DISCO*: DIstributed and SCalable Oblivious Joins and Oblivious Primitives

Apostolos Mavrogiannakis
UCSC
USA

Xian Wang
HKUST
Hong Kong

Ioannis Demertzis
UCSC
USA

Dimitrios Papadopoulos
HKUST
Hong Kong

Minos Garofalakis
ATHENA Research Center &
Technical University of Crete
Greece

Abstract

Trusted Execution Environments (TEEs) let clients outsource computation to untrusted clouds, but they remain vulnerable to side-channel and leakage-abuse attacks. Recent work closes this gap by combining hardware enclaves with oblivious computation, which offers strong security guarantees yet struggles to meet the performance and scalability demands of real-world deployments. In this work, we present DISCO, the first *scalable, distributed, and fully oblivious* database system that seamlessly combines hardware enclaves with cutting-edge oblivious primitives and novel optimizations that significantly reduce inter-server communication overheads. What distinguishes DISCO from prior work is a suite of distributed frameworks that unlock *parallel computation* and *fundamentally shift the design paradigm* of distributed oblivious systems, enabling practical deployment at scales that were unattainable with previous solutions. We provide a thorough evaluation of the performance of our system on both synthetic and real-world datasets. Our evaluation demonstrates that our design significantly reduces the gap between theory and practice, achieving 88× speedup over Jodes and 180× over SODA on non-foreign key joins with $N = 2^{25}$ elements. Most importantly, we report, for the *first* time, results for computations at the *terabyte scale*.

CCS Concepts: • **Security and privacy → Management and querying of encrypted data**.

Keywords: Oblivious computation, Trusted Execution Environments, Distributed Databases

ACM Reference Format:
Apostolos Mavrogiannakis, Xian Wang, Ioannis Demertzis, Dimitrios Papadopoulos, and Minos Garofalakis. 2026. DISCO*: DIstributed and SCalable Oblivious Joins and Oblivious Primitives.



This work is licensed under a Creative Commons Attribution 4.0 International License.
SOSP '26, Prague, Czech Republic
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2585-2/2026/09
https://doi.org/10.1145/3830418.3843862

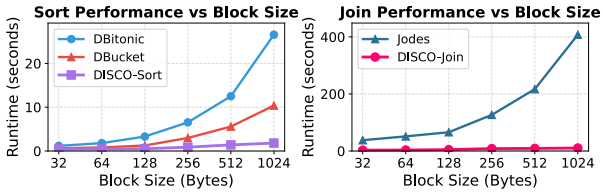

Figure 1. Block-size experiment with $N = 10^7$, 16 threads, 16 servers against Jodes [98] and sort algorithms [77].

In *ACM SIGOPS 32nd Symposium on Operating Systems Principles (SOSP '26), September 29-October 02, 2026, Prague, Czech Republic.* ACM, New York, NY, USA, 26 pages. https://doi.org/10.1145/3830418.3843862

1 Introduction

Trusted Execution Environments (TEEs)—hardware-isolated execution environments where code and data are encrypted and protected from the host OS and hypervisor—enable outsourcing sensitive computation to untrusted cloud infrastructure while preserving data confidentiality and integrity. TEEs have become a cornerstone of modern privacy-preserving systems—from Signal's Private Contact Discovery [72, 88], which enables secure contact matching without exposing users' contact lists, to searchable encryption, privacy-preserving semantic search, secure retrieval-augmented generation (RAG) [104], anonymous communication [36], and private relational databases. Industry giants have taken notice: e.g., Google Privacy Sandbox, FLEDGE, Google's Key Transparency [52], Titan Security Key [12], and Asylo [41] all build on these foundations.

However, TEEs alone do not protect against adversaries who can observe memory access patterns, execution traces, and communication behavior during computation. In particular, they do not protect against leakage-abuse or access-pattern attacks [4, 14, 24, 37, 44–46, 49, 53, 54, 57–60, 63, 69, 71, 79, 101], nor against software side-channel attacks [18, 42, 56, 64, 68, 81, 92, 97]. To close this gap, prior works [62, 66, 73, 77, 98, 102] combine TEEs with *doubly-oblivious computation*: a strong cryptographic notion requiring that an algorithm's memory access patterns, control flow, and communication are entirely independent of the input data. Under

obliviousness, even an adversary with full visibility learns nothing beyond public parameters such as the input size. This property, when realized within TEE-based systems, enables secure cloud outsourcing that safeguards data confidentiality and integrity against both software side-channel and leakage-abuse attacks.

However, such combination introduces many performance limitations. In particular, on a single server, performance inevitably plateaus. In SGX environments [62, 73], performance collapses once data exceeds the Enclave Page Cache (EPC) capacity. Even with modern Confidential Virtual Machines (CVMs) –such as Intel TDX, AMD SEV– which remove the EPC bottleneck by encrypting entire VMs and eliminating explicit memory restrictions, scalability remains limited: additional cores cease to yield gains, memory bandwidth saturates, and the computational cost of oblivious algorithms on large datasets simply overwhelms any one node.

To address these limitations, recent work has explored distributed architectures [66, 77, 98, 102]. However, this shift introduces new challenges. In the oblivious setting, even *communication patterns* can leak sensitive information, meaning that both computation *and* communication must be data-independent. As a result, existing distributed solutions rely on multiple rounds of expensive oblivious sorting, sequential execution steps, and heavy inter-server coordination, introducing performance degradation at scale (see Section 3). More critically, a fundamental bottleneck has been largely overlooked. All prior distributed oblivious systems [62, 66, 66, 73, 77, 98, 102] adopt *horizontal partitioning* (see Section 4), a design choice that creates two key performance barriers. First, large block sizes—common in real-world databases with composite keys and multi-attribute records—cause performance degradation, since every oblivious swap must move entire tuples (see Figure 1). Second, horizontal partitioning requires servers to exchange data during computation, causing communication costs to grow rapidly with block sizes and dataset scale. Together, these bottlenecks have confined prior oblivious systems to small-scale deployments, with the largest reported evaluation reaching only 140GB.

In this work, we focus on *oblivious relational database systems*, where obliviousness must be preserved during query processing. In detail, we make a simple but key observation: *vertical partitioning*—where each server holds a shard of every tuple rather than a subset of complete tuples (see Figure 2)—has never been explored in the oblivious setting, yet it offers profound advantages. Under vertical partitioning, every server already possesses a shard of every record, enabling fully parallel execution with minimal inter-server communication during computation. Moreover, each server operates on smaller shards rather than full-sized tuples, directly *neutralizing* the block-size bottleneck. For small tuples, where the benefits of vertical partitioning diminish, we introduce a *hybrid* strategy that combines vertical and horizontal partitioning, adapting the partitioning granularity to each algorithm's characteristics.

Building on these frameworks, we present DISCO, the first scalable, distributed, and fully oblivious database system. DISCO guarantees parallelization at every step of computation, scales to *terabytes* of data, and minimizes communication overhead. In detail, DISCO proposes:

1. New Distributed Framework. We are the *first* to leverage hybrid and vertical partitioning in the oblivious setting. This enables fully-parallel execution across servers, eliminates inter-server communication during computation, and allows each server to operate on smaller shards rather than full-sized tuples, directly solving both performance bottlenecks that limit all prior approaches.

2. Distributed Oblivious Primitives. We design a complete suite of distributed oblivious primitives that serve as the core building blocks of our system. Our distributed compaction and expansion are the *first* to achieve optimal/single-round communication. We design an optimized distributed sort that synthesizes techniques from [48, 77] and a distributed aggregation tree with constant-size communication per round.

3. Distributed Oblivious Query Operators. Combining these primitives with optimization techniques, we build operators for filtering, aggregation, and both foreign-key and non-foreign-key joins that outperform all existing approaches.

4. Oblivious Query Optimizer. To efficiently execute complex multi-operator queries, we design the *first* oblivious query optimizer for distributed TEE-based systems. The optimizer transforms logical query plans into efficient physical plans through a cost model over public parameters and seven deterministic rewrite rules that eliminate redundant primitives and reduce communication while preserving obliviousness, achieving significant speedups.

5. Scalability & Evaluation. We are the *first* to achieve near-linear latency growth with block size and to break the **terabyte** barrier in oblivious computation, scaling to datasets as large as **16TB**, a two-orders-of-magnitude increase over the largest prior evaluation of 140GB. On $N = 2^{25}$ elements, our non-foreign key join is 88× faster than Jodes [98] and 180× faster than SODA [66], while our expansion is 38×–129× faster than Jodes [98]. On complex queries from TPC-H and the Big Data Benchmark, our system achieves speedups of 17×–48× over Opaque, which our optimizer further improves by 5.1×–1,027×. Our implementation is available online[1].

Capabilities & Limitations. DISCO is a low-level oblivious database engine that operates directly on physical query plans, exposing no developer-facing SQL interface. It supports queries composed of joins, filters, and aggregations, while applying the optimization rules described in Section 7. A relational or dataflow front-end that compiles SQL into DISCO's physical plans is orthogonal to our contributions and left to future work. DISCO targets OLAP workloads over

[1]https://doi.org/10.5281/zenodo.22214076

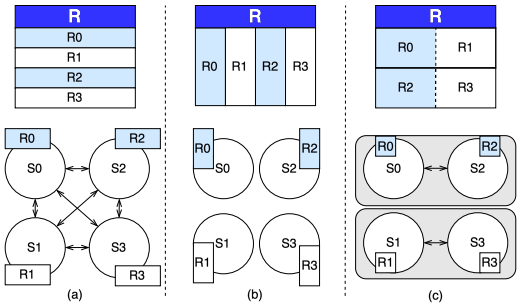

Figure 2. Partitioning Approaches. (a) shows horizontal partitioning, where each server holds an equal portion of the tuples. (b) shows vertical partitioning, where each server stores a shard of every tuple. (c) shows hybrid partitioning, where each server holds a subset of shards from all tuples.

static databases; consistent with prior work, oblivious database maintenance (e.g., inserts, updates, and deletions) is outside the scope of this work. Finally, DISCO focuses on equi-join workloads and not on band/range joins.

2 Preliminaries

Notation. Before presenting the preliminaries, we first introduce the notation used throughout the paper. Our system operates in a *shared-nothing* architecture, where each server has its own private memory and storage; servers communicate by exchanging messages over the network. We denote the number of servers by E, with server k referred to as E_k. The total number of tuples is denoted by N, and the block size of each tuple in bytes by B. We further denote the payload size by $B_p = B - |k|$, where $|k|$ is the size of the computation key (see Section 4). Each server holds a local array of size n. We use R to denote the table that is accessed obliviously, r_i to refer to its i-th tuple, and $s_{i,j}$ to represent the j-th shard of r_i. For instance, $s_{0,2}$ is the third shard of the first tuple. The total number of clusters is denoted by C, and the number of servers per cluster by G. Finally, we define $[x] = \{0, \ldots, x-1\}$.

Hardware Enclave. Similar to prior systems (Opaque [103], SODA [66], and Jodes [98]), this work focuses on oblivious algorithms and operators designed for TEEs. Our techniques are hardware-agnostic and applicable across a range of TEEs, including ARM TrustZone [8] and AMD SEV [22], which provide secure key management, sealing, and remote attestation. TEEs protect code and data through encryption and access control, ensuring that only authorized code can interact with sensitive data. The isolated memory for user's data and code to be executed is called EPC. SGXv1 had a limited 128 MB EPC that introduced paging overhead [34]. Subsequent SGXv2 supports EPC sizes of up to 512GB.

Obliviousness in shared-nothing environments. In the distributed setting, an algorithm is said to be oblivious *iff*, for any two executions with different inputs of the same size, all memory access patterns and communication between servers are data-independent and indistinguishable. Oblix [74] strengthened this to *doubly-oblivious*, which additionally requires that each server's execution traces are data-independent and indistinguishable as well. In our paper, we adopt the stronger, doubly-oblivious definition and refer to it simply as obliviousness. Previous works (e.g., [77, 98, 103]) operate under the same security guarantees in TEEs.

To formally prove that an algorithm is oblivious, a Simulator *Sim* must be introduced that copies the flow of the desired algorithm. *Sim* takes as input any information that can be observed by the adversary (also called leakage $\mathcal{L}$– see below), and based on this information simulates identical communication, memory accesses and execution traces. Then, given two games: a **Real** and an **Ideal**$_{\mathcal{A},\mathcal{L},Sim}$, an adversary must not be able to distinguish between the two games with more than negligible probability. We defer the formal definition and the corresponding real/ideal experiment to Appendix C.1.

Oblivious Compaction. Given an array of elements R and an auxiliary array D that marks selected entries (i.e., $D[i] = 1$), oblivious compaction reorders R so that all marked elements are moved to the front, while obscuring any access patterns that could be observed by the adversary. Sasy et al. [82] proposed ORCompact, the current state-of-the-art compaction algorithm in the shared-memory setting. ORCompact recursively brings marked elements to the front in an oblivious fashion and achieves an asymptotic complexity of $O(N \log N)$. Although linear-time compaction algorithms have been developed [33, 40], they are often impractical in real-world scenarios due to large hidden constants. A notable advantage of ORCompact is its stability: it preserves the relative order of marked elements. Specifically, if two marked elements i and j appear in order ($i < j$) before compaction, their positions after compaction (i' and j') will still satisfy $i' < j'$. Even though ORCompact is efficient in the single-server setting, it is not in the distributed setting, since it incurs $O(\log E)$ rounds of communication.

Oblivious Sorting. Oblivious sorting outputs a sorted array based on a computation key while hiding access patterns. In databases, this corresponds to reordering a table by one or more columns. In single-server, multi-threaded settings, bitonic sort is the most practical choice despite its $O(N \log^2 N)$ complexity [77]. Bucket sort offers a better $O(N \log N)$ complexity by first applying an oblivious shuffle to obfuscate access patterns, followed by a standard non-oblivious sort. Although the final step is non-oblivious, the shuffle ensures overall obliviousness [9]. Crucially, elements are routed into buckets by *random tags* rather than by their sort keys, so bucket occupancy is independent of the key distribution: even if every key is identical, the assignment stays uniform and oblivious [9, 77]. A small number of dummy

elements per bucket bounds the overflow probability to a negligible 2^{-60}, so records are never dropped. In practice, however, bucket sort is often slower than bitonic sort in single-server environments [77]. Intuitively, bitonic sort wins at single-server (and small, up to ∼4-server) scales because its fixed compare-exchange network is highly parallelizable and cache-friendly—regular, strided memory access—which outweighs its worse asymptotic complexity, whereas bucket sort pays for an oblivious shuffle with random access and dummy overhead. However, on the distributed setting, bucket sort is more efficient as it requires a single round of linear communication $O(N \cdot B)$ compared to bitonic's $O(\log E)$ rounds of $O(N \cdot B)$ communication. On SGX v1, limited EPC size causes frequent paging and performance bottlenecks. Gu et al. [48] optimize bucket sort with their MergeSplit algorithm, which routes elements across k buckets simultaneously and removes the power-of-two input constraint. They further reduce costs with a cost function that computes the minimum required number of dummies, an adaptive bucket selection strategy, and improved paging between the TEE and untrusted memory. In distributed systems, column-wise sorting as in Opaque [103] and Jodes [98] incurs high computation and communication overhead [77]. Consequently, bucket sort remains the most effective approach—both in theory and practice—for distributed oblivious sorting.

Oblivious Shuffling. Oblivious shuffling randomly permutes an array such that adversaries cannot correlate output positions with input positions. Oblivious shuffling can be realized by assigning random keys and performing an oblivious sort [77] with $O(N \log N)$ time complexity. Other works [66, 98] propose shuffling variants, but these distribute elements by attributes rather than permuting at random. That is, instead of applying a uniformly random permutation, they route each element to a server determined by a key or attribute (e.g., a hash of the join key) so that related elements are grouped together. To do this obliviously, the number of elements sent to each destination is padded with dummies to a public upper bound and the elements are moved via oblivious sorting/distribution, so the per-server communication volume is data-independent. This is useful for co-locating matching tuples on the same server, but it is not a random shuffle, and the padding bound can itself leak information—e.g., SODA reveals the maximum per-server cardinality (see Threat Model & Leakage).

Threat Model & Leakage. Following prior works [77, 102], we assume a powerful adversary that can simultaneously observe all execution traces, memory access patterns, and inter-server communication across the entire system, as well as the database schema. Additionally, the adversary can compromise the operating system, but cannot tamper with the processor. We do not consider hardware side-channel attacks [13, 42, 50, 84, 97]; existing defenses [19, 20, 47, 85, 86] are complementary. Unlike SODA [66], our threat model prevents leakage of the maximum cardinality of tuples.

As in prior work, we assume that the input size is public information and allow leakage of the final output size. Queries composed exclusively of filters, aggregations, and foreign-key joins reveal no intermediate result sizes, as these operators can be made size-preserving by avoiding truncation of their outputs. Consequently, for any query containing only these operators, DISCO propagates untruncated intermediate results so that an adversary observes only the public input size and the final output size. In contrast, when a query includes a non-foreign-key join, the cardinality of that join's output is necessarily revealed. Since the output of a non-foreign-key join may be as large as $O(N^2)$, padding to the worst case is impractical; therefore, revealing its output size M is both inherent to the operator and consistent with prior work. Overall, DISCO leaks intermediate result sizes only for non-foreign-key joins. Finally, unlike SODA [66], which additionally leaks the maximum per-server join degree to hide its communication pattern, DISCO reveals no such information.

3 Related Work

Below, we review prior approaches and their limitations, starting with oblivious primitives and then, operators.

Oblivious TEE-based Primitives. Sasy et al. [82] propose a parallel algorithm for oblivious compaction, which reorganizes data by removing invalid or empty entries without leaking their positions. Sasy et al. [83] develop a shuffling method that shifts most computation to an offline phase. Oblivious sorting has also progressed significantly. Ngai et al. [77] enhance two scalable sorting techniques: bitonic sort, which performs well in shared-memory settings, and bucket sort, which is better suited to distributed environments. Gu et al. [48] further improve bucket sort to handle high page-swapping overheads (such as those in Intel SGXv1) while improving efficiency. Moving beyond basic primitives, Krastnikov et al. [62] introduce a sequential approach to oblivious expansion, a key operation in database joins.

Despite progress, two challenges persist: (1) most primitives target shared-memory systems, so performance collapses once datasets exceed EPC capacity, underlining the need for scalable distributed solutions; and (2) computation on large block sizes (≥ 1 KB) incurs heavy tuple-level overhead from oblivious operations, limiting parallelism gains.

Oblivious TEE-based DB/Joins. Numerous studies build upon the previously mentioned primitives to construct oblivious database systems. Opaque [102] pioneered this direction by presenting the first distributed oblivious system capable of executing filtering, aggregation, and foreign-key joins. However, it relies on sequential execution, does not support non-foreign-key joins, and requires multiple sorting steps for query processing. Krastnikov et al. [62] introduced the first efficient support for non-foreign-key joins in an oblivious setting, but their solution remains constrained to sequential

algorithms within a shared-memory model. Chang et al. [17] proposed a join technique using oblivious indexes (ORAMs), yet ORAMs inherently suffer from poor scalability due to their sequential nature. Mavrogiannakis et al. [73] introduced Obliviator, a system that minimizes expensive sorting to just two rounds and scales up to 140GB. Yet, Obliviator is limited to shared-memory systems, which constrains scalability due to EPC restrictions and the high cost of paging [34]. On the theoretical side, Hu et al. [51] addressed cyclic joins using oblivious worst-case optimal join algorithms.

SODA [66] and Jodes [98] are the most closely related distributed systems. SODA proposed distributed oblivious algorithms for database operations, yet suffers from poor scalability due to multiple communication rounds, sequential computation, and repeated sorting. Additionally, it assumes a weaker adversarial model that leaks the maximum cardinalities of both tables in a server. Nonetheless, even with additional leakage, our approach outperforms SODA by 180× on non-foreign key joins. Jodes improves upon SODA but remains bottlenecked by three core issues: (1) excessive sorting: nine executions of local oblivious sorting, plus two executions of global sorting; (2) inherently sequential primitives: a non-parallel execution of ODistribute (an expansion-like primitive [62]) operating on output size M (up to $O(N^2)$), multiple sequential scans, and coordination rounds that pause all worker servers; and (3) heavy communication: seven rounds of input communication N and two rounds of output communication M. Despite these improvements over SODA, our approach still outperforms Jodes by up to 88× on non-foreign key joins. Critically, despite being distributed, both SODA and Jodes underperform compared to the shared-memory Obliviator [73], failing to exploit the scalability of distributed settings.

4 Data/Compute Partitioning

In this section, we describe three partitioning strategies for distributed oblivious computation: horizontal, vertical, and hybrid, and their trade-offs. We assume that data is initially distributed evenly across servers, with each server holding a share of size $O(\frac{N \cdot B}{E})$. When this assumption does not hold, a preprocessing step is introduced, in which a designated coordinator distributes the data according to the chosen data layout. This step incurs a one-time linear communication cost of $O(N \cdot B)$, and is independent of the partitioning strategy. This communication relies on public information: the database schema and the partitioning strategy; thus, reveals no information to the adversary.

Data Layout. To illustrate the data layout, consider a global table R of N tuples, each of size B bytes, to be sorted on column A. We refer to the values of A as the computation key k_A and the remaining columns as the payload of size $B_p = B - |k_A|$. In horizontal partitioning, each server holds $n = N/E$ tuples of size B (see Figure 2(a)). In vertical partitioning, tuples are decomposed into shards, and each server stores N shards, each of size $\frac{B_p}{E} + k_A$, where B_p denotes the payload size (see Figure 2(b)). The computation key k_A is replicated across all shards to enable fully local execution of primitives such as sorting. Finally, hybrid partitioning combines both approaches to enable cluster-level computation. Servers are organized into C clusters, and the input table is first split into C vertical shards, assigning one shard per cluster. Within each cluster, the assigned shard is further horizontally partitioned across its servers so that each holds an equal share of tuples. For instance, in Figure 2(c), the first cluster (top), consisting of servers S_1 and S_3, receives the first (blue) shard, which is then evenly split between them. Computation proceeds within clusters, where servers may communicate depending on the underlying primitive.

Network Overhead. Prior distributed oblivious systems rely exclusively on horizontal partitioning, which requires inter-server communication during execution. For instance, distributed bitonic sort incurs $O(\log E)$ rounds of $O(N \cdot B)$ communication, as comparison-swap operations frequently cross server boundaries—reaching tens of gigabytes in practice [77]. Bucket sort reduces this to a single round of $O(N \cdot B)$ communication but still transfers full-sized tuples. Vertical partitioning eliminates communication during the computation phase: since each server holds a shard of every tuple, primitives such as sorting, compaction, and expansion execute locally without requiring coordination among servers. Communication is confined to two deterministic exchange phases: one before and one after local computation, whose patterns depend only on N, E, and B, preserving obliviousness (see Section 5.1).

Impact of Block Size. Oblivious algorithms rely on constant-time conditional operations that process data in register-sized increments, regardless of whether a swap is performed. For a tuple of size B, each oblivious swap requires $\lceil B/r \rceil$ register operations, where r is the register width. When B is large, this results in proportionally more operations per swap and increased cache-line accesses—swapping two 1 KB tuples touches 16 cache lines per tuple (assuming 64-byte lines), compared to a single cache line for 32-byte tuples. Under vertical partitioning, each server operates on shards of size $B_p/E + |k_A|$ rather than full B-byte tuples, directly reducing the per-swap cost and improving cache utilization.

However, when the payload is small, vertical partitioning offers diminishing returns. In practice, memory read operations are aligned to fixed granularities (e.g., 8 bytes), so each shard is padded accordingly. As a result, splitting a 24-byte payload across more than three servers does not reduce the effective shard size. Beyond this point, further partitioning fails to reduce per-swap cost and only increases overhead from replicating the computation key. This inefficiency also

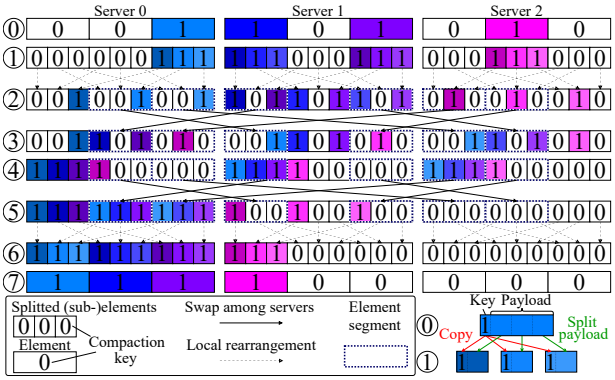

Figure 3. Distributed oblivious compaction illustrated in a vertically partitioned model.

exists when the computation key k_A is larger than the payload. For example, if k_A is 1024 B and the payload is only 16 B, replicating k_A across multiple servers quickly outweighs any benefit from further partitioning.

5 Distributed Primitives

Every oblivious database operator (filter, join, and aggregation) decomposes into a set of primitives: compaction, expansion, aggregation tree, and sort. Because each operator invokes one or more of these primitives, their efficiency directly determines end-to-end query performance: an improvement in a primitive can impact significantly an operator itself. By matching the partitioning strategies to each primitive's computation key, we obtain designs that enhance end-to-end performance.

5.1 Compaction and Expansion

Compaction and expansion are data-movement primitives that underpin the majority of our database operators. Compaction takes a table R and a marking array D and produces an output in which all marked tuples are moved to the front, preserving their original order; it is invoked in every filter and join to remove non-matching or dummy tuples. Expansion takes a table R of size N and an auxiliary array D that specifies, for each tuple $t_i \in R$, the number of copies $D[i]$ to produce, and outputs a table of size $M = \sum_i D[i]$. The algorithm first computes the target index $Q[i] = \sum_{j<i} D[j]$ for each tuple via a prefix sum, then pads the table to size M with dummy entries and obliviously pushes each real tuple t_i to position $Q[i]$—placing $D[i]-1$ dummy rows after it. Then, through a scan each dummy slot is filled with a copy of the preceding real tuple, producing the final expanded output. For example, given tuples (t_1, t_2, t_3) with $D = (2, 3, 1)$, the prefix sum yields $Q = (0, 2, 5)$; after expansion and copying, the output is $(t_1, t_1, t_2, t_2, t_2, t_3)$. Expansion is critical for non-foreign key joins, where tuples are replicated by their join cardinalities.

Prior approaches and their limitations. No prior distributed system implements either primitive efficiently, as their algorithms are communication-heavy, have sequential tasks, or converge to an unnecessary execution of an oblivious sort (the most expensive oblivious primitive). As there are no prior works that introduce compaction in the distributed setting, a naive approach would be to reduce compaction to a full oblivious distributed sort [77], which achieves single-round communication but still incurs the cost of a complete sort (see Section 8). For expansion, Jodes [98] employs the primitive of [62], which requires an initial oblivious sort followed by a recursive push-down phase that cannot be parallelized, a critical limitation, since the push-down operates over the output domain M.

Strawman Solution. A strawman solution for both primitives is to lift state-of-the-art single-server constructions to the distributed setting. In particular, one could extend the compaction and expansion algorithms of [82] and [73] to operate across multiple servers. However, this approach incurs a large communication overhead: coordinating the distributed execution requires $O(\log E)$ communication rounds, where E denotes the number of servers.

Why vertical partitioning is ideal. Both primitives have a distinctive property that makes them ideal candidates for vertical partitioning: their computation keys are minimal. Compaction requires only a single bit per tuple (marked or unmarked), and expansion requires only $\lceil \log M \rceil$ bits (the target index), where M is expansion's output size. Replicating such small keys across all E servers incurs negligible overhead, while each server holds a shard of every tuple and can execute the primitive independently. Moreover, each server operates on shards of size B/E, alleviating the block-size bottleneck that degrades all prior approaches.

Our approach. Both primitives follow the same three-phase vertical partitioning pattern, differing only in the local computation performed in phase (b):

(a) Shard generation and exchange. Each server E_k generates E shards per tuple and stores them in an array of shards S_{E_k}. The computation key is replicated across all shards, while the payload is evenly split. The shards are arranged into segments of n consecutive entries, and each server performs an all-to-all exchange—server E_k sends its m-th segment seg_m to server E_m and receives the k-th segment seg_k in return. After the exchange, each server E_k holds the k-th shard of every original tuple, with consistent ordering across all servers. This phase is illustrated in Steps 1–2 of Figure 3. Due to space limitations we include the pseudocode of compaction and expansion in the Appendix A.

(b) Local computation. Since all servers hold identically ordered shards, each executes the primitive independently, with no communication:

- *Compaction:* Each server runs a stable, shared-memory compaction algorithm (ORCompact [82]) on its local shard

array. Marked shards are moved to the front, preserving their relative order (Steps 3–4 in Figure 3).
- *Expansion:* Each server computes destination indices via a parallel scan and obliviously pushes tuples to their target positions using the recursive algorithm from [73].

Determinism guarantees that all servers produce identically ordered outputs, so shards at the same index across servers are guaranteed to originate from the same tuple.

(c) Restructure. The servers perform a second all-to-all exchange to return segments to their original owners (Step 5 in Figure 3). Each server then reconstructs the original tuples by merging shards that are n positions apart (Steps 6-7).

Efficiency. The computational complexity of both primitives is dominated by the local computation phase, which runs in $O(\frac{N \log N}{p})$ for compaction and $O(\frac{M \log M}{p})$ for expansion per server with p threads. Shard generation and restructure are linear, $O(\frac{N}{p})$ and $O(\frac{M}{p})$ in time and communication complexity. In particular, during execution, total communication is linear: $2N \cdot B$ bits for compaction (one exchange in each direction) and $(N + M) \cdot B$ bits for expansion.

Security. Assuming the local computation is oblivious (which holds for both [82] and [73]), the entire protocol is oblivious. All phases operate on public parameters (input size, tuple size, number of servers) and follow fixed, data-independent communication patterns. The adversary learns nothing beyond the input size: $\mathcal{L} = \perp$. The formal proofs are provided in Appendix C.1 (Theorems 1 and 2).

5.2 Hybrid Aggregation Tree

An aggregation tree is a parallel data structure that replaces sequential scans for computing group-wise operations (e.g., summing values within groups of matching keys). It is realized by Blelloch's work-efficient up-sweep/down-sweep parallel prefix-sum network [11], applied within groups: it operates in $O(\log n)$ rounds, propagating partial results up and down a binary tree over the array. The aggregation tree takes as input a table R, an auxiliary array G, and a function $f(R, p)$, where $p \in \{\texttt{sum}, \texttt{copy}\}$ specifies the operation to be applied. A tuple t is considered the first in its group if $G[t] = 0$; all subsequent tuples in the group are marked with $G[t] = 1$. When $p = \texttt{sum}$, the aggregation tree behaves like a group-wise prefix sum: instead of computing the prefix sum across the entire array, it performs the operation within each group.

The function selector f determines the aggregation mode and can be one of $f \in \{\texttt{Prefix}, \texttt{Suffix}, \texttt{Full}\}$. When $f = \texttt{Prefix}$ (or $\texttt{Suffix}$), the tree computes the prefix (or suffix) sum within each group. Furthermore, when $f = \texttt{Full}$, the function combines both prefix and suffix computations to produce the total sum of the group, which is then assigned to every tuple within that group. In contrast, when $p = \texttt{copy}$, the value of the first tuple is propagated to all tuples within the same group. We differentiate two primary use cases of

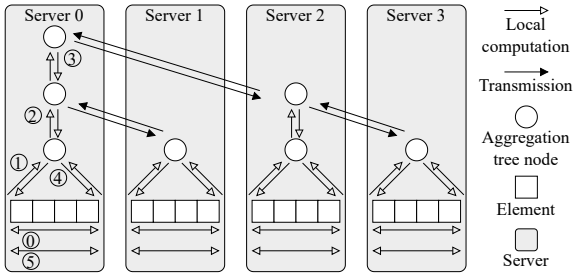

Figure 4. Distributed Aggregation Tree when $p = sum$.

$(R, D) \leftarrow$ DISCO-AggTree(R, D, f, p)

1: **for** each server $E_k \in E$ **do**
2: /* **First distribution of the elements** */
3: Retrieve R_{E_k} and D_{E_k}
4: /* **Phase 1 – Local aggregation** */
5: $Res \leftarrow$AggTree(R_{E_k}, D_{E_k}, f, p)
6: Initialize A with $\log E$ empty nodes
7: $A[\log E\text{–}1] \leftarrow (Res[0], D[0], Res[n\text{–}1], D[n\text{–}1])$
8: /* **Phase 2 – Distributed Upstream** */
9: **for** $L \leftarrow \log E - 1, ..., 1$ **do**
10: **for** each pair (E_R, E_S) on this layer **do**
11: Sender E_S sends $A^{L-1}_{E_S}$ to recipient E_R
12: $A^{L}_{E_R} \leftarrow$ DAggUp$(A^{L-1}_{E_R}, A^{L-1}_{E_S})$
13: /* **Phase 3 – Distributed Downstream** */
14: **for** $L \leftarrow 1, ..., \log E - 1$ **do**
15: **for** each pair (E_R, E_S) on this layer **do**
16: Sender E_S sends $A^{L}_{E_S}$ to recipient E_R
17: $A^{L-1}_{E_S} \leftarrow$ DAggDown$(A^{L}_{E_R}, A^{L-1}_{E_S})$
18: **for** each server $E_k \in E$ **do**
19: /* **Phase 4 – Second Local aggregation** */
20: $Res \leftarrow$AggTree(R_{E_k}, D_{E_K}, f, p)
21: Send back the result (R_{E_k}, D_{E_k})
22: **return** (R, D)

Figure 5. Pseudocode for Distributed Aggregation Tree

the aforementioned aggregation tree. First, we describe the distributed approach to compute an aggregation tree when $p = sum$ (see Figure 4), and then when $p = copy$.

Sum. In this scenario, inter-server communication is minimal. Each server only needs to transmit a constant number of aggregation nodes, where each node stores a single bit from table G and the aggregated value (i.e., the result of applying f) for the first and last tuples of its local array. To compute prefix and suffix aggregates, we implement the aggregation tree in a hybrid partition. The algorithm consists of four phases: (a) local aggregation, (b) distributed upstream, (c) distributed downstream, and (d) a final round of local aggregation (see Figure 5). DAggUP and DAggDown appear in Appendix A.1.

In (a), each server $E_k \in E$ locally computes the aggregation function on its array R_{E_k} using the aggregation tree

primitive introduced in [73]. Each server then constructs an array of aggregation nodes with a size equal to the number of tree layers in which it participates. In the leaf node of this array, each server stores data corresponding to the first and last tuples of its local array (see Line 6). Phases (b) and (c) implement a distributed aggregation tree across servers, as illustrated in Steps 1–4 of Figure 4. In particular, in (b), each layer defines sender servers E_S and receiver servers E_R. A sender E_S transmits its aggregation node $A^{L-1}_{E_S}$, and the receiver E_R updates its own node $A^{L}_{E_R}$ based on the received data. For example, if the first tuple of S_1 belongs to the same group as the last tuple of S_0 in Step 2, then S_0 will aggregate this group using function f (Lines 8–12). In (c), the downstream phase, servers propagate prefix and suffix values to the left and right subtrees, respectively. For instance, in the case of prefix aggregation at node $A^1_{S_0}$, server S_1 will obliviously aggregate its value depending on whether the last tuple of R_{S_0} belongs to the same group as the first tuple of R_{S_1}—i.e., if $G_{S_1}[0] = 1$ (Lines 14–17). Finally, in (d), each server uses the aggregated information received from other servers to perform a final round of local aggregation, updating its tuples accordingly. We defer a detailed explanation of the algorithm, together with a step-by-step worked example, to Appendix A.1.

Copying. While the aggregation tree incurs minimal communication overhead, in scenarios where $p =$ copy the entire tuple must be copied into the output group. When working with large blocks, it is more efficient to generate shards for each tuple and perform the aggregation vertically. Following the strategy outlined in Distributed Compaction: (a) each server receives shards of all original tuples (N total), (b) it performs the aggregation tree locally using multiple processing units. For this step, we use the aggregation tree from [73] as a black-box primitive. Finally, (c) each server sends the computed results back, and the original tuples are reconstructed, as described in Section 5.1. This vertical approach matters in Section 6, where joins require tuple copying.

Efficiency. In the sum approach, all local computations in Phases 1 and 4 execute in linear time, $O(n/p)$, where p is the number of processing units. Communication involves $\log E$ rounds, each consisting of $O(1)$ messages. In the copy approach, computation is also linear as each server handles N shards of size B/E, with no communication during execution.

Security. Both approaches presented in this section are fully oblivious. All control flow (such as loops and conditionals) depends solely on the input size or the number of servers. Inter-server communication is deterministic and independent of data values. Finally, all local aggregations utilize the aggregation tree primitive from [73], which has been proven oblivious. Regarding leakage, this algorithm leaks zero information to the adversary, thus $\mathcal{L} = \perp$. The formal proof is provided in Appendix C.1 (Theorem 4).

5.3 Hybrid Sort

Unlike compaction and expansion, sorting requires a computation key (the sort column) that is large enough (e.g., sorting on multiple columns) that replicating it across all E servers makes pure vertical partitioning inefficient for small tuples. At the same time, pure horizontal partitioning suffers from high communication overhead, as comparison-swaps frequently cross server boundaries. Hybrid partitioning offers the best of both worlds: servers are organized into C clusters, with vertical partitioning across clusters (reducing per-shard size and inter-cluster communication) and horizontal partitioning within each cluster (enabling efficient intra-cluster sorting). The optimal C depends on the tuple size and is selected by the query optimizer (Section 7.3).

Since oblivious sorting is the most expensive operator in the query pipeline, any speedup achieved here has a major impact on performance. In the following, we detail the optimizations applied to bucket sort from two disjoint papers and explain how we implement them atop our hybrid framework to get the most efficient distributed oblivious sort.

Optimizations of Ngai et al. [77]. Ngai et al. introduced optimizations for both bucket sort and bitonic sort (see Section 2). They show that bitonic sort offers superior performance in shared-memory environments. However, in distributed settings, bitonic sort becomes less practical due to its high communication overhead—requiring $O(\log E)$ rounds, each involving $O(N)$ communication. In contrast, their optimized bucket sort–featuring in-place bucket swaps to minimize memory usage and a communication pattern with linear cost–is significantly better suited for distributed systems.

Optimizations of Gu et al. [48]. Concurrently, Gu et al. proposed k-MergeSplit, which routes tuples through k buckets per layer rather than two [9, 77], and relaxes the power-of-two constraint on input sizes and bucket capacities. Their cost function selects the optimal bucket size Z, computes the minimum dummies ϵ for negligible overflow probability (2^{-60}), and determines the butterfly network structure via depth-first search. We incorporate these optimizations into our hybrid framework, discarding their SGXv1 page-swapping techniques (since we operate on SGXv2 with larger EPCs) and extending the cost model with communication constraints—in particular, enforcing that k is divisible by the number of servers for balanced workloads (Line 5 in Figure 13). Each cluster C independently constructs its butterfly network. Every server $E_k \in C$ applies the initial k-MergeSplit layers using the first $\lceil \log G \rceil$ bits, then exchanges buckets with servers within the cluster [77]. Each server completes the remaining layers and performs intra-bucket oblivious sorting to finish the Oblivious Random Permutation (ORP) [9, 77], followed by a final sort within each cluster.

Efficiency. The total communication—without accounting for the initial data distribution and result collection—across all clusters is $N \cdot B(1 + \epsilon) + N \cdot B = N \cdot B\,(2 + \epsilon)$, where

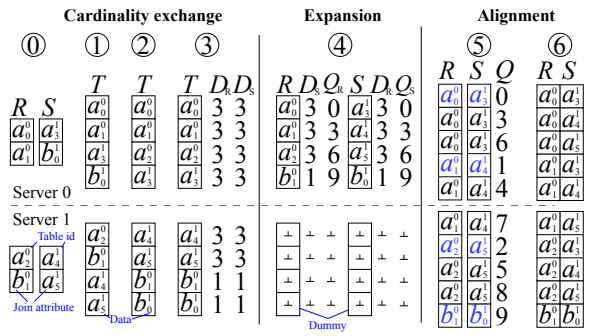

Figure 6. Distributed Oblivious Join.

$N \cdot B\ (1+\epsilon)$ comes from the ORP and $N \cdot B$ comes from the non-oblivious sort. The computation cost is $O((n \log n)/p)$.

Security. In Figure 13, all inter-server communication (Lines 3–4 and 31–32) is predetermined and depends solely on the input size N, the number of clusters C, and the number of servers E. The heuristic function invoked in Line 5 also takes as input the number of servers and the number of input tuples to compute the required parameters. Furthermore, all computation and communication between Lines 7–28 depend only on the number of clusters C, the number of servers E, and the parameters computed earlier. Each function invoked in this range is oblivious by prior work. For instance, the k-MergeSplit algorithm has been proven oblivious in [48], and OSort used in Line 28 is the oblivious bitonic sort from [77]. Finally, as stated in [9], after an oblivious random permutation, all ties with prior access patterns are broken, allowing us to use a non-oblivious algorithm, such as MergeSort (Lines 29-30). Thus, this algorithm has no leakage, $\mathcal{L} = \perp$. The formal proof is in Appendix C.1 (Theorem 3).

6 Database Operators

Non-Foreign Key Joins (NFK-Join). A non-foreign key join query is one of the most challenging individual operators in relational databases. Given two input tables, R and S, and a join condition θ, the result of the join $R \bowtie_\theta S$ consists of all tuple pairs $(r, s) \in R \times S$ for which θ holds. For example, if $\theta : R.a = S.a$, then the output includes all tuples $(r, s) \in R \times S$ where the values of a in R and a in S are equal. In this paper, we emphasize joins where θ means equality. Unlike a foreign-key join, an NFK-join permits duplicate keys on *both* sides: a key may appear multiple times in both R and S, so a key with multiplicities a in R and b in S produces $a \cdot b$ output tuples. Its complexity is therefore $O((N+M)\log(N+M))$ and it leaks only the output size M.

Operator Details. The join proceeds in three phases, illustrated in Figure 6. In the *cardinality exchange* phase (a), each server merges its local portions of R and S into a table T, which is then globally sorted by join key and table identifier using our hybrid sort (Section 5.3, Steps 1–2). A distributed aggregation tree with f = Full computes, for each tuple, the cardinality of its join key in each table: if a key appears ℓ times in R, every matching tuple in S must generate ℓ copies, and symmetrically. Two executions of our vertical compaction then separate T back into R and S, each annotated with the other table's cardinalities (Steps 3–4).

In the *expansion* phase (b), both tables are expanded to the output size M following the procedure described in Section 5.1: target indices are computed via prefix sum on the received cardinalities, and our distributed expansion pushes each real tuple to its target position (Step 5, blue entries). An aggregation tree with p = *copy* then fills the dummy slots, replicating the tables.

Since compaction and expansion both favor vertical partitioning, we apply *shard reuse*: shards are generated and distributed once before compaction and reconstructed only after expansion completes, rather than paying a full shard exchange between the two primitives (see Section 7.2). Vertical partitioning is precisely what makes this possible: because both chained primitives operate over the *same* shard layout, the intermediate tuples never need to be reconstructed and re-sharded between them, so the shard exchange is paid once for the whole chain rather than once per primitive.

Finally, in the *alignment* phase (c), we compute the final output index Q for each tuple in S by interleaving positions within each group (column Q in Step 5). A final hybrid sort on Q aligns S so that matching tuples from R and S occupy corresponding positions, producing the joined output (Step 6).

Foreign Key Join. A foreign key join consists of a primary table P and a foreign table F, where the entries in F reference primary keys in P. In this setting, we avoid the need to create duplicates or align tuples from F, since for every join key k in F, there is exactly one matching key in P. This property allows us to simplify our design.

One might observe that a foreign key join can be executed using the same algorithm, with an early termination. Specifically: (a) each server $E_k \in E$ merges its local partitions P_{E_k} and F_{E_k} into a table T_{E_k}; (b) a distributed sort is performed to globally sort T across all servers; and (c) a distributed aggregation tree, as described in Section 5.2, propagates the values from P to F for matching join keys. To remove redundant tuples, (d) we apply our vertical compaction and finalize the output by discarding dummy records. As in the non-foreign key join, shard reuse can be used between steps (c) and (d), allowing shards to be communicated only once. This optimization enables both the aggregation tree and the compaction to be executed locally before returning the final results. In summary, a foreign-key join terminates at the cardinality exchange phase by adjusting the aggregation tree.

Efficiency. Our NFK-Join achieves $O(((n+m)\log n)/p)$ computation and $(N \cdot B \cdot (2+\epsilon)) + (N+M) \cdot B + O(1) \cdot \log E + M/2 \cdot B \cdot (2+\epsilon)$ bits of communication, where the first and last terms stem from sorting ($M/2$ because alignment sorts

only one table) and the middle terms from vertical compaction, expansion, and aggregation. Shard reuse reduces the middle term from $(3N + M) \cdot B$ to $(N + M) \cdot B$ by avoiding redundant shard exchanges between chained primitives. For Foreign-Key Joins, computation is $O((n \log n)/p)$ with $N \cdot B(2 + \epsilon) + 2N \cdot B$ bits of communication.

Distributed Oblivious Filter & Aggregation.

For completeness, we discuss filter and aggregation, following the algorithms of [73] with our distributed primitives.

Filter. Given a table R and a predicate p, a filter returns all tuples $r \in R$ satisfying p. Prior approaches rely on oblivious sorting: Opaque [102] marks non-matching tuples and sorts them to the end, while SODA [66] redistributes tuples via an oblivious shuffle before local sorting. We replace sorting entirely with a parallel scan followed by our distributed vertical compaction.

Aggregation. Given a table R, an aggregation query computes a function $F \in \{\texttt{COUNT}, \texttt{SUM}, \texttt{AVG}, \texttt{MIN}, \texttt{MAX}\}$ over groups of tuples. Our approach first sorts R globally using our hybrid sort, then applies the distributed aggregation tree ($\log E$ rounds of $O(1)$ communication) to compute F within each group, and finally removes excess tuples via vertical compaction. With shard reuse, we distribute shards once before the aggregation tree and reconstruct only after compaction, enabling both phases to execute locally.

7 Oblivious Query Optimization

Prior distributed oblivious systems either lack a query optimizer entirely [62, 66, 98] or rely solely on fixed rewrite rules without a cost model [73, 102]. In this section, we present the first query optimizer for distributed oblivious databases that combines a cost model with obliviousness-preserving rewrite rules and adaptive partitioning selection. All optimization decisions depend exclusively on public parameters (N, B, E, and the query structure), ensuring that the generated physical plan is data-independent; obliviousness therefore follows directly from the obliviousness of the underlying primitives (Section 5). Unlike conventional database optimizers, which rely on data-dependent statistics such as cardinality estimates, DISCO's optimizer operates entirely on public information. It performs two complementary tasks: (i) rewrites the physical plan using seven obliviousness-preserving optimization rules (e.g., join–aggregation fusion), and (ii) selects the execution strategy and resource configuration through a cost model, including the partitioning scheme, all without observing or inferring any data-dependent information.

7.1 Cost Model

The optimizer minimizes *critical-path latency*. Each primitive (sort, compaction, expansion, aggregation tree) has a closed-form cost expressed as the maximum of its communication time (total bytes divided by network bandwidth) and its computation time (maximum local work across servers). For instance, distributed sort communicates $N \cdot B \cdot (2+\varepsilon)$ bytes in a single round, where ε accounts for the dummy overhead of the oblivious routing protocol. Because the objective is the maximum of the communication and computation costs, the optimizer minimizes both, with the bottleneck depending on the operator and workload. Execution is bandwidth-bound for large block sizes (B) or inputs (N), and latency-bound for small tuples. Whenever increasing communication volume shortens the critical path, the optimizer prefers that trade-off. This is the rationale behind its sort selection (Section 7.2): choosing between single-round bucket sort and multi-round bitonic sort.

The cost of an operator is the sum of its constituent primitive costs. For a plan consisting of a chain of operators $[op_1, \ldots, op_k]$ with cluster assignments $[C_1, \ldots, C_k]$, the total cost is: $T(P) = \sum_{i=1}^{k} T(op_i, C_i) + \sum_{i=1}^{k-1} T_{reshard}(C_i, C_{i+1})$, where $T_{reshard}(C_i, C_{i+1}) = N \cdot B'$ bytes when $C_i \neq C_{i+1}$ (a full shard redistribution) and zero otherwise. The cost model is additive and monotone: adding operators to a plan cannot reduce its cost. For plans with independent branches executing on disjoint server sets, the critical-path cost of the parallel region is $\max_i T(b_i)$. All cost expressions depend exclusively on public parameters (N, B, E, C, ε, and query structure), so all optimizer decisions are data-independent and reproducible by any party with knowledge of the query and deployment configuration. Since each primitive's oblivious operation counts are *exact* functions of these public parameters, the optimizer does not statistically estimate costs the way a traditional optimizer does; it *computes* each candidate plan's predicted wall-clock latency by scaling the exact counts with deployment constants measured for the target cluster (network bandwidth, per-swap CPU cost, and the dummy factor ε), and selects the minimum-cost plan.

7.2 Optimization Rules

The optimizer applies rewrite rules in six deterministic phases.

Operator Fusion. Adjacent operators are merged to eliminate redundant primitives. *Filter–Join:* instead of executing a full filter (scan + compaction), we perform only a parallel scan that replaces failing tuples with dummy entries carrying a designated maximum key value, then proceed directly to the join's cardinality exchange. The subsequent sort orders by join key, so dummies sort last and are naturally excluded. This eliminates one compaction and one reconstruction, saving $2N \cdot B$ bytes of communication. *Filter–Aggregation:* when an aggregation includes a filtering predicate, the compaction step of the filter is unnecessary—the system marks failing tuples via a parallel scan, sorts the table, and performs the aggregation only over tuples satisfying the predicate. This eliminates one compaction ($2N \cdot B$ bytes). *Join–Aggregation:* when the aggregation's GROUP-BY key is a superset of the join key, each group in the output contains exactly one representative tuple per key, so the join's expansion and alignment

phases can be omitted entirely—avoiding the materialization of up to $O(N^2)$ intermediate tuples. The join reduces to a cardinality exchange followed directly by the aggregation. Additionally, when the GROUP-BY key equals the join key, the aggregation's initial sort is redundant, saving an additional $N \cdot B \cdot (2+\varepsilon)$ bytes.

Column Pruning and Parallel Branches. *Column Pruning:* For a given query, only clusters storing columns that contribute to the result need to participate in the computation. The optimizer computes the active cluster set $A \subseteq \{1, \ldots, C\}$ and replaces the full block size B with the pruned size $B' = \sum_{col \in cols(Q)} size(col)$. *Parallel branch execution:* independent branches in the query DAG (e.g., scans and filters on disjoint tables before a join) are scheduled concurrently on disjoint server sets when the total active clusters fit within E, changing cost from $\sum_i T(b_i)$ to $\max_i T(b_i)$.

Sort Propagation. If an operator requires its input sorted on key k and the input is already annotated as $sorted(k)$, because a preceding sort or order-preserving primitive (compaction, expansion) established that order, the redundant sort is skipped. This saves $N \cdot B \cdot (2+\varepsilon)$ bytes per eliminated sort, a significant fraction of the join's execution time.

Shard Reuse. When chained operators $op_1, \ldots, op_k$ all operate under the same partitioning configuration, the system generates shards once at the beginning of the chain and reconstructs them once at the end, rather than paying a shard exchange after each individual operator. As demonstrated in the join operator, this reduces communication from $3N + M$ to $N + M$ elements when compaction and expansion are chained under vertical partitioning. The optimizer generalizes this to any sequence of consecutive operators sharing a configuration.

Adaptive Partitioning. The optimal number of clusters C varies across operators and tuple sizes: sorting benefits from fewer clusters when tuples are small but scales better with higher C for large tuples, while compaction and expansion consistently favor maximum partitioning. The optimizer assigns each operator op_i a cluster count $C_i^* = \arg\min_{C_i} \left[T(op_i, C_i) + T_{reshard}(C_{i-1}, C_i) \right]$, introducing a re-sharding step only when the per-operator saving exceeds the $N \cdot B'$ redistribution cost.

Sort Algorithm Selection. At each sort operator, the optimizer evaluates both bucket sort ($O(N \cdot B' \cdot (2+\varepsilon))$ communication in one round) and bitonic sort ($O(N \cdot B' \cdot \log E)$ communication over $\log E$ rounds) under the assigned cluster count C_i, and selects the algorithm with the lower computed cost. The crossover depends on B', C_i, and E: when B'/C_i is small enough that further vertical partitioning yields no benefit, bucket sort dominates; when B' is large, bitonic sort benefits from increasing C_i as partitioning reduces the per-shard volume at each of its $\log E$ rounds (see Figure 8).

7.3 Query Optimizer

The optimizer converts a logical query plan into an efficient physical plan via a sequence of deterministic rewrite phases, followed by cost-based plan selection. It first performs *operator fusion*, and then applies *column pruning*, restricting execution to the clusters that hold the referenced columns and reducing the effective block size from B to B'. Next, it applies *sort propagation*, eliminating redundant sorting. This is followed by *communication optimizations*, including shard reuse to avoid unnecessary data movement, parallel execution of independent branches on otherwise idle servers, and selection of the most efficient sorting algorithm.

Finally, in the *plan selection* phase, the optimizer assigns an appropriate cluster count to each operator, balancing its local execution cost against the overhead of re-sharding. For each operator, it evaluates all feasible cluster counts—powers of two up to $\min(E, C_{\text{active}})$ and subject to $B'/C \geq 32$—and, for each choice, determines the optimal configuration of the preceding operator by accounting for both operator cost and the cost of transitioning between cluster sizes. For plans with parallel branches, each branch is optimized independently and reconciled at merge points (e.g., joins), where the incoming cluster configurations must be compatible.

8 Evaluation

Implementation. We deploy 32 Azure Standard_DC16ds_v3 machines (16 cores, 64GB EPC, Intel SGXv2) connected via a local network (RTT = 1.25ms, 31 Gbps bandwidth). All experiments use 16 threads. For scalability experiments, we deploy 128 Standard_DC32ds_v3 servers (32 cores, 192GB EPC, SGXv2). Our implementation comprises approximately 13K lines of C on top of the Open Enclave SDK 0.19.4 [3]. Inter-server communication runs over MPICH [2], secured by mutually-attested TLS channels between enclaves built with Mbed TLS [1] which also provides the enclave's authenticated encryption and pseudorandom generation; the framework builds on the distributed sorting infrastructure of [77]. Oblivious swaps use XOR-based methods for elements ≥ 32B and CMOV for smaller elements.

Baselines. We compare sort against DBucket and DBitonic [77]; joins against Opaque [102], SODA [66], and Jodes [98] using their open-source implementations (SODA via [98]); and filter/aggregation against Opaque. We also implement Naive-Obliviator, which extends the single-server Obliviator [73] to a shared-nothing setting using horizontal partitioning with $\log E$-round compaction [82] and bitonic sort for distributed primitives. All baselines run on the same SGXv2 hardware and EPC configuration. Each baseline is evaluated only on the operators it supports: Opaque [102] lacks non-foreign-key joins; SODA [66] has no specialized foreign-key join; and Jodes [98] supports only foreign-key and non-foreign-key joins.

Datasets. We evaluate on both synthetic and real-world data. Since oblivious algorithms use constant-time operations, performance depends only on input size N, output size M, and block size B, and not on data distribution. Synthetic join data is generated using [62], producing two tables of size $N/2$ with output $M = N$. Unless otherwise noted, the computation key is 8B, matching prior work [66, 77, 98]—for all synthetic, real-world, PCD and TPC-H workloads; the exception is BDB-Q3, which uses a 100B key to exercise the large-key regime where the optimizer favors horizontal partitioning. Following standard practice [66, 77, 98, 102], we assume data is pre-distributed across servers; the one-time linear redistribution cost is identical for all approaches. Finally, we clarify where enclave paging is observable. Paging occurs only when a server's working set exceeds its EPC capacity, which in our evaluation happens exclusively in the scalability experiments that intentionally push input sizes beyond the EPC limit. In contrast, all head-to-head comparisons with prior work execute entirely within the EPC, so the observed speedups stem from our optimized primitives, reduced communication, and operating on shards rather than from paging.

Cost of Obliviousness. Before comparing against prior oblivious systems, we calibrate the cost of obliviousness against non-oblivious distributed joins on the same hardware and inputs ($N = 2^{25}$, $B = 256$B, 32 servers). DISCO-V is only 11.9× slower than a plaintext distributed sort-merge join, which performs the same $O(N \log N)$ work, and 52.2× slower than a plaintext hash join, a linear time $O(N)$ access-pattern-leaking baseline. Thus, DISCO brings oblivious joins within one-to-two orders of magnitude of a fully non-private engine, while prior oblivious systems remain 15×–180× slower than DISCO itself (Section 8.2).

8.1 Microbenchmarks

We first evaluate each distributed primitive, comparing against prior methods across varying block sizes and server counts. When applicable, the figures also report a fully horizontal execution, to showcase the importance of our vertical partitioning. For example, Figure 7 for $B = 256$B reports the performance of our hybrid sort under fully horizontal partitioning ($C = 1$).

Sort. Figure 7 evaluates our distributed sort (Section 5.3) and DISCO-Bitonic—bitonic sort [77] enhanced with our partitioning—on $N = 10^7$ tuples across 32 servers at $B \in \{32, 256, 1024\}$B. DISCO-Sort selects the best configuration per block size and achieves up to 14.7× and 6.9× speedups over prior bitonic and bucket sort. Single-server DBucket at $B = 1024$B is omitted as dummy overhead exceeds EPC; our DISCO-DBucket-OPT avoids this through reduced dummy counts. For $B = 32$B, we do not report DISCO-Horiz separately because our optimizer always selects this configuration for such small block sizes; consequently, DISCO-Sort and DISCO-Horiz are identical in this setting. Figure 8 varies

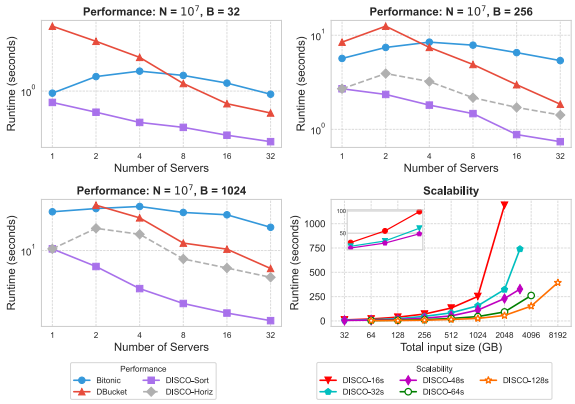

Figure 7. Distributed sort vs. prior works. DISCO-Sort showcases the best time among our proposed sorts.

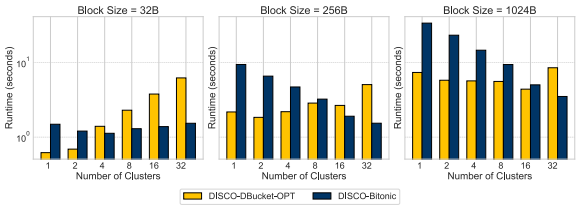

Figure 8. Performance of Hybrid Sorts. $C = 1$, and $C = 32$ result in horizontal and vertical executions.

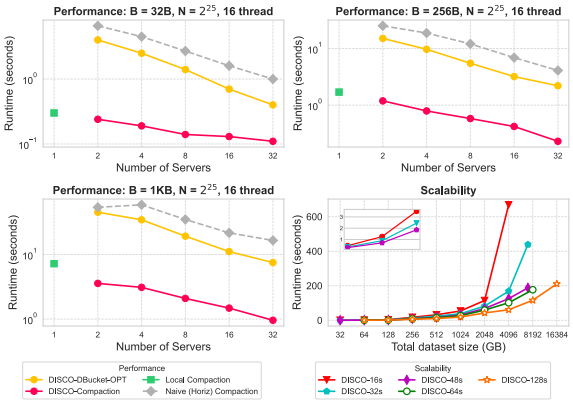

Figure 9. Performance and scalability of distributed compaction.

the cluster count C. For small blocks, DISCO-DBucket-OPT dominates, as payloads cannot split further; for larger blocks, DISCO-Bitonic benefits from increasing C. Hybrid partitioning improves both, with optimal C varying by algorithm and block size.

Compaction. Figure 9 compares our vertical compaction against a sorting-based baseline (DISCO-DBucket-OPT) and a naive $\log E$-round adaptation of [82]. Our scheme is 2.1×–9.35× faster than sorting-based compaction and 3.8×–16× than the naive approach, with the largest gains at $B = 1024$B. The naive approach performs worse than oblivious sort, underscoring the importance of single-round communication.

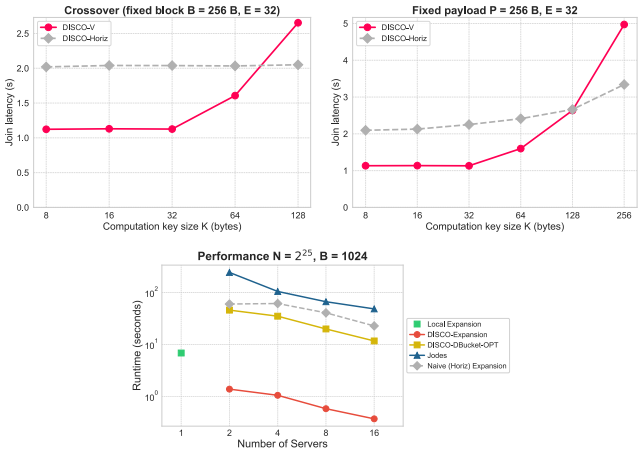

Figure 10. (Top) Large-computation key and (Bottom) Expansion experiments.

Expansion. Figure 10 (bottom) compares our vertical expansion against three baselines: the expansion primitive from Jodes [98], a naive distributed extension of [73], and our optimized distributed sort (Section 5.3). We evaluate on $N = 2^{25}$ tuples with block size $B = 1024$B across 32 servers. Our expansion achieves 28×–129× speedups over all baselines, validating the effectiveness of vertical partitioning for this primitive. Additional experiments on smaller block sizes (32B, 256B) are provided in Appendix C.3, where the speedups are 3×–38× and 16×–81×, respectively.

Large computation key. We further evaluate how the computation-key size affects the performance of our partitioning approaches. Figure 10 presents two microbenchmarks using our join operator (see § 6). First, we fix the tuple size at $B = 256$B and vary the fraction occupied by the computation key K, with the remaining $B - K$ bytes corresponding to the payload. Vertical partitioning initially outperforms fully horizontal execution, as expected. However, as the computation key occupies a larger fraction of the tuple, its advantage diminishes because the key is replicated and the payload cannot be partitioned further. In contrast, horizontal execution remains unaffected, as it operates on the full tuple regardless of the key-to-payload ratio. Second, we fix the payload size and progressively increase the computation-key size. As expected, the latency of both approaches increases with K; however, vertical partitioning is more sensitive to this increase due to the additional overhead of replicating the larger computation key across shards.

8.2 Non-Foreign Key Joins

We next evaluate the performance of our oblivious operators, which compose the primitives from the previous subsection.

Synthetic data. Figure 11 shows join results on $N = 2^{25}$ tuples. DISCO-Hy combines our hybrid sort with vertical compaction and expansion; DISCO-V is a fully vertical implementation of [73]; SS-Obliviator is the single-server

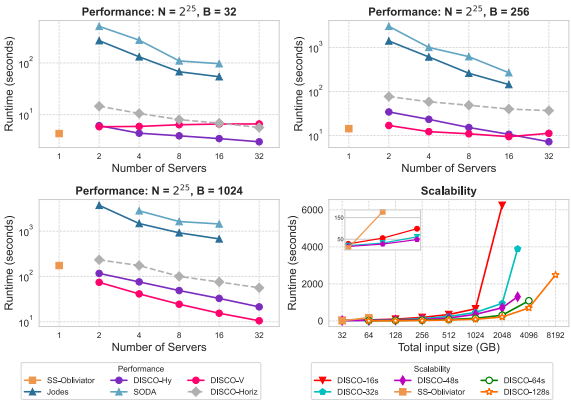

Figure 11. Performance and scalability of distributed join.

		DBLP	Email	Youtube	wiki
Input	# of records (M)	2.10M	0.84M	5.97M	57.02M
	Size (GB)	0.06	0.03	0.18	1.70
Output	# of records (M)	14M	101M	373M	5,326M
	Size (GB)	0.42	3.02	11.11	158.73
Time (secs)	SODA	7.12	40.58	275.83	N.A.
	Jodes	6.49	19.08	68.76	935.78
	DISCO-Join	0.60	1.66	7.03	94.35

Table 1. Performance of joins on real-world datasets.

baseline. Jodes cannot run beyond 16 servers due to a threading limitation that causes deadlock. Both DISCO variants yield 15×–180× speedups over SODA [66] and Jodes [98], with the lower end at $B = 256$B and upper end at $B = 1024$B with 32 servers. For $B = 256$B, DISCO-Hy and DISCO-Horiz (the latter only on $E = 32$) surpass DISCO-V since tuples cannot be further partitioned. SS-Obliviator performs well at moderate scales but collapses beyond EPC capacity. Jodes is omitted from TB-scale experiments as it does not scale and executing such experiment would be prohibitively expensive.

Real-world data. Table 1 evaluates on DBLP [99], Email [65], YouTube [99], and Wiki [55, 100] (inputs up to 1.7GB, outputs up to 158GB) across 16 servers. Our approach achieves 11.8×–39.2× speedups over SODA and 9.8×–10.8× over Jodes. SODA is omitted on Wiki due to prohibitive runtime.

8.3 Scalability Experiments

We further conduct large-scale experiments to explore the scalability limits of our system. As shown in the rightmost subplots of Figures 7, 9, and 11, our system is the *first* to break the *terabyte* barrier in distributed oblivious computation. In each of these scalability subplots, the upper-left inset zooms into the small-input region—where the logarithmic axes hide runtime differences—and its lines correspond to the same configurations as the main plot. Leveraging 32 threads and 192 GB of EPC per server, we evaluate our hybrid join (single-cluster sort) and vertical compaction on increasingly large inputs. Our algorithms scale to datasets as large as **16 TB**, an order-of-magnitude improvement over prior systems. As anticipated, once the input exceeds the

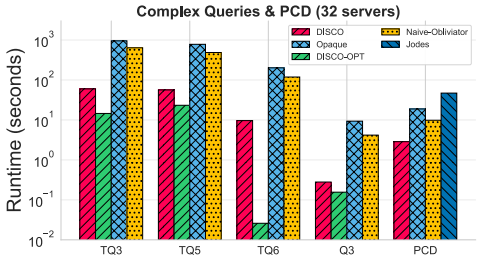

Figure 12. End-to-end runtime on TPC-H (TQ3/TQ5/TQ6), BDB Q3, and Private Contact Discovery (PCD), on 32 servers.

EPC capacity, performance begins to deteriorate—consistent with our discussion in the Introduction. For example, DISCO-16s (distributed sort across 16 servers) hits its EPC ceiling at 2TB, triggering a sharp slowdown. Our compaction and join primitives continue to scale, reaching up to 16TB and 8TB.

8.4 Complex Queries

Figure 12 evaluates end-to-end query pipelines for four queries, BDB Q3 [6] and TPC-H Q3, Q5, and Q6 [90] (scale factor $s = 100$, inputs up to 182GB), across 32 servers. We evaluate the three TPC-H queries used by prior oblivious systems for direct comparability; DISCO supports any query expressible with its primitives (equi-joins, filters, and aggregations). Even without the optimizer, DISCO achieves 17×–48× speedups over Opaque [102] and 8.7×–17× over Naive-Obliviator. With the full optimizer (Section 7), DISCO-OPT further improves performance by 15.4× on TQ3, 10.6× on TQ5, 1,027× on TQ6, and 5.1× on BDB Q3. The magnitude of these gains depends on how many optimizations can be applied within each query. TQ6 provides the most striking example. Prior work lacks a specialized algorithm for aggregation without a GROUP-BY predicate, instead relying on a costly generic aggregation operator. Our optimizer detects this pattern and eliminates all expensive primitives. Specifically, TQ6 is a single-table query that accesses only 4 of 16 columns; filter–aggregation fusion embeds the predicate directly into the aggregation function, removing an entire compaction and reconstruction phase; adaptive partitioning selects the optimal cluster count; and sort selection chooses the most efficient algorithm for the reduced block size. These optimizations compound to a 1,027× speedup. TQ3 (15.4×) benefits from sort propagation (skipping the aggregation sort when data is already ordered on the join key), 3× column pruning, and parallel execution of independent scans. TQ5 (10.6×), involving six tables and seven non-eliminable sorts, derives its gains primarily from sort selection and adaptive partitioning, as sorting dominates execution. Finally, BDB Q3 (5.1×) benefits from filter–join fusion, shard reuse, and parallel branching, though its smaller inputs limit optimization opportunities.

Private Contact Discovery. Motivated by Signal's Private Contact Discovery [72], we evaluate an oblivious foreign-key join that matches a client's contact list against Signal's database to find contacts that are already using Signal ($B = 160$B, with a computation key of 8B) on 32 servers. We consider the scenario where a client with $N_1 = 2^{12}$ contacts joins the database with $N_2 = 2^{25}$ users (shown in Figure 12). We compare against Opaque, Naive-Obliviator and Jodes, and showcase speedups of 6.6×, 3.4× and 16×, respectively.

9 Additional Related Work

Oblivious systems. A broad body of work applies obliviousness under different goals or threat models than DISCO. Oblivious RAM [39] and the oblivious key–value stores and data structures built on it—Oblix [74], ObliDB [35], Pancake [43], SONIC [89], SEAL [28] and the distributed Snoopy [23] and SHORTSTACK [95]—target point key–value access rather than the analytical scans DISCO optimizes. Oblivious techniques also protect communication metadata, e.g., SPARTA [36] for traffic-analysis-resistant anonymity, or oblivious graph queries, e.g., GraphOS [15]. In contrast, encrypted databases trade obliviousness for efficiency via order-revealing [7, 93, 96] or property-preserving [78, 91] encryption, or leaking access patterns, as in searchable and structured encryption [16, 21, 26, 27, 29–32, 76]; Weave [87] targets MapReduce-style analytics and OasisDB [5] assumes a trusted proxy.

MPC and differential privacy. Under different trust models, multi-party-computation systems—Secrecy [67], Conclave [94], ORQ [10], and CoVault [25]—provide secure database analytics by distributing trust across non-colluding parties rather than a TEE, at a higher communication cost. Beyond full systems, MPC research has produced several communication-efficient data-oblivious primitives—secret-shared shuffles and permutation networks, oblivious sorting, secure joins and group-by aggregation, and private set intersection [10, 38, 67, 75, 80]—that would be interesting to explore in the TEE setting. Orthogonally, differentially private engines such as Turbo [61] and Arboretum [70] bound what query answers reveal rather than hiding access patterns, a guarantee complementary to our fully-oblivious setting.

10 Conclusion

We presented DISCO, the first scalable distributed oblivious database system. By introducing vertical and hybrid partitioning to the oblivious setting, DISCO eliminates inter-server communication during computation and neutralizes the block-size bottleneck that limits all prior approaches. Our distributed primitives and operators outperform prior systems by up to two orders of magnitude, and our query optimizer—the first for distributed oblivious TEE-based systems—yields additional speedups of up to 1,027× on complex queries, scaling to 16TB of data. Several directions remain open: extending DISCO's optimizer (window functions, nested subqueries), or exploring cost-based rule ordering.

References

[1] [n. d.]. Mbed TLS. https://www.trustedfirmware.org/projects/mbed-tls/, 2023-04-29.

[2] [n. d.]. MPICH. https://www.mpich.org/, 2023-04-29.

[3] [n. d.]. Open Enclave SDK. https://openenclave.io/sdk/, 2023-04-29.

[4] Mohamed Ahmed Abdelraheem, Tobias Andersson, and Christian Gehrmann. 2017. Inference and Record-Injection Attacks on Searchable Encrypted Relational Databases. *IACR Cryptol. ePrint Arch.* (2017), 24. http://eprint.iacr.org/2017/024

[5] Haseeb Ahmed. 2025. OasisDB: An Oblivious and Scalable System for Relational Data. (2025).

[6] AMPLab, University of California, Berkeley. 2014. Big Data Benchmark. https://amplab.cs.berkeley.edu/benchmark/.

[7] Panagiotis Antonopoulos, Arvind Arasu, Kunal D Singh, Ken Eguro, Nitish Gupta, Rajat Jain, Raghav Kaushik, Hanuma Kodavalla, Donald Kossmann, Nikolas Ogg, et al. 2020. Azure SQL database always encrypted. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1511–1525.

[8] ARM Limited. 2004. *ARM TrustZone Technology*. Technical Report. ARM Holdings. https://developer.arm.com/documentation/102412/latest

[9] Gilad Asharov, TH Hubert Chan, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. 2020. Bucket oblivious sort: An extremely simple oblivious sort. In *SOSA*. 8–14.

[10] Eli Baum, Sam Buxbaum, Nitin Mathai, Muhammad Faisal, Vasiliki Kalavri, Mayank Varia, and John Liagouris. 2025. ORQ: Complex Analytics on Private Data with Strong Security Guarantees. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*.

[11] Guy E Blelloch. 1990. Prefix sums and their applications. (1990).

[12] Christiaan Brand and Sam Srinivas. 2018. Titan Security Key – Giving You Peace of Mind for Your Online Protection. https://blog.google/technology/safety-security/titan-security-key/ Google Security Blog.

[13] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*.

[14] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. 2015. Leakage-Abuse Attacks Against Searchable Encryption. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 668–679. doi:10.1145/2810103.2813700

[15] Javad Ghareh Chamani, Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Rasool Jalili. 2023. GraphOS: Towards Oblivious Graph Processing. *Proceedings of the VLDB Endowment* 16, 13 (2023), 4324–4338.

[16] Javad Ghareh Chamani, Dimitrios Papadopoulos, Mohammadamin Karbasforushan, and Ioannis Demertzis. 2022. Dynamic Searchable Encryption with Optimal Search in the Presence of Deletions. *USENIX Security* (2022).

[17] Zhao Chang, Dong Xie, Sheng Wang, and Feifei Li. 2022. Towards Practical Oblivious Join. In *SIGMOD*.

[18] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten-Hwang Lai. 2020. SgxPectre: Stealing Intel Secrets From SGX Enclaves via Speculative Execution . *S & P* (2020).

[19] Sanchuan Chen, Xiaokuan Zhang, Michael K Reiter, and Yinqian Zhang. 2017. Detecting privileged side-channel attacks in shielded execution with Déjá Vu. In *AsiaCCS*.

[20] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX*.

[21] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. 2006. Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions. In *CCS*.

[22] D. Kaplan, J. Powell, and T. Woller. 2016. *AMD memory encryption*. Technical Report.

[23] Emma Dauterman, Vivian Fang, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. 2021. Snoopy: Surpassing the Scalability Bottleneck of Oblivious Storage. In *SOSP*.

[24] Jonathan L Dautrich Jr and Chinya V Ravishankar. 2013. Compromising Privacy in Precise Query Protocols. In *Proceedings of the 16th International Conference on Extending Database Technology*. ACM, 155–166.

[25] Roberta De Viti, Isaac Sheff, Cason Kogan, Rodrigo Rodrigues, Jonathan Mace, and Kaushik Veeraraghavan. 2025. CoVault: Secure, Scalable Analytics of Personal Data. In *34th USENIX Security Symposium*. USENIX Association.

[26] Ioannis Demertzis, Javad Ghareh Chamani, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2020. Dynamic Searchable Encryption with Small Client Storage. In *Network and Distributed System Security Symposium*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/dynamic-searchable-encryption-with-small-client-storage/

[27] Ioannis Demertzis, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2018. Searchable Encryption with Optimal Locality: Achieving Sublogarithmic Read Efficiency. *CRYPTO* (2018).

[28] Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Saurabh Shintre. 2020. {SEAL}: Attack mitigation for encrypted databases via adjustable leakage. In *USENIX*.

[29] Ioannis Demertzis, Stavros Papadopoulos, Odysseas Papapetrou, Antonios Deligiannakis, and Minos Garofalakis. 2016. Practical Private Range Search Revisited. In *SIGMOD*.

[30] Ioannis Demertzis, Stavros Papadopoulos, Odysseas Papapetrou, Antonios Deligiannakis, Minos Garofalakis, and Charalampos Papamanthou. 2018. Practical Private Range Search in Depth. *TODS* (2018).

[31] Ioannis Demertzis and Charalampos Papamanthou. 2017. Fast searchable encryption with tunable locality. In *SIGMOD*.

[32] Ioannis Demertzis, Rajdeep Talapatra, and Charalampos Papamanthou. 2018. Efficient searchable encryption through compression. *PVLDB* (2018).

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

[34] Muhammad El-Hindi, Tobias Ziegler, Matthias Heinrich, Adrian Lutsch, Zheguang Zhao, and Carsten Binnig. 2022. Benchmarking the second generation of intel SGX hardware. In *DaMoN*.

[35] Saba Eskandarian and Matei Zaharia. 2019. ObliDB: oblivious query processing for secure databases. *Proc. VLDB Endow.* (2019).

[36] Kyle Fredrickson, Ioannis Demertzis, James P Hughes, and Darrell DE Long. 2025. Sparta: Practical anonymity with long-term resistance to traffic analysis. In *2025 IEEE Symposium on Security and Privacy (SP)*.

[37] Matthieu Giraud, Alexandre Anzala-Yamajako, Olivier Bernard, and Pascal Lafourcade. 2017. Practical Passive Leakage-abuse Attacks Against Symmetric Searchable Encryption. In *Proceedings of the 14th International Joint Conference on e-Business and Telecommunications (ICETE 2017) - Volume 4: SECRYPT*, Pierangela Samarati, Mohammad S. Obaidat, and Enrique Cabello (Eds.). SciTePress, 200–211. doi:10.5220/0006461202000211

[38] Eu-Jin Goh et al. 2003. Secure Indexes. *IACR Cryptology ePrint Archive* (2003).

[39] Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *J. ACM* 43, 3 (5 1996), 431–473. doi:10.1145/233551.233553

[40] Michael T Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. 2011. Oblivious RAM simulation with efficient worst-case access overhead. In *ACM CCSW*.

[41] Google LLC. 2018. Asylo: An Open and Flexible Framework for Enclave Applications. https://asylo.dev Open-source framework for

confidential computing in TEEs.

[42] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache attacks on Intel SGX. In *EUROSEC*.

[43] Paul Grubbs, Anurag Khandelwal, Marie-Sarah Lacharité, Lloyd Brown, Lucy Li, Rachit Agarwal, and Thomas Ristenpart. 2020. Pancake: Frequency Smoothing for Encrypted Data Stores. In *29th USENIX Security Symposium*. USENIX Association, 2451–2468.

[44] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. 2018. Pump up the Volume: Practical Database Reconstruction from Volume Leakage on Range Queries. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM, 315–331. doi:10.1145/3243734.3243864

[45] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. 2019. Learning to Reconstruct: Statistical Learning Theory and Encrypted Database Attacks. In *IEEE Symposium on Security and Privacy*. IEEE, 1067–1083. doi:10.1109/SP.2019.00030

[46] Paul Grubbs, Richard McPherson, Muhammad Naveed, Thomas Ristenpart, and Vitaly Shmatikov. 2016. Breaking Web Applications Built On Top of Encrypted Data. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM, 1353–1364. doi:10.1145/2976749.2978351

[47] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. 2017. Strong and efficient cache {Side-Channel} protection using hardware transactional memory. In *USENIX*.

[48] Tianyao Gu, Yilei Wang, Bingnan Chen, Afonso Tinoco, Elaine Shi, and Ke Yi. 2023. Efficient Oblivious Sorting and Shuffling for Hardware Enclaves. *Cryptology ePrint Archive* (2023).

[49] Zichen Gui, Oliver Johnson, and Bogdan Warinschi. 2019. Encrypted Databases: New Volume Attacks against Range Queries. In *Proceedings ACM SIGSAC Conference on Computer and Communications Security*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM, 361–378. doi:10.1145/3319535.3363210

[50] Marcus Hähnel, Weidong Cui, and Marcus Peinado. 2017. {High-Resolution} Side Channels for Untrusted Operating Systems. In *ATC*.

[51] Xiao Hu and Zhiang Wu. 2025. Optimal Oblivious Algorithms for Multi-Way Joins. In *28th International Conference on Database Theory (ICDT 2025)*.

[52] Ryan Hurst and Gary Belvin. [n. d.]. Google's Key Transparency.

[53] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation. In *19th Annual Network and Distributed System Security Symposium, NDSS*. The Internet Society.

[54] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. 2016. Generic Attacks on Secure Outsourced Databases. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM, 1329–1340. doi:10.1145/2976749.2978386

[55] Christine Klymko, David Gleich, and Tamara G Kolda. 2014. Using triangles to improve community detection in directed networks. *arXiv preprint arXiv:1404.5874* (2014).

[56] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2020. Spectre attacks: Exploiting speculative execution. *Commun. ACM* 63, 7 (2020), 93–101.

[57] Evgenios M Kornaropoulos, Nathaniel Moyer, Charalampos Papamanthou, and Alexandros Psomas. 2022. Leakage inversion: Towards quantifying privacy in searchable encryption. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 1829–1842.

[58] Evgenios M. Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. 2019. Data Recovery on Encrypted Databases with k-Nearest Neighbor Query Leakage. In *IEEE Symposium on Security and Privacy*. IEEE, 1033–1050. doi:10.1109/SP.2019.00015

[59] Evgenios M. Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. 2020. The State of the Uniform: Attacks on Encrypted Databases Beyond the Uniform Query Distribution. In *IEEE Symposium on Security and Privacy*. IEEE, 1223–1240. doi:10.1109/SP40000.2020.00029

[60] Evgenios M Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. 2021. Response-hiding encrypted ranges: Revisiting security via parametrized leakage-abuse attacks. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1502–1519.

[61] Kelly Kostopoulou, Pierre Tholoniat, Asaf Cidon, Roxana Geambasu, and Mathias Lécuyer. 2023. Turbo: Effective Caching in Differentially-Private Databases. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023*. ACM, 579–594. doi:10.1145/3600006.3613174

[62] Simeon Krastnikov, Florian Kerschbaum, and Douglas Stebila. 2020. Efficient oblivious database joins. *VLDB* (2020). https://www.vldb.org/pvldb/vol13/p2132-krastnikov.pdf

[63] Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. 2018. Improved Reconstruction Attacks on Encrypted Data Using Range Query Leakage. In *2018 IEEE Symposium on Security and Privacy, SP 2018*. IEEE Computer Society, 297–314. doi:10.1109/SP.2018.00002

[64] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring fine-grained control flow inside {SGX} enclaves with branch shadowing. In *USENIX Security 17*.

[65] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. 2007. Graph evolution: Densification and shrinking diameters. *ACM transactions on Knowledge Discovery from Data (TKDD)* 1, 1 (2007), 2–es.

[66] Xiang Li, Nuozhou Sun, Yunqian Luo, and Mingyu Gao. 2023. SODA: A Set of Fast Oblivious Algorithms in Distributed Secure Data Analytics. *Proceedings of the VLDB Endowment* 16, 7 (2023), 1671–1684.

[67] John Liagouris, Vasiliki Kalavri, Muhammad Faisal, and Mayank Varia. 2023. {SECRECY}: Secure collaborative analytics in untrusted clouds. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 1031–1056.

[68] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *arXiv preprint arXiv:1801.01207* (2018).

[69] Chang Liu, Liehuang Zhu, Mingzhong Wang, and Yu-an Tan. 2014. Search pattern leakage in searchable encryption: Attacks and new construction. *Inf. Sci.* 265 (2014), 176–188. doi:10.1016/j.ins.2013.11.021

[70] Elizabeth Margolin, Karan Newatia, Tao Luo, Edo Roth, and Andreas Haeberlen. 2023. Arboretum: A Planner for Large-Scale Federated Analytics with Differential Privacy. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023*. ACM, 451–465. doi:10.1145/3600006.3624566

[71] Evangelia Anna Markatou and Roberto Tamassia. 2019. Full Database Reconstruction with Access and Search Pattern Leakage. In *Information Security - 22nd International Conference, ISC (Lecture Notes in Computer Science, Vol. 11723)*, Zhiqiang Lin, Charalampos Papamanthou, and Michalis Polychronakis (Eds.). Springer, 25–43. doi:10.1007/978-3-030-30215-3_2

[72] Moxie Marlinspike. [n. d.]. Technology Preview: Private Contact Discovery for Signal.

[73] Apostolos Mavrogiannakis, Xian Wang, Ioannis Demertzis, Dimitrios Papadopoulos, and Minos Garofalakis. 2025. OBLIVIATOR: Oblivious Parallel Joins and other Operators in Shared Memory Environments. Cryptology ePrint Archive, Paper 2025/183. https://eprint.iacr.org/2025/183

[74] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. 2018. Oblix: An efficient oblivious search index. In *IEEE SP*.

[75] Payman Mohassel, Peter Rindal, and Mike Rosulek. 2020. Fast database joins and PSI for secret shared data. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1271–1287.

[76] Priyanka Mondal, Javad Ghareh Chamani, Ioannis Demertzis, and Dimitrios Papadopoulos. 2024. I/O-Efficient Dynamic Searchable Encryption Meets Forward & Backward Privacy. In *33rd USENIX Security Symposium (USENIX Security 24)*. 2527–2544.

[77] Nicholas Ngai, Ioannis Demertzis, Javad Ghareh Chamani, and Dimitrios Papadopoulos. 2024. Distributed & Scalable Oblivious Sorting and Shuffling. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 153–153.

[78] Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *SOSP*.

[79] David Pouliot and Charles V. Wright. 2016. The Shadow Nemesis: Inference Attacks on Efficiently Deployable, Efficiently Searchable Encryption. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM, 1341–1352. doi:10.1145/2976749.2978401

[80] Srinivasan Raghuraman and Peter Rindal. 2022. Blazing fast PSI from improved OKVS and subfield VOLE. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2505–2517.

[81] Majid Sabbagh, Yunsi Fei, Thomas Wahl, and A Adam Ding. 2018. Scadet: A side-channel attack detection tool for tracking prime-probe. In *ICCAD*.

[82] Sajin Sasy, Aaron Johnson, and Ian Goldberg. 2022. Fast Fully Oblivious Compaction and Shuffling. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*.

[83] Sajin Sasy, Aaron Johnson, and Ian Goldberg. 2023. 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*.

[84] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2017. Malware guard extension: Using SGX to conceal cache attacks. In *DIMVA*.

[85] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs.. In *NDSS*.

[86] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. 2016. Preventing page faults from telling your secrets. In *AsiaCCS*.

[87] Mahdi Soleimani, Grace Jia, and Anurag Khandelwal. 2025. Weave: Efficient and Expressive Oblivious Analytics at Scale. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. USENIX Association.

[88] Open Whisper Systems. [n. d.]. Signal Messenger. https://signal.org

[89] Nihal Talur and Ioannis Demertzis. [n. d.]. SONIC: Concurrent Oblivious RAM & Data Structures for Low-Latency and High-Throughput. ([n. d.]).

[90] Transaction Processing Performance Council. 2022. TPC-H Benchmark. http://www.tpc.org/tpch.

[91] Stephen Tu, M. Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. 2013. Processing Analytical Queries over Encrypted Data. In *Proceedings of the VLDB Endowment*, Vol. 6. 289–300.

[92] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient {Out-of-Order} execution. In *27th USENIX Security Symposium (USENIX Security 18)*.

[93] Dhinakaran Vinayagamurthy, Alexey Gribov, and Sergey Gorbunov. 2019. StealthDB: a scalable encrypted database with full SQL query support. *Proceedings on Privacy Enhancing Technologies* (2019).

[94] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. 2019. Conclave: secure multi-party computation on big data. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–18.

[95] Midhul Vuppalapati, Kushal Babel, Anurag Khandelwal, and Rachit Agarwal. 2022. SHORTSTACK: Distributed, Fault-tolerant, Oblivious Data Access. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, 719–734.

[96] Sheng Wang, Yiran Li, Huorong Li, Feifei Li, Chengjin Tian, Le Su, Yanshan Zhang, Yubing Ma, Lie Yan, Yuanyuan Sun, et al. 2022. Operon: An encrypted database for ownership-preserving data management. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3332–3345.

[97] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. 2017. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *CCS*.

[98] Yilei Wang, Xiangdong Zeng, Sheng Wang, and Feifei Li. 2025. Jodes: Efficient Oblivious Join in the Distributed Setting. *arXiv e-prints* (2025), arXiv–2501.

[99] Jaewon Yang and Jure Leskovec. 2012. Defining and evaluating network communities based on ground-truth. In *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*. 1–8.

[100] Hao Yin, Austin R Benson, Jure Leskovec, and David F Gleich. 2017. Local higher-order graph clustering. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*. 555–564.

[101] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. 2016. All Your Queries Are Belong to Us: The Power of File-Injection Attacks on Searchable Encryption. In *25th USENIX Security Symposium*, Thorsten Holz and Stefan Savage (Eds.). USENIX Association, 707–720. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/zhang

[102] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. 2017. Opaque: An Oblivious and Encrypted Distributed Analytics Platform.. In *NSDI*.

[103] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. 2017. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 283–298. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/zheng

[104] Jinhao Zhu, Liana Patel, Matei Zaharia, and Raluca Ada Popa. 2024. Compass: Encrypted semantic search with high accuracy. *Cryptology ePrint Archive* (2024).

Appendix

In the following sections, we provide the pseudocode of our distributed primitives (Appendix A), including the subroutines and a worked example of our distributed aggregation tree (Appendix A.1); a detailed description of our distributed expansion (Appendix B); and our analysis (Appendix C), comprising the computation and communication complexity comparison against prior work, the formal security definitions, simulators, and obliviousness proofs for all of our primitives (Appendix C.1), the asymptotic performance and security analysis of our filter and aggregation operators (Appendix C.2), and additional experiments on the aggregation tree, on expansion at smaller block sizes, and on the Big Data Benchmark (Appendix C.3). Appendices are not included in the peer-review process.

A Pseudocodes

A.1 Subroutines of Distributed Aggregation Tree

Here, we include the pseudocodes for the subroutines DAggUO and DAggDown, executed as part of our distributed oblivious aggregation tree (see Figure 5).

Worked example. We trace DISCO-AggTree (Figure 5) for a group-wise *sum* (p = `sum`, f = `Full`), where every tuple receives the total of its group. Consider $N = 8$ tuples spread over $E = 4$ servers ($n = 2$ each). The group-boundary array G marks the first tuple of each group with 0 and continuations with 1, defining three groups that straddle server boundaries: $A = \{p_0, p_1, p_2\}$, $B = \{p_3, p_4, p_5\}$, and $C = \{p_6, p_7\}$.

	S_0		S_1		S_2		S_3	
position	p_0	p_1	p_2	p_3	p_4	p_5	p_6	p_7
value R	4	2	3	5	1	6	2	7
group G	0	1	1	0	1	1	0	1
Phase 1 (local)	6	6	3	5	7	7	9	9
Phase 4 (final)	9	9	9	12	12	12	9	9

Phase 1 (local aggregation, Lines 6–13). Each server runs the single-server AggTree on its two local tuples, treating its first local tuple as a segment head. S_0 sums its local segment $\{4, 2\} = 6$; S_1 has a continuation (p_2) and a new head (p_3), giving local sums 3 and 5; S_2's two continuations sum to 7; and S_3's local group $\{2, 7\} = 9$. Each server then stores one boundary node $A[\log E - 1] = (Res[0], G[0], Res[n-1], G[n-1])$ carrying the aggregate and group bit of its first and last local tuples—e.g., S_1 stores $(3, 1, 5, 0)$.

Phases 2–3 (distributed upstream/downstream, Lines 14–28). These stitch the groups that cross server boundaries. Because p_2 (S_1's first tuple) has $G = 1$, it belongs to the same group A as S_0's tuples, so the upstream pass combines S_0's partial 6 with S_1's leading 3 to obtain the group-A total 9; likewise group B combines S_1's trailing 5 with S_2's $\{1, 6\}$ for a total 12. DAggUp accumulates these partials up the tree and DAggDown propagates each group's total back down to every server holding part of the group; group C lies entirely within S_3 and needs no exchange. Only the constant-size boundary nodes are communicated ($\log E$ rounds of $O(1)$ messages).

Phase 4 (second local aggregation, Lines 29–33). Each server re-runs AggTree locally, now seeded with the cross-boundary totals, and writes each tuple's group sum: $S_0 \to (9, 9)$, $S_1 \to (9, 12)$, $S_2 \to (12, 12)$, $S_3 \to (9, 9)$. The final array assigns every tuple the total of its group, as shown in the last row of the table.

$R \leftarrow$ DISCO-HybridSort(R, E, C)

1: Let G be the num. of servers in each cluster
2: Let $B' \leftarrow B/C + K$ ▷$K \leftarrow$ *computation key size*
3: **for** each server $E_k \in E$ **do**
4: Receive N/G shards each of size B'
5: $(Z, \epsilon, P) \leftarrow \text{Cost}(N, G)$
6: $Bu \leftarrow (1 + \epsilon) \cdot N/Z$
7: **for** each cluster C, $\forall E_k \in C$ **do**
8: Let E_k^i denote the i-th bucket of E_k
9: Initialize $\ell \leftarrow 1$
10: **while** $\ell < G$ **do**
11: $k \leftarrow P[\ell]$
12: **for** $i \in Bu/k$ **do**
13: Let $bu = \{E_k^i, E_k^{i+2^\ell} ..., E_k^{i+k\cdot 2^\ell}\}$
14: $bu' \leftarrow k$-MergeSplit(bu, k)
15: $\ell \leftarrow \ell \cdot k$
16: **for** each cluster C, $\forall E_k \in C$ **do**
17: **for** $j \in [Bu]$ **do**
18: $j' \leftarrow j \mod G$
19: Route $E_k^{j'}$ bucket to the j-th server of C
20: **for** each cluster C, $\forall E_k \in C$ **do**
21: Let E_k^i denote the i-th bucket of E_k
22: **for** the remaining layers ℓ **do**
23: $k \leftarrow P[\ell]$
24: **for** $i \in Bu/k$ **do**
25: Let $bu = \{E_k^i, E_k^{i+2^\ell} ..., E_k^{i+k\cdot 2^\ell}\}$
26: $bu' \leftarrow k$-MergeSplit(bu, k)
27: **for** every bucket E_k^i in server E_k **do**
28: $E_k^i \leftarrow$ OSort(E_k^i)
29: **for** each cluster C **do**
30: Non-oblivious sort R across all $E_k \in G$
31: **for** each server $E_k \in E$ **do**
32: Send the resulting N/G shards

Figure 13. Pseudocode for multi-server Hybrid Sorting.

B Details of Distributed Expansion

Expansion is a fundamental primitive used in non-foreign key joins (Section 6) to duplicate tuples according to an auxiliary table D. Given an input table R of size N and an auxiliary table D where $D[i]$ specifies the number of copies to produce for tuple $R[i]$, the expansion algorithm generates

$R \leftarrow$ DISCO-Compaction(R, D)

$B' \leftarrow B/E + 1$ ▷*1 bit for compaction*
/* **Each server partitions their tuples***/
for each server $E_k, k \in [E]$ **do**
  Let S_{E_k} be an array of $E \cdot n$ shards.
  parallel for $i \in [n]$ **do**
    for $j \in [E]$ **do**
      Let $s_{i,j}$ be the j-th shard of $R[i]$.
      $S_{E_k}[i + j \cdot n] \leftarrow s_{i,j}$ ▷*Also copy M[i]*
/* **Exchange segments between servers** */
for each pair (E_k, E_m) of servers **do**
  Swap seg_m of S_{E_k} with seg_k of S_{E_m}
/* **Locally execute ORCompact** */
for each server $E_k, k \in [E]$ **do**
  $S_{E_k} \leftarrow$ ORCompact(S_{E_k}, M)
/* **Exchange segments between servers** */
for each pair (E_k, E_m) pair of servers **do**
  Swap seg_m of S_{E_k} with seg_k of S_{E_m}
/* **Restructure original tuples** */
for each server $E_k, k \in [E]$ **do**
  parallel for $i \in [n]$ **do**
    for $j \in [E]$ **do**
      Copy data of $s_{i,j}$ to tuple $R[i]$.
return R

Figure 14. Pseudocode for Distributed Oblivious Compaction

DAggUp$_{\langle \text{TYPE, OP} \rangle}(A, B)$:
where TYPE $\in$ {prefix,suffix,full}, and OP $\in$ {+,Dup}

parallel for $i \leftarrow [1 : \frac{n}{2^h}]$ **do**
  $\langle lpx_i, px_i, rsx_i, sx_i, \ell_i, c_i \rangle \leftarrow A[h, i]$
  $lpx_i \leftarrow px_0$, $rsx_i \leftarrow sx_1$
  $l_i \leftarrow l_0, c_i \leftarrow c_0 \wedge c_1$
  $px \leftarrow \neg c_1 \cdot px_1 + c_1 \cdot (px_0 \bullet px_1)$
  $sx \leftarrow \neg c_0 \cdot sx_0 + c_0 \cdot (sx_0 \bullet sx_1)$

DAggDown$_{\langle \text{TYPE, OP} \rangle}(A, B)$:

parallel for $i \leftarrow [1 : n/2^h]$ **do**
  $\langle lpx_i, px_i, rsx_i, sx_i, \ell_i, c_i \rangle \leftarrow A[h, i]$
  $px_0 \leftarrow px_i$
  $px_1 \leftarrow l_1 \cdot \neg c_0 \cdot lpx_i + l_1 \cdot c_0 \cdot (lpx_i \bullet px_i)$
  $sx_0 \leftarrow c_1 \cdot (rsx_i \bullet sx_i) + \neg c_1 \cdot rsx_i$
  $sx_1 \leftarrow sx_i$
  $A[h, i]_0 \leftarrow \langle px_0, sx_0 \rangle$
  $A[h, i]_1 \leftarrow \langle px_1, sx_1 \rangle$

Figure 15. Oblivious aggregation for distributed upstream and downstream update.

$R \leftarrow$ DISCO-Expansion(R, D)

$B' \leftarrow B/E + K$ ▷*Size of expansion's computation key in bits*
/* **Each server partitions their tuples into shards** */
for each server $E_k, k \in [E]$ **do**
  Let S_{E_k} be an array of $E \cdot n$ shards of size B' each
  parallel for $i \in [n]$ **do**
    for $j \in [E]$ **do**
      Let $s_{i,j}$ be the j-th shard of $R[i]$.
      $S_{E_k}[i + j \cdot n] \leftarrow s_{i,j}$ ▷*Also copy Q[i]*
/* **Exchange segments between servers** */
for each pair (E_k, E_m) of servers **do**
  Swap seg_m of S_{E_k} with seg_k of S_{E_m}
/* **Locally execute Expansion** */
for each server $E_k, k \in [E]$ **do**
  $G \leftarrow 0||1^{n-1}$
  $Q \leftarrow$ AggTree$(S_{E_k}, D, G,$ Prefix, $sum)$
  $D_{out} \leftarrow Q[n-1] + D[n-1]$
  Pad S_{E_k} to output size D_{out} with dummies
  $S_{E_k} \leftarrow$ OExpand(S_{E_k}, Q)
/* **Exchange segments between servers** */
for each pair (E_k, E_m) pair of servers **do**
  Swap seg_m of S_{E_k} with seg_k of S_{E_m}
/* **Restructure original tuples** */
for each server $E_k, k \in [E]$ **do**
  parallel for $i \in [n]$ **do**
    for $j \in [E]$ **do**
      Copy the data of the j-th shard to tuple $R[i]$.
return R

Figure 16. Pseudocode for Expansion

an output table of size $M = \sum_i D[i]$ such that each tuple $R[i]$ is placed at consecutive positions starting from its target index $Q[i] = \sum_{j<i} D[j]$. In the context of joins, after the cardinality exchange phase determines how many times each tuple of the left and right tables must be replicated (Step 4 of Figure 6), expansion materializes these copies so that the subsequent alignment phase can pair them correctly.

The pseudocode for our distributed expansion is given in Figure 16. The algorithm proceeds in three phases: (a) shard generation and distribution, (b) local expansion, and (c) shard collection and tuple reconstruction. We adopt vertical partitioning, which is optimal for expansion since the computation key (the original tuple index and destination index $Q[i]$) requires only $O(\log N)$ bits, making the per-shard overhead negligible while eliminating all inter-server communication during the expensive local expansion phase. We detail each

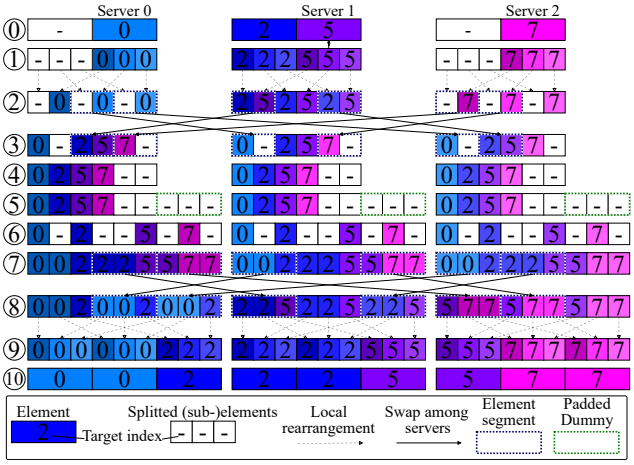

Figure 17. Distributed Oblivious Expansion illustrated in a horizontally partitioned model.

$bit \leftarrow \mathbf{Real}_{\mathcal{A},\Pi}(1^\lambda)$

1: Let N be a public parameter that denotes the input table's size.
2: op $\leftarrow \mathcal{A}(1^\lambda)$
3: $Res \leftarrow$ DICSO-Operator$_{op}(1^\lambda)$
4: $bit \leftarrow \mathcal{A}(1^\lambda, Res)$
5: **return** bit

$bit \leftarrow \mathbf{Ideal}_{\mathcal{A},\mathcal{L},\Pi}(1^\lambda)$

1: Let N be a public parameter that denotes the input table's size.
2: op $\leftarrow \mathcal{A}(1^\lambda)$
3: $Res \leftarrow$ DICSO-SimOperator$_{op}(\mathcal{L}, 1^\lambda)$
4: $bit \leftarrow \mathcal{A}(1^\lambda, Res)$
5: **return** bit

Figure 18. Real vs. Ideal experiment on DISCO-Operators, where the operators are Compaction, Expansion, Aggregation Tree, and Hybrid-Sort.

phase below. The formal proof of obliviousness is provided in Section C.1 (Theorem 2).

Algorithm details. (a) In the first phase, we generate the shards. Since expansion is based on the auxiliary array Q, each shard must retain the original index of its tuple so that every server can compute the correct output independently. An example is shown in Figure 17, where $N = 6$, $E = 3$, and table R is horizontally partitioned across servers. Step 1 shows shard generation, and Step 2 shows how these shards are locally placed into segments. These segments are then distributed across servers in the same manner as in compaction (Lines 3-11). (b) Next, each server computes the output size M by performing a prefix sum aggregation tree on the D table (Lines 14-16). Then, proceeds to pad its local array R_{E_k} to the output size with dummies (Line 17). Afterwards, each server executes the scalable expansion algorithm introduced in [73] as a black box (Line 18). By the end of this phase, all tuples in R_{E_k} for every server $E_k \in E$ will be correctly placed at their target indices as defined by Q (see Step 6 of Figure 17). Notably, each server's shards are aligned at the same positions. More formally, if a shard $s_{t,k}$ of the original tuple t appears at position j on server E_k, then all corresponding shards $s_{t,m}$ on servers E_m ($\forall m \in [E], s.t. m \neq k$) will also appear at position j. (c) Finally, the last phase gathers all shards of each original tuple onto a single server and reconstructs the original tuples. This is feasible due to the deterministic nature of the expansion process (Lines 20-25).

C Analysis Table

In Table 2, we present the asymptotic computation and communication complexities of our approach compared to state-of-the-art methods. The Bitonic Sort (V) and Bucket Sort (H) are the vertical approach for bitonic sort (Section 4) and distributed bucket sort with the optimizations (Section 5.3), respectively. The initial and final data distribution–i.e., distributing the data to the servers and collecting the results–is marked in red. As noted in our experimental section, we exclude this communication cost from all approaches, as we assume the data is already distributed. Note that for compaction, since there is no other distributed compaction in the literature, we compare our approach with the naive extension of [82], as denoted in Section 5.1. Similarly, for the aggregation tree, since there is no other distributed aggregation tree in the literature, we compare against the centralized aggregation performed during joins in the implementation of Jodes [98]. For certain methods, such as Opaque (for filter, aggregation, and foreign-key join) and Jodes (for non-foreign-key join), we provide expanded computation complexities to highlight the presence of sequential operations. In contrast, our approaches are fully parallel at every stage of execution.

C.1 Security Analysis.

Definition 1. *An oblivious operator* Π *is secure if, for any non-uniform probabilistic polynomial-time (PPT) adversary* $\mathcal{A}$*, there exists a PPT Simulator (Sim) such that* $\left| Pr\left[\mathbf{Real}_{\Pi}^{\mathcal{A}}(\lambda) = 1\right] - Pr[\mathbf{Ideal}_{Sim}^{\mathcal{L},\mathcal{A}}(\lambda) = 1\right| \leq negl(\lambda)$*, where* λ *is the security parameter.*

In this section we provide simulators for our oblivious primitives that follow Definition 1 and prove their obliviousness. We start with a generic Real vs. Ideal game in Figure 18, where the subroutines for DISCO-Operators and DISCO-Simulator in Figure 19.

Distributed Compaction

Theorem 1. *The algorithm in Figure 14 is oblivious as defined in Definition 1 in Section 2 with respect to the Real/Ideal game in Figure 18, where* $\mathcal{L} = \perp$.

Operation	Our Approach		Prior Work		Ref.
	Computation	Communication	Computation	Communication	
Bitonic Sort (V)	$O\left(\frac{n\log^2 n}{p}\right)$	$2N$	$O\left(\frac{n\log^2 n}{p}\right)$	$2N + N\log E$	[77]
Bucket Sort (H)	$O\left(\frac{n\log n}{p}\right)$	$2N + (1+\epsilon)N + N$	$O\left(\frac{n\log n}{p}\right)$	$2N + 3N$	[77]
Compaction	$O\left(\frac{n\log n}{p}\right)$	$2N$	$O\left(\frac{n\log n}{p}\right)$	$2N + N\log E$	[82]
Expansion	$O\left(\frac{n\log n}{p}\right)$	$N + M$	$O\left(m\log n + min(m,N)\log^2 N\right)$	$N + M + N + min(M, NE)$	[98]
Agg. Tree	$O\left(\frac{n}{p}\right)$	$2N + O(1)\log E$	$O(n+E)$	$O(2N + O(E))$	[98]
Filter	$O\left(\frac{n\log n}{p}\right)$	$2N + (1+\epsilon)N + 3N$	$O\left(\frac{n\log^2 n}{p} + n\right)$	$2N + 6N$	[103]
Aggregation	$O\left(\frac{n\log n}{p}\right)$	$2N + (1+\epsilon)N + 3N + O(\log E)$	$O\left(\frac{n\log^2 n}{p} + n\right)$	$2N + 6N$	[103]
FK Join	$O\left(\frac{n\log n}{p}\right)$	$2N + (1+\epsilon)N + 3N$	$O\left(\frac{n\log^2 n}{p} + n\right)$	$2N + 6N$	[103]
NFK Join	$O\left(\frac{(n+m)\log n}{p}\right)$	$N + M + N(2+\epsilon) + N + M + O(1)\log E + M/2(2+\epsilon)$	$O\left(\frac{(n+m)\log^2 n}{p} + m\log m\right)$	$N + M + 7N + 2M + \min(2M, NE)$	[98]

Table 2. Computation and communication complexity comparison between our approach and prior work for various database operators. Here, N denotes input size, M denotes output size, p denotes number of processors, E denotes number of enclave instances, n denotes the number of input tuples that each server holds, while m denotes the number of output tuples on each server, and ϵ represents a small constant factor introduced by the cost function (see Section 5.3).

Proof. To prove that the algorithm is oblivious, we construct a simulator, depicted in Figure 20, and establish indistinguishability between the real and ideal executions. In particular, we show that the sequence of computations and memory accesses depends only on public parameters (e.g., the input size) and is independent of the underlying data values.

Our proof follows a standard Real/Ideal paradigm, illustrated in Figure 18. In the real execution, the protocol invokes the general functionality composed of the oblivious primitives introduced in this paper. In the ideal execution, a simulator emulates the adversary's view without access to the real input.

More concretely, given only the input size n, the simulator (Figure 19) generates a uniformly random array R of size n, together with an auxiliary table M that marks randomly selected entries of R. The real protocol and the simulator then execute DISCO-Compaction and DISCO-SimCompaction, respectively.

Both executions proceed identically with respect to their access patterns. First, each server partitions its tuples into shards and embeds the corresponding computation key into each shard. The servers then exchange their segments so that each server receives the appropriate shards from all other servers. Next, the real protocol performs a local invocation of ORCompact [82], while the simulator invokes SimORCompact. Since ORCompact has been proven oblivious in [82], and a corresponding simulator is provided therein, this step leaks no information.

Subsequently, both executions perform a second broadcast phase, redistributing their segments among the servers according to the protocol specification. Finally, the real execution reconstructs the original input tuples, whereas the simulator reconstructs tuples from the randomly generated array R, producing an output Res of size n.

Because each step of the simulator faithfully reproduces the communication pattern, memory access sequence, and control flow of the real protocol, while operating on uniformly random data, it follows that the adversary's view in the real and ideal executions is computationally indistinguishable. Therefore, the algorithm is oblivious.

□

Distributed Expansion

Theorem 2. *The algorithm in Figure 16 is oblivious as defined in Definition 1 in Section 2 with respect to the Real/Ideal game in Figure 18, where $\mathcal{L} = D_{out}$, the output size of the expansion.*

Proof. To prove that the algorithm is oblivious, we construct a simulator, depicted in Figure 21, and establish indistinguishability between the real and ideal executions. In particular, we show that the sequence of computations and

$Res \leftarrow$ DISCO-Operator$_{op}(1^\lambda)$

Let N, E, B, C be public parameters that denote the size of the input, the number of servers, and the block size, and the number of clusters, respectively.
switch *op* **do**
  case Compaction:
    $R, M \leftarrow \mathcal{A}(1^\lambda, N)$
    $Res \leftarrow$ DISCO-Compaction(R, M)
  case Expansion:
    $R, D \leftarrow \mathcal{A}(1^\lambda, N)$
    $Res \leftarrow$ DISCO-Expansion(R, D)
  case Aggregation Tree:
    $R, G, f, p \leftarrow \mathcal{A}(1^\lambda, N)$
    $Res \leftarrow$ DISCO-AggTree(R, G, f, p)
  case Hybrid-Sort:
    $R, E, C \leftarrow \mathcal{A}(1^\lambda, N)$
    $Res \leftarrow$ DISCO-HybridSort(R, E, C)
return *Res*

$Res \leftarrow$ DISCO-SimOperator$_{op}(\mathcal{L}, 1^\lambda)$

Let N, E, B, C be public parameters that denote the size of the input, the number of servers, and the block size, and the number of clusters, respectively.
switch *op* **do**
  case Compaction:
    $R, M \leftarrow \mathcal{A}(1^\lambda, N)$
    $Res \leftarrow$ DISCO-SimCompaction(R, M)
  case Expansion:
    Parse $\mathcal{L}$ to retrieve the D_{out}
    $R, D \leftarrow \mathcal{A}(1^\lambda, N, D_{out})$
    $Res \leftarrow$
      DISCO-SimExpanion(R, D, D_{out})
  case Aggregation Tree:
    $R, G, f, p \leftarrow \mathcal{A}(1^\lambda, N)$
    $Res \leftarrow$ DISCO-SimAggTree(R, G, f, p)
  case Hybrid-Sort:
    $R, E, C \leftarrow \mathcal{A}(1^\lambda, N)$
    $Res \leftarrow$ DISCO-SimSort(R, E, C)
return *Res*

Figure 19. Functions that given the appropriate operator, generates the inputs and calls the desired function.

memory accesses depends only on public parameters (e.g., the input size and the output size) and is independent of the underlying data values. As in Theorem 1, we follow the standard Real/Ideal paradigm, illustrated in Figure 18. In the real execution, the protocol invokes the general functionality composed of the oblivious primitives introduced in this paper. In the ideal execution, a simulator emulates the adversary's view without access to the real input. More concretely, given the input size n and the output size D_{out} (obtained

$R \leftarrow$ DISCO-SimCompaction$(\mathcal{L})$

Let N, E, B be public parameters that denote the size of the input, the number of servers, and the block size, respectively.
Let $B' \leftarrow B/E + 1, n \leftarrow N/E$
/* **Each server randomly generates shards** */
for each server $E_k, k \in [E]$ **do**
  Let S_{E_k} be an empty array of $E \cdot n$ shards.
  parallel for $i \in [n]$ **do**
    for $j \in [E]$ **do**
      Let $s_{i,j}$ be the j-th shard of $R[i]$.
      $S_{E_k}[i + j \cdot n] \leftarrow s_{i,j}$ ▷*Also copy M[i]*
/* **Exchange segments between servers** */
for each pair (E_k, E_m) of servers **do**
  Swap seg_m of S_{E_k} with seg_k of S_{E_m}
/* **Locally execute SimORCompact** */
for each server $E_k, k \in [E]$ **do**
  $S_{E_k} \leftarrow$ SimORCompact(S_{E_k})
/* **Exchange segments between servers** */
for each pair (E_k, E_m) pair of servers **do**
  Swap seg_m of S_{E_k} with seg_k of S_{E_m}
/* **Restructure original random tuples** */
for each server $E_k, k \in [E]$ **do**
  parallel for $i \in [n]$ **do**
    for $j \in [E]$ **do**
      Copy data of $s_{i,j}$ to tuple $R[i]$.
return R

Figure 20. Simulator to prove the obilviousness of DISCO-Compaction.

from the leakage profile), the simulator (Figure 19) generates a uniformly random array R of size n, together with the auxiliary table D. The real protocol and the simulator then execute DISCO-Expansion and DISCO-SimExpansion, respectively.

Both executions proceed identically with respect to their access patterns. First, each server partitions its tuples into shards and embeds the corresponding computation key into each shard. The servers then exchange their segments so that each server receives the appropriate shards from all other servers. Next, the real protocol performs a local invocation of AggTree [73] to locally compute prefix sums for the target output indices, pads the local array to the output size D_{out}, and invokes OExpand [73] to obliviously push each tuple to its target position. The simulator performs the same sequence using SimAggTree and SimOExpand. Since AggTree and OExpand have been proven oblivious in [73], and corresponding simulators are provided therein, these steps leak no information beyond the output size. Subsequently, both executions perform a second broadcast phase, redistributing their segments among the servers according to the protocol specification. Finally, the real execution reconstructs the

$R \leftarrow$ DISCO-SimExpansion(R, D, D_{out})

$B' \leftarrow B/E + K$ ▷*Size of expansion's computation key in bits*
/* **Servers partition their tuples to shards** */
for each server $E_k, k \in [E]$ **do**
  Let S_{E_k} be an array of $E \cdot n$ shards of size B'
  parallel for $i \in [n]$ **do**
    for $j \in [E]$ **do**
      Let $s_{i,j}$ be the j-th shard of $R[i]$.
      $S_{E_k}[i + j \cdot n] \leftarrow s_{i,j}$ ▷*Also copy Q[i]*
/* **Exchange segments between servers** */
for each pair (E_k, E_m) of servers **do**
  Swap seg_m of S_{E_k} with seg_k of S_{E_m}
/* **Locally execute Expansion** */
for each server $E_k, k \in [E]$ **do**
  $G \leftarrow 0||1^{n-1}$
  $Q \leftarrow$
    SimAggTree$(S_{E_k}, D, G,$ Prefix, $sum)$
  Pad S_{E_k} to output size D_{out} with dummies
  $S_{E_k} \leftarrow$ SimOExpand(S_{E_k}, Q)
/* **Exchange segments between servers** */
for each pair (E_k, E_m) pair of servers **do**
  Swap seg_m of S_{E_k} with seg_k of S_{E_m}
/* **Restructure original tuples** */
for each server $E_k, k \in [E]$ **do**
  parallel for $i \in [n]$ **do**
    for $j \in [E]$ **do**
      Copy the data of the j-th shard to tuple $R[i]$.
return R

Figure 21. Simulator to prove the obliviousness of DISCO-Expansion.

original output tuples, whereas the simulator reconstructs tuples from the randomly generated array R, producing an output Res of size D_{out}.

Because each step of the simulator faithfully reproduces the communication pattern, memory access sequence, and control flow of the real protocol, while operating on uniformly random data, it follows that the adversary's view in the real and ideal executions is computationally indistinguishable. The only information revealed is the output size D_{out}, which is captured by the leakage profile. Therefore, the algorithm is oblivious.

□

Distributed Sort

Theorem 3. *The algorithm in Figure 13 is oblivious as defined in Definition 1 in Section 2 with respect to the Real/Ideal game in Figure 18, where* $\mathcal{L} = \perp$.

Proof. To prove that the algorithm is oblivious, we construct a simulator, depicted in Figure 22, and establish indistinguishability between the real and ideal executions. In particular, we show that the sequence of computations and memory accesses depends only on public parameters (e.g., the input size, the number of servers, and the number of clusters) and is independent of the underlying data values.

Our proof follows a standard Real/Ideal paradigm, illustrated in Figure 18. In the real execution, the protocol invokes the general functionality composed of the oblivious primitives introduced in this paper. In the ideal execution, a simulator emulates the adversary's view without access to the real input.

More concretely, given only the input size N, the number of servers E, and the number of clusters C, the simulator (Figure 19) generates a uniformly random array R of size N. The real protocol and the simulator then execute DISCO-HybridSort and DISCO-SimSort, respectively.

Both executions proceed identically with respect to their access patterns. First, each server receives N/G shards of size B', where G is the number of servers per cluster. This initial distribution is deterministic and depends solely on N, E, and C. The cost function then computes the parameters (Z, ϵ, P) from the public values N and G, determining the bucket size, the number of dummies, and the branching factor at each layer of the butterfly network. Next, each cluster independently executes the initial layers of the butterfly network. The real protocol invokes k-MergeSplit [48], while the simulator invokes Sim-k-MergeSplit. Since k-MergeSplit has been proven oblivious in [48], and the number of layers, branching factor k at each layer, and bucket sizes are all derived from the cost function and are independent of the data, these steps leak no information. Subsequently, both executions route the j-th bucket to its designated server within each cluster. This communication pattern is fixed and depends only on the bucket index and the cluster structure, not on the data. Each server then proceeds with the remaining butterfly network layers, again invoking k-MergeSplit (resp. Sim-k-MergeSplit), followed by oblivious sorting within each bucket. The real protocol invokes OSort [77], while the simulator invokes SimOSort. Since OSort has been proven oblivious in [77], this step leaks no information. Together, the butterfly network and the local oblivious sort constitute an Oblivious Random Permutation (ORP). After the ORP, as established in [9], all ties with prior access patterns are broken. This permits the use of a non-oblivious sorting algorithm (MergeSort) to produce the final sorted output. Both the real and simulated executions perform the same deterministic non-oblivious sort on data that is uniformly randomly permuted from the adversary's perspective, and thus this step reveals no additional information. Finally, each server sends the resulting N/G shards, which is a deterministic communication step.

Because each step of the simulator faithfully reproduces the communication pattern, memory access sequence, and control flow of the real protocol, while operating on uniformly random data, it follows that the adversary's view in the real and ideal executions is computationally indistinguishable. Therefore, the algorithm is oblivious.

□

$R \leftarrow$ DISCO-SimSort(R, E, C)

Let G be the num. of servers in each cluster
Let $B' \leftarrow B/C + K$ ▷$K \leftarrow$ *computation key size*
for each server $E_k \in E$ **do**
  Receive N/G shards each of size B'
$(Z, \epsilon, P) \leftarrow \text{Cost}(N, G)$
$Bu \leftarrow (1+\epsilon) \cdot N/Z$
for each cluster C, $\forall E_k \in C$ **do**
  Let E_k^i denote the i-th bucket of E_k
  Initialize $\ell \leftarrow 1$
  while $\ell < G$ **do**
    $k \leftarrow P[\ell]$
    for $i \in Bu/k$ **do**
      Let $bu = \{E_k^i, E_k^{i+2^\ell} ..., E_k^{i+k\cdot 2^\ell}\}$
      $bu' \leftarrow$ Sim-k-MergeSplit(bu, k)
    $\ell \leftarrow \ell \cdot k$
for each cluster C, $\forall E_k \in C$ **do**
  for $j \in [Bu]$ **do**
    $j' \leftarrow j \mod G$
    Route $E_k^{j'}$ bucket to the j-th server of C
for each cluster C, $\forall E_k \in C$ **do**
  Let E_k^i denote the i-th bucket of E_k
  for the remaining layers ℓ **do**
    $k \leftarrow P[\ell]$
    for $i \in Bu/k$ **do**
      Let $bu = \{E_k^i, E_k^{i+2^\ell} ..., E_k^{i+k\cdot 2^\ell}\}$
      $bu' \leftarrow$ Sim-k-MergeSplit(bu, k)
  for every bucket E_k^i in server E_k **do**
    $E_k^i \leftarrow$ SimOSort(E_k^i)
for each cluster C **do**
  Non-oblivious sort R across all $E_k \in G$
for each server $E_k \in E$ **do**
  Send the resulting N/G shards

Figure 22. Simulator to prove the obliviousness of DISCO-HybridSort.

Distributed Aggregation Tree

Theorem 4. *The algorithm in Figure 5 is oblivious as defined in Definition 1 in Section 2 with respect to the Real/Ideal game in Figure 18, where* $\mathcal{L} = \perp$.

Proof. To prove that the algorithm is oblivious, we construct a simulator, depicted in Figure 23, and establish indistinguishability between the real and ideal executions. In particular, we show that the sequence of computations and memory accesses depends only on public parameters (e.g., the input size and the number of servers) and is independent of the underlying data values.

Our proof follows a standard Real/Ideal paradigm, illustrated in Figure 18. In the real execution, the protocol invokes the general functionality composed of the oblivious primitives introduced in this paper. In the ideal execution, a simulator emulates the adversary's view without access to the real input.

More concretely, given only the input size n, the simulator (Figure 19) generates a uniformly random array R of size n, together with a random group array G, an aggregation function f, and an operation type p. The real protocol and the simulator then execute DISCO-AggTree and DISCO-SimAggTree, respectively.

Both executions proceed identically with respect to their access patterns. The algorithm consists of four phases. In Phase 1, each server locally computes the aggregation function on its local array. The real protocol invokes AggTree [73], while the simulator invokes SimAggTree. Since AggTree has been proven oblivious in [73], and a corresponding simulator is provided therein, this step leaks no information. Each server then initializes an array of aggregation nodes of size $\log E$, storing the aggregated values and group bits of its first and last tuples. In Phase 2 (Distributed Upstream), servers exchange constant-size aggregation nodes across $\log E$ layers. At each layer, a sender E_S transmits its aggregation node to a recipient E_R, and the recipient updates its own node. The real protocol invokes DAggUp, while the simulator invokes SimDAggUp. The communication pattern is fully deterministic: exactly $\log E$ rounds, each consisting of $O(1)$ messages of constant size. The sender–receiver pairs at each layer are predetermined by the tree structure and depend only on E. Moreover, DAggUp and SimDAggUp perform identical fixed arithmetic operations on the aggregation nodes, regardless of the underlying data values. In Phase 3 (Distributed Downstream), servers propagate prefix and suffix values across $\log E$ layers in the reverse direction. The real protocol invokes DAggDown, while the simulator invokes SimDAggDown. As in Phase 2, the communication pattern and the operations performed are entirely deterministic and data-independent. In Phase 4, each server performs a second round of local aggregation to incorporate the information received from other servers. The real protocol invokes AggTree, while the simulator invokes SimAggTree. As in Phase 1, this step leaks no information.

Because each step of the simulator faithfully reproduces the communication pattern, memory access sequence, and control flow of the real protocol, while operating on uniformly random data, it follows that the adversary's view

$(R, G) \leftarrow$ DISCO-SimAggTree(R, G, f, p)

1: **for** each server $E_k \in E$ **do**
2: /* **First distribution of the elements** */
3: Retrieve R_{E_k} and G_{E_k}
4: /* **Phase 1 – Local aggregation** */
5: $Res \leftarrow$SimAggTree(R_{E_k}, G_{E_k}, f, p)
6: Initialize A with $\log E$ empty nodes
7: $A[\log E{-}1] \leftarrow (res[0], G[0], res[\text{n-}1], G[\text{n-}1])$
8: /* **Phase 2 – Distributed Upstream** */
9: **for** $L \leftarrow \log E - 1, ..., 1$ **do**
10: **for** each pair (E_R, E_S) on this layer **do**
11: Sender E_S sends $A^{L-1}_{E_S}$ to recipient E_R
12: $A^{L}_{E_R} \leftarrow$ SimDAggUp$(A^{L-1}_{E_R}, A^{L-1}_{E_S})$
13: /* **Phase 3 – Distributed Downstream** */
14: **for** $L \leftarrow 1, ..., \log E - 1$ **do**
15: **for** each pair (E_R, E_S) on this layer **do**
16: Sender E_S sends $A^{L}_{E_S}$ to recipient E_R
17: $A^{L-1}_{E_S} \leftarrow$ SimDAggDown$(A^{L}_{E_R}, A^{L-1}_{E_S})$
18: **for** each server $E_k \in E$ **do**
19: /* **Phase 4 – Second Local aggregation** */
20: $Res \leftarrow$SimAggTree(R_{E_k}, G_{E_K}, f, p)
21: Send back the result (R_{E_k}, G_{E_k})
22: **return** (R, G)

Figure 23. Simulator to prove the obliviousness of DISCO-AggTree.

in the real and ideal executions is computationally indistinguishable. Therefore, the algorithm is oblivious.

□

Here, we include the pseudocodes for the subroutines DAggUP and DAggDown, executed as part of our distributed oblivious aggregation tree (see Figure 5).

C.2 Asymptotic Performance and Security for Oblivious Filter & Aggregation

Efficiency. We begin with oblivious filter. The proposed algorithm employs a local parallel scan that runs in linear time, followed by a vertical compaction phase where each server computes execution time on shards of blocks rather than on entire blocks. This yields a time complexity of $O(N \cdot \frac{B}{E} \log N)$ per server, independently. As a result, our method achieves better complexity than prior approaches. Moreover, it ensures linear communication overhead, transferring only $2 \cdot N \cdot B$ bits. Every step of our algorithm is fully parallel, improving upon earlier works that rely on sequential scans and other non-parallel primitives. For example, Opaque uses sequential scans to mark tuples, while SODA combines sequential scans with a non-parallel distribution algorithm, originally introduced in [62], to implement its pseudorandom communication strategy.

SimDAggUp$_{\langle \text{TYPE, OP} \rangle}(A, B)$:
where TYPE $\in$ {prefix,suffix,full}, and OP $\in$ {+,Dup}

1: **parallel for** $i \leftarrow [1 : \frac{n}{2^h}]$ **do**
2: $\langle lpx_i, px_i, rsx_i, sx_i, \ell_i, c_i \rangle \leftarrow A[h, i]$
3: $lpx_i \leftarrow px_0$, $rsx_i \leftarrow sx_1$
4: $l_i \leftarrow l_0, c_i \leftarrow c_0 \wedge c_1$
5: $px \leftarrow \neg c_1 \cdot px_1 + c_1 \cdot (px_0 \bullet px_1)$
6: $sx \leftarrow \neg c_0 \cdot sx_0 + c_0 \cdot (sx_0 \bullet sx_1)$

SimDAggDown$_{\langle \text{TYPE, OP} \rangle}(A, B)$:

1: **parallel for** $i \leftarrow [1 : n/2^h]$ **do**
2: $\langle lpx_i, px_i, rsx_i, sx_i, \ell_i, c_i \rangle \leftarrow A[h, i]$
3: $px_0 \leftarrow px_i$
4: $px_1 \leftarrow l_1 \cdot \neg c_0 \cdot lpx_i + l_1 \cdot c_0 \cdot (lpx_i \bullet px_i)$
5: $sx_0 \leftarrow c_1 \cdot (rsx_i \bullet sx_i) + \neg c_1 \cdot rsx_i$
6: $sx_1 \leftarrow sx_i$
7: $A[h, i]_0 \leftarrow \langle px_0, sx_0 \rangle$
8: $A[h, i]_1 \leftarrow \langle px_1, sx_1 \rangle$

Figure 24. Simulation of upstream and downstream phases of the distributed aggregation tree.

For aggregation, both approaches we discussed are bottlenecked by the sorting operator, resulting in an overall asymptotic complexity of $O(N \log N)$. The communication costs for the two approaches are $2 \cdot N \cdot B + O(1) \cdot \log E$ bits and $2 \cdot N \cdot B$ bits, respectively. Both approaches are expected to outperform existing methods. In particular, Opaque performs two global sorts and relies on sequential scans, which leads to slower execution. SODA [66], in turn, requires two rounds of sequential scans for aggregation, two local sorts per server, and a non-parallel distribution algorithm. In contrast, both of our proposed approaches are fully parallel at every step, offering a significant performance improvement over prior work.

Security. To prove the security of our oblivious filtering approach, we can simply breakdown the steps of the algorithm. The first step is to perform a parallel scan that depends on the size of each server's local array n, which is public information. Then, the distributed oblivious compaction is a direct execution of the building block described in Section 5.1. The leakage of this algorithm is the output size M of the filter, i.e., the number of tuples that satisfy the query's predicate, $\mathcal{L} = M$. As discussed in Section 2, leaking such information is unavoidable without incurring significant performance overhead. To avoid worst-case padding, one can use adjustable padding [28].

For aggregation, both approaches described are oblivious, as long as all underlying primitives are themselves oblivious. Specifically, the first approach is oblivious because the global sort is proven oblivious C.1, the aggregation tree is shown to be oblivious in Section 5.2, and the distributed compaction is

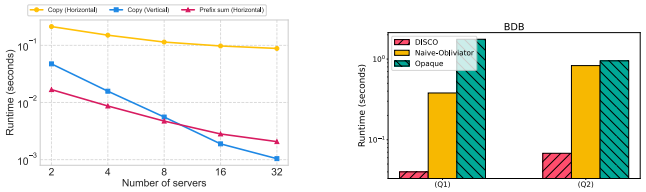

Figure 25. Performance of duplication and prefix sum based on horizontal and vertical aggregation tree (left), and our performance for Big Data Benchmark (right).

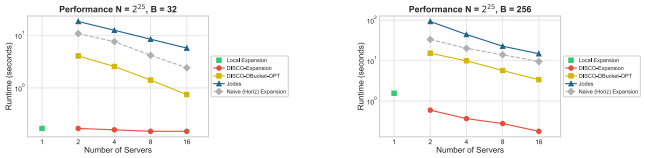

Figure 26. Performance of our expansion against prior works on smaller block sizes $B \in \{32, 256\}$B.

analyzed in Section 5.1. The computations, access patterns, and communication for each of these components depend only on the number of servers E and the number of tuples N. Similarly, the second approach maintains obliviousness since the communication and shard generation are based solely on E, and the aggregation algorithm we use is proven to be oblivious in [73]. In both cases, the only information leaked is the output size $\mathcal{L} = M$, i.e., the number of groups in the table. As discussed in Section 6, if hiding the output size is required, the client must fetch the entire database, or use adjustable padding as mentioned in [28].

C.3 Additional Experiments

Aggregation Tree We evaluate the performance of our distributed oblivious aggregation tree using 2^{25} elements of size 4KB, as shown in Figure 25. For duplication tasks (i.e., copying data), the vertical approach consistently outperforms the horizontal one across all configurations, from 2 to 32 servers—achieving speedups of 4.49× and 84.90×, respectively. For summation, the horizontal approach is superior to vertical partitioning on small number of servers (2-8 servers) but becomes 2× slower on more machines.

BDB Filter & Aggregation. Here we include the experiments mentioned in Section 8 on BDB-Q1 and BDB-Q2 [6] to show the performance of our filter, and aggregation query. As aforementioned, we denote as naive obliviator, the approach that is introduced in [73] with the trivial extension of compaction to the distributed setting. Our algorithms are 9x - 43x faster than the competitors. In detail, our filter (Q1) is 9.46x and 43.84x faster than the naive approach of Obliviator and Opaque. It is mainly due to our efficient distributed oblivious compaction. The aggregation query of DISCO (Q2) is 14.75x and 16.82x faster compared to the naive Obliviator and Opaque.

Expansion. As denoted in Section 8, here we demonstrate additional experiments on smaller block sizes for expansion (see Figure 26), against the naive extension of Obliviator [73], the expansion introduced in Jodes [98], and expansion through sorting. Our expansion achieves speedups of 3×–38× and 16×–81× over all the baselines for block sizes 32B and 256B, respectively.