentially implementable and suitable for SGX-type scenarios. Implementing DO algorithms in practical database systems and evaluating their concrete performance is another exciting future direction.

3 Preliminaries

We assume that the algorithm is executed in a standard Random Access Machine (RAM) model. The adversary can observe the access patterns of the program, i.e., in each step, which memory location is accessed and whether each access is a read or write operation. The adversary, however, cannot observe the data contents – for example, in a secure processor setting, the data contents protected by encryption. We will be concerned about two metrics:

1) the program's runtime; and 2) the program's cache complexity [3]. For the runtime statements, we assume a standard word-RAM and the CPU has only $O(1)$ number of private registers. For the cache complexity metric, we assume a standard external-memory RAM model [3] where the CPU has M bits of private cache, and every time it needs to transmit data to and from memory, an atomic unit (called a "block") of B bits is transferred. The cache complexity metric measures how many blocks are transmitted between the CPU and memory. All of our algorithms are cache agnostic [38, 43], i.e., the algorithm need not know the cache's parameters M and B.

3.1 Database Join

Database join is the following problem. Let $\mathcal{K}$ denote the space of join key and $\mathcal{V}$ denote the space of payloads. Let $\mathbf{I}_1$ and $\mathbf{I}_2$ be two input arrays each containing pairs of the form (k, v), where $k \in \mathcal{K} \cup \{\perp\}$ is called a *join key* and $v \in \mathcal{V} \cup \{\perp\}$ is called the *payload*. If $k \in \mathcal{K}$ and $v \in \mathcal{V}$, the element (k, v) is said to be a *real* element. Without loss of generality, we may assume that if an element's join key k is $\perp$, it payload v must be $\perp$ too: such elements, of the form $(\perp, \perp)$, are said to be *filler* elements.

Our goal is to output an array $\mathbf{O}$ such that for each non-filler join key $k \in \mathcal{K}$ that appears in both $\mathbf{I}_1$ and $\mathbf{I}_2$: let $\{(k, v_1), (k, v_2), \ldots, (k, v_m)\}$ be the multi-set of elements having join key k in $\mathbf{I}_1$, and let $\{(k, w_1), (k, w_2), \ldots, (k, w_{m'})\}$ be the multi-set of elements having join key k in $\mathbf{I}_2$; then the multi-set $\{k, v_i, w_j\}_{i\in[m], j\in[m']} \subseteq \mathbf{O}$. We use R to denote the size of this multi-set. Moreover, besides the multi-set $\{k, v_i, w_j\}_{i\in[m], j\in[m']}$, $\mathbf{O}$ should contain no other element with the join key k. Additionally, the output array $\mathbf{O}$ may contain any number of filler elements of the form $(\perp, \perp, \perp)$.

In other words, the output array $\mathbf{O}$ contains, for each join key $k \in \mathcal{K}$, the Cartesian product of the elements in both input arrays under join key k; and additionally, $\mathbf{O}$ may contain some filler elements. The filler elements in the output array $\mathbf{O}$ may be needed for privacy reasons as will become clear later.

▶ Remark 5 (Motivation for allowing fillers in the input array). In our formulation, we allow the input arrays $\mathbf{I}_1$ and $\mathbf{I}_2$ to contain filler elements, because the length of the input arrays may already be noisy to mask the true number of elements contained in it. For example, if the input array comes from a differentially oblivious database such as in the work by Chan et al. [28], then the input arrays would already contain a random number of filler elements.

3.2 Full Obliviousness

In this paper, we consider execution of algorithms on the Random Access Machine (RAM) model. Let Alg denote a possibly randomized algorithm and let $\mathbf{I}$ denote an input to the algorithm. We use the notation $\mathsf{Accesses}^{\mathsf{Alg}}(\mathbf{I})$, a random variable denoting the sequence of memory addresses accessed and whether each access is a read or write, generated by a random execution of the algorithm Alg on input $\mathbf{I}$. Therefore, $\mathsf{Accesses}^{\mathsf{Alg}}(\mathbf{I})$ is also called the "access patterns" of Alg on input $\mathbf{I}$.

▶ **Definition 6** (δ-obliviousness). *We say that an algorithm* Alg *satisfies* δ*-obliviousness w.r.t. the leakage function* $\mathsf{Leak}(\cdot)$*, iff there exists a simulator* Sim*, such that* $\mathsf{Accesses}^{\mathsf{Alg}}(\mathbf{I})$ *has statistical distance at most* δ *from the simulated access patterns* $\mathsf{Sim}(\mathsf{Leak}(\mathbf{I}))$*.*

In other words, the access patterns are simulatable by a simulator Sim which knows only the leakage function but nothing more about the input $\mathbf{I}$. Note also that δ is allowed to be a function in $N = |\mathbf{I}|$.

A typical leakage function is leaking only the length of the input and nothing else, i.e., $\mathsf{Leak}(\mathbf{I}) := |\mathbf{I}|$.

▶ **Definition 7** (Full obliviousness). *Henceforth, whenever we say* Alg *is (fully) oblivious (i.e., omitting the leakage function and δ), the leakage function would the default one* $\mathsf{Leak}(\mathbf{I}) := |\mathbf{I}|$, *and δ is assumed to be a negligible function in N.*

Throughout the paper, we say that a function $\nu(N)$ is a negligible function, iff for any $c \in \mathbb{N}$, there exists a sufficiently large N_0 such that for all $N \geq N_0$, $\nu(N) \leq 1/N^c$. In other words, ν drops faster than any inverse-polynomial function.

3.3 Differential Obliviousness

Neighboring inputs. Two inputs $(\mathbf{I}_1, \mathbf{I}_2)$ and $(\mathbf{J}_1, \mathbf{J}_2)$ are said to be neighboring, iff $|\mathbf{I}_1| = |\mathbf{J}_1|$ and $|\mathbf{I}_2| = |\mathbf{J}_2|$, and moreover, the following holds:

- either $\mathbf{I}_1 = \mathbf{J}_1$, and moreover, $\mathbf{I}_2$ and $\mathbf{J}_2$ differ in exactly one position;
- or $\mathbf{I}_2 = \mathbf{J}_2$, and moreover, $\mathbf{I}_1$ and $\mathbf{J}_1$ differ in exactly one position.

Differential obliviousness. Imagine that a database join algorithm Alg is executed on a Random Access Machine (RAM). The two input arrays $(\mathbf{I}_1, \mathbf{I}_2)$ reside in memory, and at the end of the algorithm, the output array $\mathbf{O}$ is written to a designated position in memory.

▶ **Definition 8** ((ϵ, δ)-differential obliviousness). *We say that a database join algorithm* Alg *satisfies (ϵ, δ)-differential obliviousness or (ϵ, δ)-DO for short, iff for any neighboring inputs $(\mathbf{I}_1, \mathbf{I}_2)$ and $(\mathbf{J}_1, \mathbf{J}_2)$, for any set S,*

$$\Pr\left[\mathsf{Accesses}^{\mathsf{Alg}}(\mathbf{I}_1, \mathbf{I}_2) \in S\right] \leq e^{\epsilon} \cdot \Pr\left[\mathsf{Accesses}^{\mathsf{Alg}}(\mathbf{J}_1, \mathbf{J}_2) \in S\right] + \delta$$

where $\mathsf{Accesses}^{\mathsf{Alg}}(\mathbf{I}_1, \mathbf{I}_2)$ is a random variable denoting the sequence of memory addresses (also called access patterns) generated by a random execution of the algorithm Alg *on input $(\mathbf{I}_1, \mathbf{I}_2)$.*

Specifically, in the standard RAM model, in each time step, the machine visits one memory location, reading it and then updating it with either the old value or a new value. Therefore, $\mathsf{Accesses}^{\mathsf{Alg}}(\mathbf{I}_1, \mathbf{I}_2)$ is just the ordered list of all memory addresses accessed in all time steps. Moreover, the length of $\mathsf{Accesses}^{\mathsf{Alg}}(\mathbf{I}_1, \mathbf{I}_2)$ is also the (randomized) runtime of the algorithm. Like the standard notion of differential privacy, our notion secures against unbounded adversaries.

Typical choices of ϵ and δ. Typically, we would like $\epsilon = \Theta(1)$. The standard differential privacy literature [85] recommends that δ be set to $1/N^c$ for some constant $c > 1$. In the cryptography literature, sometimes we would like δ to be negligibly small in N.

3.4 Mathematical Building Blocks

▶ **Definition 9** (Symmetric geometric distribution). *Let $\alpha > 1$. The symmetric geometric distribution* $\mathsf{Geom}(\alpha)$ *takes integer values such that the probability mass function at k is $\frac{\alpha-1}{\alpha+1} \cdot \alpha^{-|k|}$.*

As we shall see, our algorithm will hide the true cardinality of a set by padding it with a random number of filler elements. Below we define a useful distribution from which we shall sample the noises.

▶ **Definition 10** (Shifted and truncated geometric distribution). *Let $\epsilon > 0$ and $\delta \in (0,1)$ and $\Delta \geq 1$. Let k_0 be the smallest positive integer such that $\Pr[|\mathsf{Geom}(e^{\frac{\epsilon}{\Delta}})| \geq k_0] \leq \delta$, where $k_0 = \frac{\Delta}{\epsilon} \ln \frac{2}{\delta} + O(1)$. The shifted and truncated geometric distribution $\mathcal{G}(\epsilon, \delta, \Delta)$ has support in $[0, 2(k_0 + \Delta - 1)]$, and is defined as:*

$$\min\{\max\{0, k_0 + \Delta - 1 + \mathsf{Geom}(e^{\epsilon})\}, 2(k_0 + \Delta - 1)\}$$

For the special case $\Delta = 1$, we write $\mathcal{G}(\epsilon, \delta) := \mathcal{G}(\epsilon, \delta, 1)$.

In the main body of the paper, for simplicity, we shall first assume that we can sample from the shifted and truncated geometric distribution in $O(1)$ time. In the online full version [34], we discuss how to remove this assumption without blowing up the runtime.

Notation. Given two random variables X and Y, we use $X \sim_{(\epsilon,\delta)} Y$ to denote that X and Y satisfy the standard (ϵ, δ)-differentially private inequality, i.e., for all subsets S, $\Pr[X \in S] \leq e^{\epsilon} \cdot \Pr[Y \in S] + \delta$, and $\Pr[Y \in S] \leq e^{\epsilon} \cdot \Pr[X \in S] + \delta$,

▶ **Fact 11** (Differential privacy through adding truncated and shifted geometric noise [15, 28]). *Let $\epsilon > 0$ and $\delta \in (0,1)$. Suppose u and v are two non-negative integers such that $|u - v| \leq \Delta$. Then,*

$$u + \mathcal{G}(\epsilon, \delta, \Delta) \sim_{(\epsilon,\delta)} v + \mathcal{G}(\epsilon, \delta, \Delta).$$

▶ **Fact 12** (Post-processing). *Let $X \in \mathcal{X}$ and $X' \in \mathcal{X}$ be random variables and let $F : \mathcal{X} \to \mathcal{Y}$ be a possibly randomized function. Suppose that $X \sim_{(\epsilon,\delta)} X'$. Then, we have that*

$$F(X) \sim_{(\epsilon,\delta)} F(X').$$

▶ **Fact 13** (Composition of differentially private mechanisms (Theorem B.1 of [40])). *Suppose that for any neighboring $\mathbf{I}$ and $\mathbf{I}'$, $T_1(\mathbf{I}) \sim_{(\epsilon_1,\delta_1)} T_1(\mathbf{I}')$. Henceforth let $\mathsf{supp}(T_1(\mathbf{I}))$ denote the support of applying the function T_1 to the data $\mathbf{I}$. Suppose that for any neighboring $\mathbf{I}$ and $\mathbf{I}'$, for any $s_1 \in \mathsf{supp}(T_1(\mathbf{I})) \cup \mathsf{supp}(T_1(\mathbf{I}'))$, $T_2(\mathbf{I}, s_1) \sim_{(\epsilon_2,\delta_2)} T_2(\mathbf{I}', s_1)$. Then, for any neighboring $\mathbf{I}$, $\mathbf{I}'$, we have that*

$$(T_2, T_1)(\mathbf{I}) \sim_{(\epsilon_1+\epsilon_2,\delta_1+\delta_2)} (T_2, T_1)(\mathbf{I}')$$

The following operational lemma will guide our algorithm design and analysis.

▶ **Lemma 14** (Operational lemma for differential obliviousness [28]). *If an algorithm is δ_0-oblivious w.r.t. a leakage function $\mathsf{Leak}(\cdot)$, and moreover, Leak is (ϵ, δ)-differentially private, then the algorithm satisfies $(\epsilon, \delta_0 + \delta)$-differential obliviousness.*

3.5 Naïve Solutions

Insecure algorithm. If privacy is not needed, there is an algorithm that computes the join result in (expected) $O(N + R)$ time, where R is the actual result size and $N := |\mathbf{I}_1| + |\mathbf{I}_2|$. Basically, hash elements in each input array into a separate Cuckoo hash table [8, 72] and within each entry of the hash table, use a linked list to store all elements with the same join key. Now, we can pairwise-join elements with the same join key from the two input arrays by querying the Cuckoo hash table. The entire algorithm completes in $O(N + R)$ expected time (where the randomness comes from hashing).

The Cuckoo hashing based insecure solution, however, does not have great cache complexity. For cache complexity, we use a different insecure baseline, that is, first sort the two input arrays by join key, and then use the most straightforward approach to compute pairwise-join elements with the same join key from the two arrays. This algorithm achieves $O(\frac{R}{B} + \frac{N}{B} \cdot \log_{\frac{M}{B}} \frac{N}{B})$.

Observe also that any insecure algorithm must at least read the entire input to guarantee correctness. Thus an always correct insecure algorithm must incur at least $\Omega(N)$ runtime.

Fully oblivious algorithm. The following naïve algorithm, which tries every potential pair of elements from the two input arrays, can achieve fully oblivious database join with $O(|\mathbf{I}_1| \cdot |\mathbf{I}_2|)$ runtime:

For each element $(k, v) \in \mathbf{I}_1$, for each element $(k', v') \in \mathbf{I}_2$: if $k = k'$, add (k, v, v') to the output; else add $(\perp, \perp, \perp)$ to the output.

As argued in the online full version [34], $O(|\mathbf{I}_1| \cdot |\mathbf{I}_2|)$ is also the best we can hope for if full obliviousness is desired.

Naïve differentially oblivious algorithm. A naïve approach to achieve differentially oblivious join is to rely on a *statistically* secure Oblivious RAM (ORAM) that defends against unbounded adversaries, such as Circuit ORAM [29, 88], to simulate the aforementioned insecure algorithm with $O(\log^2 N)$ blowup in runtime. Suppose that the result size is R and let N be the total input length. We know that the insecure algorithm must complete in $T \le C \cdot (R + N)$ steps for some constant C, and it produces an output of length R. Now, given the $O(\log^2 N)$ ORAM simulation overhead, simulating the insecure algorithm with ORAM requires at most $C'(R + N) \log^2 N$ steps for some sufficiently large $C' > C$. However, if we just stopped here, then the algorithm would leak information through the result length R and its running time (which is the same as the total length of the physical memory accesses in the RAM model).

To plug this leakage, the algorithm proceeds to add noise to the result length and its own runtime. To achieve this, we continue to use the ORAM to simulate the following steps:

1. Append $\xi := \mathcal{G}(\epsilon, \delta, N) = O(\frac{N}{\epsilon} \log \frac{1}{\delta})$ number of filler elements to the joined result – this requires the ORAM to simulate $O(\xi)$ additional steps. Henceforth let $\widehat{R} := R + \xi$.
2. Finally, pad the running time of the simulated algorithm to $C'(\widehat{R} + N) \log^2 N$.

Note that in the above, we add noise $\mathcal{G}(\epsilon, \delta, N)$ noise to R because the global sensitivity of R is upper bounded by N, that is, changing one element in the input can change R by at most N.

To obtain better cache complexity, we can place the ORAM schemes' binary tree data structures in an Emde Boas layout; specifically, each access in the original program will incur a cache complexity of $O(\log N \cdot \log_B N)$ in the ORAM simulation.

▶ **Theorem 15** (Naïve DO algorithm). *The above naïve algorithm satisfies* $(\epsilon, \delta + \mathsf{negl}(N))$*-differential obliviousness where* $\mathsf{negl}(\cdot)$ *denotes a suitable negligible function.*

Further, suppose that $\epsilon = \Theta(1)$ *and* $\delta = \frac{1}{\mathsf{poly}(N)}$*, let* $U = O(\frac{1}{\epsilon} \log \frac{1}{\delta}) = O(\log N)$*; then, the above naïve algorithm achieves* $O((R+NU) \log^2 N)$ *runtime and* $O((R + NU) \log N \cdot \log_B N)$ *cache complexity, and outputs a result of* $O(R + NU)$ *length.*

Proof. The runtime, cache complexity and output length follows from the above description. Observe that because of ORAM (which can fail with $\mathsf{negl}(N)$ probability), the access pattern are simulatable given N and $\widehat{R}$. By Lemma 14, it suffices to prove that the leakage $\widehat{R}$ satisfies (ϵ, δ)-differential privacy.

Consider changing one element in the input $(\mathbf{I}_1, \mathbf{I}_2)$ to form $(\mathbf{I}'_1, \mathbf{I}'_2)$. We have that $|R(\mathbf{I}_1, \mathbf{I}_2) - R(\mathbf{I}'_1, \mathbf{I}'_2)| \leq N$ where $R(\mathbf{I}_1, \mathbf{I}_2)$ denotes the length of the exact result on the input $(\mathbf{I}_1, \mathbf{I}_2)$. By Fact 11, we have that

$$\widehat{R}(\mathbf{I}_1, \mathbf{I}_2) \sim_{(\epsilon,\delta)} \widehat{R}(\mathbf{I}'_1, \mathbf{I}'_2).$$

where $\widehat{R}(\mathbf{I}_1, \mathbf{I}_2)$ denotes the length of the result output by the naïve DO algorithm upon input $(\mathbf{I}_1, \mathbf{I}_2)$. Hence, the algorithm is $(\epsilon, \delta + \mathsf{negl}(N))$-differentially oblivious, where the extra $\mathsf{negl}(N)$ comes from the failure probability of ORAM. ◀

3.6 Oblivious Algorithm Building Blocks

We describe several oblivious algorithm building blocks. Unless otherwise noted, obliviousness is defined w.r.t. the input-length leakage (i.e., informally, speaking, only the input length is leaked).

Oblivious compaction. Given an input array where some elements are marked as distinguished, output an array where all distinguished elements are moved to the front, and all non-distinguished elements are moved to the end. The very recent works by Asharov et al. [11, 12] constructed a $O(n)$-time oblivious compaction algorithm that can compact any input array of length n. Their linear-time compaction algorithm is not *stable*, i.e., among the distinguished (or non-distinguished) elements, the output does not preserve the relative order the elements appeared in the input, and this non-stability is inherent [67].

To get our cache complexity result, we will adopt the randomized, cache-agnostic, oblivious compaction algorithm by Lin, Shi, and Xie [67]: their algorithm achieves optimal $O(n/B)$ cache complexity and $O(n \log \log n)$ runtime assuming that $M = \Omega(B^2)$ and $B \geq \log^{0.55} n$.

Oblivious sort. Ajtai, Komlós, and Szemerédi [4] showed that there is a sorting circuit with $O(n \log n)$ comparators that can correctly sort any input array containing n elements. Such a sorting circuit can be executed on a Random Access Machine (RAM) in $O(n \log n)$ time assuming that each element can be represented using $O(1)$ words.

The recent work by Ramachandran and Shi [75] constructed a *randomized*, cache-agnostic, oblivious sort algorithm that achieves $O(n \log n)$ runtime and $O((n/B) \log_{M/B}(n/B))$ cache complexity, assuming the tall cache assumption that $M = \Omega(B^2)$ and further $M = \Omega(\log^{1+\epsilon} n)$ for an arbitrarily small constant $\epsilon \in (0, 1)$.

Given oblivious sorting, we can realize a couple intermediate abstractions including oblivious send-receive and oblivious bin placement which we define below. Both primitives can be realized by invoking oblivious sorting constant number of times.

Oblivious send-receive. The send-receive primitive[11] solves the following problem. In the input, there is a source array and a destination array. The source array represents n senders, each of whom holds a key and a value; it is promised that all join keys are distinct. The destination array represents n' receivers each holding a join key. Now, have each receiver learn the value corresponding to the join key it is requesting from one of the sources. If the

[11] The send-receive abstraction is often referred to as oblivious routing in the data-oblivious algorithms literature [21, 27, 29]. We avoid the name "routing" because of its other connotations in the algorithms literature.

join key is not found, the receiver should receive $\perp$. Note that although each receiver wants only one value, a sender can send its values to multiple receivers.

Prior works [21, 27, 29] have shown that oblivious send-receive can be accomplished through a constant number of oblivious sorts on arrays of length $O(n + n')$. The algorithm is oblivious w.r.t. the leakage n and n', (i.e., informally, only the lengths of the input arrays are leaked).

Oblivious bin placement. A bin placement algorithm solves the following problem. Suppose we have m bins each of capacity $s_1, s_2, \ldots, s_m$, respectively. We are given an input array denoted $\mathbf{I}$, where each element is either a *filler* denoted $\perp$ or a *real* element that is tagged with a bin identifier $\beta \in [m]$ denoting which bin it wants to go to. It is promised that every bin will receive no more elements than its capacity. Now, move each real element in $\mathbf{I}$ to its desired bin. If any bin is not full after the real elements have been placed, pad it with filler elements at the end to its desired capacity. Finally, output the concatenation of the resulting bins.

Chan and Shi [29] describes an oblivious bin placement algorithm that solves the special case of the problem when all the bin sizes are equal. Their algorithm relies on a constant number of oblivious sorts. It is not difficult to extend their algorithm to the case when the bin sizes are not equal. For completeness, we describe the modified algorithm in the online full version [34]. Specifically, *the algorithm satisfies obliviousness w.r.t. to the leakage that contains the input size, as well as the sizes of all bins.* Let n denote the size of the input array, and let $S := \sum_{\beta \in [m]} s_\beta$ be the sum of the sizes of all bins. The runtime of the algorithm is upper bounded by $T_{\text{sort}}(n + S)$ and the cache-agnostic, cache complexity of the algorithm is upper bounded by $Q_{\text{sort}}(n + S)$, where $T_{\text{sort}}(n')$ and $Q_{\text{sort}}(n')$ denote the runtime and (cache-agnostic) cache complexity of oblivious sort over an input array of size n'.

Deferred Contents

Due to space constraints, we defer the full algorithmic details and proofs to the online full version [34].

References

1 Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

2 Archita Agarwal, Maurice Herlihy, Seny Kamara, and Tarik Moataz. Encrypted databases for differential privacy. *Proc. Priv. Enhancing Technol.*, 2019(3):170–190, 2019. `doi:10.2478/popets-2019-0042`.

3 Alok Aggarwal and S. Vitter, Jeffrey. The Input/Output Complexity of Sorting and Related Problems. *Commun. ACM*, 31(9):1116–1127, September 1988. `doi:10.1145/48529.48535`.

4 M. Ajtai, J. Komlós, and E. Szemerédi. An O(n log n) sorting network. In *STOC*, 1983.

5 Prabhanjan Ananth, Xiong Fan, and Elaine Shi. Towards attribute-based encryption for rams from LWE: sub-linear decryption, and more. In Steven D. Galbraith and Shiho Moriai, editors, *Advances in Cryptology - ASIACRYPT 2019 - 25th International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, December 8-12, 2019, Proceedings, Part I*, volume 11921 of *Lecture Notes in Computer Science*, pages 112–141. Springer, 2019.

6 Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravishankar Ramamurthy, and Ramarathnam Venkatesan. Orthogonal security with cipherbase. In *CIDR*, 2013.

7 Arvind Arasu and Raghav Kaushik. Oblivious query processing. In *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014*, pages 26–37. OpenProceedings.org, 2014.

8 Yuriy Arbitman, Moni Naor, and Gil Segev. De-amortized cuckoo hashing: Provable worst-case performance and experimental results. In *ICALP*, 2009.

9 Lars Arge, Michael A. Bender, Erik D. Demaine, Bryan Holland-Minkley, and J. Ian Munro. An optimal cache-oblivious priority queue and its application to graph algorithms. *SIAM Journal on Computing*, 36(6):1672–1695, 2007. doi:10.1137/S0097539703428324.

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 Martin Farach-Colton and Inge Li Gørtz, editors, *3rd Symposium on Simplicity in Algorithms, SOSA@SODA*, pages 8–14. SIAM, 2020.

11 Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. OptORAMa: Optimal Oblivious RAM. In *Advances in Cryptology - EUROCRYPT 2020*, 2020. To appear. See also: https://eprint.iacr.org/2018/892.

12 Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Enoch Peserico, and Elaine Shi. Oblivious parallel tight compaction. In *Information-Theoretic Cryptography (ITC)*, 2020.

13 Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. In *FOCS*, page 739–748, USA, 2008. IEEE Computer Society.

14 Sumeet Bajaj and Radu Sion. Trusteddb: A trusted hardware-based database with privacy and data confidentiality. *IEEE Trans. on Knowl. and Data Eng.*, 26(3):752–765, 2014.

15 Victor Balcer and Salil P. Vadhan. Differential privacy on finite computers. In Anna R. Karlin, editor, *9th Innovations in Theoretical Computer Science Conference (ITCS)*, volume 94, pages 43:1–43:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

16 Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. *J. ACM*, 64(6):40:1–40:58, 2017.

17 Amos Beimel, Kobbi Nissim, and Mohammad Zaheri. Exploring differential obliviousness. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2019, September 20-22, 2019, Massachusetts Institute of Technology, Cambridge, MA, USA*, volume 145 of *LIPIcs*, pages 65:1–65:20, 2019.

18 Andreas Björklund, Rasmus Pagh, Virginia Vassilevska Williams, and Uri Zwick. Listing Triangles. In *Proceedings of the 41st International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 8572 of *Lecture Notes in Computer Science*, pages 223–234. Springer International Publishing, 2014.

19 Jeremiah Blocki, Anupam Datta, and Joseph Bonneau. Differentially private password frequency lists. In *NDSS*, 2016.

20 Elette Boyle, Kai-Min Chung, and Rafael Pass. Large-scale secure computation: Multi-party computation for (parallel) RAM programs. In *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II*, volume 9216 of *Lecture Notes in Computer Science*, pages 742–762. Springer, 2015.

21 Elette Boyle, Kai-Min Chung, and Rafael Pass. Oblivious parallel ram. In *Theory of Cryptography Conference (TCC)*, 2015.

22 Mark Bun, Kobbi Nissim, and Uri Stemmer. Simultaneous private learning of multiple concepts. In *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science (ITCS)*, pages 369–380, 2016.

23 Ran Canetti, Yilei Chen, Justin Holmgren, and Mariana Raykova. Adaptive succinct garbled RAM or: How to delegate your database. In *Theory of Cryptography - 14th International Conference, TCC 2016-B, Beijing, China, October 31 - November 3, 2016, Proceedings, Part II*, volume 9986 of *Lecture Notes in Computer Science*, pages 61–90, 2016.

24 David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *ACM CCS*, page 668–679, 2015.

25 David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society, 2014.

26 David Cash, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *CRYPTO*, 2013.

27 Hubert Chan, Kai-Min Chung, and Elaine Shi. On the depth of oblivious parallel oram. manuscript, 2017.

28 T-H. Hubert Chan, Kai-Min Chung, Bruce M. Maggs, and Elaine Shi. Foundations of differentially oblivious algorithms. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, page 2448–2467, USA, 2019. Society for Industrial and Applied Mathematics.

29 T.-H. Hubert Chan and Elaine Shi. Circuit OPRAM: unifying statistically and computationally secure ORAMs and OPRAMs. In *Theory of Cryptography Conference, (TCC)*, 2017.

30 T.-H. Hubert Chan, Elaine Shi, and Dawn Song. Private and continual release of statistics. *TISSEC*, 14(3):26, 2011.

31 Jianer Chen, Iyad A. Kanj, and Ge Xia. Improved parameterized upper bounds for vertex cover. In *In MFCS*, 2006.

32 Yu-Chi Chen, Sherman S. M. Chow, Kai-Min Chung, Russell W. F. Lai, Wei-Kai Lin, and Hong-Sheng Zhou. Cryptography for parallel RAM from indistinguishability obfuscation. In *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science, Cambridge, MA, USA, January 14-16, 2016*, pages 179–190. ACM, 2016.

33 Shumo Chu, Magdalena Balazinska, and Dan Suciu. From theory to practice: Efficient join query evaluation in a parallel database system. In *SIGMOD Conference*, pages 63–78. ACM, 2015.

34 Shumo Chu, Danyang Zhuo, Elaine Shi, and T-H. Hubert Chan (randomized author ordering). Differentially oblivious database joins: Overcoming the worst-case curse of fully oblivious algorithms. Online full version of this paper. Cryptology ePrint Archive, Report 2021/593, 2021. URL: `https://eprint.iacr.org/2021/593`.

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

36 Reza Curtmola, Juan A. Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *ACM Conference on Computer and Communications Security*, pages 79–88, 2006.

37 Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Daniel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer Publishing Company, Incorporated, 1st edition, 2015.

38 Erik D. Demaine. Cache-oblivious algorithms and data structures. In *Lecture Notes from the EEF Summer School on Massive Data Sets*. BRICS, BRICS, University of Aarhus, Denmark, June 27–July 1 2002.

39 Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *TCC*, 2006.

40 Cynthia Dwork and Aaron Roth. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3-4):211–407, 2014. URL: `http://dblp.uni-trier.de/db/journals/fttcs/fttcs9.html#DworkR14`.

41 Saba Eskandarian and Matei Zaharia. Oblidb: Oblivious query processing for secure databases. *Proc. VLDB Endow.*, 13(2):169–183, 2019.

42 J. Flum and M. Grohe. *Parameterized Complexity Theory (Texts in Theoretical Computer Science. An EATCS Series)*. Springer-Verlag, Berlin, Heidelberg, 2006.

43 Matteo Frigo, Charles E Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 285–297. IEEE, 1999.

44 Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database systems - the complete book (2. ed.)*. Pearson Education, 2009.

45 Craig Gentry, Shai Halevi, Steve Lu, Rafail Ostrovsky, Mariana Raykova, and Daniel Wichs. Garbled ram revisited. In *EUROCRYPT*, pages 405–422, 2014.

46 Craig Gentry, Shai Halevi, Mariana Raykova, and Daniel Wichs. Outsourcing private ram computation. In *STOC*, 2014.

47 O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *STOC*, 1987.

48 Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 1996.

49 Shafi Goldwasser, Yael Tauman Kalai, Raluca A. Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. How to run turing machines on encrypted data. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II*, volume 8043 of *Lecture Notes in Computer Science*, pages 536–553. Springer, 2013.

50 S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In *CCS*, 2012.

51 Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. Learning to reconstruct: Statistical learning theory and encrypted database attacks. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 1067–1083. IEEE, 2019.

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

53 Anupam Gupta, Aaron Roth, and Jonathan Ullman. Iterative constructions and private data release. In *TCC*, volume 7194, pages 339–356, 2012.

54 Moritz Hardt and Guy N. Rothblum. A multiplicative weights mechanism for privacy-preserving data analysis. In *FOCS*, pages 61–70, 2010.

55 Xiao Hu and Ke Yi. Instance and output optimal parallel algorithms for acyclic joins. In *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS)*, page 450–463, 2019.

56 Mohammad Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *Network and Distributed System Security Symposium (NDSS)*, 2012.

57 Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel C. Rosu, and Michael Steiner. Outsourced symmetric private information retrieval. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

58 Yael Tauman Kalai and Omer Paneth. Delegating RAM computations. In *Theory of Cryptography - 14th International Conference, TCC 2016-B, Beijing, China, October 31 - November 3, 2016, Proceedings, Part II*, volume 9986 of *Lecture Notes in Computer Science*, pages 91–118, 2016.

59 Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. Generic attacks on secure outsourced databases. In *ACM CCS*, page 1329–1340, 2016.

60 Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. Accessing data while preserving privacy. *CoRR*, abs/1706.01552, 2017.

61 Mahmoud Abo Khamis, Hung Q. Ngo, Christopher Ré, and Atri Rudra. Joins via geometric resolutions: Worst-case and beyond. In *PODS*, pages 213–228. ACM, 2015.

62 Ilan Komargodski and Elaine Shi. Differentially oblivious turing machines. In *ITCS*, 2021.

63 Evgenios M. Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. The state of the uniform: Attacks on encrypted databases beyond the uniform query distribution. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1223–1240. IEEE, 2020.

64 Aleksandra Korolova, Krishnaram Kenthapadi, Nina Mishra, and Alexandros Ntoulas. Releasing search queries and clicks privately. In *Proceedings of the 18th International Conference on World Wide Web (WWW)*, pages 171–180, 2009.

65 Ios Kotsogiannis, Yuchao Tao, Xi He, Maryam Fanaeepour, Ashwin Machanavajjhala, Michael Hay, and Gerome Miklau. Privatesql: A differentially private sql query engine. *Proc. VLDB Endow.*, 12(11):1371–1384, 2019.

66 Ios Kotsogiannis, Yuchao Tao, Ashwin Machanavajjhala, Gerome Miklau, and Michael Hay. Architecting a differentially private SQL engine. In *CIDR*, 2019.

67 Wei-Kai Lin, Elaine Shi, and Tiancheng Xie. Can we overcome the $n \log n$ barrier for oblivious sorting? In *SODA*, 2019.

68 Sahar Mazloom and S. Dov Gordon. Secure computation with differentially private access patterns. In *CCS*, 2018.

69 Shay Moran and Amir Yehudayoff. A note on average-case sorting. *Order*, 33:23–28, 2015.

70 Muhammad Naveed, Seny Kamara, and Charles V. Wright. Inference attacks on property-preserving encrypted databases. In *ACM CCS*, page 644–655, 2015.

71 Hung Q. Ngo, Dung T. Nguyen, Christopher Ré, and Atri Rudra. Beyond worst-case analysis for joins with minesweeper. In *PODS*, pages 234–245. ACM, 2014.

72 Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, 2004.

73 Vasilis Pappas, Mariana Raykova, Binh Vo, Steven M. Bellovin, and Tal Malkin. Private search in the real world. In *Annual Computer Security Applications Conference (ACSAC)*, 2011.

74 Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: Protecting confidentiality with encrypted query processing. In *SOSP*, page 85–100, New York, NY, USA, 2011. Association for Computing Machinery.

75 Vijaya Ramachandran and Elaine Shi. Data oblivious algorithms for multicores. `https://eprint.iacr.org/2020/947.pdf`, 2020.

76 Vijaya Ramachandran and Elaine Shi. Data oblivious algorithms for multicores. In *SPAA*, 2021.

77 Tim Roughgarden. *Beyond the Worst-Case Analysis of Algorithms*. Cambridge University Press, 2020.

78 Leonard D. Shapiro. Join processing in database systems with large main memories. *ACM Trans. Database Syst.*, 11(3):239–264, 1986.

79 Micha Sharir and Mark H. Overmars. A simple output-sensitive algorithm for hidden surface removal. *ACM Trans. Graph.*, 11(1):1–11, 1992.

80 Elaine Shi, John Bethencourt, T-H. Hubert Chan, Dawn Song, and Adrian Perrig. Multi-dimensional range query over encrypted data. In *IEEE Symposium on Security and Privacy*, pages 350–364, 2007.

81 Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *ASIACRYPT*, 2011.

82 Dawn Xiaodong Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, Washington, DC, USA, 2000. IEEE Computer Society.

83 Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. Practical dynamic searchable symmetric encryption with small leakage. In *Network and Distributed System Security Symposium (NDSS)*, 2014.

84 Ananda Theertha Suresh. Differentially private anonymized histograms. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *NeurIPS*, pages 7969–7979, 2019.

85 Salil Vadhan. The complexity of differential privacy. In *Tutorials on the Foundations of Cryptography: Dedicated to Oded Goldreich*, pages 347–450, Cham, 2017. Springer International Publishing. `doi:10.1007/978-3-319-57048-8_7`.

86 Jeffrey Scott Vitter. External Memory Algorithms and Data Structures: Dealing with Massive Data. *ACM Comput. Surv.*, 33(2):209–271, June 2001. `doi:10.1145/384192.384193`.

87 Sameer Wagh, Paul Cuff, and Prateek Mittal. Differentially private oblivious RAM. *PoPETs*, 2018(4):64–84, 2018.

88 Xiao Shaun Wang, T-H. Hubert Chan, and Elaine Shi. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. In *CCS*, 2015.

89 Mihalis Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, page 82–94, 1981.

90 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*, 2017.