# Efficient Oblivious Multi-Way Joins with Band Conditions

by

Ruidi Wei

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2025

**Examining Committee Membership**

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

| | |
|---|---|
| External Examiner: | Bruce Bruce<br>Professor, Dept. of Philosophy of Zoology, University of Wallamaloo |
| Supervisor(s): | Ann Elk<br>Professor, Dept. of Zoology, University of Waterloo<br>Andrea Anaconda<br>Professor Emeritus, Dept. of Zoology, University of Waterloo |
| Internal Member: | Pamela Python<br>Professor, Dept. of Zoology, University of Waterloo |
| Internal-External Member: | Meta Meta<br>Professor, Dept. of Philosophy, University of Waterloo |
| Other Member(s): | Leeping Fang<br>Professor, Dept. of Fine Art, University of Waterloo |

## Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

**Abstract**

This thesis introduces the first efficient oblivious algorithm for acyclic multi-way joins with band conditions, extending the classical Yannakakis algorithm to support inequality predicates ($>, <, \geq, \leq$) without leaking sensitive information through memory access patterns. Band joins, which match tuples over value ranges rather than exact keys, are widely used in temporal, spatial, and proximity-based analytics but present unique challenges in oblivious computation. Our approach employs a novel dual-entry technique that transforms range matching into cumulative sum computations, enabling multiplicity computation in an oblivious manner. The algorithm achieves $O(N \log N + OUT \log OUT)$ complexity, matching state-of-the-art oblivious equality joins while supporting full band constraints. We implement the method in Intel SGX and evaluate it on TPC-H and Twitter datasets, demonstrating practical performance and strong obliviousness guarantees under an honest-but-curious adversary model.

**Acknowledgements**

I would like to thank all the little people who made this thesis possible.

## Dedication

This is dedicated to the one I love.

# Table of Contents

# List of Figures

# List of Tables

# List of Abbreviations

This document is incomplete. The external file associated with the glossary 'abbreviations' (which should be called `uw-ethesis.gls-abr`) hasn't been created.

Check the contents of the file `uw-ethesis.glo-abr`. If it's empty, that means you haven't indexed any of your entries in this glossary (using commands like `\gls` or `\glsadd`) so this list can't be generated. If the file isn't empty, the document build process hasn't been completed.

You may need to rerun LaTeX. If you already have, it may be that TeX's shell escape doesn't allow you to run makeindex. Check the transcript file `uw-ethesis.log`. If the shell escape is disabled, try one of the following:

- Run the external (Lua) application:

  ```
  makeglossaries-lite "uw-ethesis"
  ```

- Run the external (Perl) application:

  ```
  makeglossaries "uw-ethesis"
  ```

Then rerun LaTeX on this document.

This message will be removed once the problem has been fixed.

# List of Symbols

This document is incomplete. The external file associated with the glossary 'symbols' (which should be called `uw-ethesis.symbols-gls`) hasn't been created.

Check the contents of the file `uw-ethesis.symbols-glo`. If it's empty, that means you haven't indexed any of your entries in this glossary (using commands like `\gls` or `\glsadd`) so this list can't be generated. If the file isn't empty, the document build process hasn't been completed.

You may need to rerun LaTeX. If you already have, it may be that TeX's shell escape doesn't allow you to run makeindex. Check the transcript file `uw-ethesis.log`. If the shell escape is disabled, try one of the following:

- Run the external (Lua) application:

  ```
  makeglossaries-lite "uw-ethesis"
  ```

- Run the external (Perl) application:

  ```
  makeglossaries "uw-ethesis"
  ```

Then rerun LaTeX on this document.

This message will be removed once the problem has been fixed.

# Chapter 1

# Introduction

Many applications need joins that are not exact matches but based on ranges. For example, a bank may link transfers that happen within ten minutes to detect fraud, or a hospital may connect lab results taken within a week of a diagnosis. These *band joins* are common in finance, healthcare, and time-based analytics. When such queries are done on sensitive data, organizations often encrypt the data before sending it to the cloud. Encryption hides the contents, but not the way the cloud processes the query. In fact, the pattern of memory accesses itself can leak information—for example, which records are considered "close" or how many results are returned. To prevent this leakage, we need algorithms that run *obliviously*, meaning the cloud sees only generic access patterns that reveal nothing about the private data.

For acyclic multi-way joins, the classical Yannakakis algorithm [10] provides optimal complexity—it evaluates queries in time linear in the input size ($N$) and output size (OUT), avoiding the exponential blowup that plagues naive approaches. Recent work has successfully adapted Yannakakis to secure settings, such as the Secure Yannakakis protocol for two-party computation [9]. However, these adaptations handle only equality joins where tuples match on exact values. Band joins present a fundamental new challenge: when matching ranges of values, even the number of matches becomes sensitive information. Consider joining employees with meetings that occurred within their work hours—the access pattern would reveal how many meetings each employee attended, leaking information about their activity level. While generic approaches like Oblivious RAM (ORAM) [5] could hide these patterns, they introduce logarithmic overhead per memory access, with large constant factors that make them impractical for large-scale data processing.

In this thesis, we present the first efficient oblivious algorithm for multi-way band joins.

Our approach extends the oblivious Yannakakis framework to handle inequality predicates through a novel dual-entry technique that transforms range matching into cumulative sum computations. We achieve $O(N \log N + \text{OUT})$ complexity for acyclic queries, where $N$ is the input size and OUT is the actual output size. This matches the complexity of oblivious equality joins while supporting the full generality of band conditions. We implement our algorithm in Intel SGX [3] and demonstrate its practicality on real-world datasets from TPC-H and Twitter.

## 1.1 Problem Statement

Our goal is to design an efficient algorithm for evaluating acyclic multi-way joins with band conditions in the oblivious setting. We focus on acyclic queries, which form a large and practical class of queries that can be represented as join trees. Cyclic queries can be transformed to acyclic ones using Generalized Hypertree Decomposition (GHD) obliviously at a cost that becomes impractical for queries with large GHW. In the equality-condition case, the problem is manageable: tuples can be partitioned into *groups* based on the join key, and each group in one table matches exactly one group in another table. This makes it possible to assign the same multiplicity to all tuples in a group without revealing anything sensitive, and also enables techniques like hash joins and oblivious B-trees. Band joins, however, are fundamentally harder. A single group may match to an *entire range* of groups in the other table, and the number of matching groups itself depends on the data. This number is sensitive, so naively accumulating multiplicities across groups would leak information through the access pattern. The challenge is therefore to extend oblivious multi-way join processing beyond equality to support inequality predicates such as $<, >$ $, \leq, \geq$ without leaking information.

## 1.2 Contributions

This thesis presents the first oblivious algorithm for acyclic multi-way joins that supports band conditions, extending the classical Yannakakis algorithm to handle inequality predicates $(>, <, \geq, \leq)$ while maintaining oblivious access patterns. Our algorithm achieves $O(N \log N + \text{OUT})$ complexity for acyclic queries in the oblivious setting, matching the complexity of existing oblivious equality join algorithms while supporting the full generality of range constraints.

At the core of our approach is a novel dual-entry technique for encoding range constraints obliviously. Unlike equality joins where each tuple matches a single group, band joins require matching against ranges of values. Our dual-entry technique transforms this range matching problem into cumulative sum computations that can be performed with data-independent access patterns. We develop a variant of the Yannakakis algorithm that computes actual tuple multiplicities rather than just existence, enabling precise output size determination without leaking information.

To integrate with existing oblivious join frameworks, we design modified bottom-up and top-down passes that are compatible with the ODBJ framework [8] while extending its multiplicity computation to support band join conditions. This includes new oblivious expansion and alignment algorithms specifically designed for range-based joins, ensuring that variable-sized outputs from range queries do not reveal sensitive information through access patterns.

We implement our algorithm in Intel SGX and provide a comprehensive experimental evaluation on real-world datasets from TPC-H and Twitter. Our security analysis formally proves that all access patterns remain oblivious throughout the band join processing, ensuring that an adversary observing memory accesses learns nothing about the actual data values or result sizes beyond what is revealed by the public parameters.

## 1.3 Thesis Organization

The remainder of this thesis is organized as follows:

- **Chapter 2** reviews related work on oblivious joins and identifies the gap our work addresses.
- **Chapter 3** provides background on database joins, Yannakakis algorithm, oblivious computation, and secure hardware.
- **Chapter 4** presents an overview of our algorithm, developing the approach from binary to multi-way joins.
- **Chapter 5** provides the formal algorithm specification with detailed pseudocode and proofs.
- **Chapter 6** analyzes the security properties and proves obliviousness.

- **Chapter 7** describes our implementation in Intel SGX.
- **Chapter 8** evaluates performance on TPC-H and Twitter datasets.
- **Chapter 9** concludes and discusses future work.

# Chapter 2

# Related Work

This chapter reviews the existing literature on oblivious database operations, focusing on join algorithms. We trace the development from binary equi-joins to our target problem of multi-way band joins, identifying the critical gap that our work addresses.

## 2.1 Efficient Oblivious Database Join

Krastnikov et al. proposed the first efficient oblivious algorithm for binary database equi-joins. Their algorithm achieves $O(n \log^2 n + \mathrm{OUT} \log \mathrm{OUT})$ complexity where $n$ is input size and OUT is output size, matching the standard non-oblivious sort-merge join up to a logarithmic factor.

The key innovation of ODBJ is using sorting networks and novel provably-oblivious constructions without relying on ORAM. The algorithm operates in two main phases: multiplicity computation and result construction. During multiplicity computation, tables are combined and sorted by join attribute, with linear passes counting occurrences. The result construction phase uses oblivious distribute and expand operations to create the appropriate number of copies of each tuple.

However, ODBJ is limited to **equality predicates only** and **binary joins** (two tables). It serves as the foundational algorithm for oblivious join processing that we extend in this work.

## 2.2 Extension to Band Joins (Inequality Constraints)

Chang et al. made two important extensions to oblivious joins:

1. **Binary band joins**: They extended Krastnikov's algorithm to support inequality predicates like $T_1.A \geq T_2.B - c_1$ and $T_1.A \leq T_2.B + c_2$. This maintains oblivious access patterns while handling $>, <, \geq, \leq$ predicates between attributes, but is limited to **binary joins only**.
2. **Multiway equi-joins**: They use ORAM-based index nested-loop join with B-tree indices to support joins over multiple tables, but only for **equality predicates**.

The B-tree approach used for multiway equi-joins cannot be extended to support band conditions. While B-trees are efficient for exact key lookups, range queries in the oblivious setting become problematic—accessing a variable number of nodes for range queries would leak information about the data distribution and result size. To maintain obliviousness, one would need to pad accesses to the worst case, essentially scanning entire tables and negating the benefits of using an index. Therefore, no existing algorithm combines multiple tables with inequality predicates obliviously.

## 2.3 Multi-Way Joins (Classical Non-Oblivious)

The classical Yannakakis algorithm achieves optimal $O(N + \text{OUT})$ complexity for acyclic multi-way joins in the non-oblivious setting. It uses a two-phase approach:

1. **Bottom-up phase**: Semi-join reductions to eliminate tuples that don't contribute to the final result
2. **Top-down phase**: Result reconstruction by propagating constraints down the tree

This approach eliminates tuples that don't contribute to the final result, bounding runtime by output size. While Yannakakis achieves optimal complexity for acyclic queries, it is **not oblivious**—the access patterns reveal information about data distribution and intermediate result sizes. Yannakakis serves as the theoretical foundation for optimal multi-way join processing that we aim to make oblivious.

## 2.4 Worst-Case Optimal Join Algorithms

Recent work by Hu and Wu [7] has made significant progress in developing oblivious algorithms for worst-case optimal multi-way joins, representing an important achievement in oblivious multi-way query processing.

Worst-case optimal algorithms optimize for the theoretical upper bound on output size for a given query structure, assuming maximal matches between tuples regardless of actual data content. This "worst-case" bound represents the maximum possible output size that could occur for any instance with the given query and input sizes. In contrast, Yannakakis' algorithm—and our approach building upon it—optimizes for the actual output size of the specific data instance. This output-sensitive approach is particularly beneficial when tuples do not exhibit maximal matching patterns.

Our work thus follows a complementary direction to Hu and Wu's approach. Their worst-case optimal algorithm is particularly valuable for cyclic queries where there is no known efficient method to compute the exact output size. In contrast, for acyclic queries, the exact output size can be efficiently computed, allowing our oblivious Yannakakis-based approach to achieve $O(N \log N + \mathrm{OUT})$ complexity on acyclic queries, where OUT is the actual output size.

## 2.5 Critical Gap in the Literature

The existing literature reveals a critical gap: **No existing solution combines multi-way joins with band conditions obliviously**.

Table 2.1 summarizes the capabilities of existing approaches:

Opaque uses oblivious sort-merge join but is limited to primary-foreign key joins [11]. ObliDB supports general multi-way joins using hash join, but this approach essentially computes the Cartesian product, leading to poor performance [4, 2]. Hash-based join methods are particularly unsuitable for extension to range queries, as they rely on exact key matching rather than ordering.

A critical limitation of performing multi-way joins as a series of oblivious binary joins is that it discloses intermediate table sizes, leaking sensitive information about the data distribution and selectivity.

Table 2.1: Comparison of Existing Oblivious Join Approaches

| **Approach** | **Binary** | **Multi-way** | **Equality** | **Band** |
|---|---|---|---|---|
| ODBJ (Krastnikov et al.) | ✓ | | ✓ | |
| Chang et al. (binary) | ✓ | | ✓ | ✓ |
| Opaque/ObliDB | | ✓ | ✓ | |
| Chang et al. (multi-way) | | ✓ | ✓ | |
| Hu and Wu (WCO) | | ✓ | ✓ | |
| **Our Work** | ✓ | ✓ | ✓ | ✓ |

## 2.6 Our Approach: Bridging the Gap

Our work bridges this gap by implementing an **oblivious Yannakakis algorithm** that supports both **multi-way joins** and **band conditions**. We achieve this by:

- Using **odbj as the base algorithm** for processing neighboring table pairs in the join tree
- Extending oblivious Yannakakis to support **inequality predicates** through a novel dual-entry technique
- Achieving $O(N \log N + \text{OUT})$ complexity for acyclic queries with full band join support
- Being the **first algorithm** to combine efficient oblivious multi-way processing with general range constraints

This approach maintains the optimal complexity of Yannakakis (up to logarithmic factors) while supporting the full generality of band conditions, all within the oblivious computation model.

# Chapter 3

# Background

This chapter provides the necessary background for understanding our oblivious multi-way join algorithm with band conditions. We cover fundamental database concepts, classical join algorithms including Yannakakis' algorithm, and the principles of oblivious computation and secure hardware.

## 3.1 Database Joins and Query Processing

### 3.1.1 Database Join Operations

A database join is a fundamental operation that combines rows from two or more tables based on a related column between them. The most common type is the equi-join, where rows are matched when they have equal values in specified columns. For example, joining an `Orders` table with a `Customers` table on the customer ID creates a result containing order information enriched with customer details.

Join operations form the backbone of relational database queries. In practice, queries often involve multiple tables that need to be joined together—these are called multi-way joins. The order and method of executing these joins significantly impacts query performance, especially as data sizes grow.

### 3.1.2 Join Trees and Query Structure

Multi-way join queries can be represented as join graphs, where each node represents a table and edges represent join conditions between tables. This tree structure captures the relationships between tables in the query. For instance, in a supply chain query joining `Suppliers`, `Parts`, and `Orders`, the join tree might have `Parts` at the center, connected to both `Suppliers` and `Orders`.

The structure of the join graph determines many properties of the query. When the join graph forms a tree (no cycles), the query is called acyclic. Acyclic queries have special properties that enable more efficient processing algorithms.

### 3.1.3 Acyclic vs Cyclic Queries

Queries are classified as either acyclic or cyclic based on their join graph structure. Acyclic queries form a tree structure where there is exactly one path between any two tables. This property allows them to be decomposed hierarchically and processed efficiently. For example, a typical business query joining `Customer` $\rightarrow$ `Order` $\rightarrow$ `LineItem` $\rightarrow$ `Product` forms an acyclic chain.

Cyclic queries contain cycles in their join graph. The classic example is the triangle query where three tables each join with the other two, forming a cycle. For instance, in a social network, finding groups of three people who all know each other requires joining `Person` with itself three times in a triangular pattern. These cyclic structures prevent direct application of tree-based algorithms like Yannakakis and generally require more complex processing strategies.

### 3.1.4 Handling Cyclic Queries with GHD

Generalized Hypertree Decomposition (GHD) provides a systematic way to transform cyclic queries into acyclic ones. The key idea is to group relations into "bags" arranged in a tree structure, where each bag may contain multiple relations. By pre-computing joins within each bag, we create an acyclic structure that can be processed with tree-based algorithms.

The efficiency of this transformation depends on the Generalized Hypertree Width (GHW) of the query—the minimum number of relations needed in any bag across all possible decompositions. Acyclic queries naturally have GHW = 1 (no grouping needed), while the triangle query has GHW = 2, and a k-cycle has GHW = $\lceil k/2 \rceil$.

The transformation can increase data size exponentially: from N to potentially $N^{\text{GHW}}$, as bags may contain Cartesian products. This exponential blowup makes GHD transformation impractical for queries with large GHW, especially in the oblivious setting where we cannot optimize based on actual data distributions.

## 3.2 Band Joins and Range Queries

### 3.2.1 From Equality to Inequality Joins

While traditional database joins match tuples with exactly equal values, many real-world queries require matching based on ranges or inequalities. These band joins (also called band conditions or range joins) are essential for temporal queries, spatial proximity searches, and interval-based analytics.

Consider a fraud detection query that links credit card transactions occurring within 10 minutes of each other at different locations. This requires joining transactions where the timestamp difference falls within a specified range—a band join rather than an exact match. Similarly, healthcare analytics might join patient visits with lab results taken within a week, or supply chain queries might match orders with shipments arriving within a delivery window.

### 3.2.2 Why Band Joins are Challenging

Band joins are fundamentally harder than equality joins for several reasons. In an equality join, each value in one table matches at most one group of values in another table. This relationship allows efficient processing using techniques like hash joins or B-tree indexed nested-loop joins.

With band joins, a single value can match an entire range of values in the other table. The number of matches depends on the data distribution—some values might match hundreds of tuples while others match none. This variable fan-out makes it difficult to predict resource requirements and optimize query execution. In the oblivious setting, this challenge is amplified because we cannot allow the access pattern to reveal how many matches each tuple has, as this would leak information about the data distribution.

## 3.3 The Yannakakis Algorithm

### 3.3.1 Optimal Processing for Acyclic Queries

The Yannakakis algorithm [10], developed by Mihalis Yannakakis in 1981, provides an elegant solution for evaluating acyclic multi-way joins with optimal complexity. The algorithm achieves $O(N + \text{OUT})$ time, where $N$ is the total input size and OUT is the output size—this is optimal because any algorithm must at least read the input and write the output.

The key insight is to exploit the tree structure of acyclic queries through a two-phase approach. First, a bottom-up pass eliminates tuples that cannot possibly join with tuples in its own subtree. Then, a top-down pass propagates the global constraints to produce the final result. This approach avoids the exponential blowup that can occur with naive join ordering.

### 3.3.2 The Two-Phase Approach

In the bottom-up phase, the algorithm performs semi-join reductions starting from the leaves of the join tree. Each child table sends information to its parent about which values actually exist, allowing the parent to eliminate tuples that have no matching partners. This process continues up to the root, with each table keeping only tuples that can contribute to the final result based on their subtree.

The top-down phase then propagates constraints from the root back to the leaves. Starting from the filtered root table containing only tuples that exist in the final result, each parent informs its children about which values remain valid in the global context. This ensures that every tuple in the final result participates in the complete join across all tables.

While Yannakakis' algorithm is optimal for non-oblivious settings, it reveals information through its access patterns—which tuples are eliminated and when reveals the selectivity of different join conditions. Our work extends this algorithm to maintain its efficiency while hiding these access patterns.

## 3.4 Oblivious Computation

### 3.4.1 The Need for Oblivious Algorithms

When sensitive data is processed in untrusted environments like public clouds, encryption alone is insufficient. Even with encrypted data, the pattern of memory accesses during computation can leak sensitive information. For example, a binary search reveals the approximate location of the target value through its access pattern, even if all data is encrypted.

Oblivious algorithms address this by ensuring that memory access patterns are independent of the input data. The sequence of memory locations accessed depends only on public parameters like data size, query structure, or a random variable, not on the actual values being processed. This prevents an adversary who can observe all memory accesses from learning anything about the private data.

### 3.4.2 The Oblivious Security Model

In our security model, we assume an honest-but-curious adversary who can observe all memory access patterns but cannot tamper with the computation. The adversary knows certain public parameters: the sizes of input and output tables, the structure of the join query, and any constants in the join conditions. However, the actual data values, their distribution, and the selectivity of join conditions remain private.

An algorithm is oblivious if two different datasets with the same public parameters produce identical access patterns. This means an adversary watching the memory accesses cannot distinguish between a dataset where two tables are selective and one where the other two tables are selective, as long as the table sizes are the same.

### 3.4.3 Building Blocks for Oblivious Algorithms

Oblivious algorithms rely on data-independent primitives that operate on tables (arrays of rows) where each row contains values for multiple columns. These primitives ensure access patterns reveal no information about the input data. Table 3.1 summarizes the key primitives used in our algorithm.

**Oblivious Sorting** applies to a table using a sorting order function that takes two rows and returns either $-1$ (first row is "smaller") or 1 (second row is "smaller"). The algorithm

Table 3.1: Oblivious Primitives Used in Our Algorithm

| **Primitive** | **Description** | **Runtime** | **Reference** |
|---|---|---|---|
| Oblivious Sorting | Sorts data using fixed comparison networks independent of input values | $O(n \log^2 n)$ | Batcher [1] |
| Oblivious Distribution | Moves rows to computed target positions without revealing data patterns | $O(n \log n)$ | ODBJ [8] |
| Oblivious Expansion | Creates multiple copies of rows based on precomputed multiplicities | $O(n \log n)$ | ODBJ [8] |
| Map | Applies an oblivious function that reads from and writes to fixed locations in each row | $O(n)$ | Standard technique |
| Linear Scan | Applies an oblivious function to a fixed-size sliding window over a table | $O(n)$ | Standard technique |
| Parallel Scan | Applies an oblivious function that takes two rows and operates on fixed locations | $O(n)$ | Standard technique |

uses fixed comparison networks where the sequence of row comparisons is predetermined based only on the table size, not the actual row values. This ensures that regardless of the data distribution, the same row positions are accessed in the same order, preventing information leakage through access patterns. Our implementation uses Batcher's bitonic sort with $O(n \log^2 n)$ comparisons for its simplicity and deterministic structure. However, this can be replaced with optimal algorithms like Zig-zag sort [6] achieving $O(n \log n)$ complexity to obtain the theoretical guarantee of $O(N \log N + \mathrm{OUT})$ for our overall algorithm, where $N$ is the input size and OUT is the output size.

**Oblivious Distribution** moves rows to computed target positions within a table without revealing information about where rows are being moved or how many rows end up

in each location. This primitive is essential in the ODBJ framework for repositioning rows according to their multiplicities before expansion. The algorithm uses a series of oblivious sorting and permutation operations to achieve the desired redistribution while ensuring that the access pattern depends only on the table size and target position computation, not on the actual row values or movement patterns.

**Oblivious Expansion** creates multiple copies of rows based on precomputed multiplicities, ensuring that the duplication process reveals no information about how many copies each row requires. This primitive works in conjunction with oblivious distribution to construct join result tables where each row appears exactly as many times as required by the join semantics. The challenge lies in handling variable expansion factors obliviously—some rows may need many copies while others need few, but the algorithm must access memory in a pattern that depends only on the maximum possible expansion factor, not the actual requirements.

**Map** applies an oblivious function that reads from and writes to fixed locations within each row of a table. The function operates independently on every row, maintaining data-independent access patterns by accessing predetermined fixed locations within each row. This primitive enables row transformations while preserving obliviousness, such as computing new attributes, applying selection conditions, or reformatting tuple structures. The resulting table may contain modified rows but maintains the same size as the input table.

**Linear Scan** takes a table and an oblivious function that operates on a fixed number of rows (the window). The function reads from and writes to fixed locations within the window. Linear scan places this fixed-size sliding window on the table and applies the function to each window position as it moves through the table. This maintains data-independent access patterns since the window size and movement are predetermined, ensuring that the memory access sequence depends only on the table size and window size, not on the actual data values encountered.

**Parallel Scan** takes a table and an oblivious function that operates on two rows (one from each table). The function reads from and writes to fixed locations within these two rows. Parallel scan processes corresponding row pairs with equal indexes from two tables of the same size, applying the oblivious function to each pair sequentially. This maintains data-independent access patterns since the function operates on predetermined fixed locations, ensuring that the memory access sequence depends only on the table sizes and function structure, not on the actual data values.

## 3.5 Intel SGX and Secure Hardware

### 3.5.1 Trusted Execution Environments

Intel Software Guard Extensions (SGX) [3] provides hardware-based trusted execution environments called enclaves. These enclaves protect code and data from observation or modification by any external software, including the operating system and hypervisor. When combined with oblivious algorithms, SGX provides end-to-end security for sensitive computations in untrusted environments.

SGX encrypts enclave memory in hardware, ensuring that even physical memory dumps reveal only ciphertext. However, SGX does not hide memory access patterns—the sequence of addresses accessed by the enclave is visible to the OS through page faults and cache effects. This is why oblivious algorithms are essential: they ensure these visible access patterns leak no information about the protected data.

### 3.5.2 Implementing Oblivious Joins in SGX

Our implementation runs entirely within an SGX enclave, processing encrypted data obliviously. The enclave receives encrypted tables, decrypts them internally, performs the oblivious join computation, and returns encrypted results. Throughout this process, the memory access patterns visible to the untrusted host reveal nothing about the data. The combination of hardware protection and our algorithmic obliviousness provides strong security guarantees against side-channel attacks.

## 3.6 Summary

This background establishes the basic concepts underlying our work: the structure and challenges of multi-way band joins, the elegance and optimality of the Yannakakis algorithm for acyclic queries, the principles of oblivious computation for protecting sensitive data, and the role of secure hardware in practical deployments. Building on these foundations, we develop the first oblivious algorithm that combines all these elements—supporting multi-way joins with band conditions while maintaining data-independent access patterns throughout the computation.

# Chapter 4

# Algorithm Overview

This chapter provides an intuitive overview of our algorithm before diving into formal specifications. We begin with ODBJ's [8] binary join solution, which separates multiplicity computation from result construction. We then explain how to extend this to multi-way joins by computing multiplicities recursively through tree traversals—a structure that surprisingly mirrors Yannakakis' [10] classical algorithm. Finally, we introduce our dual-entry technique that enables these computations to work with band conditions, transforming range matching into simple cumulative sums through sorted sequences.

## 4.1 From odbj to Oblivious Yannakakis

Our work builds upon recent advances in oblivious database operations, extending binary join techniques to handle multi-way joins with band conditions.

### 4.1.1 Starting with odbj's Architecture

Krastnikov et al.'s ODBJ [8] provides an elegant solution for oblivious binary joins, achieving $O(n \log^2 n + \text{OUT} \log \text{OUT})$ complexity where $n$ is input size and OUT is output size. The ODBJ architecture can be separated into two distinct parts: multiplicity computation and result construction.

#### Multiplicity Computation Phase

The algorithm begins by combining both input tables into a single table sorted by the join attribute, with each tuple tagged by its source table. This combined representation enables counting and recording multiplicities. A forward pass through the sorted table counts occurrences of each unique join key, with two counters tracking tuples from $T_1$ and $T_2$. These counts are then propagated backward to ensure every tuple with the same join key receives the complete count information.

Through this process, each tuple $(j, d)$ is augmented with two metadata values representing the local multiplicities ($\alpha_{\text{local}}$): $\alpha_1(j)$, the occurrence of key $j$ in $T_1$, and $\alpha_2(j)$, the occurrence in $T_2$. The significance of these local multiplicity values becomes clear when we consider the join result—each tuple from $T_1$ must appear $\alpha_2(j)$ times (once for each match in $T_2$), while each tuple from $T_2$ must appear $\alpha_1(j)$ times. Thus each tuple obtains its own multiplicity for result construction.

#### Result Construction Phase

With multiplicities computed, ODBJ constructs the actual join result through three oblivious operations. The **distribute** operation and the **expand** operation work together to duplicate each tuple by its multiplicity. Then, the **align** operation reorders one expanded table to match the other, ensuring that tuples appear at correct locations, ready to be zipped into the binary join result.

This separation means we must obtain the size of the join result, along with multiplicities of all tuples in the join result, before we can duplicate them for the correct number of times or proceed with any further step. For binary joins, ODBJ demonstrates this can be done obliviously using only sorting networks and linear scans, avoiding expensive primitives like ORAM [5].

### 4.1.2 The Multi-Way Multiplicity Challenge

To extend ODBJ [8] to multi-way joins, we must obtain the multiplicity of each tuple in the full join result before constructing it. For binary joins, ODBJ [8] computes this directly. For multi-way joins over a tree structure, the challenge is: how do we compute the final multiplicity of each tuple when it depends on tables across the entire tree?

We start by looking at a smaller picture, joining the subtree for every table, and call the table tuple's multiplicity in this sub-tree join result "local multiplicity ($\alpha_{\text{local}}$)". For

a root tuple, local multiplicity is the same as the final multiplicity ($\alpha_{\text{final}}$). This "local multiplicity of root tuples" can be computed recursively.

We observe that for an arbitrary parent table tuple, its local multiplicity ($\alpha_{\text{local}}$) is a product of contributions from joining with each of the child tables. The contribution from each child table is the sum of local multiplicities of matching child table tuples. With a bottom-up traversal of the join tree, we can compute local multiplicities $\alpha_{\text{local}}$ of all root table tuples.

After obtaining the local multiplicities $\alpha_{\text{local}}$ of the root table tuples, we view them as final multiplicities ($\alpha_{\text{final}}$), and we use a top-down join tree traversal to propagate this final multiplicity $\alpha_{\text{final}}$ information across the join tree.

We then perform the distribute and expand, alignment and concatenation phases using the multiplicities.

### 4.1.3 Connection to Yannakakis

Interestingly, our two-phase structure mirrors Yannakakis's algorithm [10] for acyclic joins. Yannakakis also uses bottom-up semi-join reduction followed by top-down reconstruction. While Yannakakis computes a boolean value for each tuple indicating whether it exists in the join result or not, we count multiplicities for each tuple indicating how many times it exists in the join result.

## 4.2 Band Join Enhancement: Dual Entry Approach

### 4.2.1 The Challenge of Range-Based Multiplicity Computation

The extension from equality joins to band joins introduces a fundamental challenge in multiplicity computation. Consider two tables A and B with band join condition $A.x \geq B.y - c_1$ and $A.x \leq B.y + c_2$. For a tuple from table A with attribute value $A.x = v$, we must sum the local multiplicities of all tuples from table B that satisfy the range constraint. This differs significantly from equality joins where the matching relationship is one-to-one between groups.

In equality joins where $A.x = B.y$, the multiplicity computation is straightforward: we sort the combined tables by the join attribute and perform a linear pass, summing tuples with identical values and resetting the sum when the join attribute value changes. This

direct accumulation works because each tuple matches exactly those tuples with the same join key value.

Band joins complicate this process because each tuple from table A matches all tuples from table B where $v - c_2 \leq B.y \leq v + c_1$. The challenge lies in efficiently computing the sum of multiplicities across this range without revealing information about the data distribution. A naive approach would require examining each possible matching tuple individually, but this would be inefficient and potentially leak information through access patterns.

### 4.2.2 The Dual Entry Solution

Our solution transforms the range matching problem into a cumulative sum computation through a dual entry technique. For each tuple $t$ in table A with join attribute value $v$, we create two boundary markers: a start entry at position $v - c_2$ representing the smallest possible matching B value, and an end entry at position $v + c_1$ representing the largest possible matching B value. These boundary markers, combined with the actual tuples from table B, are then sorted by their join attribute values to create a unified sequence.

During a single linear pass through this sorted sequence, we maintain a cumulative sum counter ($C$) that increments by the local multiplicity ($\alpha_{\text{local}}$) of each tuple from table B. When we reach the start and end boundary markers for a given tuple from table A, we record the current cumulative sum values. The difference between the end counter and start counter gives precisely the sum of local multiplicities $\alpha_{\text{local}}$ for all table B tuples that fall within the required range.

This dual entry approach transforms a complex range matching problem into a simple interval computation. The key insight is that start and end entries define interval boundaries in the sorted combined table, and the cumulative counter tracks all relevant contributions seen so far. The difference between consecutive boundary markers captures exactly the multiplicities needed for the range-based join. Crucially, this process remains oblivious since all operations rely solely on oblivious sorting and fixed linear passes with predetermined access patterns, ensuring that no information about the actual data values or match counts is leaked through memory access patterns.

# Chapter 5

# Detailed Algorithm

## 5.1 Algorithm Overview and Notation

Our oblivious multi-way band join algorithm operates on acyclic join trees in four distinct phases, each maintaining data-independent access patterns while computing the complete join result.

### 5.1.1 Algorithm Input and Output

The algorithm takes as input a join tree $T = (V, E)$ where $V$ are table nodes and $E$ are join edges, along with tables $\{R_1, R_2, \ldots, R_k\}$ where each $R_i$ corresponds to node $v_i \in V$. For each edge $(v_i, v_j) \in E$, band join constraints specify predicates between join attributes. The algorithm produces as output an oblivious join result table $R_{result}$ containing all tuples satisfying the multi-way band join constraints.

### 5.1.2 Table Type Definitions

Following Krastnikov et al.'s terminology [8], we distinguish between different types of tables based on their state in the algorithm:

- **input tables**: Original unmodified tables $\{R_1, R_2, \ldots, R_k\}$ as provided to the algorithm

- **augmented tables**: input tables extended with persistent multiplicity metadata
- **combined tables**: Arrays of entries from multiple augmented tables with temporary metadata, sorted by join attribute for dual-entry processing
- **expanded tables**: augmented tables where each tuple appears exactly $\alpha_{\text{final}}$ times
- **aligned tables**: expanded tables reordered to enable correct concatenation for join result construction

Table 5.1: Table Schema Evolution Throughout Algorithm Phases

| **Type** | **Original Attrs** | **Persistent Meta** | **Temporary Meta** |
|---|---|---|---|
| $R_{input}$ | $\{a_1, a_2, \ldots, a_n\}$ | No | No |
| $R_{aug}$ | $\{a_1, a_2, \ldots, a_n\}$ | Yes | No |
| $R_{\text{comb}}$ | $\{\text{type}, \text{join\_attr}, d\}$ | Yes | Yes |
| $R_{exp}$ | $\{a_1, a_2, \ldots, a_n\}$ | Yes | No |
| $R_{align}$ | $\{a_1, a_2, \ldots, a_n\}$ | Yes | No |

**Metadata presence indicators:**

- **Persistent Meta**: Whether table contains metadata that carries forward through phases (orig_idx, $\alpha_{\text{local}}$, $\alpha_{\text{final}}$, $F_{\text{sum}}$)
- **Temporary Meta**: Whether table contains metadata used only during specific computations. Combined tables use either $C_{\text{local}}$ (bottom-up) or $C_{\text{foreign}}$ (top-down), not both simultaneously
- **Note**: Combined tables have a special dual-entry structure where original attributes are transformed into $\{\text{type}, a, d\}$ format

### 5.1.3 Data Structures and Notation

The following table summarizes the key data structures, variables, and notation used throughout our oblivious multi-way band join algorithm. The notation distinguishes between entry type constants (using $\tau$ symbols), field accessors for tuple metadata, and various counter variables used in multiplicity computation.

Table 5.2: Algorithm Data Structures and Notation

| **Notation** | **Description** |
|---|---|
| $T = (V, E)$ | Join tree with table nodes $V$ and join edges $E$ |
| $R_i$ | Relation/table at node $v_i \in V$ |
| $t$ | Tuple/entry in any table (may include metadata depending on processing phase) |
| $\alpha_{\text{local}}$ | Local multiplicity ($\alpha_{\text{local}}$): number of times a tuple appears in subtree join result |
| $\alpha_{\text{final}}$ | Final multiplicity ($\alpha_{\text{final}}$): number of times a tuple appears in complete join result |
| $F_{\text{sum}}$ | Foreign cumulative sum: accumulated foreign contributions from parent multiplicities |
| SOURCE | SOURCE entry type constant ($\tau_{\text{src}}$) |
| START | TARGET_START entry type constant ($\tau_{\text{start}}$) |
| END | TARGET_END entry type constant ($\tau_{\text{end}}$) |
| $e$ | General entry variable |
| $e_s, e_t$ | Start and end entry variables |
| $C$ | Generic counter variable ($C$) |
| $C_{\text{foreign}}$ | Foreign cumulative sum (temporary): intermediate values during dual-counter computation |
| $w_{\text{local}}$ | Local weight counter ($w_{\text{local}}$) |
| $C_{\text{copy}}$ | Copy counter ($C_{\text{copy}}$) |
| $C_{\text{local}}$ | Local cumulative sum (temporary): intermediate values during bottom-up computation |
| $\pi$ | Entry type precedence mapping ($\pi$) |
| $e$.type | Entry type field (replaces .type) |
| $e.d$ | Entry data/tuple reference (replaces .data) |
| $t$.orig_idx | Original tuple index (replaces .orig_idx) |
| $t$.join_attr | Join attribute ($a$) |
| $R_{\text{comb}}$ | combined table of entries for dual-entry processing, sorted by join attribute |
| $(c_1, c_2)$ | Band join constraint parameters |

### 5.1.4 Formal Definitions of Multiplicities

We define three key multiplicities that track tuple participation throughout the join computation:

**Local Multiplicity ($\alpha_{\text{local}}$):** For a tuple $t$ in table $R_v$ at node $v$ in the join tree, the local multiplicity represents the number of times $t$ participates in the join result when considering only the visited portion of the subtree rooted at $v$. During the bottom-up phase, this is computed incrementally: after processing child $c_i$ of $v$, we have:

$$t.\alpha_{\text{local}} = |\{r \in \bowtie_{T_v^{(i)}} : t \in r\}|$$

where $T_v^{(i)}$ denotes the subtree rooted at $v$ restricted to $v$ itself and its first $i$ processed children (and their subtrees). After all children are processed, $T_v^{(k)} = T_v$ where $k = |children(v)|$. For leaf nodes, $\alpha_{\text{local}} = 1$ for all tuples.

**Final Multiplicity ($\alpha_{\text{final}}$):** For any tuple $t$ in any table, the final multiplicity represents the number of times $t$ appears in the complete join result across all tables. Formally:

$$t.\alpha_{\text{final}} = |\{r \in \bowtie_T : t \in r\}|$$

where $T$ is the entire join tree and $\bowtie_T$ represents the complete join result. For the root node, $\alpha_{\text{final}} = \alpha_{\text{local}}$. For all other nodes, $\alpha_{\text{final}}$ is computed during the top-down phase by propagating information from parent to children.

**Foreign Multiplicity ($\alpha_{\text{foreign}}$):** For a tuple $t$ in table $R_v$ at node $v$ in the join tree, the foreign multiplicity represents the number of times $t$ participates in the join result when considering all tables *outside* the subtree rooted at $v$, plus the node $v$ itself. Formally:

$$t.\alpha_{\text{foreign}} = |\{r \in \bowtie_{T \setminus T_v^-} : t \in r\}|$$

where $T_v^-$ denotes the subtree rooted at $v$ excluding $v$ itself, and $T \setminus T_v^-$ represents all tables in the tree except those in the children's subtrees. This counts how many times $t$ appears when joining with all tables not in its subtree. The key relationship $\alpha_{\text{final}} = \alpha_{\text{local}} \times \alpha_{\text{foreign}}$ holds because we assume an acyclic join tree. Specifically, there are no join conditions connecting any node in $T_v^-$ to any node in $T \setminus T_v$ (all connections must go through $v$). This independence allows the multiplicities to multiply. In practice, we compute $\alpha_{\text{foreign}} = \frac{\alpha_{\text{final}}}{\alpha_{\text{local}}}$ during the top-down phase.

**Foreign Multiplicity Sum ($F_{\text{sum}}$):** For a child tuple $t_c$ in table $R_c$ with parent node $v$, if we were to join all tables in $T \setminus T_c^-$ and sort the result by the join attribute between $v$ and $c$, then $t_c.F_{\text{sum}}$ is the index of the first entry from the parent table that matches $t_c$. This value is computed during the top-down phase and is used in the align-concatenate phase to determine the correct positioning of tuples in the final result.

### 5.1.5 Common Utilities Across Multiple Phases

Our algorithm employs several oblivious operations that serve as common utilities across multiple phases.

**ObliviousSort** utility is the foundation of our approach, utilizing predetermined comparison networks [1] to sort tables with fixed access patterns that remain independent of actual data values. The sorting network's structure is determined solely by the input size, ensuring that the sequence of comparisons and swaps follows the same pattern regardless of the data being sorted, which is essential for maintaining oblivious properties in secure computation environments.

**LinearPass** utility represents our core primitive for processing sorted tables through stateless window operations. This utility applies functions to sliding windows of size 2 over sorted data, where each function operates exclusively on the current window content and position index without any external state dependencies. The function must access (read / write) fixed locations relative to the window, ensuring oblivious access patterns.

**Algorithm 1** LinearPass: Apply window function across table with sliding window size 2

```
1: function LinearPass(R, WindowFunc)
2:     for i = 1 to |R| − 1 do
3:         window ← R[i : i + 1]                ▷ Extract window of size 2
4:         WindowFunc(window, i)                ▷ Apply function to window
5:     end for
6:     return R                                 ▷ Return modified table
7: end function
```

**Map** utility provides element-wise transformations across table entries, applying the same function to each row independently. The function reads the input row, and creates an output row with potentially different schema. This is used to change schema of table, adding or removing columns.

**Algorithm 2** Map: Apply transformation function to each row independently

```
function MAP(R, TransformFunc)
    R_out ← []                                   ▷ Initialize output table
    for i = 1 to |R| do
        R_out[i] ← TRANSFORMFUNC(R[i], i)
    end for
    return R_out
end function
```

**ParallelPass** utility processes two tables of same size in parallel by applying a window function to corresponding pairs of rows. The function modifies the rows in-place, similar to LinearPass but operating on aligned pairs from two tables rather than a sliding window.

**Algorithm 3** ParallelPass: Apply window function to aligned pairs from two tables

```
function PARALLELPASS(R_1, R_2, WindowFunc)
Require: |R_1| = |R_2|                           ▷ Tables must have same size
    for i = 1 to |R_1| do
        window ← [R_1[i], R_2[i]]                ▷ Create window from aligned pair
        WINDOWFUNC(window, i)                    ▷ Apply function to modify in-place
    end for
    return (R_1, R_2)                            ▷ Return modified tables
end function
```

**Join Condition Encoding:** Any join condition between columns can be expressed as an interval constraint. Specifically, a condition between parent column $v$.join_attr and child column $c$.join_attr can be parsed as: $c.\text{join\_attr} \in v.\text{join\_attr} + [x, y]$, where the interval $[x, y]$ may use open or closed boundaries and $x, y \in \mathbb{R} \cup \{\pm\infty\}$.

Sample join predicates map to intervals as follows:

- Equality: $v.\text{join\_attr} = c.\text{join\_attr}$ maps to $c.\text{join\_attr} \in v.\text{join\_attr} + [0, 0]$
- Inequality: $v.\text{join\_attr} > c.\text{join\_attr}$ maps to $c.\text{join\_attr} \in v.\text{join\_attr} + (-\infty, 0)$
- Band constraint: $v.\text{join\_attr} \geq c.\text{join\_attr} - 1$ maps to $c.\text{join\_attr} \in v.\text{join\_attr} + [-1, \infty)$

When multiple conditions constrain the same join, we compute their interval intersection. For instance, combining $v.\text{join\_attr} > c.\text{join\_attr}$ (yielding $(-\infty, 0)$) with $v.\text{join\_attr} \leq c.\text{join\_attr} + 1$ (yielding $[-1, \infty)$) produces the final interval $[-1, 0)$.

The constraint function $\mathcal{C}(v, c)$ operationalizes this interval representation by mapping each parent-child relationship to boundary parameters $\theta = ((d_1, eq_1), (d_2, eq_2))$. Here, $d_1$ and $d_2$ define the interval endpoints, while $eq_1$ and $eq_2$ specify whether boundaries are closed (EQ) or open (NEQ). This encoding is fundamental to the dual-entry technique used throughout the algorithm. For a target tuple with join attribute value $v$, the boundary parameters create: (i) a START entry at $v + d_1$ where if $eq_1 = \text{EQ}$, it includes values $\geq v + d_1$, and if $eq_1 = \text{NEQ}$, it includes values $> v + d_1$; and (ii) an END entry at $v + d_2$ where if $eq_2 = \text{EQ}$, it includes values $\leq v + d_2$, and if $eq_2 = \text{NEQ}$, it includes values $< v + d_2$. This encoding allows the dual-entry technique to handle arbitrary range predicates by converting them into boundary entries that can be processed obliviously.

### 5.1.6 Algorithm Structure

The algorithm begins with initialization to add metadata columns, then operates in four main phases:

1. **Initialization (Section 5.2):** Add metadata columns to create augmented tables
2. **Phase 1 - Bottom-Up (Section 5.3):** Compute local multiplicities ($\alpha_{\text{local}}$) using dual-entry technique for band constraints
3. **Phase 2 - Top-Down (Section 5.4):** Propagate final multiplicities ($\alpha_{\text{final}}$) from root to leaves using foreign multiplicity computation
4. **Phase 3 - Distribution and Expansion (Section 5.5):** Create expanded tables by replicating each tuple according to its $\alpha_{\text{final}}$ using oblivious distribution
5. **Phase 4 - Alignment and Concatenation (Section 5.6):** Reorder expanded tables using $F_{\text{sum}}$ for alignment, then concatenate to form the final join result

Each phase maintains oblivious access patterns by using the primitives described above. The dual-entry technique transforms range-based band constraints into cumulative sum computations, enabling efficient oblivious processing of inequality joins.

**Algorithm 4** Main Algorithm Framework: Oblivious multi-way band join with initialization and four phases

**function** ObliviousMultiWayBandJoin($T = (V, E)$)
    $T_{init} \leftarrow$ InitializeAllTables($T$) ▷ Initialization: Add metadata columns
    $T_{local} \leftarrow$ BottomUpPhase($T_{init}$) ▷ Phase 1: Compute local multiplicities
    $T_{final} \leftarrow$ TopDownPhase($T_{local}$) ▷ Phase 2: Compute final multiplicities
    $T_{expanded} \leftarrow$ DistributeExpand($T_{final}$) ▷ Phase 3: Distribute and expand
    $Result \leftarrow$ AlignConcat($T_{expanded}$, $root$) ▷ Phase 4: Align and concatenate
    **return** $Result$ ▷ Return final join result
**end function**

## 5.2 Initialization

The initialization phase prepares the join tree for multiplicity computation by transforming input tables into augmented tables with empty metadata columns using the Map primitive. All metadata fields are initialized with null placeholders, and actual values are computed in the bottom-up and top-down phases.

**Algorithm 5** Initialize augmented tables: Add metadata columns $\{\text{orig\_idx}, \alpha_{\text{local}}, \alpha_{\text{final}}, F_{\text{sum}}\}$ to input tables using Map primitive with null placeholders. $n_i = |R_i|$, $N = \sum_{i=1}^{k} n_i$.

**function** InitializeAllTables($T$)
    **for all** nodes $v \in V$ **do**
        $R_v \leftarrow$ Map($R_v$, AddMetadataColumns)
        LinearPass($R_v$, WindowSetOriginalIndex)
    **end for**
    **return** $T$
**end function**

**Algorithm 6** Add Metadata Columns: Map function to extend tuples with null metadata

```
function AddMetadataColumns(t, index)
    t.orig_idx ← 0
    t.α_local ← null
    t.α_final ← null
    t.F_sum ← null
    return t
end function
```

**Algorithm 7** Window Set Original Index: Assign sequential indices with sliding window size 2

```
function WindowSetOriginalIndex(window)
    window[1].orig_idx ← window[0].orig_idx + 1
end function
```

The initialization adds metadata columns using Map, then uses LinearPass to assign sequential original indices. This demonstrates the stateless window-based approach where each tuple's index is computed from its predecessor in the sliding window.

## 5.3 Phase 1: Bottom-Up Multiplicity Computation

The bottom-up phase computes local multiplicities ($\alpha_{\text{local}}$) by traversing the join tree $T$ in post-order, as shown in Algorithm 8. For leaf nodes, we initialize each tuple $t \in R_{leaf}$ with $t.\alpha_{\text{local}} = 1$. For non-leaf nodes, the algorithm processes each parent-child pair $(v, c)$ where $v$ is the parent and $c \in children(v)$. The key insight is that at any point during the traversal, for each visited node $v$, each tuple $t \in R_v$ has $t.\alpha_{\text{local}}$ equal to the number of join results it participates in when considering only the portion of the subtree rooted at $v$ that has been visited so far. After all children of $v$ have been processed, $t.\alpha_{\text{local}} = |\{r \in \bowtie_{T_v^{visited}} : t \in r\}|$ where $T_v^{visited}$ represents the subtree rooted at $v$ restricted to nodes that have been visited in the post-order traversal.

For each parent-child pair $(v, c)$, the algorithm invokes ComputeLocalMultiplicities (Algorithm 10) with tables $R_v$ (target) and $R_c$ (source), along with constraint parameters $\theta = \mathcal{C}(v, c)$ that encode the join condition. This updates each tuple $t_v \in R_v$ by computing $t_v.\alpha_{\text{local}}^{\text{new}} = t_v.\alpha_{\text{local}}^{\text{old}} \times \sum_{t_c \in R_c : (t_v, t_c) \text{ satisfy } \mathcal{C}(v,c)} t_c.\alpha_{\text{local}}$, where the second term represents the sum of local multiplicities of all matching tuples from child $c$.

The core innovation lies in the dual-entry technique for handling band join constraints. The CombineTable function (Algorithm 11) creates two boundary markers for each tuple in the target (parent) table—START and END entries—that mark where the matching range begins and ends. For example, if a parent tuple with value 10 matches child tuples between values 8 and 12, CombineTable creates a START entry at 8 and an END entry at 12, then combines these boundary entries with the source (child) tuples into a single table.

We then sort by ComparatorJoinAttr (Algorithm 12), which orders entries primarily by join attribute value and secondarily by a precedence based on entry type and equality type. The precedence ordering (defined by GetPrecedence in Algorithm 18) ensures that (START, EQ) and (END, NEQ) entries come first with precedence 1, SOURCE entries have precedence 2, and (START, NEQ) and (END, EQ) entries come last with precedence 3. This careful ordering guarantees that for any target entry $e_{target}$ that derives boundary entries $e_s$ and $e_t$, the set of source entries $\{e_{source}\}$ appearing between $e_s$ and $e_t$ in the sorted order is exactly the set of source entries that satisfy the join condition with $e_{target}$.

We apply WindowComputeLocalSum (Algorithm 13) via a linear pass to maintain a running sum of local multiplicities: the sum increases by $\alpha_{\text{local}}$ when we encounter SOURCE entries, and the current sum gets recorded when we hit START/END boundaries. We then sort by ComparatorPairwise to place START and END pairs (which originated from the same target tuple) next to each other. Finally, we apply WindowComputeLocalInterval (Algorithm 15) via a linear pass to compute the difference between each pair's cumulative sums, yielding the local interval that represents the local multiplicity contribution from the child's subtree for that target tuple.

After creating and sorting the combined table, we apply UpdateTargetMultiplicity (Algorithm 17) via a parallel pass to propagate the computed intervals back to the parent table, multiplying each target tuple's existing local multiplicity by the contribution from this child (the interval value) to produce the updated local multiplicities.

**Algorithm 8** Bottom-Up Phase: Compute local multiplicities from leaves to root

```
1: function BottomUpPhase(T, root)
2:     order ← PostOrderTraversal(T, root)
3:     for all nodes v in order do
4:         if v is a leaf then
5:             for all tuple t ∈ R_v do
6:                 t.α_local ← 1
7:             end for
8:         else
9:             for all child nodes c of v do
10:                R_v ← ComputeLocalMultiplicities(R_v, R_c, C(v, c))
11:            end for
12:        end if
13:    end for
14:    return T
15: end function
```

**Algorithm 9** Post-Order Traversal: Visit children before parents in tree

```
1: function PostOrderTraversal(T, root)
2:     order ← empty list
3:     for all child nodes c of root do
4:         order ← order+ PostOrderTraversal(T, c)
5:     end for
6:     Append root to order
7:     return order
8: end function
```

**Algorithm 10** Compute Local Multiplicities: Compute new local multiplicities for parent node in bottom-up phase

1: **function** ComputeLocalMultiplicities($R_{\text{target}}$, $R_{\text{source}}$, $\theta$)
2: $R_{combined} \leftarrow$ CombineTable($R_{\text{target}}$, $R_{\text{source}}$, $\theta$)
3: $R_{combined} \leftarrow$ Map($R_{\text{comb}}$, $\lambda e : (e.\text{local\_sum} \leftarrow e.\alpha_{\text{local}}, e.\text{local\_interval} \leftarrow 0, e)$)
4: ObliviousSort($R_{\text{comb}}$, ComparatorJoinAttr)
5: LinearPass($R_{\text{comb}}$, WindowComputeLocalSum)
6: ObliviousSort($R_{\text{comb}}$, ComparatorPairwise)
7: LinearPass($R_{\text{comb}}$, WindowComputeLocalInterval)
8: ObliviousSort($R_{\text{comb}}$, ComparatorEndFirst)
9: $R_{truncated} \leftarrow R_{combined}[1 : |R_{target}|]$
10: ParallelPass($R_{\text{truncated}}$, $R_{\text{target}}$, UpdateTargetMultiplicity)
11: **return** $R_{\text{target}}$
12: **end function**

**Algorithm 11** Combine Table: Create start/end boundary entries for each target tuple and merge with source entries

**function** CombineTable($R_{\text{target}}$, $R_{\text{source}}$, $\theta$)
  $((d_1, eq_1), (d_2, eq_2)) \leftarrow \theta$
  $R'_{source} \leftarrow$ Map($R_{\text{source}}$, **function**($t$):)
    $e$.type $\leftarrow$ SOURCE
    $e$.equality $\leftarrow$ *null*
    $e$.join_attr $\leftarrow$ $t$.join_attr
    $e$.orig_idx $\leftarrow$ $t$.orig_idx
    $e.\alpha_{\text{local}} \leftarrow t.\alpha_{\text{local}}$
    $e.\alpha_{\text{final}} \leftarrow t.\alpha_{\text{final}}$
    $e.F_{\text{sum}} \leftarrow t.F_{\text{sum}}$
    **return** $e$
  $R'_{begin} \leftarrow$ Map($R_{\text{target}}$, **function**($t$):)
    $e$.type $\leftarrow$ START
    $e$.equality $\leftarrow eq_1$
    $e$.join_attr $\leftarrow$ $t$.join_attr $+ d_1$
    $e$.orig_idx $\leftarrow$ $t$.orig_idx
    $e.\alpha_{\text{local}} \leftarrow t.\alpha_{\text{local}}$
    $e.\alpha_{\text{final}} \leftarrow t.\alpha_{\text{final}}$
    $e.F_{\text{sum}} \leftarrow t.F_{\text{sum}}$
    **return** $e$
  $R'_{end} \leftarrow$ Map($R_{\text{target}}$, **function**($t$):)
    $e$.type $\leftarrow$ END
    $e$.equality $\leftarrow eq_2$
    $e$.join_attr $\leftarrow$ $t$.join_attr $+ d_2$
    $e$.orig_idx $\leftarrow$ $t$.orig_idx
    $e.\alpha_{\text{local}} \leftarrow t.\alpha_{\text{local}}$
    $e.\alpha_{\text{final}} \leftarrow t.\alpha_{\text{final}}$
    $e.F_{\text{sum}} \leftarrow t.F_{\text{sum}}$
    **return** $e$
  $R_{\text{comb}} \leftarrow R'_{source} + R'_{begin} + R'_{end}$
  **return** $R_{\text{comb}}$
**end function**

**Algorithm 12** Comparator Join Attribute: Sort entries by join attribute, then by entry type precedence

*Precedence: (START, EQ) → 1, (END, NEQ) → 1, (SOURCE, null) → 2, (START, NEQ) → 3, (END, EQ) → 3*

**function** ComparatorJoinAttr($e_1$, $e_2$)
    **if** $e_1$.join_attr $<$ $e_2$.join_attr **then return** -1
    **else if** $e_1$.join_attr $>$ $e_2$.join_attr **then return** 1
    **else**
        $p_1 \leftarrow$ GetPrecedence(($e_1$.type, $e_1$.equality))
        $p_2 \leftarrow$ GetPrecedence(($e_2$.type, $e_2$.equality))
        **if** $p_1 < p_2$ **then return** -1
        **else if** $p_1 > p_2$ **then return** 1
        **elsereturn** 0
        **end if**
    **end if**
**end function**

**Algorithm 13** Window Compute Local Sum: Compute cumulative sum with sliding window size 2

**function** WindowComputeLocalSum(*window*)
    **if** *window*[1].type = SOURCE **then**
        *window*[1].local_sum $\leftarrow$ *window*[0].local_sum + *window*[1].$\alpha_{\text{local}}$
    **else** ▷ *window*[1].type $\in$ {START, END}
        *window*[1].local_sum $\leftarrow$ *window*[0].local_sum
    **end if**
**end function**

**Algorithm 14** Comparator Pairwise: Organize entries for pairwise START/END processing by grouping targets first, then by index

```
1: function ComparatorPairwise(e1, e2)        ▷ First: Target entries (START/END) before SOURCE entries
2:     if e1.type ∈ {START, END} and e2.type = SOURCE then return -1
3:     else if e1.type = SOURCE and e2.type ∈ {START, END} then return 1
4:     end if                                  ▷ Second: Sort by original index
5:     if e1.orig_idx < e2.orig_idx then return -1
6:     else if e1.orig_idx > e2.orig_idx then return 1
7:     end if                                  ▷ Third: START before END for same index
8:     if e1.type = START and e2.type = END then return -1
9:     else if e1.type = END and e2.type = START then return 1
10:    elsereturn 0
11:    end if
12: end function
```

**Algorithm 15** Window Compute Local Interval: Compute range difference between start/end entries with window size 2

```
1: function WindowComputeLocalInterval(window)
2:     if window[0].type = START and window[1].type = END then
3:         window[1].local_interval ← window[1].local_sum − window[0].local_sum
4:     end if
5: end function
```

**Algorithm 16** Comparator End First: Put END entries first, then sort by original index

```
1: function ComparatorEndFirst(e1, e2)        ▷ First: END entries before all others
2:     if e1.type = END and e2.type ≠ END then return -1
3:     else if e1.type ≠ END and e2.type = END then return 1
4:     end if                                  ▷ Second: Sort by original index
5:     if e1.orig_idx < e2.orig_idx then return -1
6:     else if e1.orig_idx > e2.orig_idx then return 1
7:     elsereturn 0
8:     end if
9: end function
```

**Algorithm 17** Update Target Multiplicity: Multiply target's local multiplicity by computed interval

**function** UPDATETARGETMULTIPLICITY($e_{combined}$, $e_{target}$)
    $e_{target}.\alpha_{\text{local}} \leftarrow e_{target}.\alpha_{\text{local}} \times e_{combined}.\text{local\_interval}$
**end function**

**Algorithm 18** Get Entry Type Precedence: Map (entry_type, equality_type) tuple to precedence value

**function** GETPRECEDENCE($(entry\_type, equality\_type)$)
    **if** $(entry\_type, equality\_type) = (\text{START}, \text{EQ})$ **then return** 1
    **else if** $(entry\_type, equality\_type) = (\text{END}, \text{NEQ})$ **then return** 1
    **else if** $(entry\_type, equality\_type) = (\text{SOURCE}, null)$ **then return** 2
    **else if** $(entry\_type, equality\_type) = (\text{START}, \text{NEQ})$ **then return** 3
    **else if** $(entry\_type, equality\_type) = (\text{END}, \text{EQ})$ **then return** 3
    **end if**
**end function**

## 5.4 Phase 2: Top-Down Final Multiplicity Propagation

The top-down phase propagates final multiplicities ($\alpha_{\text{final}}$) from the root to all nodes in the tree, mirroring the reconstruction phase of Yannakakis [10]. This phase computes how many times each tuple appears in the complete join result by considering contributions from outside its subtree. The traversal proceeds in pre-order, starting from the root where $\alpha_{\text{final}} = \alpha_{\text{local}}$ (since the root has no ancestors), then propagating downward to compute each child's final multiplicity based on its parent's values.

For each parent-child pair $(v, c)$ during the pre-order traversal, the algorithm invokes PROPAGATEFINALMULTIPLICITIES (Algorithm 20) to compute the final multiplicities for child table $R_c$. The key insight is that each child tuple's final multiplicity equals its local multiplicity times its foreign multiplicity, where the foreign multiplicity ($\alpha_{\text{foreign}}$) represents the number of join results from tables outside the child's subtree that connect through the parent. This is computed as: $t_c.\alpha_{\text{final}} = t_c.\alpha_{\text{local}} \times t_c.\alpha_{\text{foreign}}$.

The core question in the top-down phase is: what would be the multiplicity of each parent tuple if we excluded the child table and its entire subtree? That is, what is the

multiplicity of parent (source) table entries in the join result of $\mathcal{T} \setminus \mathcal{T}_c$? Since the final multiplicity is the product of contributions from all neighbors, we can recover this by division. We use a running sum called "local weight" to track the sum of matching child tuples' local multiplicities—this represents the child subtree's contribution. By dividing a parent tuple's final multiplicity by this local weight, we recover its multiplicity in $\mathcal{T} \setminus \mathcal{T}_c$. The sum of these multiplicities for all matching parent tuples gives us the foreign multiplicity ($\alpha_{\text{foreign}}$), which represents the contribution from $\mathcal{T} \setminus \mathcal{T}_c$ and complements the local multiplicity (contribution from $\mathcal{T}_c$).

To compute these values obliviously, we employ a similar structure as the bottom-up phase. We use CombineTable to create START and END boundaries for target table tuples, while SOURCE entries represent source table tuples. The difference from bottom-up is that here the child table is the target (receiving multiplicities) and the parent table is the source (providing multiplicities). After sorting by ComparatorJoinAttr (Algorithm 12), we apply WindowComputeForeignSum (Algorithm 21) via a linear pass that simultaneously tracks two counters. When we encounter START/END boundaries, we update the local weight by adding or subtracting the child tuple's local multiplicity. When we encounter SOURCE entries (parent tuples), we increment the foreign cumulative sum by the parent's final multiplicity divided by the current local weight. This division recovers the parent's multiplicity in $\mathcal{T} \setminus \mathcal{T}_c$, and the accumulation gives each child tuple its foreign multiplicity sum ($F_{\text{sum}}$). This $F_{\text{sum}}$ serves dual purposes: it provides the foreign multiplicity for computing $\alpha_{\text{final}} = \alpha_{\text{local}} \times \alpha_{\text{foreign}}$, and later serves as the alignment key during result construction.

After processing all parent-child pairs in pre-order, every tuple in every table has its final multiplicity computed, representing exactly how many times it will appear in the complete join result. This prepares the tables for the distribution and expansion phase where tuples are replicated according to their final multiplicities.

**Algorithm 19** Top-Down Phase: Propagate final multiplicities from root to leaves

**function** TopDownPhase($T$, $root$)
  **for all** tuple $t \in R_{root}$ **do**
    $t.\alpha_{\text{final}} \leftarrow t.\alpha_{\text{local}}$ ▷ Root final = local
  **end for**
  **for all** nodes $v$ in pre-order traversal of $T$ from $root$ **do**
    **for all** child nodes $c$ of $v$ **do**
      $R_c \leftarrow$ PropagateFinalMultiplicities($R_v$, $R_c$, $\mathcal{C}(v, c)$)
    **end for**
  **end for**
  **return** $T$ ▷ Return tree with tables containing computed final multiplicities
**end function**

**Algorithm 20** Propagate Final Multiplicities: Distribute parent multiplicities to children using dual counters

**function** PropagateFinalMultiplicities($R_{\text{source}}$, $R_{\text{target}}$, $\theta$)
  $R_{\text{comb}} \leftarrow$ CombineTable($R_{\text{target}}$, $R_{\text{source}}$, $\theta$)
  $R_{\text{comb}} \leftarrow$ Map($R_{\text{comb}}$, $\lambda e$ :)
    ($e.w_{\text{local}} \leftarrow e.\alpha_{\text{local}}$,
    $e.C_{\text{foreign}} \leftarrow 0$,
    $e$.foreign_interval $\leftarrow 0, e$)
  ObliviousSort($R_{\text{comb}}$, ComparatorJoinAttr)
  LinearPass($R_{\text{comb}}$, WindowComputeForeignSum)
  ObliviousSort($R_{\text{comb}}$, ComparatorPairwise)
  LinearPass($R_{\text{comb}}$, WindowComputeForeignInterval)
  ObliviousSort($R_{\text{comb}}$, ComparatorEndFirst)
  $R_{\text{truncated}} \leftarrow R_{\text{comb}}[1 : |R_{\text{target}}|]$
  ParallelPass($R_{\text{truncated}}$, $R_{\text{target}}$, UpdateTargetFinalMultiplicity)
  **return** $R_{\text{target}}$
**end function**

**Algorithm 21** Window Compute Foreign Sum: Track foreign and local weight counters simultaneously

**function** WINDOWCOMPUTEFOREIGNSUM($window$, $i$)
  **if** $window[1]$.type = START **then**
    $window[1].w_{\text{local}} \leftarrow window[0].w_{\text{local}} + window[1].\alpha_{\text{local}}$
    $window[1].C_{\text{foreign}} \leftarrow window[0].C_{\text{foreign}}$
  **else if** $window[1]$.type = END **then**
    $window[1].w_{\text{local}} \leftarrow window[0].w_{\text{local}} - window[1].\alpha_{\text{local}}$
    $window[1].C_{\text{foreign}} \leftarrow window[0].C_{\text{foreign}}$
  **else if** $window[1]$.type = SOURCE **then**
    $window[1].w_{\text{local}} \leftarrow window[0].w_{\text{local}}$
    $window[1].C_{\text{foreign}} \leftarrow window[0].C_{\text{foreign}} + window[1].\alpha_{\text{final}}/window[1].w_{\text{local}}$
  **end if**
**end function**

**Algorithm 22** Window Compute Foreign Interval: Compute foreign multiplicity from START/END cumulative sums

**function** WINDOWCOMPUTEFOREIGNINTERVAL($window$, $i$)
  **if** $window[0]$.type = START and $window[1]$.type = END **then**
    foreign_interval $\leftarrow window[1].C_{\text{foreign}} - window[0].C_{\text{foreign}}$
    $window[1]$.foreign_interval $\leftarrow$ foreign_interval
    $window[1].F_{\text{sum}} \leftarrow window[0].C_{\text{foreign}}$ ▷ Record alignment position
  **end if**
**end function**

**Algorithm 23** Update Target Final Multiplicity: Propagate foreign intervals to compute final multiplicities

**function** UPDATETARGETFINALMULTIPLICITY($e$, $t$)
  $t.\alpha_{\text{final}} \leftarrow e$.foreign_interval $\times\ t.\alpha_{\text{local}}$
  $t.F_{\text{sum}} \leftarrow e.F_{\text{sum}}$ ▷ For alignment
**end function**

## 5.5 Phase 3: Distribution and Expansion

Each tuple must be replicated according to its final multiplicity $\alpha_{\text{final}}$. We use the oblivious distribute-and-expand technique from ODBJ [8], which creates exactly $\alpha_{\text{final}}$ copies of each tuple while maintaining oblivious access patterns. This technique first distributes tuples to their target positions, then expands them to fill the required space. The key property is that the expansion is data-oblivious: the access pattern depends only on the multiplicities, not on the actual data values.

## 5.6 Phase 4: Alignment and Concatenation

After expansion, tables must be aligned so that matching tuples appear in the same rows. The parent table is sorted by join attributes (and secondarily by other attributes for deterministic ordering), creating groups of identical tuples. Each group represents a distinct combination from the parent table that will be matched with corresponding child tuples.

The child table alignment uses the formula $F_{\text{sum}} + (\text{copy\_index} \div \alpha_{\text{local}})$, where:

- $F_{\text{sum}}$ is the index of the first parent group that matches this child tuple
- copy_index is the index of this copy among all copies of the same original tuple (0 to $\alpha_{\text{final}} - 1$)
- $\alpha_{\text{local}}$ is the child tuple's local multiplicity

This formula ensures that every $\alpha_{\text{local}}$ copies of a child tuple increment to the next parent group, correctly distributing child copies across matching parent groups. After sorting by this alignment key, corresponding rows from parent and child tables are horizontally concatenated to form the partial join result. This process continues recursively through the join tree until all tables are combined.

**Algorithm 24** Result Construction

**function** ConstructJoinResult($T$, $root$)
    $result \leftarrow$ ObliviousExpand($R_{root}$) ▷ Expand root table
    **for all** nodes $v$ in pre-order traversal of $T$ from $root$ **do**
        **for all** child nodes $c$ of $v$ **do**
            $R_c^{expanded} \leftarrow$ ObliviousExpand($R_c$) ▷ Expand child table
            $result \leftarrow$ AlignAndConcatenate($result$, $R_c^{expanded}$)
        **end for**
    **end forreturn** $result$
**end function**

**Algorithm 25** Align and Concatenate

**function** AlignAndConcatenate($R_{accumulator}$, $R_{child}$)
    ObliviousSort($R_{accumulator}$, JoinThenOtherAttributes) ▷ Sort by join attrs, then others
    ParallelPass($R_{child}$, ComputeAlignmentKey) ▷ Set alignment key for each tuple
    ObliviousSort($R_{child}$, AlignmentKeyComparator)
    **return** HorizontalConcatenate($R_{accumulator}$, $R_{child}$)
**end function**

**Algorithm 26** Compute Alignment Key

**function** ComputeAlignmentKey($tuple$)
    $tuple$.alignment_key $\leftarrow tuple.F_{\text{sum}} + (tuple.\text{copy\_index} \div tuple.\alpha_{\text{local}})$
**end function**

**Algorithm 27** Join Then Other Attributes Comparator

**function** JoinThenOtherAttributes($t_1$, $t_2$)
    **if** $t_1$.join_attr $< t_2$.join_attr **then return** -1
    **else if** $t_1$.join_attr $> t_2$.join_attr **then return** 1
    **elsereturn** CompareOtherAttributes($t_1$, $t_2$) ▷ Secondary sort for deterministic ordering
    **end if**
**end function**

**Algorithm 28** Alignment Key Comparator

```
1: function AlignmentKeyComparator(t_1, t_2)
2:     if t_1.alignment_key < t_2.alignment_key then return -1
3:     else if t_1.alignment_key > t_2.alignment_key then return 1
4:     elsereturn 0
5:     end if
6: end function
```

# Chapter 6

# Security Analysis

This chapter provides a formal security analysis of our oblivious multi-way band join algorithm. We prove that the algorithm's memory access patterns reveal no information about the input data beyond what is explicitly allowed (table sizes and tree structure). Our proof follows a modular approach, building from simple components to the complete algorithm using the composition theorem for oblivious operations.

## 6.1 Security Model and Definitions

We begin by formally defining oblivious operations and stating the composition theorem that underlies our security proof.

### 6.1.1 Oblivious Operations

**Definition 6.1** (Oblivious Operation)**.** *An operation $\mathcal{O} : \mathcal{D} \to \mathcal{D}'$ is* oblivious *if for any two input sequences $X, Y \in \mathcal{D}$ with $|X| = |Y|$, the access patterns $\mathcal{AP}(\mathcal{O}(X))$ and $\mathcal{AP}(\mathcal{O}(Y))$ are identically distributed.*

Intuitively, an oblivious operation accesses memory in a pattern that depends only on the size of the input, not on the actual data values. An adversary observing the memory accesses learns nothing about the data beyond its size.

### 6.1.2 Composition Theorem

The following theorem, standard in the oblivious algorithms literature, allows us to build complex oblivious algorithms from simple oblivious components:

**Theorem 6.2** (Sequential Composition)**.** *If $\mathcal{O}_1 : \mathcal{D} \to \mathcal{D}'$ and $\mathcal{O}_2 : \mathcal{D}' \to \mathcal{D}''$ are oblivious operations, then their sequential composition $(\mathcal{O}_2 \circ \mathcal{O}_1) : \mathcal{D} \to \mathcal{D}''$ defined by $(\mathcal{O}_2 \circ \mathcal{O}_1)(x) = \mathcal{O}_2(\mathcal{O}_1(x))$ is also oblivious.*

*Proof.* For any inputs $x, y \in \mathcal{D}$ with $|x| = |y|$:

1. $\mathcal{AP}(\mathcal{O}_1(x)) \equiv \mathcal{AP}(\mathcal{O}_1(y))$ since $\mathcal{O}_1$ is oblivious
2. Let $x' = \mathcal{O}_1(x)$ and $y' = \mathcal{O}_1(y)$. Since $\mathcal{O}_1$ is oblivious, $|x'| = |y'|$
3. $\mathcal{AP}(\mathcal{O}_2(x')) \equiv \mathcal{AP}(\mathcal{O}_2(y'))$ since $\mathcal{O}_2$ is oblivious
4. Therefore, $\mathcal{AP}((\mathcal{O}_2 \circ \mathcal{O}_1)(x)) \equiv \mathcal{AP}((\mathcal{O}_2 \circ \mathcal{O}_1)(y))$

Hence, $\mathcal{O}_2 \circ \mathcal{O}_1$ is oblivious. □

### 6.1.3 Security Goal

Our security goal is to prove the following theorem:

**Theorem 6.3** (Main Security Theorem)**.** *The oblivious multi-way band join algorithm is oblivious. That is, for any two sets of input tables with the same sizes, tree structure, and output size, the memory access patterns are identically distributed.*

We prove this theorem through a hierarchical approach, starting with individual components and building up to the complete algorithm.

## 6.2 Level 1: Base Component Security

We first prove that our custom window functions, comparators, and update functions can be converted to oblivious implementations. Our conversion strategy relies on two key techniques:

1. **Arithmetic conversion**: Replace all conditional branches with arithmetic operations using 0/1 predicates. For any condition, we compute a predicate $p \in \{0, 1\}$ and use multiplication: result $= p \cdot \text{value}_{\text{true}} + (1 - p) \cdot \text{value}_{\text{false}}$.

2. **Access pattern uniformity**: Ensure all execution paths access the same memory locations in the same order, regardless of data values.

This approach transforms data-dependent control flow into data-oblivious arithmetic operations, ensuring that the memory access pattern is independent of the input data values.

### 6.2.1 Window Functions

**Lemma 6.4.** WindowComputeLocalSum *(Algorithm 13) can be converted to an oblivious implementation.*

*Proof.* The function's conditional logic on entry type can be converted to oblivious form:

1. **Access pattern**: Always reads window[0].local_sum and window[1].type, window[1].$\alpha_{\text{local}}$, and always writes to window[1].local_sum.

2. **Arithmetic conversion**: The conditional branch becomes:

$$\begin{aligned} \text{is\_source} &= (\text{window}[1].\text{type} == \text{SOURCE}) \in \{0, 1\} \\ \text{window}[1].\text{local\_sum} &= \text{window}[0].\text{local\_sum} \\ &\quad + \text{is\_source} \cdot \text{window}[1].\alpha_{\text{local}} \end{aligned}$$

This eliminates the conditional branch while preserving functionality: when SOURCE, adds $\alpha_{\text{local}}$; otherwise adds 0. □

**Lemma 6.5.** WindowComputeLocalInterval *(Algorithm 15) can be converted to an oblivious implementation.*

*Proof.* The function's conditional interval computation can be made oblivious:

1. **Access pattern**: Always read window[0] and window[1] fields, always write to window[1].local_interval.

2. **Arithmetic conversion**: The conditional check becomes:

$$\begin{aligned}
\text{is_pair} &= (\text{window}[0].\text{type} == \text{START}) \\
&\quad \cdot (\text{window}[1].\text{type} == \text{END}) \in \{0, 1\} \\
\text{interval} &= \text{window}[1].\text{local_sum} - \text{window}[0].\text{local_sum} \\
\text{window}[1].\text{local_interval} &= \text{is_pair} \cdot \text{interval} \\
&\quad + (1 - \text{is_pair}) \cdot \text{window}[1].\text{local_interval}
\end{aligned}$$

The write always happens (either new interval or preserving existing value). □

**Lemma 6.6.** WindowComputeForeignSum *(Algorithm 21) can be converted to an oblivious implementation.*

*Proof.* The function's three-way branch can be converted to arithmetic operations:

1. **Access pattern**: Always read window[0] fields and window[1] fields, always write to window[1].$w_{\text{local}}$ and window[1].$C_{\text{foreign}}$.
2. **Arithmetic conversion**: The type-based branching becomes:

$$\begin{aligned}
\text{is_start} &= (\text{window}[1].\text{type} == \text{START}) \in \{0, 1\} \\
\text{is_end} &= (\text{window}[1].\text{type} == \text{END}) \in \{0, 1\} \\
\text{is_source} &= (\text{window}[1].\text{type} == \text{SOURCE}) \in \{0, 1\} \\
\text{weight_delta} &= \text{is_start} \cdot \text{window}[1].\alpha_{\text{local}} \\
&\quad - \text{is_end} \cdot \text{window}[1].\alpha_{\text{local}} \\
\text{window}[1].w_{\text{local}} &= \text{window}[0].w_{\text{local}} + \text{weight_delta} \\
\text{safe_denom} &= \text{is_source} \cdot \text{window}[0].w_{\text{local}} + (1 - \text{is_source}) \cdot 1 \\
\text{foreign_delta} &= \text{is_source} \cdot (\text{window}[1].\alpha_{\text{final}} / \text{safe_denom}) \\
\text{window}[1].C_{\text{foreign}} &= \text{window}[0].C_{\text{foreign}} + \text{foreign_delta}
\end{aligned}$$

The safe denominator ensures division is never by zero: it uses the actual weight for SOURCE entries and 1 otherwise. □

**Lemma 6.7.** WindowComputeForeignInterval *(Algorithm 22) can be converted to an oblivious implementation.*

*Proof.* The function's conditional logic can be made oblivious:

1. **Access pattern**: Always read window[0] and window[1] fields, always write to window[1].foreign_interval and window[1].$F_{\text{sum}}$.

2. **Arithmetic conversion**: The conditional becomes:

$$\begin{aligned}
\text{is_pair} &= (\text{window}[0].\text{type} == \text{START}) \\
&\quad \cdot (\text{window}[1].\text{type} == \text{END}) \in \{0, 1\} \\
\text{interval} &= \text{window}[1].C_{\text{foreign}} - \text{window}[0].C_{\text{foreign}} \\
\text{window}[1].\text{foreign_interval} &= \text{is_pair} \cdot \text{interval} \\
&\quad + (1 - \text{is_pair}) \cdot \text{window}[1].\text{foreign_interval} \\
\text{window}[1].F_{\text{sum}} &= \text{is_pair} \cdot \text{window}[0].C_{\text{foreign}} \\
&\quad + (1 - \text{is_pair}) \cdot \text{window}[1].F_{\text{sum}}
\end{aligned}$$

□

### 6.2.2 Comparators

**Lemma 6.8.** ComparatorJoinAttr *(Algorithm 12) can be converted to an oblivious implementation.*

*Proof.* The comparator's conditional logic can be made oblivious:

1. **Access pattern**: Always read both elements' join_attr, type, and equality fields, and always access the precedence table.

2. **Arithmetic conversion**: Convert the nested conditionals to arithmetic:

$$\begin{aligned}
\text{cmp} &= \text{sign}(e_1.\text{join_attr} - e_2.\text{join_attr}) \in \{-1, 0, 1\} \\
\text{is_equal} &= (\text{cmp} == 0) \in \{0, 1\} \\
p_1 &= \text{GetPrecedence}(e_1.\text{type}, e_1.\text{equality}) \\
p_2 &= \text{GetPrecedence}(e_2.\text{type}, e_2.\text{equality}) \\
\text{prec_cmp} &= \text{sign}(p_1 - p_2) \in \{-1, 0, 1\} \\
\text{result} &= (1 - \text{is_equal}) \cdot \text{cmp} + \text{is_equal} \cdot \text{prec_cmp}
\end{aligned}$$

The precedence lookup uses both type and equality type fields as indices. □

**Lemma 6.9.** ComparatorPairwise *(Algorithm 14) can be converted to an oblivious implementation.*

*Proof.* The comparator has three-level comparison logic that can be made oblivious:

1. **Access pattern**: Always read both elements' type and orig_idx fields.
2. **Arithmetic conversion**: Convert the three-level priority system:

$$\begin{aligned}
\text{is_target}_1 &= (e_1.\text{type} \in \{\text{START}, \text{END}\}) \in \{0, 1\} \\
\text{is_target}_2 &= (e_2.\text{type} \in \{\text{START}, \text{END}\}) \in \{0, 1\} \\
\text{type_priority} &= \text{is_target}_2 - \text{is_target}_1 \in \{-1, 0, 1\} \\
\text{idx_cmp} &= \text{sign}(e_1.\text{orig_idx} - e_2.\text{orig_idx}) \in \{-1, 0, 1\} \\
\text{is_start}_1 &= (e_1.\text{type} == \text{START}) \in \{0, 1\} \\
\text{is_start}_2 &= (e_2.\text{type} == \text{START}) \in \{0, 1\} \\
\text{start_first} &= \text{is_start}_1 - \text{is_start}_2 \in \{-1, 0, 1\} \\
\text{same_priority} &= (\text{type_priority} == 0) \in \{0, 1\} \\
\text{same_index} &= (\text{idx_cmp} == 0) \in \{0, 1\} \\
\text{result} &= (1 - \text{same_priority}) \cdot \text{type_priority} \\
&\quad + \text{same_priority} \cdot (1 - \text{same_index}) \cdot \text{idx_cmp} \\
&\quad + \text{same_priority} \cdot \text{same_index} \cdot \text{start_first}
\end{aligned}$$

Priority order: (1) Target entries before SOURCE, (2) by original index, (3) START before END. □

### 6.2.3 Update Functions

**Lemma 6.10.** UpdateTargetMultiplicity *(Algorithm 17) is inherently oblivious.*

*Proof.* The function performs pure arithmetic:

1. **Access pattern**: Always read from both $t$ and $e$, always write to $t.\alpha_{\text{local}}$.
2. **No conversion needed**: The multiplication $t.\alpha_{\text{local}} \times e.\text{local\_interval}$ is already oblivious.

□

**Lemma 6.11.** UpdateTargetFinalMultiplicity *(Algorithm 23) is inherently oblivious.*

*Proof.* The function performs pure arithmetic:

1. **Access pattern**: Always read $e$.foreign_interval, $e.F_{\text{sum}}$, and $t.\alpha_{\text{local}}$, always write to $t.\alpha_{\text{final}}$ and $t.F_{\text{sum}}$.
2. **No conversion needed**: The operations $t.\alpha_{\text{final}} = e.\text{foreign\_interval} \times t.\alpha_{\text{local}}$ and $t.F_{\text{sum}} = e.F_{\text{sum}}$ are pure arithmetic/assignment.

□

## 6.3 Level 2: Composed Operation Security

Having shown that our base components can be converted to oblivious implementations, we now prove that composing these converted oblivious versions with established oblivious primitives yields oblivious operations.

### 6.3.1 Oblivious Primitives

We rely on the following well-established oblivious primitives:

**Assumption 6.12.** *The following operations are oblivious:*

- ObliviousSort*: Uses Batcher's bitonic sort [1] with a fixed comparison network*
- ObliviousExpand*: From ODBJ [8], expands tables obliviously*
- LinearPass*: Iterates through a table with fixed window size 2*
- ParallelPass*: Applies a function to each element independently*
- Map*: Transforms each element independently*

### 6.3.2 Composed Operations

**Lemma 6.13.** *For any comparator $C$ that can be converted to oblivious form,* ObliviousSort$(T, C_{obliviou}$ *is oblivious.*

*Proof.* By Assumption 6.12, ObliviousSort has a fixed comparison pattern based only on table size. By Lemmas 6.8-6.9, our comparators can be converted to oblivious implementations. Using the converted oblivious versions $C_{oblivious}$ and applying Theorem 6.2, the composition is oblivious. □

**Lemma 6.14.** *For any window function $W$ that can be converted to oblivious form,* LinearPass$(T, W_{oblivious})$ *is oblivious.*

*Proof.* LinearPass has a deterministic iteration pattern based only on table size (with fixed window size 2). By Lemmas 6.4-6.7, our window functions can be converted to oblivious implementations. Using the converted versions $W_{oblivious}$ and applying Theorem 6.2, the composition is oblivious. □

**Lemma 6.15.** *For any update function $U$ that is inherently oblivious or can be converted to oblivious form,* ParallelPass$(T, U_{oblivious})$ *is oblivious.*

*Proof.* ParallelPass applies $U$ to each element independently with a fixed access pattern. By Lemmas 6.10-6.11, our update functions are inherently oblivious (pure arithmetic). The parallel application maintains obliviousness. □

## 6.4 Level 3: Phase Security

We prove that each phase of our algorithm is oblivious.

### 6.4.1 Initialization Phase

**Lemma 6.16.** *The Initialization phase (Algorithm 5) is oblivious.*

*Proof.* Initialization consists of:

1. Map to add metadata columns

2. LinearPass with WindowSetOriginalIndex (Algorithm 7)

Both operations access each element exactly once in a predetermined order. By Theorem 6.2, their composition is oblivious. □

### 6.4.2 Bottom-Up Phase

**Lemma 6.17.** *The Bottom-Up phase is oblivious.*

*Proof.* For each node in post-order (public tree structure), the phase performs (Algorithm 10):

$$\begin{aligned}\text{BottomUp} = \text{CombineTable (Algorithm 11)} \\ \to \text{ObliviousSort(ComparatorJoinAttr)} \\ \to \text{LinearPass(WindowComputeLocalSum)} \\ \to \text{ObliviousSort(ComparatorPairwise)} \\ \to \text{LinearPass(WindowComputeLocalInterval)} \\ \to \text{ObliviousSort(ComparatorEndFirst)} \\ \to \text{ParallelPass(UpdateTargetMultiplicity)}\end{aligned}$$

Each operation is oblivious by Lemmas 6.13-6.15. The number of iterations depends only on the public tree structure. By repeated application of Theorem 6.2, the entire phase is oblivious. □

### 6.4.3 Top-Down Phase

**Lemma 6.18.** *The Top-Down phase is oblivious.*

*Proof.* The structure mirrors the Bottom-Up phase but with pre-order traversal and different window/update functions (Algorithm 20). Each component operation is oblivious by the same arguments. By Theorem 6.2, the phase is oblivious. □

### 6.4.4 Distribution and Expansion Phase

**Lemma 6.19.** *The Distribution and Expansion phase is oblivious.*

*Proof.* This phase applies ObliviousExpand to each table. By Assumption 6.12, ObliviousExpand is oblivious. The operation is applied to each table independently based on the public tree structure. □

### 6.4.5 Alignment and Concatenation Phase

**Lemma 6.20.** *The Alignment and Concatenation phase is oblivious.*

*Proof.* For each parent-child pair, the phase performs:

1. ObliviousSort on parent table
2. ParallelPass to compute alignment keys
3. ObliviousSort on child table
4. HorizontalConcatenate

Each operation is oblivious, and their composition is oblivious by Theorem 6.2. □

## 6.5 Level 4: Complete Algorithm Security

We now prove our main security theorem.

*Proof of Theorem 6.3.* The complete algorithm performs:

$$\begin{aligned}\text{Algorithm} = {} & \text{Initialization} \\ & \to \text{Bottom-Up Phase} \\ & \to \text{Top-Down Phase} \\ & \to \text{Distribution \& Expansion} \\ & \to \text{Alignment \& Concatenation}\end{aligned}$$

By Lemmas 6.16, 6.17, 6.18, 6.19, and 6.20, each phase is oblivious.

By repeated application of Theorem 6.2 (sequential composition), the complete algorithm is oblivious.

Therefore, for any two sets of input tables with the same sizes, tree structure, and output size, the memory access patterns are identically distributed, revealing no information about the actual data values, join selectivities, or which tuples match. □

## 6.6 Memory Access Pattern Analysis

*[This section is reserved for detailed analysis of memory access patterns and will be completed in future work.]*

## 6.7 Summary

We have proven that our oblivious multi-way band join algorithm maintains complete data obliviousness through a modular security proof. The proof builds from simple oblivious components (window functions, comparators, update functions) through composed operations and phases, ultimately establishing that the complete algorithm reveals no information through its memory access patterns beyond what is explicitly allowed (table sizes and tree structure).

The security guarantee holds even for band joins with inequality constraints, where the number of matching tuples and the distribution of values within ranges remain completely hidden from any adversary observing the execution.

# Chapter 7

# Evaluation

We evaluate our oblivious multi-way band join algorithm by comparing its performance against OJOIN [7], the state-of-the-art oblivious multi-way join algorithm. Our experiments use the same TPC-H benchmark setup to ensure a fair comparison, focusing on both multi-way equality joins and band joins with inequality constraints.

## 7.1 Implementation

We implemented our algorithm in C++ for Intel SGX2.

### 7.1.1 Data Preprocessing

To simplify the implementation and focus on the core algorithmic performance, we preprocess the TPC-H tables as follows:

- **Type conversion**: All date and string fields are converted to integers. Dates are represented as days since a reference date (1970-01-01), while strings are mapped to integer identifiers through a preprocessing dictionary.
- **Table duplication**: When a query requires using the same table multiple times (self-joins or multiple references), we create distinct copies with appropriate renaming. This simplifies the tree structure handling without affecting the algorithm's complexity.

- **Memory layout**: Tables are stored as arrays of row objects within the enclave. This row-oriented storage is a source of performance overhead compared to column-oriented formats that would better match our sequential access patterns.

## 7.2 Experimental Setup

### 7.2.1 Dataset and Queries

We use the TPC-H benchmark with scale factor 0.1 (SF=0.1), matching the setup used in the OJOIN [7] evaluation. This generates approximately 100MB of raw data across eight tables. We evaluate pure `SELECT-FROM-WHERE` queries without subqueries or aggregation operations, focusing on the core join performance. The queries from the OJOIN [7] paper are:

**Multi-way Equality Joins (TM series):**

- **TM1**: 3-way join between `lineitem`, `orders`, and `customer`
- **TM2**: 4-way join between `lineitem`, `orders`, `customer`, and `nation`
- **TM3**: 5-way join adding `region` to TM2

**Band Joins (TB series):**

- **TB1**: Band join query with date range constraints on `lineitem`
- **TB2**: Extended band join with overlapping date ranges across `lineitem` and `orders`

All queries follow the standard SQL pattern without `GROUP BY`, `HAVING`, subqueries, or aggregate functions. This allows us to focus purely on the join algorithm performance without the complexity of aggregation processing.

The specific band join queries tested are:

**TB1 Query:**

```
SELECT * FROM lineitem, orders
WHERE l_orderkey = o_orderkey
  AND l_shipdate >= o_orderdate
  AND l_shipdate <= o_orderdate + 30
```

**TB2 Query:**

```
SELECT * FROM lineitem l1, lineitem l2, orders
WHERE l1.l_orderkey = o_orderkey
  AND l2.l_orderkey = o_orderkey
  AND l1.l_shipdate >= o_orderdate
  AND l1.l_shipdate <= o_orderdate + 30
  AND l2.l_shipdate >= o_orderdate + 15
  AND l2.l_shipdate <= o_orderdate + 45
```

These queries test band join performance with date range constraints, where TB1 performs a single band join and TB2 involves overlapping date ranges across multiple joins.

### 7.2.2 Hardware Configuration

Experiments were conducted on a server with the following specifications:

- Intel Xeon E-2374G processor @ 3.70GHz with SGX support
- 4 cores, 8 threads with AVX-512 support
- 125GB RAM (120GB available)
- Ubuntu 22.04.4 LTS with Linux kernel 5.15.0
- NVMe SSD storage

### 7.2.3 Metrics

We measure the total execution time for each query, comparing our algorithm's runtime against OJOIN [7].

## 7.3 Results: Multi-way Equality Joins

Table 7.1 shows the performance comparison between our algorithm and OJOIN [7] for multi-way equality joins.

Table 7.1: Performance comparison for multi-way equality joins

| Query | Scale Factor | Output Size | OJOIN (s) | Ours (s) | Speedup |
|---|---|---|---|---|---|
| TM2 | 0.001 | 292 | – | 0.77 | – |
| TM2 | 0.01 | 29,929 | 10 | 10.38 | 0.96× |
| TM2 | 0.1 | 2,999,594 | 100 | OOM[1] | – |

For TM2 at scale factor 0.01, our algorithm performs comparably to OJOIN [7] with both completing in approximately 10 seconds. However, at SF=0.1, the output size of nearly 3 million rows exceeds our current 512MB heap limit, while OJOIN completes in 100 seconds.

## 7.4 Results: Band Joins

Table 7.2 presents the results for band join queries with inequality constraints.

Table 7.2: Performance comparison for band joins at different scale factors

| Query | Scale Factor | Output Size | OJOIN (s) | Ours (s) | Speedup |
|---|---|---|---|---|---|
| TB1 | 0.001 | 10 | – | 2.58 | – |
| TB1 | 0.01 | 111 | – | 2.59 | – |
| TB1 | 0.1 | 2,042 | 100 | 2.95 | 33.9× |
| TB2 | 0.001 | 200 | – | 2.61 | – |
| TB2 | 0.01 | 4,002 | – | 3.50 | – |
| TB2 | 0.1 | 397,380 | 100,000 | OOM[2] | – |

Our dual-entry technique provides significant advantages for band joins. For TB1 at scale factor 0.1, we achieve a remarkable 33.9× speedup over OJOIN [7], completing the query in 2.95 seconds compared to OJOIN's 100 seconds. The performance remains consistent across different scale factors, demonstrating the efficiency of our approach.

For TB2, while we successfully process smaller scale factors, the SF=0.1 dataset produces 397,380 output rows which exceeds our current enclave heap limit of 512MB. OJOIN

[1]Out of memory error due to large output size (3M rows) exceeding enclave heap limit
[2]Out of memory error due to large output size (397K rows) exceeding enclave heap limit

requires 100,000 seconds (over 27 hours) for this query, highlighting the computational challenge of large band joins.

## 7.5 Discussion

### 7.5.1 Key Findings

Our evaluation demonstrates:

1. **Exceptional band join performance**: TB1 shows 33.9× speedup over OJOIN [7] at SF=0.1
2. **Competitive equality joins**: TM2 performs comparably to OJOIN at SF=0.01
3. **High SGX overhead on small data**: TB1 runtime remains nearly constant (2.58-2.95s) across scale factors due to SGX enclave overhead dominating on smaller datasets
4. **Memory limitations**: Large output sizes (TB2: 397K rows, TM2: 3M rows) exceed current 512MB heap limit

### 7.5.2 Future Improvements

- **Selective SGX execution**: Execute only the critical oblivious operations (window functions and comparators) inside SGX while keeping data outside the enclave. This would reduce memory requirements to constant space, enabling processing of arbitrarily large inputs and outputs.
- **Columnar storage**: Transition from row-oriented to column-oriented storage for better cache utilization and sequential access patterns.
- **Streaming preprocessing**: Integrate type conversion and table preparation into the main algorithm to avoid separate preprocessing passes.
- **Cyclic query support**: Extend the algorithm to handle cyclic join graphs using generalized hypertree decomposition.

## 7.6 Summary

Our oblivious multi-way band join algorithm demonstrates substantial performance improvements over the state-of-the-art OJOIN [7] algorithm, particularly for band join queries. For TB1 at scale factor 0.1, we achieve a remarkable 33.9× speedup, reducing execution time from 100 seconds to just 2.95 seconds. For equality joins like TM2, we achieve comparable performance at smaller scale factors. The dual-entry technique proves highly effective for handling inequality constraints while maintaining complete data obliviousness. While memory constraints limit processing of very large result sets (millions of output rows), our implementation in SGX demonstrates the practicality of our approach for secure database applications.

# Chapter 8

# Conclusion

This thesis presented the first oblivious algorithm for multi-way band joins, addressing a critical gap in secure database query processing. By adapting the classical Yannakakis algorithm to the oblivious computation model and introducing novel techniques for handling inequality constraints, we achieved both theoretical elegance and practical performance improvements.

## 8.1 Summary of Contributions

Our work makes three primary contributions to the field of oblivious database algorithms:

**1. Oblivious Adaptation of Yannakakis Algorithm**: We successfully transformed the classical Yannakakis algorithm for acyclic joins into a fully oblivious version. This required careful redesign of the two-phase semi-join approach, replacing data-dependent operations with oblivious primitives while preserving the algorithm's optimal complexity. Our adaptation maintains the algorithm's elegance while ensuring that memory access patterns reveal nothing about the input data.

**2. Dual-Entry Technique for Band Joins**: We introduced a novel dual-entry approach that enables efficient processing of inequality constraints in an oblivious manner. By creating START and END entries for range boundaries and using window-based computation, we can handle band joins without the exponential blowup that would result from naive approaches. This technique proved particularly effective, achieving a 33.9× speedup over OJOIN for the TB1 query.

**3. Rigorous Security Analysis**: We provided a comprehensive four-level security proof, demonstrating that our algorithm maintains complete data obliviousness. Starting from base components (window functions, comparators) and building up through composed operations, phases, and the complete algorithm, we proved that all memory access patterns are independent of input data values.

## 8.2 Experimental Validation

Our implementation in Intel SGX demonstrated the practicality of our approach:

- For band join query TB1 at scale factor 0.1, we achieved execution in 2.95 seconds compared to OJOIN's 100 seconds—a 33.9× improvement
- For equality join TM2 at scale factor 0.01, we achieved comparable performance to OJOIN (approximately 10 seconds)

These results validate that sophisticated algorithmic techniques can overcome the performance penalties typically associated with oblivious computation.

## 8.3 Practical Implications

This work contributes to oblivious database research in several ways:

**Practical Secure Analytics**: By dramatically improving band join performance, we make secure processing of temporal and range queries practical for real applications. This is crucial for privacy-preserving analytics on sensitive data such as medical records or financial transactions.

**Dual-Entry Technique**: The dual-entry approach provides a new method for handling inequality constraints obliviously, which could be useful for other researchers working on similar problems.

**Implementation Experience**: Our SGX implementation provides insights into the practical challenges of deploying oblivious algorithms in secure hardware, including memory constraints and performance overheads.

## 8.4 Future Directions

Several promising avenues extend from this work:

**Complete SQL Engine**: Implementing a full oblivious SQL engine that supports GROUP BY, aggregation functions, and subqueries would enable processing of more complex analytical queries beyond simple joins.

**Columnar Memory Layout**: Transitioning from the current row-oriented storage to a columnar format would improve cache utilization and enable more efficient sequential access patterns, potentially providing significant performance gains.

**Extending to Cyclic Queries**: While our current algorithm handles acyclic join trees, many real queries involve cycles. Extending our techniques to work with generalized hypertree decompositions would broaden applicability.

## 8.5 Closing Remarks

This thesis addressed the specific problem of performing multi-way band joins obliviously, demonstrating that adapting classical algorithms to secure computation models can yield significant performance improvements. The 33.9× speedup achieved for TB1 queries shows that oblivious computation does not necessarily require accepting poor performance.

While memory constraints currently limit the size of queries we can process, the path forward is clear: selective execution of only the critical oblivious operations within SGX would enable handling of much larger datasets. Our work provides a foundation for continued research in oblivious database algorithms, contributing one piece to the larger puzzle of building practical privacy-preserving systems.

# References


[1] Kenneth E Batcher. Sorting networks and their applications. *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314, 1968.

[2] Zhaoyan Chang, Dong Xie, and Feifei Li. Oblivious query processing. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 1445–1458. IEEE, 2022.

[3] Victor Costan and Srinivas Devadas. Intel sgx explained. Technical Report 2016/086, IACR Cryptology ePrint Archive, 2016.

[4] Saba Eskandarian and Matei Zaharia. Oblidb: Oblivious query processing for secure databases. *Proceedings of the VLDB Endowment*, 13(2):169–183, 2020.

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

[6] Michael T Goodrich. Zig-zag sort: a simple deterministic data-oblivious sorting algorithm running in o(n log n) time. *arXiv preprint arXiv:1403.2777*, 2014.

[7] Xiao Hu and Zhiang Wu. Optimal oblivious algorithms for multi-way joins. *arXiv preprint arXiv:2501.04216*, 2025.

[8] Simeon Krastnikov, Florian Kerschbaum, and Douglas Stebila. Efficient oblivious database joins. *Proceedings of the VLDB Endowment*, 13(11):2132–2145, 2020.

[9] Yilei Wang and Ke Yi. Secure yannakakis: Join-aggregate queries over private data. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)*, pages 1445–1457, 2021.

[10] Mihalis Yannakakis. Algorithms for acyclic database schemes. *Proceedings of the 7th International Conference on Very Large Data Bases*, pages 82–94, 1981.

[11] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 283–298, 2017.

# Glossary

This document is incomplete. The external file associated with the glossary 'main' (which should be called `uw-ethesis.gls`) hasn't been created.

Check the contents of the file `uw-ethesis.glo`. If it's empty, that means you haven't indexed any of your entries in this glossary (using commands like `\gls` or `\glsadd`) so this list can't be generated. If the file isn't empty, the document build process hasn't been completed.

You may need to rerun LaTeX. If you already have, it may be that TeX's shell escape doesn't allow you to run makeindex. Check the transcript file `uw-ethesis.log`. If the shell escape is disabled, try one of the following:

- Run the external (Lua) application:

  ```
  makeglossaries-lite "uw-ethesis"
  ```

- Run the external (Perl) application:

  ```
  makeglossaries "uw-ethesis"
  ```

Then rerun LaTeX on this document.

This message will be removed once the problem has been fixed.