

# 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 when encryption 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 the 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.

## 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.

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



© Author: Please provide a copyright holder;  
licensed under Creative Commons License CC-BY 4.0  
28th International Conference on Database Theory (ICDT 2025).



Leibniz International Proceedings in Informatics  
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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, 7], 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 [15]. 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  $\mathcal{Q} = (\mathcal{V}, \mathcal{E})$  [1], where  $\mathcal{V}$  models the set of attributes, and  $\mathcal{E} \subseteq 2^{\mathcal{V}}$  models the set of relations. Let  $\text{dom}(x)$  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  $\text{dom}(x)$  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.,

$$\mathcal{Q}(\mathcal{R}) = \left\{ t \in \prod_{x \in \mathcal{V}} \text{dom}(x) \mid \forall e \in \mathcal{E}, \exists t_e \in R_e, \pi_e t = t_e \right\}.$$

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  $\text{OUT} = |\mathcal{Q}(\mathcal{R})|$  be the *output size* of the join query  $\mathcal{Q}$  over instance  $\mathcal{R}$ . We study the data complexity [1] 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 [3, 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 [3, 51, 6, 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  $\text{Access}_{\mathcal{A}}(\mathcal{Q}, \mathcal{R})$  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, 14]). *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  $N$ ,  $\text{Access}_{\mathcal{A}}(\mathcal{Q}, \mathcal{R}) \stackrel{\delta}{\equiv} \text{Access}_{\mathcal{A}}(\mathcal{Q}, \mathcal{R}')$ , where  $\delta$  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  $\stackrel{\delta}{\equiv}$  indicates the statistical distance between two distributions is at most  $\delta$ .*

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 statistical distance between two probability distributions is typically quantified using standard

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

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. [7] 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<sup>2</sup>, where  $\rho^*$  is the fractional edge cover number of the join query (formally defined in Section 2.1). The optimality is implied by the AGM bound [8].<sup>3</sup> 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  $O((N^{\text{subw}} + \text{OUT}) \cdot \text{polylog} N)$  time [54, 2], where  $\text{subw}$  is the submodular-width of the join query. For example,  $\text{subw} = 1$  if and only if the join query is acyclic [11, 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  $O\left(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B} + \frac{\text{OUT}}{B}\right)$ . For multi-way join queries, an EM algorithm with cache complexity  $O\left(\frac{N^{\rho^*}}{M^{\rho^*-1} B} \cdot \log_{\frac{M}{B}} \frac{N}{B} + \frac{\text{OUT}}{B}\right)$  has been achieved for Berge-acyclic joins [37],  $\alpha$ -acyclic joins [36, 39], graph joins [38, 22] and Loomis-Whitney joins [39].<sup>4</sup> These results were previously stated without including the output-dependent term  $\frac{\text{OUT}}{B}$  since they do not

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

<sup>3</sup> 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.

<sup>4</sup> Some of these algorithms have been developed for the *Massively Parallel Computation* (MPC) model [10] and can be adapted to the EM model through the MPC-to-EM reduction [39].

	Previous Results	New Results
RAM model	$O(N^{\rho^*} \cdot \log N)$ [44, 7] (one-way function assumption)	$O(N^{\rho^*} \cdot \log N)$ (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)

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

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  $O(N^{\rho^*} \cdot \text{polylog} N)$  time complexity. By integrating the insecure WCOJ algorithm [44] with the optimal ORAM [7], 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  $O(N^{\rho^*} \cdot \log^2 N)$  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  $O(N^{\rho^*} \cdot \log N)$  or  $O(N^{\rho^*} \cdot \log N \cdot \log_B N)$ , 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  $O((N^{\text{subw}} + \text{OUT}) \cdot \text{polylog} N)$  time complexity. It is noted that relaxed oblivious algorithm with the same time complexity  $O((N + \text{OUT}) \cdot \log N)$  have been proposed without requiring ORAM [5, 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 [14, 12, 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  $O(N^{\min\{\rho^*+1, \rho\}} \cdot \log N)$  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  $\rho^*$  and  $\rho$  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  $O(N^{\rho^*} \cdot \log N)$  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  $O((N^{\text{fhtw}} + \text{OUT}) \cdot \log N)$  time complexity and  $O\left(\frac{N^{\text{fhtw}} + \text{OUT}}{B} \cdot \log \frac{M}{B} \frac{N^{\text{fhtw}} + \text{OUT}}{B}\right)$  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  $W : \mathcal{E} \rightarrow [0, 1]$  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:

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

The optimal value of (1) is the *fractional edge cover number* of  $\mathcal{Q}$ , denoted as  $\rho^*(\mathcal{Q})$ . Similarly, a function  $W : \mathcal{E} \rightarrow \{0, 1\}$  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 [4, 9].** Given an input array of  $N$  elements, the goal is to output an array according to some predetermined ordering. The classical bitonic sorting network [9] requires  $O(N \cdot \log^2 N)$  time. Later, this time complexity has been improved to  $O(N \cdot \log N)$  [4] 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  $O(\frac{N}{B} \log \frac{M}{B} \frac{N}{B})$  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  $O(\frac{N}{B})$  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  $O(\frac{N}{B} \cdot \log \frac{M}{B} \frac{N}{B})$  cache complexity. Further details are provided in Appendix B.

**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  $S_1, S_2, \dots, S_k$  (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  $O(\frac{|R| \cdot |S|}{B})$  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 [5, 40] takes as input two relations  $R, S$  and a parameter  $\tau \geq |R \bowtie S|$ , and output a table of  $\tau$  elements containing join results of  $R \bowtie S$ , whose access pattern only depends on  $|R|, |S|$  and  $\tau$ . 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 Appendix C, we show how to improve this algorithm with better cache complexity 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  $\tau$ .*

### 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  $W : \mathcal{E} \rightarrow \{0, 1\}$ , 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  $O(N^{\rho^*})$ . In this way, the number of intermediate results materialized by line 7 is at most  $O(N^{\rho^*+1})$ . 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  $O(N^{\min\{\rho, \rho^*+1\}})$  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  $\rho^*$  and  $\rho$  are the optimal fractional and integral edge cover number of  $\mathcal{Q}$ , respectively.*

It is important to note that any oblivious join algorithm incurs a cache complexity of  $O\left(\frac{N^{\rho^*}}{B}\right)$ , 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** ( $\alpha$ -acyclic Join). A join query  $\mathcal{Q}$  is  $\alpha$ -acyclic [11, 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  $\alpha$ -acyclic join admits an optimal fractional edge cover that is integral [36].



---

**Algorithm 1** OBLIVIOUSNESTEDLOOPJOIN( $\mathcal{Q}, \mathcal{R}$ )
 

---

```

1  $W^* \leftarrow$  an optimal integral edge cover of  $\mathcal{Q}$ ,  $L \leftarrow \emptyset$ ;
2  $\mathcal{E}' \leftarrow \{e \in \mathcal{E} : W^*(e) = 1\}$ ;
3 while  $\mathcal{E}' \neq \emptyset$  do
4    $e \leftarrow$  an arbitrary relation in  $\mathcal{E}'$ ;
5    $\mathcal{E}' \leftarrow \mathcal{E}' - \{e\}$ ;
6   if  $L = \emptyset$  then  $L \leftarrow R_e$ ;
7   else  $L \leftarrow$  NESTEDLOOP( $L, R_e$ );
8   foreach  $e' \in \mathcal{E} - \{e\}$  do  $L \leftarrow$  SEMIJOIN( $L, R_{e'}$ );
9 return  $L$ ;
```

---

► **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  $\{R_1, R_3, \dots, R_{k-1}\}$  and  $\{R_2, R_4, \dots, R_k\}$ , 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  $\mathcal{Q} = R_1(x_1, y_1) \bowtie R_2(x_2, y_2) \bowtie \cdots \bowtie R_k(x_k, y_k) \bowtie R_{k+1}(x_1, x_2, \dots, x_k) \bowtie R_{k+2}(y_1, y_2, \dots, y_k)$ . 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  $\mathcal{Q}_\Delta = R_1(x_2, x_3) \bowtie R_2(x_1, x_3) \bowtie R_3(x_1, x_2)$ , 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  $a \in \text{dom}(x_1)$  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  $a \in \text{dom}(x_1)$  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:

- $|(\pi_{x_1} R_2) \cap (\pi_{x_1} R_3)|$  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.

## XX:10 Optimal Oblivious Algorithms for Multi-way Joins

■ **Algorithm 2** Compute  $\mathcal{Q}_\Delta$  by power of two choices

---

```

1  $L \leftarrow \emptyset$ ;
2 foreach  $a \in (\pi_{x_1}R_2) \cap (\pi_{x_1}R_3)$  do
3   if  $|\sigma_{x_1=a}R_2| \cdot |\sigma_{x_1=a}R_3| \leq |R_1|$  then
4     foreach  $(b, c) \in (\pi_{x_2}\sigma_{x_1=a}R_3) \times (\pi_{x_3}\sigma_{x_1=a}R_2)$  do
5       if  $(b, c) \in R_1$  then write  $(a, b, c)$  to  $L$ ;
6   else
7     foreach  $(b, c) \in R_1$  do
8       if  $(a, b) \in R_3$  and  $(a, c) \in R_2$  then write  $(a, b, c)$  to  $L$ ;
9 return  $L$ ;
```

---

■ **Algorithm 3** Inject Obliviousness to Algorithm 2

---

```

1  $A \leftarrow (\pi_{x_1}R_2) \cap (\pi_{x_1}R_3)$  by PROJECT and INTERSECT;
2  $A \leftarrow$  AUGMENT( $A, \{R_2, R_3\}, x_1$ );
3  $A_1, A_2 \leftarrow \emptyset$ ;
4 while read  $(a, \Delta_1, \Delta_2)$  from  $A$  do //  $\Delta_1 = |\pi_{x_3}\sigma_{x_1=a}R_2|$  and  $\Delta_2 = |\pi_{x_2}\sigma_{x_1=a}R_3|$ 
5   if  $\Delta_1 \cdot \Delta_2 \leq |R_1|$  then write  $a$  to  $A_1$ , write  $\perp$  to  $A_2$ ;
6   else write  $a$  to  $A_2$ , write  $\perp$  to  $A_1$ ;
7  $L_1 \leftarrow$  RELAXEDTOWWAY( $A_2, R_1, N^{\frac{3}{2}}$ );
8  $L_1 \leftarrow$  SEMIJOIN( $L_1, R_2$ ),  $L_1 \leftarrow$  SEMIJOIN( $L_1, R_3$ );
9  $R_2 \leftarrow$  SEMIJOIN( $R_2, A_1$ ),  $R_3 \leftarrow$  SEMIJOIN( $R_3, A_1$ );
10  $L_2 \leftarrow$  RELAXEDTOWWAY( $R_2, R_3, N^{\frac{3}{2}}$ );
11  $L_2 \leftarrow$  SEMIJOIN( $L_2, R_1$ );
12 return COMPACT  $L_1 \cup L_2$  while keeping the first  $N^{\frac{3}{2}}$  tuples;
```

---

**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  $(R_2 \times A_1) \bowtie (R_3 \times A_2)$  by invoking the RELAXEDTOWWAY 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.

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

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

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

■ **Algorithm 4** Compute  $\mathcal{Q}_\Delta$  by delaying computation

---

```

1  $L \leftarrow \emptyset$ ;
2 foreach  $a \in (\pi_{x_1} R_2) \cap (\pi_{x_1} R_3)$  do
3   foreach  $b \in (\pi_{x_2} \sigma_{x_1=a} R_3) \cap (\pi_{x_2} R_1)$  do
4     foreach  $c \in (\pi_{x_3} \sigma_{x_2=b} R_1) \cap (\pi_{x_3} \sigma_{x_1=a} R_2)$  do
5       write  $(a, b, c)$  to  $L$ ;
6 return  $L$ ;
```

---

■ **Algorithm 5** Inject Obliviousness to Algorithm 4

---

```

1  $R_3 \leftarrow \text{AUGMENT}(R_3, R_1, x_2)$ ,  $R_3 \leftarrow \text{AUGMENT}(R_3, R_2, x_1)$ ;
2  $K_1, K_2 \leftarrow \emptyset$ ;
3 while read  $(t, \Delta_1, \Delta_2)$  from  $R_3$  do // Suppose  $\Delta_i = |R_i \times \{t\}|$ 
4   if  $\Delta_1 \leq \Delta_2$  then write  $t$  to  $K_1$ , write  $\perp$  to  $K_2$ ;
5   else write  $t$  to  $K_2$ , write  $\perp$  to  $K_1$ ;
6  $L_1 \leftarrow \text{RELAXEDTOWWAY}(K_1, R_1, N^{\frac{3}{2}})$ ,  $L_1 \leftarrow \text{SEMIJOIN}(L_1, R_2)$ ;
7  $L_2 \leftarrow \text{RELAXEDTOWWAY}(K_2, R_2, N^{\frac{3}{2}})$ ,  $L_2 \leftarrow \text{SEMIJOIN}(L_2, R_1)$ ;
8 return  $\text{COMPACT } L_1 \cup L_2$  while keeping the first  $N^{\frac{3}{2}}$  tuples;
```

---

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;
- $|(\pi_{x_3} \sigma_{x_2=b} R_1) \cap (\pi_{x_3} \sigma_{x_1=a} R_2)|$  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.

**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 RELAXEDTOWWAY 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  $|K_1 \bowtie R_1| \leq N^{\frac{3}{2}}$  and  $|K_2 \bowtie R_2| \leq$

## XX:12 Optimal Oblivious Algorithms for Multi-way Joins

■ **Algorithm 6** GENERICJOIN( $\mathcal{Q} = (\mathcal{V}, \mathcal{E}), \mathcal{R}$ ) [44]

---

```

1 if  $|\mathcal{V}| = 1$  then return  $\bigcap_{e \in \mathcal{E}} R_e$  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( $(J, \mathcal{E}[J]), \{\pi_J(R_e \times t) : e \in \mathcal{E}\}$ );
5 return  $\bigcup_{t \in \mathcal{Q}_I} \{t\} \times \mathcal{Q}_t$ ;
```

---

$N^{\frac{3}{2}}$ , which directly follows from the query decomposition lemma [44]:

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

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  $\mathcal{Q}_\Delta$ , 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:

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

---

**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      $L_e \leftarrow$  RELAXEDTOWWAY( $Q_I^e, R_e, N^{\rho^*}(\mathcal{Q})$ );
10    for  $e' \in \mathcal{E}_x - \{e\}$  do  $L_e \leftarrow$  SEMIJOIN( $L_e, R_{e'}$ );
11   $L \leftarrow \cup_{e \in \mathcal{E}_x} L_e$ ;
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    $\{Q_I^{e_1, e_2}\}_{(e_1, e_2) \in (\mathcal{E}_y - \mathcal{E}_z) \times (\mathcal{E}_z - \mathcal{E}_y)}, \{Q_I^{e_3}\}_{e_3 \in \mathcal{E}_x \cap \mathcal{E}_y} \leftarrow$  PARTITION-II( $\mathcal{Q}_I, \mathcal{E}_y, \mathcal{E}_z$ );
15   foreach  $(e_1, e_2) \in (\mathcal{E}_y - \mathcal{E}_z) \times (\mathcal{E}_z - \mathcal{E}_y)$  do
16      $L_{e_1, e_2} \leftarrow$  RELAXEDTOWWAY( $Q_I^{e_1, e_2}, R_{e_1}, N^{\rho^*}(\mathcal{Q})$ );
17      $L_{e_1, e_2} \leftarrow$  RELAXEDTOWWAY( $L_{e_1, e_2}, R_{e_2}, N^{\rho^*}(\mathcal{Q})$ );
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      $L_{e_3} \leftarrow$  RELAXEDTOWWAY( $Q_I^{e_3}, R_{e_3}, N^{\rho^*}(\mathcal{Q})$ );
21     foreach  $e \in \mathcal{E} - \{e_3\}$  do  $L_{e_3} \leftarrow$  SEMIJOIN( $L_{e_3}, R_e$ );
22    $L \leftarrow \left( \cup_{(e_1, e_2) \in (\mathcal{E}_y - \mathcal{E}_z) \times (\mathcal{E}_z - \mathcal{E}_y)} L_{e_1, e_2} \right) \cup \left( \cup_{e_3 \in \mathcal{E}_y \cap \mathcal{E}_z} L_{e_3} \right)$ ;
23 return COMPACT  $L$  while keeping the first  $N^{\rho^*}(\mathcal{Q})$  tuples;

```

---

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  $O(\min_{e \in \mathcal{E}} |R_e|)$  time. However, these indexes do not work well in the external memory model since naively extending this algorithm could result in  $O(N^{\rho^*})$  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  $|J|$ :

## XX:14 Optimal Oblivious Algorithms for Multi-way Joins

### ■ Algorithm 8 PARTITION-I( $Q_I, \mathcal{E}_x$ )

---

```

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

```

---

### ■ Algorithm 9 PARTITION-II( $Q_I, \mathcal{E}_y, \mathcal{E}_z$ )

---

```

1 foreach  $(e_1, e_2) \in (\mathcal{E}_y - \mathcal{E}_z) \times (\mathcal{E}_z - \mathcal{E}_y)$  do  $Q_I^{e_1, e_2} \leftarrow \emptyset$ ;
2 foreach  $e_3 \in \mathcal{E}_y \cap \mathcal{E}_z$  do  $Q_I^{e_3} \leftarrow \emptyset$ ;
3 while read  $(t, \{\Delta_e(t)\}_{e \in \mathcal{E}_y \cup \mathcal{E}_z})$  from  $Q_I$  do // Suppose  $\Delta_e(t) = |R_e \times \{t\}|$ 
4    $e_1, e_2, e_3 \leftarrow \arg \min_{e \in \mathcal{E}_y - \mathcal{E}_z} \Delta_e(t), \arg \min_{e \in \mathcal{E}_z - \mathcal{E}_y} \Delta_e(t), \arg \min_{e \in \mathcal{E}_y \cap \mathcal{E}_z} \Delta_e(t)$ ;
5   if  $\Delta_{e_1}(t) \cdot \Delta_{e_2}(t) \leq \Delta_{e_3}(t)$  then
6     write  $t$  to  $Q_I^{e_1, e_2}$ ;
7     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  $Q_I^{e'_1, e'_2}$ ;
8     foreach  $e'_3 \in \mathcal{E}_y \cap \mathcal{E}_z$  do write  $\perp$  to  $Q_I^{e'_3}$ ;
9   else
10    write  $t$  to  $Q_I^{e_3}$ ;
11    foreach  $(e'_1, e'_2) \in (\mathcal{E}_y - \mathcal{E}_z) \times (\mathcal{E}_z - \mathcal{E}_y)$  do write  $\perp$  to  $Q_I^{e'_1, e'_2}$ ;
12    foreach  $e'_3 \in \mathcal{E}_y \cap \mathcal{E}_z - \{e_3\}$  do write  $\perp$  to  $Q_I^{e'_3}$ ;
13 return  $\{Q_I^{e_1, e_2}\}_{(e_1, e_2) \in (\mathcal{E}_y - \mathcal{E}_z) \times (\mathcal{E}_z - \mathcal{E}_y)}, \{Q_I^{e_3}\}_{e_3 \in \mathcal{E}_y \cap \mathcal{E}_z}$ ;

```

---

**General Case 1:**  $|J| = 1$ . Suppose  $J = \{x\}$ . Recall that for each tuple  $t \in Q_I$ , Algorithm 6 computes the intersection  $\cap_{e \in \mathcal{E}_x} (R_e \times t)$  on  $x$  in the base case. To ensure this step remains oblivious, we must conceal the size of  $R_e \times t$ . To achieve this, we augment each tuple  $t \in Q_I$  with its *degree* in  $R_e$ , which is defined as  $\Delta_e(t) = |R_e \times t|$ , using the AUGMENT primitive. Then, we partition tuples in  $Q_I$  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  $Q_I^e \subseteq Q_I$  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  $t \in Q_I^e$ . Whenever we write one tuple  $t \in Q_I$  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  $R_e \bowtie Q_I^e$  by invoking the RELAXEDTOWAY primitive (line 9), with upper bound  $N^{\rho^*}$ , and further filter them by remaining relations with semi-joins (line 10).

**General Case 2:**  $|J| = 2$ . Suppose  $J = \{y, z\}$ . Consider an arbitrary tuple  $t \in Q_I$ . Algorithm 6 computes the residual query  $\left\{ \cap_{e \in \mathcal{E}_y \cap \mathcal{E}_z} (R_e \times t) \right\} \bowtie \left\{ \cap_{e \in \mathcal{E}_y - \mathcal{E}_z} (R_e \times t) \right\} \bowtie \left\{ \cap_{e \in \mathcal{E}_z - \mathcal{E}_y} (R_e \times t) \right\}$ . 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  $Q_I$  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

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

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 RELAXEDTOWWAY 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 RELAXEDTOWWAY primitive (line 20), with the upper bound  $N^{\rho^*(\mathcal{Q})}$ , and filter these results by remaining relations with semi-joins (line 21).

### 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  $O(N^{\rho^*(\mathcal{Q})} \cdot \log N)$  time and  $O\left(\frac{N^{\rho^*}}{B} \cdot \log \frac{M}{B} \frac{N}{B}\right)$  cache complexity, since  $\rho^*((I, \mathcal{E}[I])) \leq \rho^*(\mathcal{Q})$ . We then show the correctness and complexity for all invocations of RELAXEDTOWWAY 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:

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

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

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

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

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

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

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

## XX:16 Optimal Oblivious Algorithms for Multi-way Joins

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  $O(N^{\rho^*} \cdot \log N)$  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  $\rho^*$  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  $\lambda : \mathcal{T} \rightarrow 2^{\mathcal{V}}$  is a labeling function which associates to each vertex  $u \in \mathcal{T}$  a subset of attributes in  $\mathcal{V}$ ,  $\lambda_u$ , 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  $\min_{(\mathcal{T}, \lambda)} \max_{u \in \mathcal{T}} \rho^*(\lambda_u, \{e \cap \lambda : e \in \mathcal{E}\})$ .*

The pseudocode of our algorithm is given in Appendix D. 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 RELAXEDTOWWAY 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,  $\tau$ . This leads to a relaxed oblivious algorithm whose access pattern only depends on  $N$  and  $\tau$ .

► **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  $O((N^w + \tau) \cdot \log(N^w + \tau))$  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  $\tau$ , 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 [14].
- *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 [13, 49], extending these ideas to multi-way join processing remains an open challenge.



## References

- 1 S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases*, volume 8. Addison-Wesley Reading, 1995.
- 2 M. Abo Khamis, H. Q. Ngo, and A. Rudra. Faq: questions asked frequently. In *PODS*, pages 13–28, 2016.
- 3 A. Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.
- 4 M. Ajtai, J. Komlós, and E. Szemerédi. An  $O(n \log n)$  sorting network. In *STOC*, pages 1–9, 1983.
- 5 A. Arasu and R. Kaushik. Oblivious query processing. *ICDT*, 2013.
- 6 L. Arge, M. A. Bender, E. D. Demaine, B. 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.
- 7 G. Asharov, I. Komargodski, W.-K. Lin, K. Nayak, E. Peserico, and E. Shi. Oporama: Optimal oblivious ram. In *Eurocrypt*, pages 403–432. Springer, 2020.
- 8 A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins. In *FOCS*, pages 739–748. IEEE, 2008.
- 9 K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314, 1968.
- 10 P. Beame, P. Koutris, and D. Suciu. Communication steps for parallel query processing. *JACM*, 64(6):1–58, 2017.
- 11 C. Beeri, R. Fagin, D. Maier, and M. Yannakakis. On the desirability of acyclic database schemes. *JACM*, 30(3):479–513, 1983.
- 12 A. Beimel, K. Nissim, and M. Zaheri. Exploring differential obliviousness. In *APPROX/RANDOM*, 2019.
- 13 T. Chan, K.-M. Chung, W.-K. Lin, and E. Shi. Mpc for mpc: secure computation on a massively parallel computing architecture. In *ITCS*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.
- 14 T. H. Chan, K.-M. Chung, B. M. Maggs, and E. Shi. Foundations of differentially oblivious algorithms. In *SODA*, pages 2448–2467. SIAM, 2019.
- 15 Z. Chang, D. Xie, and F. Li. Oblivious ram: A dissection and experimental evaluation. *Proc. VLDB Endow.*, 9(12):1113–1124, 2016.
- 16 Z. Chang, D. Xie, F. Li, J. M. Phillips, and R. Balasubramonian. Efficient oblivious query processing for range and knn queries. *TKDE*, 2021.
- 17 Z. Chang, D. Xie, S. Wang, and F. Li. Towards practical oblivious join. In *SIGMOD*, 2022.
- 18 S. Chu, D. Zhuo, E. Shi, and T.-H. H. Chan. Differentially Oblivious Database Joins: Overcoming the Worst-Case Curse of Fully Oblivious Algorithms. In *ITC*, volume 199, pages 19:1–19:24, 2021.
- 19 V. Costan and S. Devadas. Intel sgx explained. *Cryptology ePrint Archive*, 2016.
- 20 N. Crooks, M. Burke, E. Cecchetti, S. Harel, R. Agarwal, and L. Alvisi. Obladi: Oblivious serializable transactions in the cloud. In *OSDI*, pages 727–743, 2018.
- 21 E. 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 S. Deng and Y. Tao. Subgraph enumeration in optimal i/o complexity. In *ICDT*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2024.
- 23 S. Devadas, M. v. Dijk, C. W. Fletcher, L. Ren, E. Shi, and D. Wachs. Onion oram: A constant bandwidth blowup oblivious ram. In *TCC*, pages 145–174. Springer, 2016.
- 24 S. Eskandarian and M. Zaharia. Oblidb: Oblivious query processing for secure databases. *Proc. VLDB Endow.*, 13(2).
- 25 R. Fagin. Degrees of acyclicity for hypergraphs and relational database schemes. *JACM*, 30(3):514–550, 1983.

- 26 A. Z. Fan, P. Koutris, and H. Zhao. Tight bounds of circuits for sum-product queries. *SIGMOD*, 2(2):1–20, 2024.
- 27 J. Flum, M. Frick, and M. Grohe. Query evaluation via tree-decompositions. *JACM*, 49(6):716–752, 2002.
- 28 M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *FOCS*, pages 285–297. IEEE, 1999.
- 29 C. Gentry, K. A. Goldman, S. Halevi, C. Julta, M. Raykova, and D. Wichs. Optimizing oram and using it efficiently for secure computation. In *PETs*, pages 1–18. Springer, 2013.
- 30 O. Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *STOC*, pages 182–194, 1987.
- 31 O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *JACM*, 43(3):431–473, 1996.
- 32 M. T. Goodrich. Data-oblivious external-memory algorithms for the compaction, selection, and sorting of outsourced data. In *SPAA*, pages 379–388, 2011.
- 33 G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions and tractable queries. *JCSS*, 64(3):579–627, 2002.
- 34 H. Hacigümüş, B. Iyer, C. Li, and S. Mehrotra. Executing sql over encrypted data in the database-service-provider model. In *SIGMOD*, pages 216–227, 2002.
- 35 B. He and Q. Luo. Cache-oblivious nested-loop joins. In *CIKM*, pages 718–727, 2006.
- 36 X. Hu. Cover or pack: New upper and lower bounds for massively parallel joins. In *PODS*, pages 181–198, 2021.
- 37 X. Hu, M. Qiao, and Y. Tao. I/o-efficient join dependency testing, loomis–whitney join, and triangle enumeration. *JCSS*, 82(8):1300–1315, 2016.
- 38 B. Ketsman and D. Suci. A worst-case optimal multi-round algorithm for parallel computation of conjunctive queries. In *PODS*, pages 417–428, 2017.
- 39 P. Koutris, P. Beame, and D. Suci. Worst-case optimal algorithms for parallel query processing. In *ICDT*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.
- 40 S. Krastnikov, F. Kerschbaum, and D. Stebila. Efficient oblivious database joins. *VLDB*, 13(12):2132–2145, 2020.
- 41 E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in) security of hash-based oblivious ram and a new balancing scheme. In *SODA*, pages 143–156. SIAM, 2012.
- 42 W.-K. Lin, E. Shi, and T. Xie. Can we overcome the  $n \log n$  barrier for oblivious sorting? In *SODA*, pages 2419–2438. SIAM, 2019.
- 43 H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms. *JACM*, 65(3):1–40, 2018.
- 44 H. Q. Ngo, C. Ré, and A. Rudra. Skew strikes back: New developments in the theory of join algorithms. *Acm Sigmod Record*, 42(4):5–16, 2014.
- 45 V. Ramachandran and E. Shi. Data oblivious algorithms for multicores. In *SPAA*, pages 373–384, 2021.
- 46 S. Sasy, A. Johnson, and I. Goldberg. Fast fully oblivious compaction and shuffling. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2565–2579, 2022.
- 47 E. Shi. Path oblivious heap: Optimal and practical oblivious priority queue. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 842–858. IEEE, 2020.
- 48 E. Stefanov, M. V. Dijk, E. Shi, T.-H. H. Chan, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: an extremely simple oblivious ram protocol. *JACM*, 65(4):1–26, 2018.
- 49 Y. Tao, R. Wang, and S. Deng. Parallel communication obliviousness: One round and beyond. *Proceedings of the ACM on Management of Data*, 2(5):1–24, 2024.
- 50 T. L. Veldhuizen. Leapfrog triejoin: A simple, worst-case optimal join algorithm. In *ICDT*, 2014.
- 51 J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing surveys (CsUR)*, 33(2):209–271, 2001.

- 52 X. Wang, H. Chan, and E. Shi. Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In *CCS*, pages 850–861, 2015.
- 53 Y. Wang and K. Yi. Query evaluation by circuits. In *PODS*, 2022.
- 54 M. Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, pages 82–94, 1981.
- 55 W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *NSDI 17*, pages 283–298, 2017.

## A Missing Materials in Section 1

**Graph Joins.** A join query  $Q = (\mathcal{V}, \mathcal{E})$  is a graph join if  $|e| \leq 2$  for each  $e \in \mathcal{E}$ , i.e., each relation contains at most two attributes.

**Loomis-Whitney Joins.** A join query  $Q = (\mathcal{V}, \mathcal{E})$  is a Loomis-Whitney join if  $\mathcal{V} = \{x_1, x_2, \dots, x_k\}$  and  $\mathcal{E} = \{\mathcal{V} - \{x_i\} : i \in [k]\}$ .

## B Oblivious Primitives in Section 2

We provide the algorithm descriptions and pseudocodes for the oblivious primitives declared in Section 2.2. For the local variables used in these primitives, **key**, **val**, **pos** and **cnt**, we do not need to establish obliviousness for them because they are stored in the trusted memory during the entire execution of the algorithms and the adversaries cannot observe the access pattern to them. But for all the temporal sets with non-constant size,  $K$  and  $L$ , they are stored in the untrusted memory.

**SemiJoin.** Given two input relations  $R, S$  and their common attribute(s)  $x$ , the goal is to replace each tuple in  $R$  that cannot be joined with any tuple in  $S$  with a dummy tuple  $\perp$ , i.e., compute  $R \times S$ . As shown in Algorithm 10, we first sort all tuples by their join values and break ties by putting  $S$ -tuples before  $R$ -tuples if they share the same join value in  $x$ . We then perform a linear scan, using an additional variable **key** to track the largest join value of the previous tuple that is no larger than the join value of the current tuple  $t$  visited. More specifically, we distinguish two cases on  $t$ . Suppose  $t \in R$ . If  $\pi_x t = \text{key}$ , we just write  $t$  to the result array  $L$ . Otherwise, we write a dummy tuple  $\perp$  to  $L$ . Suppose  $t \in S$ . We simply write a dummy tuple  $\perp$  to  $L$  and update **key** with  $\pi_x t$ . At last, we compact the elements in  $L$  to move all  $\perp$  to the last and keep the first  $|R|$  tuples in  $L$ .

### Algorithm 10 SEMIJOIN( $R, S, x$ )

---

```

1  $K \leftarrow \text{SORT } R \cup S$  by attribute(s)  $x$ , breaking ties by putting  $S$ -tuples before  $R$ -tuples
   when they have the same value in  $x$ ;
2  $\text{key} \leftarrow \perp, L \leftarrow \emptyset$ ;
3 while read  $t$  from  $K$  do
4   if  $t \in R$  then
5     if  $t \neq \perp$  and  $\pi_x t = \text{key}$  then write  $t$  to  $L$ ;
6     else write  $\perp$  to  $L$ ;
7   else write  $\perp$  to  $L$ ;
8    $\text{key} \leftarrow \pi_x t$ ;
9 return COMPACT  $L$  while keeping the first  $|R|$  tuples;

```

---

**ReduceByKey.** Given an input relation  $R_e$ , some of which are distinguished as  $\perp$ , a set of key attribute(s)  $x \subseteq e$ , a weight function  $w$ , and an aggregate function  $\oplus$ , the goal is

## XX:20 Optimal Oblivious Algorithms for Multi-way Joins

### ■ Algorithm 11 REDUCEBYKEY( $R, x, w(\cdot), \oplus$ )

---

```
1  $K \leftarrow$  SORT  $R$  by attribute(s)  $x$  with all  $\perp$  moved to the last;
2  $\text{key} \leftarrow \perp, \text{val} \leftarrow 0, L \leftarrow \emptyset$ ;
3 while read  $t$  from  $K$  do
4   if  $t = \perp$  then write  $\perp$  to  $L$ ;
5   else if  $t \neq \perp$  and  $\pi_x t = \text{key}$  then write  $\perp$  to  $L, \text{val} \leftarrow \text{val} \oplus w(t)$ ;
6   else write ( $\text{key}, \text{val}$ ) to  $L, \text{val} \leftarrow w(t), \text{key} \leftarrow \pi_x t$ ;
7 write ( $\text{key}, \text{val}$ ) to  $L$ ;
8 return COMPACT  $L$  while keeping the first  $|R|$  tuples;
```

---

### ■ Algorithm 12 ANNOTATE( $R, S, x$ )

---

```
1  $K \leftarrow$  SORT  $R \cup S$  by attribute(s)  $x$  while moving all  $\perp$  to the last and breaking ties
  by putting  $S$ -tuples before  $R$ -tuples when they have the same value in  $x$ ;
2  $\text{key} \leftarrow \perp, \text{val} \leftarrow 0, L \leftarrow \emptyset$ ;
3 while read  $t$  from  $K$  do
4   if  $t = \perp$  then write  $\perp$  to  $L$ ;
5   else if  $t \in S$  then
6     write  $\perp$  to  $L, \text{val} \leftarrow \pi_x t, \text{key} \leftarrow \pi_x t$ 
7   else if  $t \in R$  and  $\pi_x t = \text{key}$  then write ( $t, \text{val}$ ) to  $L$ ;
8   else write  $\perp$  to  $L$ ;
9 return COMPACT  $L$  while keeping the first  $|R|$  tuples;
```

---

to output the aggregation of each key value, which is defined as the function  $\oplus$  over the weights of all tuples with the same key value. This primitive can be used to compute *degree information*, i.e., the number of tuples displaying a specific key value in a relation.

As shown in Algorithm 11, we sort all tuples by their key values (values in attribute(s)  $x$ ) while moving all distinguished tuples to the last of the relation. Then, we perform a linear scan, using an additional variable  $\text{key}$  to track the key value of the previous tuple, and  $\text{val}$  to track the aggregation over the weights of tuples visited. We distinguish three cases. If  $t = \perp$ , the remaining tuples in  $K$  are all distinguished as  $\perp$ , implied by the sorting. We write a dummy tuple  $\perp$  to  $L$  in this case. If  $t \neq \perp$  and  $\pi_x t = \text{key}$ , we simply write a dummy tuple  $t$  to  $L$ , and increase  $\text{val}$  by  $w(t)$ . If  $t \neq \perp$  and  $\text{key} \neq \pi_x t$ , the values of all elements with key  $\text{key}$  are already aggregated into  $\text{val}$ . In this case, we need to write ( $\text{key}, \text{val}$ ) to  $L$  and update  $\text{val}$  with  $w(t)$ , i.e., the value of current tuple, and  $\text{key}$  with  $\pi_x t$ . At last, we compact the tuples in  $L$  by moving all  $\perp$  to the last and keep the first  $|R|$  tuples in  $L$  for obliviousness.

**Annotate.** Given an input relation  $R$ , where each tuple is associated with a key, and a list  $S$  of key-value pairs, where each pair is associated with a distinct key, the goal is to attach, for each tuple in  $R$ , the value of the corresponding distinct pair in  $S$  matched by the key. As shown in Algorithm 12, we first sort all tuples in  $R$  and  $S$  by their key values in attribute  $x$ , while moving all  $\perp$  to the last of the relation and breaking ties by putting all  $S$ -tuples before  $R$ -tuples when they have the same key value. We then perform a linear scan, using another two variables  $\text{key}, \text{val}$  to track the  $S$ -tuple with the largest key but no larger than the key of the current tuple visited. We distinguish the following cases. If  $t$  is a  $S$ -tuple and  $t \neq \perp$ , we update  $\text{key}, \text{val}$  with  $t$ . If  $t$  is a  $R$ -tuple and  $t \neq \perp$ , we attach  $\text{val}$  to  $t$  by writing ( $t, \text{val}$ ) to  $L$ . We write a dummy tuple  $\perp$  to  $L$  in the remaining cases. Finally, we compact the tuples in  $L$

---

**Algorithm 13** MULTINUMBER( $R, x$ )
 

---

```

1  $K \leftarrow$  SORT  $R$  by attribute(s)  $x$ ;
2  $\text{key} \leftarrow \perp$ ,  $\text{val} \leftarrow 0$ ,  $L \leftarrow \emptyset$ ;
3 foreach  $t \in K$  do
4   if  $\pi_x t = \text{key}$  then  $\text{val} \leftarrow \text{val} + 1$  ;
5   else  $\text{val} \leftarrow 1$ ,  $\text{key} \leftarrow \pi_x t$ ;
6   write  $(t, \text{val})$  to  $L$ ;
7 return  $L$ ;

```

---

**Algorithm 14** AUGMENT( $R, \{S_1, S_2, \dots, S_k\}, x$ )
 

---

```

1 foreach  $i \in [k]$  do
2    $L \leftarrow$  REDUCEBYKEY( $S_i, x$ );
3    $R \leftarrow$  ANNOTATE( $R, L, x$ );
4 return  $R$ ;

```

---

to remove unnecessary dummy tuples.

**MultiNumber.** Given an input relation  $R$ , each associated with a key attribute(s)  $x$ , the goal is to attach consecutive numbers  $1, 2, 3, \dots$ , to tuples with the same key.

As shown in Algorithm 13, we first sort all tuples in  $R$  by attribute  $x$ . We then perform a linear scan, using two additional variables  $\text{key}$ ,  $\text{val}$  to track the key of the previous tuples, and the number assigned to the previous tuple. Consider  $t$  as the current element visited. If  $\pi_x t = \text{key}$ , we simply increase  $\text{val}$  by 1. Otherwise, we set  $\text{val}$  to 1 and update  $\text{key}$  with  $\pi_x t$ . In both cases, we assign  $\text{val}$  to tuple  $t$  and write  $(t, \text{val})$  to  $L$ .

**Project.** Given an input relation  $R$  defined over attributes  $e$ , and a subset of attributes  $x \subseteq e$ , the goal is to output the list  $\{t \in R : \pi_x t\}$  (without duplication). This primitive can be simply solved by sorting by attribute(s)  $x$  and then removing duplicates by a linear scan.

**Intersect.** Given two input arrays  $R, S$  of distinct elements separately, the goal is to output the common elements appearing in both  $R$  and  $S$ . This primitive can be done with sorting by attribute(s)  $x$ , and then a linear scan would suffice to find out common elements.

**Augment.** Given two relations  $R, S$  of at most  $N$  tuples and their common attribute(s)  $x$ , the goal is to attach each tuple  $t$  the number of tuples in  $S$  that can be joined with  $t$  on  $x$ . The AUGMENT primitive can be implemented by the REDUCEBYKEY and ANNOTATE primitives. See Algorithm 14.

## C RelaxedTwoWay Primitive

Given two relations  $R, S$  of  $N_1, N_2$  tuples and an integral parameter  $\tau$ , where  $N_1 + N_2 = N$  and  $|R \bowtie S| \leq \tau$ , the goal is to output a relation of size  $\tau$  whose first  $|R \bowtie S|$  tuples are the join results and the remaining tuples are dummy tuples. Arasu et al. [5] first proposed an oblivious algorithm for  $\tau = |R \bowtie S|$ , but it involves rather complicated primitive without giving complete details [16]. Krastnikov et al. [40] later showed a more clean and effective version, but this algorithm does not have a satisfactory cache complexity. Below, we present our own version of the relaxed two-way join. We need one important helper primitive first.

**Expand Primitive.** Given a sequence of  $\langle (t_i, w_i) : w_i \in \mathbb{Z}^+, i \in [N] \rangle$  and a parameter  $\tau \geq \sum_{i \in [N]} w_i$ , the goal is to expand each tuple  $t_i$  with  $w_i$  copies and output a table of  $\tau$

■ **Algorithm 15**  $\text{EXPAND}(R = \langle (t_i, w_i) : i \in [N] \rangle, \tau)$

---

```

1 pos  $\leftarrow$  1,  $K \leftarrow \emptyset$ ;
2 while read  $(t_i, w_i)$  from  $R$  do
3   if  $(t_i, w_i) = (\perp, \perp)$  then write  $(\perp, +\infty)$  to  $X$ ;
4   else write  $(t_i, \text{pos})$  to  $K$ ,  $\text{pos} \leftarrow \text{pos} + w_i$ ;
5 pos  $\leftarrow$  1.5;
6 foreach  $i \in [\tau]$  do write  $(\perp, \text{pos})$  to  $K$ ,  $\text{pos} \leftarrow \text{pos} + 1$ ;
7 Sort  $K$  by pos;
8  $t \leftarrow \perp$ ,  $\text{cnt} \leftarrow 0$ ,  $L \leftarrow \emptyset$ ;
9 while read (key, pos) from  $K$  do
10  if pos =  $+\infty$  then write  $\perp$  to  $L$ ;
11  else if key  $\neq \perp$  then  $t \leftarrow$  key, write  $\perp$  to  $L$ ;
12  else if  $\text{cnt} < \tau$  then write  $t$  to  $L$ ;
13  else write  $\perp$  to  $L$ ;
14   $\text{cnt} \leftarrow \text{cnt} + 1$ ;
15 return COMPACT  $L$  while keeping the first  $\tau$  elements;

```

---

tuples. The naive way of reading a pair  $(t_i, w_i)$  and then writing  $w_i$  copies does not preserve obliviousness since the number of consecutive writes can leak the information. Alternatively, one might consider writing a fixed number of tuples after reading each pair. Still, the ordering of reading pairs is critical for avoiding dummy writes and avoiding too many pairs stored in trusted memory (this strategy is exactly adopted by [5]).

We present a simpler algorithm by combining the oblivious primitives. Suppose  $L$  is the output table of  $R$ , such that  $L$  contains  $w_i$  copies of  $t_i$ , and any tuple  $t_i$  comes before  $t_j$  if  $i < j$ . As described in Algorithm 15, it consists of four phases:

- (lines 1-4). for each pair  $(t_i, w_i) \in R$  with  $w_i \neq 0$ , attach the beginning position of  $t_i$  in  $\tilde{R}$ , which is  $\sum_{j < i} w_j$ . For the remaining pairs with  $w_i = 0$ , replace them with  $\perp$  and attach with the infinite position as these tuples will not participate in any join result;
- (lines 5-7) pad  $\tau$  dummy tuples and attach them with consecutive numbers  $1.5, 2.5, \dots$ ; after sorting the well-defined positions, each tuple  $t_i$  will be followed by  $w_i$  dummy tuples, and all dummy tuples with infinite positions are put at last;
- (lines 8-14) for each tuple  $t_i$ , we replace it with  $\perp$  but the following  $w_i$  dummy tuples with  $t_i$ . After moving all dummy tuples to the end, the first  $\tau$  elements are the output.

It can be easily checked that the access pattern of  $\text{EXPAND}$  only depends on the values of  $\tau$  and  $N$ . Moreover,  $\text{EXPAND}$  is cache-agnostic since they are constructed by sequential compositions of cache-agnostic primitives ( $\text{SCAN}$ ,  $\text{SORT}$  and  $\text{COMPACT}$ ).

► **Lemma 12.** *Given a relation  $\mathcal{R}$  of input size  $N$  and a parameter  $\tau$ , the  $\text{EXPAND}$  primitive is cache-agnostic with  $O((N + \tau) \cdot \log(N + \tau))$  time complexity and  $O\left(\frac{N + \tau}{B} \log_{\frac{M}{B}} \frac{N + \tau}{B}\right)$  cache complexity, whose access pattern only depends on  $N$  and  $\tau$ .*

Now, we are ready to describe the algorithmic details of  $\text{RELAXEDTOWWAY}$  primitive. The high-level idea is to simulate the sort-merge join algorithm without revealing the movement of pointers in the merge phase. Let  $L = R(x_1, x_2) \bowtie S(x_2, x_3)$  be the join results sorted by  $x_2, x_3, x_1$  lexicographically. The idea is to transform  $R, S$  into a sub-relation of  $L$  by keeping attributes  $(x_1, x_2), (x_2, x_3)$  separately, without removing duplicates. Then, doing a one-to-one merge to obtain the final join results suffices. As described in Algorithm 16,

---

**Algorithm 16** RELAXEDTWOWAY( $R(x_1, x_2), S(x_2, x_3), \tau$ )
 

---

```

1  $\hat{R} \leftarrow \text{AUGMENT}(R, S, x_2), \hat{S} \leftarrow \text{AUGMENT}(S, R, x_2);$ 
2  $\tilde{R} \leftarrow \text{EXPAND}(\hat{R}, \tau), \tilde{S} \leftarrow \text{EXPAND}(\hat{S}, \tau);$ 
3  $\bar{S} \leftarrow \text{MULTINUMBER}(\tilde{S}, x_2);$  //  $\bar{S}$  is enriched with another attribute num
4 SORT  $\bar{S}$  by attributes  $x_2$  and num lexicographically;
5  $L \leftarrow \emptyset;$ 
6 while read  $t_1$  from  $\tilde{R}$  and read  $t_2$  from  $\bar{S}$  do
7   if  $t_1 \neq \perp$  and  $t_2 \neq \perp$  then write  $t_1 \bowtie t_2$  to  $L;$ 
8   else write  $\perp$  to  $L;$ 
9 return  $L;$ 

```

---

we construct these two sub-relations from the input relations  $R, S$  via the following steps (a running example is given in Figure 1):

- (line 1) attach each tuple with the number of tuples it can be joined in the other relation;
- (line 2) expand each tuple to the annotated number of copies;
- (lines 3-4) prepare the expanded  $\tilde{R}$  and  $\tilde{S}$  with the “correct” ordering, as it appears in the final sort-merge join results;
- (lines 5-8) perform a one-to-one merge of ordered tuples in  $\tilde{R}$  and  $\tilde{S}$ ;

As a sequential composition of (relaxed) oblivious primitives, RELAXEDTWOWAY is cache-agnostic, with  $O((N + \tau) \cdot \log(N + \tau))$  time complexity and  $O(\frac{N+\tau}{B} \cdot \log \frac{M}{B} \cdot \frac{N+\tau}{B})$  cache complexity, whose access pattern only depends on  $N$  and  $\tau$ .

## D Missing Materials in Section 5

---

**Algorithm 17** RELAXEDJOIN( $Q = (\mathcal{V}, \mathcal{E}), \mathcal{R}, \tau$ )
 

---

```

1  $(\mathcal{T}, \lambda) \leftarrow$  a GHD of  $Q;$ 
2 foreach node  $u \in \mathcal{T}$  do
3    $\mathcal{E}_u \leftarrow \{e \cap \lambda_u : e \in \mathcal{E}\};$ 
4   foreach  $e \in \mathcal{E}$  do  $S_{e,u} \leftarrow \pi_{e \cap \lambda_u} R_e$  by PROJECT;
5    $\mathcal{Q}_u \leftarrow \text{OBLIVIOUSGENERICJOIN}((\lambda_u, \mathcal{E}_u), \{S_{e,u} : e \in \mathcal{E}\});$ 
6 while visit nodes  $u \in \mathcal{T}$  in a bottom-up way (excluding the root) do
7    $p_u \leftarrow$  the parent node of  $u;$ 
8    $\mathcal{Q}_{p_u} \leftarrow \text{SEMIJOIN}(\mathcal{Q}_{p_u}, \mathcal{Q}_u);$ 
9 while visit nodes  $u \in \mathcal{T}$  in a top-down way (excluding the leaves) do
10  foreach child node  $v$  of  $u$  do  $\mathcal{Q}_v \leftarrow \text{SEMIJOIN}(\mathcal{Q}_v, \mathcal{Q}_u);$ 
11 while visit nodes  $u \in \mathcal{T}$  in a bottom-up way (excluding the root) do
12   $p_u \leftarrow$  the parent node of  $u;$ 
13   $\mathcal{Q}_{p_u} \leftarrow \text{RELAXEDTWOWAY}(\mathcal{Q}_{p_u}, \mathcal{Q}_u, \tau);$ 
14 return  $\mathcal{Q}_r$  for the root node  $r$  of  $\mathcal{T};$ 

```

---

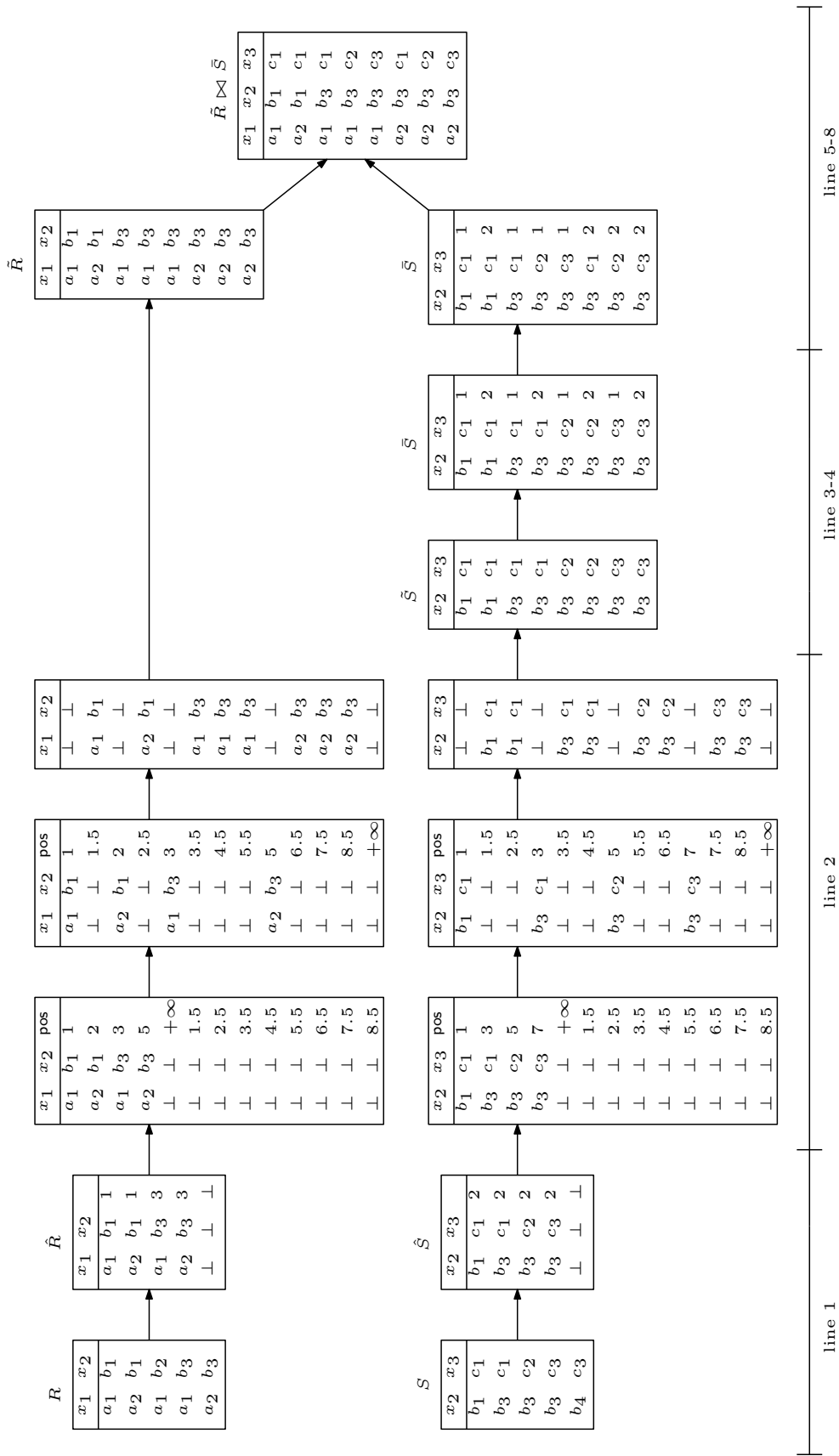


Figure 1 A running example of Algorithm 16.