NORTHWESTERN
UNIVERSITY

Computer Science Department

Technical Report
Number: NU-CS-2025-11

June, 2025

Towards Oblivious Worst-Case Optimal Joins

Yihang Du

Abstract

As outsourced data storage and computation become increasingly prevalent, security concerns have intensified, particularly due to the threat of access pattern leakage from database operators, which can reveal sensitive information even when the data has been encrypted. Numerous works have demonstrated that such attacks can be powerful, even leading to reconstructions of the whole database. Among those operators, the join draws much attention as it is especially critical and vulnerable, yet designing oblivious join algorithms is a non-trivial task. Although several practical oblivious join algorithms have been proposed, they are inherently suboptimal for *worst-case optimal join* (WCOJ) problems. While generic approaches exist, such as Oblivious RAM (ORAM) and circuits, they suffer from prohibitive computational overhead in practice. Due to the complexity of WCOJ queries, only limited progress has been made on efficient, problem-specific oblivious WCOJ algorithms.

This work focuses on a fundamental WCOJ problem: the *triangle join query*. We survey the structural insights underlying the triangle join query evaluations and investigate the intricacies of adapting such algorithms to oblivious settings. In addition, applying the classical notion of obliviousness to triangle join algorithms requires padding intermediate results and the final output to the worst-case length to prevent access pattern leakages, further inflating the cost. Therefore, to overcome this, we present two triangle join algorithms that satisfy *differential obliviousness* (DO), a recently proposed relaxation of access pattern privacy. By leveraging

differentially oblivious primitives, our algorithm can achieve instance-specific performance while providing meaningful privacy guarantees. Both algorithms have the worst-case runtime matching the insecure WCOJ algorithms, up to poly-logarithmic factors.

Keywords

Secure Database Join, Data Privacy, Oblivious Query Processing, Worst-Case Optimal Join, Triangle Join Query, Differential Obliviousness

NORTHWESTERN UNIVERSITY

Towards Oblivious Worst-Case Optimal Joins

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

MASTER OF SCIENCE

Field of Computer Science

By

Yihang Du

EVANSTON, ILLINOIS

June 2025

ABSTRACT

As outsourced data storage and computation become increasingly prevalent, security concerns have intensified, particularly due to the threat of access pattern leakage from database operators, which can reveal sensitive information even when the data has been encrypted. Numerous works have demonstrated that such attacks can be powerful, even leading to reconstructions of the whole database. Among those operators, the join draws much attention as it is especially critical and vulnerable, yet designing oblivious join algorithms is a non-trivial task. Although several practical oblivious join algorithms have been proposed, they are inherently suboptimal for *worst-case optimal join* (WCOJ) problems. While generic approaches exist, such as Oblivious RAM (ORAM) and circuits, they suffer from prohibitive computational overhead in practice. Due to the complexity of WCOJ queries, only limited progress has been made on efficient, problem-specific oblivious WCOJ algorithms.

This work focuses on a fundamental WCOJ problem: the *triangle join query*. We survey the structural insights underlying the triangle join query evaluations and investigate the intricacies of adapting such algorithms to oblivious settings. In addition, applying the classical notion of obliviousness to triangle join algorithms requires padding intermediate results and the final output to the worst-case length to prevent access pattern leakages, further inflating the cost. Therefore, to overcome this, we present two triangle join algorithms that satisfy *differential obliviousness* (DO), a recently proposed relaxation of access pattern privacy. By leveraging differentially oblivious primitives, our algorithm can achieve instance-specific performance while providing meaningful privacy guarantees. Both algorithms have the worst-case runtime matching the insecure WCOJ algorithms, up to poly-logarithmic factors.

ACKNOWLEDGEMENTS

I would like to express my DEEPEST gratitude to Professor Jennie Rogers for her invaluable guidance, patience, and encouragement throughout the research. As someone new to the topic, I struggled like a caged bird loosed into an ancient forest and drifted through the undergrowth of thought. Yet, Professor Rogers has consistently offered clarity and insight, guiding me to move forward with courage and confidence, as the pole star in the wilderness. Her unwavering support and exemplary professionalism have shaped not only my research but my spirit as a scholar.

I am also sincerely grateful to Professor Xiao Wang. Without him, this thesis would not have been possible. His engaging lectures and sense of humor can always turn the dense woods of cryptography into a path lit with laughter. They remain the most enjoyable lectures I have ever attended here.

Additionally, I would like to thank Professor Chris Riesbeck and Jensen Smith, the graduate program assistant, for their kind support and assistance. Their help with navigating procedures and ensuring a stable and smooth experience at the university has been truly appreciated.

人生达命岂暇愁， 且饮美酒登高楼。

Glossary

AGM Bound An upper bound on the output size of conjunctive queries (including joins), defined by the AGM inequality. For triangle joins, the bound is $O(N^{3/2})$ if each input size is N.

Differential Obliviousness (DO) A relaxed notion of access pattern privacy that allows limited leakage by requiring that observable behaviors (access patterns, control flow, etc.) be computationally indistinguishable for neighboring inputs, with the indistinguishability controlled via parameters.

Full Obliviousness (FO) The strongest classical notion of obliviousness where only the input size is leaked. All memory access patterns, control flows, and output sizes must be independent of sensitive data.

Instance-Specific Performance An algorithmic property where the runtime scales with the actual (instance-specific) output size rather than a fixed worst-case bound, enabled under differential obliviousness.

Leapfrog Triejoin (LFTJ) A worst-case optimal join algorithm that simultaneously probes multiple sorted input relations in a trie-like structure to find satisfying tuples.

Oblivious RAM (ORAM) A cryptographic construction that enables a client to access public memory in a data-oblivious manner, hiding the access patterns by shuffling locations where data is stored.

Obliviousness A security property requiring that a program's access pattern (e.g., memory access trace) is independent of the input data, protecting against leakage through side channels.

Triangle Join Qeury A canonical join query of the form $Q_{\blacktriangle} = R(A, B) \bowtie S(B, C) \bowtie T(A, C)$; often used to benchmark WCOJ algorithms due to its structured cyclic dependency.

Trusted Execution Environment (TEE) A hardware-based isolated execution environment (e.g., Intel SGX) that provides data confidentiality and integrity against a potentially compromised OS.

Worst-Case Optimal Join (WCOJ) A class of join algorithms that run in time proportional to the worst-case output size of a join query, rather than the size of the input relations or a fixed join plan.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1
INTRODUCTION AND BACKGROUND

With the growing demand for storing and managing large amounts of data, cloud-based services have become a critical infrastructure. The outsourced mode of data management inevitably raises serious concerns about the security and privacy of sensitive data, as they are beyond the direct control of the data owner.

To address these concerns, numerous research and development efforts have focused on encrypted databases that enable computations over encrypted data [1]–[3], as well as hardware-assisted approaches that maintain data privacy and confidentiality during computation. On the hardware side, trusted execution environments (TEE) enable secure execution within an isolated memory region, serving as a secure enclave even in the presence of a compromised operating system. For example, Intel SGX [4] provides limited but non-constant secure memory to hold and decrypt the necessary data involved in active computations. Once the computation is complete, the data is re-encrypted and transferred back to the untrusted server for storage.

While these approaches provide strong data privacy guarantees and prevent the disclosure of sensitive data, they alone do not eliminate all security risks, particularly those arising from the leakage of data access patterns. Recent studies [5]–[8] have shown that adversaries who observe access patterns would be able to infer sensitive information. For example, consider two relations *Company(CompanyID, City)* and *Supplier(SupplierID, Materials, CompanyID)*. Suppose all data stored in the database is encrypted, and all computations are complete either over encrypted data or in the secure enclaves. It is reasonable to assume that any compromised server or curious adversary has no access to the actual data contents. However, con-

sider a query that joins the two tables on `CompanyID` with the sort-merge join algorithm. For each tuple $t_c \in$ `Company`, we will scan contiguous tuples $t_s \in$ `Supplier` to produce join result (t_c, t_s) if matched tuples are found. Then, an adversary who can observe the access patterns can easily infer the number of suppliers each company has by simply counting the number of matched t_s for each t_c. By observing the join degrees, an adversary may be able to reconstruct a significant amount of data and queries, as Islam et al. demonstrated a successful inference on approximately 80% of the search queries [9], [10].

To mitigate such leakage risks, database operators should be implemented using oblivious algorithms. An algorithm is considered *oblivious* if its decisions about which locations to access are independent of the actual data values [11]. Hiding access patterns could be easy for certain operators that can be computed through linear scans. For those non-trivial ones, one can always use Oblivious RAM (ORAM), which hides access patterns on remote storage by shuffling the locations where data is stored and re-encrypting data after each access [10], [12]. However, the high computational overhead incurred on every memory access often makes ORAM an impractical choice for designing efficient oblivious programs. An alternative generic solution is to use circuits, where the query evaluation process is represented as Boolean or arithmetic circuits. A program built with circuits is inherently oblivious because all the computation logic is pre-defined and data-independent, which makes them well-suited for secure multi-party computation and outsourced query processing. However, this requires the server to be capable of generating circuits, which can be computationally expensive. Moreover, since circuits must be generated for worst-case input and output sizes, they can be inefficient for queries with highly variable result sizes [13]. Therefore, the pursuit of designing problem-specific oblivious algorithms that do not rely on generic primitives or assumptions of secure enclaves has never stopped.

Among the extensive studies on designing such oblivious database operator algorithms, join

algorithms have received significant attention due to their inherent complexity and the potential for further optimization. Arasu and Kaushik [14] introduce a suite of oblivious query processing algorithms. Their oblivious binary join algorithm leverages core primitives to compute each tuple's contribution to the final join result, duplicate them as needed, and then stitch both sides together to produce the output. However, Krastnikov et al. [11] later highlight the missing implementation details in [14] and propose a binary join algorithm based on sorting networks, which achieves asymptotic complexity of the insecure sort-merge join up to a logarithmic factor. Other systems, such as Opaque [15] and ObliDB [16], also present oblivious binary join algorithms, but their designs are limited to primary-foreign key joins [11], [17]. Although many studies have proposed oblivious binary join algorithms—and extending these approaches to handle acyclic multi-way joins is relatively straightforward—there has been little research focused on designing *worst-case optimal join* (WCOJ) algorithms in the oblivious setting. We will focus on a special and fundamental case of WCOJ algorithms—the triangle join problem—which has remained a central topic of interest for over two decades [18]–[20], and encapsulates many of the key insights underlying the general WCOJ problem.

For triangle query

$$Q_{\blacktriangle} = R(A, B) \bowtie S(B, C) \bowtie T(A, C),$$

Atserias, Grohe, and Marx (AGM) [21] developed a tight bound on $|Q_{\blacktriangle}|$ the size of $Q_{\blacktriangle}$, called the fractional edge cover bound $N^{\rho^*(Q_{\blacktriangle})}$. Suppose $|R| = |S| = |T| = N$, then $N^{\rho^*(Q_{\blacktriangle})} = N^{3/2}$. The tight upper bound on the worst-case output size determines the worst-case running time of algorithms designed to evaluate such queries [22]. Several well-known non-secure WCOJ algorithms [22]–[24] have been proposed with worst-case running times that match the bound up to a logarithmic factor. This line of work is motivated by the fact that traditional pair-wise join strategies generate intermediate results of size $O(N^2)$, leading to inherently suboptimal running

times. In the oblivious setting, we might further need to pad the intermediate results to the worst-case size N^2 to prevent leakage through access patterns. This makes the development of an efficient oblivious WCOJ algorithm both challenging and critically important.

Few works have been proposed in the field. Wang et al. [13] proposed circuit-based query evaluation algorithms for conjunctive queries, although generating circuits is often expensive in practice. Hu and Wu [25] recently proposed a suite of oblivious versions of algorithms presented in [23]. Their running times match the AGM bound up to a logarithmic factor and present practical cache complexity. However, their algorithms pad the necessary intermediate results to the worst-case upper bound $N^{\rho^*(Q_\blacktriangle)}$ to prevent potential leakages, although, in practice, the actual sizes might be less than that. We present our algorithms that, instead of achieving full obliviousness and padding results to the worst-case upper bound, achieve *instance-specific* performance by applying the relaxed notion of access pattern privacy called (ϵ, δ)-*differential obliviousness* (DO) [26], which was first introduced by Chan et al. [27]. With the relaxed notion of access privacy, we can achieve meaningful privacy guarantees while avoiding matching the worst-case performance for all instances of triangle joins.

1.1 Contributions

This thesis investigates the intersection of WCOJ problems and oblivious algorithm design, with a particular focus on the triangle join query. We provide analyses of the scarcity of practical, oblivious WCOJ algorithms by identifying the challenges stemming from their structural complexity and the incompatibility of traditional WCOJ strategies with oblivious techniques. To bridge the gap, we construct two naïve baseline algorithms that satisfy full obliviousness and DO, respectively. Both constructions simulate existing insecure WCOJ algorithms using ORAM. While similar constructions for WCOJ problems are often briefly mentioned, they are barely formalized and analyzed, as

ORAM-based methods suffer from substantial computational overhead.

To overcome various barriers, we detail reformulations of existing WCOJ algorithms into equivalent binary-join-based variants. We demonstrate that such reformulations retain the worst-case optimality while allowing us to exploit existing oblivious primitives. By leveraging a recently proposed DO binary join algorithm, we construct two problem-specific algorithms for evaluating triangle join queries. Our constructions avoid the overhead of padding intermediate and final results to the worst-case upper bound and instead achieve instance-specific performance, adapting runtime to the actual output size. We further demonstrate that our DO WCOJ algorithms, based on practical assumptions, both achieve runtimes that match the worst-case optimality, up to poly-logarithmic factors. We summarize our results in Table 1.1.

Moreover, since our constructions, although satisfactory, still reflect limitations imposed by the structural complexity of WCOJ problems and available oblivious tools, we point out future directions such as the development of lightweight oblivious primitives for multi-way joins or the design of specialized DO algorithms for WCOJ problems.

Algorithm	**Runtime**	**Output Length**
Naïve FO WCOJ	$O((N^{3/2}\log N + N)\log^2 N)$	$\Theta(N^{3/2})$
Naïve DO WCOJ	$O((\lvert Q_\blacktriangle\rvert \log N + N\log N)\log^2 N)$	$\lvert Q_\blacktriangle\rvert + O(N\log N)$
Our Algorithm 3	$O(M\log^2 M + N\log^2 N)$	$\lvert Q_\blacktriangle\rvert + O\left((\mu + \log N)\cdot \log N\right)$
Our Algorithm 4	$O(M\log^2 M + N\log^2 N)$	$\lvert Q_\blacktriangle\rvert + O\left((\mu + \log N)\cdot \log N\right)$

Table 1.1: Comparison of runtime and output length across baselines and proposed WCOJ algorithms. Here, μ denotes the maximum multiplicity of join keys across input relations, and M represents the maximum cardinality of intermediate results, where $M \leq N^{3/2}$. FO refers to full obliviousness.

1.2 Roadmap

The remainder of this work is organized as follows. Chapter 2 introduces essential background knowledge, including theoretical foundations of threat models, degrees of obliviousness, and the triangle join query. While it does not encompass all conventional demands for preliminaries, they are crucial for understanding the algorithms and constructions presented later. Chapter 3 includes a problem-driven review. We begin by formally defining the problem we would like to address, and then we outline the necessary insecure WCOJ algorithms and discuss the challenges in making existing WCOJ algorithms oblivious. Given the limited number of existing problem-specific WCOJ algorithms and their close relevance to our work, we discuss them alongside our own constructions rather than in a standalone survey. Chapter 4 contains our main contributions. It first describes the core primitives we adopt from existing literature. Next, we present two DO WCOJ algorithms for the triangle query, accompanied by thorough security and performance analyses. Finally, Chapter 5 concludes the thesis and outlines potential future research directions.

CHAPTER 2
PRELIMINARIES

In this chapter, we will introduce the necessary preliminaries to provide the theoretical basis for the results presented later. We first clarify the threat model and assumptions about the capabilities of the potential adversaries. We then review the notion of differential obliviousness and formally present corresponding definitions, a relaxed notion of access pattern privacy. To provide context for oblivious programs, we briefly introduce ORAM and discuss its advantages and limitations. Finally, we will talk about the triangle join query problem and related concepts, which play a fundamental role in the study of WCOJ problems.

2.1 Threat Model

Unless otherwise specified, we will present our work under the standard Random Access Machine (RAM) model of computation, in contrast to the External Memory (EM) model. The general adversarial scenario assumes a trusted client that outsources encrypted data to an untrusted storage. Since all data is encrypted, any adversary, including the storage provider, can only observe the sequence of memory accesses but cannot learn the data contents.

2.1.1 Degrees of Obliviousness

To address the concerns of data access pattern leakages, the notion of obliviousness was initially introduced by Goldreich and Ostrovsky [12], [28]. As research in oblivious computations progresses, more fine-grained leakage models have been proposed, leading to more refined notions of obliviousness. Understanding the degrees of obliviousness a program may satisfy is critical in

designing oblivious programs, as they serve as rigorous guidance under various leakage models. Hence, we briefly introduce the distinctions between the three common, established definitions of obliviousness.

Level I. Programs that satisfy this level of obliviousness are typically discussed under the EM model. They assume the existence of a non-constant amount of trusted memory. The memory could reside on a client's local machine or within TEE on the server, such as Intel SGX. A program is said to satisfy level I obliviousness if its memory accesses to external (untrusted) memory are independent of data contents. This level of obliviousness is commonly referred to as *external-memory obliviousness* or *singly-obliviousness*. The notion of obliviousness introduced by Goldreich and Ostrovsky satisfies external-memory obliviousness, although such a distinction was not explicitly necessary at the time of its introduction.

Level II. In contrast to the level I obliviousness, programs satisfying level II obliviousness must ensure that their access patterns to both the external memory and within the TEE are independent of data contents. Therefore, the available amount of private memory is bounded by $O(1)$, typically encompassing available CPU registers and on-core cache memory. Consequently, both the computation and related input at each step should conform to this limited budget, ensuring no data-dependent access is made due to cache eviction. This is driven by the fact that hardware enclaves like Intel SGX are vulnerable to side-channel attacks [11], [29], [30]. In particular, page-level memory accesses within the enclave can leak sensitive information. We refer to this stronger notion as *doubly-obliviousness* or *protected-memory obliviousness*.

Level III. Level III obliviousness is the strongest notion of obliviousness among those discussed. A program that satisfies level III obliviousness must ensure that its control flow is entirely independent of data contents. Hence, it is often referred to as *control-flow obliviousness*. At the instruction level, sensitive information can be leaked not only through memory accesses but also through the

number of executed instructions, branching behavior, and timing variations [31], [32]. These side channels can be exploited by adversaries to infer data-dependent executions, even when memory access patterns are protected. Therefore, achieving such obliviousness requires careful program design and specific implementation engineering to ensure that conditional branches, loop iterations, and cache timings are all independent of data contents.

2.2 Full Obliviousness

In common oblivious algorithm literature, the term "full obliviousness" typically refers to programs that satisfy all three levels of obliviousness. However, in the context of DO, the term "full obliviousness" is often reinterpreted to mean that the only allowed leakage is the input size, primarily as a way to contrast it with the notion of DO. This divergence potentially introduces ambiguity, as "full obliviousness" does not necessarily imply satisfaction of all three levels. To avoid confusion, we adopt precise and consistent terminology throughout this work. We will stick to the meaning of "full obliviousness" in the context of DO. When referring to full obliviousness, we explicitly state the level at which it is satisfied (e.g., full obliviousness at level II). Similarly, we refer to DO at level II when describing programs that simultaneously meet the corresponding guarantees for both notions. Unless otherwise stated, when we say an algorithm is "oblivious", we mean that it satisfies at least one of the three levels of obliviousness, whether full obliviousness or DO.

To state the notions of obliviousness, we introduce some necessary notations. A database $\mathcal{D}$ is modeled as a collection of records drawn from some domain, and an algorithm $\mathcal{A}$ interacts with $\mathcal{D}$ through a sequence of memory accesses, denoted by $\texttt{Access}_{\mathcal{A}}(\mathcal{D})$. The database $\mathcal{D}$ may contain real and dummy elements, so $\texttt{Access}_{\mathcal{A}}(\mathcal{D})$ encompasses `Write` and `Read` operations on both of them. For an algorithm $\mathcal{A}$ to be oblivious, its access pattern should be independent of the actual contents of records in $\mathcal{D}$.

Definition 2.2.1 (Full Obliviousness [25], [26], [33], [34])**.** *An algorithm $\mathcal{A}$ is fully oblivious if for any pair of databases $\mathcal{D}_1$, $\mathcal{D}_2$ of the same length and any subset of possible memory access patterns S:*

$$\Pr[\texttt{Access}_{\mathcal{A}}(\mathcal{D}_1) \in S] = \Pr[\texttt{Access}_{\mathcal{A}}(\mathcal{D}_2) \in S].$$

While full obliviousness offers strong data privacy guarantees, it often comes at a significant performance cost, as the only allowed leakage is the input size $|\mathcal{D}|$. Fully oblivious join algorithms require padding outputs and especially intermediate results to the worst-case sizes, which are quadratic in their input sizes, resulting in a substantial performance penalty compared to the insecure baseline [26].

2.3 Differential Obliviousness

To overcome the barrier, a relaxed notion called (ϵ, δ)-*differential obliviousness* (DO) has been proposed and defined by Chan et al. [27]. The notion of DO adapts the principles of differential privacy [35], [36] to the setting of memory access patterns. In general, differential privacy aims to protect the privacy of individual users by adding noise to the published outputs released by a trusted server, whereas DO considers an untrusted server and requires the access patterns to be (ϵ, δ)-differentially private [33]. In contrast to full obliviousness, instead of requiring indistinguishability for any pair of inputs, DO only requires that for *neighboring databases*. Two databases $\mathcal{D}_1$ and $\mathcal{D}_2$ are considered to be neighboring if they are of the same length N but differ in one record (i.e., Hamming distance 1) [26], [33].

Definition 2.3.1 ((ϵ, δ)-Differential Obliviousness [27], [33])**.** *An algorithm $\mathcal{A}$ satisfies (ϵ, δ)-differential obliviousness if for any two* ***neighboring databases*** *$\mathcal{D}_1$, $\mathcal{D}_2$ of size N, and any subset*

of memory access patterns S,

$$\Pr[\texttt{Access}_{\mathcal{A}}(\mathcal{D}_1) \in S] \leq e^{\epsilon} \cdot \Pr[\texttt{Access}_{\mathcal{A}}(\mathcal{D}_2) \in S] + \delta,$$

where δ is a negligible function in N and ϵ controls privacy loss.

The relaxation allows for a mathematically rigorous trade-off between privacy and efficiency. As demonstrated by recent works [26], [33], algorithms that satisfy (ϵ, δ)-DO can substantially outperform operators implemented with fully oblivious algorithms, achieving near instance-specific performance that matches insecure baselines [26]. By allowing parametrized leakages, (ϵ, δ)-DO can provide greater flexibility in algorithm design and lift many constraints imposed by full obliviousness, while still providing meaningful privacy guarantees. For typical parameter choices, we follow the convention that: $\epsilon = \Theta(1)$ and $\delta = 1/N^c$ for some constant $c \geq 1$ [26].

2.3.1 Mathematical Building Blocks

Here we present some necessary theoretical building blocks we will need later. We follow the conventions in typical differential privacy works and adopt facts and lemmas from [26].

Definition 2.3.2 (δ-oblivious)**.** *We say that an algorithm $\mathcal{A}$ satisfies δ-obliviousness with respect to the leakage function $\texttt{Leak}(\cdot)$, iff there exists a simulator $\texttt{Sim}$, such that $\texttt{Access}_{\mathcal{A}}(\mathcal{D})$ has statistical distance at most δ from the simulated access patterns $\texttt{Sim}(\texttt{Leak}(\mathcal{D}))$.*

To define full obliviousness with Definition 2.3.2, we simply set $\delta = 0$ and make the leakage function $\texttt{Leak}(\mathcal{D}) = |\mathcal{D}|$.

Definition 2.3.3 (Symmetric geometric distribution)**.** *Let $\alpha > 1$. The symmetric geometric distribution $\texttt{Geom}(\alpha)$ takes integer values such that the probability mass function at k is $\frac{\alpha-1}{\alpha+1} \cdot \alpha^{-|k|}$.*

Differential oblivious algorithms typically conceal access patterns and result sizes by adding noise to them. With Definition 2.3.3, following the method of [26], we sample noises from the distribution shown below.

Definition 2.3.4 (Shifted and truncated geometric distribution)**.** *Let $\epsilon > 0$ and $\delta \in (0,1)$ and $\Delta \geq 1$. Let k_0 be the smallest positive integer such that $\Pr[|\texttt{Geom}(e^{\epsilon/\Delta})| \geq k_0] \leq \delta$, where $k_0 = \frac{\Delta}{\epsilon} \ln \frac{2}{\delta} + O(1)$. The shifted and truncated geometric distribution $\mathcal{G}(\epsilon, \delta, \Delta)$ has support in $[0, 2(k_0 + \Delta - 1)]$, and is defined as:*

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

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

Fact 2.3.5 (Differential privacy by adding truncated and shifted geometric noise [27], [37])**.** *Let $\epsilon > 0$ and $\delta \in (0,1)$. Suppose u and v are two non-negative integers such that $|u - v| \leq \Delta$. Then,*

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

Fact 2.3.6 (Post-processing)**.** *Let $X \in \mathcal{X}$ and $X' \in \mathcal{X}$ be random variables and let $F : \mathcal{X} \to \mathcal{Y}$ be a possibly randomized function. Suppose that $X \sim_{(\epsilon,\delta)} X'$. Then, we have that*

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

Lemma 2.3.7 (Operational lemma for DO [27])**.** *If an algorithm is δ_0-oblivious w.r.t. a leakage function $\texttt{Leak}(\cdot)$, and moreover, $\texttt{Leak}$ is (ϵ, δ)-differentially private, then the algorithm satisfies (ϵ, δ)-DO.*

2.4 ORAM

Goldreich and Ostrovsky [12], [28] first proposed ORAM, a software architecture that hides a client's access patterns to outsourced storage by dynamically shuffling the locations of accessed data. ORAM guarantees that any two access sequences of the same length are computationally indistinguishable to anyone monitoring the system, thereby effectively concealing the access patterns. A key advantage of ORAM is that it can be deployed with arbitrary programs to make them (singly) oblivious with respect to the total number of accesses (i.e., the total number of accesses, or runtime, is revealed, and an adversary can simulate the access pattern with it). We can easily make such programs fully oblivious by padding ORAM accesses to the worst-case upper bound. Among the many ORAM constructions, one of the most prominent is *Path ORAM* [10] proposed by Stefanov et al. In Path ORAM, encrypted data blocks are stored on the server in a binary tree with height approximately $\Theta(\log N)$, where N denotes the total number of outsourced data blocks. Each leaf in the tree defines a unique path from that leaf node to the root. The client maintains a position map that associates each data block with a leaf label. When a data block is accessed, the client retrieves all data blocks residing along the corresponding path to a *locally maintained stash*. For example, in Figure 2.1, the client would like to perform operations on the violet block, which is associated with the leaf label numbered by 2. Then, all blocks along the path defined by that leaf label will be retrieved to the stash. After completing the operation on the target block, the client randomly assigns new leaf labels to all the retrieved blocks and evicts them back to some block on their designated path. This ensures that their new storage locations are independent of past access patterns.

Although ORAM has strong generality and a broad range of designs have been proposed to improve its practicality and bandwidth efficiency, it is often not the preferred solution for many

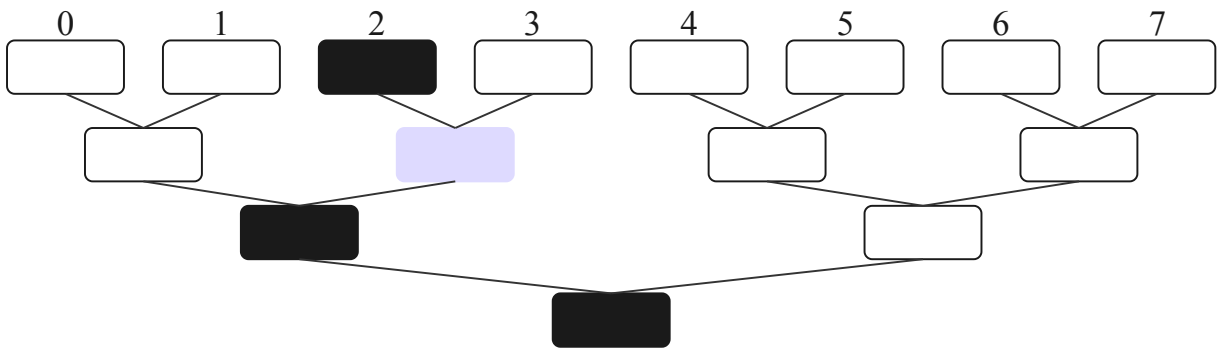

Figure 2.1: Example of an ORAM tree. Numbers at the top labels each leaf node, each of which uniquely defines a path from that leaf to the root. Violet denotes the target block associated with leaf label 2. Black blocks are not targeted but will be retrieved together with the violet block. Other blocks may or may not contain actual data.

applications. Firstly, ORAM-based constructions typically work under the EM model, requiring the existence of a non-constant amount of protected memory to maintain the position map and the stash. Access patterns to them are not oblivious, and making the program doubly-oblivious requires careful design and incurs more computational overhead. Moreover, each ORAM access incurs significant overhead, introducing large hidden constants that impact performance, despite having an adequate asymptotic cost [11], [25]. While more advanced designs have been proposed to reduce the communication bandwidth [38] and to make position map accesses oblivious [39], the inherent performance costs remain significant. Therefore, it is essential to explore algorithms for specific problems without relying on ORAM as a basic primitive.

2.5 Triangle Join Query

Although our work focuses on triangle join queries, it aims to highlight the key insights underlying general WCOJ problems. They both share the same core challenges, but triangle join queries provide a simpler setting. To that end, we first introduce the necessary definitions and descriptions

for the general WCOJ problems, and subsequently narrow down to triangle join queries. For a natural join query Q, we model it as a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, where each $v \in \mathcal{V}$ models an attribute and each $e \in \mathcal{E} \subseteq 2^{\mathcal{V}}$ has a corresponding relation R_e modeled by it. Let $\mathcal{H}_{\blacktriangle}$ denote the hypergraph for a triangle natural query $Q_{\blacktriangle} = R(A, B) \bowtie S(B, C) \bowtie T(A, C)$. As shown in Figure 2.2, $\mathcal{V}_{\blacktriangle} = \{A, B, C\}$ and $\mathcal{E}_{\blacktriangle} = \{\{A, B\}, \{B, C\}, \{A, C\}\}$. Without further specification, we will assume $|R| = |S| = |T| = N$ as the size of each input relation for $Q_{\blacktriangle}$.

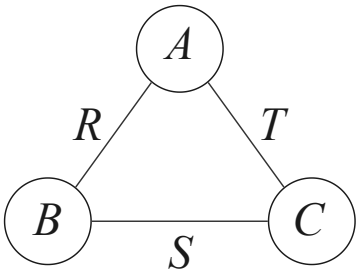

Figure 2.2: Graphical representation of the triangle join query structure. Each edge corresponds to a binary relation over a pair of attributes, visually modeling the query as a triangle-shaped graph.

2.5.1 AGM Bound

Understanding upper bounds on join output sizes is essential in guiding database operator designs. To build intuition for the size bound of the triangle join query, let us first consider a simple yet instructive example: the path join query, also refereed to as a chain join query, defined as

$$Q_P = T_1(A, B) \bowtie_B T_2(B, C) \bowtie_C T_3(C, D),$$

where $|T_1| = |T_2| = |T_3| = N$. In the worst-case scenarios for many-to-many joins, a tight upper bound for the size $|Q_P|$ is $O(N^3)$, perhaps computed by first computing the Cartesian product $T_1 \times T_2 \times T_3$ followed by selection on matching join keys. Moreover, we might similarly assume an upper bound of $O(N^3)$ for the triangle join query $Q_{\blacktriangle}$, but its structure inherently introduces

additional complexity that requires more careful examination.

Atserias, Grohe, and Marx [21] provided a tight characterization of the worst-case output size of natural join queries. They introduced a structural method for determining the maximum output size of a join query Q through the notion of a *fractional edge cover* of the query's associated hypergraph. Formally, given a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ modeling query Q, a vector $\mathbf{x} = (x_e)_{e \in \mathcal{E}}$ is called a fractional edge cover of $\mathcal{H}$ if it satisfies $\sum_{e:v \in e} x_e \geq 1$ for all $v \in \mathcal{V}$, and $x_e \geq 0$ for all $e \in \mathcal{E}$. Consequently, $|Q|$ is bounded by $\prod_{e \in \mathcal{E}} |R_e|^{x_e}$ for $x_e \in \mathbf{x}$ [23].

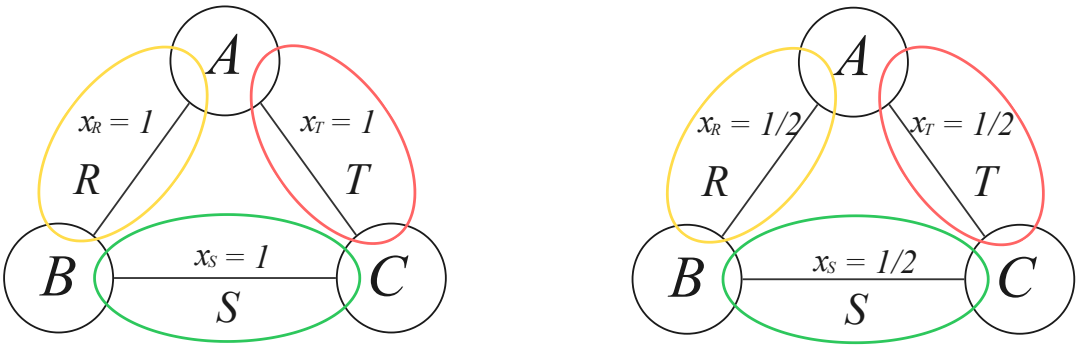

Figure 2.3: Example edge covers for $\mathcal{H}_\blacktriangle$.

Note that not all covers yield a tight bound. For instance, Figure 2.3 showcases two edge covers for $\mathcal{H}_\blacktriangle$ that models the triangle query $Q_\blacktriangle$. On the left side of Figure 2.3, $x_R = x_S = x_T = 1$ is a valid bound as for each vertex (e.g., A) the sum of the cover numbers of all edges connected to that vertex (e.g., $x_R + x_T = 2$ for edges $\{A, B\}$ and $\{A, C\}$) is greater than 1, and all cover numbers are non-negative, satisfying the conditions. This cover implies the bound $|Q| \leq |R| \cdot |S| \cdot |T| = N^3$, which is valid but not tight. Without loss of generality, as $R(A, B) \subseteq \pi_{(A,B)}(S(B, C) \bowtie T(A, C))$, a tighter bound N^2 can be inferred. To find an optimal fractional edge cover, one must solve the

following linear program:

$$\begin{aligned} \text{minimize} \quad & \sum_{e \in \mathcal{E}} x_e \\ \text{subject to} \quad & \sum_{e: v \in e} x_e \geq 1, \quad \forall v \in \mathcal{V}, \\ & x_e \geq 0, \quad \forall e \in \mathcal{E}. \end{aligned}$$

For $\mathcal{H}_{\blacktriangle}$, solving the linear program yields $\mathbf{x} = (1/2, 1/2, 1/2)$, corresponding to the second example shown in Figure 2.3. This cover results in the tightest upper bound $O(N^{3/2})$ for $|Q_{\blacktriangle}|$. More generally, the minimum objective value of the linear program, denoted $\rho^*(Q)$, provides a worst-case upper bound $O(N^{\rho^*(Q)})$ for any natural join query Q.

Ngo et al. [23] developed an inductive proof of the AGM bound based on a query decomposition lemma and provided an abundant set of illustrations. Although we omit the proof details and more detailed discussion, we present the query decomposition lemma here, as it offers valuable insights that will be discussed in later sections.

Lemma 2.5.1 (Query decomposition lemma [23])**.** *Let $Q = \ltimes_{e \in \mathcal{E}} R_e$ be a natural join query represented by a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, and $\mathbf{x} = (x_e)_{e \in \mathcal{E}}$ be any fractional edge cover for $\mathcal{H}$. Arbitrarily partition $\mathcal{V}$ into I and J such that $1 \leq |I| \leq |J|$. Let*

$$L = \ltimes_{e \in \mathcal{E}_I} \pi_I(R_e).$$

Then,

$$\sum_{\mathbf{t}_I \in L} \prod_{e \in \mathcal{E}_J} |R_e \ltimes \mathbf{t}_I|^{x_e} \leq \prod_{e \in \mathcal{E}} |R_e|^{x_e}.$$

To gain better insight into the triangle join query and Lemma 2.5.1, observe that Lemma 2.5.1 naturally matches the structure of the triangle join query. Without loss of generality, for query $Q_{\blacktriangle}$,

$R(A,B)$	
a_1	b_1
a_1	b_2
a_1	b_3
a_2	b_3
a_3	b_1
a_4	b_2

$S(B,C)$	
b_1	c_2
b_1	c_3
b_1	c_4
b_2	c_4
b_3	c_1
b_3	c_3

$T(A,C)$	
a_1	c_2
a_1	c_3
a_2	c_3
a_2	c_4
a_4	c_1
a_4	c_2

Figure 2.4: Example: (a_1, b_1) invokes sub-query $Q_\blacktriangle[(a_1, b_1)]$. Blue portions of S correspond to $S \ltimes b_1$, and violet portions of T correspond to $T \ltimes a_1$.

each $(a, b) \in R$ invokes a sub-query

$$Q_\blacktriangle[(a,b)] = \sigma_{A=a,B=b}R \bowtie \sigma_{B=b}S \bowtie \sigma_{A=a}T. \tag{2.1}$$

In terms of Lemma 2.5.1, $Q_\blacktriangle[(a, b)]$ matches the case where $I = \{A, B\}$ and $J = \{C\}$, which implies $\mathcal{E}_I = \{\{A, B\}, \{B, C\}, \{A, C\}\}$ and $\mathcal{E}_J = \{\{A, C\}, \{B, C\}\}$. In this case,

$$L = \{(a, b) : a \in \pi_A(R) \cap \pi_A(T), b \in \pi_B(R) \cap \pi_B(S)\}.$$

Iterating through each $(a, b) \in L$, $Q_\blacktriangle[(a, b)]$ is given by the Cartesian product[1]

$$Q_\blacktriangle[(a,b)] = (a,b) \times \{\pi_C(T \ltimes a) \cap \pi_C(S \ltimes b)\}.$$

Thus, Lemma 2.5.1 tells us that $\sum_{(a,b)\in L} |\pi_C(T \ltimes a)|^{1/2} \cdot |\pi_C(S \ltimes b)|^{1/2} \leq N^{3/2}$.

For example, Figure 2.4 illustrates an instance of input relations. The tuple (a_1, b_1) invokes the evaluation of sub-query $Q_\blacktriangle[(a_1, b_1)]$. By taking intersections between the corresponding sets

[1]We intentionally align the notation with Lemma 2.5.1 here, but $T \ltimes a = \sigma_{A=a}T$ and $S \ltimes b = \sigma_{B=b}S$.

of c_i's, we obtain the output $Q_{\blacktriangle}[(a_1, b_1)] = \{(a_1, b_1, c_2), (a_1, b_1, c_3)\}$.

$R(A, B)$	
a_1	b_1
a_1	b_2
a_1	b_3
a_2	b_3
a_3	b_1
a_4	b_2

$S(B, C)$	
b_1	c_2
b_1	c_3
b_1	c_4
b_2	c_4
b_3	c_1
b_3	c_3

$T(A, C)$	
a_1	c_2
a_1	c_3
a_2	c_3
a_2	c_4
a_4	c_1
a_4	c_2

Figure 2.5: Example: a_1 invokes sub-query $Q_{\blacktriangle}[a_1]$. Orange portions of S correspond to $\pi_B(R \ltimes a_1) \ltimes S$ and pink portions of S correspond to $\pi_C(T \ltimes a_1) \ltimes S$.

Similarly, suppose we select $I = \{A\}$ and $J = \{B, C\}$. Then, $L = \pi_A(R) \cap \pi_A(T)$. In this case, each $a \in \pi_A(R) \cap \pi_A(T)$ invokes sub-query

$$Q_{\blacktriangle}[a] = \sigma_{A=a} R \bowtie S \bowtie \sigma_{A=a} T, \tag{2.2}$$

which is computed as

$$Q_{\blacktriangle}[a] = a \times ((S \ltimes \pi_B(R \ltimes a)) \cap (S \ltimes \pi_C(T \ltimes a))).$$

Then, by Lemma 2.5.1, we have $\sum_{a \in L} |\pi_B(R \ltimes a)|^{1/2} \cdot |S|^{1/2} \cdot |\pi_C(T \ltimes a)|^{1/2} \leq N^{3/2}$.

For the example shown in Figure 2.5, since the overlapped portions in S are $\{(b_1, c_2), (b_1, c_3), (b_3, c_3)\}$, the output of the sub-query is $Q_{\blacktriangle}[a_1] = \{(a_1, b_1, c_2), (a_1, b_1, c_3), (a_1, b_3, c_3)\}$.

2.5.2 Sub-Optimality of Pair-wise Join Strategy

Let us revisit the path join query Q_P. In practice, query optimizers typically select efficient join plans and construct corresponding join trees, such as $(T_1(A, B) \bowtie T_2(B, C)) \bowtie T_3(C, D)$, which

result in no worse than $O(N^3)$ performance if following a reasonable algorithm such as the Yannakakis algorithm [40]. Also, $T_1(A, B) \bowtie T_2(B, C)$ is tightly bounded by $O(N^2)$, which is less than the worst-case bound for Q_P. Thus, such a pair-wise plan is not suboptimal in this case. However, when applied to a triangle join query, pair-wise join plans inevitably cause a blowup in the size of the intermediate result compared to $|Q_{\blacktriangle}| \leq N^{3/2} \leq N^2$.

The worst-case upper bound for $Q_{\blacktriangle}$ implies that an optimal algorithm should achieve a worst-case running time of $O(N^{3/2})$. Since, under any pair-wise join plan, the worst-case running time is tightly bounded by $O(N^2)$, which implies that the pair-wise join strategy is inherently suboptimal in computing triangle queries. Interested readers may refer to [23] for more detailed explanations and illustrative examples. To bridge the gap, several algorithms have been proposed that inherently join multiple relations simultaneously or, in other words, attribute-at-a-time. Ngo, Porat, Ré, and Rudra [24] proposed a WCOJ algorithm, often referred to as `NPRR`, that partitions tuples based on degrees to match the AGM bound. Veldhuizen [22] presented `Leapfrog Triejoin`, which leverages the trie structure to enumerate join results one attribute at a time. Soon after that, Ngo et al. [23] presented `Generic Join`, a general WCOJ algorithm that unifies previous ideas.

CHAPTER 3

PROBLEM OVERVIEW

In this chapter, we begin by defining the problem we will address. Next, we describe insecure triangle join algorithms, from which we later develop their oblivious counterparts. We then analyze the core challenges in designing efficient triangle join algorithms. After reviewing recent works on oblivious WCOJ algorithms, we provide a high-level overview of the key ideas behind our approach.

3.1 Problem Definition

Given input relations $R(A, B)$, $S(B, C)$, and $T(A, C)$ each of size N. Our goal is to evaluate the triangle natural join query

$$Q_{\blacktriangle} = R(A, B) \bowtie S(B, C) \bowtie T(A, C),$$

with WCOJ algorithms that have runtime matching the worst-case upper bound $O(N^{3/2})$ up to a poly-logarithmic factor and satisfy Definition 2.3.1 of (ϵ, δ)-DO at level II. The output that our (ϵ, δ)-DO algorithms generate is the multi-set

$$Q_{\blacktriangle} = \{(a, b, c) | (a, b) \in R, (b, c) \in S, (a, c) \in T\},$$

which is padded with a randomized number of noise (dummy tuples), where the noise is sampled from the truncated and shifted geometric distribution $\mathcal{G}(\epsilon, \delta, \Delta) = O(\frac{\Delta}{\epsilon} \log \frac{1}{\delta})$ as defined in

Definition 2.3.4. We denote the output padded with noise by $\hat{Q}_{\blacktriangle}$.

3.2 Non-Secure WCOJ Algorithms for Triangle Joins

In this section, we will introduce insecure WCOJ counterparts that we will adapt to oblivious settings. Veldhuizen [22] introduced `Leapfrog Triejoin` (`LFTJ`), which utilizes the trie presentations of relations and the leapfrog join to evaluate cyclic queries. Its asymptotic runtime matches the worst-case output size up to a logarithmic factor, aligning with the AGM bound. Notably, the algorithm was implemented in the commercial database LogicBlox prior to its formal publication [22]. Inspired by Veldhuizen's work, Ngo et al. [23] surveyed the existing WCOJ algorithms and provided a unified theoretical analysis for their key insights. As part of their work, they presented two algorithms for evaluating triangle join queries, both of which achieve the asymptotically optimal bound of $O(N^{3/2})$. In particular, many existing WCOJ algorithms are instantiations of Algorithm 2. We will briefly outline the theoretical principles underlying these algorithms and examine the sources of information leakages that violate obliviousness.

3.2.1 Computing $Q_{\blacktriangle}$ with Power of Two Choices

Analysis of Algorithm 1. Ngo et al. [23] demonstrated that the size blowup in the intermediate result size is attributed to the skew in the degrees of join keys. Certain join keys may contribute disproportionately to the intermediate results compared to their contribution to the final output, essentially generating many unnecessary tuples. To address the skew, they proposed employing separate strategies for join keys with high and low skew, referred to as *heavy* and *light* join keys, respectively. To be specific, for a join key $a \in \pi_A(R) \cap \pi_A(T)$, its degree in the intermediate result $R \bowtie T$ is given by $|\sigma_{A=a}(R \bowtie T)| = |\sigma_{A=a}R| \cdot |\sigma_{A=a}T|$, while its contribution to the final output is $|\sigma_{A=a}Q_{\blacktriangle}|$. A join key is said to be heavy if $|\sigma_{A=a}R| \cdot |\sigma_{A=a}T| \geq |\sigma_{A=a}Q_{\blacktriangle}|$ and light otherwise.

However, despite the strategies, the actual contribution to the final join result is unknown until we have $Q_\blacktriangle$ ready. To address this, Ngo et al. used $|S|$ as a heuristic for estimating $|\sigma_{A=a}Q_\blacktriangle|$ and proved that this approximation is valid. Then, the two choices are as follows.

(1) If join key a is heavy (lines 3-8), for each $(b, c) \in S$, we check the presence of $(a, b) \in R$ and $(a, c) \in T$.

(2) If a is light (lines 10-15), then for each tuple (b, c) in $(\pi_B(\sigma_{A=a}R)) \times (\pi_C(\sigma_{A=a}T))$, we check if $(b, c) \in S$.

Algorithm 1 Computing $Q_\blacktriangle$ with power of two choices.

Input: $R(A, B)$, $S(B, C)$, $T(A, C)$ sorted on $\langle A, B, C\rangle$

1: $Q_\blacktriangle \leftarrow \emptyset$
2: **for** each $a \in \pi_A(R) \cap \pi_A(T)$ **do**
3: **if** $|\sigma_{A=a}R| \cdot |\sigma_{A=a}T| \geq |S|$ **then** ▷ Heavy join keys
4: **for** each $(b, c) \in S$ **do**
5: **if** $(a, b) \in R$ and $(a, c) \in T$ **then** ▷ If match exists on both sides
6: Add (a, b, c) to $Q_\blacktriangle$
7: **end if**
8: **end for**
9: **else**
10: **for** each $(b, c) \in (\pi_B(\sigma_{A=a}(R)) \times \pi_C(\sigma_{A=a}(T)))$ **do** ▷ Light join keys
11: **if** $(b, c) \in S$ **then** ▷ If two sides are connected by S
12: Add (a, b, c) to $Q_\blacktriangle$
13: **end if**
14: **end for**
15: **end if**
16: **end for**
17: **return** $Q_\blacktriangle$

Leakages in Algorithm 1. To achieve level II obliviousness, an algorithm should have identical and constant accesses that are independent of data values. Algorithm 1 leaks the following sensitive information [25]:

- The size $|\pi_A(R) \cap \pi_A(T)|$ through line 2;
- The number of heavy and light join keys, as well as the skew of each join key through the conditional branches in lines 3 and 9;
- $|\pi_B(\sigma_{A=a}(R))|$, $|\pi_C(\sigma_{A=a}(T))|$, and $|\pi_B(\sigma_{A=a}(R)) \times \pi_C(\sigma_{A=a}(T))|$ through line 10.

3.2.2 Computing $Q_\blacktriangle$ by Delaying the Computation

Analysis of Algorithm 2. Algorithm 2 looks more straightforward and concise. It clearly demonstrates the attribute-at-a-time strategy. For each join key $a \in \pi_A(R) \cap \pi_A(T)$, the algorithm finds its corresponding subset $\sigma_{A=a}R$ and ensures each $b \in \pi_B(\sigma_{A=a}R)$ exists in $\pi_B(S)$ (line 3). It then computes the intersection of the (a, b)-pair's associated c values to finalize the join. In terms of SQL, line 2 enforces the predicate `R.a = T.a`, line 3 corresponds to `R.b = S.b`, and line 4 satisfies `S.c = T.c`. From the perspective of skew, if the join key $a \in \pi_A(R) \cap \pi_A(T)$ is heavy, then candidates b and c progressively narrow the space to the necessary final output [23]. In addition, one may also find out that Algorithm 2 matches Equations 2.1 and 2.2.

Algorithm 2 Computing $Q_\blacktriangle$ by delaying computation.

Input: $R(A, B)$, $S(B, C)$, $T(A, C)$ sorted on $\langle A, B, C\rangle$

```
1: Q▲ ← ∅
2: for each a ∈ π_A(R) ∩ π_A(T) do
3:     for each b ∈ π_B(σ_{A=a}R) ∩ π_B(S) do
4:         for each c ∈ π_C(σ_{A=a}T) ∩ π_C(σ_{B=b}S) do
5:             Add (a, b, c) to Q▲
6:         end for
7:     end for
8: end for
9: return Q▲
```

Leakages in Algorithm 2. Algorithm 2 reveals the following sensitive information [25]:

- $|\pi_A(R) \cap \pi_A(T)|$ is leaked by the for-loop in line 2;
- $|\pi_B(\sigma_{A=a}R)|$ and $|\pi_B(\sigma_{A=a}R) \cap \pi_B(S)|$ are leaked by the intersection operation and for-loop in line 3;
- $|\pi_C(\sigma_{A=a}T)|$, $|\pi_C(\sigma_{B=b}S)|$, and $|\pi_C(\sigma_{A=a}T) \cap \pi_C(\sigma_{B=b}S)|$, are leaked by the selection and intersection operations and for-loop in line 4.

3.2.3 Leapfrog Triejoin

We talk about the `LFTJ` after Algorithm 2 because it is an instantiation of Algorithm 2 [23].[1] Below, we provide a high-level description of its strategy. `LFTJ` primarily consists of two components: trie presentation of relations and leapfrog join. We first introduce each of them separately and then describe how joins are computed.

Trie Presentation. For each relation, the algorithm constructs a *trie iterator*, which can be viewed as a search tree where each level stores the values of an attribute, and the values of the corresponding attribute are in sorted order at each level. For instance, given a relation $R(A, B)$, Figure 3.1 shows its corresponding trie presentation, which consists of two levels (excluding the root), and each tuple forms a unique path from the root to a leaf [22]. Using trie iterators, the algorithm can efficiently navigate the attribute during evaluation. In the context of Algorithm 2, for each $a \in \pi_A(R) \cap \pi_A(T)$, to retrieve the associated values $b \in \pi_B(\sigma_{A=a}R)$ and $c \in \pi_C(\sigma_{A=a}T)$, the algorithm proceeds the iterator to the next level of the current node.

Leapfrog Join. During the initialization of the trie iterators, the algorithm constructs a *leapfrog join* instance for each attribute. The instance maintains a leapfrog iterator for every relation that

[1] Although we discuss `LFTJ` in the context of triangle joins, it is a general-purpose WCOJ algorithm capable of evaluating any conjunctive join query, not limited to triangle ones. In addition, `NPRR` is an instantiation of Algorithm 1, but we do not discuss it in detail here.

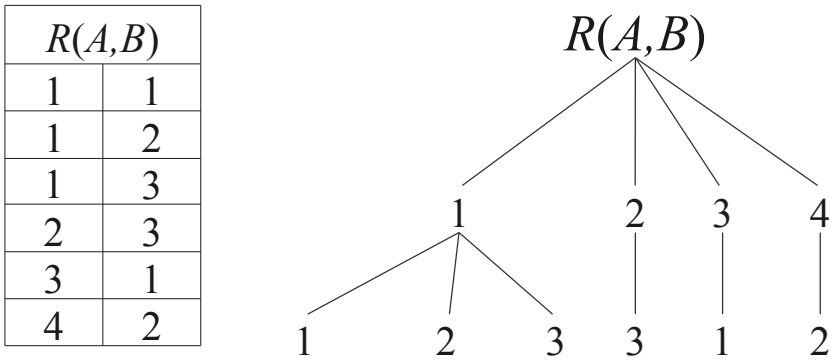

Figure 3.1: Example trie presentation of a relation $R(A, B)$.

contains the attribute. Intuitively, a leapfrog iterator traverses a single level of the trie horizontally, which essentially computes the intersections in Algorithm 2. The leapfrog join operates analogously to the sort-merge join over unary relations: an attribute's iterators for corresponding relations advance in tandem to identify matched elements.

In terms of Algorithm 2, to compute $\pi_A(R) \cap \pi_A(T)$, the leapfrog join simultaneously advances the iterators for attribute A in both R and T to find the first matching value $a_0 \in \pi_A(R) \cap \pi_A(T)$. Upon finding the match, the trie iterators for R and T navigate to the next level of the trie to probe the subtrees. Now, the leapfrog iterators for B proceed to find the first matching value $b_0 \in \pi_B(\sigma_{A=a}R) \cap \pi_B(S)$. This process repeats until all trie iterators reach the deepest level and matching c values are found. The algorithm then generates output tuples associated with the prefix (a_0, b_0). Overall, `LFTJ` achieves time complexity of $O(N^{3/2} \cdot \log N)$ in evaluating triangle join queries, where the logarithmic term comes from the amortized cost of the leapfrog join.

3.3 On the Intricacies of Making WCOJ Oblivious

While WCOJ algorithms offer provable asymptotic advantages over traditional pair-wise strategies, adapting these algorithms to an oblivious setting presents significant challenges. As demonstrated

by existing implementations of WCOJ operators [22], [41]–[45], even deriving insecure WCOJ algorithms that match their theoretical performance already requires carefully engineered designs. Due to the column-wise strategies of WCOJ algorithms, their implementations depend on specialized indices to navigate between attributes and retrieve corresponding tuples, which imposes non-trivial computational and storage overhead [41]. Systems that support WCOJ operations either rely on precomputed, read-only indices [42], [43] or yield orders of magnitude slower performances if they prefer mutable data [46]. Although the recent adoption of `Generic Join` [41] outperforms the earlier systems, it also heavily relies on low-level primitives, such as hash tables, trie iterators, and dynamic memory layouts, to achieve the desired performance and flexibility. However, while these techniques are effective in insecure settings, they are infeasible in the oblivious setting due to their data-dependent interactions with memory. For instance, hashing is often avoided in oblivious algorithms as it may leak value distributions and access frequencies.

To better understand the necessity of the above techniques, it is helpful to examine the assumptions underlying Algorithm 1 and 2. Let $L_A = \pi_A(R) \cap \pi_A(T)$ and $L_B = \pi_B(R) \cap \pi_B(S)$. In short, the proof of Algorithm 1's optimality presented by Ngo et al. [23] is as follows:

$$\sum_{a \in L_A} \min\{|\sigma_{A=a}R| \cdot |\sigma_{A=a}T|, |S|\} \tag{3.1}$$

$$\leq \sum_{a \in L_A} \sqrt{|\sigma_{A=a}R| \cdot |\sigma_{A=a}T| \cdot |S|}$$

$$\leq \sqrt{|S|} \cdot \sqrt{\sum_{a \in L_A} |\sigma_{A=a}R|} \cdot \sqrt{\sum_{a \in L_A} |\sigma_{A=a}T|} \qquad \text{[by the Cauchy-Schwarz inequality]} \tag{3.2}$$

$$\leq \sqrt{|R|} \cdot \sqrt{|S|} \cdot \sqrt{|T|}$$

$$\leq N^{3/2} \qquad \text{[by Lemma 2.5.1].}$$

The proof inherently assumes that the time spent is constant on checking a tuple's presence in a relation (lines 5 and 11) and retrieving a tuple associated with $a \in L$ (line 10).

In addition, the optimality of Algorithm 2 relies on the assumption that the cost of computing intersections is equivalent to the minimum size of the two input sets [23]. A brief sketch of the proof is as follows:

$$\sum_{a \in L_A} \sum_{b \in L_B} \min\{|\pi_C(\sigma_{A=a}T)|, |\pi_C(\sigma_{B=b}S)|\} \tag{3.3}$$

$$\leq \sum_{a \in L_A} \sum_{b \in L_B} \sqrt{|\pi_C(\sigma_{A=a}T)| \cdot |\pi_C(\sigma_{B=b}S)|}$$

$$\leq \sum_{a \in L_A} \sqrt{|S|} \cdot \sqrt{|\pi_C(\sigma_{A=a}T)|} \cdot \sqrt{|\pi_B(\sigma_{A=a}R)|} \quad \text{[by the Cauchy-Schwarz inequality]}$$

$$\leq \sqrt{|R|} \cdot \sqrt{|S|} \cdot \sqrt{|T|}$$

$$\leq N^{3/2} \quad \text{[by Lemma 2.5.1].} \tag{3.4}$$

These assumptions are practical in an insecure setting, as Freitag et al. [41] leverage nested hash tables to compute intersections and efficiently retrieve tuples. However, typical oblivious counterparts for intersection, selection, or semi-join reduction are based on oblivious sort [14], [47], [48] or cryptographic protocols [49] that incur a linear runtime proportional to the total input size, up to a logarithmic factor. As a result, the overall runtime can be proportional to $O(N^2)$, equivalent to pair-wise join strategies.

3.4 Naïve Oblivious WCOJ Algorithms

As discussed earlier, ORAM enables us to make arbitrary programs singly-oblivious. It can be applied to achieve both full obliviousness and (ϵ, δ)-DO at level I, depending on how the program

is constructed. We briefly describe possible ORAM-based constructions of WCOJ algorithms that satisfy these respective notions at the level of singly-obliviousness, as they provide great insights into understanding DO WCOJ algorithmic constructions. For our discussion purposes and consistency with the rest of the paper, we evaluate the performance of these constructions under the RAM model as the total number of accesses to ORAM.[2]

Naïve full oblivious WCOJ algorithm. Typical constructions of oblivious programs simulate insecure algorithms with ORAM. Here, we briefly describe a simulation of `LFTJ` with Path ORAM [10]. Recall that the `LFTJ` builds a trie presentation for each relation and a leapfrog join instance for each attribute, both of which are implemented as iterator interfaces backed by B-tree-like indices. To make the algorithm oblivious, we integrate B-tree indices into the Path ORAM [17] and store the index tree for each relation separately in its own ORAM tree. Specifically, each index node of a B-tree is stored in a block within the ORAM tree, along with its children's block identifiers and position tags. This design allows us to traverse B-trees and advance iterators without position maps. However, each time retrieved nodes are relocated for shuffling, we recursively update position tags stored in their ancestors, which introduces an overhead of $O(\log^2 N)$ in the context of Path ORAM [50].

To prevent leakage of access intent across relations, we retrieve tuples from ORAM trees in a round-robin fashion; when accessing a target block from one relation, we perform dummy retrievals for the remaining ORAM trees. The internal join logic follows that of `LFTJ`. To hide the true output size, we continue dummy accesses after all output tuples are obtained and pad the output result to the worst-case size $N^{3/2}$. The overall cost will be $O((N^{3/2} \log N + N) \log^2 N)$, and the output length, including dummy elements, is $\Theta(N^{3/2})$.

Naïve differential oblivious WCOJ Algorithm The core construction is the same to the naïve

[2]While ORAM is often employed under the EM model and assumes the existence of a client to handle computations privately, our discussion abstracts away the details of client-side executions.

full oblivious WCOJ algorithm above. However, in practice, particularly for large databases, triangles can be sparse, resulting in a true cardinality of the output that is significantly smaller than the worst-case size [51]–[53]. Consider the above simulation without the extra dummy accesses or dummy padding at the end. (We still access round-robin, but stop once we obtain all output tuples.) In this case, as ORAM conceals all the access patterns that happen in the middle, the only leakages are the total number of accesses and output length $|Q_\blacktriangle|$. To prevent such leakages, we follow the standard differential obliviousness approach and add noise to the total number of accesses and output length.[3] Specifically, we append $\mathcal{U}$ dummy tuples to the output, where $\mathcal{U}$ is sampled from $\mathcal{G}(\epsilon, \delta, N) = O(\frac{N}{\epsilon} \log \frac{1}{\delta})$. During this, we keep performing dummy accesses to ORAM until the total number of accesses matches the need for generating $|Q_\blacktriangle| + \mathcal{U}$ output tuples. Note that we sample noise $\mathcal{G}(\epsilon, \delta, N)$ for $\Delta = N$ because the global sensitivity is upper bounded by N. Although the work [26] considers the context of binary join, the same sensitivity argument applies here. The resulting algorithm incurs a total cost is $O((|Q_\blacktriangle| \log N + N + \mathcal{U}) \log^2 N)$, and produces an output of size $O(|Q_\blacktriangle| + \mathcal{U})$. For simplicity, with typical parameter choices, we have $O(\frac{1}{\epsilon} \log \frac{1}{\delta}) = O(\log N)$, then $\mathcal{U} = N \log N$.

Theorem 3.4.1 (Naïve DO WCOJ algorithm)**.** *There exists an algorithm that satisfies* $(\epsilon, \delta + 3 \cdot \mathtt{negl}(N))$*-differential obliviousness for* $\mathtt{negl}(\cdot)$ *being a negligible function. Furthermore, suppose* $\epsilon = \Theta(1)$ *and* $\delta = 1/\mathtt{poly}(N)$*, where* $\mathtt{poly}(\cdot)$ *is a polynomial function. Then, the naïve DO WCOJ algorithm achieves* $O((|Q_\blacktriangle| \log N + N \log N) \log^2 N)$ *runtime with output length of* $O(|Q_\blacktriangle| + N \log N)$.

Proof. The runtime is straightforward following the algorithm construction. We need $O(|Q_\blacktriangle| \log N + N \log N)$ number of accesses in total, and each access incurs an overhead of $O(\log^2 N)$. Note that

[3] Since leakages consist of output size and total number of accesses, padding noise is essentially making the output differentially private.

we omit the term $O(N)$, the number of accesses for input.

Let $\hat{Q}_{\blacktriangle}$ denote output $Q_{\blacktriangle}$ padded with noise. Then, $|\hat{Q}_{\blacktriangle}| = |Q_{\blacktriangle}| + O(N \log N)$. Recall that the access pattern with ORAM is simulatable given the input size $O(N)$ and output size $|\hat{Q}_{\blacktriangle}|$, and each ORAM independently can fail with $\texttt{negl}(N)$ probability. Then, jointly, the full memory access pattern is simulatable up to a statistical distance at most $3 \cdot \texttt{negl}(N)$. Moreover, consider the neighboring inputs (R, S, T) and (R', S', T'). Changing one element results in at most N changes in the output (the global sensitivity), i.e., $|Q_{\blacktriangle}(R, S, T)| - |Q_{\blacktriangle}(R', S', T')| \leq N$, where $|Q_{\blacktriangle}(\cdot)|$ denotes the true output size on the given input. By Fact 2.3.5, we have

$$|\hat{Q}_{\blacktriangle}(R, S, T)| \sim_{(\epsilon,\delta)} |\hat{Q}_{\blacktriangle}(R', S', T')|.$$

As it is clear the leakage $|\hat{Q}_{\blacktriangle}|$ satisfies (ϵ, δ)-differential privacy, by Lemma 2.3.7, the algorithm is $(\epsilon, \delta + 3 \cdot \texttt{negl}(N))$-differential oblivious. □

3.5 Overview of Approach

Our objective is to design triangle join algorithms that achieve worst-case optimality and satisfy (ϵ, δ)-DO. Although insecure WCOJ algorithms rely on specialized indices and dynamic memory layouts that are generally incompatible with oblivious algorithm designs, we provide reformulations of existing insecure WCOJ algorithms to leverage provable, ORAM-free, oblivious primitives to ensure access pattern privacy. While conventional pair-wise join plans are suboptimal in worst-case optimality, this does not rule out the possibility of adopting binary join algorithms for solving WCOJ problems. Our approaches, therefore, build on this observation and leverage a recently proposed DO binary join primitive [26] to obtain instance-specific performance while ensuring meaningful privacy guarantees.

We present two algorithms that are DO counterparts of Algorithm 1 and 2, respectively. By re-ordering computation steps, we can reformulate both algorithms into equivalent binary-join-based ones while preserving their worst-case optimality. Such reformulations incur little computational overhead as they directly follow the logic underlying the proofs of Algorithm 1 and 2. With the reformulations, we leverage the DO binary join, acting as the core primitive, to build intermediate relations of candidate tuples. We then exploit a fully oblivious semi-join primitive to retrieve valid outputs, which is based on an $O(N \log^2 N)$ sorting algorithm. Finally, we compact the output and pad a randomized number of noise to hide the true output length effectively. Both of our constructions achieve runtime matching the AGM bound $O(N^{3/2})$ up to poly-logarithmic factors, and they can be practically reduced to logarithmic factors by allowing a certain depth of parallelism for sorting.

Specifically, in the case of Algorithm 1, we begin by partitioning join keys $a \in \pi_A(R) \cap \pi_A(T)$ into heavy and light. For light join keys, we binary join corresponding tuples in R and T, and then verify whether the resulting (b, c) pairs exist in S using the semi-join. For heavy join keys, defined as $W_h = \{a \in \pi_A(R) \cap \pi_A(T) : |\sigma_{A=a}R| \cdot |\sigma_{A=a}T| \geq |S|\}$, we compute the Cartesian product $L'_h = W_h \times S$, and validate that each (a, b) and (a, c) pair also appears in R and T, respectively.

For Algorithm 2, let $K_1 = \{(a, b) \in R : |\pi_C(\sigma_{A=a}T)| \geq |\pi_C(\sigma_{B=b}S)|\}$ and $K_2 = R \setminus K_1$. We compute $L'_1 = K_1 \bowtie_B S$ and $L'_2 = K_2 \bowtie_A T$ to obtain initial candidates. We then perform semi-joins to check if each $(a, c) \in L'_1$ and each $(b, c) \in L'_2$ also appear in T and S, respectively, to obtain the valid outputs.

CHAPTER 4
ALGORITHM DESCRIPTION

In this chapter, we formally present our DO WCOJ algorithms for evaluating the triangle join query $Q_{\blacktriangle}$. We begin by introducing the core primitives we adopt from existing works, which serve as building blocks of our algorithms. Among these, a DO binary join algorithm plays a central role: it allows our constructions to preserve access pattern privacy while retaining instance-specific performance, eliminating the need to pad outputs and intermediate results to the worst-case bound. Following that, we detail our two algorithmic constructions and formally analyze their security and performance characteristics.

4.1 Oblivious Primitives

We introduce several important oblivious primitives that serve as building blocks for our algorithms. All the following algorithms satisfy either full obliviousness or DO at level II.

Sort [47], [54], [55]. Given an input array R of size N and a key order of chosen attributes $\langle X, Y, Z\rangle$, the primitive $\texttt{Sort}(R, \langle X, Y, Z\rangle)$ outputs the array R sorted in increasing order by that key. Although sorting algorithms with $O(N \log N)$ time complexity exist, most of them require a certain amount of private memory or depth of parallelization. We take the practical assumption that our oblivious sorting algorithm, without parallelization, achieves runtime $O(N \log^2 N)$ using $O(1)$ private memory in the RAM model. The oblivious sorting will serve as a fundamental building block for many other primitives we employ for our algorithms.

Augment [11]. Given as inputs relations R and S with sharing join key $\langle X\rangle$. By a constant number of sorts and linear scans, $\texttt{Augment}(R, S, \langle X\rangle, \{0, 1\})$ computes multiplicity annotations for each

tuple $t \in R \cup S$. Specifically, it appends a pair (α_t^R, α_t^S) to t, where α_t^R and α_t^S denote the count of $t[X]$ in R and S, respectively. An additional boolean flag controls whether the output retains α_t^R. The output retains only α_t^S if the flat is set to 0. In the end, we only keep tuples of R with needed counts. As the sorting step dominates the runtime, `Augment` runs in $O(N \log^2 N)$ time w.r.t. the input size.

Semi-join. Given two input relations R, S and join key attributes $\langle X \rangle$, $\texttt{SemiJoin}(R, S, \langle X \rangle)$ outputs the subset R that have matching keys in S, i.e., retaining all $t \in R$ such that $t[X] \in \pi_X(R) \cap \pi_X(S)$.

De-duplicate. Given an input array R of length N in sorted order on some attribute $\langle X \rangle$, primitive $\texttt{Deduplicate}(R, \langle X \rangle)$ returns a new array $R' \subseteq R$ that retains distinct values of $\langle X \rangle$, while preserving length N by padding dummy elements. Since we assume the input array is sorted, this can be done in $O(N)$ time.

Compact [56], [57]. Given an input array of length N with some elements marked as distinguished. The goal is to rearrange the array such that all distinguished elements appear at the front. This can be achieved in $O(N \log N)$ time. The output will be padded with noise $\mathcal{U} := \max \mathcal{G}(\epsilon, \delta, 2\mu)$ to maintain differential privacy, where μ is the maximum true multiplicity of any join key. The output length is at most the true output size plus $O(\frac{1}{\epsilon}(\mu + \frac{1}{\epsilon}\log\frac{1}{\delta}) \cdot \log\frac{1}{\delta})$ [26, Theorem 4.7]. Typical choices of parameters, $\epsilon = \Theta(1)$ and $\delta = 1/N^c$ for some constant $c \geq 1$, result in $O((\mu + \log N)\log N)$ overhead. The primitive satisfies (ϵ, δ)-DO.

Here, we also include introductions to essential subroutines within a later introduced DO binary join primitive. We defer detailed complexity discussions for them and will analyze them as part of the binary join primitive.

Send-receive[1] **[26], [58], [59].** Given as inputs relations D of size N_d and S of size N_s. Each tuple

[1]The Send-receive's mechanism may be referred to as oblivious routing or distribution. We adopt the primitive from [26] and follow the naming style to distinguish it from common connotations of routing in algorithm literature.

in S holds a distinct join key k_i and an associated value v_i, and each tuple in D holds a join key k_i. The goal is to allow each tuple in D with join key k_i to retrieve the corresponding value v_i from S. Informally, this process is analogous to a primary-foreign key join: the relation S serves as a table of primary keys with associated payloads, and D acts as the foreign key relation. Each key in S may match multiple elements in D, but each element in D receives exactly one value from S, based on key equality. This functionality is implemented using a constant number of oblivious sorts on the concatenated array $S \cup D$.

Bin placement [26]. Given an input array R, where each element (real or dummy) is labeled with a bin identifier $i \in \{1, \ldots, m\}$, and target bins with capacities $\{l_1, l_2, \ldots, l_m\}$. The goal is to move (route) each element to its destined bin. It is promised that each bin receives elements no more than its capacity. After all routings are finished, each bin may be padded with dummy elements to ensure all bins are full. The output is a concatenation of all resulting bins. This is based on a constant number of sorts.

4.1.1 Differentially Oblivious Binary Join Primitive

We briefly introduce the binary join primitive `DOBinaryJoin` [26], which we adopt as a building block in our work. The algorithm achieves (ϵ, δ)-DO by carefully injecting noise into the multiplicities of join keys, thereby providing meaningful privacy guarantees while avoiding the inefficiency of padding results to worst-case bounds. A detailed running example illustrating this construction is provided in Appendix A.1.

Given two input relations $R(A, B)$ and $T(A, C)$, each of length N. The algorithm proceeds through several stages to compute $R \bowtie_A T$ as described below.

Load array construction. The algorithm begins by constructing a load array L that contains each distinct join key $a_i \in \pi_A(R) \cap \pi_A(T)$, along with their noisy multiplicities $(\hat{\alpha}^R_{a_i}, \hat{\alpha}^T_{a_i})$, obtained by

adding sampled noise to true counts of a_i in R and T, respectively. The noise is sampled from the truncated and shifted distribution $\mathcal{G}(\epsilon, \delta, \Delta)$, ensuring they are non-negative and differentially private. L is constructed using the modified `Augment` primitive. To ensure the obliviousness, it is padded with dummy elements to maintain the length of $|R| + |T|$ and is *sorted on* $(\hat{\alpha}^R_{a_i}, \hat{\alpha}^T_{a_i})$.

Binning elements. Given L sorted on $(\hat{\alpha}^R_{a_i}, \hat{\alpha}^T_{a_i})$, each element in R and T is assigned a bin index based on its join key's index in L. The algorithm employs the `SendReceive` algorithm to distribute bin indices to all associated tuples. Elements are then routed to their destined bins using `BinPlacement`, where each bin's capacity is prescribed by the corresponding join key's noisy multiplicity. For each join key, a pair of bins is created: one containing tuples from R and one for those from T, each padded with dummy elements to its full capacity.

Bin-wise Cartesian product. Pairs of bins for elements in R and T are joined via Cartesian product to generate all possible combinations between tuples from R and T. Dummy tuples are generated when real matches are missing.

Compaction. The resulting output, a concatenation of bin-wise joins, is then compacted to the length $|R \bowtie_A T| + \mathcal{U}$ for $\mathcal{U} \coloneqq \max \mathcal{G}(\epsilon, \delta, 2\mu)$, where μ is the maximum true multiplicity of any join key.

We adopt our assumption for sorting algorithms, under which the algorithm achieves run time $O(|R \bowtie_A T| + N(\log^2 N + \frac{1}{\epsilon} \log \frac{1}{\delta}))$ and output length $O(|R \bowtie_A T| + \frac{1}{\epsilon}(\mu + \frac{1}{\epsilon} \log \frac{1}{\delta}) \log \frac{1}{\delta})$ [26]. With typical parameter choices, $\epsilon = \Theta(1)$ and $\delta = 1/N^c$ for some constant $c \geq 1$, the algorithm has runtime $O(|R \bowtie_A T| + N \log^2 N)$, and produces output of length $O(|R \bowtie_A T| + (\mu + \log N) \log N)$.

Primitive	Obliviousness Model	Time complexity	
		With parallelization	Without parallelization
Sort Augment Semi-join De-duplicate	FO	$O(N \log N)$	$O(N \log^2 N)$
		—	$O(N)$
Compact DO Binary Join	DO	$O(N)$ $O(\texttt{OUT} + N \log N)$	$O(N \log N)$ $O(\texttt{OUT} + N \log^2 N)$

Table 4.1: Time complexity of oblivious primitives under different models and parallelization assumptions. All primitives satisfy level II obliviousness. OUT denotes the size of true output.

4.2 Differentially Oblivious Triangle Join Algorithms

In this section, we formally present two DO triangle join algorithms that evaluate query

$$Q_{\blacktriangle} = R(A, B) \bowtie S(B, C) \bowtie T(A, C).$$

Both algorithms are designed to satisfy (ϵ, δ)-DO. As a result, they produce an output denoted by $\hat{Q}_{\blacktriangle}$, which corresponds to the true result $Q_{\blacktriangle}$ padded with noise to maintain DO. Our algorithms are obtained by injecting DO into algorithms 1 and 2. The construction of our algorithms directly follows the theoretical insights underlying their insecure counterparts. Due to that, our construction is analogous to the work in [25], but our algorithms avoid padding output to the worst-case upper bound and thus achieve instance-specific performance. Both of our algorithms achieve the runtime $O(N^{3/2})$, up to poly-logarithmic factors. *Nonetheless, this overhead can be reduced to a logarithmic factor by permitting a certain depth of parallelization in the underlying sorting primitive.*

4.2.1 Integrate DO into Algorithm 1

Although traditional pair-wise join is suboptimal for evaluating $Q_{\blacktriangle}$, we will introduce a way to reformulate Algorithm 1 to a binary-join-based strategy, enabling a seamless application of the `DOBinaryJoin` primitive while preserving its worst-case optimal guarantees. It is straightforward to reformulate Algorithm 1 to evaluate $Q_{\blacktriangle}$ using our binary join primitive. Observe that the structure of Algorithm 1 is to iterate over each join key $a \in \pi_A(R) \cap \pi_A(T)$, and classify if it is heavy or light. The algorithm processes join keys in an interleaved, key-by-key manner based on their classification: if a join key a is heavy, the algorithm scans the entire table S; otherwise, it scans $\pi_B(\sigma_{A=a}R) \times \pi_C(\sigma_{A=a}T)$. Instead of doing that, we restructure the computation by grouping all heavy join keys together and all light ones together, which enables batch processing for each group. Specifically, let W_h be a set of all heavy join keys, i.e.,

$$W_h = \{a \in \pi_A(R) \cap \pi_A(T) : |\sigma_{A=a}R| \cdot |\sigma_{A=a}T| \geq |S|\}.$$

Observe that the accumulated number of accesses to S is $|W_h| \cdot |S|$, which is equivalent to $|W_h \times S|$. Thus, we can compute the Cartesian product $W_h \times S$ to obtain all candidates (a, b) and (a, c) in a single step. This way, we can check their presence in R and T altogether via semi-joins. Similarly, let W_l be a subset of R consisting of tuples associated with light join keys, i.e.,

$$W_l = \{(a, b) \in R : a \notin W_h\}.$$

Computing Cartesian products $\pi_B(\sigma_{A=a}R) \times \pi_C(\sigma_{A=a}T)$ over all light a's is equivalent to $W_l \bowtie T$. As we obtain all (b, c) candidates, we then subsequently check their existence in S. One can easily verify that the construction directly follows from Algorithm 1, with the primary distinction being

the processing order. We formally present the DO variant of the construction illustrated above in Algorithm 3.

Algorithm 3 Integrate DO into Algorithm 1

Input: $R(A, B)$, $S(B, C)$, $T(A, C)$

1: $R \leftarrow \texttt{Augmemt}(R, T, \langle A \rangle, 1)$
2: $W_h, W_l \leftarrow \emptyset$
3: **for** each $(t, \alpha_t^R, \alpha_t^T) \in R$ **do**
4: **if** $\alpha_t^R \cdot \alpha_t^T \geq |S|$ **then**
5: Add t to W_h, Add $\perp$ to W_l ▷ Heavy key
6: **else if** $\alpha_t^R \cdot \alpha_t^T = 0$ **then**
7: Add $\perp$ to W_h, Add $\perp$ to W_l ▷ Eliminate unmatched tuples
8: **else**
9: Add t to W_l, Add $\perp$ to W_h ▷ Light key
10: **end if**
11: **end for**
12: $W_h \leftarrow \texttt{Deduplicate}(W_h)$
13: $L'_h \leftarrow \texttt{DOBinaryJoin}(W_h, S)$
14: $L'_h \leftarrow \texttt{SemiJoin}(L'_h, R, \langle A, B \rangle)$, $L_h \leftarrow \texttt{SemiJoin}(L'_h, T, \langle A, C \rangle)$
15: $L'_l \leftarrow \texttt{DOBinaryJoin}(W_l, T)$, $L_l \leftarrow \texttt{SemiJoin}(L'_l, S, \langle B, C \rangle)$
16: $\hat{Q}_\blacktriangle \leftarrow \texttt{Compact}(L_h \cup L_l)$
17: **return** $\hat{Q}_\blacktriangle$

To prevent leakages through control flow and memory accesses to W_h and W_l, whenever we add a real tuple to one of the two tables, we symmetrically add a dummy tuple $\perp$ to the other one (lines 4 to 9). Note that we add complete tuples to W_h for obliviousness, but we only care about A values. As we perform a linear scan over the augmented R, we remove tuples with unmatched join keys in T on-the-fly (line 7), effectively completing the intersection step in Algorithm 1. Since Algorithm 1 linearly scans S for each distinct heavy join key, we need to de-duplicate W_h (line 12) to remove redundant entries. This ensures that the binary join in line 13 produces the Cartesian product $W_h \times S$. In line 14, we employ the semi-join primitive to verify the presence of (a, b) and

(a, c) pairs in R and S, respectively, which corresponds to line 5 in Algorithm 1. At this point, we have obtained all join results for heavy join keys, and turn to deal with light join keys. However, W_l contains all tuples in R associated with light join keys. In line 15, we compute $W_l \bowtie T$, which retains only outputs associated with light join keys. Next, we subsequently check the existence of (b, c) pairs in S via semi-join, which matches the logic in lines 10-11 of Algorithm 1. Finally, $L_h \cup L_l$ contains all the join results, padded with noise for differential obliviousness. For clarity, a running example demonstrating the algorithm can be found in Appendix A.2.

4.2.2 Security and Performance Analysis for Algorithm 3

Let $\texttt{RealOUT}(\cdot)$ be a function that removes all noise, or dummy elements, from its input relation. The bounds $|\texttt{RealOUT}(L'_h)| \leq N^{3/2}$ and $|\texttt{RealOUT}(L'_l)| \leq N^{3/2}$ follow directly from the inference that comes before the algorithm. Since semi-join operations only eliminate (i.e., dummy out) real elements from L'_h and L'_l, the upper bounds still hold afterwards. For $\texttt{RealOUT}(\hat{Q}_{\blacktriangle}) = Q_{\blacktriangle}$, observe that the algorithm still leverages the power of two choices and takes different actions accordingly, iterating over all a in de-duplicated W_h followed by $a \in W_l$, we have

$$\sum_{a \in W_h \cup W_l} \min\{|\sigma_{A=a} R| \cdot |\sigma_{A=a} T|, |S|\},$$

which mirrors the access sequence of Algorithm 1 and thus implies the same upper bound on $|Q_{\blacktriangle}|$.

Theorem 4.2.1 (Main result for Algorithm 3)**.** *Let* $M \coloneqq \max\{|\texttt{RealOUT}(L'_h)|, |\texttt{RealOUT}(L'_l)|\}$; *then, for any* $\epsilon \in (0, 1)$ *and* $\delta \in (0, 1)$*, there exists an* (ϵ, δ)*-differentially oblivious algorithm that runs in time* $O(M \log^2 M + N(\log^2 N + \frac{1}{\epsilon} \log \frac{1}{\delta}))$ *with a output length of at most* $|Q_{\blacktriangle}| + O(\frac{1}{\epsilon}(\mu + \frac{1}{\epsilon} \log \frac{1}{\delta}) \cdot \log \frac{1}{\delta})$.

Proof. Algorithm 3 satisfying (ϵ, δ)-differentially obliviousness follows from the fact that `DOBinaryJoin`

is (ϵ, δ)-differentially oblivious and all other primitives are fully oblivious (no other leakages except for input sizes).

For $M = \max\{|\texttt{RealOUT}(L'_h)|, |\texttt{RealOUT}(L'_l)|\}$, `DOBinaryJoin` has runtime $O(M + N(\log^2 N + \frac{1}{\epsilon}\log\frac{1}{\delta}))$, and other primitives' runtime is dominated by semi-joins, which incur cost of $O(M \log^2 M)$. The output length follows the description of the `Compact` primitive. □

4.2.3 Integrate DO into Algorithm 2

Recall that the fundamental assumption that makes Algorithm 2 worst-case optimal is that time spent on intersections is equivalent to the minimum size of two input arrays, i.e., Equation (3.3). In what follows, we demonstrate that Algorithm 2 can also be equivalently reformulated as a binary-join-based algorithm like Algorithm 3. Suppose, for example, $(\{a_0\} \times \{b_0, \ldots, b_p\}) \subseteq R$ for some $p \in \{0, \ldots, N\}$. Assume that for all $b_i \in \{b_0, \ldots, b_p\}$, we have $|\pi_C(\sigma_{A=a_0}T)| \geq |\pi_C(\sigma_{B=b_i}S)|$. Then, the time spent in lines 3 and 4 of Algorithm 2 is equal to

$$\sum_{i=0}^{p} |\pi_C(\sigma_{B=b_i}S)|, \tag{4.1}$$

which is equivalent to

$$|(\sigma_{A=a_0}R) \bowtie_B S|.$$

Then, let K_1 be a set of such (a, b) pairs in R, i.e.,

$$K_1 = \{(a, b) \in R : |\pi_C(\sigma_{A=a}T)| \geq |\pi_C(\sigma_{B=b}S)|\}.$$

If we sum Equation (4.1) over all distinct $a \in \texttt{Dist}(\pi_A(K_1))$, following the proof led by Equation (3.3), we have

$$\begin{aligned}
&\sum_{a \in \texttt{Dist}(\pi_A(K_1))} \sum_{b \in \pi_B(\sigma_{A=a}R)} |\pi_C(\sigma_{B=b}S)| \\
=&\sum_{(a,b) \in K_1} |\pi_C(\sigma_{B=b}S)| \\
=&|K_1 \bowtie_B S| \\
\leq& N^{3/2}. \qquad (4.2)
\end{aligned}$$

Note that Equation (4.2) follows from the fact that for all $(a, b) \in K_1$, we have $|\pi_C(\sigma_{A=a}T)| \geq |\pi_C(\sigma_{B=b}S)|$, thereby satisfying Equation (3.3). Consequently, summing over all a and b is followed by the same logic.

Similarly, for $K_2 = R \setminus K_1$, by switching the order of a and b (lines 2 and 3 in Algorithm 2), one can obtain $|K_2 \bowtie_A T| \leq N^{3/2}$. Building on these observations, we proceed to present the differentially oblivious version of Algorithm 4.

Note that, analogous to Algorithm 3, after we obtain the binary join results L_1' and L_2', we have not "closed the loop" of triangles. Specifically, for tuples in L_1', we must verify whether each candidate (a, b, c) is connected by edge $(a, c) \in T$. Similarly, we need to check L_2' for $(b, c) \in S$. By performing semi-join operations, it is ensured that all real tuples (a, b, c) appearing in L_1 and L_2 are valid triangles. Readers may refer to Appendix A.3 for a step-by-step example.

Algorithm 4 Integrate DO into Algorithm 2

Input: $R(A, B)$, $S(B, C)$, $T(A, C)$

$K_1, K_2 \leftarrow \emptyset$

$R \leftarrow \texttt{Augment}(R, T, \langle A \rangle, 0)$, $R \leftarrow \texttt{Augment}(R, S, \langle B \rangle, 0)$

for each $(t, \alpha_t^T, \alpha_t^S) \in R$ **do**
    if $\alpha_t^T \geq \alpha_t^S$ **then**
        Add t to K_1, Add $\perp$ to K_2
    else
        Add t to K_2, Add $\perp$ to K_1
    end if
end for

$L_1' \leftarrow \texttt{DOBinaryJoin}(K_1, S)$, $L_2' \leftarrow \texttt{DOBinaryJoin}(K_2, T)$ $\quad \triangleright\ K_1 \bowtie_B S$, $K_2 \bowtie_A T$

$L_1 \leftarrow \texttt{SemiJoin}(L_1', T, \langle A, C \rangle)$, $L_2 \leftarrow \texttt{SemiJoin}(L_2', S, \langle B, C \rangle)$

$\hat{Q}_{\blacktriangle} \leftarrow \texttt{Compact}(L_1 \cup L_2)$

return $\hat{Q}_{\blacktriangle}$

4.2.4 Security and Performance analysis for Algorithm 4

Similar to our analysis for Algorithm 3, $\texttt{RealOUT}(L_1')$ and $\texttt{RealOUT}(L_2')$ are bounded by $N^{3/2}$, which follows our prior arguments. By Lemma 2.5.1,

$$\begin{aligned}
&\sum_{(a,b)\in R} \min\{|\pi_C(\sigma_{A=a}T)|, |\pi_C(\sigma_{B=b})S|\} \\
\leq &\sum_{(a,b)\in R} |T \ltimes (a,b)|^{1/2} \cdot |S \ltimes (a,b)|^{1/2} \\
\leq &N^{3/2}
\end{aligned}$$

Theorem 4.2.2 (Main result for Algorithm 4)**.** *Let* $M := \max\{|\texttt{RealOUT}(L_1')|, |\texttt{RealOUT}(L_2')|\}$; *then, for any* $\epsilon \in (0, 1)$ *and* $\delta \in (0, 1)$, *there exists an* (ϵ, δ)*-differential oblivious algorithm*

that runs in time $O(M \log^2 M + N(\log^2 N + \frac{1}{\epsilon} \log \frac{1}{\delta}))$ *and produces output of length at most* $|Q_{\blacktriangle}| + O(\frac{1}{\epsilon}(\mu + \frac{1}{\epsilon} \log \frac{1}{\delta}) \cdot \log \frac{1}{\delta})$.

Given that `DOBinaryJoin` is (ϵ, δ)-differentially oblivious and all other primitives satisfy full obliviousness, it suffices to say Algorithm 4 is (ϵ, δ)-differentially oblivious. We omit the detailed proof since it closely follows the structure and reasoning presented in the proof of Theorem 4.2.1.

CHAPTER 5
CONCLUSION AND FUTURE WORK

This thesis investigates the intersection of WCOJ algorithms and oblivious join processing, culminating in constructing differentially oblivious WCOJ algorithms for the triangle join query. Our work demonstrates the possibilities of achieving instance-specific performance while preserving meaningful access pattern privacy guarantees. Although traditional pair-wise join plans are known to be suboptimal for WCOJ problems, this does not preclude using binary join algorithms for such problems. By following existing theories, we show that WCOJ algorithms for the triangle join query can be reformulated to equivalent binary-join-based constructions that preserve the worst-case optimality. By leveraging a recently proposed DO binary join primitive, the resulting algorithms achieve meaningful privacy guarantees without incurring the heavy overhead typically associated with full obliviousness or generic oblivious primitives such as ORAM. Benefiting from decades of research on binary join algorithms, our constructions can accommodate different choices of efficient binary join algorithms, allowing convenient deployment. Our analysis shows that both algorithms match the optimal output size bound of $O(N^{3/2})$ up to a poly-logarithmic factor under non-parallel assumptions and incur only mild overhead due to noise injection.

More importantly, besides presenting algorithms that advance the state of the art, we aim to provide meaningful insights for designing oblivious WCOJ algorithms. Several potential future directions are implied by our work's limitations that await being addressed. Firstly, although our algorithms provide instance-specific performance in theory, they remain to be integrated into full-fledged engines and benchmarked against secure and insecure baselines. We hope that in the future, we can verify the advantageous practical performance of our algorithms. Moreover, the presented

algorithms are specially constructed for triangle joins. Thus, the next step closely followed is the extension to (cyclic) multi-way WCOJ scenarios. It has been demonstrated by [25] that integrating similar approaches into `Generic Join` is possible. With our construction, such integration also provides instance-specific performance. However, existing approaches, including ours, rely heavily on partitioning join keys and invoking multiple binary joins, which leads to intermediate results larger than $|Q_{\blacktriangle}|$. It would not be fully optimal until we eliminate this marginal blowup. Thus, there is a pressing demand for specialized, efficient primitives tailored to multi-way joins, such as efficient tuple retrieval or semi-join reduction. In addition, although our algorithms achieve instance-specific performance from the perspective of obliviousness, their insecure counterparts are still weak in demonstrating such a characteristic. Therefore, new theoretical frameworks towards WCOJ algorithms are in demand that better align with the constraints of oblivious algorithm designs.

Finally, differential obliviousness, as a relaxed yet rigorous privacy notion, shows great potential and a promising avenue for oblivious algorithmic designs. The DO binary join primitive we adopt also highlights the power and flexibility under this notion. Thus, a compelling future direction is designing DO algorithms specialized for multi-way joins, especially WCOJ problems, to fully realize the performance and privacy trade-offs enabled by differential obliviousness.

REFERENCES

[1] A. Arasu, S. Blanas, K. Eguro, *et al.*, "Secure database-as-a-service with cipherbase," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13, New York, New York, USA: Association for Computing Machinery, 2013, pp. 1033–1036, ISBN: 9781450320375.

[2] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan, "Cryptdb: Protecting confidentiality with encrypted query processing," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11, Cascais, Portugal: Association for Computing Machinery, 2011, pp. 85–100, ISBN: 9781450309776.

[3] H. Hacigümüş, B. Iyer, C. Li, and S. Mehrotra, "Executing sql over encrypted data in the database-service-provider model," in *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '02, Madison, Wisconsin: Association for Computing Machinery, 2002, pp. 216–227, ISBN: 1581134975.

[4] V. Costan and S. Devadas, *Intel SGX explained*, Cryptology ePrint Archive, Paper 2016/086, 2016.

[5] G. Kellaris, G. Kollios, K. Nissim, and A. O'Neill, "Generic attacks on secure outsourced databases," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, Vienna, Austria: Association for Computing Machinery, 2016, pp. 1329–1340, ISBN: 9781450341394.

[6] P. Grubbs, K. Sekniqi, V. Bindschaedler, M. Naveed, and T. Ristenpart, "Leakage-abuse attacks against order-revealing encryption," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 655–672.

[7] M. Giraud, A. Anzala-Yamajako, O. Bernard, and P. Lafourcade, *Practical passive leakage-abuse attacks against symmetric searchable encryption*, Cryptology ePrint Archive, Paper 2017/046, 2017.

[8] P. Grubbs, R. McPherson, M. Naveed, T. Ristenpart, and V. Shmatikov, *Breaking web applications built on top of encrypted data*, Cryptology ePrint Archive, Paper 2016/920, 2016.

[9] M. S. Islam, M. Kuzu, and M. Kantarcioglu, "Access pattern disclosure on searchable encryption: Ramification, attack and mitigation.," in *Ndss*, Citeseer, vol. 20, 2012, p. 12.

[10] E. Stefanov, M. V. Dijk, E. Shi, *et al.*, "Path oram: An extremely simple oblivious ram protocol," *J. ACM*, vol. 65, no. 4, Apr. 2018.

[11] S. Krastnikov, F. Kerschbaum, and D. Stebila, "Efficient oblivious database joins," *Proc. VLDB Endow.*, vol. 13, no. 12, pp. 2132–2145, Jul. 2020.

[12] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *J. ACM*, vol. 43, no. 3, pp. 431–473, May 1996.

[13] Y. Wang and K. Yi, "Query evaluation by circuits," in *Proceedings of the 41st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, ser. PODS '22, Philadelphia, PA, USA: Association for Computing Machinery, 2022, pp. 67–78, ISBN: 9781450392600.

[14] A. Arasu and R. Kaushik, *Oblivious query processing*, 2013. arXiv: 1312.4012 [cs.DB].

[15] 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 *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, Boston, MA: USENIX Association, Mar. 2017, pp. 283–298, ISBN: 978-1-931971-37-9.

[16] S. Eskandarian and M. Zaharia, "Oblidb: Oblivious query processing for secure databases," *Proc. VLDB Endow.*, vol. 13, no. 2, pp. 169–183, Oct. 2019.

[17] Z. Chang, D. Xie, S. Wang, and F. Li, "Towards practical oblivious join," in *Proceedings of the 2022 International Conference on Management of Data*, ser. SIGMOD '22, Philadelphia, PA, USA: Association for Computing Machinery, 2022, pp. 803–817, ISBN: 9781450392495.

[18] C. E. Tsourakakis, "Fast counting of triangles in large real networks without counting: Algorithms and laws," in *2008 Eighth IEEE International Conference on Data Mining*, 2008, pp. 608–617.

[19] S. K. Bera and C. Seshadhri, "How to count triangles, without seeing the whole graph," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '20, Virtual Event, CA, USA: Association for Computing Machinery, 2020, pp. 306–316, ISBN: 9781450379984.

[20] M. Al Hasan and V. S. Dave, "Triangle counting in large networks: A review," *WIREs Data Mining and Knowledge Discovery*, vol. 8, no. 2, e1226, 2018. eprint: https://wires.onlinelibrary.wiley.com/doi/pdf/10.1002/widm.1226.

[21] A. Atserias, M. Grohe, and D. Marx, *Size bounds and query plans for relational joins*, 2017. arXiv: 1711.03860 [cs.DB].

[22] T. L. Veldhuizen, *Leapfrog triejoin: A worst-case optimal join algorithm*, 2013. arXiv: 1210.0481 [cs.DB].

[23] H. Q. Ngo, C. Ré, and A. Rudra, "Skew strikes back: New developments in the theory of join algorithms," *SIGMOD Rec.*, vol. 42, no. 4, pp. 5–16, Feb. 2014.

[24] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra, "Worst-case optimal join algorithms: [extended abstract]," in *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, ser. PODS '12, Scottsdale, Arizona, USA: Association for Computing Machinery, 2012, pp. 37–48, ISBN: 9781450312486.

[25] X. Hu and Z. Wu, *Optimal oblivious algorithms for multi-way joins*, 2025. arXiv: 2501.04216 [cs.DB].

[26] S. Chu, D. Zhuo, E. Shi, and T.-H. H. Chan, *Differentially oblivious database joins: Overcoming the worst-case curse of fully oblivious algorithms*, Cryptology ePrint Archive, Paper 2021/593, 2021.

[27] T.-H. H. Chan, K.-M. Chung, B. Maggs, and E. Shi, *Foundations of differentially oblivious algorithms*, Cryptology ePrint Archive, Paper 2017/1033, 2017.

[28] O. Goldreich, "Towards a theory of software protection and simulation by oblivious rams," in *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, ser. STOC '87, New York, New York, USA: Association for Computing Machinery, 1987, pp. 182–194, ISBN: 0897912217.

[29] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 640–656.

[30] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx, "Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution," in *Proceed-*

ings of the 26th USENIX Conference on Security Symposium, ser. SEC'17, Vancouver, BC, Canada: USENIX Association, 2017, pp. 1041–1056, ISBN: 9781931971409.

[31] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside sgx enclaves with branch shadowing," in *Proceedings of the 26th USENIX Conference on Security Symposium*, ser. SEC'17, Vancouver, BC, Canada: USENIX Association, 2017, pp. 557–574, ISBN: 9781931971409.

[32] W. Wang, G. Chen, X. Pan, *et al.*, "Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17, Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 2421–2434, ISBN: 9781450349468.

[33] L. Qin, R. Jayaram, E. Shi, Z. Song, D. Zhuo, and S. Chu, "Adore: Differentially oblivious relational database operators," *Proc. VLDB Endow.*, vol. 16, no. 4, pp. 842–855, Dec. 2022.

[34] S. Sasy, A. Johnson, and I. Goldberg, *Fast fully oblivious compaction and shuffling*, Cryptology ePrint Archive, Paper 2022/1333, 2022.

[35] C. Dwork, F. McSherry, K. Nissim, and A. Smith, "Calibrating noise to sensitivity in private data analysis," in *Proceedings of the Third Conference on Theory of Cryptography*, ser. TCC'06, New York, NY: Springer-Verlag, 2006, pp. 265–284, ISBN: 3540327312.

[36] C. Dwork and A. Roth, "The algorithmic foundations of differential privacy," *Found. Trends Theor. Comput. Sci.*, vol. 9, no. 3–4, pp. 211–407, Aug. 2014.

[37] V. Balcer and S. Vadhan, *Differential privacy on finite computers*, 2019. arXiv: 1709.05396 [cs.DS].

[38] L. Ren, C. Fletcher, A. Kwon, *et al.*, "Constants count: Practical improvements to oblivious ram," in *Proceedings of the 24th USENIX Conference on Security Symposium*, ser. SEC'15, Washington, D.C.: USENIX Association, 2015, pp. 415–430, ISBN: 9781931971232.

[39] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa, "Oblix: An efficient oblivious search index," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 279–296.

[40] M. Yannakakis, "Algorithms for acyclic database schemes," in *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*, ser. VLDB '81, Cannes, France: VLDB Endowment, 1981, pp. 82–94.

[41] M. Freitag, M. Bandle, T. Schmidt, A. Kemper, and T. Neumann, "Adopting worst-case optimal joins in relational database systems," *Proc. VLDB Endow.*, vol. 13, no. 12, pp. 1891–1904, Jul. 2020.

[42] C. R. Aberger, A. Lamb, S. Tu, A. Nötzli, K. Olukotun, and C. Ré, "Emptyheaded: A relational engine for graph processing," *ACM Trans. Database Syst.*, vol. 42, no. 4, Oct. 2017.

[43] C. Aberger, A. Lamb, K. Olukotun, and C. Re, "Levelheaded: A unified engine for business intelligence and linear algebra querying," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, 2018, pp. 449–460.

[44] A. Mhedhbi and S. Salihoglu, "Optimizing subgraph queries by combining binary and worst-case optimal joins," *Proc. VLDB Endow.*, vol. 12, no. 11, pp. 1692–1704, Jul. 2019.

[45] Y. R. Wang, M. Willsey, and D. Suciu, "Free join: Unifying worst-case optimal and traditional joins," *Proc. ACM Manag. Data*, vol. 1, no. 2, Jun. 2023.

[46] M. Aref, B. ten Cate, T. J. Green, *et al.*, "Design and implementation of the logicblox system," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15, Melbourne, Victoria, Australia: Association for Computing Machinery, 2015, pp. 1371–1382, ISBN: 9781450327589.

[47] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, ser. AFIPS '68 (Spring), Atlantic City, New Jersey: Association for Computing Machinery, 1968, pp. 307–314, ISBN: 9781450378970.

[48] E. Shi, *Path oblivious heap: Optimal and practical oblivious priority queue*, Cryptology ePrint Archive, Paper 2019/274, 2019.

[49] E. De Cristofaro and G. Tsudik, "Practical private set intersection protocols with linear complexity," in *Financial Cryptography and Data Security*, R. Sion, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 143–159, ISBN: 978-3-642-14577-3.

[50] X. S. Wang, K. Nayak, C. Liu, *et al.*, "Oblivious data structures," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14, Scottsdale, Arizona, USA: Association for Computing Machinery, 2014, pp. 215–226, ISBN: 9781450329576.

[51] M. Joglekar and C. Re, *It's all a matter of degree: Using degree information to optimize multiway joins*, 2015. arXiv: 1508.01239 [cs.DB].

[52] M. N. Kolountzakis, G. L. Miller, R. Peng, and C. E. Tsourakakis, "Efficient triangle counting in large graphs via degree-based vertex partitioning," in *Algorithms and Models for the Web-Graph*. Springer Berlin Heidelberg, 2010, pp. 15–24, ISBN: 9783642180095.

[53] C. E. Tsourakakis, "Fast counting of triangles in large real networks without counting: Algorithms and laws," in *2008 Eighth IEEE International Conference on Data Mining*, 2008, pp. 608–617.

[54] G. Asharov, T.-H. H. Chan, K. Nayak, R. Pass, L. Ren, and E. Shi, *Bucket oblivious sort: An extremely simple oblivious sort*, 2021. arXiv: 2008.01765 [cs.DS].

[55] M. Ajtai, J. Komlós, and E. Szemerédi, "An 0(n log n) sorting network," in *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, ser. STOC '83, New York, NY, USA: Association for Computing Machinery, 1983, pp. 1–9, ISBN: 0897910990.

[56] G. Asharov, I. Komargodski, W.-K. Lin, E. Peserico, and E. Shi, "Oblivious Parallel Tight Compaction," in *1st Conference on Information-Theoretic Cryptography (ITC 2020)*, Y. Tauman Kalai, A. D. Smith, and D. Wichs, Eds., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 163, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020, 11:1–11:23, ISBN: 978-3-95977-151-1.

[57] S. Sasy, A. Johnson, and I. Goldberg, *Fast fully oblivious compaction and shuffling*, Cryptology ePrint Archive, Paper 2022/1333, 2022.

[58] T.-H. H. Chan, K.-M. Chung, and E. Shi, "On the depth of oblivious parallel ram," in *International Conference on the Theory and Application of Cryptology and Information Security*, 2017.

[59] T.-H. H. Chan and E. Shi, *Circuit OPRAM: Unifying statistically and computationally secure ORAMs and OPRAMs*, Cryptology ePrint Archive, Paper 2016/1084, 2016.

TOWARDS OBLIVIOUS WORST-CASE OPTIMAL JOINS

Approved by:

Jennie Rogers
Computer Science
Northwestern University

Xiao Wang
Computer Science
Northwestern University

Date Approved: May 28, 2025

APPENDIX A
RUNNING EXAMPLES

A.1 DO Binary Join Running Example

Figure A.1 presents a running example to illustrate the key ideas of the DO binary join primitive. While the example does not fully demonstrate every detail of the whole algorithm, it is designed to convey the core components and the overall workflow. For example, the binning strategy in the algorithm involves additional subtleties to limit the total number of dummy tuples in the Cartesian product output. Similarly, the noise added to join key multiplicities and the final output is arbitrarily chosen for illustrative purposes rather than drawn from the true distribution used in practice.

A.2 Algorithm 3 Running Example

Figure A.2 includes a running example of Algorithm 3. The purple portion corresponds to the processing of tuples associated with heavy keys, and the green color labels those associated with light keys. All binary joins follow the process shown in Figure A.1.

A.3 Algorithm 4 Running Example

Figure A.3 shows a running example of Algorithm 4. It follows the same convention as previous ones. Red and violet are used to highlight the sides with smaller multiplicities.

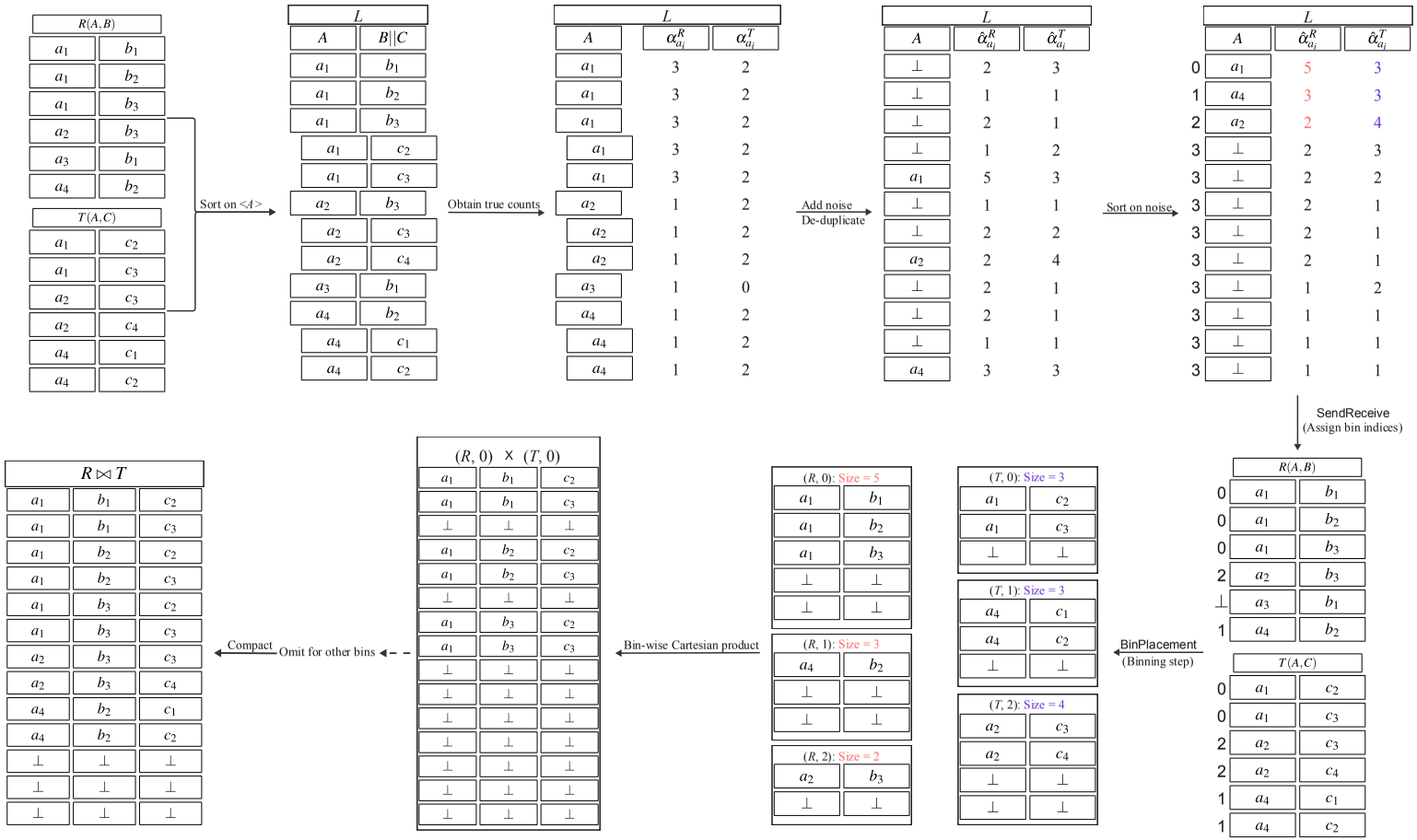

Figure A.1: A running example of the DO binary join primitive

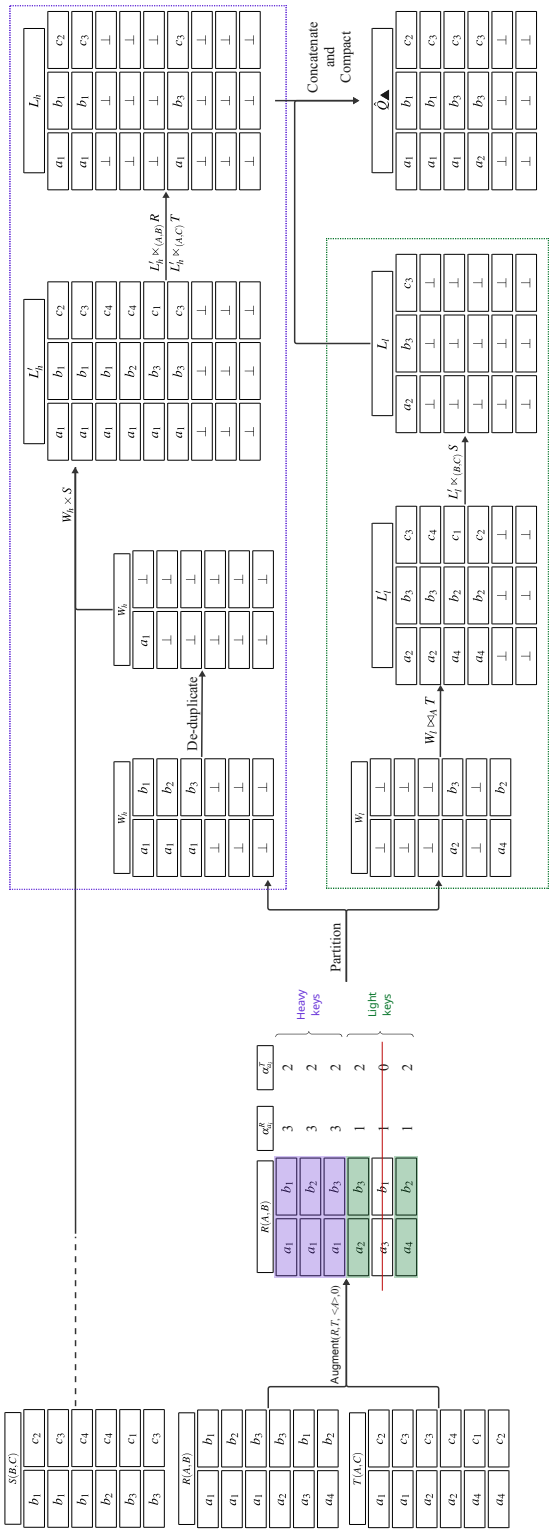

Figure A.2: A running example of Algorithm 3

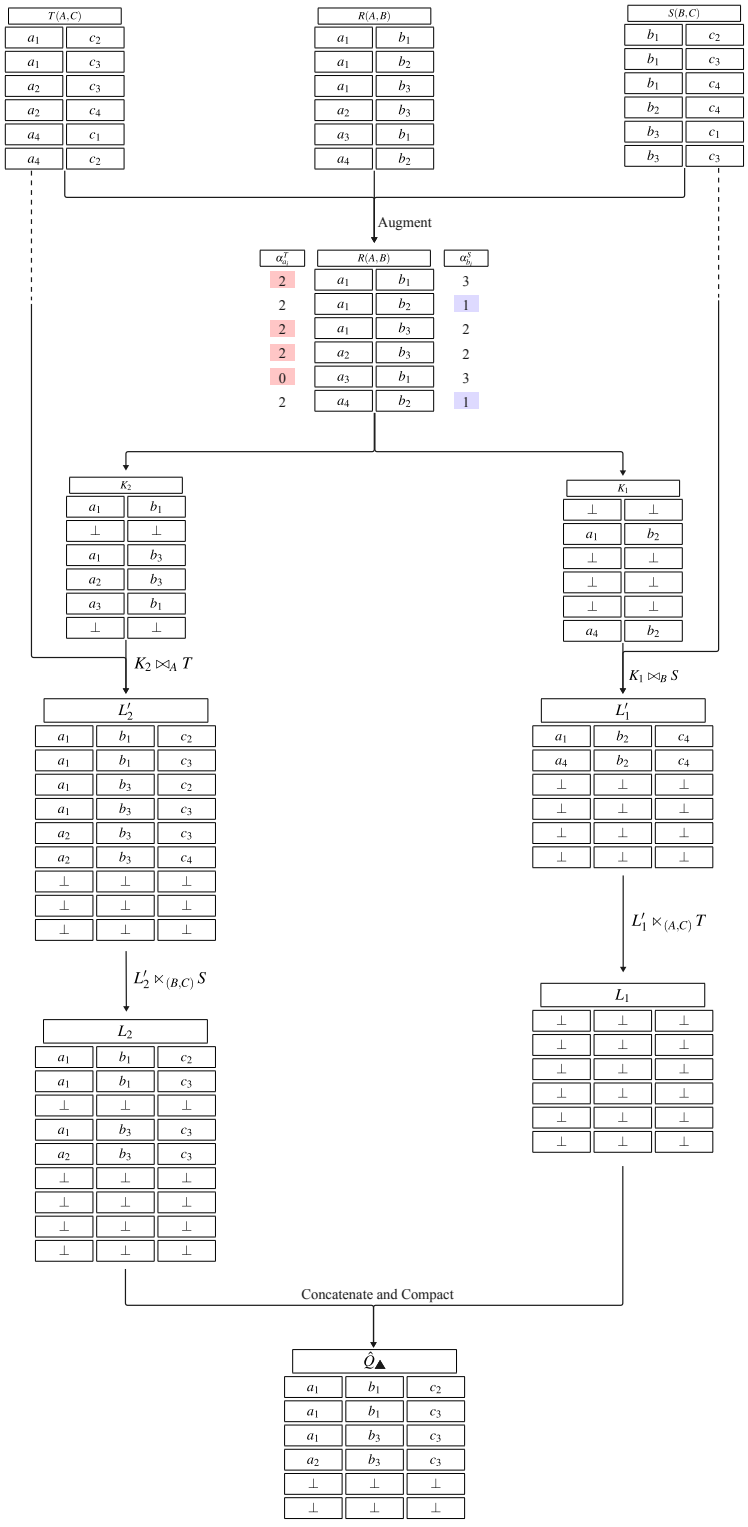

Figure A.3: A running example of Algorithm 4