Optimal Oblivious Algorithms for Multi-Way Joins

Xiao Hu
University of Waterloo, Canada

Zhiang Wu
University of Waterloo, Canada

Abstract

In cloud databases, cloud computation over sensitive data uploaded by clients inevitably causes concern about data security and privacy. Even if cryptographic primitives and trusted computing environments are integrated into query processing to safeguard the actual contents of the data, access patterns of algorithms can still leak private information about data. *Oblivious RAM* (ORAM) and *circuits* are two generic approaches to address this issue, ensuring that access patterns of algorithms remain oblivious to the data. However, deploying these methods on insecure algorithms, particularly for multi-way join processing, is computationally expensive and inherently challenging.

In this paper, we propose a novel sorting-based algorithm for multi-way join processing that operates without relying on ORAM simulations or other security assumptions. Our algorithm is a non-trivial, provably oblivious composition of basic primitives, with time complexity matching the insecure worst-case optimal join algorithm, up to a logarithmic factor. Furthermore, it is *cache-agnostic*, with cache complexity matching the insecure lower bound, also up to a logarithmic factor. This clean and straightforward approach has the potential to be extended to other security settings and implemented in practical database systems.

2012 ACM Subject Classification Security and privacy → Management and querying of encrypted data; Information systems → Join algorithms

Keywords and phrases oblivious algorithms, multi-way joins, worst-case optimality

Digital Object Identifier 10.4230/LIPIcs.ICDT.2025.25

Related Version *Full Version*: `https://arxiv.org/pdf/2501.04216` [1]

1 Introduction

In outsourced query processing, a client entrusts sensitive data to a cloud service provider, such as Amazon, Google, or Microsoft, and subsequently issues queries to the provider. The service provider performs the required computations and returns the results to the client. Since these computations are carried out on remote infrastructure, ensuring the security and privacy of query evaluation is a critical requirement. Specifically, servers must remain oblivious to any information about the underlying data throughout the computation process. To achieve this, advanced cryptographic techniques and trusted computing hardware are employed to prevent servers from inferring the actual contents of the data [34, 19]. However, the memory accesses during execution may still lead to information leakage, posing an additional challenge to achieving comprehensive privacy. For example, consider the basic (natural) join operator on two database instances: $R_1 = \{(a_i, b_i) : i \in [N]\} \bowtie S_1 = \{(b_i, c_i) : i \in [N]\}$ and $R_2 = \{(a_i, b_1) : i \in [N]\} \bowtie S_2 = \{(b_1, c_i) : i \in [N]\}$ for some $N \in \mathbb{Z}^+$, where each pair of tuples can be joined if and only if they have the same b-value. Suppose each relation is sorted by their b-values. Using the merge join algorithm, there is only one access to S_1 between two consecutive accesses to R_1, but there are N accesses to S_2 between two consecutive accesses to R_2. Hence, the server can distinguish the degree information of join keys of the input data by observing the sequence of memory accesses. Moreover, if the server counts the total number of memory accesses, it can further infer the number of join results of the input data.


© Xiao Hu and Zhiang Wu;
licensed under Creative Commons License CC-BY 4.0

The notion of *obliviousness* was proposed to formally capture such a privacy guarantee on the *memory access pattern* of algorithms [31, 30]. This concept has inspired a substantial body of research focused on developing algorithms that achieve obliviousness in practical database systems [55, 24, 20, 17]. A generic approach to achieving obliviousness is *Oblivious RAM* (ORAM) [31, 41, 29, 52, 23, 48, 8], which translates each logical access into a poly-logarithmic (in terms of the data size) number of physical accesses to random locations of the memory. but the poly-logarithmic additional cost per memory access is very expensive in practice [16]. Another generic approach involves leveraging *circuits* [53, 26]. Despite their theoretical promise, generating circuits is inherently complex and resource-intensive, and integrating such constructions into database systems often proves to be inefficient. These challenges highlight the advantages of designing algorithms that are inherently oblivious to the input data, eliminating the need for ORAM frameworks or circuit constructions.

In this paper, we take on this question for *multi-way join processing*, and examine the insecure *worst-case optimal join* (WCOJ) algorithm [43, 44, 50], that can compute any join queries in time proportional to the maximum number of join results. Our objective is to investigate the intrinsic properties of the WCOJ algorithm and transform it into an oblivious version while preserving its optimal complexity guarantee.

1.1 Problem Definition

Multi-way join. A (natural) join query can be represented as a hypergraph $\mathcal{Q} = (\mathcal{V}, \mathcal{E})$ [2], where $\mathcal{V}$ models the set of attributes, and $\mathcal{E} \subseteq 2^{\mathcal{V}}$ models the set of relations. Let $\mathrm{dom}(x)$ be the domain of attribute $x \in \mathcal{V}$. An instance of $\mathcal{Q}$ is a function $\mathcal{R}$ that maps each $e \in \mathcal{E}$ to a set of tuples R_e, where each tuple $t \in R_e$ specifies a value in $\mathrm{dom}(x)$ for each attribute $x \in e$. The result of a join query $\mathcal{Q}$ over an instance $\mathcal{R}$, denoted by $\mathcal{Q}(\mathcal{R})$, is the set of all combinations of tuples, one from each relation R_e, that share the same values in their common attributes, i.e., $\mathcal{Q}(\mathcal{R}) = \left\{ t \in \prod_{x \in \mathcal{V}} \mathrm{dom}(x) \mid \forall e \in \mathcal{E}, \exists t_e \in R_e, \pi_e t = t_e \right\}$.

Let $N = \sum_{e \in \mathcal{E}} |R_e|$ be the *input size* of instance $\mathcal{R}$, i.e., the total number of tuples over all relations. Let $\mathrm{OUT} = |\mathcal{Q}(R)|$ be the *output size* of the join query $\mathcal{Q}$ over instance $\mathcal{R}$. We study the data complexity [2] of join algorithms by measuring their running time in terms of input and output size of the instance. We consider the size of $\mathcal{Q}$, i.e., $|\mathcal{V}|$ and $|\mathcal{E}|$, as constant.

Model of computation. We consider a two-level hierarchical memory model [40, 18]. The computation is performed within *trusted memory*, which consists of M registers of the same width. For simplicity, we assume that the trusted memory size is $c \cdot M$, where c is a constant. This assumption will not change our results by more than a constant factor. Since we assume the query size as a constant, the arity of each relation is irrelevant. Each tuple is assumed to fit into a single register, with one register allocated per tuple, including those from input relations as well as intermediate results. We further assume that $c \cdot M$ tuples from any set of relations can fit into the trusted memory. Input data and all intermediate results generated during the execution are encrypted and stored in an *untrusted memory* of unlimited size. Both trusted and untrusted memory are divided into blocks of size B. One memory access moves a block of B consecutive tuples from trusted to untrusted memory or vice versa. The complexity of an algorithm is measured by the number of such memory accesses.

An algorithm typically operates by repeating the following three steps: (1) read encrypted data from the untrusted memory into the trusted memory, (2) perform computation inside the trusted memory, and (3) Encrypt necessary data and write them back to the untrusted

memory. Adversaries can only observe the address of the blocks read from or written to the untrusted memory in (1) and (3), but not data contents. They also cannot interfere with the execution of the algorithm. The sequence of memory accesses to the untrusted memory in the execution is referred to as the "access pattern" of the algorithm. In this context, we focus on two specific scenarios of interest:

- **Random Access Model (RAM).** This model can simulate the classic RAM model with $M = O(1)$ and $B = 1$, where the trusted memory corresponds to $O(1)$ registers and the untrusted memory corresponds to the main memory. The *time complexity* in this model is defined as the number of accesses to the main memory by a RAM algorithm.
- **External Memory Model (EM).** This model can naturally simulate the classic EM model [4, 51], where the trusted memory corresponds to the main memory and the untrusted memory corresponds to the disk. Following prior work [28, 21, 18], we focus on the *cache-agnostic* EM algorithms, which are unaware of the values of M (memory size) and B (block size), a property commonly referred to as *cache-oblivious* in the literature. To avoid ambiguity, we use the terms "cache-agnostic" to refer to "cache-oblivious" and "oblivious" to refer to "access-pattern-oblivious" throughout this paper. The advantages of cache-agnostic algorithms have been extensively studied, particularly in multi-level memory hierarchies. A cache-agnostic algorithm can seamlessly adapt to operate efficiently between any two adjacent levels of the hierarchy. We adopt the *tall cache* assumption, $M = \Omega(B^2)$ and further $M = \Omega(\log^{1+\epsilon} N)$[1] for an arbitrarily small constant $\epsilon \in (0, 1)$, and the *wide block* assumption, $B = \Omega(\log^{0.55} N)$. These are standard assumptions widely adopted in the literature of EM algorithms [4, 51, 7, 28, 21, 18]. The *cache complexity* in this model is defined as the number of accesses to the disk by an EM algorithm.

Oblivious Algorithms. The notion of obliviousness is defined based on the access pattern of an algorithm. Memory accesses to the trusted memory are invisible to the adversary and, therefore, have no impact on security. Let $\mathcal{A}$ be an algorithm, $\mathcal{Q}$ a join query, and $\mathcal{R}$ an arbitrary input instance of $\mathcal{Q}$. We denote $\mathsf{Access}_{\mathcal{A}}(\mathcal{Q}, \mathcal{R})$ as the sequence of memory accesses made by $\mathcal{A}$ to the untrusted memory when given $(\mathcal{Q}, \mathcal{R})$ as the input, where each memory access is a read or write operation associated with a physical address. The join query $\mathcal{Q}$ and the size N of the input instance are considered non-sensitive information and can be safely exposed to the adversary. In contrast, all input tuples are considered sensitive information and should be hidden from adversaries. This way, the access pattern of an oblivious algorithm $\mathcal{A}$ should only depend on $\mathcal{Q}$ and N, ensuring no leakage of sensitive information.

▶ **Definition 1** (Obliviousness [30, 31, 15]). *An algorithm $\mathcal{A}$ is oblivious for a join query $\mathcal{Q}$, if given an arbitrary parameter $N \in \mathbb{Z}^+$, for any pair of instances $\mathcal{R}, \mathcal{R}'$ of $\mathcal{Q}$ with input size N, $\mathsf{Access}_{\mathcal{A}}(\mathcal{Q}, \mathcal{R}) \overset{\delta}{\equiv} \mathsf{Access}_{\mathcal{A}}(\mathcal{Q}, \mathcal{R}')$, where δ is a negligible function in terms of N. Specifically, for any positive constant c, there exists N_c such that $\delta(N) < \frac{1}{N^c}$ for any $N > N_c$. The notation $\overset{\delta}{\equiv}$ indicates the statistical distance between two distributions is at most δ.*

This notion of obliviousness applies to both deterministic and randomized algorithms. For a randomized algorithm, different execution states may arise from the same input instance due to the algorithm's inherent randomness. Each execution state corresponds to a specific sequence of memory accesses, allowing the access pattern to be modeled as a random variable with an associated probability distribution over the set of all possible access patterns. The

[1] In this work, $\log(\cdot)$ always means $\log_2(\cdot)$ and should be distinguished from $\log_{\frac{M}{B}}(\cdot)$.

statistical distance between two probability distributions is typically quantified using standard metrics, such as the total variation distance. A randomized algorithm is indeed oblivious if its access pattern exhibits statistically indistinguishable distributions across all input instances of the same size. Relatively simpler, a deterministic algorithm is oblivious if it displays an identical access pattern for all input instances of the same size.

1.2 Review of Existing Results

Oblivious RAM. ORAM is a general randomized framework designed to protect access patterns [31]. In ORAM, each logical access is translated into a poly-logarithmic number of random physical accesses, thereby incurring a poly-logarithmic overhead. Goldreich et al. [31] established a lower bound $\Omega(\log N)$ on the access overhead of ORAMs in the RAM model. Subsequently, Asharov et al. [8] proposed a theoretically optimal ORAM construction with an overhead of $O(\log N)$ in the RAM model under the assumption of the existence of a one-way function, which is rather impractical [47]. It remains unknown whether a better cache complexity than $O(\log N)$ can be shown for such a construction. Path ORAM [48] is currently the most practical ORAM construction, but it introduces an $O(\log^2 N)$ overhead and requires $\Omega(1)$ trusted memory. In the EM model, one can place the tree data structures for ORAM in an Emde Boas layout, resulting in a memory access overhead of $O(\log N \cdot \log_B N)$.

Insecure Join Algorithms. The WCOJ algorithm [43] have been developed to compute any join query in $O(N^{\rho^*})$ time[2], where ρ^* is the fractional edge cover number of the join query (formally defined in Section 2.1). The optimality is implied by the AGM bound [9]. [3] However, these WCOJ algorithms are not oblivious. In Section 4, we use triangle join as an example to illustrate the information leakage from the WCOJ algorithm. Another line of research also explored output-sensitive join algorithms. A join query can be computed in $O((N^{\mathsf{subw}} + \mathrm{OUT}) \cdot \mathsf{polylog} N))$ time [54, 3], where subw is the submodular-width of the join query. For example, $\mathsf{subw} = 1$ if and only if the join query is acyclic [12, 25]. These algorithms are also not oblivious due to various potential information leakages. For instance, the total number of memory accesses is influenced by the output size, which can range from a constant to a polynomially large value relative to the input size. A possible mitigation strategy is *worst-case padding*, which involves padding dummy accesses to match the worst case. However, this approach does not necessarily result in oblivious algorithms, as their access patterns may still vary significantly across instances with the same input size.

In contrast, there has been significantly less research on multi-way join processing in the EM model. First of all, we note that an EM version of the WCOJ algorithm incurs at least $\Omega\left(\frac{N^{\rho^*}}{B}\right)$ cache complexity since there are $\Theta(N^{\rho^*})$ join results in the worst case and all join results should be written back to disk. For the basic two-way join, the nested-loop algorithm has cache complexity $O\left(\frac{N^2}{B}\right)$ and the sort-merge algorithm has cache complexity $O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B} + \frac{\mathrm{OUT}}{B}\right)$. For multi-way join queries, an EM algorithm with cache complexity $O\left(\frac{N^{\rho^*}}{M^{\rho^*-1}B}\cdot\log_{\frac{M}{B}}\frac{N}{B} + \frac{\mathrm{OUT}}{B}\right)$ has been achieved for Berge-acyclic joins [37], α-acyclic joins [36, 39], graph joins [38, 22] and Loomis-Whitney joins [39].[4] These results

[2] A hashing-based algorithm achieves $O(N^{\rho^*})$ time in the worst case using the lazy array technique [27].

[3] The maximum number of join results produced by any instance of input size N is $O(N^{\rho^*})$, which is also tight in the sense that there exists some instance of input size N that can produce $\Theta(N^{\rho^*})$ join results.

[4] Some of these algorithms have been developed for the *Massively Parallel Computation* (MPC) model [11] and can be adapted to the EM model through the MPC-to-EM reduction [39].

Table 1 Comparison between previous and new oblivious algorithms for multi-way joins. N is the input size. ρ^* and ρ are the input join query's fractional and integral edge cover numbers, respectively. M is the trusted memory size. B is the block size.

	Previous Results	New Results
RAM model	$O\left(N^{\rho^*} \cdot \log N\right)$ [44, 8] (one-way function assumption)	$O\left(N^{\rho^*} \cdot \log N\right)$ (no assumption)
Cache-agnostic EM model	$O\left(\frac{N^{\min\{\rho^*+1,\rho\}}}{B} \cdot \log_{\frac{M}{B}} \frac{N^{\min\{\rho^*+1,\rho\}}}{B}\right)$ (no assumption)	$O\left(\frac{N^{\rho^*}}{B} \cdot \log_{\frac{M}{B}} \frac{N^{\rho^*}}{B}\right)$ (tall cache and wide block assumptions)

were previously stated without including the output-dependent term $\frac{\text{OUT}}{B}$ since they do not consider the cost of writing join results back to disk. Again, even padding the output size to be as large as the worst case, these algorithms remain non-oblivious since their access patterns heavily depend on the input data. Furthermore, even in the insecure setting, no algorithm with a cache complexity $O\left(\frac{N^{\rho^*}}{B}\right)$ is known for general join queries.

Oblivious Join Algorithms. Oblivious algorithms have been studied for join queries in both the RAM and EM models. In the RAM model, the naive nested-loop algorithm can be transformed into an oblivious one by incorporating some dummy writes, as it enumerates all possible combinations of tuples from input relations in a fixed order. This algorithm runs in $O(N^{|\mathcal{E}|})$ time, where $|\mathcal{E}|$ is the number of relations in the join query. Wang et al. [53] designed circuits for conjunctive queries - capturing all join queries as a special case - whose time complexity matches the AGM bound up to poly-logarithmic factors. Running such a circuit will automatically yield an oblivious join algorithm with $O\left(N^{\rho^*} \cdot \mathsf{polylog} N\right)$ time complexity. By integrating the insecure WCOJ algorithm [44] with the optimal ORAM [8], it is possible to achieve an oblivious algorithm with $O(N^{\rho^*} \cdot \log N)$ time complexity, albeit under restrictive theoretical assumptions. Alternatively, incorporating the insecure WCOJ algorithm into the Path ORAM yields an oblivious join algorithm with $O\left(N^{\rho^*} \cdot \log^2 N\right)$ time complexity.

In the EM model, He et al. [35] proposed a cache-agnostic nested-loop join algorithm for the basic two-way join $R \bowtie S$ with $O\left(\frac{|R| \cdot |S|}{B}\right)$ cache complexity, which is also oblivious. Applying worst-case padding and the optimal ORAM construction to the existing EM join algorithms, we can derive an oblivious join algorithm with $O\left(\frac{N^{\rho^*}}{B} \cdot \log_{\frac{M}{B}} \frac{N}{B} \cdot \log N\right)$ cache complexity for specific cases such as acyclic joins, graph joins and Loomis-Whitney joins. However, these algorithms are not cache-agnostic. For general join queries, no specific oblivious algorithm has been proposed for the EM model, aside from results derived from the oblivious RAM join algorithm. These results yield cache complexities of either $O\left(N^{\rho^*} \cdot \log N\right)$ or $O\left(N^{\rho^*} \cdot \log N \cdot \log_B N\right)$, as they rely heavily on retrieving tuples from hash tables or range search indices.

Relaxed Variants of Oblivious Join Algorithms. Beyond fully oblivious algorithms, researchers have explored relaxed notions of obliviousness by allowing specific types of leakage, such as the join size, the multiplicity of join values, and the size of intermediate results. One relevant line of work examines join processing with released input and output sizes. For example, integrating an insecure output-sensitive join algorithm into an ORAM framework produces a

relaxed oblivious algorithm with $O\left((N^{\mathsf{subw}} + \mathrm{OUT}) \cdot \mathrm{polylog} N\right)$ time complexity. It is noted that relaxed oblivious algorithm with the same time complexity $O((N + \mathrm{OUT}) \cdot \log N)$ have been proposed without requiring ORAM [6, 40] for the basic two-way join as well as acyclic joins. Although not fully oblivious, these algorithms serve as fundamental building blocks for developing our oblivious algorithms for general join queries. Another line of work considered *differentially oblivious* algorithms [15, 13, 18], which require only that access patterns appear similar across *neighboring* input instances. However, differentially oblivious algorithms have so far been limited to the basic two-way join [18]. This paper does not pursue this direction further.

1.3 Our Contribution

Our main contribution can be summarized as follows (see Table 1):

- We give a nested-loop-based algorithm for general join queries with $O\left(N^{\min\{\rho^*+1,\rho\}} \cdot \log N\right)$ time complexity and $O\left(\frac{N^{\min\{\rho^*+1,\rho\}}}{B} \cdot \log_{\frac{M}{B}} \frac{N^{\min\{\rho^*+1,\rho\}}}{B}\right)$ cache complexity, where ρ^* and ρ are the fractional and integral edge cover number of the join query, respectively (formally defined in Section 2.1). This algorithm is also cache-agnostic. For classes of join queries with $\rho^* = \rho$, such as acyclic joins, even-length cycle joins and boat joins (see Section 3), this is almost optimal up to logarithmic factors.
- We design an oblivious algorithm for general join queries with $O\left(N^{\rho^*} \cdot \log N\right)$ time complexity, which has matched the insecure counterpart by a logarithmic factor and recovered the previous ORAM-based result, which assumes the existence of one-way functions. This algorithm is also cache-agnostic, with $O\left(\frac{N^{\rho^*}}{B} \cdot \log_{\frac{M}{B}} \frac{N^{\rho^*}}{B}\right)$ cache complexity. This cache complexity can be simplified to $O\left(\frac{N^{\rho^*}}{B} \cdot \log_{\frac{M}{B}} \frac{N}{B}\right)$ when $B < N^{\frac{c-\rho^*}{c-1}}$ for some sufficiently large constant c. This result establishes the first worst-case near-optimal join algorithm in the insecure EM model when all join results are returned to disk.
- We develop an improved algorithm for relaxed two-way joins with better cache complexity, which is also cache-agnostic. By integrating our oblivious algorithm with generalized hypertree decomposition [33], we obtain a relaxed oblivious algorithm for general join queries with $O\left((N^{\mathsf{fhtw}} + \mathrm{OUT}) \cdot \log N\right)$ time complexity and $O\left(\frac{N^{\mathsf{fhtw}}+\mathrm{OUT}}{B} \cdot \log_{\frac{M}{B}} \frac{N^{\mathsf{fhtw}}+\mathrm{OUT}}{B}\right)$ cache complexity, where fhtw is the fractional hypertree width of the input query.

Roadmap. This paper is organized as follows. In Section 2, we introduce the preliminaries for building our algorithms. In Section 3, we show our first algorithm based on the nested-loop algorithm. While effective, this algorithm is not always optimal, as demonstrated with the triangle join. In Section 4, we use triangle join to demonstrate the leakage of insecure WCOJ algorithm and show how to transform it into an oblivious algorithm. We introduce our oblivious WCOJ algorithm for general join queries in Section 5, and conclude in Section 6.

2 Preliminaries

2.1 Fractional and Integral Edge Cover Number

For a join query $\mathcal{Q} = (\mathcal{V}, \mathcal{E})$, a function $W : \mathcal{E} \to [0, 1]$ is a *fractional edge cover* for $\mathcal{Q}$ if $\sum_{e \in \mathcal{E}: x \in e} W(e) \geq 1$ for any $x \in \mathcal{V}$. An *optimal fractional edge cover* is the one minimizing $\sum_{e \in \mathcal{E}} W(e)$, which is captured by the following linear program:

$$\min \sum_{e \in \mathcal{E}} W(e) \quad \text{s.t.} \quad \sum_{e \in \mathcal{E}: x \in e} W(e) \geq 1, \forall x \in \mathcal{V}; \quad W(e) \in [0, 1], \forall e \in \mathcal{E} \tag{1}$$

The optimal value of (1) is the *fractional edge cover number* of $\mathcal{Q}$, denoted as $\rho^*(\mathcal{Q})$. Similarly, a function $W : \mathcal{E} \to \{0, 1\}$ is an *integral edge cover* if $\sum_{e \in \mathcal{E}: x \in e} W(e) \geq 1$ for any $x \in \mathcal{V}$. The *optimal integral edge cover* is the one minimizing $\sum_{e \in \mathcal{E}} W(e)$, which can be captured by a similar linear program as (1) except that $W(e) \in [0, 1]$ is replaced with $W(e) \in \{0, 1\}$. The optimal value of this linear program is the *integral edge cover number* of $\mathcal{Q}$, denoted as $\rho(\mathcal{Q})$.

2.2 Oblivious Primitives

We introduce the following oblivious primitives, which form the foundation of our algorithms. Each primitive displays an identical access pattern across instances of the same input size.

Linear Scan. Given an input array of N elements, a linear scan of all elements can be done with $O(N)$ time complexity and $O(\frac{N}{B})$ cache complexity in a cache-agnostic way.

Sort [5, 10]. Given an input array of N elements, the goal is to output an array according to some predetermined ordering. The classical bitonic sorting network [10] requires $O(N \cdot \log^2 N)$ time. Later, this time complexity has been improved to $O(N \cdot \log N)$ [5] in 1983. However, due to the large constant parameter hidden behind $O(N \cdot \log N)$, the classical bitonic sorting is more commonly used in practice, particularly when the size N is not too large. Ramachandran and Shi [45] showed a randomized algorithm for sorting with $O(N \cdot \log N)$ time complexity and $O\left(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}\right)$ cache complexity under the tall cache assumption.

Compact [32, 46]. Given an input array of N elements, some of which are distinguished as $\perp$, the goal is to output an array with all non-distinguished elements moved to the front before any $\perp$, while preserving the ordering of non-distinguished elements. Lin et al. [42] showed a randomized algorithm for compaction with $O(N \cdot \log \log N)$ time complexity and $O\left(\frac{N}{B}\right)$ cache complexity under the tall cache assumption.

We use the above primitives to construct additional building blocks for our algorithms. To ensure obliviousness, all outputs from these primitives include a fixed size equal to the worst-case scenario, i.e., N, comprising both real and dummy elements. All these primitives achieve $O(N \cdot \log N)$ time complexity and $O\left(\frac{N}{B} \cdot \log_{\frac{M}{B}} \frac{N}{B}\right)$ cache complexity. Further details are provided in the full version [1].

SemiJoin. Given two input relations R, S of at most N tuples and their common attribute(s) x, the goal is to output the set of tuples in R that can be joined with at least one tuple in S.

Project. Given an input relation R of N tuples defined over attributes e, and a subset of attributes $x \subseteq e$, the goal is to output $\{t \in R : \pi_x t\}$, ensuring no duplication.

Intersect. Given two input arrays R, S of at most N elements, the goal is to output $R \cap S$.

Augment. Given a relation R and k additional relations $S_1, S_2, \cdots, S_k$ (each with at most N tuples) sharing common attribute(s) x, the goal is to attach each tuple t the number of tuples in S_i (for each $i \in [k]$) that can be joined with t on x.

We note that any sequential composition of oblivious primitives yields more complex algorithms that remain oblivious, which is the key principle underlying our approach.

2.3 Oblivious Two-way Join

NestedLoop. Nested-loop algorithm can compute $R \bowtie S$ with $O(|R| \cdot |S|)$ time complexity, which iterates all combinations of tuples from R, S and writes a join result (or a dummy result, if necessary, to maintain obliviousness). He et al. [35] proposed a cache-agnostic version in the EM model with $O\left(\frac{|R|\cdot|S|}{B}\right)$ cache complexity, which is also oblivious.

▶ **Theorem 2** ([35]). *For $R \bowtie S$, there is a cache-agnostic algorithm that can compute $R \bowtie S$ with $O(|R| \cdot |S|)$ time complexity and $O\left(\frac{|R|\cdot|S|}{B}\right)$ cache complexity, whose access pattern only depends on $M, B, |R|$ and $|S|$.*

RelaxedTwoWay. The relaxed two-way join algorithm [6, 40] takes as input two relations R, S and a parameter $\tau \geq |R \bowtie S|$, and output a table of τ elements containing join results of $R \bowtie S$, whose access pattern only depends on $|R|, |S|$ and τ. This algorithm can also be easily transformed into a cache-agnostic version with $O((|R|+|S|+\tau)\cdot\log(|R|+|S|+\tau))$ time complexity and $O\left(\frac{|R|+|S|+\tau}{B}\cdot\log\tau\right)$ cache complexity. In the full version [1], we show how to improve the cache complexity of this algorithm without sacrificing the time complexity.

▶ **Theorem 3.** *For $R \bowtie S$ and a parameter $\tau \geq |R \bowtie S|$, there is a cache-agnostic algorithm that can compute $R \bowtie S$ with $O((|R| + |S| + \tau) \cdot \log(|R| + |S| + \tau))$ time complexity and $O\left(\frac{|R|+|S|+\tau}{B} \cdot \log_{\frac{M}{B}} \frac{|R|+|S|+\tau}{B}\right)$ cache complexity under the tall cache and wide block assumptions, whose access pattern only depends on $M, B, |R|, |S|$ and τ.*

3 Beyond Oblivious Nested-loop Join

Although the nested-loop join algorithm is described for the two-way join, it can be extended to multi-way joins. For a general join query with k relations, the nested-loop primitive can be recursively invoked $k-1$ times, resulting in an oblivious algorithm with $O\left(\frac{N^k}{B}\right)$ cache complexity. Careful inspection reveals that we do not necessarily feed all input relations into the nested loop; instead, we can restrict enumeration to combinations of tuples from relations included in an integral edge cover of the join query. Recall that for $\mathcal{Q} = (\mathcal{V}, \mathcal{E})$, an integral edge cover of $\mathcal{Q}$ is a function $W : \mathcal{E} \to \{0, 1\}$, such that $\sum_{e:x\in e} W(e) \geq 1$ holds for every $x \in \mathcal{V}$. While enumerating combinations of tuples from relations "chosen" by W, we can apply semi-joins using remaining relations to filter intermediate join results.

As described in Algorithm 1, it first chooses an optimal integral edge cover W^* of $\mathcal{Q}$ (line 1), and then invokes the NestedLoop primitive to iteratively compute the combinations of tuples from relations with $W^*(e) = 1$ (line 7), whose output is denoted as L. Meanwhile, we apply the semi-join between L and the remaining relations (line 8).

Below, we analyze the complexity of this algorithm. First, as $|\mathcal{E}'| \leq \rho$, the intermediate join results materialized in the while-loop is at most $O(N^\rho)$. After semi-join filtering, the number of surviving results is at most $O\left(N^{\rho^*}\right)$. In this way, the number of intermediate results materialized by line 7 is at most $O\left(N^{\rho^*+1}\right)$. Putting everything together, we obtain:

▶ **Theorem 4.** *For a general join query $\mathcal{Q}$, there is an oblivious and cache-agnostic algorithm that can compute $\mathcal{Q}(\mathcal{R})$ for an arbitrary instance $\mathcal{R}$ of input size N with $O\left(N^{\min\{\rho,\rho^*+1\}}\right)$ time complexity and $O\left(\frac{N^{\min\{\rho,\rho^*+1\}}}{B}\cdot\log_{\frac{M}{B}}\frac{N^{\min\{\rho,\rho^*+1\}}}{B}\right)$ cache complexity under the tall cache and wide block assumptions, where ρ^* and ρ are the optimal fractional and integral edge cover number of $\mathcal{Q}$, respectively.*

Algorithm 1 ObliviousNestedLoopJoin$(\mathcal{Q}, \mathcal{R})$.

```
W* ← an optimal integral edge cover of Q, L ← ∅;
E' ← {e ∈ E : W*(e) = 1};
while E' ≠ ∅ do
    e ← an arbitrary relation in E';
    E' ← E' − {e};
    if L = ∅ then L ← R_e;
    else L ← NestedLoop(L, R_e);
    foreach e' ∈ E − {e} do L ← SemiJoin(L, R_e');
return L;
```

It is important to note that any oblivious join algorithm incurs a cache complexity of $\Omega\left(\frac{N^{\rho^*}}{B}\right)$, so Theorem 4 is optimal up to a logarithmic factor for join queries where $\rho = \rho^*$. Below, we list several important classes of join queries that exhibit this desirable property:

▶ **Example 5** (α-acyclic Join)**.** A join query $\mathcal{Q}$ is α-acyclic [12, 25] if there is a tree structure $\mathcal{T}$ of $\mathcal{Q} = (\mathcal{V}, \mathcal{E})$ such that (1) there is a one-to-one correspondence between relations in $\mathcal{Q}$ and nodes in $\mathcal{T}$; (2) for every attribute $x \in \mathcal{V}$, the set of nodes containing x form a connected subtree of $\mathcal{T}$. Any α-acyclic join admits an optimal fractional edge cover that is integral [36].

▶ **Example 6** (Even-length Cycle Join)**.** An even-length cycle join is defined as $\mathcal{Q} = R_1(x_1, x_2) \bowtie R_2(x_2, x_3) \bowtie \cdots \bowtie R_{k-1}(x_{k-1}, x_k) \bowtie R_k(x_k, x_1)$ for some even integer k. It has two integral edge covers $\{R_1, R_3, \cdots, R_{k-1}\}$ and $\{R_2, R_4, \cdots, R_k\}$, both of which are also an optimal fractional edge cover. Hence, $\rho^* = \rho = \frac{k}{2}$.

▶ **Example 7** (Boat Join)**.** A boat join is defined as $\mathcal{Q} = R_1(x_1, y_1) \bowtie R_2(x_2, y_2) \bowtie \cdots \bowtie R_k(x_k, y_k) \bowtie R_{k+1}(x_1, x_2, \cdots, x_k) \bowtie R_{k+2}(y_1, y_2, \cdots, y_k)$. It has an integral edge cover $\{R_1, R_2\}$ that is also an optimal fractional edge cover. Hence, $\rho^* = \rho = 2$.

4 Warm Up: Triangle Join

The simplest join query that oblivious nested-loop join algorithm cannot solve optimally is the triangle join $\mathcal{Q}_\triangle = R_1(x_2, x_3) \bowtie R_2(x_1, x_3) \bowtie R_3(x_1, x_2)$, which has $\rho = 2$ and $\rho^* = \frac{3}{2}$. While various worst-case optimal algorithms for the triangle join have been proposed in the RAM model, none of these are oblivious due to their inherent leakage of intermediate statistics. Below, we outline the issues with existing insecure algorithms and propose a strategy to make them oblivious.

Insecure Triangle Join Algorithm 2. We start with attribute x_1. Each value $a \in \mathrm{dom}(x_1)$ induces a subquery $\mathcal{Q}_a = R_1 \bowtie (\sigma_{x_1=a} R_2) \bowtie (\sigma_{x_1=a} R_3)$. Moreover, a value $a \in \mathrm{dom}(x_1)$ is *heavy* if $|\pi_{x_3}\sigma_{x_1=a}R_2| \cdot |\pi_{x_2}\sigma_{x_1=a}R_3|$ is greater than $|R_1|$, and *light* otherwise. If a is light, $\mathcal{Q}_a$ is computed by materializing the Cartesian product between $\pi_{x_3}\sigma_{x_1=a}R_1$ and $\pi_{x_2}\sigma_{x_1=a}R_3$, and then filter the intermediate result by a semi-join with R_1. Every surviving tuple forms a join result with a, which will be written back to untrusted memory. If a is heavy, $\mathcal{Q}_a$ is computed by applying the semi-joins between R_1 with $\sigma_{x_1=a}R_2$ and $\sigma_{x_1=a}R_3$. This algorithm achieves a time complexity of $O(N^{\frac{3}{2}})$ (see [43] for detailed analysis), but it leaks sensitive information through the following mechanisms:

Algorithm 2 Compute $\mathcal{Q}_\triangle$ by power of two choices.

```
1 L ← ∅;
2 foreach a ∈ (π_{x1} R2) ∩ (π_{x1} R3) do
3     if |σ_{x1=a} R2| · |σ_{x1=a} R3| ≤ |R1| then
4         foreach (b, c) ∈ (π_{x2} σ_{x1=a} R3) × (π_{x3} σ_{x1=a} R2) do
5             if (b, c) ∈ R1 then write (a, b, c) to L;
6     else
7         foreach (b, c) ∈ R1 do
8             if (a, b) ∈ R3 and (a, c) ∈ R2 then write (a, b, c) to L;
9 return L;
```

Algorithm 3 Inject Obliviousness to Algorithm 2.

```
1  A ← (π_{x1} R2) ∩ (π_{x1} R3) by Project and Intersect;
2  A ← Augment(A, {R2, R3}, x1);
3  A1, A2 ← ∅;
4  while read (a, Δ1, Δ2) from A do // Δ1 = |π_{x3} σ_{x1=a} R2| and Δ2 = |π_{x2} σ_{x1=a} R3|
5      if Δ1 · Δ2 ≤ |R1| then write a to A1, write ⊥ to A2;
6      else write a to A2, write ⊥ to A1;
7  L1 ← RelaxedTwoWay(A2, R1, N^{3/2});
8  L1 ← SemiJoin(L1, R2), L1 ← SemiJoin(L1, R3);
9  R2 ← SemiJoin(R2, A1), R3 ← SemiJoin(R3, A1);
10 L2 ← RelaxedTwoWay(R2, R3, N^{3/2});
11 L2 ← SemiJoin(L2, R1);
12 return Compact L1 ∪ L2 while keeping the first N^{3/2} tuples;
```

- $|(\pi_{x_1} R_2) \cap (\pi_{x_1} R_3)|$ is leaked by the number of for-loop iterations in line 2;
- $|\pi_{x_2}\sigma_{x_1=a} R_3|$ and $|\pi_{x_3}\sigma_{x_1=a} R_2|$ are leaked by the number of for-loop iterations in line 4;
- The sizes of heavy and light values in $(\pi_{x_1} R_2) \cap (\pi_{x_1} R_3)$ are leaked by the if-else condition in lines 3 and 6;

To protect intermediate statistics, we pad dummy tuples to every intermediate result (such as $(\pi_{x_1} R_2) \cap (\pi_{x_1} R_3)$, $\pi_{x_3}\sigma_{x_1=a} R_2$ and $\pi_{x_2}\sigma_{x_1=a} R_3$) to match the worst-case size N. To hide heavy and light values, we replace conditional if-else branches with a unified execution plan by visiting every possible combination of $(\pi_{x_2}\sigma_{x_1=a} R_3) \times (\pi_{x_3}\sigma_{x_1=a} R_2)$ and every tuple of R_1. By integrating these techniques, this strategy leads to N^2 memory accesses, hence destroying the power of two choices that is a key advantage in the insecure WCOJ algorithm.

Inject Obliviousness to Algorithm 2. To inject obliviousness into Algorithm 2, Algorithm 3 leverages oblivious primitives to ensure the same access pattern across all instances of the input size. Here's a breakdown of how this is achieved and why it works. We start with computing $A = (\pi_{x_1} R_2) \cap (\pi_{x_1} R_3)$ by the Intersect primitive. Then, we partition values in A into two subsets A_1, A_2, depending on the relative order between $|\pi_{x_3}\sigma_{x_1=a} R_2| \cdot |\pi_{x_2}\sigma_{x_1=a} R_3|$ and $|R_1|$. We next compute the following two-way joins $A_2 \bowtie R_1$ and $(R_2 \ltimes A_1) \bowtie (R_3 \ltimes A_2)$ by invoking the RelaxedTwoWay primitive separately, each with the upper bound $N^{\frac{3}{2}}$. At last, we filter intermediate join results by the SemiJoin primitive and remove unnecessary dummy tuples by the Compact primitive.

Algorithm 4 Compute $\mathcal{Q}_\triangle$ by delaying computation.

```
1 L ← ∅;
2 foreach a ∈ (π_{x1} R2) ∩ (π_{x1} R3) do
3     foreach b ∈ (π_{x2} σ_{x1=a} R3) ∩ (π_{x2} R1) do
4         foreach c ∈ (π_{x3} σ_{x2=b} R1) ∩ (π_{x3} σ_{x1=a} R2) do
5             write (a, b, c) to L;
6 return L;
```

Algorithm 5 Inject Obliviousness to Algorithm 4.

```
1 R3 ← Augment(R3, R1, x2), R3 ← Augment(R3, R2, x1);
2 K1, K2 ← ∅;
3 while read (t, Δ1, Δ2) from R3 do // Suppose Δ_i = |R_i ⋉ {t}|
4     if Δ1 ≤ Δ2 then write t to K1, write ⊥ to K2;
5     else write t to K2, write ⊥ to K1;
6 L1 ← RelaxedTwoWay(K1, R1, N^{3/2}), L1 ← SemiJoin(L1, R2);
7 L2 ← RelaxedTwoWay(K2, R2, N^{3/2}), L2 ← SemiJoin(L2, R1);
8 return Compact L1 ∪ L2 while keeping the first N^{3/2} tuples;
```

Analysis of Algorithm 3. It suffices to show that $|(R_2 \ltimes A_1) \bowtie (R_3 \ltimes A_1)| \leq N^{\frac{3}{2}}$ and $|A_2 \bowtie R_1| \leq N^{\frac{3}{2}}$, which directly follows from the query decomposition lemma [44]:

$$\sum_{a \in A} \min\left\{|\sigma_{x_1=a} R_2| \cdot |\sigma_{x_1=a} R_3|, |R_1|\right\} \leq \sum_{a \in A} \left(|R_2 \ltimes a| \cdot |R_3 \ltimes a|\right)^{\frac{1}{2}} \cdot |R_1 \ltimes a|^{\frac{1}{2}} \leq N^{\frac{3}{2}}.$$

All other primitives have $O(N \cdot \log N)$ time complexity and $O\left(\frac{N}{B} \cdot \log_{\frac{M}{B}} \frac{N}{B}\right)$ cache complexity. Hence, this whole algorithm incurs $O\left(N^{\frac{3}{2}} \cdot \log N\right)$ time complexity and $O\left(\frac{N^{\frac{3}{2}}}{B} \cdot \log_{\frac{M}{B}} \frac{N^{\frac{3}{2}}}{B}\right)$ cache complexity. As each step is oblivious, the composition of all these steps is also oblivious.

Insecure Triangle Join Algorithm 4. We start with attribute x_1. We first compute the candidate values in x_1 that appear in some join results, i.e., $(\pi_{x_1} R_2) \cap (\pi_{x_1} R_3)$. For each candidate value a, we retrieve the candidate values in x_2 that can appear together with a in some join results, i.e., $(\pi_{x_2} \sigma_{x_1=a} R_3) \cap (\pi_{x_2} R_1)$. Furthermore, for each candidate value b, we explore the possible values in x_3 that can appear together with (a, b) in some join results. More precisely, every value c appears in $\pi_{x_3} \sigma_{x_2=b} R_1$ as well as $\pi_{x_3} \sigma_{x_1=a} R_2$ forms a triangle with a, b. This algorithm runs in $O(N^{\frac{3}{2}})$ time (see [44] for detailed analysis). Similarly, it is not oblivious as the following intermediate statistics may be leaked:

- $|(\pi_{x_1} R_2) \cap (\pi_{x_1} R_3)|$ is leaked by the number of for-loop iterations in line 2;
- $|(\pi_{x_2} \sigma_{x_1=a} R_3) \cap (\pi_{x_2} R_1)|$ is leaked by the number of for-loop iterations in line 3;
- $|(\pi_{x_3} \sigma_{x_2=b} R_2) \cap (\pi_{x_3} \pi_{x_1=a} R_2)|$ is leaked by the number of for-loop iterations in line 4;

To achieve obliviousness, a straightforward solution is to pad every intermediate result with dummy tuples to match the worst-case size N. However, this would result in N^3 memory accesses, which is even less efficient than the nested-loop-based algorithm in Section 3.

Algorithm 6 GenericJoin($\mathcal{Q} = (\mathcal{V}, \mathcal{E}), \mathcal{R}$) [44].

1 **if** $|\mathcal{V}| = 1$ **then return** $\cap_{e\in\mathcal{E}} R_e$ by Intersect;
2 $(I, J) \leftarrow$ an arbitrary partition of $\mathcal{V}$;
3 $\mathcal{Q}_I \leftarrow$ GenericJoin($(I, \mathcal{E}[I]), \{\pi_I R_e : e \in \mathcal{E}\}$);
4 **foreach** $t \in \mathcal{Q}_I$ **do** $\mathcal{Q}_t \leftarrow$ GenericJoin($(J, \mathcal{E}[J]), \{\pi_J(R_e \ltimes t) : e \in \mathcal{E}\}$);
5 **return** $\bigcup_{t\in\mathcal{Q}_I} \{t\} \times \mathcal{Q}_t$;

Inject Obliviousness to Algorithm 4. We transform Algorithm 4 into an oblivious version, presented as Algorithm 5, by employing oblivious primitives. The first modification merges the first two for-loops (lines 2–3 in Algorithm 4) into one step (line 1 in Algorithm 5). This is achieved by applying the semi-joins on R_3 using R_1, R_2 separately. Then, the third for-loop (line 4 in Algorithm 4) is replaced with a strategy based on the power of two choices. Specifically, for each surviving tuple $(a, b) \in R_3$, we first compute the size of two lists, $|\pi_{x_3}\sigma_{x_2=b}R_1|$ and $|\pi_{x_3}\sigma_{x_1=a}R_2|$, and put (a, b) into either K_1 or K_2, based on the relative order between $|\pi_{x_3}\sigma_{x_2=b}R_1|$ and $|\pi_{x_3}\sigma_{x_1=a}R_2|$. We next compute the following two-way joins $K_1 \bowtie R_1$ and $K_2 \bowtie R_2$ by invoking the RelaxedTwoWay primitive, each with the upper bound $N^{\frac{3}{2}}$ separately. Finally, we filter intermediate join results by the SemiJoin primitive and remove unnecessary dummy tuples by the Compact primitive.

Complexity of Algorithm 5. It suffices to show that $|K_1 \bowtie R_1| \le N^{\frac{3}{2}}$ and $|K_2 \bowtie R_2| \le N^{\frac{3}{2}}$, which directly follows from the query decomposition lemma [44]:

$$\sum_{(a,b)\in R_3} \min\left\{\left|\pi_{x_3}\sigma_{x_2=b}R_1\right|, \left|\pi_{x_3}\sigma_{x_1=a}R_2\right|\right\} \le \sum_{(a.b)\in R_3} |R_1 \ltimes (a,b)|^{\frac{1}{2}} \cdot |R_2 \ltimes (a,b)|^{\frac{1}{2}} \le N^{\frac{3}{2}}.$$

All other primitives incur $O(N \log N)$ time complexity and $O\left(\frac{N}{B} \cdot \log_{\frac{M}{B}} \frac{N}{B}\right)$ cache complexity. Hence, this algorithm incurs $O\left(N^{\frac{3}{2}} \cdot \log N\right)$ time complexity and $O\left(\frac{N^{\frac{3}{2}}}{B} \cdot \log_{\frac{M}{B}} \frac{N^{\frac{3}{2}}}{B}\right)$ cache complexity. As each step is oblivious, the composition of all these steps is also oblivious.

▶ **Theorem 8.** *For triangle join $\mathcal{Q}_\triangle$, there is an oblivious and cache-agnostic algorithm that can compute $\mathcal{Q}(\mathcal{R})$ for any instance $\mathcal{R}$ of input size N with $O\left(N^{\frac{3}{2}} \cdot \log N\right)$ time complexity and $O\left(\frac{N^{\frac{3}{2}}}{B} \cdot \log_{\frac{M}{B}} \frac{N^{\frac{3}{2}}}{B}\right)$ cache complexity under the tall cache and wide block assumptions.*

5 Oblivious Worst-case Optimal Join Algorithm

In this section, we start with revisiting the insecure WCOJ algorithm in Section 5.1 and then present our oblivious algorithm in Section 5.2 and present its analysis in Section 5.3. Subsequently, in Section 5.4, we explore the implications of our oblivious algorithm for relaxed oblivious algorithms designed for cyclic join queries.

5.1 Generic Join Revisited

In a join query $\mathcal{Q} = (\mathcal{V}, \mathcal{E})$, for a subset of attributes $S \subseteq \mathcal{V}$, we use $\mathcal{Q}[S] = (S, \mathcal{E}[S])$ to denote the sub-query induced by attributes in S, where $\mathcal{E}[S] = \{e \cap S : e \in \mathcal{E}\}$. For each attribute $x \in \mathcal{V}$, we use $\mathcal{E}_x = \{e \in \mathcal{E} : x \in e\}$ to denote the set of relations containing x. The insecure WCOJ algorithm described in [44] is outlined in Algorithm 6, which takes as input a

general join query $\mathcal{Q} = (\mathcal{V}, \mathcal{E})$ and an instance $\mathcal{R}$. In the base case, when only one attribute exists, it computes the intersection of all relations. For the general case, it partitions the attributes into two disjoint subsets, I and J, such that $I \cap J = \emptyset$ and $I \cup J = \mathcal{V}$. The algorithm first computes the sub-query $\mathcal{Q}[I]$, induced by attributes in I, whose join result is denoted $\mathcal{Q}_I$. Then, for each tuple $t \in \mathcal{Q}_I$, it recursively invokes the whole algorithm to compute the sub-query $\mathcal{Q}[J]$ induced by attributes in J, over tuples that can be joined with t. The resulting join result for each tuple t is denoted as $\mathcal{Q}_t$. Finally, it attaches each tuple in $\mathcal{Q}_t$ with t, representing the join results in which t participates. The algorithm ultimately returns the union of all join results for tuples in $\mathcal{Q}_I$. However, Algorithm 6 exhibits significant leakage of data statistics that violates the obliviousness constraint, for example:

- $\left|\bigcap_{e\in\mathcal{E}} R_e\right|$ is leaked in line 1;
- $|\pi_I R_e|$ for each relation $e \in \mathcal{E}$ is leaked in line 3;
- $|\mathcal{Q}_I|$, $|\pi_J (R_e \ltimes t)|$, and $|\mathcal{Q}_t|$ for each tuple $t \in \mathcal{Q}_I$ are leaked in line 4.

More importantly, this algorithm heavily relies on hashing indexes or range search indexes for retrieving tuples, such that the intersection at line 1 can be computed in $O\left(\min_{e\in\mathcal{E}} |R_e|\right)$ time. However, these indexes do not work well in the external memory model since naively extending this algorithm could result in $O\left(N^{\rho^*}\right)$ cache complexity, which is too expensive. Consequently, designing a WCOJ algorithm that simultaneously maintains cache locality and achieves obliviousness remains a significant challenge.

5.2 Our Algorithm

Now, we extend our oblivious triangle join algorithms from Section 4 to general join queries, as described in Algorithm 7. It is built on a recursive framework:

Base Case: $|\mathcal{V}| = 1$. In this case, the join degenerates to the set intersection of all input relations, which can be efficiently computed by the Intersect primitive.

General Case: $|\mathcal{V}| > 1$. In general, we partition $\mathcal{V}$ into two subsets I and J, but with the constraint that $|J| = 1$ or $|J| = 2$ but the two attributes y, z in J must satisfy $\mathcal{E}_y - \mathcal{E}_z \neq \emptyset$ and $\mathcal{E}_z - \mathcal{E}_y \neq \emptyset$. Similar to Algorithm 6, we compute the sub-query $\mathcal{Q}[I]$ by invoking the whole algorithm recursively, whose join result is denoted as $\mathcal{Q}_I$. To prevent the potential leakage, we must be careful about the projection of each relation involved in this subquery, which is computed by the Project primitive. We further distinguish two cases based on $|J|$:

General Case 1: $|J| = 1$. Suppose $J = \{x\}$. Recall that for each tuple $t \in \mathcal{Q}_I$, Algorithm 6 computes the intersection $\cap_{e\in\mathcal{E}_x} (R_e \ltimes t)$ on x in the base case. To ensure this step remains oblivious, we must conceal the size of $R_e \ltimes t$. To achieve this, we augment each tuple $t \in \mathcal{Q}_I$ with its *degree* in R_e, which is defined as $\Delta_e(t) = |R_e \ltimes t|$, using the Augment primitive. Then, we partition tuples in $\mathcal{Q}_I$ into $|\mathcal{E}_x|$ subsets based on their smallest *degree* across all relations in $\mathcal{E}_x$. The details are described in Algorithm 8. Let $\mathcal{Q}_I^e \subseteq \mathcal{Q}_I$ denote the set of tuples whose degree is the smallest in R_e, i.e., $e = \arg\min_{e'\in\mathcal{E}_x} \Delta_{e'}(t)$ for each $t \in \mathcal{Q}_I^e$. Whenever we write one tuple $t \in \mathcal{Q}_I$ to one subset, we also write a dummy tuple $\perp$ to the other $|\mathcal{E}_x| - 1$ subsets. At last, for each $e \in \mathcal{E}_x$, we compute $R_e \bowtie \mathcal{Q}_I^e$ by invoking the RelaxedTwoWay primitive (line 9), with upper bound N^{ρ^*}, and further filter them by remaining relations with semi-joins (line 10).

Algorithm 7 ObliviousGenericJoin$(\mathcal{Q} = (\mathcal{V}, \mathcal{E}), \mathcal{R})$.

1 **if** $|\mathcal{V}| = 1$ **then return** $\cap_{e\in\mathcal{E}} R_e$ by Intersect;
2 $(I, J) \leftarrow$ a partition of $\mathcal{V}$ such that (1) $|J| = 1$; or (2) $|J| = 2$ (say $J = \{y, z\}$) and $\mathcal{E}_y - \mathcal{E}_z \neq \emptyset$ and $\mathcal{E}_z - \mathcal{E}_y \neq \emptyset$;
3 **foreach** $e \in \mathcal{E}$ **do** $S_e \leftarrow$ Project$(R_e, e \cap I)$;
4 $\mathcal{Q}_I \leftarrow$ ObliviousGenericJoin$((I, \mathcal{E}[I]), \{S_e : e \in \mathcal{E}\})$;
5 **if** $|J| = 1$ **then** // Suppose $J = \{x\}$
6 **foreach** $e \in \mathcal{E}_x$ **do** $\mathcal{Q}_I \leftarrow$ Augment$(\mathcal{Q}_I, R_e, e \cap I)$;
7 $\{Q_I^e\}_{e\in\mathcal{E}_x} \leftarrow$ Partition-I$(\mathcal{Q}_I, \mathcal{E}_x)$;
8 **foreach** $e \in \mathcal{E}_x$ **do**
9 $L_e \leftarrow$ RelaxedTwoWay$\left(\mathcal{Q}_I^e, R_e, N^{\rho^*(\mathcal{Q})}\right)$;
10 **for** $e' \in \mathcal{E}_x - \{e\}$ **do** $L_e \leftarrow$ SemiJoin$(L_e, R_{e'})$;
11 $L \leftarrow \bigcup_{e\in\mathcal{E}_x} L_e$;
12 **else** // Suppose $J = \{y, z\}$
13 **foreach** $e \in \mathcal{E}_y \cup \mathcal{E}_z$ **do** $\mathcal{Q}_I \leftarrow$ Augment$(\mathcal{Q}_I, R_e, e \cap I)$;
14 $\{\mathcal{Q}_I^{e_1,e_2}\}_{(e_1,e_2)\in(\mathcal{E}_y-\mathcal{E}_z)\times(\mathcal{E}_z-\mathcal{E}_y)}, \{\mathcal{Q}_I^{e_3}\}_{e_3\in\mathcal{E}_x\cap\mathcal{E}_y} \leftarrow$ Partition-II$(Q_I, \mathcal{E}_y, \mathcal{E}_z)$;
15 **foreach** $(e_1, e_2) \in (\mathcal{E}_y - \mathcal{E}_z) \times (\mathcal{E}_z - \mathcal{E}_y)$ **do**
16 $L_{e_1,e_2} \leftarrow$ RelaxedTwoWay$\left(\mathcal{Q}_I^{e_1,e_2}, R_{e_1}, N^{\rho^*(\mathcal{Q})}\right)$;
17 $L_{e_1,e_2} \leftarrow$ RelaxedTwoWay$\left(L_{e_1,e_2}, R_{e_2}, N^{\rho^*(\mathcal{Q})}\right)$;
18 **foreach** $e \in \mathcal{E} - \{e_1, e_2\}$ **do** $L_{e_1,e_2} \leftarrow$ SemiJoin(L_{e_1,e_2}, R_e);
19 **foreach** $e_3 \in \mathcal{E}_y \cap \mathcal{E}_z$ **do**
20 $L_{e_3} \leftarrow$ RelaxedTwoWay$\left(\mathcal{Q}_I^{e_3}, R_{e_3}, N^{\rho^*(\mathcal{Q})}\right)$;
21 **foreach** $e \in \mathcal{E} - \{e_3\}$ **do** $L_{e_3} \leftarrow$ SemiJoin(L_{e_3}, R_e);
22 $L \leftarrow \left(\bigcup_{(e_1,e_2)\in(\mathcal{E}_y-\mathcal{E}_z)\times(\mathcal{E}_z-\mathcal{E}_y)} L_{e_1,e_2}\right) \cup \left(\bigcup_{e_3\in\mathcal{E}_y\cap\mathcal{E}_z} L_{e_3}\right)$;
23 **return** Compact L while keeping the first $N^{\rho^*(\mathcal{Q})}$ tuples;

General Case 2: $|J| = 2$. Suppose $J = \{y, z\}$. Consider an arbitrary tuple $t \in \mathcal{Q}_I$. Algorithm 6 computes the residual query $\left\{\bigcap_{e\in\mathcal{E}_y\cap\mathcal{E}_z}(R_e \ltimes t)\right\} \bowtie \left\{\bigcap_{e\in\mathcal{E}_y-\mathcal{E}_z}(R_e \ltimes t)\right\} \bowtie \left\{\bigcap_{e\in\mathcal{E}_z-\mathcal{E}_y}(R_e \ltimes t)\right\}$. Like the case above, we first compute its degree in R_e as $\Delta_e(t)$, by the Augment primitive. We then partition tuples in $\mathcal{Q}_I$ into $|\mathcal{E}_y \cap \mathcal{E}_z| + |\mathcal{E}_y - \mathcal{E}_z| \cdot |\mathcal{E}_z - \mathcal{E}_y|$ subsets based on their degrees, but more complicated than Case 1. The details are described in Algorithm 9. More specifically, for each $e_3 \in \mathcal{E}_y \cap \mathcal{E}_z$, let

$$\mathcal{Q}_I^{e_3} = \left\{t \in \mathcal{Q}_I : \Delta_{e_3}(t) = \min_{e''\in\mathcal{E}_y\cap\mathcal{E}_z} \Delta_{e''}(t), \Delta_{e_3}(t) < \min_{e\in\mathcal{E}_y-\mathcal{E}_z, e'\in\mathcal{E}_z-\mathcal{E}_y} \Delta_e(t) \cdot \Delta_{e'}(t)\right\};$$

and for each pair $(e_1, e_2) \in (\mathcal{E}_y - \mathcal{E}_z) \times (\mathcal{E}_z - \mathcal{E}_y)$, let

$$\mathcal{Q}_I^{e_1,e_2} = \left\{t \in \mathcal{Q}_I : \Delta_{e_1}(t) \cdot \Delta_{e_2}(t) = \min_{e\in\mathcal{E}_y-\mathcal{E}_z, e'\in\mathcal{E}_z-\mathcal{E}_y} \Delta_e(t) \cdot \Delta_{e'}(t) \leq \min_{e''\in\mathcal{E}_y\cap\mathcal{E}_z} \Delta_{e''}(t)\right\}$$

For each $(e_1, e_2) \in (\mathcal{E}_y - \mathcal{E}_z) \times (\mathcal{E}_z - \mathcal{E}_y)$, we compute $R_{e_1} \bowtie R_{e_2} \bowtie \mathcal{Q}_I^{e_1,e_2}$ by invoking the RelaxedTwoWay primitive iteratively (line 16-17), with the upper bound $N^{\rho^*(\mathcal{Q})}$, and filter these results by remaining relations with semi-joins (line 18). For each $e_3 \in \mathcal{E}_y \cap \mathcal{E}_z$, we compute $R_{e_3} \bowtie \mathcal{Q}_I^{e_3}$ by invoking the RelaxedTwoWay primitive (line 20), with the upper bound $N^{\rho^*(\mathcal{Q})}$, and filter these results by remaining relations with semi-joins (line 21).

Algorithm 8 PARTITION-I($\mathcal{Q}_I, \mathcal{E}_x$).

foreach $e \in \mathcal{E}_x$ **do** $\mathcal{Q}_I^e \leftarrow \emptyset$;
while read $(t, \{\Delta_e(t)\}_{e\in\mathcal{E}_x})$ from $\mathcal{Q}_I$ **do** // Suppose $\Delta_e(t) = |R_e \ltimes \{t\}|$
  $e' \leftarrow \arg\min_{e\in\mathcal{E}_x} \Delta_e(t)$;
  write t to $\mathcal{Q}_I^{e'}$ and **write** $\perp$ to $\mathcal{Q}_I^{e''}$ for each $e'' \in \mathcal{E}_x - \{e'\}$;
return $\{Q_I^e\}_{e\in\mathcal{E}_x}$;

Algorithm 9 PARTITION-II($\mathcal{Q}_I, \mathcal{E}_y, \mathcal{E}_z$).

foreach $(e_1, e_2) \in (\mathcal{E}_y - \mathcal{E}_z) \times (\mathcal{E}_z - \mathcal{E}_y)$ **do** $\mathcal{Q}_I^{e_1,e_2} \leftarrow \emptyset$;
foreach $e_3 \in \mathcal{E}_y \cap \mathcal{E}_z$ **do** $\mathcal{Q}_I^{e_3} \leftarrow \emptyset$;
while read $(t, \{\Delta_e(t)\}_{e\in\mathcal{E}_y\cup\mathcal{E}_z})$ from $\mathcal{Q}_I$ **do** // Suppose $\Delta_e(t) = |R_e \ltimes \{t\}|$
  $e_1, e_2, e_3 \leftarrow \arg \min_{e\in\mathcal{E}_y-\mathcal{E}_z} \Delta_e(t), \arg \min_{e\in\mathcal{E}_z-\mathcal{E}_y} \Delta_e(t), \arg \min_{e\in\mathcal{E}_y\cap\mathcal{E}_z} \Delta_e(t)$;
  if $\Delta_{e_1}(t) \cdot \Delta_{e_2}(t) \le \Delta_{e_3}(t)$ **then**
    write t to $\mathcal{Q}_I^{e_1,e_2}$;
    foreach $(e_1', e_2') \in (\mathcal{E}_y - \mathcal{E}_z) \times (\mathcal{E}_z - \mathcal{E}_y) - \{(e_1, e_2)\}$ **do write** $\perp$ to $\mathcal{Q}_I^{e_1',e_2'}$;
    foreach $e_3' \in \mathcal{E}_y \cap \mathcal{E}_z$ **do write** $\perp$ to $\mathcal{Q}_I^{e_3'}$;
  else
    write t to $\mathcal{Q}_I^{e_3}$;
    foreach $(e_1', e_2') \in (\mathcal{E}_y - \mathcal{E}_z) \times (\mathcal{E}_z - \mathcal{E}_y)$ **do write** $\perp$ to $\mathcal{Q}_I^{e_1',e_2'}$;
    foreach $e_3' \in \mathcal{E}_y \cap \mathcal{E}_z - \{e_3\}$ **do write** $\perp$ to $\mathcal{Q}_I^{e_3'}$;
return $\{\mathcal{Q}_I^{e_1,e_2}\}_{(e_1,e_2)\in(\mathcal{E}_y-\mathcal{E}_z)\times(\mathcal{E}_z-\mathcal{E}_y)}, \{\mathcal{Q}_I^{e_3}\}_{e_3\in\mathcal{E}_x\cap\mathcal{E}_y}$;

5.3 Analysis of Algorithm 7

Base Case: $|\mathcal{V}| = 1$. The obliviousness is guaranteed by the INTERSECT primitive. The cache complexity is $O\left(\frac{N}{B} \cdot \log_{\frac{M}{B}} \frac{N}{B}\right)$. In this case, $\rho^* = 1$. Hence, Theorem 9 holds.

General Case: $|\mathcal{V}| > 1$. By hypothesis, the recursive invocation of OBLIVIOUSGENERICJOIN at line 4 takes $O\left(N^{\rho^*(\mathcal{Q})} \cdot \log N\right)$ time and $O\left(\frac{N^{\rho^*}}{B} \cdot \log_{\frac{M}{B}} \frac{N}{B}\right)$ cache complexity, since $\rho^*((I, \mathcal{E}[I])) \le \rho^*(\mathcal{Q})$. We then show the correctness and complexity for all invocations of RELAXEDTWOWAY primitive. Let $\rho^*(\cdot)$ be an optimal fractional edge cover of $\mathcal{Q}$. The real size of the two-way join at line 9 can be first rewritten as:

$$\sum_{e\in\mathcal{E}_x} |R_e \ltimes \mathcal{Q}_I^e| = \sum_{e\in\mathcal{E}_x} \sum_{t\in\mathcal{Q}_I^e} |R_e \ltimes t| = \sum_{e\in\mathcal{E}_x} \sum_{t\in\mathcal{Q}_I^e} \min_{e'\in\mathcal{E}_x} |R_{e'} \ltimes t| \le \sum_{t\in\mathcal{Q}_I} \prod_{e'\in\mathcal{E}_x} |R_{e'} \ltimes t|^{\rho^*(e')} \le N^{\rho^*}$$

where the inequalities follow the facts that $\sum_{e'\in\mathcal{E}_x} \rho^*(e') \ge 1$, $\bigcup_{r\in\mathcal{E}_x} \mathcal{Q}_I^e = \mathcal{Q}_I$, and the query decomposition lemma [44]. Hence, $N^{\rho^*(\mathcal{Q})}$ is valid upper bound for $R_e \ltimes \mathcal{Q}_I^e$ for each $e \in \mathcal{E}_x$.

The real size of the two-way join at lines 18-19 and line 22 can be rewritten as

$$\begin{aligned}
&\sum_{e_1\in\mathcal{E}_y-\mathcal{E}_z, e_2\in\mathcal{E}_z-\mathcal{E}_y} |R_{e_1} \ltimes R_{e_2} \ltimes \mathcal{Q}_I^{e_1,e_2}| + \sum_{e_3\in\mathcal{E}_y\cap\mathcal{E}_z} |R_{e_3} \ltimes \mathcal{Q}_I^{e_3}| \\
&= \sum_{e_1\in\mathcal{E}_y-\mathcal{E}_z, e_2\in\mathcal{E}_z-\mathcal{E}_y} \sum_{t\in\mathcal{Q}_I^{e_1,e_2}} |(R_{e_1} \ltimes t) \ltimes (R_{e_2} \ltimes t)| + \sum_{e_3\in\mathcal{E}_y\cap\mathcal{E}_z} \sum_{t\in\mathcal{Q}_I^{e_3}} |R_{e_3} \ltimes t| \\
&= \sum_{t\in\mathcal{Q}_I} \min\left\{ \min_{e_1\in\mathcal{E}_y-\mathcal{E}_z, e_2\in\mathcal{E}_z-\mathcal{E}_y} |R_{e_1} \ltimes t| \cdot |R_{e_2} \ltimes t|, \min_{e_3\in\mathcal{E}_y\cap\mathcal{E}_z} |R_{e_3} \ltimes t| \right\} \qquad (2)
\end{aligned}$$

Let $\rho_1 = \sum_{e\in\mathcal{E}_y-\mathcal{E}_z} \rho^*(e)$, $\rho_2 = \sum_{e\in\mathcal{E}_z-\mathcal{E}_y} \rho^*(e)$ and $\rho_3 = \sum_{e\in\mathcal{E}_y\cap\mathcal{E}_z} \rho^*(e)$. Note $\rho_3 \geq 1-\min\{\rho_1,\rho_2\}$ as $\rho^*(\cdot)$ is a valid fractional edge cover for both y and z. For each tuple $t\in\mathcal{Q}_I$, we have

$$\begin{aligned}
&\min\left\{ \min_{e_1\in\mathcal{E}_y-\mathcal{E}_z, e_2\in\mathcal{E}_z-\mathcal{E}_y} |R_{e_1} \ltimes t| \cdot |R_{e_2} \ltimes t|, \min_{e_3\in\mathcal{E}_y\cap\mathcal{E}_z} |R_{e_3} \ltimes t| \right\} \\
&\leq \left(\min_{e_1\in\mathcal{E}_y-\mathcal{E}_z} |R_{e_1} \ltimes t|\right)^{\rho_1} \cdot \left(\min_{e_2\in\mathcal{E}_z-\mathcal{E}_y} |R_{e_2} \ltimes t|\right)^{\rho_2} \cdot \left(\min_{e_3\in\mathcal{E}_y\cap\mathcal{E}_z} |R_{e_3} \ltimes t|\right)^{\rho_3} \\
&\leq \prod_{e\in\mathcal{E}_y-\mathcal{E}_z} |R_e \ltimes t|^{\rho^*(e)} \cdot \prod_{e\in\mathcal{E}_z-\mathcal{E}_y} |R_e \ltimes t|^{\rho^*(e)} \cdot \prod_{e\in\mathcal{E}_y\cap\mathcal{E}_z} |R_e \ltimes t|^{\rho^*(e)} = \prod_{e\in\mathcal{E}_y\cup\mathcal{E}_z} |R_e \ltimes t|^{\rho^*(e)},
\end{aligned}$$

where the first inequality follows from $\min\{a,b\} \leq a^p \cdot b^{1-p}$ for $a,b\geq 0$ and $p\in[0,1]$, and the third inequality follows from $\rho_1,\rho_2 \geq \min\{\rho_1,\rho_2\}$. Now, we can further bound (2) as

$$(2) \leq \sum_{t\in\mathcal{Q}_I} \prod_{e\in\mathcal{E}_y\cup\mathcal{E}_z} |R_e \ltimes t|^{\rho^*(e)} = \sum_{t\in\mathcal{Q}_I} \prod_{e\in\mathcal{E}_y\cup\mathcal{E}_z} |R_e \ltimes t|^{\rho^*(e)} \leq \prod_{e\in\mathcal{E}} |R_e|^{\rho^*(e)} \leq N^{\rho^*}$$

where the second last inequality follows the query decomposition lemma [44].

▶ **Theorem 9.** *For a general join query $\mathcal{Q}$, there is an oblivious and cache-agnostic algorithm that can compute $\mathcal{Q}(\mathcal{R})$ for any instance $\mathcal{R}$ of input size N with $O\left(N^{\rho^*}\cdot\log N\right)$ time complexity and $O\left(\frac{N^{\rho^*}}{B}\cdot\log_{\frac{M}{B}}\frac{N^{\rho^*}}{B}\right)$ cache complexity under the tall cache and wide block assumptions, where ρ^* is the optimal fractional edge cover number of $\mathcal{Q}$.*

5.4 Implications to Relaxed Oblivious Algorithms

Our oblivious WCOJ algorithm can be combined with the generalized hypertree decomposition framework [33] to develop a relaxed oblivious algorithm for general join queries.

▶ **Definition 10** (Generalized Hypertree Decomposition (GHD))**.** *Given a join query $\mathcal{Q}=(\mathcal{V},\mathcal{E})$, a GHD of $\mathcal{Q}$ is a pair $(\mathcal{T},\lambda)$, where $\mathcal{T}$ is a tree as an ordered set of nodes and $\lambda:\mathcal{T}\to 2^{\mathcal{V}}$ is a labeling function which associates to each vertex $u\in\mathcal{T}$ a subset of attributes in $\mathcal{V}$, λ_u, such that (1) for each $e\in\mathcal{E}$, there is a node $u\in\mathcal{T}$ such that $e\subseteq\lambda_u$; (2) For each $x\in\mathcal{V}$, the set of nodes $\{u\in\mathcal{T}: x\in\lambda_u\}$ forms a connected subtree of $\mathcal{T}$. The fractional hypertree width of $\mathcal{Q}$ is defined as $\min_{(\mathcal{T},\lambda)}\max_{u\in\mathcal{T}}\rho^*\left((\lambda_u,\{e\cap\lambda : e\in\mathcal{E}\})\right)$.*

The pseudocode of our algorithm is given in the full version [1]. Suppose we take as input a join query $\mathcal{Q}=(\mathcal{V},\mathcal{E})$, an instance $\mathcal{R}$, and an upper bound on the output size $\tau\geq|\mathcal{Q}(\mathcal{R})|$. Let $(\mathcal{T},\lambda)$ be an arbitrary GHD of $\mathcal{Q}$. We first invoke Algorithm 7 to compute the subquery $\mathcal{Q}_u=(\lambda_u,\mathcal{E}_u)$ defined by each node $u\in\mathcal{T}$, where $\mathcal{E}_u=\{e\cap u : e\in\mathcal{E}\}$, and materialize its join result as one relation. We then apply the classic Yannakakis algorithm [54] on the materialized relations by invoking the SemiJoin primitive for semi-joins and the

RelaxedTwoWay primitive for pairwise joins. After removing dangling tuples, the size of each two-way join is upper bound by the size of the final join results and, therefore, τ. This leads to a relaxed oblivious algorithm whose access pattern only depends on N and τ.

▶ **Theorem 11.** *For a join query $\mathcal{Q}$, an instance $\mathcal{R}$ of input size N, and parameter $\tau \geq |\mathcal{Q}(\mathcal{R})|$, there is a cache-agnostic algorithm that can compute $\mathcal{Q}(\mathcal{R})$ with $O\left((N^w + \tau) \cdot \log(N^w + \tau)\right)$ time complexity and $O\left(\frac{N^w+\tau}{B} \cdot \log_{\frac{M}{B}} \frac{N^w+\tau}{B}\right)$ cache complexity, whose access pattern only depends on N and τ, where w is the fractional hypertree width of $\mathcal{Q}$.*

6 Conclusion

This paper has introduced a general framework for oblivious multi-way join processing, achieving near-optimal time and cache complexity. However, several intriguing questions remain open for future exploration:

- *Balancing Privacy and Efficiency.* Recent research has investigated improved trade-offs between privacy and efficiency, aiming to overcome the challenges of worst-case scenarios, such as differentially oblivious algorithms [15].
- *Emit model for EM algorithms.* In the context of EM join algorithms, the *emit* model - where join results are directly outputted without writing back to disk - has been considered. It remains open whether oblivious, worst-case optimal join algorithms can be developed without requiring all join results to be written back to disk.
- *Communication-oblivious join algorithm for MPC model.* A natural connection exists between the MPC and EM models in join processing. While recent work has explored communication-oblivious algorithms in the MPC model [14, 49], extending these ideas to multi-way join processing remains an open challenge.

References

1 https://arxiv.org/pdf/2501.04216.
2 Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases*, volume 8. Addison-Wesley Reading, 1995.
3 Mahmoud Abo Khamis, Hung Q Ngo, and Atri Rudra. Faq: questions asked frequently. In *PODS*, pages 13–28, 2016. doi:10.1145/2902251.2902280.
4 Alok Aggarwal and S Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988. doi:10.1145/48529.48535.
5 Miklós Ajtai, János Komlós, and Endre Szemerédi. An 0 (n log n) sorting network. In *STOC*, pages 1–9, 1983.
6 Arvind Arasu and Raghav Kaushik. Oblivious query processing. *ICDT*, 2013.
7 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.
8 Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. Optorama: Optimal oblivious ram. In *Eurocrypt*, pages 403–432. Springer, 2020. doi:10.1007/978-3-030-45724-2_14.
9 Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. In *FOCS*, pages 739–748. IEEE, 2008. doi:10.1109/FOCS.2008.43.
10 Kenneth E Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314, 1968. doi:10.1145/1468075.1468121.
11 Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. *JACM*, 64(6):1–58, 2017. doi:10.1145/3125644.

12 C. Beeri, R. Fagin, D. Maier, and M. Yannakakis. On the desirability of acyclic database schemes. *JACM*, 30(3):479–513, 1983. doi:10.1145/2402.322389.

13 Amos Beimel, Kobbi Nissim, and Mohammad Zaheri. Exploring differential obliviousness. In *APPROX/RANDOM*, 2019.

14 TH Chan, Kai-Min Chung, Wei-Kai Lin, and Elaine Shi. Mpc for mpc: secure computation on a massively parallel computing architecture. In *ITCS*, 2020.

15 TH Hubert Chan, Kai-Min Chung, Bruce M Maggs, and Elaine Shi. Foundations of differentially oblivious algorithms. In *SODA*, pages 2448–2467. SIAM, 2019.

16 Zhao Chang, Dong Xie, and Feifei Li. Oblivious ram: A dissection and experimental evaluation. *Proc. VLDB Endow.*, 9(12):1113–1124, 2016. doi:10.14778/2994509.2994528.

17 Zhao Chang, Dong Xie, Sheng Wang, and Feifei Li. Towards practical oblivious join. In *SIGMOD*, 2022.

18 Shumo Chu, Danyang Zhuo, Elaine Shi, and T-H. Hubert Chan. Differentially Oblivious Database Joins: Overcoming the Worst-Case Curse of Fully Oblivious Algorithms. In *ITC*, volume 199, pages 19:1–19:24, 2021. doi:10.4230/LIPICS.ITC.2021.19.

19 Victor Costan and Srinivas Devadas. Intel sgx explained. *Cryptology ePrint Archive*, 2016.

20 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, 2018. URL: https://www.usenix.org/conference/osdi18/presentation/crooks.

21 Erik D Demaine. Cache-oblivious algorithms and data structures. *Lecture Notes from the EEF Summer School on Massive Data Sets*, 8(4):1–249, 2002.

22 Shiyuan Deng and Yufei Tao. Subgraph enumeration in optimal i/o complexity. In *ICDT*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024.

23 Srinivas Devadas, Marten van Dijk, Christopher W Fletcher, Ling Ren, Elaine Shi, and Daniel Wichs. Onion oram: A constant bandwidth blowup oblivious ram. In *TCC*, pages 145–174. Springer, 2016. doi:10.1007/978-3-662-49099-0_6.

24 Saba Eskandarian and Matei Zaharia. Oblidb: Oblivious query processing for secure databases. *Proc. VLDB Endow.*, 13(2), 2019. doi:10.14778/3364324.3364331.

25 R. Fagin. Degrees of acyclicity for hypergraphs and relational database schemes. *JACM*, 30(3):514–550, 1983. doi:10.1145/2402.322390.

26 Austen Z Fan, Paraschos Koutris, and Hangdong Zhao. Tight bounds of circuits for sum-product queries. *SIGMOD*, 2(2):1–20, 2024.

27 Jörg Flum, Markus Frick, and Martin Grohe. Query evaluation via tree-decompositions. *JACM*, 49(6):716–752, 2002. doi:10.1145/602220.602222.

28 Matteo Frigo, Charles E Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *FOCS*, pages 285–297. IEEE, 1999. doi:10.1109/SFFCS.1999.814600.

29 Craig Gentry, Kenny A Goldman, Shai Halevi, Charanjit Julta, Mariana Raykova, and Daniel Wichs. Optimizing oram and using it efficiently for secure computation. In *PETs*, pages 1–18. Springer, 2013. doi:10.1007/978-3-642-39077-7_1.

30 Oded Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *STOC*, pages 182–194, 1987. doi:10.1145/28395.28416.

31 Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *JACM*, 43(3):431–473, 1996. doi:10.1145/233551.233553.

32 Michael T Goodrich. Data-oblivious external-memory algorithms for the compaction, selection, and sorting of outsourced data. In *SPAA*, pages 379–388, 2011. doi:10.1145/1989493.1989555.

33 Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions and tractable queries. *JCSS*, 64(3):579–627, 2002. doi:10.1006/JCSS.2001.1809.

34 Hakan Hacigümüş, Bala Iyer, Chen Li, and Sharad Mehrotra. Executing sql over encrypted data in the database-service-provider model. In *SIGMOD*, pages 216–227, 2002.

35 Bingsheng He and Qiong Luo. Cache-oblivious nested-loop joins. In *CIKM*, pages 718–727, 2006. doi:10.1145/1183614.1183717.

36 Xiao Hu. Cover or pack: New upper and lower bounds for massively parallel joins. In *PODS*, pages 181–198, 2021. doi:10.1145/3452021.3458319.

37 Xiaocheng Hu, Miao Qiao, and Yufei Tao. I/o-efficient join dependency testing, loomis–whitney join, and triangle enumeration. *JCSS*, 82(8):1300–1315, 2016. doi:10.1016/J.JCSS.2016.05.005.

38 Bas Ketsman and Dan Suciu. A worst-case optimal multi-round algorithm for parallel computation of conjunctive queries. In *PODS*, pages 417–428, 2017. doi:10.1145/3034786.3034788.

39 Paraschos Koutris, Paul Beame, and Dan Suciu. Worst-case optimal algorithms for parallel query processing. In *ICDT*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016.

40 Simeon Krastnikov, Florian Kerschbaum, and Douglas Stebila. Efficient oblivious database joins. *VLDB*, 13(12):2132–2145, 2020. URL: http://www.vldb.org/pvldb/vol13/p2132-krastnikov.pdf.

41 Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in) security of hash-based oblivious ram and a new balancing scheme. In *SODA*, pages 143–156. SIAM, 2012. doi:10.1137/1.9781611973099.13.

42 Wei-Kai Lin, Elaine Shi, and Tiancheng Xie. Can we overcome the n log n barrier for oblivious sorting? In *SODA*, pages 2419–2438. SIAM, 2019. doi:10.1137/1.9781611975482.148.

43 Hung Q Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms. *JACM*, 65(3):1–40, 2018. doi:10.1145/3180143.

44 Hung Q Ngo, Christopher Ré, and Atri Rudra. Skew strikes back: New developments in the theory of join algorithms. *ACM SIGMOD Record*, 42(4):5–16, 2014. doi:10.1145/2590989.2590991.

45 Vijaya Ramachandran and Elaine Shi. Data oblivious algorithms for multicores. In *SPAA*, pages 373–384, 2021. doi:10.1145/3409964.3461783.

46 Sajin Sasy, Aaron Johnson, and Ian Goldberg. Fast fully oblivious compaction and shuffling. In *CCS*, pages 2565–2579, 2022. doi:10.1145/3548606.3560603.

47 Elaine Shi. Path oblivious heap: Optimal and practical oblivious priority queue. In *SP*, pages 842–858. IEEE, 2020. doi:10.1109/SP40000.2020.00037.

48 Emil Stefanov, Marten Van Dijk, Elaine Shi, T-H Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: an extremely simple oblivious ram protocol. *JACM*, 65(4):1–26, 2018. doi:10.1145/3177872.

49 Yufei Tao, Ru Wang, and Shiyuan Deng. Parallel communication obliviousness: One round and beyond. *Proceedings of the ACM on Management of Data*, 2(5):1–24, 2024. doi:10.1145/3695832.

50 Todd L Veldhuizen. Leapfrog triejoin: A simple, worst-case optimal join algorithm. In *ICDT*, 2014.

51 Jeffrey Scott Vitter. External memory algorithms and data structures: Dealing with massive data. *CsUR*, 33(2):209–271, 2001.

52 Xiao Wang, Hubert Chan, and Elaine Shi. Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In *CCS*, pages 850–861, 2015. doi:10.1145/2810103.2813634.

53 Yilei Wang and Ke Yi. Query evaluation by circuits. In *PODS*, 2022.

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

55 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 17*, pages 283–298, 2017. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/zheng.