Parallel Oblivious Joins using Radix Partitioning

by

Nafis Ahmed

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

© Nafis Ahmed 2025

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the 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.

Statement of Contributions

This thesis is based on a manuscript submitted for publication. Nafis Ahmed was the lead researcher for all of the algorithmic and experimental work presented in this thesis, and was the lead implementer of the prototype. Nafis Ahmed was the primary contributing author to all sections of this thesis, with secondary contributions from Dr. Sujaya Maiyya.

Abstract

We present parallel *doubly oblivious* algorithms for both non-foreign key and foreign key joins using an *oblivious radix partitioning* technique. Oblivious query processing enables secure execution over encrypted data when organizations outsource data to the cloud. When the cloud server processes encrypted data within hardware enclaves, the data is vulnerable to side-channel leaks caused by data-dependent memory access patterns and control flow. Our algorithms efficiently defend against these vulnerabilities by combining data partitioning with parallel execution. Specifically, we propose a doubly oblivious radix partitioning approach that divides input arrays into disjoint partitions without leaking the multiplicity of individual elements, unlike vanilla radix partitioning. This is especially important for join operations, where duplicate keys are common. To construct our join algorithm, we apply oblivious radix partitioning independently to each input table, allowing the algorithm *to compare tuples only within corresponding partitions*. When input tables are presorted, our oblivious join algorithm is the first to avoid combining and obliviously resorting them, yielding performance improvements over the state-of-the-art scheme, Obliviator. Beyond joins, our oblivious radix partitioning technique is a standalone primitive with applications to a broad class of problems, including oblivious aggregation and private set intersection.

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Dr. Sujaya Maiyya. She has been so much more than an inspirational teacher and mentor; she has truly nurtured the researcher in me, giving me the freedom to unearth research questions and pursue unexplored directions. She has always indulged my numerous queries and ideas while gently nudging me in the right direction. I am eternally grateful for her unwavering support in my academic endeavors and beyond.

I am also grateful to my thesis committee members, Dr. Florian Kerschbaum and Dr. Xiao Hu. Their insightful questions and thoughtful feedback challenged me to think more broadly, refine my ideas, and confront the blind spots I did not even realize I had.

I owe a special thank you to Dr. Catie Chang, who took a chance on a completely clueless college freshman and introduced me to the world of research. That leap of faith set me on a path that has profoundly shaped my academic journey and personal growth.

I would also like to acknowledge you, the reader, for taking the time to read this thesis. It is through your attention and curiosity that this work will live on and grow beyond the year-long effort that produced it.

I am thankful to my family—Ammu, Abbu, Ma, Baba, Joy Bhaiya, and Mochi—for always being there for me. I would like to give special credit to my mother, who has relentlessly prioritized her children's education, even when the odds were stacked against her. Her sacrifices and dedication have shaped the opportunities I have today, and I carry her example of perseverance with me in all that I do.

Finally, I am forever indebted to my best friend, my partner, my wife, Nowreen. This thesis is as much hers as it is mine, and it is not even the slightest stretch to say so. She believed in me pursuing graduate school long before the thought even occurred to me. She has been the constant source of my strength and the light guiding me through the darkest of times. Her unconditional lovw has shaped not only who I am today, but also drives me to become a better person every day. Words cannot bear the weight of her impact on my life, but every page of this thesis carries an imprint of her care, her patience, and all the immeasurable ways she enriches our relationship. Through it, I hope to honor her monumental role in my life.

Dedication

To the life of my being, Nowreen

Table of Contents

Author's Declaration ii

Statement of Contributions iii

Abstract iv

Acknowledgements v

Dedication vi

List of Figures ix

List of Tables x

1 Introduction 1

2 Preliminaries 4

2.1 TEEs 4

2.2 System and Threat Model 4

2.3 Security Definition 5

2.4 Oblivious Primitives 6

2.5 Radix Partitioning 7

3 Overview 9

4 Algorithm Description **13**
4.1 De-duplicating Input Tables . . . 13
4.1.1 Counting Duplicates . . . 14
4.1.2 Replacing Duplicate Keys with Dummies . . . 14
4.2 Radix Partitioning . . . 15
4.3 Exchanging Repetition Counts . . . 19
4.4 Re-populate Input Tables . . . 20
4.5 Oblivious Distribution and Expansion . . . 21
4.6 Table Alignment and Merging . . . 24
4.7 Foreign Key Joins . . . 24

5 Parallelization Techniques **26**

6 Security Arguments **29**
6.1 Security Arguments . . . 29

7 Experimental Evaluation **35**
7.1 Non-Foreign Key Joins . . . 37
7.2 Foreign Key Joins . . . 40

8 Discussion **41**

9 Related Work **43**

10 Conclusion **45**

References **46**

List of Figures

3.1 Overview of the join algorithm, where k_i denotes a join key with payload p_x and d_j denotes a dummy key. 10

4.1 Result of distribution on R and S. 21

7.1 Comparison with five real-world datasets 38

7.2 TPC-H non-foreign key joins (log scale) 39

7.3 TPC-H foreign key joins 40

List of Tables

5.1 Parallel approach of calculating repetition counts (as in Algorithm 2). The sorted input is evenly divided among T=4 threads with each thread computing local counts. 27

5.2 Forwarding repetition counts across slice boundaries when keys span multiple slices. If the last key in slice i equals the first key in slice i+1, its local repetition count from slice i is forwarded and cleared from slice i. 27

7.1 Various dataset sizes used in the experiments 36

Chapter 1

Introduction

Organizations are increasingly relying on cloud computing to outsource their data storage and computation needs due to the many benefits of the cloud. While encryption remains a foundational layer of defense to keep sensitive data secure in outsourced settings, it is insufficient to guarantee privacy in the face of leakage-abuse attacks that exploit observable memory access patterns to infer sensitive information [43, 35, 44, 45, 15, 25, 34, 38, 47, 65, 68, 88, 66]. This concern is especially acute in domains with strict privacy requirements, such as finance, healthcare, and national security.

Oblivious algorithms protect against such threats by hiding an application's memory access patterns from a semi-honest adversary who can monitor the storage layer. Many schemes such as Oblivious RAM [79], frequency smoothing [33], and dynamic key shuffling [54] propose mechanisms to obliviously access outsourced data. Works such as [25, 51, 20, 19, 1] extend the above oblivious schemes to execute relational operators, including joins, obliviously. However, these schemes require a client-side trusted proxy to perform query processing, introducing substantial computational demands on the client and high communication overhead due to data transfer from the cloud.

Trusted execution environments (TEEs) such as Intel SGX/TDX [42, 80] and AMD SEV-SNP [77] overcome these challenges by allowing encrypted data processing on a cloud server. TEEs isolate an application's environment within a system and execute application code securely within the protected environment. However, TEEs alone fail to fully safeguard application data because they expose data- or query-dependent memory footprint and program traces. Many attacks [14, 39, 48, 49, 60, 76, 82, 86] have leveraged such side-channel leakages to uncover enclave secrets.

Seminal works by Arasu and Kaushik [5], Opaque [90], and ObliDB [28] partially de-

fended against such leakages for relational operators. However, these works assume the existence of a limited yet *specialized oblivious* memory whose accesses are hidden from the adversary. No TEE provider to-date provides such a memory, rendering these solutions risky in practice. *Doubly* or *fully* oblivious [59, 74] schemes eliminate this reliance by designing algorithms that intrinsically make data-independent memory accesses and program flow decisions. Our goal is to design efficient doubly oblivious algorithms for both non-foreign key and foreign key joins.

We observe that existing oblivious join algorithms using enclaves [46, 57, 90, 28, 5] are *sort-merge joins*, with [46, 57] being doubly oblivious. This research began with a question of: *can we design a doubly oblivious hash join?* At a high level, plaintext hash joins of tables R and S first compute the hash of the join keys of the smaller table R[1], and store each key's corresponding payload in a hash table. In the second step, table S probes the hash table by hashing its keys to find matching entries. Because this scheme scans the input tables just once, its complexity is $O(n_1+n_2)$, where n_1 and n_2 are input table lengths.

Making this algorithm oblivious, however, raises non-trivial challenges because (i) repeated join keys in R create hash collisions and can reveal the number of unique keys in R, and/or (ii) repeated probing of the same hash table entry leaks the distribution of repeated keys in S – both leaking data dependent information to an adversary. Mitigating this leakage requires de-duplicating repeated keys from both tables, which in turn requires an oblivious sort [11], rendering this scheme as a sort-merge join. Therefore, while an oblivious hash join *may* be constructed when both R and S have unique keys, this fails to be secure for generalized joins.

Given that our join algorithm has to be sort-merge based, we ask: *can the algorithm leverage hash-based techniques in the "merge" step to improve efficiency?* Before answering this question, let's take a step back and examine the challenges with plaintext sort-merge joins. Given tables R and S sorted on the join key, the join scheme scans both tables tuple by tuple, and when a match is found, it writes the corresponding payloads to an output table. The merge step achieves two goals: (i) identify the join keys common in both R and S, and (ii) write the common join key in the output table $c_{i_R} \times c_{i_S}$ times where c_{i_R} and c_{i_S} are counts of key k_i in R and S. We call these c_i's the *expansion counts* of a join key. Because this approach reveals the cardinality of each key based on the pointer traversal patterns during the merge phase, it becomes unsuitable for oblivious settings.

State-of-the-art oblivious join solutions [46, 57] mitigate this leakage by first concatenating the input tables and then obliviously sorting the combined table on the $<$`join key`,

[1]Without loss of generality, we always assume R to be the smaller table throughout the paper and use terms 'join key' and 'key' interchangeably.

`table id`> pair. Because the concatenated and sorted table places common keys in both tables next to each other, this allows their algorithms to identify the join keys present in both tables as well as compute and exchange the expansion counts. Each matching key k_i is then expanded in R by the count of its occurrences in S (i.e., c_{i_S}), and vice versa, before securely merging the expanded tables.

We identify two main disadvantages with this approach of exchanging expansion counts: (i) it incurs an expensive oblivious sort over the combined input of size n_1+n_2, and (ii) more importantly, it fails to exploit presorted inputs — a common case in analytical pipelines — thus missing a key optimization leveraged by plaintext sort-merge joins [64]. Our goal is to address these two drawbacks by integrating techniques from hash joins.

***Contributions**:* In this work, we design parallel, doubly oblivious relational join operators that runs within TEEs. Unlike prior approaches, our join algorithm is the first to exploit presorted input tables to avoid materializing and obliviously sorting the combined input table. Even when inputs are unsorted, our method obliviously sorts each table independently and in parallel, avoiding the costly oblivious sort on the combined table required by existing solutions [46, 57].

Given sorted tables R and S, the algorithm first deduplicates repeated keys and applies *radix partitioning* [56] but with obliviousness guarantees (Chapter 4). Partitioning enables the algorithm to apply the hash-and-probe technique per partition to exchange expansion counts *in parallel*. After exchanging expansion counts, the algorithm expands the input tables such that the expanded table only consists of overlapping join keys, each repeated exactly $c_{i_R} \times c_{i_S}$ times.

To our knowledge, this is the first use of hashing techniques in a doubly oblivious non-foreign key and foreign key join algorithms, i.e., the inputs may have duplicate keys. The proposed oblivious radix partitioning primitive is of independent interest and applicable to other secure data processing tasks (Chapter 8). For foreign key joins on presorted tables, *our algorithm runs in linearithmic complexity*, unlike any existing oblivious schemes.

Our evaluation (Chapter 7) compares our join algorithm against the state-of-the-art scheme, Obliviator [57], using five real-world datasets and TPC-H queries [81]. Results demonstrate the advantage of reducing the sizes of oblivious sorts, with the difference in execution times being more pronounced for large datasets (>100M tuples). Since our approach avoids the oblivious sort on the combined input tables, it enables efficient offline/online oblivious join designs that were not feasible in prior work. We plan to explore these possibilities in future work.

Chapter 2

Preliminaries

2.1 TEEs

Trusted Execution Environments (TEEs) provide secure runtimes that protect code and data from adversaries on the same system. They use hardware-based encryption engines to secure data in memory with encryption keys derived from a processor-embedded root key. Applications using TEEs rely on remote attestation to verify the integrity of software running within the TEE. Compared to protected-memory TEE solutions (e.g., Intel SGX), Intel TDX [80] and AMD SEV-SNP [77] isolate an entire virtual machine (VM). With this, applications incur minimal overhead in porting existing codebases into TEEs. Therefore, many cloud providers are offering VM-based confidential computing solutions [58, 32, 3]. Both Intel TDX and AMD SEV-SNP, however, rely on an untrusted host for scheduling and paging, which exposes side-channel risks. The goal of this work is to address these side-channel leakages in TEEs by designing fully or doubly oblivious (referred to as "oblivious" throughout this thesis unless stated otherwise) join algorithms that protect data privacy, even against an adversary capable of observing all memory access patterns and control flow.

2.2 System and Threat Model

We consider a system model wherein an organization outsources its storage and compute needs to the cloud and utilizes server-hosted TEEs to securely compute on its encrypted data.

Unlike the trusted client-side proxy model, once the client encrypts and outsources the data, it does not participate in executing the oblivious algorithm. Any computation is fully outsourced to the enclave physically residing on the server. This distinction means that the enclave may be exposed to strong adversaries who control the server's software stack and operating system, and hence can not only observe accesses in and out of the enclave but also monitor memory accesses and program control flow both inside and outside the enclave at cache line granularity [63, 77]. The attacker may leverage these observable information to launch side-channel attacks, including cache attacks, branch prediction and paging-based attacks, and memory bus snooping [14, 39, 60, 76, 49, 82, 86, 48]. These avenues of side channels are absent in client-side proxy models because the proxy machine is trusted; thus, such systems are considered *singly* oblivious. In our threat model, the adversary cannot breach the secure processor or access its secret key. The data owner can verify whether the code running in the enclave is correct using remote attestation [42, 80, 77]. In line with prior works [24, 57, 17, 28, 59, 89, 72], we consider attacks based on power consumption, rollback, and denial-of-service out of scope. This threat model is inline with commonly adopted models in enclave-based approaches [24, 46, 59, 17, 57].

2.3 Security Definition

Informally, an algorithm is considered oblivious if for any two inputs and outputs of the same length, the information observable to an adversary remains indistinguishable. Within the context of TEEs, this implies that the logical data accesses (i.e., rows of a table being accessed), the physical memory accesses (i.e., memory locations being touched), and the program control flow of the algorithm remains independent of the the data or the query. A formal definition of oblivious algorithms typically relies on a simulator that can simulate the algorithm's behavior, including any leakages allowed by the algorithm, given only public parameters. In our work, the lengths of the two input tables, n_1 and n_2, is considered public information known to the adversary. Our algorithm reveals the size of the join output m and the number of unique keys m' in the join, both modeled as allowed leakage, $\mathcal{L}$.

Formally, consider a game $\mathcal{G}$ that either executes the actual algorithm Alg or the ideal simulated version of the algorithm Sim, which only takes the public parameters and the leakage profile $\mathcal{L}$ as inputs. The algorithm Alg is considered oblivious if, for any probabilistic polynomial time adversary $\mathcal{A}$, $\mathcal{A}$ cannot distinguish if the game executed Alg or Sim, except with negligible probability, based on observing the resulting events of the game.

Definition 1. *An oblivious algorithm Alg is oblivious if there exists a polynomial time*

simulator Sim such that

$$|Pr[\mathcal{G}_{Alg} \Rightarrow \mathit{True}] - Pr[\mathcal{G}_{Sim,\mathcal{L}} \Rightarrow \mathit{True}]| \leq negl(\lambda)$$

where λ is the security parameter.

Although the proposed join algorithm reveals the size of the output m, this leakage can be mitigated by standard techniques such as worst-case padding to the size $n_1 \times n_2$ or setting the output size to $m' = m + \Delta$ where Δ can be a positive noise derived from differential privacy [27] as proposed in [13] or a heuristically derived noise as proposed in [25].

2.4 Oblivious Primitives

The proposed oblivious join algorithm relies on a number of oblivious building blocks, which we explain next.

Oblivious Compare-and-Set: Because double obliviousness requires the program trace of the enclave code to be independent of the data or query, our join algorithm cannot have statements such as *if* `secret:` *... else ....* Therefore, similar to prior work [57, 17, 62], we use bitwise manipulation operations and SIMD intrinsics to replace such conditional statements. For example, to set a variable *key* to either a real or a dummy key based on some secret-dependent boolean `flag`, we execute:

```
key = (flag & real) | (¬flag & dummy)
```

For brevity, we use `OCmpSet(f,x,y)` to indicate the function that obliviously sets x to y if $f = true$.

Oblivious Sorting: Oblivious sorting produces a sorted permutation of its input without revealing the relative order of elements. Among various oblivious sorting algorithms, bitonic sort [11] is widely adopted due to its in-place design, input-independent control flow, and inherent parallelism. It runs in $O(n \log^2 n)$ time for input size n, and in $O((n \log^2 n)/p)$ with p parallel threads [62]. Although asymptotically faster $O(n \log n)$ constructions exist [2, 31, 36], they either suffer from large constant factors or typically under perform in practice [62, 57]. As a result, many oblivious systems and join schemes [90, 28, 89, 24, 57, 46] rely on bitonic sort as the preferred oblivious sorting primitive. A recent line of work performs oblivious sorting by first applying an oblivious shuffle followed

by a non-oblivious comparison-based sort, achieving an overall complexity of $O(n \log n)$ [40, 6, 37]; however, the available implementation is not parallelized [37]. In our system, we adopt a parallelized bitonic sort [62] to remain consistent with the baselines as sorting dominates the runtime of the join operator [46, 57].

Oblivious Compaction: Oblivious compaction takes an input array containing a mix of "marked" and "unmarked" elements and rearranges it so that all marked elements appear contiguously at the top with their original relative order preserved. The remaining positions are then filled with the unmarked elements whose original ordering may not be maintained. In this work, we use the tight order-preserving compaction algorithm proposed by Sasy et al. [74] which achieves $O(n \log n)$ runtime complexity. Although linear-time oblivious compaction algorithms exist [30, 26], they suffer from large constant factors and lack parallelism. Reversing compaction allows the set of marked elements to be placed into an array at predetermined positions according to an injective index mapping. Rather than pulling marked items toward the top, the inverse operation deterministically pushes them outward to their designated locations. This perspective forms the basis for the oblivious expansion primitive used in Section 4.5.

Oblivious Shuffling: An oblivious shuffle transforms an input array into a uniformly random permutation such that an external observer cannot learn anything about the data values, the original order, the final order, or the mapping between them. In principle, one could obtain a random permutation by assigning each element a random key and performing an oblivious sort. However, prior work has shown that dedicated oblivious shuffling algorithms are asymptotically more efficient and offer better practical performance than using a full oblivious sort [74, 62, 37]. Some constructions even support offline precomputation of the permutation to reduce online cost [75], though we do not assume such a phase in our current setting. For our system, we integrate the sequential shuffle implementation of Gu et al. [37], which achieves $O(n \log n)$ complexity. This matches the asymptotic complexity of Ngai et al. [62], but their implementation is not open-source. While Sasy et al. [74] offer a parallelized construction, their approach incurs a higher $O(n \log^2 n)$ complexity.

2.5 Radix Partitioning

Radix partitioning divides a relation into multiple smaller partitions based on the radix (bitwise representation) of a chosen partitioning key. For each key (typically the hash of the key) in the input, it examines a fixed number of bits, r, called the radix bits, based on which, it places each key and its payload into one of 2^r partitions. Applying radix

partitioning on both inputs R and S places matching keys in the same partition. This is particularly helpful for hash joins because rather than storing the entire R in a hash table and probing all of S on this large table, now one can apply the hash-and-probe technique per partition in parallel.

Typically, radix partitioning is recursively applied in multiple passes, with each pass diving the existing partitions further using a different subset of the radix bits (e.g., first $r/2$ bits in the first pass, followed by the remaining radix bits in the second pass in a two-pass radix partitioning scheme). This multi-pass approach keeps the size of each partition small and can minimize cache and Translation Lookaside Buffer (TLB) thrashing by ensuring that the working set size of each pass remains within hardware limits [56].

Chapter 3

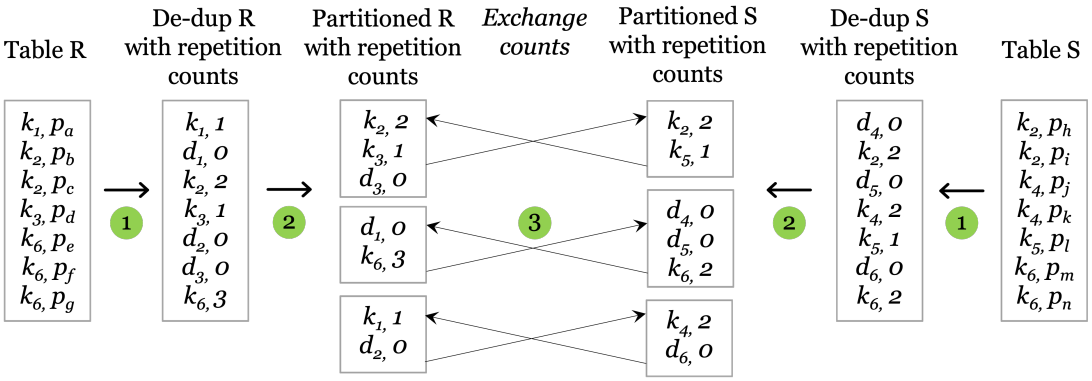
Overview

We begin with a high-level overview of the algorithm in this chapter, with the following chapter delving into each component in detail.

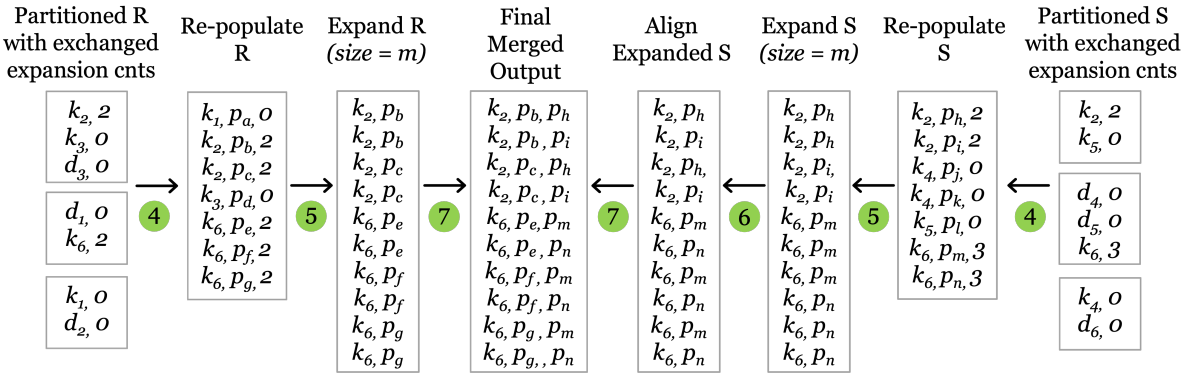
As shown in Figure 3.1, the algorithm takes as input the two tables, R and S, to be joined on keys indicated with k_i. If the tables are unsorted, the algorithm first sorts one on the join attribute. Figure 3.1 assumes presorted tables, and the thesis adopts the same assumption unless stated otherwise. Each table may contain repeated join keys, which pose challenges in oblivious settings (chapter 1). Therefore, in step 1, the algorithm identifies repetition counts of each key and de-duplicates the two tables by replacing repeated keys with unique dummy keys. The two tables can be de-duplicated in parallel.

In 2, the algorithm applies radix partitioning on each table to split the input into disjoint sets. Although not depicted in the figure, the algorithm first obliviously shuffles both R and S and randomizes the de-duplicated keys using a hash function (Chapter 4 explains why). In step 3, each partition of S probes the corresponding partition of R to identify if the key being probed exists in R and if so, exchange its repetition counts. *Why?* Because if a key k_i appears c_{i_R} times in R and c_{i_S} times in S, it will appear $c_{i_R} \times c_{i_S}$ in the output. We call the swapped repetition counts *expansion counts* because each k_i in R must be expanded (i.e., repeated) c_{i_S} times and vice versa. Dummy keys always have an expansion count of 0 because they should occur in the output 0 times.

Figure 3.1b starts with the two partitioned tables with the expansion counts obtained from the other table. For example, the counts of k_6 are swapped for the two tables. In step 4, the algorithm combines the partitions and removes the dummies by re-populating each table with its original keys and corresponding expansion counts. Since R and S were obliviously shuffled in step 2, we follow the approach outlined in [40, 6, 37, 85] to restore



(a) Steps to exchange expansion counts of the two tables



(b) Steps to expand and merge the two tables

Figure 3.1: Overview of the join algorithm, where k_i denotes a join key with payload p_x and d_j denotes a dummy key.

their original order. The re-populated tables contain the exchanged expansion count c for each key, using which step 5 expands the key and its payload c times on to a table of size m. While the previous two steps can occur in parallel for the two tables, step 6 only affects table S by aligning it to group unique payloads of a join key together as shown in the figure. In the final step 7, the algorithm simply row-wise merges both expanded tables to produce the final output.

We summarize the overall algorithm in Algorithm 1. The core components of our proposed algorithm are explained sequentially in Chapter 4.

Algorithm 1 Overall Oblivious Join Algorithm

Input: Tables R, S, $radix_bits$
Output: Oblivious join result T_{join}
Step 0: If either R or S is unsorted, obliviously sort each table independently
1: $R \leftarrow \texttt{BitonicSort}(R)$, $S \leftarrow \texttt{BitonicSort}(S)$
Step 1.1: Count duplicate keys in sorted input tables
2: $\texttt{countR} \leftarrow$ Count-Consecutive-Duplicates(R)
3: $\texttt{countS} \leftarrow$ Count-Consecutive-Duplicates(S)
Step 1.2: Generate unique dummy keys to replace duplicates
4: $\texttt{dedupR} \leftarrow$ Replace-With-Dummies$(R, \texttt{countR})$
5: $\texttt{dedupS} \leftarrow$ Replace-With-Dummies$(S, \texttt{countS})$
Step 2: Radix partition the shuffled, de-duplicated tables
6: $r \leftarrow radix_bits$
7: $\texttt{partitionedR}, \texttt{mrkrR} \leftarrow$ Radix-Partition(OShuffle$(\texttt{dedupR}), r)$
8: $\texttt{partitionedS}, \texttt{mrkrS} \leftarrow$ Radix-Partition(OShuffle$(\texttt{dedupS}), r)$
Step 3: Exchange expansion counts per partition based on mrkrR and mrkrS. Let R' and S' be re-constructed tables.
9: $R', S' \leftarrow$ Exchange-Counts$(\texttt{partR}, \texttt{partS})$
Step 4: Re-populate input tables with a comparison-based sort
10: $R' \leftarrow$ Sort(R')
11: $S' \leftarrow$ Sort(S')
12: Replace dummy keys in R' and S' with original keys and associated expansion counts
Step 5: Obliviously distribute and expand tuples from R' and S'
13: $\texttt{expandedR} \leftarrow$ Distribute-And-Expand(R')
14: $\texttt{expandedS} \leftarrow$ Distribute-And-Expand(S')
Step 6: Align the expanded table $expandedS$ with $expandedR$
15: $\texttt{expandedS} \leftarrow$ Align(expandedS)
Step 7: Merge $expandedR$ and $expandedS$ to produce T_{join}
16: $T_{join} \leftarrow$ Merge(expandedR, expandedS)
17: **return** T_{join}

Chapter 4

Algorithm Description

This chapter describes the doubly oblivious join algorithm using radix partitioning as shown in Algorithm 1. For ease of exposition, this chapter will describe each component in a sequential way, with Chapter 5 describing how to parallelize the sequential steps.

Preprocessing step: If the two tables to be joined, R and S, are unsorted, the algorithm first obliviously sorts each table independently based on the join key. In this work, we assume the input tables to be presorted unless stated otherwise. To help describe each algorithmic step, we will use the following two sorted tables as a running example (same as in Figure 3.1):

$$R = [(k_1, p_a), (k_2, p_b), (k_2, p_c), (k_3, p_d), (k_6, p_e), (k_6, p_f), (k_6, p_g)]$$

$$S = [(k_2, p_h), (k_2, p_i), (k_4, p_j), (k_4, p_k), (k_5, p_l), (k_6, p_m), (k_6, p_n)]$$

4.1 De-duplicating Input Tables

Recall from Chapter 1, the main challenge with applying vanilla hash-and-probe techniques in oblivious settings is the existence of duplicates in input tables. Repeated join keys in R will create hash collisions, whereas repeated join keys in S will probe the same hash bucket multiple times – both revealing data dependent frequency distribution to an adversary. Therefore, the algorithm first de-duplicates repeated join keys in R and S, as depicted in step 1 of Figure 3.1a.

4.1.1 Counting Duplicates

To de-duplicate the input tables, the algorithm first computes the duplication counts per key. We define arrays `countR` and `countS` to store the number of preceding duplicate keys in R and S respectively. For the running example, we obtain:

$$\texttt{countR} = [1, 0, 2, 1, 0, 0, 3], \quad \texttt{countS} = [0, 2, 0, 2, 1, 0, 2].$$

- A value `0` at index i indicates that the key has duplicates and that i is not the last index in its duplicate run.
- A non-zero value at index i represents the total number of duplicates (i.e., repetitions) for the key ending at index i.

Algorithm 2 describes an oblivious approach to calculate repetition counts of each key for an input table T. With a linear scan (Chapter 5 discusses a parallel approach), the algorithm uses variable j to propagate a running count over repeated keys by comparing the current key with the previous key (line 8). Importantly, the algorithm resets j only when a key changes (line 11). To ensure data-independent branching and memory accesses, the algorithm replaces any conditional logic with `OCmpSet`. Algorithm 2 returns the count array (i.e., `countR`/`countS`) for a given table.

4.1.2 Replacing Duplicate Keys with Dummies

After computing the duplicate counts, the join algorithm next de-duplicates the input by replacing repeated join keys with unique dummy keys to hide the join key frequency distributions. For the running example, the algorithm generates `dedupR` and `dedupS`, where each tuple includes its index in the sorted table, the join key, and its repetition count (obtained from `countR/S`):

$$\begin{aligned}\texttt{dedupR} = [&(0, k_1, 1),\ (1, d_1, 0),\ (2, k_2, 2),\\ &(3, k_3, 1),\ (4, d_2, 0),\ (5, d_3, 0),\ (6, k_6, 3)]\\ \texttt{dedupS} = [&(0, d_4, 0),\ (1, k_2, 2),\ (2, d_5, 0),\\ &(3, k_4, 2),\ (4, k_5, 1),\ (5, d_6, 0),\ (6, k_6, 2)]\end{aligned}$$

Algorithm 3 describes how to obliviously replace duplicates and construct `dedupR` and `dedupS`. For each tuple in $T \in \{R, S\}$, it obtains the repetition count from `countT` based

on the index. It then generates a new dummy key using a hash function (line 7) for each key. If the count value at an index i is not `0`, indicating the end of a duplicate run, the algorithm replaces the dummy key with the actual key (line 8); otherwise, it retains the dummy key. Both Algorithms 2 and 3 can be applied in parallel to R and S.

4.2 Radix Partitioning

With join keys deduplicated and their repetition counts computed, the join algorithm next identifies keys overlapping in both tables and swaps their repetition counts. A naive but oblivious way to swap the counts is to probe the entire table R for every element in S and apply oblivious compare and swap for each compared tuples. However, this incurs $O(n_1 \times n_2)$ complexity, which is significantly worse than the complexity of state-of-the-art oblivious joins $O(nlog^2n+mlog^2m)$ where $n = n_1+n_2$ [46, 57]. Our algorithm applies radix partitioning [56] to reduces the search space that each tuple in S needs to probe to identify overlapping keys and swap their repetition counts (shown in step 2 of Figure 3.1a).

This section explains the single-pass radix partitioning approach (Section 2.5) with Chapter 5 discussing the optimized two-pass technique. Radix partitioning splits the input dataset into 2^r partitions with r radix bits indicating the number of low order bits of a key used to group elements into the same partition. The algorithm first randomizes each key in `dedupR/S` using a hash function and a secret to remove any correlations in the join keys of each input table.

Algorithm 4 describes the radix partitioning mechanism applied on `dedupR/S` to divide the deduplicated tables into multiple partitions. The algorithm first computes the number of partitions $P = 2^r$. For each tuple in the input, it first randomizes the join key using a hash function (line 8) and extracts a fixed number, r, of low-order bits from the hashed value to determine the partition index. Because hash functions are deterministic, applying the same hash function on both inputs, `dedupR` and `dedupS` generates the same randomized output for matching keys. This ensures that matching keys from both tables will be mapped into the same partition, enabling independent and parallel join operations on each partition pair.

Since matching de-duplicated keys are always placed in corresponding partitions but non-matching keys can fall into any partition, an adversary could potentially infer the number of unique keys in the join output m' from repeated executions on the same dataset. Thus, m' is publicly revealed and modeled as a leakage simulated by a polynomial time simulator. Note that our algorithm ensures indistinguishability between any two sets of input

Algorithm 2 Oblivious Count of Consecutive Duplicate Keys

```
1: function COUNT-CONSECUTIVE-DUPLICATES(T)
2:     n ← length(T)                          ▷ T is an input table/array of join keys
3:     count ← [0]^n
4:     prev ← T[0].k                          ▷ Initialize with the first key
5:     j ← 1                                  ▷ Running count of consecutive duplicates
6:     for i ← 1 to n − 1 do
7:         curr ← T[i].k                      ▷ Load current key
8:         flag ← (curr ≠? prev)
9:         OCmpSet(flag, count[i−1], j)       ▷ Write count only if keys differ
10:        j ← j + 1
11:        OCmpSet(flag, j, 1)                ▷ Reset j if keys differ
12:        prev ← curr                        ▷ Update previous key
13:    end for
14:    count[n−1] ← j                         ▷ Flush final count
15:    return count
16: end function
```

tables that have the same n_1, n_2, m, and m' (Chapter 6). To hide which indices in `dedupR` and `dedupS` correspond to m', we perform oblivious shuffling [37] before partitioning.

The partitioning algorithm first computes the partition markers, `partMrkr`, which define the starting offsets of each partition by calculating the number of elements per partition in `hist` (lines 7-14). It then iterates over each tuple and appends them to the appropriate partition in `partitioned` (lines 15-19). The algorithm outputs `partitioned` and `partMrkr` with tuples in each partition placed consecutively and `partMrkr` marking partition boundaries. For example, given the deduplicated tables `dedupR` and `dedupS` from Section 4.1.2, we get:

$$\texttt{partMrkrR} = [0, 3, 5] \quad \texttt{partitionedR} = [(2, k_2, 2),\ (3, k_3, 1),$$
$$(5, d_3, 0), (1, d_1, 0),\ (6, k_6, 3),\ (0, k_1, 1),\ (4, d_2, 0)]$$
$$\texttt{partMrkrS} = [0, 2, 5] \quad \texttt{partitionedS} = [(1, k_2, 2),\ (4, k_5, 1),$$
$$(0, d_4, 0),\ (2, d_5, 0),\ (6, k_6, 2),\ (3, k_4, 2),\ (5, d_6, 0)]$$

The number of partitions $P{=}2^r$ is chosen such that each partition fits into the processor's cache or TLB. As a result, radix partitioning is considered a "hardware-conscious" partitioning algorithm. Previous studies on radix partitioning [56, 10, 9] have extensively

Algorithm 3 Replacement of Duplicate Keys with Dummies

```
1: function REPLACE-WITH-DUMMIES(T, count)
2:     n ← length(T)                                  ▷ T is array of join keys
3:     dedup ← [ ]^n                                  ▷ Initialize output array
4:     for i ← 0 to n − 1 do
5:         k ← T[i]                                   ▷ Load current key
6:         cnt ← count[i]                             ▷ Repetition count from section 4.1.1
7:         newKey ← hash(T‖k‖i)                       ▷ Generate unique dummy
8:         OCmpSet((cnt ≠? 0), newKey, k)
9:         dedup[i] ← (i, newKey, cnt)
10:    end for
11:    return dedup
12: end function
```

analyzed the performance impact of different partition sizes and the cache behavior of the algorithm. Its ability to benefit from cache coherence is one of its major advantages compared to other partitioning mechanisms.

Note that Algorithm 4 does not hide which tuples belong to a partition nor does it strictly enforce each partition to be of a fixed length. Apart from revealing m', this has another repercussion on the privacy guarantees of the algorithm. Because both `dedupR` and `dedupS` use the same parameter r, both tables will create the same number of partitions, P, enabling pair-wise comparisons. However, if one of the tables (say `dedupR`) is very small and P is large enough, then some partitions of `dedupR` may be empty. The adversary can infer that none of the tuples in the corresponding partition of `dedupS` will be in the output. This can be mitigated by reducing the number of partitions created. We model this as a balls-into-bins problem to derive the appropriate bounds.

Lemma 1. *Let n balls be uniformly randomly put into m bins* [1] *and p be the probability indicating that any bin will be empty. Then $p \leq m \cdot e^{-n/m}$.*

Proof. $\forall_i \in [m] : E_i$ be an event that bin i is empty. Then probability $p = P(E_1 \vee E_2 \vee \cdots E_m)$.

$$\text{Probability that bin 1 is empty: } P(E_1) = (1 - 1/m)^n$$

[1] In our case, $n = min(n_1, n_2)$ and $m = P = 2^r$ partitions with r radix bits.

Algorithm 4 Radix Partitioning

```
1: function RADIX-PARTITION(A, r)
2:     n ← length(A)                                   ▷ A is dedupR/S
3:     P ← 2^r                                         ▷ Number of partitions
4:     hist ← [0]^P                                    ▷ Counts of tuples per partition
5:     partMrkr ← [0]^P                                ▷ Beginning idx of each partition
6:     partitioned ← ∅                                 ▷ Partitioned output array
7:     for i ← 0 to n − 1 do                           ▷ Count tuples per partition
8:         A[i].hKey ← hash(A[i].key)                  ▷ Hash the key
9:         idx ← partition index from lower r bits of A[i].hKey
10:        hist[idx] ← hist[idx] + 1
11:    end for
12:    for i ← 1 to P − 1 do
13:        partMrkr[i] ← partMrkr[i − 1] + hist[i − 1]
14:    end for
15:    for i ← 0 to n − 1 do                           ▷ Scatter tuples into their partitions
16:        idx ← partition index from lower r bits of A[i].hKey
17:        partitioned[partMrkr[idx]] ← A[i]
18:        partMrkr[idx] ← partMrkr[idx] + 1
19:    end for
20:    return partitioned, partMrkr
21: end function
```

$$
\begin{aligned}
\text{Union bound: } P(E_1 \vee E_2 \vee \cdots E_m) &\leq m \cdot P(E_1) \\
&\leq m \cdot (1 - 1/m)^n \\
&\leq m \cdot e^{-n/m} \\
\text{(by the inequality } 1 - x <= e^{-x} \text{ using the Taylor expansion of } e^{-x})&
\end{aligned}
$$

Thus, $p \leq m \cdot e^{-n/m}$ □

Therefore, given p, a low probability value that any bin (or partition) is empty, we can either reduce m or increase n, if necessary, such that $m \cdot e^{-n/m}$ will be less than or equal to p. We note that for all the datasets used in our evaluation (Chapter 7), we set m to guarantee non-empty partitions.

4.3 Exchanging Repetition Counts

The next step of the algorithm is to exchange the counts of overlapping keys between the two partitioned tables, as shown in step 3 of Figure 3.1a. These exchanged values determine the *expansion counts* for each tuple in the input tables. The expansion count associated with each key indicates how many matches exist in the opposing table, and consequently, how many times the key must be repeated in the output. An expansion count of 0 indicates that no matching key exists in the other input table, while a non-zero value indicates the presence of one or more matches.

Data partitioning allows different partitions to exchange the counts in parallel without impacting correctness. Algorithm 5 describes how to obliviously compare a partition p_r from R with the corresponding partition p_s of S and exchange the counts.

We adopt a hash-and-probe strategy to keep the number of comparisons within a partition minimal by inserting tuples in p_r into a hash table and probing it for each element in p_s. However, to preserve obliviousness, the use of any conventional (plaintext) hash table as-is is problematic as it introduces two data-dependent leakages: (i) any tuple in p_r's hash table that is never probed can be inferred to not appear in the output; (ii) any tuple in p_s that fails to find a match is similarly guaranteed to be excluded from the output. Both leakages violate the obliviousness guarantees of the join.

While we can use oblivious hash tables to mitigate these leakages, existing constructions typically incur $O(N \cdot \text{polylog}N)$ overhead [18, 89, 67, 8] or suffer from high constant factors [7], making them impractical in our setting. Because the keys are randomized and due to the specific join use case, the only leakages we need to mitigate are those outlined above. Thus, full-fledged oblivious hash tables are unnecessary and overly expensive for our use case.

We address the above leakages using two key techniques. First, we ensure that every bucket in the hash table is probed at least once with a high probability by bounding the number of hash buckets, b. This can again be modeled as a balls-into-bins problem, where $|p_r|$ (or $|p_s|$) unique randomized elements are uniformly mapped to b buckets. Using Lemma 1, we can determine a value of b that yields, with a high probability $1 - p$, no bucket remains empty or un-probed. As a result, the adversary cannot infer whether any particular tuple in R will be excluded from the output.

Second, to hide whether a tuple $sTup$ in p_s matches any in p_r, the algorithm compares it with all tuples in the corresponding bucket (line 11). For each comparison, it obliviously updates the counts in p_r and p_s (lines 13–15); this ensures a uniform memory access pattern regardless of whether a match occurs. Together, these techniques eliminate the leakages

Algorithm 5 Oblivious Exchange of Repetition Counts

1: **function** EXCHANGE-COUNTS($partR$, $partS$)
   Input: $partR$, $partS$: one radix-partition pair of tuples
   Output: Expansion counts updated in place for both partitions
2: $N \leftarrow$ a power of two that satisfies Lemma 1
3: $bucket \leftarrow [0]^N$ ▷ An array of arrays

   Build a bucket-chained hash table
4: **for** $i = 0$ **to** $|partR| - 1$ **do**
5: $idx \leftarrow$ `GetBucketIndex(`$partR[i].hKey$`)`
6: $bucket[idx] \xleftarrow{\cup} partR[i]$
7: **end for**
   Probe the hash table
8: **for** $i = 0$ **to** $|partS| - 1$ **do**
9: $idx \leftarrow$ `GetBucketIndex(`$partS[i].hKey$`)`
10: $sTup \leftarrow partS[i]$
11: **for** $j = 0$ **to** $|bucket[idx]| - 1$ **do**
12: $rTup \leftarrow bucket[idx][j]$
13: $flag \leftarrow \neg((rTup.cnt \stackrel{?}{=} 0)\ \&\ (sTup.cnt \stackrel{?}{=} 0)\ \&\ (rTup.key \stackrel{?}{=} sTup.key))$
14: `OCmpSet(`$flag, sTup.exp, rTup.cnt$`)`
15: `OCmpSet(`$flag, rTup.exp, sTup.cnt$`)`
16: **end for**
17: **end for**
18: **end function**

in regular hash tables discussed above. Specifically, because every tuple in R and S is accessed and updated at least once in Algorithm 5, the adversary cannot distinguish which tuples contribute to the final output.

4.4 Re-populate Input Tables

With the expansion counts exchanged across corresponding partitions of the input tables, the next step re-populates the tables by replacing the dummy keys of de-duplicated entries with real keys and by propagating the expansion counts, as shown in step 4 of

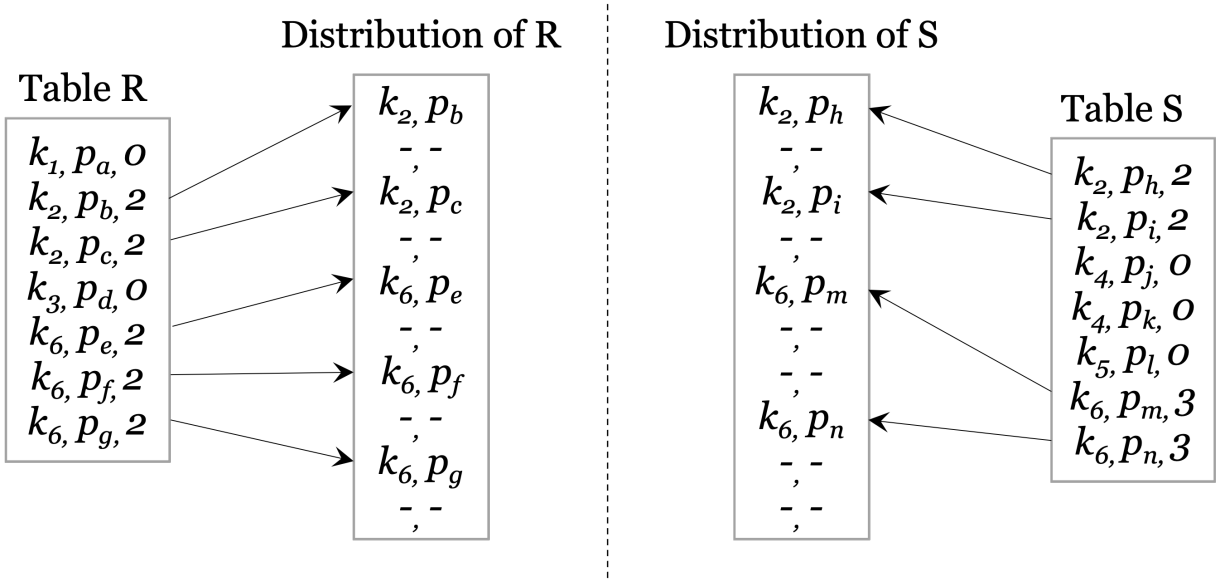


Figure 4.1: Result of distribution on R and S.

Figure 3.1b. First, all keys (real or dummy) from the partitions are scattered back to their pre-partitioned positions in `dedupR` and `dedupS` using the associated indices. Note that `dedupR` and `dedupS` were obliviously shuffled before partitioning, so the scatter operation returns them to their shuffled order. The original order is restored using techniques from prior work [40, 6, 37, 85].

Next, the algorithm performs a reverse scan over the input tables and replaces each dummy key with the most recent preceding real key and its associated expansion count, as illustrated in Figure 3.1b.

4.5 Oblivious Distribution and Expansion

Step 5 of the join algorithm *distributes and expands* the tuples of R and S into their respective output arrays. This section describes how distribution is performed using oblivious primitives, followed by the expansion procedure.

As depicted in Figure 4.1, the goal of distribution is to place each tuple of R and S that participates in the join into its correct location in an output array of size m, which we refer to as `expandedT` for $T \in \{R, S\}$. For a tuple with a non-zero expansion count (i.e.,

Algorithm 6 Oblivious Distribution and Expansion

1: **function** DISTRIBUTEANDEXPAND(T)
2: $prefixSum \leftarrow 0$
3: **for** $i \leftarrow 0$ **to** $|T| - 1$ **do**
4: $flag \leftarrow (T[i].exp > 0)$
5: $T[i].idx \leftarrow \mathsf{OCmpSet}(flag, prefixSum, dummy)$ ▷ *dummy* is a large sentinel index (conceptually $+\infty$)
6: $marked[i] \leftarrow flag$
7: $prefixSum \leftarrow prefixSum + T[i].exp$
8: **end for**
9: $m \leftarrow prefixSum$ ▷ total expanded length
10: OCOMPACT(T, $marked$)
11: `expandedT` $\leftarrow$ PADTOSIZE(T, m) ▷ append dummies if $|T| < m$, truncate otherwise
12: ODISTRIBUTE(`expandedT`, 0, m)
Forward fill dummies with real rows
13: Scan `expandedT` from position 0. Retain non-dummy rows as is and replace dummy rows with the most recent non-empty row. Index 0 is guaranteed to be non-dummy.
14: **return** `expandedT`
15: **end function**

key is present in both input tables), its output position in `expandedT` corresponds to the summation of all expansion counts of preceding tuples. For example, consider tuple 3 in R with key k_2 and payload p_c: its target position in `expandedR` is 2, obtained by summing the expansion counts of tuples 1 and 2. We refer to these target output indices as *prefix sums*.

A naive strategy would write each tuple with a non-zero expansion count to its prefix-sum–determined position while leaving the remaining positions empty. This leaks (i) which tuples in T contribute to the final result, and (ii) how many times each key participates in the join, as evidenced by the gaps in the partially filled output array. Preventing these leakages by writing to all m locations for each of the $|T|$ input tuples incurs an inefficient $O(|T| \cdot m)$ cost.

Algorithm 6 begins by scanning T to compute prefix sums and marking matching keys. For each index i:

- A flag is computed indicating whether $T[i].exp > 0$.

- If the flag is true, the tuple's `idx` field is set to the current prefix sum; otherwise, it is assigned a dummy value. The dummy value need not be unique but must exceed any valid join output position. Because the final output size m is not yet known during this scan, we assign a dummy index equal to the maximum representable value of the index data type (e.g., `UINT32_MAX` for a 32-bit unsigned integer).
- The flag is recorded at index i in a boolean array `marked` of size $|T|$.

When the scan completes, the final prefix sum equals m, the total number of tuples that will appear in the join output.

Given `marked`, we apply oblivious compaction to pull all tuples with non-zero expansion counts to the front of T, preserving their original relative order. This produces an array with exactly m relevant tuples at the top. We then resize T to length m to obtain `expandedT`. If $m > |T|$, we append dummy tuples; if $m < |T|$, we truncate the remaining entries, which do not contribute to the join.

At this point, the relevant tuples are stored contiguously, but each must be moved to its target index (stored in `idx`) in the output array. Conceptually, this is the inverse of compaction: rather than pulling tuples together, we spread them out to their previously calculated prefix sum positions.

We perform this distribution by applying the compaction technique of Sasy et al. [74] in reverse. Combined with the earlier oblivious compaction step, this yields a complexity of $O(|T| \log |T| + m \log m)$, improving upon the $O(|T_C| \log^2 |T_C| + m \log m)$ approach of KKS [46] (where $T_C = R + S$), which relies on the compaction design of Goodrich et al. [30]. The corresponding phases in Obliviator [57] have a theoretical complexity of $O(|T_C| \log |T_C| + m \log m)$; however, our experiments show that their implementation produces incorrect outputs in both sequential and parallel settings.

After distribution, `expandedT` contains the matching tuples at their correct prefix-sum positions and dummy entries elsewhere. A final linear scan expands the array: every dummy entry is replaced with the most recent non-dummy tuple. Since index 0 is guaranteed to contain a non-dummy tuple, this expansion procedure ensures both correctness and obliviousness.

The join algorithm applies the above distribution-and-expansion steps to relations R and S in parallel.

4.6 Table Alignment and Merging

After oblivious distribution and expansion, tables `expandedR` and `expandedS` contain the correct number of repeated entries per join key, but their positions are not yet aligned for the final one-to-one merge. Therefore, the algorithm next aligns expanded S, as illustrated in step 6 of Figure 3.1b.

We do not introduce a new alignment technique and adopt the method used in [46], where each tuple in `expandedS` belongs to a key-run and is numbered by a 0-based counter `q`. The join can be conceptually viewed as a `cnt×exp` grid for each key, where `cnt` is a tuple's repetition count and `exp` its expansion count. For each tuple in `expandedS`, the algorithm computes a new position by mapping `q` into this grid and flattening it in column-major order. This causes tuples in `expandedS` to be spaced apart by `cnt`, aligning precisely with the block layout in `expandedR`.

To enforce this layout, the algorithm obliviously sorts `expandedS`, using a composite key of `(join key, computed position)`. This guarantees that the rows in `expandedS` appear in the same order as their matching blocks in `expandedR`, without revealing any data-dependent access patterns. Alignment techniques in both KKS [46] and Obliviator [57] rely on this sorting step to complete the transformation. For more details, we refer to the alignment algorithm from [46] in Algorithm 7.

Upon aligning `expandedS`, the final step of the algorithm simply merges `expandedR` and `expandedS` tuple-wise to produce the final output, as shown in step 7 of Figure 3.1b.

4.7 Foreign Key Joins

Our non-foreign key (i.e., many-to-many) join in Algorithm 1 can be easily adapted for a foreign key join (i.e., one-to-many). Without loss of generality, assume R is the primary key table and S is the foreign key table. We exploit two properties specific to foreign key joins: (i) join keys in R are unique and hence have a repetition count of 1, and (ii) if a key appears in S, it must also appear in R.

Because R contains no duplicates, the algorithm can apply RadixPartition (Section 4.2 to R directly after obliviously shuffling it, skipping the steps involved in de-duplication. Even for unsorted inputs, it only needs to obliviously sort S for de-duplication, unlike Obliviator [57], which requires an oblivious sort on $n_1 + n_2$ to identify keys from R that are in S. Since all keys in R have a repetition count of 1, we further optimize the exchange counts step by directly copying the payload from R to S upon a match, eliminating the need

Algorithm 7 AlignTable

function AlignTable($expandedS$) ▷ $expandedS$ has m tuples
  $m \leftarrow |expandedS|$
  $W[0 \ldots m-1] \leftarrow 0$ ▷ within-key counters
  $prevKey \leftarrow expandedS[0].key$
  $idx \leftarrow 0$
  $W[0] \leftarrow 0$
  for $i \leftarrow 1$ **to** $m-1$ **do**
    $cur \leftarrow expandedS[i].key$
    $same \leftarrow -(\, cur = prevKey \,)$ ▷ $0/-1$ word mask
    $diff \leftarrow\sim same$ ▷ bitwise NOT
    $idx \leftarrow (same \wedge (idx+1)) \vee (diff \wedge 0)$
    $prevKey \leftarrow (same \wedge prevKey) \vee (diff \wedge cur)$
    $W[i] \leftarrow idx$
  end for
  for $i \leftarrow 0$ **to** $m-1$ **do** ▷ compute final `idx`
    $q \leftarrow W[i]$
    $a_2 \leftarrow expandedS[i].exp$; $a_1 \leftarrow expandedS[i].cnt$
    $row \leftarrow \lfloor q/a_2 \rfloor$
    $col \leftarrow q - row \times a_2$
    $expandedS[i].idx \leftarrow row + col \times a_1$
  end for
  sort $expandedS$ by (`key`, `idx`)
  return $expandedS$
end function

to distribute and expand R. Moreover, because S now contains the matching payloads, the alignment and merging steps can also be replaced with an oblivious compaction.

To summarize: for presorted inputs, the algorithm de-duplicates S, shuffles and partitions both R and S, and compares partitions pairwise to identify matching keys. For each match, it copies R's payload to S obliviously, restores the original order of S, and then invokes oblivious compaction on S to produce the final output.

Chapter 5

Parallelization Techniques

This chapter discusses the parallelization techniques we employ in the implementation to speed-up various phases of the algorithm, beginning with the partitioning phase. While the algorithms for partitioning and per-partition comparison are described separately in Chapter 4, our implementation executes them into a single function that jointly partitions both input tables and computes per-partition comparisons. We adopt the two-pass radix partitioning scheme from prior work [9, 55] with multi-threading support. The first pass partitions the input into $2^{\lfloor r/2 \rfloor}$ sets, and each of these is further partitioned into $2^{\lceil r/2 \rceil}$ sets. Future optimizations could include extending this to a three-pass partitioning scheme to reduce TLB thrashing for larger datasets or adopting software-managed buffers to improve cache efficiency.

Radix partitioning is parallelized by subdividing both input relations into sub-relations assigned to individual threads. Each thread first scans its input slice to build local histograms, and a global prefix-sum determines the exact output offset for every thread-partition pair. Each thread is then allocated a private contiguous range in every partition, eliminating contention. In the second pass, all threads write their tuples into their reserved regions without synchronization. Importantly, both the number of partitions and the per-thread output ranges are deterministic functions of the public input size and radix parameters. Hence, even if an adversary observes per-core memory activity, the access pattern depends only on public parameters and not on the secret data values.

The remaining stages in our algorithm adopt a slice-based parallelization model tailored for modern shared-memory, multi-core architectures. The central idea is to partition the input relations into contiguous slices that are processed independently by multiple threads. Within each slice, the computation leverages a combination of SIMD acceleration, cache-

Table 5.1: Parallel approach of calculating repetition counts (as in Algorithm 2). The sorted input is evenly divided among T=4 threads with each thread computing local counts.

Index	**Key**	**Slice/Thread**	**Local Rep. Count**
0	k_1	Slice 0 (T0)	0
1	k_1	Slice 0 (T0)	2
2	k_1	Slice 1 (T1)	0
3	k_1	Slice 1 (T1)	2
4	k_1	Slice 2 (T2)	1
5	k_2	Slice 2 (T2)	1
6	k_3	Slice 3 (T3)	1
7	k_4	Slice 3 (T3)	1

Table 5.2: Forwarding repetition counts across slice boundaries when keys span multiple slices. If the last key in slice i equals the first key in slice i+1, its local repetition count from slice i is forwarded and cleared from slice i.

Slice $i \to i$+1	**Last = First Key**	**Count(i) Forwarded**	**Clear Count(i)**
0→1	$k_1 = k_1$	2	✓
1→2	$k_1 = k_1$	4	✓
2→3	$k_2 \neq k_3$	0	×

Count(i) refers to the local repetition count of the last key in slice i. Forwarded values accumulate across slices when runs span multiple boundaries (e.g., $2+2=4$ for k_1). This step is sequential over $T-1$ boundaries and runs in $O(T)$ time, where T is the number of threads.

aware loop structures, and branch-free operations to maximize throughput while maintaining obliviousness. At the end of each stage, a lightweight global barrier ensures data consistency before moving on to the next phase. This static partitioning strategy ensures low synchronization overhead and avoids the need for complex locking or fine-grained coordination. To further improve load balance, the number of threads allocated to each input relation is dynamically determined in proportion to its size, accommodating variations in input table sizes.

Importantly, our parallelization preserves obliviousness: slices are partitioned deterministically according to public input size and number of threads. Each slice executes the same sequence of operations, independent of key values. Even the boundary-correction step proceeds in fixed order over the $T-1$ slice boundaries, with no data-dependent branching.

Thus, parallel execution leaks no additional information beyond the public parameters.

Consider the sequential repetition count computing phase in Algorithm 2. Our implementation parallelizes this by first slicing the input across threads and computing local statistics, as shown in Table 5.1. Then in the next pass, slice i compares if its last key matches slice $i+1$'s first key; if yes, the counts are carried forward to the next slice and the count is cleared from slice i, as shown in Table 5.2. In the final pass, each slice recalculates the repetition count by first adding the forwarded value to their first remaining non-zero key's count. Thus, the final count array is [0,0,0,0,5,1,1,1] for the given example. Note that all comparisons and count computation are executed with `OCmpSet`. All linear steps of our join algorithm follow a similar design in which threads process local slices first and then apply boundary corrections, if necessary.

Because oblivious sorts dominate the runtime of oblivious join algorithms, our implementation integrates Obliviator's sorting algorithm [57, 62] together with its parallelization strategy to eliminate performance differences arising from variations in oblivious sorting implementations.

For oblivious compaction and oblivious distribution, we adopt the parallelization strategy proposed by Sasy et al. [74], which is proven to be oblivious. For oblivious shuffling, we integrate the implementation of Gu et al. [37]. Their construction is not parallelized in the publicly available code, although they note that it is parallelizable in principle. As a result, our system exhibits lower scalability under multi-threaded execution than would otherwise be possible, as observed in our evaluation (Chapter 7).

Chapter 6

Security Arguments

Theorem. *Our join algorithm presented in Algorithm 1 is oblivious according to Definition 1.*

Proof sketch: Informally, our join algorithm is oblivious because its observable behavior—namely, logical and physical memory accesses, and control flow— is independent of the input data. A formal proof presented in Chapter 6.1 demonstrates that a probabilistic polynomial-time (PPT) simulator can replicate the algorithm's behavior using only public inputs (the sizes of the input tables) and a predefined leakage function (the size of the join output and the number of unique keys in the output). We argue that each core component of the join algorithm maintains obliviousness. Specifically, the radix partitioning phase maintains obliviousness by de-duplicating and randomizing join keys before partitioning, thereby hiding the frequency distribution of keys in the input. We also show that the exchange counts phase, which relies on a hash-and-probe strategy, is oblivious by bounding the number of hash buckets and accessing all elements per bucket using oblivious compare-and-set (`OCmpSet`). The simulator can emulate these phases using only public parameters and generate identical memory access behavior. By simulating each of the sub-algorithms presented in Chapter 4, our join construction is shown to satisfy obliviousness.

6.1 Security Arguments

This section discusses arguments to show the proposed join algorithm is oblivious according to Definition 1. Informally, an algorithm is considered oblivious if for any two inputs

and outputs of the same length, the adversary observable information for the two remains indistinguishable. Algorithms executed within TEEs must ensure that their logical data accesses (i.e., rows of a table being accessed), the physical memory accesses (i.e., memory locations being touched), and the program control flow of the algorithm remain independent of the data. If a probabilistic polynomial time (PPT) simulator exists that can emulate the behavior of an algorithm based solely on public parameters and defined leakage functions, then the algorithm is considered oblivious because its observable behavior remains indistinguishable to a data-oblivious simulator. In our work, the lengths of the two input tables are considered public parameters, and the size of their join output along with the number of unique keys in that output are modeled as the allowed leakage, $\mathcal{L}$. Recall the formal definition:

Definition. *An algorithm Alg is oblivious if there exists a polynomial time simulator Sim such that*

$$|Pr[\mathcal{G}_{Alg} \Rightarrow \mathsf{True}] - Pr[\mathcal{G}_{Sim,\mathcal{L}} \Rightarrow \mathsf{True}]| \leq negl(\lambda)$$

where λ is the security parameter.

We show our join algorithm is oblivious by first discussing the obliviousness of its components.

Theorem 1. *Our radix partitioning approach is oblivious.*

Proof. Let R and S be input arrays of tuples (k_i, p_i), where k_i is the key and p_i an arbitrary payload. Our oblivious radix partitioning proceeds in three conceptual steps: sorting, counting/de-duplication, and partitioning.

1. Oblivious Sort: If either of the inputs is unsorted, we first independently sort the unsorted input(s) on the keys using a provably oblivious sorting algorithm (e.g., bitonic sort [11]). Note that the proposed scheme can use any oblivious sorting algorithm; our implementation uses bitonic sort. If the data is known to be presorted, our algorithm skips this step.

2. Oblivious Counting and De-duplication: To handle duplicate keys, we compute repetition counts and replace repeated keys with dummy values using Algorithms 2 and 3. Both algorithms operate in $O(n_1)$ and $O(n_2)$ for R and S respectively, where $n_1 = |R|$ and $n_2 = |S|$ are public parameters. Neither algorithm contains data-dependent branching. The counting procedure and the step that replaces a repeated key with a dummy key are written using bitwise operations and exhibit identical memory-access patterns independent of the actual key values or their multiplicities.

Since all steps depend only on the public input sizes, a PPT simulator *Sim* can generate arrays of the same sizes and perform the same sequence of memory accesses, producing a transcript that is computationally indistinguishable from the real execution. This guarantees that the algorithm reveals no additional information beyond the public parameters n_1 and n_2.

Note that if the data is known to contain only unique keys, as in the primary-key table in foreign-key joins, our algorithm skips the first two steps.

3. Oblivious Partitioning: After de-duplication, the algorithm applies the radix partitioning logic. Algorithm 4 depicts the procedure for radix partitioning R' and S', which are de-duplicated versions of R and S respectively consisting of unique keys. The algorithm first randomizes each key in R' and S' using the same hash function and secret. Based on the last r bits of the hash value, it partitions both de-duplicated tables into $P = 2^r$ parts. Key de-duplication and randomization ensure that all partitions have approximately uniform sizes.

Note that this partitioning function does not hide which partition an element at a given index falls into. Matching de-duplicated keys from R' and S' are always placed in corresponding partitions, whereas non-matching keys can fall into any partition. As a result, the total number of unique keys in the join output, m', is considered a public parameter. To randomize which indices in the de-duplicated input tables correspond to these unique keys, we first obliviously shuffle both input tables. Shuffling produces a randomized ordering in each run, ensuring that the assignment of keys to partitions does not leak information about the original positions of unique keys. By leveraging this randomized ordering and uniformly distributing keys across all partitions, the algorithm ensures that no information about the original input ordering or key matches is revealed.

Because Algorithm 4 contains no data-dependent branching and all looping constructs depend only on $n_1 = |R'| = |R|$ and $n_2 = |S'| = |S|$, its behavior can be efficiently simulated by a probabilistic polynomial-time (PPT) simulator *Sim*. *Sim* takes as input the parameters n_1, n_2, r, m', and the same hash function as the real algorithm. It first generates m' unique random keys and assigns them to both R' and S'. It then extends R' to size n_1 by adding $n_1 - m'$ unique random keys that appear only in R' and similarly extends S' to size n_2 by adding $n_2 - m'$ unique random keys that appear only in S'. After constructing these inputs, *Sim* applies the same partitioning logic to R' and S' as the real algorithm does. The resulting distribution is computationally indistinguishable from the output of the real execution, establishing that the radix partitioning procedure is oblivious.

Note that if the number of partitions, P, is too large or if the input array is too small, some partitions may be empty. Our oblivious join construction requires partitions to be

non-empty with a high probability. Therefore, based on Lemma 1, both the real algorithm and the simulator can first either increase n by adding (unique) dummy elements or decrease P to ensure that the probability of a partition being empty remains negligibly low. Because n and P are public parameters, the probability that a partition is empty is already known to the adversary; thus, adjusting n or P to prevent empty partitions does not violate obliviousness, as this behavior can be simulated by *Sim*. □

Next, we discuss the obliviousness of the EXCHANGE-COUNTS component of the join algorithm.

Theorem 2. *The exchange counts procedure described in Algorithm 5 is oblivious.*

Proof. Algorithm 5 takes as input a partition p_r of R' and the corresponding partition p_s of S'. Using these inputs, Algorithm 5 compares tuples in the two partitions using a hash-and-probe technique. For each comparison, it exchanges the repetition count `cnt` of a key from one table as expansion count `exp` of the other table if a matching key is found, and otherwise, rewrites the `exp` as is (set to a default of 0) using `OCmpSet`. By ensuring that the number of hash buckets generated by p_r satisfies Lemma 1 such that every bucket will be probed with a sufficiently high probability, Algorithm 5 accesses and updates the counts of each tuple in p_r and p_s.

The above functionality can be emulated by a simulator *Sim*, which takes as input $|p_r|$ and $|p_s|$, which is about $|T|/P$ (where $T \in \{R', S'\}$) and generates two random arrays a_r and a_s of the corresponding sizes. The simulator then applies partition-wise comparison by storing the elements from a_r in a hash table and probing the elements of a_s. For each comparison, the simulator mimics the memory accesses of compare and swap, similar to the real algorithm. Since the number of hash buckets created depend on Lemma 1, which relies on the sizes of the partitions, the number of buckets per hash table in the simulator will be indistinguishable from that of the real algorithm. Because Algorithm 5 can be simulated by a data-independent simulator, it is oblivious. □

Theorem 3. *Repopulating the input tables as described in Section 4.4 is oblivious.*

Proof. Before repopulating the obliviously shuffled tables, we first restore their original relative ordering. This follows the proven-oblivious procedures used in [40, 6, 37, 85].

To repopulate the input tables, the algorithm performs a reverse scan over R' and S' and rewrites each dummy key using the most recently encountered real key. The distinction between real and dummy keys is determined by the tuple's `cnt` field: Algorithm 3 obliviously replaces a key with a dummy key when its corresponding `cnt` is zero. The

reverse scan works because duplicate keys appear contiguously as the original inputs R and S are sorted, and Algorithm 3 preserves only the last key in each run of duplicates. During repopulation, the algorithm performs a write at every index using `OCmpSet`, regardless of whether a key is real or dummy. Thus, the memory-access pattern is identical for all possible inputs.

A PPT simulator *Sim*, given only the public parameters $n_1 = |R|$ and $n_2 = |S|$, can reproduce the exact same memory-access patterns. During a linear reverse scan, the simulator rewrites each entry of a dummy array of size n_1 or n_2. This simulated behavior is identical to the real algorithm, demonstrating that repopulating both input tables is oblivious. □

Theorem 4. *The oblivious distribution and expansion described using Algorithm 6 are oblivious.*

Proof. Algorithm 6 takes as input a table T of size n, where each tuple has an expansion count exp. The algorithm performs a linear scan over T to compute prefix sums, which determine each tuple's target position in an expanded table of size m. Tuples with $exp = 0$ are assigned a dummy prefix sum (e.g., `UINT32_MAX`), indicating that they do not contribute to the join output. During the scan, in each iteration i, the algorithm assigns the target index for $T[i]$ to either the running prefix sum or the dummy value using `OCmpSet`, based on the flag $T[i].exp > 0$. The corresponding flag is always written in a data-independent manner to `marked`[i] in every iteration.

A PPT simulator *Sim*, given only the public parameter n, can reproduce the same memory-access pattern as follows. It generates a dummy array of n values and scans it while treating every tuple as having expansion count 0. Thus, the simulator always assigns the dummy prefix value in each iteration and always stores "false" in `marked`[i] for all $i \in [0, n)$. This exactly matches the sequence of memory accesses performed by the real algorithm, as both the prefix-sum update and the flag write occur at every index regardless of the underlying data.

Using `marked`, the algorithm next performs oblivious compaction to bring all tuples with matching join keys to the top of T while preserving their relative order. The table is then resized to m, which is a public parameter. After resizing, the algorithm applies oblivious compaction in reverse to distribute the matching tuples to their target positions in the join output according to their prefix-sum values.

The oblivious compaction primitive we use—and consequently its reverse operation—has been formally proved oblivious by Sasy et al. [74] and is widely adopted in subsequent oblivious systems [71, 57, 84]. Because both n and m are public parameters, the resizing

step is also easy for *Sim* to reproduce: it can simply truncate $n - m$ elements, append $m - n$ random elements, or leave the array unchanged, depending on the values of of any randomly generated n and m.

Then, in a linear scan from the top, the simulator re-writes each entry of the expanded dummy array of size m. This simulated behavior is identical to Algorithms 6, indicating that the distribution and expansion phases are oblivious. □

Putting the pieces together:

Theorem. *Our join algorithm presented in Algorithm 1 is oblivious according to Definition 1.*

Proof. Steps 0-2 of our join approach (Algorithm 1) relates to applying radix partitioning obliviously, as discussed in Theorem 1. Next, we discuss the remaining components step by step.

– Step 3: The obliviousness of executing an oblivious exchange of each matching key's repetition count in one table as expansion count in the other was proved in Theorem 2.

– Step 4: Repopulating the input tables is oblivious as proved in Theorem 3.

– Step 5: Obliviousness of the distribution and expansion step of the algorithm is discussed in Theorem 4.

– Step 6: The alignment step, inherited from KKS [46], is already proven to be oblivious.

– Step 7: The final merge step simple combines the two expanded tables row-wise, which can be modeled by a simulator.

Thus, the obliviousness of our join construction holds. □

Chapter 7

Experimental Evaluation

This chapter presents the experimental evaluation of our oblivious join algorithm using a range of real-world and TPC-H [81] datasets.

Setup: Our prototype—implemented in C and C++—is built on Ubuntu 22.04. All experiments are deployed on an Azure [58] Standard DC48eds_v5 VM with 48 vCPUs (24 physical cores) and 192 GiB RAM, powered by 4$^{\text{th}}$-Generation Intel(R) Xeon(R) Scalable processors with Trusted Domain Extensions (TDX). All implementations, including the baseline, are executed with compiler optimization flag `-O3` and a default block size of 32 bytes. Unless otherwise stated, the execution uses 32 threads. Every reported number is an average of three runs to account for the performance variation in a multi-tenant cloud.

Baseline: We compare our radix-based oblivious join algorithm (termed `Radix`) with the state-of-the-art baseline, Obliviator [57], termed `OBL`. For each experiment, we evaluate our scheme `Radix` with presorted input tables to highlight the benefit of our design over `OBL` for sorted inputs. While the scheme by Krastnikov et al. (KKS) [46] also has similar asymptotic complexity `OBL`, we do not compare against KKS because (i) it requires more oblivious sorts than `OBL`, and (ii) [57] reports that `OBL` outperforms KKS by upto 80×. Join schemes proposed in Opaque [90] and ObliDB [28] do not ensure double obliviousness, and hence, are not appropriate baselines for our scheme.

Obliviator [57] is deployed on Intel SGXv2, whereas we develop our scheme to be compatible with Intel TDX due to the VM-level isolation it provides. To ensure a fair comparison, we extracted `OBL`'s enclave code along with their custom multi-threading implementation directly from their SGX codebase and executed it unmodified within TDX. We also selected the TDX RAM to match the SGXv2 EPC size used by the baseline so that the evaluations are done under identical protected-memory constraints.

Table 7.1: Various dataset sizes used in the experiments

Dataset	Table 1 size (n_1)	Table 2 size (n_2)	Join output size (m)
Real-World Datasets			
Jokes	54,905	54,905	1,763,146
Amazon	1,231,439	1,231,439	4,804,985
Slashdot	549,202	549,202	49,313,268
Twitter-1	12,077,387	127,301,736	330,607
Twitter-2	31,900,338	127,301,736	111,679,643
IMDb	10,349,072	23,771,331	23,061,006
TPC-H			
Scale Factor 0.5			
q1	5,000	75,000	15,003,061
q2	75,000	75,000	225,046,028
Scale Factor 1			
q1	10,000	150,000	60,000,414
q2	150,000	150,000	900,094,328
Scale Factor 5			
q3	7,500,000	29,999,795	29,999,795
Scale Factor 50			
q3	75,000,000	300,005,811	300,005,811

An important distinction is that `OBL` has a fully parallelized implementation, whereas the oblivious shuffle in `Radix` runs sequentially. Consequently, the multi-threaded results we report for `Radix` include the time spent in this sequential shuffle phase. In Figures 7.1, 7.2, and 7.3, the shuffle cost for `Radix` is shown separately in a distinct color to highlight its impact on the total runtime.

Datasets: We compare `Radix` and `OBL` on a range of real-world datasets – the same ones used in [57]. These include five real-world datasets: Amazon [87, 52], Jokes [29, 52], Slashdot [50, 52], and Twitter Social Graph [16, 20, 53] for non-foreign-key joins and IMDb [50, 52] for foreign-key joins. Additionally, we use the standardized TPC-H benchmark [81] at scale factors 0.5 through 1 for non-foreign-key joins [20] and scale factors 5 through 50 for foreign-key joins [57].

Table 7.1 summarizes the input and output sizes for each dataset. For the foreign-key join datasets (IMDb and TPC-H scale factors 5 and 50), n_1 denotes the primary-key table, while n_2 denotes the foreign-key table.

TPC-H Queries: Non-foreign key join (adapted from [20], also used in [57]):

q1:
```
SELECT s_suppkey, c_custkey, s_nationkey
FROM Supplier, Customer
WHERE s_nationkey = c_nationkey;
```

q2:
```
SELECT c1.c_custkey, c2.c_custkey, c1.c_nationkey
FROM Customer c1, Customer c2
WHERE c1.c_nationkey = c2.c_nationkey;
```

Foreign key join (used in [57]):

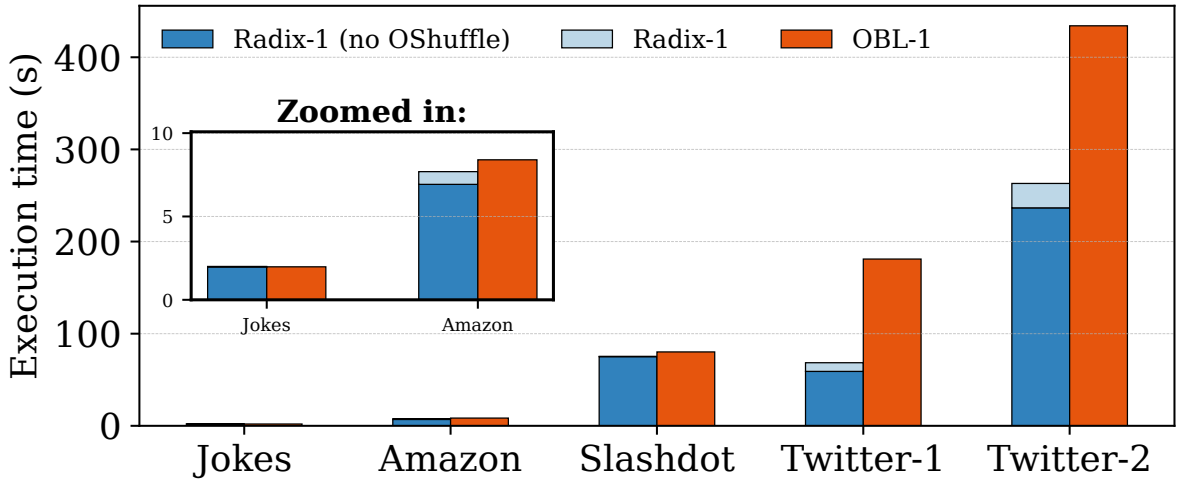
q3:
```
SELECT l_orderkey, o_orderkey
FROM LineItems, Orders
WHERE l_orderkey = o_orderkey;
```

7.1 Non-Foreign Key Joins

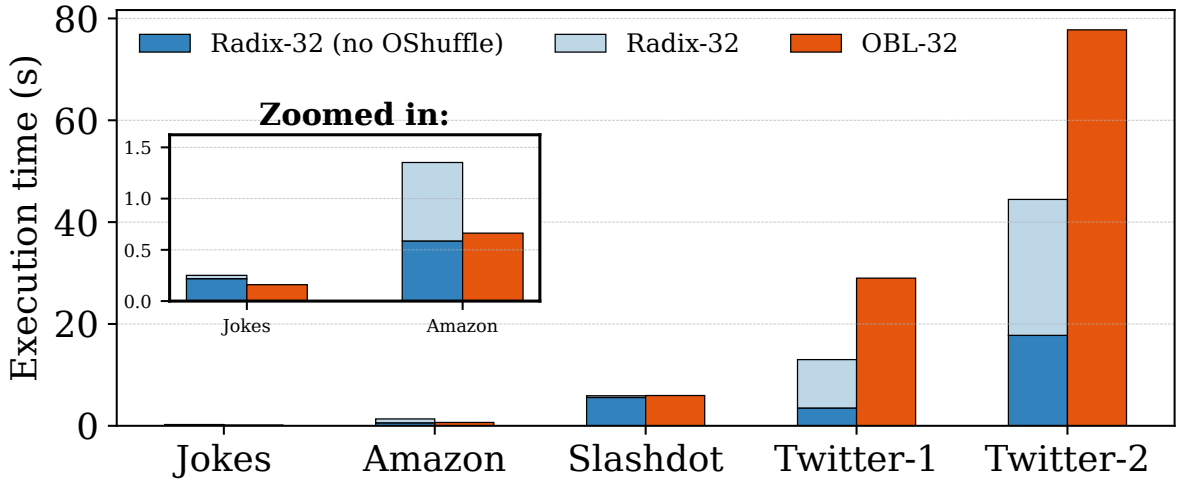
Real-World Datasets: The first experiment, shown in Figure 7.1a, compares our construction with `OBL` on real-world datasets in a single-threaded setting. We highlight two key findings. (1) When $n_1 + n_2 \ll m$ (e.g., Jokes and Slashdot), both schemes exhibit similar latency because the alignment step's oblivious sort over size m dominates the overall cost. (2) When $n_1 + n_2 \gg m$ (e.g., Twitter-1), `Radix-1` significantly outperforms `OBL-1` because it completely avoids obliviously sorting the combined input table at query time.

The next experiment studies the impact of multi-threading by running both `Radix` and `OBL` with 32 threads. The results in Figure 7.1b show that `Radix` benefits substantially when the workload aligns with its design strengths. Although its runtime is significantly slowed down by the sequential oblivious shuffle implementation of [37], which dominates the total execution time, `Radix-32` still substantially outperforms `OBL-32` on Twitter-1 and Twitter-2. When n_1 and n_2 are large (>1M), the sequential oblivious shuffle alone takes longer than all other parallel stages combined. This underscores the considerable advantage of avoiding an oblivious sort over $n_1 + n_2$ on already presorted inputs and highlight the effectiveness of our novel approach in identifying join keys shared between tables.

TPC-H Queries: We next evaluate `Radix-32` and `OBL-32` on the standardized TPC-H benchmarking for analytical workloads using 32 threads. As shown in Table 7.1, we have



(a) Results from single-threaded experiments



(b) Results from multi-threaded experiments using 32 threads. OShuffle runs sequentially.

Figure 7.1: Comparison with five real-world datasets

$n_1 + n_2 \ll m$ for these datasets; therefore, the cost of obliviously sorting the input tables is negligible. This further implies that, for both our scheme and `OBL`, the ExchangeCounts components (Section 4.3) incur minimal overhead, since they operate only on $O(n_1 + n_2)$ elements. Moreover, the proportion of runtime spent on the sequential oblivious shuffle in `Radix-32` is much smaller than in Figure 7.1b, and both schemes share similar align and merge phases. Consequently, the performance differences observed in this experiment primarily stem from how the two schemes distribute tuples to `expandedT`.

As shown in Figure 7.2, the two schemes exhibit comparable execution times on the smaller *q1* dataset, but the gap widens for *q2*. This indicates that our implementation of oblivious distribution, which follows the algorithm of [74], scales more effectively (and also ensures correctness, unlike `OBL`, as discussed in Section 4.5).

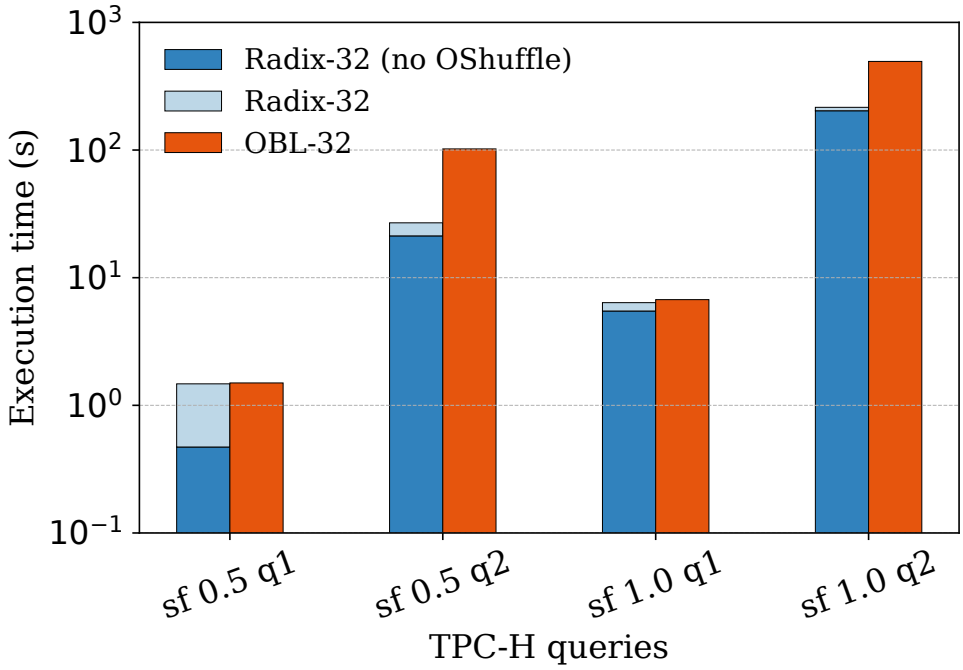


Figure 7.2: TPC-H non-foreign key joins (log scale)

7.2 Foreign Key Joins

The next set of experiments measure the execution times of `Radix` and `OBL` in performing foreign key joins in both single-threaded and multi-threaded settings. Figure 7.3 presents results on the TPC-H benchmark using the foreign key join query from [57], where n_1 ranges from 7.5M to 75M and $n_2 = m$ ranges from 30M to 300M as the scale factor increases from 5 to 50. We also compare the two schemes on the real-world IMDb dataset.

Across all single-threaded experiments, `Radix-1` consistently achieves the lowest execution times because it incurs linearithmic overhead, and its runtime is not dominated by the oblivious shuffle to the same extent as in the multi-threaded setting (`Radix-32`). Since foreign key joins require no distribution, alignment (including the oblivious sort over m), or merge phases in either scheme, the sequential oblivious shuffle becomes the primary bottleneck for `Radix-32` in the multi-threaded experiments using 32 threads, accounting for approximately 90% of the overall runtime for the IMDb and TPC-H scale factor 5 datasets.

These outcomes are in accordance with Amdahl's Law [4], which states that the overall speedup of an algorithm is limited by its sequential portions. Because the oblivious shuffle algorithm used in our implementation is claimed to be parallelizable [37], these results further highlight the potential benefits of a fully parallelized `Radix` implementation.

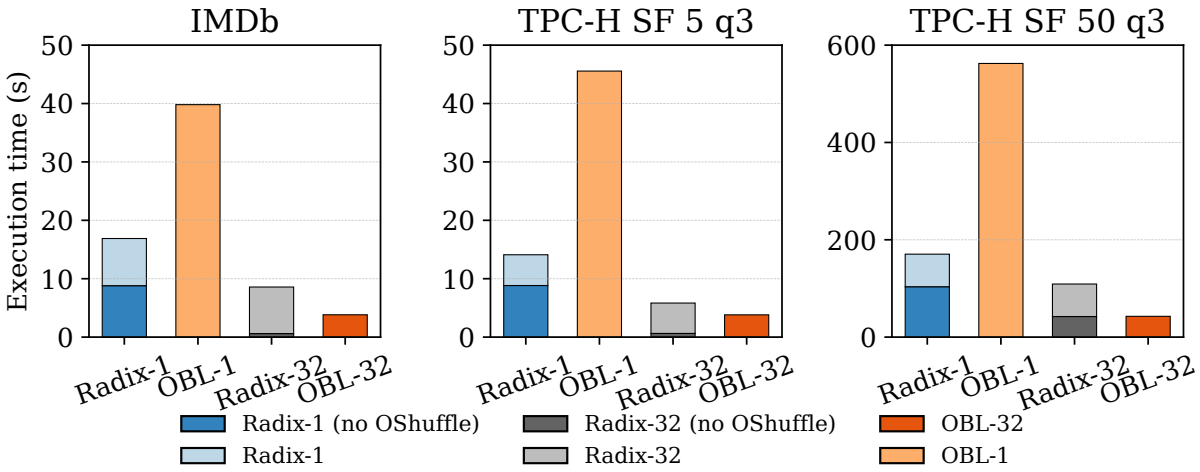


Figure 7.3: TPC-H foreign key joins

Chapter 8

Discussion

The oblivious radix partitioning (steps 0-2 in Chapter 4) presented in this work can be applied to a wide class of problems, as we briefly discuss in this chapter.

1. Oblivious Aggregation: Many analytical systems compute aggregate operations such as `MIN, MAX, SUM, AVG, and COUNT` after applying a `GROUP BY` clause on two or more tables, as seen in a majority of the TPC-H queries [81]. The oblivious radix partitioning is a natural fit to parallelize executing these operators per partition. In particular, instead of exchanging counts in step 3 of the proposed algorithm, the two tables being combined can exchange the aggregated information per unique key.

2. Offline/Online Oblivious Join: Recent privacy-preserving systems have employed an offline/online paradigm, in which part of the computation is performed in an offline preprocessing phase prior to the invocation of the operation. Examples include oblivious shuffling [75] and private nearest-neighbor search [41], both of which achieve significant performance improvements over their baselines. Such a model has not been previously applied to oblivious joins primarily because determining the multiplicity of matching keys across all tables required a combined oblivious sort over the input tables [46, 53, 84, 57], which would need to be performed in the online phase. By eliminating the need for this combined sort, our approach enables the computationally intensive components—namely, the oblivious sort and the oblivious shuffle—to be executed in the offline phase. This separation of work between offline and online stages paves the way for more efficient oblivious join implementations.

2. Private Contact Discovery: Private contact discovery, deployed in Signal messaging app using hardware enclaves [78], allows users to securely find which of their contacts are also using a service. For example, users of Signal may want to connect with their

friends who use the app without sharing their entire contact list with the server, while the server desires to not disclose the entire user list to each user. Private contact discovery also played a vital role during the COVID'19 pandemic [22]. Our oblivious radix partitioning can be applied to securely discover a user's contacts in a platform efficiently because contact comparisons occur in parallel per partition. By verifying the enclave attestation, a user can check if the server is running the correct program before sharing their data.

3. Private Set Intersection/Union (PSI/PSU): PSI and PSU are a class of problems where two (or more) parties want to compute the intersection or union of their private inputs. PSI is a generalized version of private contact discovery. PSI has broad use cases such as fraud detection and ad targeting whereas PSU can be deployed to combine medical data from multiple hospitals and release aggregate statistics. Both PSI and PSU computation can fit into the proposed technique of partitioning the inputs and applying the intersection or union operation per partition to form the final result. Any aggregate computation on the combined inputs can also be applied per partition without incurring additional computational cost. Since set operations only contain unique keys, there would be no m' leakage (Chapter 6).

4. Inventory Matching: The inventory matching problem [69] securely pairs *buy* and *sell* orders for financial stocks without revealing sensitive client information or market intent to the bank. Our radix partitioning can improve the efficiency of this problem by grouping buy and sell orders based on stock identifiers in a data-oblivious manner. Reducing comparisons to per partition, and to specific hash-buckets within a partition, can be particularly helpful for such latency sensitive applications. The enclave based approach can significantly reduce bandwidth usage compared to secure multi-party computation techniques.

5. Oblivious Distributed Computing: Distributed computing is a popular paradigm where a task is divided across multiple interconnected machines that work together to solve it. In oblivious settings, ensuring equal load per machine is not only important for load balancing but also for security. Along with above discussed specialized cases, our oblivious radix partitioning can be applied to perform distributed computations securely. One specific use case is to extend the proposed two-table join to a multi-way join where each machine recursively computes the join across multiple tables for the partitions allocated to it.

Chapter 9

Related Work

Oblivious data systems using TEEs have received significant attention [90, 28, 24, 72, 57, 46, 5, 17, 53]. Early systems like Opaque [90], ObliDB [28], and [5] relied on TEEs with specialized memory that hides access patterns. In practice, this assumption is not supported by any current hardware. More recent works define *double-* or *fully oblivious* guarantees [59, 74], which protect both memory access and control flow without relying on special hardware — a guarantee also provided by our scheme.

Oblivious algorithms can be built using either generic memory controllers [73] based on ORAM [79] or through problem-specific designs. Maliszewski et al. [55] evaluate secure join algorithms in TEEs and show that generic ORAM-based approaches incur prohibitive overheads. Among the alternatives, KKS [46] stood out as the most efficient in their evaluation, demonstrating that join-specific constructions significantly outperform generic mechanisms.

A critical bottleneck in KKS [46] lies in its repeated use of oblivious sorts, which dominates its runtime complexity. Obliviator [57] improves upon this by reducing the number of sorts from five to two by leveraging the efficient oblivious compaction design by Sasy et al. [74], achieving up to 80× speedup in parallel settings over their sequential KKS* baseline. KKS* is oblivious even when compiled with the `-O3` optimization flag, unlike the original KKS implementation. Recent systems such as SODA [53] and Jodes [84] extend oblivious joins to distributed settings, distributing the work to multiple servers. Compared to these schemes, our approach—by using radix partitioning and a hash-and-probe strategy—is the only one that avoids obliviously sorting a large combined input table. In particular, we never materialize an N-sized table with $N = |R| + |S|$, unlike prior designs.

Other works explore *differentially oblivious* joins [23, 70, 71], which trade off strict obliviousness for formal (ε, δ)-differentially private guarantees [27]. Separately, a rich line of work [51, 20, 21, 25, 1] propose singly oblivious join schemes, which rely on a trusted client-side proxy and treat the server as a simple key-value store. While some schemes [51, 19, 20] introduce efficient join schemes for non-distributed settings, others like SEAL [25] and OasisDB [1] focus on a distributed storage for scalability. These designs incur high WAN communication costs and differ fundamentally in their architecture and threat model. Other systems like SMCQL [12] and Conclave [83] operate in multi-party federated settings and are orthogonal to our enclave-based design. Mohassel et al. [61] also employ hashing techniques to perform joins, albeit over secret-shared input tables. A notable limitation of their approach is the requirement for each table to contain unique primary keys, which constrains the join to a strictly one-to-one correspondence between rows.

Chapter 10

Conclusion

We present parallel and doubly oblivious algorithms for both non-foreign key and foreign key joins executed within hardware enclaves. Our join construction utilizes *radix partitioning* but with oblivious guarantees to divide input tables into smaller partitions and operate only on corresponding partition pairs *in parallel*. The proposed algorithm reduces the size of oblivious sorts compared to state-of-the-art schemes, and it is the *only scheme that avoids obliviously re-sorting the combined inputs for both non–foreign key and foreign key joins*. We implement and evaluate our system on Intel TDX machines, demonstrating significant performance gains over the state-of- the-art scheme, particularly on large datasets. The oblivious radix partitioning is an independent contribution that can be applied to many problems, enabling several promising directions for future work.

References

[1] Haseeb Ahmed, Nachiket Rao, Abdelkarim Kati, Florian Kerschbaum, and Sujayya Maiyya. OasisDB: An oblivious and scalable system for relational data. 2025.

[2] Miklós Ajtai, János Komlós, and Endre Szemerédi. An $O(n \log n)$ sorting network. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing (STOC)*, pages 1–9, New York, NY, USA, 1983. ACM.

[3] AMD SEV-SNP for Amazon EC2 instances. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/sev-snp.html. Accessed Jul 10, 2025.

[4] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, 1967.

[5] Arvind Arasu and Raghav Kaushik. Oblivious query processing. *arXiv preprint arXiv:1312.4012*, 2013.

[6] Gilad Asharov, TH Hubert Chan, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. Bucket oblivious sort: An extremely simple oblivious sort. In *Symposium on Simplicity in Algorithms*, pages 8–14. SIAM, 2020.

[7] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. Optorama: Optimal oblivious ram. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part II (Lecture Notes in Computer Science, Vol. 12106)*, pages 403–432, Heidelberg, 2020. Springer.

[8] Gilad Asharov, Ilan Komargodski, and Yehuda Michelson. Futorama: A concretely efficient hierarchical oblivious ram. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2023.

[9] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M Tamer Özsu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 362–373. IEEE, 2013.

[10] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M Tamer Özsu. Main-memory hash joins on modern processor architectures. *IEEE Transactions on Knowledge and Data Engineering*, 26(7):1754–1766, 2014.

[11] Kenneth E. Batcher. Sorting networks and their applications. In *Proceedings of the Spring Joint Computer Conference (AFIPS)*, pages 307–314. ACM, 1968.

[12] Johes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel N Kho, and Jennie Rogers. Smcql: Secure query processing for private data networks. *Proc. VLDB Endow.*, 10(6):673–684, 2017.

[13] Johes Bater, Xi He, William Ehrich, Ashwin Machanavajjhala, and Jennie Rogers. Shrinkwrap: Efficient sql query processing in differentially private data federations. *Proceedings of the VLDB Endowment*, 12(3):307–320, 2018.

[14] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure:{SGX} cache attacks are practical. In *11th USENIX workshop on offensive technologies (WOOT 17)*, 2017.

[15] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 668–679, 2015.

[16] Meeyoung Cha, Hamed Haddadi, Fabricio Benevenuto, and Krishna P. Gummadi. Measuring user influence in twitter: The million follower fallacy. In *Proceedings of the International AAAI Conference on Web and Social Media (ICWSM)*, 2010.

[17] Javad Ghareh Chamani, Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Rasool Jalili. Graphos: Towards oblivious graph processing. *Proceedings of the VLDB Endowment*, 16(13):4324–4338, 2023.

[18] T-H. Hubert Chan, Yue Guo, Wei-Kai Lin, and Elaine Shi. Oblivious hashing revisited, and applications to asymptotically efficient oram and opram. In *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security*, pages 660–690, Hong Kong, China, 2017. Springer International Publishing.

[19] Zhao Chang, Dong Xie, Feifei Li, Jeff M Phillips, and Rajeev Balasubramonian. Efficient oblivious query processing for range and knn queries. *IEEE Transactions on Knowledge and Data Engineering*, 34(12):5741–5754, 2021.

[20] Zhao Chang, Dong Xie, Sheng Wang, and Feifei Li. Towards practical oblivious join. In *Proceedings of the 2022 International Conference on Management of Data*, pages 803–817, 2022.

[21] Zhao Chang, Dong Xie, Sheng Wang, Feifei Li, and Yulong Shen. Towards practical oblivious join processing. *IEEE Transactions on Knowledge and Data Engineering*, 36(4):1829–1842, 2023.

[22] Mohammad Jabed Morshed Chowdhury, Md Sadek Ferdous, Kamanashis Biswas, Niaz Chowdhury, and Vallipuram Muthukkumarasamy. Covid-19 contact tracing: challenges and future directions. *Ieee Access*, 8:225703–225729, 2020.

[23] Shumo Chu, Danyang Zhuo, Elaine Shi, and TH Hubert Chan. Differentially oblivious database joins: Overcoming the worst-case curse of fully oblivious algorithms. *Cryptology ePrint Archive*, 2021.

[24] Emma Dauterman, Vivian Fang, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. Snoopy: Surpassing the scalability bottleneck of oblivious storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*, pages 655–671, 2021.

[25] Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Saurabh Shintre. Seal: Attack mitigation for encrypted databases via adjustable leakage. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2433–2450, 2020.

[26] Sam Dittmer and Rafail Ostrovsky. Oblivious tight compaction in o (n) time with smaller constant. In *International Conference on Security and Cryptography for Networks*, pages 253–274. Springer, 2020.

[27] Cynthia Dwork, Aaron Roth, et al. The algorithmic foundations of differential privacy. *Foundations and Trends® in Theoretical Computer Science*, 9(3–4):211–407, 2014.

[28] Saba Eskandarian and Matei Zaharia. Oblidb: Oblivious query processing for secure databases. *Proc. VLDB Endow.*, 13(2):169–183, 2019.

[29] Ken Goldberg, Theresa Roeder, Dhruv Gupta, and Chris Perkins. Eigentaste: A constant time collaborative filtering algorithm. In *Information Retrieval: Data Structures and Algorithms*. Springer, 2001.

[30] Michael T Goodrich. Data-oblivious external-memory algorithms for the compaction, selection, and sorting of outsourced data. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 379–388, 2011.

[31] Michael T. Goodrich. Zig-zag sort: A simple deterministic data-oblivious sorting algorithm running in $O(n \log n)$ time. In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing (STOC)*, pages 684–693. ACM, 2014.

[32] Google Cloud Confidential Computing. https://cloud.google.com/security/products/confidential-computing. Accessed Jul 10, 2025.

[33] Paul Grubbs, Anurag Khandelwal, Marie-Sarah Lacharité, Lloyd Brown, Lucy Li, Rachit Agarwal, and Thomas Ristenpart. Pancake: Frequency smoothing for encrypted data stores. In *29th USENIX Security Symposium*, pages 2451–2468, 2020.

[34] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. Pump up the volume: Practical database reconstruction from volume leakage on range queries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 315–331, 2018.

[35] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. Learning to reconstruct: Statistical learning theory and encrypted database attacks. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1067–1083, 2019.

[36] Tianyao Gu, Yilei Wang, Bingnan Chen, Afonso Tinoco, Elaine Shi, and Ke Yi. Efficient oblivious sorting and shuffling for hardware enclaves. *sort*, 23:35, 2023.

[37] Tianyao Gu, Yilei Wang, Afonso Tinoco, Bingnan Chen, Ke Yi, and Elaine Shi. Flexway {O-Sort}:{Enclave-Friendly} and optimal oblivious sorting. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 7563–7582, 2025.

[38] Zichen Gui, Oliver Johnson, and Bogdan Warinschi. Encrypted databases: New volume attacks against range queries. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 361–378, 2019.

[39] Marcus Hähnel, Weidong Cui, and Marcus Peinado. {High-Resolution} side channels for untrusted operating systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 299–312, 2017.

[40] 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*, 2014.

[41] Alexandra Henzinger, Emma Dauterman, Henry Corrigan-Gibbs, and Nickolai Zeldovich. Private web search with tiptoe. In *Proceedings of the 29th symposium on operating systems principles*, pages 396–416, 2023.

[42] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using innovative instructions to create trustworthy software solutions. *HASP@ ISCA*, 11(10.1145):2487726–2488370, 2013.

[43] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: ramification, attack and mitigation. In *19th Annual Network and Distributed System Security Symposium (NDSS)*, 2012.

[44] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'neill. Generic attacks on secure outsourced databases. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1329–1340, 2016.

[45] Evgenios M Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. Data recovery on encrypted databases with k-nearest neighbor query leakage. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1033–1050. IEEE, 2019.

[46] Simeon Krastnikov, Florian Kerschbaum, and Douglas Stebila. Efficient oblivious database joins. *Proceedings of the VLDB Endowment*, 13(11):2051–2064, 2020.

[47] Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. Improved reconstruction attacks on encrypted data using range query leakage. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 297–314. IEEE, 2018.

[48] Dayeol Lee, Dongha Jung, Ian T Fang, Chia-Che Tsai, and Raluca Ada Popa. An {Off-Chip} attack on hardware enclaves via the memory bus. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[49] 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 *26th USENIX Security Symposium (USENIX Security 17)*, pages 557–574, 2017.

[50] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. Signed networks in social media. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI)*, pages 1361–1370. ACM, 2010.

[51] Rui Li, Alex X Liu, Ann L Wang, and Bezawada Bruhadeshwar. Fast range query processing with strong privacy protection for cloud computing. *Proceedings of the VLDB Endowment*, 7(14):1953–1964, 2014.

[52] Shuyuan Li, Yuxiang Zeng, Yuxiang Wang, Yiman Zhong, Zimu Zhou, and Yongxin Tong. An experimental study on federated equi-joins. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2024.

[53] Xiang Li, Nuozhou Sun, Yunqian Luo, and Mingyu Gao. Soda: A set of fast oblivious algorithms in distributed secure data analytics. *Proceedings of the VLDB Endowment*, 16(7):1671–1684, 2023.

[54] Sujaya Maiyya, Sharath Chandra Vemula, Divyakant Agrawal, Amr El Abbadi, and Florian Kerschbaum. Waffle: An online oblivious datastore for protecting data access patterns. *Proceedings of the ACM on Management of Data (SIGMOD)*, 1(4):1–25, 2023.

[55] Kajetan Maliszewski, Jorge-Arnulfo Quiané-Ruiz, Jonas Traub, and Volker Markl. What is the price for joining securely? benchmarking equi-joins in trusted execution environments. *Proceedings of the VLDB Endowment*, 15(3):659–672, November 2021.

[56] Stefan Manegold, Peter Boncz, and Martin Kersten. Optimizing main-memory join on modern hardware. *IEEE Transactions on Knowledge and Data Engineering*, 14(4):709–730, July 2002.

[57] Apostolos Mavrogiannakis, Xian Wang, Ioannis Demertzis, Dimitrios Papadopoulos, and Minos Garofalakis. Obliviator: Oblivious parallel joins and other operators in shared memory environments. *Cryptology ePrint Archive*, 2025.

[58] Microsoft Azure Confidential Computing. https://learn.microsoft.com/en-us/azure/confidential-computing/. Accessed Jul 10, 2025.

[59] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An efficient oblivious search index. In *2018 IEEE symposium on security and privacy (SP)*, pages 279–296. IEEE, 2018.

[60] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How sgx amplifies the power of cache attacks. In *Cryptographic Hardware and Embedded Systems–CHES 2017: 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, pages 69–90. Springer, 2017.

[61] Payman Mohassel, Peter Rindal, and Mike Rosulek. Fast database joins and psi for secret shared data. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1271–1287, New York, NY, USA, 2020. Association for Computing Machinery.

[62] Nicholas Ngai, Ioannis Demertzis, Javad Ghareh Chamani, and Dimitrios Papadopoulos. Distributed & scalable oblivious sorting and shuffling. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 4277–4295. IEEE, 2024.

[63] MKTME Side Channel Impact on Intel TDX. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/mktme-side-channel-impact-on-intel-tdx.html. Accessed June 16, 2025.

[64] Oracle. Joins — sql tuning guide, oracle database 21c. urlhttps://docs.oracle.com/en/database/oracle/oracle-database/21/tgsql/joins.html#GUID-5FCD34FE-ED04-4AB2-BC90-9752FED94F4F. Accessed: 2025-12-01.

[65] Simon Oya and Florian Kerschbaum. Hiding the access pattern is not enough: Exploiting search pattern leakage in searchable encryption. In *USENIX Security Symposium*, pages 127–142, 2021.

[66] Simon Oya and Florian Kerschbaum. IHOP: Improved statistical query recovery against searchable symmetric encryption through quadratic optimization. In *31st USENIX Security Symposium*, pages 2407–2424, 2022.

[67] Sarvar Patel, Giuseppe Persiano, Mariana Raykova, and Kevin Yeo. PanORAMa: Oblivious RAM with logarithmic overhead. In *IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 871–882, 2018.

[68] Rishabh Poddar, Stephanie Wang, Jianan Lu, and Raluca Ada Popa. Practical volume-based attacks on encrypted databases. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 354–369. IEEE, 2020.

[69] Antigoni Polychroniadou, Gilad Asharov, Benjamin Diamond, Tucker Balch, Hans Buehler, Richard Hua, Suwen Gu, Greg Gimler, and Manuela Veloso. Prime match:

A {Privacy-Preserving} inventory matching system. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6417–6434, 2023.

[70] Lianke Qin, Rajesh Jayaram, Elaine Shi, Zhao Song, Danyang Zhuo, and Shumo Chu. Adore: Differentially oblivious relational database operators. *Proceedings of the VLDB Endowment*, 16(4):842–855, 2022.

[71] Lina Qiu, Georgios Kellaris, Nikos Mamoulis, Kobbi Nissim, and George Kollios. Doquet: Differentially oblivious range and join queries with private data structures. *Proceedings of the VLDB Endowment*, 16(13):4160–4173, 2023.

[72] Leonie Reichert, Gowri R Chandran, Phillipp Schoppmann, Thomas Schneider, and Björn Scheuermann. Menhir: An oblivious database with protection against access and volume pattern leakage. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, ASIA CCS '24, page 1675–1690, New York, NY, USA, 2024. Association for Computing Machinery.

[73] Sajin Sasy, Sergey Gorbunov, and Christopher W Fletcher. Zerotrace: Oblivious memory primitives from intel sgx. *Cryptology ePrint Archive*, 2017.

[74] Sajin Sasy, Aaron Johnson, and Ian Goldberg. Fast fully oblivious compaction and shuffling. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2565–2579, 2022.

[75] Sajin Sasy, Aaron Johnson, and Ian Goldberg. Waks-on/waks-off: Fast oblivious offline/online shuffling and sorting with waksman networks. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 3328–3342, 2023.

[76] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using sgx to conceal cache attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Proceedings 14*, pages 3–24. Springer, 2017.

[77] AMD Sev-Snp. Strengthening vm isolation with integrity protection and more. *White Paper, January*, 53(2020):1450–1465, 2020.

[78] Signal's Private Contact Discovery. https://signal.org/blog/building-faster-oram/. Accessed Jul 10, 2025.

[79] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 299–310, 2013.

[80] Intel TDX. https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/overview.html. Accessed March 10, 2025.

[81] TPC-H Descriton. https://www.tpc.org/tpch/. Accessed Jul 10, 2025.

[82] 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 *26th USENIX Security Symposium (USENIX Security 17)*, pages 1041–1056, 2017.

[83] 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*, pages 1–18, 2019.

[84] Yilei Wang, Xiangdong Zeng, Sheng Wang, and Feifei Li. Jodes: Efficient oblivious join in the distributed setting. *arXiv preprint arXiv:2501.09334*, 2025.

[85] Ruidi Wei. Oblivious multi-way band joins: An efficient algorithm for secure range queries. Master's thesis, University of Waterloo, 2025.

[86] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*, pages 640–656. IEEE, 2015.

[87] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. In *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics (MDS)*, 2012.

[88] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *USENIX Security Symposium*, volume 2016, pages 707–720, 2016.

[89] Leqian Zheng, Zheng Zhang, Wentao Dong, Yao Zhang, Ye Wu, and Cong Wang. H_2 o_2 ram: A high-performance hierarchical doubly oblivious ram. *arXiv preprint arXiv:2409.07167*, 2024.

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