

# Oblivious Multi-Way Band Joins: An Efficient Algorithm for Secure Range Queries

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

© Ruidi Wei 2025

## **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 challenges in oblivious computation. Our approach employs a dual-entry technique that transforms range matching into cumulative sum computations, enabling multiplicity computation in an oblivious manner. The algorithm achieves  $O(N \log N + k \cdot \text{OUT} \log \text{OUT})$  complexity, where  $k$  is the number of tables in the join query,  $N$  is the input size, and  $\text{OUT}$  is the output size, matching state-of-the-art oblivious equality joins up to a factor of  $k$  while supporting full band constraints. We implement the method using Intel SGX with batch processing and evaluate it on the TPC-H benchmark dataset [17], demonstrating practical performance and strong obliviousness guarantees under an honest-but-curious adversary model.

## Acknowledgements

First and foremost, I would like to express my deepest gratitude to my supervisor, Professor Florian Kerschbaum, for his guidance, patience, and unwavering support throughout this research. His deep expertise in secure computation and insightful feedback were instrumental in shaping this work.

I am also thankful to the members of my thesis committee, Professor Sujaya Maiyya and Professor Xiao Hu, for their time and valuable feedback that helped improve this thesis.

Special thanks to Qinjia Yu for providing crucial intuitions that laid the foundation of this research, particularly the insights in “The Multi-Way Multiplicity Intuition” section. Especially grateful for the discussion on research direction during the algorithm design phase, and for editorial assistance in improving the clarity and presentation of this thesis.

Lastly, I could not have accomplished this journey without my family members, Dan Juan Yu, Dan Bing Yu, Kiki Fox, and Chicago Dan, for their unconditional love and support. Their boundless patience and endless companionship provided me with a spiritual anchor while structuring this work. Thank you for always being by my side, facing challenges together, sharing the joy of progress, and reminding me of where my heart lies.

# Table of Contents

<b>Author’s Declaration</b>	<b>ii</b>
<b>Abstract</b>	<b>iii</b>
<b>Acknowledgements</b>	<b>iv</b>
<b>List of Figures</b>	<b>x</b>
<b>List of Tables</b>	<b>xi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Problem Statement . . . . .	2
1.2 Contributions . . . . .	3
1.3 Thesis Organization . . . . .	3
<b>2 Related Work</b>	<b>5</b>
2.1 Prior Oblivious Join Approaches . . . . .	5
2.2 Efficient Oblivious Database Join . . . . .	6
2.3 Extension to Band Joins (Inequality Constraints) . . . . .	6
2.4 Multi-Way Joins (Classical Non-Oblivious) . . . . .	7
2.5 Worst-Case Optimal Join Algorithms . . . . .	7
2.6 Critical Gap in the Literature . . . . .	8
2.7 Our Approach: Bridging the Gap . . . . .	9

<b>3</b>	<b>Background</b>	<b>10</b>
3.1	Database Joins and Query Processing . . . . .	10
3.1.1	Database Join Operations . . . . .	10
3.1.2	Join Trees and Query Structure . . . . .	11
3.1.3	Acyclic vs Cyclic Queries . . . . .	12
3.1.4	Handling Cyclic Queries with GHD . . . . .	12
3.2	Band Joins and Range Queries . . . . .	13
3.2.1	From Equality to Inequality Joins . . . . .	13
3.2.2	Why Band Joins are Challenging . . . . .	13
3.3	The Yannakakis Algorithm . . . . .	14
3.3.1	Optimal Processing for Acyclic Queries . . . . .	14
3.3.2	The Two-Phase Approach . . . . .	14
3.4	Oblivious Computation . . . . .	15
3.4.1	The Need for Oblivious Algorithms . . . . .	15
3.4.2	The Oblivious Security Model . . . . .	15
3.4.3	Building Blocks for Oblivious Algorithms . . . . .	15
3.5	Intel SGX and Secure Hardware . . . . .	17
3.5.1	Trusted Execution Environments . . . . .	17
3.5.2	Implementing Oblivious Joins in SGX . . . . .	17
<b>4</b>	<b>Algorithm Overview</b>	<b>20</b>
4.1	From ODBJ to Oblivious Yannakakis . . . . .	20
4.1.1	Starting with ODBJ's Architecture . . . . .	20
4.1.2	The Multi-Way Multiplicity Challenge . . . . .	21
4.1.3	Connection to Yannakakis . . . . .	23
4.2	Band Join Enhancement: Dual Entry Approach . . . . .	23
4.2.1	The Challenge of Range-Based Multiplicity Computation . . . . .	23
4.2.2	The Dual Entry Solution . . . . .	23

<b>5</b>	<b>Detailed Algorithm</b>	<b>25</b>
5.1	Algorithm Overview and Notation . . . . .	25
5.1.1	Algorithm Input and Output . . . . .	25
5.1.2	Table Type Definitions . . . . .	25
5.1.3	Data Structures and Notation . . . . .	27
5.1.4	Formal Definitions of Multiplicities . . . . .	27
5.1.5	Common Utilities Across Multiple Phases . . . . .	29
5.1.6	Algorithm Structure . . . . .	32
5.2	Initialization . . . . .	33
5.3	Phase 1: Bottom-Up Multiplicity Computation . . . . .	34
5.4	Phase 2: Top-Down Final Multiplicity Propagation . . . . .	40
5.5	Phase 3: Distribution and Expansion . . . . .	44
5.6	Phase 4: Alignment and Concatenation . . . . .	44
<b>6</b>	<b>Complexity Analysis</b>	<b>47</b>
6.1	Phase 1: Bottom-Up Multiplicity Computation . . . . .	47
6.2	Phase 2: Top-Down Multiplicity Propagation . . . . .	48
6.3	Phase 3: Distribution and Expansion . . . . .	48
6.4	Phase 4: Alignment and Concatenation . . . . .	48
6.5	Total Complexity . . . . .	49
6.6	Counting Sorting Operations . . . . .	49
6.6.1	Sorting Operations in Current Algorithm . . . . .	50
6.6.2	Improved Sorting Count . . . . .	51
<b>7</b>	<b>Security Analysis</b>	<b>53</b>
7.1	Security Model and Definitions . . . . .	53
7.1.1	Oblivious Operations . . . . .	53
7.1.2	Composition Theorem . . . . .	54

7.1.3	Security Goal	54
7.2	Level 1: Base Component Security	54
7.2.1	Window Functions	55
7.2.2	Comparators	57
7.2.3	Update Functions	59
7.3	Level 2: Composed Operation Security	60
7.3.1	Oblivious Primitives	60
7.3.2	Composed Operations	60
7.4	Level 3: Phase Security	61
7.4.1	Initialization Phase	61
7.4.2	Bottom-Up Phase	61
7.4.3	Top-Down Phase	62
7.4.4	Distribution and Expansion Phase	62
7.4.5	Alignment and Concatenation Phase	62
7.5	Level 4: Complete Algorithm Security	63
<b>8</b>	<b>Evaluation</b>	<b>64</b>
8.1	Implementation	64
8.1.1	Data Preprocessing	64
8.1.2	Batch Processing	65
8.1.3	Oblivious Sorting Implementation	65
8.2	Experimental Setup	66
8.2.1	Dataset and Queries	66
8.2.2	Hardware Configuration	68
8.2.3	Metrics	69
8.3	Results: Band Joins	69
8.4	Results: Multi-way Equality Joins	70
8.5	Discussion	71
8.5.1	Key Findings	71
8.5.2	Future Improvements	72

<b>9 Conclusion</b>	<b>73</b>
9.1 Summary of Contributions . . . . .	73
9.2 Experimental Validation . . . . .	74
9.3 Practical Implications . . . . .	74
9.4 Future Directions . . . . .	75
<b>References</b>	<b>76</b>

# List of Figures

3.1	Example join tree showing five tables with their schemas and join conditions. Each node represents a table with its columns, and edges are labeled with the equi-join conditions. . . . .	11
4.1	Example tables with data showing local and final multiplicities in a 5-table join tree. Each table shows its tuples and their computed multiplicities after the bottom-up and top-down phases. . . . .	22

# List of Tables

2.1	Comparison of Existing Oblivious Join Approaches . . . . .	8
3.1	Oblivious Primitives Used in Our Algorithm . . . . .	19
5.1	Table Schema Evolution Throughout Algorithm Phases . . . . .	26
5.2	Algorithm Data Structures and Notation . . . . .	28
6.1	Comparison of sorting operations between current and improved algorithms. The improvement reduces input-sized sorts by $2k - 2$ operations by eliminating the end-first comparator sort. . . . .	50
8.1	Performance comparison for band joins at different scale factors . . . . .	69
8.2	Performance comparison for multi-way equality joins . . . . .	70

# 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 allow 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, memory access patterns alone can reveal sensitive information about the underlying data [8]—for example, the selectivity of join conditions or the distribution of values—even when the data itself is encrypted. 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 [24] 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 [22]. However, these adaptations handle only equality joins where tuples match on exact values. Band joins present a fundamental algorithmic challenge that goes beyond the obliviousness requirements shared with equality joins. While equality joins can leverage hash-based partitioning and exact-match indexing even in oblivious settings, band joins require matching against ranges of values, making these techniques inapplicable. Band joins cannot be partitioned into groups with clean boundaries, and computing these matches efficiently without revealing access patterns requires transforming the range-matching problem into operations that can be performed obliviously. While generic approaches like Oblivious RAM (ORAM) [8] could hide these patterns, they intro-

duce 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. We extend the Efficient Oblivious Database Joins (ODBJ) framework [14] in two dimensions: first generalizing from binary to multi-way joins using Yannakakis-style tree traversals, then from equality to band conditions through a dual-entry technique that transforms range matching into cumulative sum computations. We achieve  $O(N \log N + k \cdot \text{OUT} \log \text{OUT})$  complexity for acyclic queries, where  $k$  is the number of tables,  $N$  is the input size and  $\text{OUT}$  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 [6] and demonstrate its practicality on the TPC-H benchmark dataset [17].

## 1.1 Problem Statement

Multi-way joins with inequality predicates (band conditions) are essential for many applications, yet existing oblivious join algorithms handle only equality predicates. The challenge lies in the fundamental difference between equality and band joins: while equality joins match tuples with identical keys—enabling techniques like hash partitioning and exact-match indexing—band joins require matching against ranges of values, where tuples cannot be cleanly partitioned into matching groups.

In the oblivious setting, this distinction becomes critical. For equality joins, tuples can be partitioned into groups based on join keys, with each group in one table matching exactly one group in another. This one-to-one correspondence allows efficient oblivious processing through techniques like oblivious sorting and controlled group-wise operations. Band joins break this structure: the ranges that different tuples match can overlap, preventing clean partitioning into groups. This overlapping nature must be handled without revealing which ranges actually contain matches in the oblivious setting.

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 (see Section 3.1 for discussion of query acyclicity and transformations for cyclic queries). The challenge is to support inequality predicates ( $<$ ,  $>$ ,  $\leq$ ,  $\geq$ ) while maintaining data-independent access patterns, achieving comparable efficiency to oblivious equality joins without revealing information through operation counts or memory accesses.

## 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  $O(N \log N + k \cdot \text{OUT} \log \text{OUT})$  complexity for acyclic queries in the oblivious setting, where  $k$  is the number of tables,  $N$  is the input size, and  $\text{OUT}$  is the output size, matching the complexity of existing oblivious equality join algorithms up to a factor of  $k$  while supporting the full generality of range constraints.

At the core of our approach is a dual-entry technique for encoding range constraints obliviously. Unlike equality joins where tuples form groups with clean one-to-one correspondence between tables, band joins require matching against overlapping 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 Efficient Oblivious Database Joins (ODBJ) framework [14] 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 outputs from range queries do not reveal sensitive information through access patterns.

We implement our algorithm using batch processing in Intel SGX and provide a comprehensive experimental evaluation on the TPC-H benchmark dataset. Our security analysis provides a formal proof 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 existing oblivious join algorithms and identifies the gap that no solution combines multi-way joins with band conditions.

- Chapter 3 provides background on database joins, the Yannakakis algorithm, oblivious computation, and Intel SGX.
- Chapter 4 presents an intuitive overview of our approach, extending ODBJ to multi-way joins and introducing the dual-entry technique for band conditions.
- Chapter 5 describes the formal algorithm specification with detailed pseudocode.
- Chapter 6 analyzes the complexity of our algorithm, including detailed phase-by-phase analysis and sorting operation counts.
- Chapter 7 proves the algorithm maintains oblivious access patterns through a hierarchical security analysis.
- Chapter 8 evaluates performance on TPC-H benchmark queries, comparing against the state-of-the-art OJOIN [5] algorithm.
- Chapter 9 summarizes contributions and discusses future work.

# Chapter 2

## Related Work

This chapter reviews the literature on oblivious database operations, examining prior approaches to oblivious joins and identifying the gap in multi-way band joins that our work addresses.

### 2.1 Prior Oblivious Join Approaches

The development of oblivious database joins began with Agrawal et al. [1], who introduced the first oblivious join algorithms using secure coprocessors, though their security model assumed the maximum matches per tuple was public information, inadvertently leaking query selectivity. Li and Chen [16] addressed this privacy limitation by eliminating the maximum-matches assumption, achieving true privacy where only input sizes, schemas, and output size are revealed. Arasu and Kaushik [2] formalized the theoretical foundations by proving the equivalence between secure and oblivious query processing and introduced the algorithmic framework of dimension computation and table expansion, though key algorithmic details for the barely prefix heavy reordering were deferred and no implementation was provided. SMCQL [3] built a practical SQL system using garbled circuits for federated databases, but required  $O(n_1 n_2)$  join comparisons for oblivious evaluation. In the same year, Opaque [25] achieved efficient distributed oblivious sort-merge joins using Intel SGX enclaves, but was restricted to primary-foreign key joins where output size scales linearly with input. Conclave [19] adopted hybrid protocols that reveal join keys to a selectively-trusted party (STP) and output sizes to all parties, relaxing obliviousness for better performance. OblIDB [7] extended Opaque’s approach with additional operators

including hash join, though the hash join degenerates to quadratic complexity for general join cases beyond primary-foreign key relationships.

## 2.2 Efficient Oblivious Database Join

Krastnikov et al. [14] proposed the first efficient oblivious algorithm for binary database equi-joins. Their algorithm achieves  $O(N \log N + \text{OUT} \log \text{OUT})$  complexity where  $N$  is input size and  $\text{OUT}$  is output size, matching the standard non-oblivious sort-merge join up to a logarithmic factor using oblivious sorting with  $O(n \log n)$  complexity.

The 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.

While ODBJ is limited to **equality predicates only** and **binary joins** (two tables), its efficient oblivious primitives—particularly the multiplicity computation and result construction phases—provide essential building blocks for more complex join algorithms. As we show in this work, these primitives can be adapted to handle multi-way joins by applying them recursively across a join tree, and extended to support band conditions through novel techniques.

## 2.3 Extension to Band Joins (Inequality Constraints)

Chang et al. [5] made two 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.

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

The classical Yannakakis algorithm [24] 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 do not contribute to the final result
2. **Top-down phase:** Result reconstruction by propagating constraints down the tree

This approach eliminates tuples that do not 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.

## 2.5 Worst-Case Optimal Join Algorithms

Recent work by Hu and Wu [13] has made progress in developing oblivious algorithms for worst-case optimal multi-way joins, 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  $O(N \log N + k \cdot \text{OUT} \log \text{OUT})$  complexity on acyclic queries, where  $k$  is the number of tables,  $N$  is the input size, and  $\text{OUT}$  is the actual output size.

## 2.6 Critical Gap in the Literature

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

Table 2.1 summarizes the capabilities of existing approaches:

Table 2.1: Comparison of Existing Oblivious Join Approaches

Approach	Binary	Multi-way	Equality	Band
ODBJ (Krstnikov et al.)	✓		✓	
Chang et al. (binary)	✓		✓	✓
Opaque/ObliDB	✓	✓	✓	
Chang et al. (multi-way)	✓	✓	✓	
Hu and Wu (WCO)	✓	✓	✓	
<b>Our Work</b>	✓	✓	✓	✓

Opaque uses oblivious sort-merge join but is limited to primary-foreign key joins [25]. ObliDB supports general multi-way joins using hash join, but this approach essentially computes the Cartesian product, leading to poor performance [7, 5]. Hash-based join methods are particularly unsuitable for extension to range queries, as they rely on exact key matching rather than ordering.

A 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.

## 2.7 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** [14] as the base algorithm for processing neighboring table pairs in the join tree
- Extending our multi-way join approach to support **inequality predicates** through a dual-entry technique
- Achieving  $O(N \log N + k \cdot \text{OUT} \log \text{OUT})$  complexity for acyclic queries with full band join support, where  $k$  is the number of tables,  $N$  is the input size, and  $\text{OUT}$  is the output size
- 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 execution strategy for 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 the query on tables Leaf1(l1), Leaf2(l2), Leaf3(l3), Center(l1, l2, c), and Root(c, l3):

```
SELECT *  
FROM Leaf1, Leaf2, Leaf3, Center, Root  
WHERE Center.l1 = Leaf1.l1  
      AND Center.l2 = Leaf2.l2  
      AND Root.c = Center.c  
      AND Root.l3 = Leaf3.l3
```

we can construct the join tree as shown in Figure 3.1.

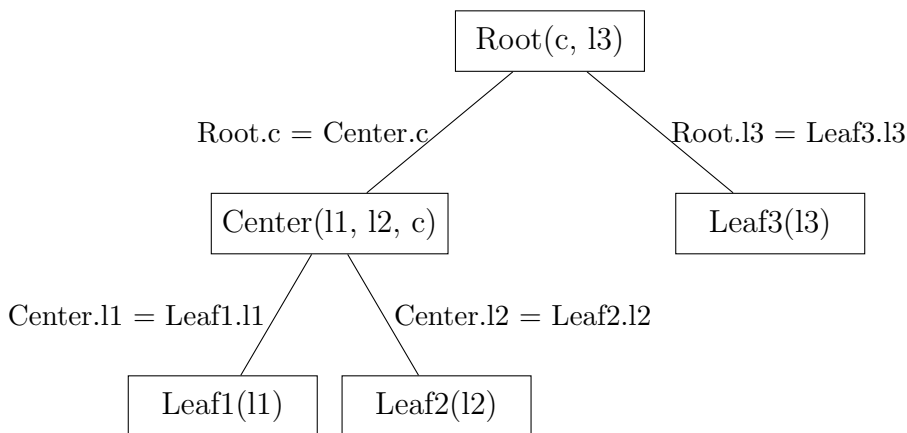


Figure 3.1: Example join tree showing five tables with their schemas and join conditions. Each node represents a table with its columns, and edges are labeled with the equi-join conditions.

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) [10] 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  $\text{GHW} = 1$  (no grouping needed), while the triangle query has  $\text{GHW} = 2$ , and a  $k$ -cycle has  $\text{GHW} = \lceil k/2 \rceil$ .

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, we cannot partition tuples into groups where each group in one table corresponds to exactly one group in another table. In equality joins, all tuples with value  $v$  in table A match all tuples with value  $v$  in table B—a clean group-to-group correspondence. Band joins break this structure: a tuple with value  $v$  might match tuples with values in  $[v - c_1, v + c_2]$ , overlapping with the ranges of other tuples. This lack of partitionability makes it difficult to predict resource requirements and optimize query execution. In the oblivious setting, this challenge is amplified because we must process overlapping ranges without revealing which tuples actually match.

## 3.3 The Yannakakis Algorithm

### 3.3.1 Optimal Processing for Acyclic Queries

The Yannakakis algorithm [24], 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  $\text{OUT}$  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—the size of intermediate results 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 [4, 11, 15, 18, 21, 23]. Even with encrypted data, the pattern of memory accesses during computation can leak sensitive information. For example, a binary search reveals the 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 any two 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”). Following the shuffle-then-reveal paradigm [12], the algorithm first obliviously shuffles the data

to randomize positions, then performs sorting where comparison outcomes no longer leak information about the original data distribution. This ensures that regardless of the input values, the memory access patterns reveal nothing about the actual data. Our implementation achieves  $O(n \log n)$  complexity using efficient oblivious sorting algorithms [12, 9], obtaining the theoretical guarantee of  $O(N \log N + k \cdot \text{OUT} \log \text{OUT})$  for our overall algorithm, where  $k$  is the number of tables,  $N$  is the input size and  $\text{OUT}$  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 to every row of a table. The function operates independently on each row, maintaining data-independent access patterns by reading and writing at predetermined positions 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.

**LinearPass** 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 pass 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.

**ParallelPass** 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 pass 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) [6] 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 uses a hybrid architecture that combines untrusted storage with trusted SGX processing. Tables are stored encrypted using AES-CTR, where each row is encrypted with a unique sequential nonce to enable fine-grained access. The untrusted application orchestrates the overall algorithm phases—managing linear passes, sorts, and parallel operations—while the SGX enclave performs the actual oblivious computations.

During execution, the untrusted application sends batches of 2000 operations to the enclave. For each batch, the enclave:

1. Receives encrypted row data for the batch
2. Decrypts the rows using their individual nonces

3. Performs the oblivious operations (window functions, comparators, or update functions)
4. Re-encrypts the modified rows with new nonces
5. Returns the encrypted results to untrusted storage

This batch-processing approach minimizes SGX overhead while maintaining complete obliviousness. The memory access patterns visible to the untrusted host reveal only the predetermined batch structure, not the actual data values or computation results. The combination of per-row encryption, batch processing, and oblivious operations within the enclave provides strong security guarantees against both data breaches and side-channel attacks.

Table 3.1: Oblivious Primitives Used in Our Algorithm

<b>Primitive</b>	<b>Description</b>	<b>Runtime</b>	<b>Reference</b>
Oblivious Sorting	Sorts data using shuffle-then-reveal approach for data-independent patterns	$O(n \log n)$	Oblivious sort [12, 9]
Oblivious Distribution	Moves rows to computed target positions without revealing data patterns	$O(n \log n)$	ODBJ [14]
Oblivious Expansion	Creates multiple copies of rows based on precomputed multiplicities	$O(n \log n)$	ODBJ [14]
Map	Applies an oblivious function to every row, reading and writing at predetermined positions within each row	$O(n)$	Standard technique
LinearPass	Applies an oblivious function to a fixed-size sliding window over a table	$O(n)$	Standard technique
ParallelPass	Applies an oblivious function that takes two rows and operates on fixed locations	$O(n)$	Standard technique

# Chapter 4

## Algorithm Overview

This chapter provides an intuitive overview of our algorithm before diving into formal specifications. We begin with ODBJ’s [14] 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’ [24] 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 [14] provides an elegant solution for oblivious binary joins, achieving  $O(N \log N + \text{OUT} \log \text{OUT})$  complexity where  $N$  is input size and  $\text{OUT}$  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 output-sized 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 passes, avoiding expensive primitives like ORAM [8].

### 4.1.2 The Multi-Way Multiplicity Challenge

To extend ODBJ [14] to multi-way joins, we must obtain the multiplicity of each tuple in the full join result before constructing it. For binary joins, ODBJ [14] 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

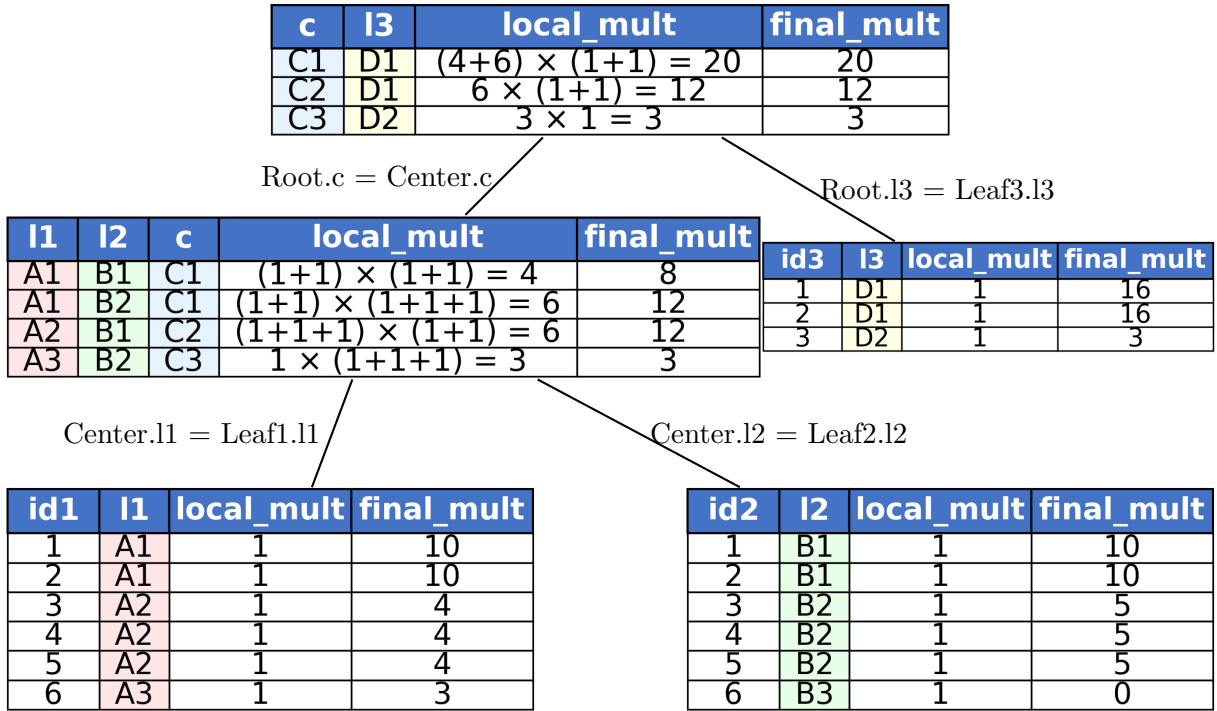


Figure 4.1: Example tables with data showing local and final multiplicities in a 5-table join tree. Each table shows its tuples and their computed multiplicities after the bottom-up and top-down phases.

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 [24] 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  $\{R_1, R_2, \dots, R_k\}$  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 [14], we distinguish between different types of tables based on their state in the algorithm:

- **input tables:** Original unmodified tables  $\{R_1, R_2, \dots, R_k\}$  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
- **output-sized tables:** augmented tables where each tuple appears exactly  $\alpha_{\text{final}}$  times
- **aligned tables:** output-sized tables reordered to enable correct concatenation for join result construction

Table 5.1: Table Schema Evolution Throughout Algorithm Phases

Type	Original Attributes	At-	Persistent Metadata	Meta-	Temporary Metadata
$R_{\text{input}}$	$\{a_1, a_2, \dots, a_m\}$		None		None
$R_{\text{aug}}$	$\{a_1, a_2, \dots, a_m\}$		orig_idx, $\alpha_{\text{final}}, F_{\text{sum}}$	$\alpha_{\text{local}},$	None
$R_{\text{comb}}$	$\{cca_1, a_2, \dots, a_m\}$		orig_idx, $\alpha_{\text{final}}, F_{\text{sum}}$	$\alpha_{\text{local}},$	$C_{\text{local}}, w_{\text{local}},$ $C_{\text{foreign}}$
$R_{\text{exp}}$	$\{a_1, a_2, \dots, a_m\}$		orig_idx, $\alpha_{\text{final}}, F_{\text{sum}}$	$\alpha_{\text{local}},$	copy_index
$R_{\text{align}}$	$\{a_1, a_2, \dots, a_m\}$		orig_idx, $\alpha_{\text{final}}, F_{\text{sum}}$	$\alpha_{\text{local}},$	alignment_key

#### Schema Evolution Notes:

- **Original Attributes:**  $\{a_1, a_2, \dots, a_m\}$  where  $m$  is the number of attributes in each table's schema
- **Persistent Metadata:** Carries forward through phases
  - orig\_idx: Original tuple index for tracking provenance
  - $\alpha_{\text{local}}$ : Local multiplicity within subtree
  - $\alpha_{\text{final}}$ : Final multiplicity in complete join
  - $F_{\text{sum}}$ : Foreign sum for alignment
- **Temporary Metadata:** Phase-specific fields

- Combined tables:  $C_{\text{local}}$  (local cumulative sum, bottom-up),  $w_{\text{local}}$  (local weight) and  $C_{\text{foreign}}$  (foreign cumulative sum, top-down)
  - Expanded tables: `copy_index` (index among copies of same tuple)
  - Aligned tables: `alignment_key` (computed positioning value)
- **Combined Table Structure:** Contains entries from multiple tables organized for dual-entry processing, each entry maintaining its original attributes and metadata

### 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.

### 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  $k = |\text{children}(v)|$ . 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.

Table 5.2: Algorithm Data Structures and Notation

<b>Notation</b>	<b>Description</b>
$T = (V, E)$	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
$C_{\text{foreign}}$	Foreign cumulative sum: intermediate values during dual-counter computation
$w_{\text{local}}$	Local weight counter ( $w_{\text{local}}$ )
$C_{\text{local}}$	Local cumulative sum (temporary): intermediate values during bottom-up computation
$\pi$	Entry type precedence mapping ( $\pi$ )
$e.\text{type}$	Entry type field
$e.d$	Entry data/tuple
$t.\text{orig\_idx}$	Original tuple index
$t.\text{join\_attr}$	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

**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  $p$ , if we were to join all tables in  $T \setminus T_c^-$  and sort the result by the join attribute between  $p$  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 the shuffle-then-reveal approach [12] where data is first obliviously shuffled before sorting to ensure fixed access patterns 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:      $window \leftarrow R[i : i + 1]$  ▷ 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

---

```
1: function MAP( $R$ ,  $TransformFunc$ )
2:    $R_{out} \leftarrow []$  ▷ Initialize output table
3:   for  $i = 1$  to  $|R|$  do
4:      $R_{out}[i] \leftarrow TRANSFORMFUNC(R[i], i)$ 
5:   end for
6:   return  $R_{out}$ 
7: 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

---

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

---

**Additional Primitives:** Our algorithm also relies on two additional oblivious primitives:

- **ObliviousExpand:** This primitive from the ODBJ framework [14] duplicates each tuple according to its multiplicity, creating an output-sized table where each original tuple appears the specified number of times.
- **HorizontalConcatenate:** This operation concatenates two tables horizontally, combining all columns from both tables while maintaining the same number of rows. Each row in the result contains the attributes from the corresponding rows in both input tables.

**Join Condition Encoding:** Any join condition between columns can be expressed as an interval constraint. Specifically, a condition between parent column  $p.\text{join\_attr}$  and child column  $c.\text{join\_attr}$  can be parsed as:  $c.\text{join\_attr} \in p.\text{join\_attr} + [x, y]$ , 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:  $p.\text{join\_attr} = c.\text{join\_attr}$  maps to  $c.\text{join\_attr} \in p.\text{join\_attr} + [0, 0]$
- Inequality:  $p.\text{join\_attr} > c.\text{join\_attr}$  maps to  $c.\text{join\_attr} \in p.\text{join\_attr} + (-\infty, 0)$
- Band constraint:  $p.\text{join\_attr} \geq c.\text{join\_attr} - 1$  maps to  $c.\text{join\_attr} \in p.\text{join\_attr} + [-1, \infty)$

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

The constraint function  $\mathcal{C}(p, c)$  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  $val$ , the boundary parameters create: (i) a START entry at  $val + d_1$  where if  $eq_1 = \text{EQ}$ , it includes values  $\geq val + d_1$ , and if  $eq_1 = \text{NEQ}$ , it includes values  $> val + d_1$ ; and (ii) an END entry at  $val + d_2$  where if  $eq_2 = \text{EQ}$ , it includes values  $\leq val + d_2$ , and if  $eq_2 = \text{NEQ}$ , it includes values  $< val + d_2$ . 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 output-sized tables by replicating each tuple according to its  $\alpha_{\text{final}}$  using oblivious distribution
5. **Phase 4 - Alignment and Concatenation (Section 5.6):** Reorder output-sized 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

---

```

1: function OBLIVIOUSMULTIWAYBANDJOIN( $T = (V, E)$ )
2:    $T_{\text{init}} \leftarrow \text{INITIALIZEALLTABLES}(T)$            ▷ Initialization: Add metadata columns
3:    $T_{\text{local}} \leftarrow \text{BOTTOMUPPHASE}(T_{\text{init}})$        ▷ Phase 1: Compute local multiplicities
4:    $T_{\text{final}} \leftarrow \text{TOPDOWNPHASE}(T_{\text{local}})$      ▷ Phase 2: Compute final multiplicities
5:    $T_{\text{expanded}} \leftarrow \text{DISTRIBUTEEXPAND}(T_{\text{final}})$    ▷ Phase 3: Distribute and expand
6:    $\text{Result} \leftarrow \text{CONSTRUCTJOINRESULT}(T_{\text{expanded}}, \text{root})$   ▷ Phase 4: Construct join
   result
7:   return  $\text{Result}$                                      ▷ Return final join result
8: 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.  $n_i = |R_i|$ ,  $N = \sum_{i=1}^k n_i$ .

---

```

1: function INITIALIZEALLTABLES( $T$ )
2:   for all nodes  $v \in V$  do
3:      $R_v \leftarrow \text{MAP}(R_v, \text{AddMetadataColumns})$ 
4:      $\text{LINEARPASS}(R_v, \text{WindowSetOriginalIndex})$ 
5:   end for
6:   return  $T$ 
7: end function

```

---



---

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

---

```

1: function ADDMETADATACOLUMNS( $t, \text{index}$ )
2:    $t.\text{orig\_idx} \leftarrow 0$ 
3:    $t.\alpha_{\text{local}} \leftarrow \text{null}$ 
4:    $t.\alpha_{\text{final}} \leftarrow \text{null}$ 
5:    $t.F_{\text{sum}} \leftarrow \text{null}$ 
6:   return  $t$ 
7: end function

```

---



---

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

---

```

1: function WINDOWSETORIGINALINDEX( $\text{window}$ )
2:    $\text{window}[1].\text{orig\_idx} \leftarrow \text{window}[0].\text{orig\_idx} + 1$ 
3: 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  $t \in R_{\text{leaf}}$  with  $t.\alpha_{\text{local}} = 1$ . For non-leaf nodes, the algorithm processes each parent-child pair  $(p, c)$  where  $p$  is the parent and  $c \in \text{children}(p)$ . 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,  $t.\alpha_{\text{local}} = |\{r \in \bowtie_{T_v^{\text{visited}}}: t \in r\}|$  where  $T_v^{\text{visited}}$  represents the subtree rooted at  $v$  restricted to nodes that have been visited in the post-order traversal.

For each parent-child pair  $(p, c)$ , the algorithm invokes COMPUTELOCALMULTIPLICITIES (Algorithm 10) with tables  $R_p$  (target) and  $R_c$  (source), along with constraint parameters  $\theta = \mathcal{C}(p, c)$  that encode the join condition. This updates each tuple  $t_p \in R_p$  by computing  $t_p.\alpha_{\text{local}}^{\text{new}} = t_p.\alpha_{\text{local}}^{\text{old}} \times \sum_{t_c \in R_c: (t_p, t_c) \text{ satisfy } \mathcal{C}(p, c)} t_c.\alpha_{\text{local}}$ , 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 COMBINE TABLE 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, COMBINE TABLE 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  $e_{\text{target}}$  that derives boundary entries  $e_s$  and  $e_t$ , the set of source entries  $\{e_{\text{source}}\}$  appearing between  $e_s$  and  $e_t$  in the sorted order is exactly the set of source entries that satisfy the join condition with  $e_{\text{target}}$ .

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 WINDOW-

COMPUTELocalInterval (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:    $order \leftarrow$  POSTORDERTRAVERSAL( $T, root$ )
3:   for all nodes  $v$  in  $order$  do
4:     if  $v$  is a leaf then
5:       for all tuple  $t \in R_v$  do
6:          $t.\alpha_{local} \leftarrow 1$ 
7:       end for
8:     else
9:       for all child nodes  $c$  of  $v$  do
10:         $R_v \leftarrow$  COMPUTELocalMULTIPLICITIES( $R_v, R_c, \mathcal{C}(v, c)$ )
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:    $order \leftarrow$  empty list
3:   for all child nodes  $c$  of  $root$  do
4:      $order \leftarrow order +$  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:    $R_{\text{comb}} \leftarrow \text{COMBINE\_TABLE}(R_{\text{target}}, R_{\text{source}}, \theta)$ 
3:    $R_{\text{comb}} \leftarrow \text{MAP}(R_{\text{comb}}, \lambda e : (e.\text{local\_sum} \leftarrow e.\alpha_{\text{local}}, e.\text{local\_interval} \leftarrow 0, e))$ 
4:    $\text{OBLIVIOUS\_SORT}(R_{\text{comb}}, \text{ComparatorJoinAttr})$ 
5:    $\text{LINEARPASS}(R_{\text{comb}}, \text{WindowComputeLocalSum})$ 
6:    $\text{OBLIVIOUS\_SORT}(R_{\text{comb}}, \text{ComparatorPairwise})$ 
7:    $\text{LINEARPASS}(R_{\text{comb}}, \text{WindowComputeLocalInterval})$ 
8:    $\text{OBLIVIOUS\_SORT}(R_{\text{comb}}, \text{ComparatorEndFirst})$ 
9:    $R_{\text{truncated}} \leftarrow R_{\text{comb}}[1 : |R_{\text{target}}|]$ 
10:   $\text{PARALLEL\_PASS}(R_{\text{truncated}}, R_{\text{target}}, \text{UpdateTargetMultiplicity})$ 
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

---

```

1: function COMBINE_TABLE( $R_{\text{target}}, R_{\text{source}}, \theta$ )
2:    $((d_1, eq_1), (d_2, eq_2)) \leftarrow \theta$ 
3:    $R'_{\text{source}} \leftarrow \text{MAP}(R_{\text{source}}, \text{function}(t):)$ 
4:      $e.\text{type} \leftarrow \text{SOURCE}$ 
5:      $e.\text{equality} \leftarrow \text{null}$ 
6:      $e.\text{join\_attr} \leftarrow t.\text{join\_attr}$ 
7:      $e.\text{orig\_idx} \leftarrow t.\text{orig\_idx}$ 
8:      $e.\alpha_{\text{local}} \leftarrow t.\alpha_{\text{local}}$ 
9:      $e.\alpha_{\text{final}} \leftarrow t.\alpha_{\text{final}}$ 
10:     $e.F_{\text{sum}} \leftarrow t.F_{\text{sum}}$ 
11:    return  $e$ 
12:   $R'_{\text{begin}} \leftarrow \text{MAP}(R_{\text{target}}, \text{function}(t):)$ 
13:     $e.\text{type} \leftarrow \text{START}$ 
14:     $e.\text{equality} \leftarrow eq_1$ 
15:     $e.\text{join\_attr} \leftarrow t.\text{join\_attr} + d_1$ 
16:     $e.\text{orig\_idx} \leftarrow t.\text{orig\_idx}$ 
17:     $e.\alpha_{\text{local}} \leftarrow t.\alpha_{\text{local}}$ 
18:     $e.\alpha_{\text{final}} \leftarrow t.\alpha_{\text{final}}$ 
19:     $e.F_{\text{sum}} \leftarrow t.F_{\text{sum}}$ 
20:    return  $e$ 
21:   $R'_{\text{end}} \leftarrow \text{MAP}(R_{\text{target}}, \text{function}(t):)$ 
22:     $e.\text{type} \leftarrow \text{END}$ 
23:     $e.\text{equality} \leftarrow eq_2$ 
24:     $e.\text{join\_attr} \leftarrow t.\text{join\_attr} + d_2$ 
25:     $e.\text{orig\_idx} \leftarrow t.\text{orig\_idx}$ 
26:     $e.\alpha_{\text{local}} \leftarrow t.\alpha_{\text{local}}$ 
27:     $e.\alpha_{\text{final}} \leftarrow t.\alpha_{\text{final}}$ 
28:     $e.F_{\text{sum}} \leftarrow t.F_{\text{sum}}$ 
29:    return  $e$ 
30:   $R_{\text{comb}} \leftarrow R'_{\text{source}} + R'_{\text{begin}} + R'_{\text{end}}$ 
31:  return  $R_{\text{comb}}$ 
32: end function

```

---

---

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

*Precedence:*  $(START, EQ) \rightarrow 1$ ,  $(END, NEQ) \rightarrow 1$ ,  $(SOURCE, null) \rightarrow 2$ ,  $(START, NEQ) \rightarrow 3$ ,  $(END, EQ) \rightarrow 3$

---

```
1: function COMPARATORJOINATTR( $e_1, e_2$ )
2:   if  $e_1$ .join_attr <  $e_2$ .join_attr then return -1
3:   else if  $e_1$ .join_attr >  $e_2$ .join_attr then return 1
4:   else
5:      $p_1 \leftarrow$  GETPRECEDENCE( $(e_1.type, e_1.equality)$ )
6:      $p_2 \leftarrow$  GETPRECEDENCE( $(e_2.type, e_2.equality)$ )
7:     if  $p_1 < p_2$  then return -1
8:     else if  $p_1 > p_2$  then return 1
9:     elsereturn 0
10:    end if
11:  end if
12: end function
```

---

---

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

---

```
1: function WINDOWCOMPUTELOCALSUM( $window$ )
2:   if  $window[1].type = SOURCE$  then
3:      $window[1].local\_sum \leftarrow window[0].local\_sum + window[1].\alpha_{local}$ 
4:   else  $\triangleright window[1].type \in \{START, END\}$ 
5:      $window[1].local\_sum \leftarrow window[0].local\_sum$ 
6:   end if
7: end function
```

---

---

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

---

```
1: function COMPARATORPAIRWISE( $e_1, e_2$ )    ▷ First: Target entries (START/END)
   before SOURCE entries
2:   if  $e_1.type \in \{\text{START}, \text{END}\}$  and  $e_2.type = \text{SOURCE}$  then return -1
3:   else if  $e_1.type = \text{SOURCE}$  and  $e_2.type \in \{\text{START}, \text{END}\}$  then return 1
4:   end if                                ▷ Second: Sort by original index
5:   if  $e_1.orig\_idx < e_2.orig\_idx$  then return -1
6:   else if  $e_1.orig\_idx > e_2.orig\_idx$  then return 1
7:   end if                                ▷ Third: START before END for same index
8:   if  $e_1.type = \text{START}$  and  $e_2.type = \text{END}$  then return -1
9:   else if  $e_1.type = \text{END}$  and  $e_2.type = \text{START}$  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  $window[0].type = \text{START}$  and  $window[1].type = \text{END}$  then
3:      $window[1].local\_interval \leftarrow window[1].local\_sum - window[0].local\_sum$ 
4:   end if
5: end function
```

---

---

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

---

```
1: function COMPARATORENDFIRST( $e_1, e_2$ )    ▷ First: END entries before all others
2:   if  $e_1.type = \text{END}$  and  $e_2.type \neq \text{END}$  then return -1
3:   else if  $e_1.type \neq \text{END}$  and  $e_2.type = \text{END}$  then return 1
4:   end if                                ▷ Second: Sort by original index
5:   if  $e_1.orig\_idx < e_2.orig\_idx$  then return -1
6:   else if  $e_1.orig\_idx > e_2.orig\_idx$  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

---

```

1: function UPDATETARGETMULTIPLICITY( $e_{combined}, e_{target}$ )
2:    $e_{target}.\alpha_{local} \leftarrow e_{target}.\alpha_{local} \times e_{combined}.local\_interval$ 
3: end function

```

---



---

**Algorithm 18** Get Entry Type Precedence: Map (entry\_type, equality\_type) tuple to precedence value

---

```

1: function GETPRECEDENCE( $(entry\_type, equality\_type)$ )
2:   if  $(entry\_type, equality\_type) = (START, EQ)$  then return 1
3:   else if  $(entry\_type, equality\_type) = (END, NEQ)$  then return 1
4:   else if  $(entry\_type, equality\_type) = (SOURCE, null)$  then return 2
5:   else if  $(entry\_type, equality\_type) = (START, NEQ)$  then return 3
6:   else if  $(entry\_type, equality\_type) = (END, EQ)$  then return 3
7:   end if
8: end function

```

---

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

The top-down phase propagates final multiplicities ( $\alpha_{final}$ ) from the root to all nodes in the tree, mirroring the reconstruction phase of Yannakakis [24]. 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  $\alpha_{final} = \alpha_{local}$  (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  $(p, c)$  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 ( $\alpha_{foreign}$ ) represents the number of join results from tables outside the child’s subtree that connect through the parent. This is computed as:  $t_c.\alpha_{final} = t_c.\alpha_{local} \times t_c.\alpha_{foreign}$ .

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 `COMBINE`TABLE 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

---

```
1: function TOPDOWNPHASE( $T, root$ )
2:   for all tuple  $t \in R_{root}$  do
3:      $t.\alpha_{final} \leftarrow t.\alpha_{local}$  ▷ Root final = local
4:   end for
5:   for all nodes  $v$  in pre-order traversal of  $T$  from  $root$  do
6:     for all child nodes  $c$  of  $v$  do
7:        $R_c \leftarrow \text{PROPAGATEFINALMULTIPLICITIES}(R_v, R_c, \mathcal{C}(v, c))$ 
8:     end for
9:   end for
10:  return  $T$  ▷ Return tree with tables containing computed final multiplicities
11: end function
```

---

---

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

---

```
1: function PROPAGATEFINALMULTIPLICITIES( $R_{source}, R_{target}, \theta$ )
2:    $R_{comb} \leftarrow \text{COMBINE\_TABLE}(R_{target}, R_{source}, \theta)$ 
3:    $R_{comb} \leftarrow \text{MAP}(R_{comb}, \lambda e :)$ 
4:      $(e.w_{local} \leftarrow e.\alpha_{local},$ 
5:      $e.C_{foreign} \leftarrow 0,$ 
6:      $e.foreign\_interval \leftarrow 0, e)$ 
7:    $\text{OBLIVIOUS\_SORT}(R_{comb}, \text{ComparatorJoinAttr})$ 
8:    $\text{LINEAR\_PASS}(R_{comb}, \text{WindowComputeForeignSum})$ 
9:    $\text{OBLIVIOUS\_SORT}(R_{comb}, \text{ComparatorPairwise})$ 
10:   $\text{LINEAR\_PASS}(R_{comb}, \text{WindowComputeForeignInterval})$ 
11:   $\text{OBLIVIOUS\_SORT}(R_{comb}, \text{ComparatorEndFirst})$ 
12:   $R_{truncated} \leftarrow R_{comb}[1 : |R_{target}|]$ 
13:   $\text{PARALLEL\_PASS}(R_{truncated}, R_{target}, \text{UpdateTargetFinalMultiplicity})$ 
14:  return  $R_{target}$ 
15: end function
```

---

---

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

---

```

1: function WINDOWCOMPUTEFOREIGNSUM(window)
2:   if window[1].type = START then
3:     window[1]. $w_{\text{local}}$   $\leftarrow$  window[0]. $w_{\text{local}}$  + window[1]. $\alpha_{\text{local}}$ 
4:     window[1]. $C_{\text{foreign}}$   $\leftarrow$  window[0]. $C_{\text{foreign}}$ 
5:   else if window[1].type = END then
6:     window[1]. $w_{\text{local}}$   $\leftarrow$  window[0]. $w_{\text{local}}$  - window[1]. $\alpha_{\text{local}}$ 
7:     window[1]. $C_{\text{foreign}}$   $\leftarrow$  window[0]. $C_{\text{foreign}}$ 
8:   else if window[1].type = SOURCE then
9:     window[1]. $w_{\text{local}}$   $\leftarrow$  window[0]. $w_{\text{local}}$ 
10:    window[1]. $C_{\text{foreign}}$   $\leftarrow$  window[0]. $C_{\text{foreign}}$  + window[1]. $\alpha_{\text{final}}$ /window[1]. $w_{\text{local}}$   $\triangleright$ 
      Real division
11:   end if
12: end function

```

---



---

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

---

```

1: function WINDOWCOMPUTEFOREIGNINTERVAL(window)
2:   if window[0].type = START and window[1].type = END then
3:     foreign_interval  $\leftarrow$  window[1]. $C_{\text{foreign}}$  - window[0]. $C_{\text{foreign}}$ 
4:     window[1].foreign_interval  $\leftarrow$  foreign_interval
5:     window[1]. $F_{\text{sum}}$   $\leftarrow$  window[0]. $C_{\text{foreign}}$   $\triangleright$  Record alignment position
6:   end if
7: end function

```

---



---

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

---

```

1: function UPDATETARGETFINALMULTIPLICITY(e, t)
2:   t. $\alpha_{\text{final}}$   $\leftarrow$  e.foreign_interval  $\times$  t. $\alpha_{\text{local}}$ 
3:   t. $F_{\text{sum}}$   $\leftarrow$  e. $F_{\text{sum}}$   $\triangleright$  For alignment
4: 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 [14], 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), 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  $F_{\text{sum}} + \lfloor \text{copy\_index} / \alpha_{\text{local}} \rfloor$ , where:

- $F_{\text{sum}}$  is the index of the first parent group that matches this child tuple
- $\text{copy\_index}$  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

---

```
1: function CONSTRUCTJOINRESULT( $T, root$ )
2:    $result \leftarrow$  OBLIVIOUSEXPAND( $R_{root}$ )  $\triangleright$  Expand root table
3:   for all nodes  $v$  in pre-order traversal of  $T$  from  $root$  do
4:     for all child nodes  $c$  of  $v$  do
5:        $R_c^{expanded} \leftarrow$  OBLIVIOUSEXPAND( $R_c$ )  $\triangleright$  Expand child table
6:        $result \leftarrow$  ALIGNANDCONCATENATE( $result, R_c^{expanded}$ )
7:     end for
8:   end for return  $result$ 
9: end function
```

---

---

**Algorithm 25** Align and Concatenate

---

```
1: function ALIGNANDCONCATENATE( $R_{accumulator}, R_{child}$ )
2:   OBLIVIOUSSORT( $R_{accumulator}, JoinThenOtherAttributes$ )  $\triangleright$  Sort by join attrs, then
   others
3:   LINEARPASS( $R_{child}, ComputeAlignmentKey$ )  $\triangleright$  Set alignment key for each tuple
4:   OBLIVIOUSSORT( $R_{child}, AlignmentKeyComparator$ )
5:   return HORIZONTALCONCATENATE( $R_{accumulator}, R_{child}$ )
6: end function
```

---

---

**Algorithm 26** Compute Alignment Key

---

```
1: function COMPUTEALIGNMENTKEY( $tuple$ )
2:    $tuple.alignment\_key \leftarrow tuple.F_{sum} + \lfloor tuple.copy\_index / tuple.\alpha_{local} \rfloor$ 
3:   Integer division
4: end function
```

---

---

**Algorithm 27** Join Then Other Attributes Comparator

---

```
1: function JOINTHENOTHERATTRIBUTES( $t_1, t_2$ )
2:   if  $t_1.join\_attr < t_2.join\_attr$  then return -1
3:   else if  $t_1.join\_attr > t_2.join\_attr$  then return 1
4:   else  $\triangleright$  Compare all other attributes return COMPAREOTHERATTR( $t_1, t_2$ )
5:   end if
6: 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:   else if  $t_1$ .join_attr <  $t_2$ .join_attr then return -1
5:   else if  $t_1$ .join_attr >  $t_2$ .join_attr then return 1
6:   else if  $t_1$ .copy_index <  $t_2$ .copy_index then return -1
7:   else if  $t_1$ .copy_index >  $t_2$ .copy_index then return 1
8:   elsereturn 0
9:   end if
10: end function
```

---

# Chapter 6

## Complexity Analysis

Our oblivious multi-way band join algorithm achieves a total complexity of  $O(N \log N + k \cdot \text{OUT} \log \text{OUT})$ , where  $k$  is the number of tables in the join,  $N$  is the total input size across all tables, and  $\text{OUT}$  is the output size. Note that with  $N$  as the total input size, each table has average size  $N/k$ . This chapter analyzes the complexity of each phase to derive this bound.

### 6.1 Phase 1: Bottom-Up Multiplicity Computation

During the bottom-up phase, for each parent-child pair in the join tree, we:

- Create a combined table with dual entries (for band joins) or regular entries (for equality joins) from two tables of average size  $N/k$  each
- Sort the combined table by join attribute:  $O((2N/k) \log(2N/k)) = O((N/k) \log(N/k))$
- Perform linear passes to compute multiplicities:  $O(N/k)$

Since the join tree has  $k - 1$  edges, and each edge is processed once, the total complexity for Phase 1 is:

$$(k - 1) \cdot O((N/k) \log(N/k)) = O(N \log(N/k)) = O(N \log N)$$

## 6.2 Phase 2: Top-Down Multiplicity Propagation

The top-down phase propagates final multiplicities from the root to all tables. For each parent-child pair:

- Sort two tables of average size  $N/k$  by join attribute:  $O((N/k) \log(N/k))$
- Linear pass to propagate multiplicities:  $O(N/k)$

Similar to Phase 1, with  $k - 1$  edges processed, the total complexity is:

$$(k - 1) \cdot O((N/k) \log(N/k)) = O(N \log N)$$

## 6.3 Phase 3: Distribution and Expansion

Each of the  $k$  tables is expanded based on its final multiplicities:

- Oblivious distribution to position tuples:  $O((N/k) \log(N/k))$  per table
- Oblivious expansion to create copies: Each table expands to OUT total size, so  $O(\text{OUT} \log \text{OUT})$  per table

Processing all  $k$  tables:

- Distribution total:  $k \cdot O((N/k) \log(N/k)) = O(N \log N)$
- Expansion total:  $k \cdot O(\text{OUT} \log \text{OUT}) = O(k \cdot \text{OUT} \log \text{OUT})$

## 6.4 Phase 4: Alignment and Concatenation

This phase builds the final result by iteratively aligning and concatenating tables:

- Start with the root table as the accumulator
- For each of the  $k - 1$  remaining tables:

- Sort the table by alignment key:  $O(\text{OUT} \log \text{OUT})$
- Sort the accumulator by corresponding alignment key:  $O(\text{OUT} \log \text{OUT})$
- Concatenate the aligned tables:  $O(\text{OUT})$

Since we perform alignment for  $k - 1$  tables, and each alignment requires sorting the growing accumulator (which has size  $O(\text{OUT})$ ), the total complexity is  $O(k \cdot \text{OUT} \log \text{OUT})$ .

## 6.5 Total Complexity

Combining all phases:

$$\begin{aligned}
 \text{Total} &= O(N \log N) + O(N \log N) \\
 &\quad + O(N \log N + k \cdot \text{OUT} \log \text{OUT}) \\
 &\quad + O(k \cdot \text{OUT} \log \text{OUT}) \\
 &= O(N \log N + k \cdot \text{OUT} \log \text{OUT}) \tag{6.1}
 \end{aligned}$$

The factor of  $k$  appears only in the output-dependent terms because:

- Phases 1-2: Process  $k - 1$  edges with data of size  $O(N/k)$  each, yielding  $O(N \log N)$  total
- Phase 3 (expansion) and Phase 4 (alignment): Each of  $k$  tables requires  $O(\text{OUT} \log \text{OUT})$  operations

For typical queries with a small constant number of tables, this remains efficient.

## 6.6 Counting Sorting Operations

To better understand the practical performance of our algorithm, we analyze the number of sorting operations required. We categorize sorts into two types: sorts on input-sized tables (before expansion) and sorts on output-sized tables (after expansion). This distinction is important because input size and output size can be very different—either can be much larger than the other depending on the query’s selectivity—making the relative cost of these operations highly query-dependent. In the following analysis,  $k$  denotes the number of tables in the multi-way join.

The following table compares the number of sorting operations between the current algorithm and an improved version that eliminates redundant sorts:

Version	Input-Sized Sorts (size $\leq 3N/k$ )	Output-Sized Sorts (size = OUT)	Total
Current	$7k - 6$	$2k - 2$	$9k - 8$
Improved	$5k - 4$	$2k - 2$	$7k - 6$

Table 6.1: Comparison of sorting operations between current and improved algorithms. The improvement reduces input-sized sorts by  $2k - 2$  operations by eliminating the end-first comparator sort.

### 6.6.1 Sorting Operations in Current Algorithm

#### Sorts on Input-Sized Tables

*Tables with average size  $\leq 3N/k$  (combined tables during multiplicity computation)*

- **Phase 1 - Bottom-up Multiplicity Computation**

**Operations:** Process each of the  $k - 1$  parent-child edges

**Sorts per edge:** 3 sorts on the combined table

- Sort by join attribute
- Sort by pairwise comparator
- Sort by end-first comparator

**Subtotal:**  $3(k - 1)$  sorts

- **Phase 2 - Top-down Multiplicity Propagation**

**Operations:** Process each of the  $k - 1$  parent-child edges

**Sorts per edge:** 3 sorts on the combined table (same pattern as Phase 1)

**Subtotal:**  $3(k - 1)$  sorts

- **Phase 3 - Distribution**

**Operations:** Prepare each of the  $k$  tables for expansion

**Sorts per table:** 1 sort for oblivious distribution

**Subtotal:**  $k$  sorts

<b>Total Input-Sized Sorts:</b> $3(k - 1) + 3(k - 1) + k = 7k - 6$
--------------------------------------------------------------------

## Sorts on Output-Sized Tables

*Tables with size equal to OUT (after expansion)*

- **Phase 4 - Alignment and Concatenation**

**Operations:** Align  $k - 1$  tables with the accumulator

**Sorts per table:** 2 sorts

- Sort the table by alignment key
- Sort the accumulator by corresponding alignment key

**Subtotal:**  $2(k - 1)$  sorts

<b>Total Output-Sized Sorts:</b> $2(k - 1) = 2k - 2$
------------------------------------------------------

### 6.6.2 Improved Sorting Count

We can reduce the number of sorting operations with a slight modification to our algorithm that preserves sort order where possible and combines certain operations.

**Key Optimization:** After sorting by the pairwise comparator in both Phase 1 and Phase 2, the combined table exhibits a predictable alternating pattern of START and END entries. Instead of performing an additional sort by the end-first comparator to move END entries to the top, we can exploit this pattern directly:

- The pairwise sort creates an alternating START-END pattern for entries with the same join attribute
- To extract the END entries (which represent target table tuples), we simply select entries at even indices (0-based) from the first  $2 \times |\text{target table}|$  positions
- This direct indexing eliminates the need for the third sort in each parent-child pair processing

**Detailed Breakdown:**

*Sorts on Input-Sized Tables:*

- **Phase 1 (Bottom-up):** 2 sorts per edge (reduced from 3) → Total:  $2(k - 1)$  sorts
- **Phase 2 (Top-down):** 2 sorts per edge (reduced from 3) → Total:  $2(k - 1)$  sorts
- **Phase 3 (Distribution):** 1 sort per table (unchanged) → Total:  $k$  sorts

<b>Improved Input-Sized Sorts:</b> $2(k - 1) + 2(k - 1) + k = 5k - 4$
-----------------------------------------------------------------------

*Sorts on Output-Sized Tables:*

- **Phase 4 (Alignment):** 2 sorts per table (unchanged) → Total:  $2(k - 1)$  sorts

<b>Output-Sized Sorts (unchanged):</b> $2(k - 1) = 2k - 2$
------------------------------------------------------------

# Chapter 7

## 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.

### 7.1 Security Model and Definitions

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

#### 7.1.1 Oblivious Operations

**Definition 7.1** (Oblivious Operation). *An operation  $\mathcal{O} : \mathcal{D} \rightarrow \mathcal{D}'$  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.

## 7.1.2 Composition Theorem

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

**Theorem 7.2** (Sequential Composition). *If  $\mathcal{O}_1 : \mathcal{D} \rightarrow \mathcal{D}'$  and  $\mathcal{O}_2 : \mathcal{D}' \rightarrow \mathcal{D}''$  are oblivious operations, then their sequential composition  $(\mathcal{O}_2 \circ \mathcal{O}_1) : \mathcal{D} \rightarrow \mathcal{D}''$  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. □

## 7.1.3 Security Goal

Our security goal is to prove the following theorem:

**Theorem 7.3** (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.

## 7.2 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:  $\text{result} = p \cdot \text{value}_{\text{true}} + (1 - p) \cdot \text{value}_{\text{false}}$ .
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.

### 7.2.1 Window Functions

**Lemma 7.4.** `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.  $\square$

**Lemma 7.5.** `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).  $\square$

**Lemma 7.6.** 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  $\text{window}[0]$  fields and  $\text{window}[1]$  fields, always write to  $\text{window}[1].w_{\text{local}}$  and  $\text{window}[1].C_{\text{foreign}}$ .
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.  $\square$

**Lemma 7.7.** 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].Fsum`.
2. **Arithmetic conversion:** The conditional becomes:

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

□

## 7.2.2 Comparators

**Lemma 7.8.** `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 7.9.** COMPARETPAIRWISE (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. □

**Lemma 7.10.** ALIGNMENTKEYCOMPARATOR (Algorithm 28) 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' alignment\_key, join\_attr, and copy\_index fields in the same order.

2. **Arithmetic conversion:** Convert the three-level comparison to arithmetic:

$$\begin{aligned}
\text{cmp}_1 &= \text{sign}(t_1.\text{alignment\_key} - t_2.\text{alignment\_key}) \\
\text{eq}_1 &= (t_1.\text{alignment\_key} == t_2.\text{alignment\_key}) \in \{0, 1\} \\
\text{cmp}_2 &= \text{sign}(t_1.\text{join\_attr} - t_2.\text{join\_attr}) \\
\text{eq}_2 &= (t_1.\text{join\_attr} == t_2.\text{join\_attr}) \in \{0, 1\} \\
\text{cmp}_3 &= \text{sign}(t_1.\text{copy\_index} - t_2.\text{copy\_index}) \\
\text{result} &= \text{cmp}_1 \cdot (1 - \text{eq}_1) \\
&\quad + \text{cmp}_2 \cdot \text{eq}_1 \cdot (1 - \text{eq}_2) \\
&\quad + \text{cmp}_3 \cdot \text{eq}_1 \cdot \text{eq}_2
\end{aligned}$$

This ensures deterministic ordering while maintaining obliviousness through tie-breaking on join attribute and copy index.  $\square$

### 7.2.3 Update Functions

**Lemma 7.11.** 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.

$\square$

**Lemma 7.12.** UPDATETARGETFINALMULTIPLICITY (Algorithm 23) is inherently oblivious.

*Proof.* The function performs pure arithmetic:

1. **Access pattern:** Always read  $e.\text{foreign\_interval}$ ,  $e.F_{\text{sum}}$ , 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.

$\square$

## 7.3 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.

### 7.3.1 Oblivious Primitives

We rely on the following well-established oblivious primitives:

**Assumption 7.13.** *The following operations are oblivious:*

- OBLIVIOUSSORT: *Uses the shuffle-then-reveal paradigm [12] for oblivious sorting*
- OBLIVIOUSEXPAND: *From ODBJ [14], 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*

### 7.3.2 Composed Operations

**Lemma 7.14.** *For any comparator  $C$  that can be converted to oblivious form,  $\text{OBLIVIOUSSORT}(T, C_{\text{oblivious}})$  is oblivious.*

*Proof.* By Assumption 7.13, OBLIVIOUSSORT has a fixed comparison pattern based only on table size. By Lemmas 7.8-7.10, our comparators can be converted to oblivious implementations. Using the converted oblivious versions  $C_{\text{oblivious}}$  and applying Theorem 7.2, the composition is oblivious.  $\square$

**Lemma 7.15.** *For any window function  $W$  that can be converted to oblivious form,  $\text{LINEARPASS}(T, W_{\text{oblivious}})$  is oblivious.*

*Proof.* LINEARPASS has a deterministic iteration pattern based only on table size (with fixed window size 2). By Lemmas 7.4-7.7, our window functions can be converted to oblivious implementations. Using the converted versions  $W_{\text{oblivious}}$  and applying Theorem 7.2, the composition is oblivious.  $\square$

**Lemma 7.16.** *For any update function  $U$  that is inherently oblivious or can be converted to oblivious form,  $\text{PARALLELPASS}(T, U_{\text{oblivious}})$  is oblivious.*

*Proof.*  $\text{PARALLELPASS}$  applies  $U$  to each element independently with a fixed access pattern. By Lemmas 7.11-7.12, our update functions are inherently oblivious (pure arithmetic). The parallel application maintains obliviousness.  $\square$

## 7.4 Level 3: Phase Security

We prove that each phase of our algorithm is oblivious.

### 7.4.1 Initialization Phase

**Lemma 7.17.** *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 7.2, their composition is oblivious.  $\square$

### 7.4.2 Bottom-Up Phase

**Lemma 7.18.** *The Bottom-Up phase is oblivious.*

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

rithm 10):

BottomUp = CombineTable (Algorithm 11)  
→ ObliviousSort(ComparatorJoinAttr)  
→ LinearPass(WindowComputeLocalSum)  
→ ObliviousSort(ComparatorPairwise)  
→ LinearPass(WindowComputeLocalInterval)  
→ ObliviousSort(ComparatorEndFirst)  
→ ParallelPass(UpdateTargetMultiplicity)

Each operation is oblivious by Lemmas 7.14-7.16. The number of iterations depends only on the public tree structure. By repeated application of Theorem 7.2, the entire phase is oblivious.  $\square$

### 7.4.3 Top-Down Phase

**Lemma 7.19.** *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 7.2, the phase is oblivious.  $\square$

### 7.4.4 Distribution and Expansion Phase

**Lemma 7.20.** *The Distribution and Expansion phase is oblivious.*

*Proof.* This phase applies OBLIVIOUSEXPAND to each table. By Assumption 7.13, OBLIVIOUSEXPAND is oblivious. The operation is applied to each table independently based on the public tree structure.  $\square$

### 7.4.5 Alignment and Concatenation Phase

**Lemma 7.21.** *The Alignment and Concatenation phase is oblivious.*

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

1. OBLIVIOUSSORT on parent table with JOINTHENOTHERATTRIBUTES comparator (oblivious by Lemma 7.14)
2. LINEARPASS to compute alignment keys (oblivious by Lemma 7.15)
3. OBLIVIOUSSORT on child table with ALIGNMENTKEYCOMPARATOR (oblivious by Lemmas 7.10 and 7.14)
4. HORIZONTALCONCATENATE (oblivious concatenation)

Each operation is oblivious, and their composition is oblivious by Theorem 7.2. □

## 7.5 Level 4: Complete Algorithm Security

We now prove our main security theorem.

*Proof of Theorem 7.3.* The complete algorithm performs:

Algorithm = Initialization  
                   → Bottom-Up Phase  
                   → Top-Down Phase  
                   → Distribution & Expansion  
                   → Alignment & Concatenation

By Lemmas 7.17, 7.18, 7.19, 7.20, and 7.21, each phase is oblivious.

By repeated application of Theorem 7.2 (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. □

# Chapter 8

## Evaluation

We evaluate our oblivious multi-way band join algorithm by comparing its performance against OJOIN [5], the state-of-the-art oblivious multi-way join algorithm. OJOIN employs Sep INLJ (Separate Index Nested-Loop Join), an oblivious approach that stores encrypted data and index blocks in separate Path-ORAM structures for each table, providing both encryption and access pattern hiding through ORAM protocols. 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.

We implement our oblivious band join algorithm in C++ using Intel SGX for secure execution. The complete implementation, including all experiments described in this chapter, is publicly available at <https://github.com/rd-wei/Oblivious-Multi-Way-Band-Joins> for reproducibility.

### 8.1 Implementation

We implemented our algorithm in C++ using a hybrid architecture that separates untrusted control flow from trusted SGX2 computation.

#### 8.1.1 Data Preprocessing

To simplify the implementation and focus on 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 1970-01-01, while strings are mapped to integer identifiers.
- **Table duplication:** When a query requires using the same table multiple times (self-joins), we create distinct copies with appropriate renaming.
- **Data encryption:** Tables are stored encrypted using AES-CTR mode, where each row is encrypted with a unique 64-bit sequential nonce, enabling efficient batch access.

### 8.1.2 Batch Processing

The system processes operations in batches of up to 2000 operations (`MAX_BATCH_SIZE`). An operation is one of three types of oblivious functions executed within the enclave: **comparators** that compare two entries for sorting (e.g., join attribute comparison, alignment key comparison), **window functions** that process a sliding window of entries during linear passes (e.g., computing multiplicities, intervals, cumulative sums), or **update functions** that modify single entries (e.g., initializing metadata, setting indices, computing derived values). We use this batching approach primarily for window functions and update functions during linear passes. During execution:

- The untrusted application collects operations into batches
- When a batch reaches 2000 operations or needs flushing, it sends the encrypted entries to the enclave
- The enclave decrypts the batch, performs the oblivious operations, and re-encrypts
- Results are written back to untrusted storage

This approach minimizes SGX2 overhead by amortizing the cost of enclave transitions across many operations. For sorting operations, which are used extensively throughout the algorithm and require different access patterns, we employ a specialized implementation described in the next section.

### 8.1.3 Oblivious Sorting Implementation

Our oblivious sorting implementation follows the shuffle-then-reveal paradigm [12], combining Waksman network shuffling [20] with merge sort to achieve both security and efficiency. The approach uses two phases:

**Phase 1: Waksman Network Shuffle.** We first obviously shuffle the array using a Waksman permutation network to randomize element positions. After shuffling, the comparison outcomes during sorting do not reveal information about the original data distribution, as the elements are randomly permuted.

**Phase 2: Merge Sort.** We then apply standard merge sort using oblivious comparators. While the memory access patterns during merging are non-oblivious (following the standard merge sort pattern), this is secure because the shuffle has already randomized the data—the adversary cannot correlate access patterns with actual data values.

For efficiency, both the Waksman shuffle and merge sort employ  $k$ -way recursion. Given an array of size  $n$  to sort, we pad it to size  $2^a \times k^b$  where  $2^a \leq \text{BATCH\_SIZE}$  (ensuring base cases fit in the enclave) and  $2^a \times k^b \geq n$  (covering the entire array).

The  **$k$ -way Waksman shuffle** operates recursively by first routing the input array to  $k$  subarrays using random routing, then recursively shuffling each of these  $k$  subarrays independently, and finally routing the shuffled subarrays back to the output array with random permutation. For base cases where arrays are smaller than `BATCH_SIZE`, we switch to a conventional 2-way Waksman network that fits entirely within the SGX2 enclave, ensuring efficient termination of the recursion.

The  **$k$ -way merge sort** follows a bottom-up approach, starting by building sorted subarrays of size `BATCH_SIZE` within the SGX2 enclave, then recursively performing  $k$ -way merges to combine these sorted subarrays into progressively larger sorted arrays until the entire array is sorted. This  $k$ -way recursion significantly reduces the depth of the merge tree compared to binary merge sort, minimizing the number of passes over the data and the associated SGX2 transition overhead.

This  $k$ -way recursive approach minimizes data transfer between untrusted memory and the SGX2 enclave. The total amount of data transferred is  $O(n \times \log_k(n/\text{BATCH\_SIZE}))$ , significantly reducing the overhead compared to binary recursion. We use  $k = 8$  for our implementation to balance the depth of recursion with the complexity of  $k$ -way operations.

## 8.2 Experimental Setup

### 8.2.1 Dataset and Queries

We use the TPC-H benchmark [17] at scale factors 0.001, 0.01, and 0.1, matching the setup used in the OJOIN [5] evaluation, generating approximately 1MB, 10MB, and 100MB of

raw data respectively 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 [5] paper are:

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

- **TM1:** 3-way join between `customer`, `orders`, and `lineitem`
- **TM2:** 4-way join between `supplier`, `customer`, and two `nation` tables with region constraints
- **TM3:** 5-way join between `nation`, `supplier`, `customer`, `orders`, and `lineitem`

**Band Joins (TB series):**

- **TB1:** Band join between two `supplier` tables comparing account balances
- **TB2:** Band join between two `part` tables comparing retail prices

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 queries tested are:

**TM1 Query:**

```
SELECT * FROM customer, orders, lineitem
WHERE customer.C_CUSTKEY = orders.O_CUSTKEY
      AND lineitem.L_ORDERKEY = orders.O_ORDERKEY
```

**TM2 Query:**

```
SELECT * FROM supplier, customer, nation1, nation2
WHERE supplier.S_NATIONKEY = nation1.N1_N_NATIONKEY
      AND customer.C_NATIONKEY = nation2.N2_N_NATIONKEY
      AND nation1.N1_N_REGIONKEY = nation2.N2_N_REGIONKEY
```

**TM3 Query:**

```
SELECT * FROM nation, supplier, customer, orders, lineitem
WHERE nation.N_NATIONKEY = supplier.S_NATIONKEY
      AND nation.N_NATIONKEY = customer.C_NATIONKEY
      AND customer.C_CUSTKEY = orders.O_CUSTKEY
      AND orders.O_ORDERKEY = lineitem.L_ORDERKEY
```

#### **TB1 Query:**

```
SELECT * FROM supplier1, supplier2
WHERE supplier1.S1_S_ACCTBAL < supplier2.S2_S_ACCTBAL
```

#### **TB2 Query:**

```
SELECT * FROM part1, part2
WHERE part1.P1_P_RETAILPRICE < part2.P2_P_RETAILPRICE
```

Following the evaluation methodology established by OJOIN [5], we test both multi-way equality joins (TM1, TM2, TM3) and binary band joins (TB1, TB2). Our algorithm advances the state of oblivious query processing by demonstrating that both query types can be efficiently handled through a unified approach based on the Yannakakis algorithm enhanced with our dual-entry technique. This represents a step forward from existing solutions that require separate algorithmic frameworks for different join types, while achieving performance improvements through our efficient SGX2-based architecture.

## **8.2.2 Hardware Configuration**

Experiments were conducted on a server with the following specifications:

- Intel Xeon E-2374G processor @ 3.70GHz with SGX2 support
- 4 cores, 8 threads with AVX-512 support (using 1 core only for experiments)
- 125GB RAM (120GB available)
- Ubuntu 22.04.4 LTS with Linux kernel 5.15.0
- NVMe SSD storage

### 8.2.3 Metrics

We measure the total execution time for each query, comparing our algorithm’s runtime against OJOIN [5]. For our implementation, execution time is measured end-to-end from program initialization to termination, encompassing the complete query processing pipeline: initializing the SGX2 enclave, reading encrypted input tables from disk, parsing the SQL query specification, executing the oblivious join algorithm with SGX2-protected operations, and writing the encrypted result set to persistent storage. This comprehensive measurement captures all system overheads including enclave initialization, I/O operations, encryption/decryption costs, and enclave transitions, providing a realistic assessment of the algorithm’s practical performance.

The OJOIN [5] results presented use their Sep INLJ configuration, which achieves obliviousness through Path-ORAM storage with separate ORAM structures for each table’s encrypted data blocks and B-tree index blocks. As OJOIN [5] did not provide a public implementation, we compare against their reported results using identical queries and dataset configurations.

## 8.3 Results: Band Joins

Table 8.1 presents the results for band join queries with inequality constraints.

Table 8.1: Performance comparison for band joins at different scale factors

Query	Scale Factor	Output Size	OJOIN (s) <sup>1</sup>	Ours (s)
TB1	0.001	45	–	0.40
TB1	0.01	4,950	100	1.77
TB1	0.1	499,499	10,000	84.98
TB2	0.001	19,900	–	3.73
TB2	0.01	1,998,097	–	358.82
TB2	0.1	199,915,491	10,000,000	–

Our dual-entry technique demonstrates superior performance for band joins. For TB1 at SF=0.01, we complete in 1.77 seconds compared to OJOIN’s [5] roughly 100 seconds—an approximately  $57\times$  speedup. At SF=0.1, TB1 generates 499,499 output rows, which we

<sup>1</sup>OJOIN results are approximate values read from log-scale graphs in [5]

process in 84.98 seconds compared to OJOIN’s [5] roughly 10,000 seconds—an approximately 118× speedup. For TB2, we complete SF=0.001 in 3.73 seconds and SF=0.01 in 358.82 seconds (about 6 minutes) for nearly 2 million output rows.

TB2 at SF=0.1 produces a staggering 199,915,491 output rows, requiring OJOIN [5] roughly 10,000,000 seconds—approximately 115 days. With each row consuming approximately 100 bytes, the output alone would require nearly 20GB of memory. Since our implementation stores multiple copies of tables during processing (for multiplicities, alignment, and result construction), the total memory requirement exceeds our system’s 120GB available RAM. We therefore could not execute this query due to memory constraints.

## 8.4 Results: Multi-way Equality Joins

Table 8.2 shows the performance comparison between our algorithm and OJOIN [5] for multi-way equality joins.

Table 8.2: Performance comparison for multi-way equality joins

Query	Scale Factor	Output Size	OJOIN (s) <sup>1</sup>	Ours (s)
TM1	0.001	6,005	–	4.83
TM1	0.01	60,175	–	37.32
TM1	0.1	600,572	100,000	302.88
TM2	0.001	292	–	0.56
TM2	0.01	29,929	10,000	11.41
TM2	0.1	2,999,594	100,000	1262.00
TM3	0.001	2,485	–	4.51
TM3	0.01	236,250	–	134.81
TM3	0.1	24,029,033	10,000,000	–

Our algorithm demonstrates significant performance advantages over OJOIN’s [5] Sep INLJ across all multi-way equality join queries. Where both systems completed experiments, we achieve dramatic speedups. For TM1 at SF=0.1 (600,572 output rows), we complete in 302.88 seconds versus OJOIN’s [5] roughly 100,000 seconds—an approximately 330× speedup. For TM2, we achieve an approximately 876× speedup at SF=0.01 (11.41 seconds versus roughly 10,000 seconds for 29,929 rows) and an approximately 79× speedup at SF=0.1 (1262.00 seconds versus roughly 100,000 seconds for nearly 3 million rows).

For smaller scale factors where OJOIN [5] did not report results, our algorithm maintains excellent performance: TM1 completes in 4.83 seconds at SF=0.001 and 37.32 seconds at SF=0.01; TM2 completes in 0.56 seconds at SF=0.001; and TM3 completes in 4.51 seconds at SF=0.001 and 134.81 seconds at SF=0.01. The largest experiment, TM3 at SF=0.1, generates 24,029,033 output rows, requiring OJOIN [5] roughly 10,000,000 seconds—approximately 115 days. With approximately 100 bytes per row, the output requires about 2.4GB. However, since our implementation maintains multiple copies of tables throughout the join process, the total memory requirement for this 5-way join exceeds available system memory, preventing us from executing this query.

## 8.5 Discussion

### 8.5.1 Key Findings

Our evaluation reveals significant performance advantages over the state-of-the-art OJOIN [5]. Our algorithm achieves approximately  $57\times$  to  $876\times$  speedup across all tested queries: TM2 at SF=0.01 completes in 11.41 seconds versus OJOIN’s [5] roughly 10,000 seconds (approximately  $876\times$  speedup), TM1 at SF=0.1 completes in 302.88 seconds versus OJOIN’s [5] roughly 100,000 seconds (approximately  $330\times$  speedup), TB1 at SF=0.01 completes in 1.77 seconds versus OJOIN’s [5] roughly 100 seconds (approximately  $57\times$  speedup), and TB1 at SF=0.1 completes in 84.98 seconds versus OJOIN’s [5] roughly 10,000 seconds (approximately  $118\times$  speedup). This dramatic improvement stems from fundamental architectural differences—while OJOIN’s [5] Sep INLJ incurs substantial overhead from Path-ORAM operations for every data access, our native oblivious algorithm design avoids these overheads entirely.

Our dual-entry technique successfully handles band joins with inequality constraints while maintaining complete obliviousness. The algorithm processes both equality joins (TM1, TM2, TM3) and band joins (TB1, TB2) efficiently at different scale factors, validating our approach of adapting the Yannakakis algorithm with novel techniques for inequality handling.

The selective SGX2 execution architecture proves highly effective. By running only critical oblivious operations (window functions and comparators) inside SGX2 with data stored encrypted using AES-CTR externally, we achieve better performance than Path-ORAM-based approaches. The batch processing of 2000 operations at a time amortizes enclave transition costs while maintaining security guarantees. This design enables pro-

cessing datasets much larger than enclave memory capacity while avoiding the logarithmic overhead that Path-ORAM imposes on every data access.

At scale factor 0.1, we successfully completed experiments with outputs ranging from 500K to 3 million rows. TM1 (600,572 rows) completed in 302.88 seconds (about 5 minutes), TB1 (499,499 rows) completed in 84.98 seconds, and TM2 (nearly 3 million rows) completed in 1262.00 seconds (about 21 minutes), demonstrating our ability to handle moderate-to-large outputs efficiently with approximately 79-330 $\times$  speedup over OJOIN [5]. However, queries producing tens to hundreds of millions of rows face memory constraints: TM3 produces 24 million rows (requiring multiple gigabytes with our multi-copy approach), and TB2 generates nearly 200 million rows (requiring 20GB just for output, far exceeding available memory when considering multiple table copies). While OJOIN [5] would require approximately 115 days for these queries, our system cannot execute them due to insufficient memory for maintaining the multiple table copies required during processing.

## 8.5.2 Future Improvements

Performance optimization presents the most immediate opportunity for improvement. Further optimizing the SGX2 boundary crossings and batch processing could reduce overhead when executing critical functions in the enclave. This includes exploring larger batch sizes, better entry deduplication strategies, and minimizing data marshaling costs.

Streaming preprocessing could eliminate the current separate preprocessing phase. By integrating type conversion and table preparation into the main algorithm, we could reduce the number of passes over the data and avoid storing intermediate preprocessed tables.

Extending support to cyclic queries would broaden the algorithm’s applicability. While our current implementation handles acyclic join trees, many real-world queries involve cycles. Incorporating generalized hypertree decomposition techniques would enable processing of these more complex query structures while maintaining obliviousness.

Finally, investigating parallel processing opportunities could significantly improve performance. While maintaining obliviousness adds constraints, there are opportunities for parallelizing independent operations across multiple cores, particularly during sorting phases and when processing independent subtrees of the join tree.

# Chapter 9

## Conclusion

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

### 9.1 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 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 effective for processing band joins, though large outputs require extensive computation time.

**3. Rigorous Security Analysis:** We provided a comprehensive four-level security proof, formally 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 theoretically that all memory access patterns are independent of input data values.

## 9.2 Experimental Validation

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

- For smaller datasets, we successfully process band joins (TB1, TB2) with consistent performance
- For equality join TM2 at scale factor 0.01, we achieved 11.41 seconds compared to OJOIN's [5] roughly 10,000 seconds—an approximately 876× improvement
- However, queries with very large outputs (millions to billions of rows) require extensive computation time for both algorithms

These results highlight both the promise of our approach and the fundamental computational challenges of processing large-scale oblivious joins.

## 9.3 Practical Implications

This work contributes to oblivious database research in several ways:

**Practical Secure Analytics:** By enabling efficient processing of band join queries with moderate output sizes, we make secure processing of temporal and range queries feasible for many real-world 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 demonstrates the effectiveness of selective enclave execution, where only critical oblivious operations run in secure hardware while data resides outside.

## 9.4 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.

**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.

# References

- [1] Rakesh Agrawal, Dmitri Asonov, Murat Kantarcioglu, and Yaping Li. Sovereign joins. In *Proceedings of the 22nd International Conference on Data Engineering, ICDE '06*, page 26, USA, 2006. IEEE Computer Society.
- [2] Arvind Arasu and Raghav Kaushik. Oblivious query processing. In *Proc. 17th International Conference on Database Theory (ICDT)*, pages 26–37, 2014.
- [3] Johes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel Kho, and Jenie Rogers. Smcql: secure querying for federated databases. *Proc. VLDB Endow.*, 10(6):673–684, February 2017.
- [4] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: Sgx cache attacks are practical. In *Proceedings of the 11th USENIX Conference on Offensive Technologies, WOOT'17*, page 11, USA, 2017. USENIX Association.
- [5] Zhao Chang, Dong Xie, Sheng Wang, and Feifei Li. Towards practical oblivious join. In *Proceedings of the 2022 International Conference on Management of Data, SIGMOD '22*, page 803–817, New York, NY, USA, 2022. Association for Computing Machinery.
- [6] Victor Costan and Srinivas Devadas. Intel SGX explained. Cryptology ePrint Archive, Paper 2016/086, 2016.
- [7] Saba Eskandarian and Matei Zaharia. Oblidb: oblivious query processing for secure databases. *Proc. VLDB Endow.*, 13(2):169–183, October 2019.
- [8] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, May 1996.

- [9] Michael T. Goodrich. Zig-zag sort: a simple deterministic data-oblivious sorting algorithm running in  $o(n \log n)$  time. In *Proceedings of the Forty-Sixth Annual ACM Symposium on Theory of Computing*, STOC '14, page 684–693, New York, NY, USA, 2014. Association for Computing Machinery.
- [10] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions and tractable queries. *J. Comput. Syst. Sci.*, 64(3):579–627, May 2002.
- [11] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on intel sgx. In *Proceedings of the 10th European Workshop on Systems Security*, EuroSec'17, New York, NY, USA, 2017. Association for Computing Machinery.
- [12] Koki Hamada, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. Oblivious radix sort: An efficient sorting algorithm for practical secure multi-party computation. Cryptology ePrint Archive, Paper 2014/121, 2014.
- [13] Xiao Hu and Zhiang Wu. Optimal oblivious algorithms for multi-way joins, 2025.
- [14] Simeon Krastnikov, Florian Kerschbaum, and Douglas Stebila. Efficient oblivious database joins. *Proc. VLDB Endow.*, 13(12):2132–2145, July 2020.
- [15] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside sgx enclaves with branch shadowing. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, page 557–574, USA, 2017. USENIX Association.
- [16] Yaping Li and Minghua Chen. Privacy preserving joins. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering (ICDE'08)*, Proceedings - International Conference on Data Engineering, pages 1352–1354, United States, April 2008. IEEE. 2008 IEEE 24th International Conference on Data Engineering, ICDE'08 ; Conference date: 07-04-2008 Through 12-04-2008.
- [17] Transaction Processing Performance Council. TPC-H benchmark specification. <http://www.tpc.org/tpch/>, 2021. Version 3.0.
- [18] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: stealthy page table-based attacks on enclaved execution. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, page 1041–1056, USA, 2017. USENIX Association.

- [19] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. Conclave: secure multi-party computation on big data. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.
- [20] Abraham Waksman. A permutation network. *J. ACM*, 15(1):159–163, January 1968.
- [21] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS '17*, page 2421–2434, New York, NY, USA, 2017. Association for Computing Machinery.
- [22] 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*, page 1969–1981, New York, NY, USA, 2021. Association for Computing Machinery.
- [23] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy, SP '15*, page 640–656, USA, 2015. IEEE Computer Society.
- [24] Mihalis Yannakakis. Algorithms for acyclic database schemes. In *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7, VLDB '81*, page 82–94. VLDB Endowment, 1981.
- [25] 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 *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation, NSDI'17*, page 283–298, USA, 2017. USENIX Association.