Secure Multi-Party Sampling over Joins

Qiyao Luo
OceanBase, Ant Group
luoqiyao.lqy@antgroup.com

Quanqing Xu
OceanBase, Ant Group
xuquanqing.xqq@oceanbase.com

Chuanhui Yang
OceanBase, Ant Group
rizhao.ych@oceanbase.com

ABSTRACT

Secure multi-party computation (MPC) enables collaborative analytics over private datasets but faces critical efficiency barriers. State-of-the-art MPC protocols for query processing with joins incur prohibitive computational costs. While sampling-based approximate query processing has revolutionized plaintext analytics, its extension to secure settings remains unexplored. This paper proposes the first efficient and secure protocol for sampling over joins. The protocol achieves near-linear asymptotic complexity while preserving the confidentiality of input and metadata (e.g., degree and join sizes). It supports a wide range of queries, including multi-way joins, comparisons, and group-by operations, and is universally applicable across secure computation settings. Experiments demonstrate significant speedups over secure join-then-sample baselines. This work bridges the gap between theoretical secure computation and practical relational analytics, advancing scalable real-world secure collaborative analytics and learning scenarios.

PVLDB Reference Format:
Qiyao Luo, Quanqing Xu, and Chuanhui Yang. Secure Multi-Party Sampling over Joins. PVLDB, 19(7): 1455 - 1468, 2026.
doi:10.14778/3801059.3801062

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/MPC-Query-Processing/SSOJ.

1 INTRODUCTION

In the contemporary data-centric landscape, data sharing and data security have emerged as critical and interdependent challenges.

Example 1.1. Consider a scenario where several hospitals and insurance companies wish to collaboratively analyze useful information based on their medical records `M(uid, datetime, disease, fee)` and insurance histories `I(uid, datetime, claims, payment)`. The insurance companies aim to analyze the reimbursment cost and ratio for different diseases:

```
SELECT disease, SUM(payment), AVG(payment/fee)
  FROM M, I
 WHERE M.uid = I.uid and M.disease = I.claims
   and M.datetime = I.datetime
GROUP BY disease
```

Meanwhile, hospitals want to learn the causes of patients' illnesses using historical insurance records:

```
SELECT claims, M.datetime - I.datetime, disease
  FROM M, I
 WHERE M.uid = I.uid
   and M.datetime > I.datetime
```

Although there is significant potential for data-driven collaboration, these datasets contain sensitive personal information and must be protected. The tension between collaboration and privacy concerns has catalyzed extensive research into secure computation, which enables computation on data without revealing it.

Secure Multi-Party Computation (MPC), a paradigm rooted in Yao's Millionaires Problem [51], is one of the most widely adopted secure computation techniques. It enables collaborative computation on private data while preserving confidentiality. Over decades, MPC has evolved from theoretical foundations for simple comparison operations [51] to practical protocols for complex tasks like secure database query processing [11, 14, 28] and machine learning [36, 38], driven by advancements in cryptographic primitives and computing infrastructure. Despite these strides, MPC protocols remain orders of magnitude slower than their plaintext equivalents, primarily due to cryptographic overheads such as secret sharing, oblivious transfers, and zero-knowledge proofs. For instance, the join operators in Example 1.1, which are fundamental to query processing, have near-linear complexity in plaintext, $\tilde{O}(N+M)$[1]. However, fully secure MPC implementations [14] escalate to $\tilde{O}(N^2)$ complexity. This stems from two requirements: (1) padding to obscure the true join size (preventing leakage of meta information like correlations between relations) and (2) cryptographic operations on secret-shared values, which amplify constant-factor overheads. Such inefficiencies render large-scale secure analytics prohibitively slow, highlighting a critical barrier to MPC's real-world adoption.

Sampling offers a promising method to mitigate the efficiency gap in plaintext systems. By extracting a representative subset of data, sampling [5, 20, 22, 33, 40, 52] enables approximate analytics with rigorous error guarantees. It is widely used in plaintext big data analytics. In Example 1.1, since the insurance companies do not necessarily require an exact result and the hospitals use mini-batch stochastic gradient descent (SGD) to train the model, secure sampling provides even stronger utility than plaintext sampling.

However, *sampling over joins*, which generates random samples from the join result, introduces unique challenges even in plaintext: (1) Materializing the full join before sampling (cost $\tilde{O}(N+M)$) produces a correct sample but incurs prohibitive computational overhead, as $M \gg N$ in most cases; (2) Sampling input relations independently and then joining the results (i.e., $\texttt{Sample}(R) \bowtie \texttt{Sample}(S)$)

Proceedings of the VLDB Endowment, Vol. 19, No. 7 ISSN 2150-8097.
doi:10.14778/3801059.3801062

[1] N denotes the input size and M the output size, with $\tilde{O}(\cdot)$ suppressing polylogarithmic factors.

fails to replicate the distribution of $\texttt{Sample}(R \bowtie S)$, as rigorously analyzed in [20]. Recent plaintext advancements [33] and [52] bridge correctness and efficiency, enabling efficient random sampling over joins with near-constant complexity (i.e., $\tilde{O}(1)$ cost per sample). However, progress in secure settings remains limited.

1.1 Secure Sampling Revisited

Recent advances in secure computation [15, 35, 44] have enabled sampling over a flat table of size N while preserving privacy. They ensure that samples are drawn without exposing individual records or memory access patterns. These methods reduced the cost of obtaining a sample of size s from $\tilde{O}(N)$ [15] to amortized $\tilde{O}(s)$ [35, 44], i.e., generating $O(N/s)$ samples of size s together at a total cost of $\tilde{O}(N)$. However, all existing secure sampling frameworks are limited to single-table scenarios and cannot address joins. The naive solution, join-then-sample, faces significant limitations. Although the practical secure sort-merge-join protocols [11, 28] have cost $\tilde{O}(N+M)$, they leak the full join size M and weakens the security guarantee. To protect M, the output is padded to its worst-case size N^k (for k relations), and a secure nested-loop join [14] suffices. But this leads to $\Omega(N^k)$ complexity for sampling over the result.

Another idea is to adapt plaintext sampling over joins algorithms [33, 52] to distributed oblivious RAM (ORAM) [45, 47]. The oblivious B-tree index [18, 19] cannot randomly select elements within groups (i.e., tuples with a specific join key), but random selection from a group is the key component in plaintext algorithms. Caching all tuples for a key before sampling reveals the group size (i.e., degree of the key) and also breaks the security guarantee. Thus, it is challenging to achieve the adaption.

Despite progress in secure sampling on flat tables and efficient plaintext algorithms for sampling over joins, a critical gap remains — ***Is there a general and efficient way to do secure sampling over joins?***

1.2 Our Contributions

In this paper, we answer the above question in the affirmative. We propose a circuit-based algorithm that enables sampling over joins without explicitly computing the full join results. The core circuit has $\tilde{O}(N+s)$ size and $\tilde{O}(1)$ depth. This is, to our knowledge, the first secure sampling over joins protocol achieving near-linear asymptotic overhead while preserving full metadata confidentiality. Notably, our circuit-based design is universally applicable across secure computation settings, including MPC, trusted execution environments (TEE), and fully homomorphic encryption (FHE). Specifically, in this paper:

(1) We propose *expanded sampling*, a novel primitive that enables sampling s elements from an expanded dataset without explicitly materializing it, and design an efficient circuit-based solution.
(2) We formalize expanded sampling as a foundational primitive for joins, generalizing from binary joins to acyclic multi-way joins. Our approach adopts the idea from [52] to assign each tuple a *weight* (i.e., the number of join results it contributes to its subtree) and performs expanded sampling according to these weights.
(3) We optimize the protocol by reducing cryptographic randomness consumption and designing a one-to-one mapping from random numbers within the full join size to its results. Additionally, we extend our protocol's scope to support queries involving group-by, comparisons, and cyclic joins.
(4) We implement a prototype at the top of ABY3 [1] and MP-SPDZ [2], and evaluate its performance. Experimental results demonstrate significant speedups over the secure join-then-sample baselines.

1.3 Related Work

Sampling-based approximate query processing has a long history in database systems. Random sampling from a single relation was formalized in [40], while the challenges in sampling over joins were identified in [20]. [5] studied a tractable special case — foreign-key joins, where one relation's primary key strictly reference another relation's foreign key. Recent advances in sampling over joins techniques include: ripple join [25] which generates uniform (but non-independent) samples via progressive aggregation; wander join [33] which uses random walks to produce independent (but non-uniform) samples; and [52] which achieves both uniformity and independence. There are also sampling-based AQP systems like BlinkDB [6], Quickr [29], and VerdictDB [41], which have operationalized sampling for practical large-scale analytics.

Query processing under secure multi-party computation has also been extensively studied: SMCQL [14] introduced the first secure two-party query engine, assuming semi-honest adversaries. It implements multi-way joins via unoptimized circuits of size $O(N^k)$, limiting scalability to datasets with a few hundred tuples. [10] proposed a three-party honest-majority protocol for foreign-key joins with $O(N \log N)$ complexity, leveraging the property that output size $M \le N$ for such joins. [11, 28] achieved $O(N \log N + M \log M)$ complexity for general binary joins; and [34] extended this to multi-way joins with $O(N+M)$ online complexity under the same threat model. Oblivious algorithms like [19, 31] attain $O(N \log N + M \log M)$ complexity for multi-way joins under TEE setting. All these methods incur $\Omega(N+M)$ computational and communication overhead, compounded by large constants from cryptographic operations (e.g., oblivious transfers, encryption).

To reduce costs, secure sampling techniques trade accuracy for efficiency: SAQE [15] performs random sampling under MPC and evaluates queries on the sample, but its sampling step retains $\tilde{O}(N)$ complexity. [44] introduced an oblivious TEE-based sampling algorithm with $\tilde{O}(s)$ amortized complexity per sample. [35] proposed a general-purpose sampling circuit for uniform/stratified sampling with/without replacement, achieving $\tilde{O}(s)$ amortized complexity.

2 PRELIMINARIES

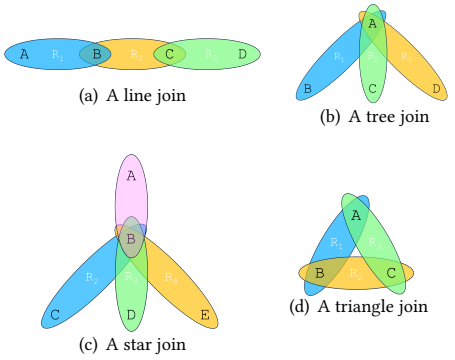
2.1 Join Basics

In this work, we focus on full join queries of the following form:

$$\mathcal{J} = R_1(\boldsymbol{E}_1) \bowtie R_2(\boldsymbol{E}_2) \bowtie \cdots \bowtie R_k(\boldsymbol{E}_k),$$

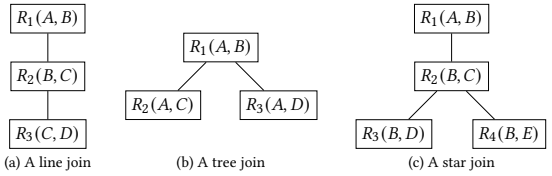
where $R_i(\boldsymbol{E}_i)$ denotes a relation with attribute sets $\boldsymbol{E}_i$ $(1 \le i \le k)$.

The join $\mathcal{J}$ can be represented as a hypergraph $\mathcal{H}$ where each vertex represents attributes across all relations and each hyperedge contains all attributes in $\boldsymbol{E}_i$. Figure 1 illustrates hypergraphs for common join types. *Acyclic* joins (e.g., line, star, or tree joins) do not contain cycles in their hypergraphs, whereas *cyclic* joins (e.g., triangle or grid joins) contain cycles. An acyclic hypergraph $\mathcal{H}$ admits a join tree $\mathcal{T}$ satisfying: (1) each node in $\mathcal{T}$ is labeled by a hyperedge from $\mathcal{H}$ and, (2) for every attribute $\boldsymbol{A} \in \mathcal{T}$, the subset of nodes in $\mathcal{T}$ containing $\boldsymbol{A}$ forms a connected subtree. See Figure 2 as a corresponding example. Acyclicity is foundational in database theory, enabling linear-time plaintext join evaluation $O(N + M)$ via the Yannakakis algorithm [12, 50], where N is the input size and M is the full join size. Joins beyond acyclicity (i.e., cyclic joins) require decomposition into acyclic substructures using techniques like generalized hypertree decomposition (GHD) [4, 24]. GHD rewrites a cyclic join as an acyclic composition of clusters (subsets of relations), enabling efficient evaluation at the cost of increased intermediate complexity.



(a) A line join (b) A tree join (c) A star join (d) A triangle join

Figure 1: An example of hypergraphs of different joins



(a) A line join (b) A tree join (c) A star join

Figure 2: An example of corresponding join trees

2.2 Sampling Basics

Sampling is a fundamental technique for approximate computation, balancing accuracy and efficiency by analyzing representative subsets of data. Given an input sequence $\mathcal{X} = (x_1, x_2, \cdots, x_n)$, a sampling method returns a multiset $\mathcal{S}$ where each element $x \in \mathcal{S}$ is drawn randomly from $\mathcal{X}$. Common strategies are classified as:

- Uniform vs. Stratified: Uniform sampling assigns equal selection probability to all elements. While stratified sampling partitions $\mathcal{X}$ into disjoint strata (subgroups) and samples uniformly from each stratum. This ensures representation of rare subgroups (e.g., small group size) and reduces variance in group-by queries.
- With vs. Without replacement: Elements are independently sampled, allowing duplicates in with replacement sampling. While Samples in without replacement sampling are unique elements, improving quality for large s/n ratios.

The goal of sampling over joins is to extract random samples from the full join result $\mathcal{J} = R_1 \bowtie \cdots \bowtie R_k$ without materializing $\mathcal{J}$. Naive join-then-sample approach incur $\tilde{O}(N + M)$ cost, where $M = |\mathcal{J}|$. Recent work removes the materialization cost. Ripple join [25] uses "blind search" to approximate aggregates (uniform but non-independent samples). Wander join [33] performs random walks on the join tree to generate independent (but non-uniform) samples. [52] achieves both uniformity and independence by iteratively refining join paths with rejection sampling and calibrated probabilities. These methods exploit acyclic join structures to avoid full materialization, making them foundational for scalable secure sampling over joins.

2.3 MPC Basics

MPC enables multiple parties to jointly compute a function over their private inputs while preserving privacy. Formally, for private inputs $x_1, \cdots, x_c$ held by c parties, an MPC protocol computes $f(x_1, \cdots, x_c)$ such that no coalition of adversarial parties learns anything about the inputs x_i beyond the function's output.

2.3.1 Security Definition. We adopt the *real-ideal paradigm* to formalize security. In the "ideal world", a trusted third party securely computes f, receiving all inputs and returning only the output. In the "real world", parties execute a protocol Π to compute f, exchanging messages over secure channels. The *view* of a party in either world includes its private input, random coins, and received messages (excluding internal randomness). For an adversary $\mathcal{A}$ corrupting a subset of parties, its view comprises the views of all corrupted parties. A protocol Π is secure if for all $\mathcal{A}$, there exists a simulator Sim in the ideal world such that $\mathcal{A}$'s real-world view is computationally indistinguishable from its ideal-world view.

Algorithm 1 is the ideal functionality for sampling over joins. It enforces that: (1) the join result and intermediate computations (e.g., join keys, tuple degrees) remain hidden from all parties; (2) only the input/output sizes are public; (3) the released samples are secret-shared, friendly to downstream secure applications.

Algorithm 1: Ideal Functionality $\mathcal{F}_{\mathsf{SSOJ}}$

Input: $[\![R_1(\boldsymbol{E}_1)]\!], \cdots, [\![R_k(\boldsymbol{E}_k)]\!]$, sample size s
Output: A sample $[\![T(\boldsymbol{E}_1 \cup \cdots \cup \boldsymbol{E}_k)]\!]$ of size s
1 Recover R_i from $[\![R_i]\!]$;
2 Compute the joins $\mathcal{J} \leftarrow R_1 \bowtie \cdots \bowtie R_k$;
3 $\mathcal{S} \leftarrow$ a random, [WR/WoR], and [uniform/stratified] sample of size s from $\mathcal{J}$;
4 Compute the secret share $[\![\mathcal{S}]\!]$ of $\mathcal{S}$;
5 **return** $[\![\mathcal{S}]\!]$

2.3.2 Secret Sharing. Secret sharing is a foundational primitive for MPC, distributing private data into shares that reveal no information unless recombined. An (t, n)-secret sharing scheme splits an l-bit secret v into n shares, such that any $t - 1$ shares reveal no information about v, while any t shares reconstruct v. We use $[\![v]\!]$ to denote the value is in secret-shared form.

2.3.3 Dummy Tuples. Dummy tuples (or dummies, denoted as $\perp$) are critical in secure computation to protect memory access patterns. There are primary use cases: (1) padding dummies tuples to a fixed size to hide the intermediate relation sizes. For example, we pad the join size to n^k in the secure join-the-sample method to conceal the actual join size m. (2) placing dummies as a placeholder for empty elements. These can also be represented as zeros or other values that do not affect subsequent processing.

2.3.4 Composition Theorem. The security of complex protocols can be reduced to sequential composition of sub-protocols. Following the composition theorem [17], a protocol Π is secure iff:

(1) Π is a sequential composition of sub-protocols $\Pi_1, \cdots, \Pi_m$;
(2) Each Π_i is proven secure under the real-ideal paradigm;
(3) All intermediate values are in secret-shared form.

This theorem ensures modular security proofs for layered MPC constructions.

2.3.5 Generic and Function-Specific Protocols. MPC protocols are categorized as:

- Generic protocols: Universal frameworks like Yao's Garbled Circuits [51], GMW [23], and BGW [16] securely evaluate any function represented as a Boolean/arithmetic circuit.
- Function-specific Protocols: Tailored protocols for specific tasks (e.g., intersection, sorting) and certain models (e.g., two-party, three-party honest-majority) optimize efficiency by leveraging problem structure. While less general, they often outperform generic solutions in practice.

In this paper, we design a circuit-based algorithm for generic secure sampling over joins solution, which is applicable in different secure computation settings. Our algorithm is a composition of several basic circuits. In specific secure setting, we can use function-specific protocols to optimize the efficiency, as detailed in our system implementation in Section 6.

2.3.6 Complexity Measures. The complexity of an MPC protocol is measured by computation time (i.e., the total computation time of all parties during the protocol), communication cost (i.e., the total size of messages sent/received during the protocol), and the number of communication rounds. The cost[2] of executing a circuit-based MPC protocol scales linearly with the circuit size, and the communication rounds scale linearly with the circuit depth (i.e., GMW, BGW) or are constant (i.e., Yao's GC and its variants).

Following the convention of prior works [11, 28, 35, 37, 42, 43, 48, 49] in asymptotic complexity analyses, we assume the common factors (such as the bit-length of data ℓ, the security parameters κ and σ) and the query size (including the number of relations k and the number of attributes in each relation) are constant. The cost $\tilde{O}(n)$, surpressing the polylogarithmic factor in asymptotic representation, is denoted as "near-linear".

[2]When the time and communication are asymptotically the same, we use the term "cost/complexity" to refer to both.

2.4 Circuit Primitives

This section defines circuit primitives used in our work (summarized in Table 1), including their asymptotic complexity.

Primitives	Circuit Size	Circuit Depth
Arithmetic ($+, -, \times, \div,$ mod)	$O(1)$	$O(1)$
Comparison ($=, \neq, >, \geq$)	$O(1)$	$O(1)$
Logical (AND, OR, MUX)	$O(1)$	$O(1)$
SCAN	$O(n)$	$O(\log n)$
SORT	$O(n \log^2 n)$	$O(\log^2 n)$
JOIN	$O(n \log^2 n)$	$O(\log^2 n)$
RAND	$O(1)$	$O(1)$
WR Sample	$O(s)$	$O(1)$
WoR Sample	$O(s \log^2 s \log \sigma)$	$O(\log^2 s \log \sigma)$

Table 1: Asymptotic costs of circuit primitives.

2.4.1 Basic Computations. Arithmetic and logical operations (e.g., $+, \times, \div$, mod, $=, >$, AND, OR, etc.) are implemented as constant-size, constant-depth circuits. For a fixed bit-length ℓ, these operations have $O(1)$ size and depth, though multiplication/division/modulo incur higher constants due to $O(\ell^2)$ bit-level operations. The multiplexer (MUX) selects between two inputs x_0, x_1 based on a control bit $c \in \{0, 1\}$, enabling conditional logic in circuits.

2.4.2 Scan (Prefix-Sum). The prefix-sum operation (SCAN) takes an associative operator $\oplus$ and a sequence $X = (x_1, \cdots, x_n)$, outputting $(x_1, x_1 \oplus x_2, \cdots, x_1 \oplus \cdots \oplus x_n)$ in $O(n)$ size and $O(\log n)$ depth [32]. A segmented variant computes prefix-sums within segments defined by a key array $A = (a_1, \ldots, a_n)$ (e.g., consecutive duplicates in A define segments). This adapts the prefix-sum circuit to segmented workloads [49], incurring identical asymptotic costs.

When $\oplus$ is the arithmetic addition $+$, we simply call the output (segmented) prefix sum as the PrefixSum of X (segmented by A). The other common choice is PrefixCopy, which replaces dummy value in X with the last non-dummy value before it (if there is one).

2.4.3 Sort and Permute. Sorting inputs $X = (x_1, x_2, \cdots, x_n)$ and outputs $Y = (y_1, y_2, \cdots, y_n)$ in ascending order (equal elements are sorted in the same order that they appear in the input) which is a permutation of X. Circuits for sorting have been extensively studied: the asymptotically optimal circuit achieves $O(n \log n)$ size and $O(\log n)$ depth (i.e., the AKS sorting network [7]), but suffer huge constants. Practically, the Bitonic sorter [13] is prefered for $O(n \log^2 n)$ size and $O(\log^2 n)$ depth.

Permutation circuits reorder inputs based on a permutation vector $P = (p_1, \cdots, p_n)$ (i.e., each x_i is moved to position p_i in the output). For public P, the Waksman permutation network [46] uses $O(n \log n)$ size and $O(\log n)$ depth; For secret P, sorting (X, P) by P suffices.

2.4.4 Join. Given relations $R(A, B)$ and $S(B, C)$ with n tuples each, their join $R \bowtie S$ combines tuples with matching B values.

$$R \bowtie S = \{(a, b, c) \mid (a, b) \in R \text{ and } (b, c) \in S\}.$$

To avoid $O(n^2)$ complexity (as the join using a circuit must prepare for the worst case where the output size can be n^2), we

focus on foreign key joins, where $R.B$ is a foreign key reference to $S.B$, ensuring $|R \bowtie S| \leq n$. Algorithm 2 implements this via sorting and prefix-sums, with unjoined tuples filled with dummies The circuit has $O(n \log^2 n)$ size and $O(\log^2 n)$ depth.

Algorithm 2: Foreign Key Join Circuit

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

Initialize $T(A, B, C, idx)$;
for $i \leftarrow 1$ **to** $|S|$ **do in parallel**
  $T[i] \leftarrow (0, S[i].B, S[i].C, i)$;
for $i \leftarrow 1$ **to** $|R|$ **do in parallel**
  $T[i + |S|] \leftarrow (R[i].A, R[i].B, \perp, i + |S|)$;
`Sort` T by $T.B$;
`PrefixCopy` $T.C$ segmented by $T.B$;
`Perm` T by $T.idx$;
Drop the first $|S|$ rows and remove column idx from T;
return T

2.4.5 Random Number Generation. Circuits are inherently deterministic, so randomness is injected via pre-shared random bits. A uniformly random integer $\texttt{RAND}(x)$ over $[x] := \{0, \cdots, x-1\}$ is generated by sampling a sufficiently long bit string (i.e., $\log x + \sigma$) and computing it modulo x. [35] proposes circuits to securely random sample s numbers with/withour replacement from $[x]$, where s is public and x is either public or secret.

3 SECURE EXPANDED SAMPLING

Let $\mathcal{X} = (x_1, \cdots, x_n)$ be a dataset with weights $\mathcal{W} = (w_1, \cdots, w_n) \in \mathbb{N}^n$. The expanded dataset $\mathcal{X}^{\mathcal{W}}$ is a conceptual sequence of size $m = \sum w_i$ where each x_i is repeated w_i times successively:

$$\mathcal{X}^{\mathcal{W}} = (\underbrace{x_1, \ldots, x_1}_{w_1 \text{ times}}, \underbrace{x_2, \ldots, x_2}_{w_2 \text{ times}}, \ldots, \underbrace{x_n, \ldots, x_n}_{w_n \text{ times}}).$$

Expanded sampling aims to draw s elements from $\mathcal{X}^{\mathcal{W}}$ without explicitly materializing the expanded dataset, which would incur $\Omega(n + m)$ overhead. Instead, we directly sample from $\mathcal{X}$ while respecting the weights.

A fundamental $\Omega(n)$ lower bound exists for secure sampling: omitting any x_i from computation or access leaks $x_i \notin \mathcal{S}$ [35]. To amortize this cost, we adopt batch sampling [35, 44], which samples all s elements simultaneously via a single dataset scan to reduce per-sample overhead. The target of secure expanded sampling is to take a sample of size s from the expanded dataset in $O(n \log^2 n + s \log^2 s)$ cost without leaking data value or acccess pattern.

3.1 Expanded Sampling vs. Weighted Sampling

Weighted sampling is a plaintext method where elements are selected with probabilities proportional to their weights $w_i / \sum w_j$. The two approaches are equivalent for single element sample cases ($s = 1$) but diverge fundamentally for multiple elements sample scenarios ($s > 1$): Expanded sampling allows repeated sampling of x_i until its weight w_i is exhausted, naturally supporting WR and WoR sampling from $\mathcal{X}^{\mathcal{W}}$. It maintains independence between samples, enabling parallel execution. Weighted sampling mirrors expanded sampling for WR sampling. While for WoR sampling, the weight w_i must be dynamically adjusted (decremented after sampling x_i), introducing dependencies that preclude parallelization. This distinction is critical for join processing: expanded sampling inherently models the multiplicity of join results (e.g., a tuple x_i may contribute w_i distinct join outputs), whereas weighted sampling cannot capture such correlations due to its static probability distribution.

3.2 Uniform Expanded Sampling

Our idea is mapping random numbers to weighted elements in $\mathcal{X}$ through a range partitioning strategy. First, the algorithm computes the total weight $m = \sum w_i$ securely using a prefix-sum circuit and defines a many-to-one mapping from $[0, m)$ to $\mathcal{X}$, where interval $[0, w_1)$ corresponds to x_1, $[w_1, w_1 + w_2)$ corresponds to x_2, $\cdots$, $[m - w_n, m)$ corresponds to x_n. Next, the algorithm generates s random numbers $y_1, \cdots, y_s \in [0, m]$ via WR/WoR circuits. Finally, the sampled elements are retrieved by resolving these random numbers against the prefix-sum intervals, a process equivalent to a θ-join between a relation $R(\boldsymbol{L}, \boldsymbol{R}, \boldsymbol{X})$ (range $[\boldsymbol{L}, \boldsymbol{R})$ corresponds to $\boldsymbol{X}$) and a relation $S(\boldsymbol{Y})$ ($\boldsymbol{Y}$ stores the random numbers), where the join condition $\boldsymbol{L} \leq \boldsymbol{Y} < \boldsymbol{R}$. This algorithm (see Algorithm 3 for details) avoids materializing the expanded dataset, instead leveraging sorting and prefix-copy circuits to propagate values from elements to their associated samples within a unified table.

Algorithm 3: Uniform Expanded Sampling Circuit

Input: $\mathcal{X} = (x_1, \ldots, x_n)$, $\mathcal{W} = (w_1, \ldots, w_n)$, s
Output: $\mathcal{S} = (s_1, \ldots, s_s)$

$(e_1, \ldots, e_n, m) \leftarrow \texttt{PrefixSum}$ of $(0, w_1, \ldots, w_n)$;
$(y_1, \cdots, y_s) \leftarrow$ WR/WoR sample of size s from $[0, m)$;
Initialize $T(pos, idx, val)$ of size $n + s$;
for $i \leftarrow 1$ **to** n **do in parallel**
  $t_i \leftarrow (e_i, i, x_i)$;
for $j \leftarrow 1$ **to** s **do in parallel**
  $t_{j+n} \leftarrow (y_j, j + n, \perp)$;
`Sort` T by $T.pos$;
`PrefixCopy` $T.val$;
`Perm` T by $T.idx$;
return $(T.val)_{j=n+1}^{n+s}$

We use a special design to retrieve elements, which is quite similar to the foreign key join circuit. The temporary table T merges the dataset and sample set, containing three attributes: pos stores the left node of every range for dataset, and the random number for sample set; idx is an identifier for element/sample tuples where $1 \leq idx \leq n$ for dataset and $n+1 \leq idx \leq n+s$ for sample; val keeps the value of dataset, while dummy for samples at the begining and sampled tuple from $\mathcal{X}$ at the end.

Sorting T by pos ensures that elements directly precede their associated samples: If element x_i has weight $w_i = 0$, the left node of element x_{i+1}'s range is $e_{i+1} = e_i + w_i = e_i$. x_i and x_{i+1} are right next to each other. No samples will be taken from x_i. Otherwise, the left node of element x_i and x_{i+1}'s ranges are e_i and $e_{i+1} = e_i + w_i$.

Tuples are copied from x_i if and only if its random number is within $[e_i, e_{i+1})$. Finally, permutation replaces the dataset and sample set the same as begining, with samples having taken proper elements.

The algorithm's complexity is dominated by the sorting circuit, having size $O((n+s)\log^2(n+s))$ and depth $O(\log^2(n+s))$, unrelated to the expanded dataset size m. Here we omit the complexity to generate random numbers, which is detailed in Section 2.4.5.

Example 3.1. Consider an example where $\mathcal{X} = (a, b, c, d)$ and $\mathcal{W} = (2, 3, 0, 1)$. The prefix-sum result is $(2, 5, 0, 6)$. Suppose the random numbers are $(4, 5, 0, 3)$. Figure 3 shows the running details of taking proper sample result (b, d, a, b), matching the result directly sampled from $\mathcal{X}^{\mathcal{W}} = (a, a, b, b, b, d)$.

3.3 Stratified Expanded Sampling

We generalize our approach to stratified expanded sampling, where a stratum identifier $\boldsymbol{A}$ partitions $\mathcal{X}$ into disjoint groups. The input comprises $(\mathcal{X}, \mathcal{W})$ and a stratum sequence $(b_1, \cdots, b_s)$ specifying the target stratum for each sampled element. Uniform expanded sampling emerges as a special case when all elements share a single stratum.

Rather than isolating strata, wherein processing each stratum separately and non-stratum elements are marked as dummies (an approach that reveals stratum sizes and incurs $\tilde{O}(k(N+s))$ overhead for k strata), we propose a unified protocol that processes all strata simultaneously. First, $(\mathcal{X}, \mathcal{W})$ is sorted by $\mathcal{X}.\boldsymbol{A}$ to group tuples with identical strata while preserving their original order. Next, a segmented prefix-sum computes stratum-specific total weights $d_{\boldsymbol{c}} = \sum_{x_i.\boldsymbol{A}=\boldsymbol{c}} w_i$ for each stratum c, where $d_{\boldsymbol{c}}$ is stored in the final tuple of the stratum. This total weight is then propagated to samples via a foreign key join between the stratum-weight relation and the sample sequence, mapping each b_j to its stratum's total weight d_{b_j}. Next, random numbers $y_j \in [0, d_{b_j})$ are generated using WR/WoR circuits, ensuring samples respect stratum-specific ranges. Finally, elements are retrieved by resolving each (b_j, y_j) pairs against the stratum's prefix-sum intervals, akin to the θ-join mechanism described in previous section.

Different from uniform expanded sampling that all sampled elements have the same sample range, stratified expanded sampling requires distinct sample ranges for each stratum. Algorithm 4 is a circuit that generates random numbers for each stratum according to its size (the example shows WR numbers; for WoR numbers an extra pointer jumping circuit is needed to handle distinction). Algorithm 5 takes elements according to stratified identifiers and random numbers. The difference between uniform expanded sampling is the sorting needs the startified identifiers as sorting key. The protocol's complexity is also dominated by sorting operations with size $O((n+s)\log^2(n+s))$ and depth $O(\log^2(n+s))$.

Example 3.2. See Figure 4 as an example of dataset $\mathcal{X}$ and weight $\mathcal{W}$, as well as stratified sequence (b, a, c, a, a). In the first stage, we sort the dataset and its weight in ascending order, and then perform prefix sum operations to get the total weight: a has total weight 6 and b, c both have weights 3. Next, for the stratified sequence, we use foreign key join circuit to get its weight, indicating the range of its random number. These random numbers, are not distinct since we are choosing WR samples. In the second stage, we use a similar method to obtain elements. The taking results match those token from the expanded datasets specifying by the stratified sequence.

Algorithm 4: Stratified Numbers Generation

Input: $\mathcal{X} = (x_1, \cdots, x_n)$; $\mathcal{W} = (w_1, \ldots, w_n)$; $B = (b_1, \cdots, b_s)$
Output: $Y = (y_1, \ldots, y_s)$

```
// Calculate the total weight of each stratum.
```
Sort $(\mathcal{X}, \mathcal{W})$ together by $\mathcal{X}.\boldsymbol{A}$;
Initialize $G(key, val) \leftarrow (\mathcal{X}.A, \mathcal{W})$;
PrefixSum $G.val$ segmented by $G.key$;
for $i \leftarrow 1$ **to** $n-1$ **do in parallel**
  if $G[i].key = G[i+1].key$ **then**
    $G[i] \leftarrow (\perp, 0)$;
```
// Get the total weight of each sample.
```
Initialize $T(key, idx, val)$ of size $n+s$;
for $i \leftarrow 1$ **to** n **do in parallel**
  $T[i] \leftarrow (G[i].key, i, G[i].val)$;
for $i \leftarrow 1$ **to** s **do in parallel**
  $T[i+n] \leftarrow (b_i, i+n, 0)$;
Sort T by $(T.key, T.idx)$ in ascending order;
PrefixCopy of $T.val$ segmented by $T.key$;
Perm T by $T.idx$ in ascending order;
```
// Generate uniform random numbers.
```
Initialize sequence Y of size s;
for $i \leftarrow 1$ **to** s **do in parallel**
  $y_i \leftarrow \mathtt{RAND}(T[i+n].val)$;
return Y

Algorithm 5: Taking Random Elements

Input: $\mathcal{X} = (x_1, \ldots, x_n)$, $\mathcal{W} = (w_1, \ldots, w_n)$, $(b_1, \cdots, b_s)$, $(y_1, \cdots, y_s)$
Output: $\mathcal{S} = (s_1, \ldots, s_s)$

```
// Compute the left node of each stratum's range.
```
Sort $(\mathcal{X}, \mathcal{W})$ together by $\mathcal{X}.\boldsymbol{A}$;
$(e_1, \ldots, e_n) \leftarrow$ PrefixSum of $\mathcal{W}$ segmented by $\mathcal{X}.\boldsymbol{A}$;
for $i \leftarrow 2$ **to** n **do in parallel**
  if $x_i.\boldsymbol{A} = x_{i-1}.\boldsymbol{A}$ **then**
    $e_i \leftarrow e_{i-1}$;
  else
    $e_i \leftarrow 0$;
```
// Get the samples.
```
Initialize $T(tag, pos, idx, val)$ of size $n+s$;
for $i \leftarrow 1$ **to** n **do in parallel**
  $t_i \leftarrow (x_i.\boldsymbol{A}, e_i, i, x_i)$;
for $j \leftarrow 1$ **to** s **do in parallel**
  $t_{j+n} \leftarrow (b_j, y_j, j+n, \perp)$;
Sort T by $(T.tag, T.pos)$;
PrefixCopy $T.val$;
Perm T by $T.idx$;
return $(T.val)_{j=n+1}^{n+s}$

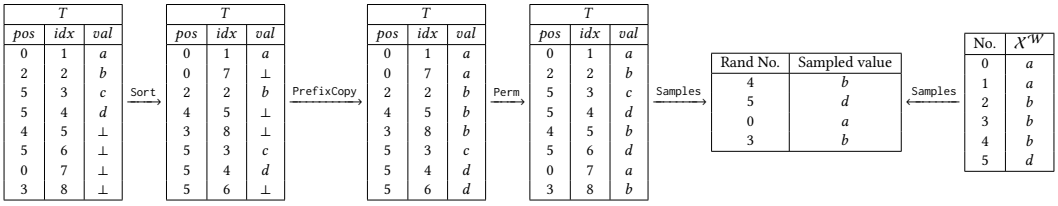
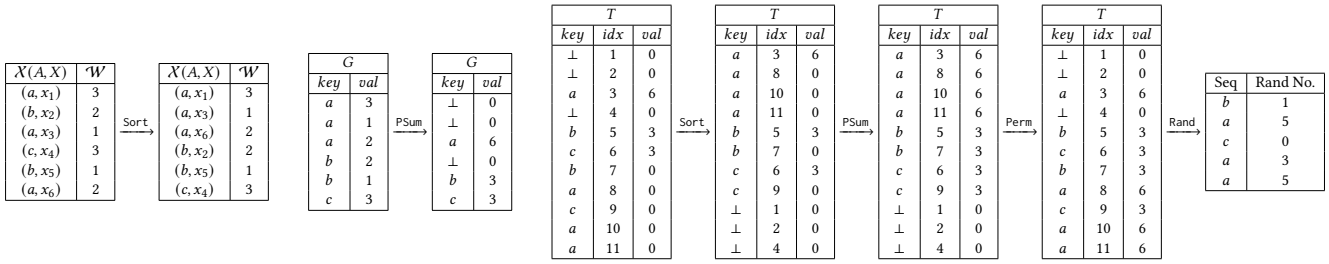
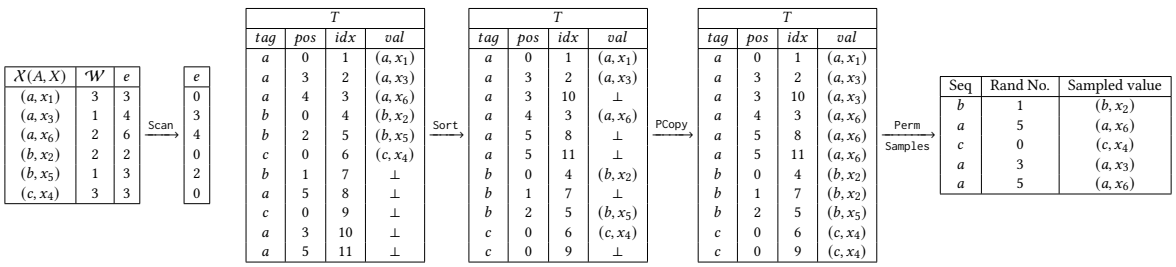


Figure 3: An example of uniform expanded sampling.



(a) Running example of stratified numbers generation.



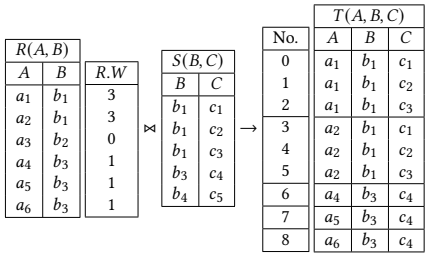
(b) Running example of taking random elements.

Figure 4: An example of stratified expanded sampling.

4 SECURE SAMPLING OVER JOINS

We present a circuit-based framework for secure sampling over joins, achieving $O(N \log^2 N + s \log^2 s)$ complexity for sample size s. The approach generalizes from binary joins to acyclic joins via weight propagation and randomness reuse, ensuring uniformity and independence while avoiding full join materialization.

4.1 Sampling over Binary Joins



Let $R(\boldsymbol{E}_R) \bowtie S(\boldsymbol{E}_S)$ be a binary join with join key $\boldsymbol{K} = \boldsymbol{E}_R \cap \boldsymbol{E}_S$. Without loss of generality, let R be the root and S be the leaf in the join tree. For each $r \in R$, define its weight $w_r = |\{s \in S \mid r.\boldsymbol{K} = s.\boldsymbol{K}\}|$ (denoted as $R.W$), representing its multiplicity in the implicit join result $\mathcal{J} = R \bowtie S$. The expanded dataset $R^{R.W}$ captures this multiplicity, enabling sampling over $\mathcal{J}$ without materializing it. In the above example, R has weights $(3, 3, 0, 1, 1, 1)$. The join result $T = R \bowtie S$ can be partitioned into $T_1(A, B) = R^{R.W}$ and $T_2(B, C) = S$ with a fixed element in attribute B.

4.1.1 Secure Sampling Protocol. The sampling protocol proceeds in two phases: (1) Expanded sampling on R: Compute weights w_r for all $r \in R$: Algorithm 6 is the circuit to update the weights of R. Then use expanded sampling to draw s samples $\mathcal{S}_R$ from R. (2) Stratified sampling on S: For each $s \in \mathcal{S}_R$, treat $s.\boldsymbol{K}$ as the stratum identifier and draw stratified weighted sampling (weights are all 1) from S. Then finally concatenate with $\mathcal{S}_R$ and return sample $\mathcal{S}$.

In stratified sampling over joins, handling without-replacement (WoR) constraints requires precise differentiation between structural duplicates and true duplicates. For samples $s_1, s_2 \in \mathcal{S}_R$, WoR sampling from S must enforce distinctness only when s_1 and s_2 are from the same expanded dataset element in $R^{R.W}$. Formally:

- If $s_1 \neq s_2$ (i.e., s_1, s_2 map to distinct tuples in R), even with identical join keys ($s_1.\boldsymbol{K} = s_2.\boldsymbol{K}$), their associated samples in S may coincide. This arises because s_1 and s_2 correspond to distinct elements in R, preserving the WoR requirement at the first join.
- If $s_1 = s_2$ (i.e., the same $r \in R$ is sampled twice due to $w_r > 1$), WoR constraints must be applied to S sampling to ensure distinctness in the final result.

Algorithm 6: Weight Updation

Input: Relation R and S
Output: Updated relation R

Initialize $T(key, idx, val)$;
for $i \leftarrow 1$ **to** $|S|$ **do in parallel**
  $T[i] \leftarrow (S[i].K, i, S[i].W)$
for $i \leftarrow 1$ **to** $|R|$ **do in parallel**
  $T[i + |S|] \leftarrow (R[i].K, i + |S|, 0)$
`Sort` T by $T.key$ in ascending order;
`PrefixSum` of $T.val$ segmented by $T.key$;
`Perm` T by $T.idx$;
for $i \leftarrow |S| + 1$ **to** $|T|$ **do in parallel**
  $R[i - |S|].W \leftarrow R[i - |S|].W \times T[i].val$;
return T

It ensures that the protocol avoids over-constraining samples. The key insight is that stratum-level uniqueness is insufficient; instead, the protocol tracks the origin of each sample in R to determine whether distinctness is required.

Lemma 4.1. *The circuit returns a uniform random sample $\mathcal{S}$ from the joined result $R \bowtie S$ in $O(N \log^2 N + s \log^2 s)$ size and $O(\log^2 N + \log^2 s)$ depth.*

Proof. $(r, s) \in R \bowtie S$ where $r \in R$, $s \in S$ and $r.\boldsymbol{K} = s.\boldsymbol{K}$. We use $R.W$ to denote the total weight of R and $S.W_{|\boldsymbol{K}}$ to denote the total weight of tuples in S with key $\boldsymbol{K}$.

$$Pr[(r, s) \in \mathcal{S}] = \Pr[r \text{ is sampled}] \cdot \Pr[s \text{ is sampled} | r \text{ is sampled}]$$
$$= \frac{w_r}{R.W} \cdot \frac{1}{S.W_{|r.\boldsymbol{K}}} = \frac{1}{|R \bowtie S|}.$$

Each tuple (r, s) is sampled uniformly and randomly.

Suppose $|R| = |S| = N$. The update weight circuit sorts $2N$ elements and the two sample circuit sorts $N + s$ elements. In all, the algorithm requires a circuit with size $O(N \log^2 N + s \log^2 s)$ and depth $O(\log^2 N + \log^2 s)$. □

4.1.2 Optimization: Reusing Randomness. A key innovation is that the initial random number generated for R inherently contains entropy for S-sampling. Let $y \in [0, m)$ be a random number mapping to $r \in R$. The offset $y' = y - a$ (where a is the start of r's mapping interval) directly samples from tuples in S joining with r (or, in stratum $r.\boldsymbol{K}$), replacing costly random number generation circuit with subtraction operations. This avoids explicit random number generation for S, reducing circuit complexity: For WR sampling, replaces a modulo operation with a subtraction; for WoR sampling, further eliminates a pointer-jumping circuit. The take random element with random number updation is listed in Algorithm 7.

In the above example, a random number $y \in [3, 6)$ mapping to $(a_2, b_1) \in R$ can be transformed into a secondary random number $y' = y - 3 \in [0, 3)$ to sample uniformly from S tuples joining with (a_2, b_1). The minus value is the prefix sum of weights before the tuple (a_2, b_1) (in its stratum if there exists).

4.1.3 Sampling over Line Joins. The methodology for binary joins naturally extends to line joins $R_1 \bowtie \cdots \bowtie R_k$ where the join tree

Algorithm 7: Taking Random Elements With Updation

Input: $\mathcal{X} = (x_1, \ldots, x_n)$, $\mathcal{W} = (w_1, \ldots, w_n)$, $(b_1, \cdots, b_s)$, $(y_1, \cdots, y_s)$
Output: $\mathcal{S} = (s_1, \ldots, s_s)$, $Y' = (y_1, \cdots, y_s)$

Sort $(\mathcal{X}, \mathcal{W})$ together in ascending order;
$(e_1, \ldots, e_n) \leftarrow$ `PrefixSum` of $\mathcal{W}$ segmented by $\mathcal{X}.\boldsymbol{A}$;
for $i \leftarrow 2$ **to** n **do in parallel**
  if $x_i.\boldsymbol{A} = x_{i-1}.\boldsymbol{A}$ **then**
    $e_i \leftarrow e_{i-1}$;
  else
    $e_i \leftarrow 0$;
Initialize $T(tag, pos, idx, val, sub)$ of size $n + s$;
for $i \leftarrow 1$ **to** n **do in parallel**
  $t_i \leftarrow (x_i.\boldsymbol{A}, e_i, i, x_i, e_i)$;
for $j \leftarrow 1$ **to** s **do in parallel**
  $t_{j+n} \leftarrow (b_j, y_j, j + n, \perp, \perp)$;
`Sort` T by $(T.tag, T.pos)$;
`PrefixCopy` $T.val$ and $T.sub$;
`Perm` T by $T.idx$;
for $i \leftarrow n + 1$ **to** $n + s$ **do in parallel**
  $y[i - n] = T[i].pos - T[i].sub$;
return $(T.val)_{j=n+1}^{n+s}$ *and* Y

forms a linear hierarchy with R_1 as the root and R_i connecting with R_{i+1}. The protocol proceeds in two stages:

(1) Bottom-up weight updation: Initialize attribute $R_i.W \leftarrow 1$ for $1 \le i \le k$. Traverse the join tree from R_{k-1} to R_1, recursively computing weights for R_i as the product of their join multiplicities in R_{i+1} (Algorithm 6). This ensures that w_r captures the total number of join paths from $r \in R_i$ to the leaf R_k.
(2) Top-down sampling: Draw s random numbers from $[0, m)$ where m is the total weight of R_1 and take elements from R_1. For $2 \le i \le k$, derive random numbers for R_i by subtracting the prefix-sum offset of the parent relation R_{i-1}, everaging the one-to-one mapping between $[0, m)$ to the join result. Then take elements from R_i using the join key $\boldsymbol{E}_{i-1} \cap \boldsymbol{E}_i$ as its stratum identifier and the derived random numbers.

This approach avoids redundant random number generation. The protocol calls $k - 1$ times weight updation and k times taking random elements. Since k is a constant, it also incurs $\tilde{O}(N + s)$ total complexity, where $N = \sum |R_i|$.

4.2 Sampling over Tree Joins

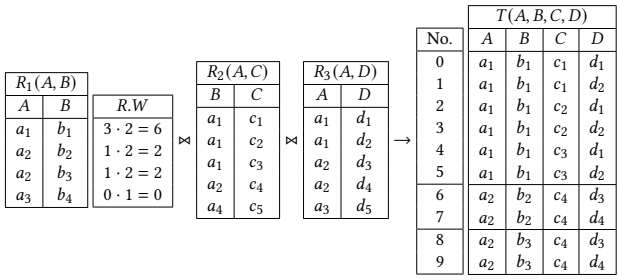
Now we consider a complicated tree joins $R_1(A, B) \bowtie R_2(A, C) \bowtie R_3(A, D)$, and its join tree is in Figure 2.

The weight, as our definition, is the total number of join results it contributes to its subtree. Thus the weights for R_2 and R_3 are all 1, and the weights for $r_1 \in R_1$ are:

$$w_{r_1} \leftarrow \left(\sum_{r_2 \in R_2, r_2.A = r_1.A} 1 \right) \times \left(\sum_{r_3 \in R_3, r_3.A = r_1.A} 1 \right).$$

This is computed via sequential applications of weight updation between (R_1, R_2) and (R_1, R_3) (via Algorithm 6).

The key challenge is resolving a sample $y \in [s, s+d1 \times d_2)$ where d_1, d_2 are the number of R_2 and R_3 tuples joinging with $r_1 \in R_1$ and s is the prefix sum of previous weights. We revisit the result of sort-merge-join below:



The law illustrates that if the tuple is joined with the i-th tuple from R_2 and j-th tuple from R_3, its random number is $y = s + i \cdot d_2 + j$. We can perform a "divide and modulo" circuit to extract $i = \lfloor (y - s)/d_2 \rfloor$ and $j = (y - s) \bmod d_2$. For each tuple $r \in R_1$, we obtain its d_2 in the update weight stage. In Algorithm 6, we not only multiply d_2 to its own weight, but also store it as an extra attribute. After taking samples from R_1 with updation, we obtain the minused value $y - s$ and the degree d_2. Then we extract the two numbers using "divide and modulo" circuit. Finally, we parallelly take elements from R_2 using stratified identifier $r.A$ and $\lfloor (y - s)/d_2 \rfloor$; and from R_3 using stratified identifier $r.A$ and $(y - s) \bmod d_2$.

4.2.1 Generalize to Many Branches. For a k-branch tree where R_1 is the root and all other relations R_i $(2 \le i \le k)$ are the leaf node connected to R_1. The weights for R_1 are the product of sum of the other relations. For each $2 \le i \le k$, perform the weight updation circuit of R_1 and R_i, and extraly store the the total number of tuples in R_i that can join with r. denoted as the degree d_i. And we generalize the decoposition for $r \in R$ to

$$y = s + \sum_{i=2}^{k} \left(l_i \cdot \Pi_{j=i+1}^{k} d_j \right),$$

where l_i denotes the l_i-th tuple that can be joined with r in the relation R_i, and d_j is the degree of relation R_j. Thus, after taking elements from R_1 we get $y \leftarrow y - s$. Then we iteratively calculate $l_i \leftarrow y \bmod d_i$ and update $y \leftarrow \lfloor y/d_i \rfloor$ for i from k to 2. Since the schema is public while the values are secret, all numbers can be calculated by a series of "divide-and-modulo" circuit. These decomposition only takes linear circuit size and constant depth, which does not affect the asymptotic complexity. Finally, we take samples from R_i using the stratified identifier $r.(\boldsymbol{E}_1 \cap \boldsymbol{E}_i)$ and random number l_i for $2 \le i \le k$.

4.3 Sampling over Acyclic Joins

We generalize our framework to acyclic joins using a join tree $\mathcal{T}$. The protocol operates in two phases:

(1) Bottom-up weight updation: Everytime we picked a leaf node from the tree and update the weight of its parent relation, and delete the node. Repeat the process $k-1$ times.

(2) Top-down sampling: Generate random numbers in the range $[0, m)$ and take samples from the root to the leaf. Everytime we finished picking the elements, we remove its prefix weights and extract random numbers for its children relations.

Algorithm 8 make uses of the Breadth-First Search (BFS) order of the join tree to take the samples.

Algorithm 8: Sampling over Acyclic Join `SOAJ`

Input: Relations $\{R_i(\boldsymbol{E}_i)\}_{i=1}^{k}$; join tree $\mathcal{T}$; sample size s
Output: Sampled relation $\mathcal{S}$

Append weight attributes $R_i.W \leftarrow 1$;
$(o_1, \cdots, o_k) \leftarrow$ the BFS order of $\mathcal{T}$;
for $i \leftarrow k$ **to** 2 **do**
  $R_c \leftarrow$ the relation of o_i;
  $R_p \leftarrow$ the parent relation of R_c;
  `UpdateWeight`(R_p, R_c);
Initialize $Y_1, \cdots, Y_k$ random numbers of each relation;
$R_r \leftarrow$ the relation related to o_1;
$m \leftarrow$ sum of $R_r.W$;
$Y_1 \leftarrow$ WR/WoR random numbers from $[0, m)$;
$\mathcal{S}.\boldsymbol{E}_r \leftarrow$ `TakeRandomElements`$(R_i, R_i.W, Y_i)$;
Update random numbers for its children relations;
for $i \leftarrow 2$ **to** k **do**
  $R_c(\boldsymbol{E}_c) \leftarrow$ the relation of o_i;
  $R_p(\boldsymbol{E}_p) \leftarrow$ the parent relation of R_c;
  $K \leftarrow \boldsymbol{E}_c \cap \boldsymbol{E}_p$ is the join key;
  $\mathcal{S}.\boldsymbol{E}_p \leftarrow$ `TakeRandomElements`$(R_c, R_c.W, \mathcal{S}.K, Y_i)$;
  Update random numbers for its children relations;
return $\mathcal{S}$

Theorem 4.2. *Algorithm 8 returns a uniform random sample of size s from $\mathcal{J} = R_1 \bowtie \cdots \bowtie R_k$ with probability $1/|\mathcal{J}|$. The circuit has $O(N \log^2 N + s \log^2 s)$ size and $O(\log^2 N + \log^2 s)$ depth.*

Proof. We establish two claims:

Correctness: In the bottom-up weight update procedure, once a relation becomes a leaf node, all its children have propagated their weights to the relation. This ensures that all weights in the relation are properly calculated, and the total sum m equals the actual join size. The reverse BFS traversal order guarantees that all child nodes are processed before their parent node.

We then define a projection from $[0, m)$ to the full join result $\mathcal{J}$. For two distinct random numbers $x_1, x_2 \in [0, m)$, our algorithm returns different tuples, as the random numbers differ in at least one relation. Since $|\mathcal{J}| = m$, the projection is bijective.

With this bijection, we use a random number generation circuit to uniformly and independently sample s random numbers from $[0, m)$. The top-down sampling procedure then enables sampling without materializing the full join. Consequently, our algorithm returns a uniform and independent random sample, where each tuple has a sample probability of $1/|\mathcal{J}|$.

Complexity: the bottom-up weight updation requires $k-1$ calls of weight updation, each incurring a circuit of size $O(N \log^2 N)$ and

depth $O(\log^2 N)$; the top-down sampling involves k calls of taking random elements, each with size $O((N+s)\log^2(N+s))$ and depth $(\log^2(N+s))$. For constant k, the total circuit has $O(N\log^2 N + s\log^2 s)$ size and $O(\log^2 N + \log^2 s)$ depth. □

Corollary 4.3. *By leveraging generic MPC protocols (e.g. Yao's GC, BGW, GMW) to implement Algorithm 8, we present a secure sampling over joins protocol that reveals nothing beyond public information including the input size, sample size and the schema.*

Proof. The proposed secure sampling protocol is fundamentally circuit-based, and its security guarantees are inherited from the underlying secure computation framework implementing it. For example, using Yao's GC [51] makes the algorithm secure against semi-honest adversary in two-party setting; using MP-SPDZ [30] makes it secure against malicious adversary in multi-party setting.

We now analyze the leakage of the circuits. The circuit hides internal values but cannot conceal its own size. In our design, we ensure that all circuit sizes depend solely on public information (e.g., input relation sizes and the sample size). In Algorithm 6, the inputs are two relations R and S, with sizes $|R|$ and $|S|$, respectively. The intermediate table T has size $|R|+|S|$, and the output is an updated relation R of the same size $|R|$. In Algorithm 7, the inputs are the relation and its weights both of size $|R|$ and s random numbers for sampling. The intermediate table T has size $|R|+s$. The output is sampled tuples of size s. In Algorithm 8, the traversal order is determined via BFS on the join tree, which is derived from the public schema. The sampling procedure relies on the relational schema (containing the join structure and attribute names), which is also public. We conclude that: (1) Computational components (e.g., join keys, tuple weights, sampled tuples) are protected by the circuit's guarantees; (2) The revealed circuit sizes and sampling order depend only on public parameters. Thus, the protocol implementing the circuit exactly simulates Algorithm 1 without explicitly materializing the full join. □

Corollary 4.4. *Setting $s = |\mathcal{J}|$ and iterating $y \in [0, |\mathcal{J}|)$ transforms Algorithm 8 into a secure acyclic join circuit with $\tilde{O}(N+M)$ complexity, where $M = |\mathcal{J}|$ is the output size. This matches the optimal asymptotic complexity of Yannakakis algorithm [50] for plaintext joins up to a polylogarithmic factor.*

5 EXTENSIONS

5.1 Queries with Group-By

Although we can answer group-by queries using uniform samples, this approach performs well only when group sizes are balanced. In Example 1.1, patients with "Type-2 Gaucher" disease vastly outnumber those with rare cancers like "Glioblastoma". Consequently, uniform sampling may fail to adequately represent rare subgroups. To address applications such as imbalanced class analysis in machine learning or compliance audits requiring subgroup guarantees, stratified sampling which ensures representation of rare groups in the sample is introduced.

We assume that the group-by attributes (or stratum identifiers) are confined to the root relation R_1, ensuring that their weights can be precomputed independently of other relations. Instead of computing the total weight of R_1 as in uniform sampling, we compute the weight for each stratum individually. Given a stratum sequence for sampling or a sizing policy for each stratum [35], we draw stratified samples from the root relation. Specifically, for each stratum, we generate the required number of random values within its weight range, as discribed in Section 3.3. The remaining steps follow the same as uniform sampling.

5.2 Queries with Comparisons

We extend the protocol to support queries with comparisons (i.e., selection predicates). The first category is type-1 predicates, which involves attributes from a single relation (e.g., $R_1.A > 5$ or $R_1.A < R_1.B$). These are handled through pre-filtering: for a tuple r and selection condition $\phi(\cdot)$ in relation R_i, we employ a parallel comparison circuit to update $w_r \leftarrow 0$ if $\phi(r)$ is unmet prior to weight updates. This incurs an additional $O(N)$-size circuit and ensures only qualifying tuples contribute to the join result.

The second category is type-2 predicates, which span multiple relations (e.g., $R_1.A > R_2.B$). These require post-sampling validation: After sampling from the join result without such comparisons, we apply a parallel comparison circuit to mark samples violating the predicate as dummy tuples. In Example 1.1, we first sample s tuples from $\texttt{M} \bowtie \texttt{I}$, and then mark those failing `M.datetime` > `I.datetime` as dummies. Filtering incurs $O(s)$-size circuit but introduces a failure probability, i.e., the final sample may have fewer than s valid tuples. However, valid samples remain uniformly distributed due to the independence of the sampling process.

5.3 Sampling over Cyclic Joins

Cyclic joins (e.g., triangle joins) lack a join tree structure and is a tough problem even in plaintext setting: There is no linear cost solution to compute full cyclic joins even on plaintext [39]. We cannot break this barrier in secure setting either. In this section, we present two different methods to address this challenge:

The first method is to convert the cyclic hypergraph into an acyclic query by renaming attributes. For example, in a triangle join $R(A,B) \bowtie S(B,C) \bowtie T(C,A)$, rename $T.A$ to $T.D$ to form a line join and introduce a new equality constraint $A = D$. Then we apply our sampling protocol to draw a sample from the acylclic join and use a parallel comparison circuit to verify equality constraints (e.g. $A = D$) and discard invalid tuples. The total complexity is still $\tilde{O}(N+s)$. This method avoids explicit join materialization but risks empty outputs for sparse joins.

The second method is using GHD to partition the hypergraph into bags with bounded treewidth w. For each bag, compute the full join using worst-case optimal join circuits [49]. Then construct an acyclic join tree over the bags and apply our protocol to sample from the bag-level join tree. Each bag's join requires $\tilde{O}(N^w)$ complexity and sampling from the bag-level join tree incurs $\tilde{O}(N^w+s)$ overhead. In this method, all sampled results are valid but the overall complexity is costly, but still outperforms the full materialization method.

In conclusion, the first method prioritizes efficiency and works well when most join results can pass the conditions, while the

second one ensures validity at the cost of higher complexity, which performs better when the treewidth is not large.

6 SYSTEM IMPLEMENTATION

6.1 System Architecture

We developed a prototype system SSOJ implementing secure sampling over joins in the semi-honest three-server honest majority (3PC) model, which is widely used in many secure computation tasks [1, 11, 28, 34].

There are three non-colluding semi-honest servers to perform secure computation, where servers follow the protocol but may infer information from intermediate data. There are one or more data owners who provide private datasets. They distrust any single server, and distribute trust across the three servers by secret-sharing their private data. Data are contributed by any number of data owners, which may partition the data either horizontally (each data owner contributes a subset of tuples of the same table), vertically (each data owner contributes a different table), or in a mixed fashion.

The protocol ensures that any adversary corrupting up to one server and all-but-one data owners learns nothing beyond the public metadata (e.g., input/output sizes). This aligns with the real-ideal paradigm (Section 2.3.1) that can protect honest data owners' data. The protocol's output remains secret-shared, enabling seamless integration with downstream secure computation tasks like query processing [35] and machine learning [1], without requiring intermediate reconstruction. This composability is critical for privacy-preserving analytics pipelines.

6.2 Implementation Details

The system is built upon the ABY3 framework [1]. It provides the implementation of circuit-based computations, including addition, comparison, division, modulo, and etc in 3PC model. The communication costs is proportional to the circuit size and the communication round is proportional to the circuit depth.

We adopt the Boolean replicated secret sharing: For a given ℓ-bit secret value v, pick two random ℓ-bit numbers v_0, v_1 independently, and set $v_2 = v \oplus v_0 \oplus v_1$ where $\oplus$ denotes bitwise XOR. The three shares are (v_0, v_1), (v_1, v_2), (v_2, v_0). It is clear that any two shares reconstruct v, while any single share just consists of two information-less random numbers.

We built the prefix sum circuit in $O(N)$ cost and $O(\log N)$ rounds. We do not use the sorting and permutation circuit, but try efficient function-specific protocols. [8] provides a 3PC permutation protocol in $O(N)$ costs and $O(1)$ rounds and [27] provides a 3PC sorting in $O(N \log N)$ cost in $O(\log N)$ rounds. With these efficient implementations, SSOJ achieves $O(N \log N + s \log s)$ costs and $O(\log N + \log s)$ rounds.

7 EXPERIMENTS

7.1 Experiment Setup

Baselines. We compare our SSOJ protocol with three baselines: (1) FJ materializes the full join result [34] but releases the join sizes; (2) Sample(m) and Sample(n^k) take a WR sample [35] from two flat tables of sizes m (actual join size) and n^k (worst-case join size), respectively. The two baselines serve as the lower bounds of the join-then-sample method, without and with join size protection, as they do not include the cost of the join phase.

Datasets and Queries. We use graph pattern queries in the experiments with real-world graphs from SNAP (Stanford Network Analysis Project) [3], summarized in Table 2. We store edge information as a relation Graph(src, dst). We evaluate 5 graph join queries[3], including three line joins, Line-2, Line-3, and Line-4, and the Tree and Star joins.

Graph	#Edges	#Line-2	#Line-3	#Line-4	#Tree	#Star
BitcoinAlpha	24186	0.12	4.28	186	26.7	556
BitcoinOtc	35592	0.23	8.31	416	88.3	1604
Wiki	103689	0.45	20.3	915	480	4212
Epinions	508837	3.99	372	37898	2826	83360
DBLP	1049866	0.71	6.75	83.5	140	277

Table 2: Graph datasets and their statistics. #Edges is the input size of graph datasets. #Q is the full join size of Q over the corresponding graph datasets in units of 10 million ($\times 10^7$).

Experimental environment. Our experiments were executed on three servers each equipped with an Intel 2.3GHz processor, 32GB RAM. The latency and bandwidth between the servers are 10ms and 1Gb/s, respectively. We executed each query 10 times on every engine, reporting the mean runtime and the maximum communication costs (including data sent and received) among the three servers. Each query runs at most 12 hours to obtain meaningful results.

7.2 Experiment Results

Time and communication. Figure 5 presents the time and communication costs across various test queries and datasets. For these experiments, we set the sample size $s = 2^{10}$. Bars touching the axis boundary indicate that the runtime exceeds the 12-hour threshold.

It is obvious that, across all tested queries and datasets, our protocol significantly outperform the baseline protocols. Our approach consistently completes all queries within a few hundred seconds and 100GB communication, making the overall evaluation both practical and acceptable. Sample(n^k), which provides the same security guarantee as our protocol, cannot finish all the queries except the Line-2 of BitcoinAlpha, whose result contains roughly 6×10^8 tuples. For Line-4 of DBLP of more than 10^{24} result tuples, Sample(n^k) needs to take 10^{20} seconds to compute in a rough estimation - even longer than the age of the universe. The two relaxed-security baselines, FJ and Sample(m), which reveal the join sizes, have similar costs, as their complexities are proportional to m. Our protocol still yields significant improvements ranging from 1 to 4 orders of magnitude across the first 4 datasets. For example, for the Line-2 query on BitcoinAlpha, materializing the full join takes 40s, and sampling from it takes 26s, whereas SSOJ completes in <3s. The only exception is on DBLP, where the statistics from Table 2 show the graph is sparse: m is close to n and the gap becomes small. Different from FJ and Sample(m) that perform based on join sizes,

[3]Find the queries in https://github.com/MPC-Query-Processing/SSOJ/blob/main/queries.md

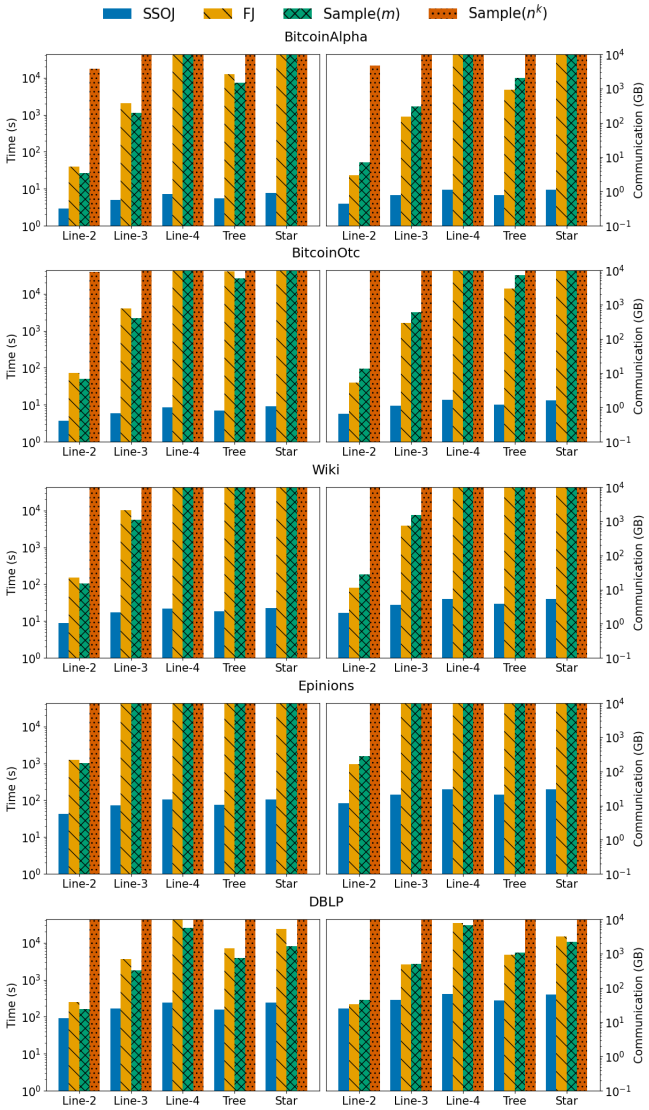


Figure 5: Time and communication costs of test queries.

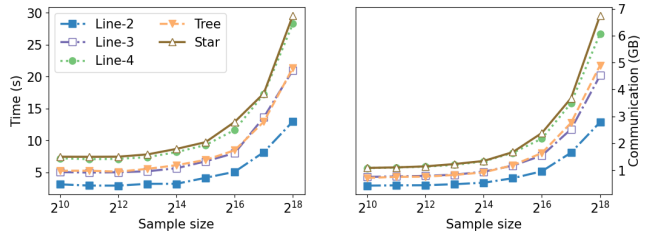


Figure 6: Costs varying different sample sizes.

SSOJ performs stably for different types of joins, as long as the number of relations is fixed (e.g., Line-3 and Tree join).

Impact of sample size. Based on our theoretical results, the complexity of our protocol is $\tilde{O}(N+s)$, or more precisely $O((k-1)\cdot 2n\log 2n + k\cdot(n+s)\log(n+s))$. We conducted experiments to measure the performance when changing different sample size from 2^{10} to 2^{18} on BitcoinAlpha. As shown in Figure 6, when s is no more than 2^{14} ($s < n$), the performances are similar as they are dominated by the Update procedure; when s is larger than 2^{15} ($s > n$), the performances rely on the Sample procedure and increased a lot. The results tell us that taking a sample with size roughly equals to n is a good choice - It can get a better representation than small sample size but do not incur much more sampling costs. In practice, one can choose small sample size from $s \approx n$ first, and try larger sample sizes if the representation does not meet the requirements.

Models	Time (s)	Comm (GB)
2PC semi-honest	138.99	16.39
2PC malicious	670.91	83.11
3PC semi-honest	11.91	0.89
3PC malicious	1291.57	98.03
4PC semi-honest	13475.6	35.83

Table 3: Costs of sample Line-2 query in different models.

We also evaluated the detailed sampling procedure across different join types and compared SSOJ with plaintext methods. The results are available in the full version[4].

7.3 Scalability

We also implemented our work using MP-SPDZ framework [2], which supports various MPC protocols across multiple security models. We evaluated two-/three-/four-party settings under semi-honest and malicious adversaries, assuming at most one corrupted party. The detailed experimental settings, implementation details, and supplemtary results are deferred to the full version.

The results of sampling $s = 1024$ from the Line-2 query on BitcoinAlpha ($n = 24186$) are summarized in Table 3. In MP-SPDZ, the sorting primitive used is radix sort [26] which incurs $O(n\log^2 n)$ costs over $O(\log^2 n)$ rounds. This complexity holds for all settings except the semi-honest 3PC setting, which achieves $O(n\log n)$ costs and $O(\log n)$ rounds [9]. Consequently, the semi-honest 3PC setting yields significantly better performance. We believe that model-specific optimizations (e.g., use oblvious quicksort [27] for 2PC) could lead to substantial additional performance improvements.

8 CONCLUSION

In this paper, we presented a circuit-based algorithm for secure sampling over joins, achieving $\tilde{O}(N+s)$ complexity for input size N and sample size s. It is the first secure sampling algorithm that can be executed without materializing the full join. It can also generalize to all secure computation systems. The experimental results show the algorithm significantly outperforms state-of-the-art methods.

There are also some interesting open questions: (1) Is there any effcient solution for type-2 predicates and cyclic joins without post-filtering? (2) The current sampling over joins algorithm consider equi-joins. What about other types of join such as θ-joins? (3) Can we adopt weaker security guarantees such as differentially obliviousness to further improve the efficiency? (4) Handling dynamic datasets in secure settings [53] has sparked significant research interest. How can we efficiently manage updates during sampling over joins [21]?

[4] https://github.com/MPC-Query-Processing/SSOJ/blob/main/full.pdf

REFERENCES

[1] ABY3. https://github.com/ladnir/aby3.

[2] MP-SPDZ. https://github.com/data61/MP-SPDZ.

[3] SNAP. https://snap.stanford.edu/snap/.

[4] Mahmoud Abo Khamis, Hung Q. Ngo, and Dan Suciu. 2017. What Do Shannon-Type Inequalities, Submodular Width, and Disjunctive Datalog Have to Do with One Another?. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Chicago, Illinois, USA) *(PODS '17)*. Association for Computing Machinery, New York, NY, USA, 429–444. https://doi.org/10.1145/3034786.3056105

[5] Swarup Acharya, Phillip B. Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. 1999. Join synopses for approximate query answering. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data (SIGMOD '99)*. Association for Computing Machinery, New York, NY, USA, 275–286. https://doi.org/10.1145/304182.304207

[6] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 2013. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *Proceedings of the 8th ACM European Conference on Computer Systems*. 29–42.

[7] M. Ajtai, J. Komlós, and E. Szemerédi. 1983. An 0(n Log n) Sorting Network. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing (STOC '83)*. Association for Computing Machinery, 1–9.

[8] Toshinori Araki, Jun Furukawa, Kazuma Ohara, Benny Pinkas, Hanan Rosemarin, and Hikaru Tsuchida. 2021. Secure Graph Analysis at Scale. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 610–629.

[9] Gilad Asharov, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Ariel Nof, Benny Pinkas, Katsumi Takahashi, and Junichi Tomida. 2022. Efficient Secure Three-Party Sorting with Applications to Data Analysis and Heavy Hitters. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 125–138. https://doi.org/10.1145/3548606.3560691

[10] Gilad Asharov, Koki Hamada, Ryo Kikuchi, Ariel Nof, Benny Pinkas, and Junichi Tomida. 2023. Secure Statistical Analysis on Multiple Datasets: Join and Group-By. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 3298–3312.

[11] Saikrishna Badrinarayanan, Sourav Das, Gayathri Garimella, Srinivasan Raghuraman, and Peter Rindal. 2022. Secret-Shared Joins with Multiplicity from Aggregation Trees. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 209–222.

[12] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. 2007. On Acyclic Conjunctive Queries and Constant Delay Enumeration. In *Computer Science Logic*. Springer Berlin Heidelberg, Berlin, Heidelberg, 208–222.

[13] K. E. Batcher. 1968. Sorting Networks and Their Applications. In *Proceedings of the AFIPS Spring Joint Computing Conference*. 307–314.

[14] Johes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel Kho, and Jennie Rogers. 2017. SMCQL: Secure Querying for Federated Databases. *Proceedings of the VLDB Endowment* 10, 6 (2017), 673–684.

[15] Johes Bater, Yongjoo Park, Xi He, Xiao Wang, and Jennie Rogers. 2020. SAQE: practical privacy-preserving approximate query processing for data federations. *Proc. VLDB Endow.* 13, 12 (July 2020), 2691–2705. https://doi.org/10.14778/3407790.3407854

[16] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. 1988. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*. 1–10.

[17] Ran Canetti. 2000. Security and Composition of Multiparty Cryptographic Protocols. *J. Cryptol.* 13, 1 (jan 2000), 143–202.

[18] Zhao Chang, Dong Xie, Feifei Li, Jeff M. Phillips, and Rajeev Balasubramonian. 2022. Efficient Oblivious Query Processing for Range and kNN Queries . *IEEE Transactions on Knowledge & Data Engineering* 34, 12 (Dec. 2022), 5741–5754. https://doi.org/10.1109/TKDE.2021.3060757

[19] Zhao Chang, Dong Xie, Sheng Wang, and Feifei Li. 2022. Towards Practical Oblivious Join. In *Proceedings of the 2022 International Conference on Management of Data*. 803–817.

[20] Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. 1999. On random sampling over joins. *SIGMOD Rec.* 28, 2 (June 1999), 263–274. https://doi.org/10.1145/304181.304206

[21] Binyang Dai, Xiao Hu, and Ke Yi. 2024. Reservoir Sampling over Joins. *Proc. ACM Manag. Data* 2, 3, Article 118 (May 2024), 26 pages. https://doi.org/10.1145/3654921

[22] Bolin Ding, Silu Huang, Surajit Chaudhuri, Kaushik Chakrabarti, and Chi Wang. 2016. Sample + Seek: Approximating Aggregates with Distribution Precision Guarantee. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 679–694. https://doi.org/10.1145/2882903.2915249

[23] O. Goldreich, S. Micali, and A. Wigderson. 1987. How to Play ANY Mental Game. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*. 218–229.

[24] Georg Gottlob, Nicola Leone, and Francesco Scarcello. 1999. Hypertree Decompositions and Tractable Queries. In *Proceedings of the Eighteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (, Philadelphia, Pennsylvania, USA,) *(PODS '99)*. Association for Computing Machinery, New York, NY, USA, 21–32. https://doi.org/10.1145/303976.303979

[25] Peter J. Haas and Joseph M. Hellerstein. 1999. Ripple joins for online aggregation. *SIGMOD Rec.* 28, 2 (June 1999), 287–298. https://doi.org/10.1145/304181.304208

[26] Koki Hamada, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. 2014. Oblivious Radix Sort: An Efficient Sorting Algorithm for Practical Secure Multi-party Computation. Cryptology ePrint Archive, Paper 2014/121. https://eprint.iacr.org/2014/121

[27] Koki Hamada, Ryo Kikuchi, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. 2013. Practically Efficient Multi-party Sorting Protocols from Comparison Sort Algorithms. In *Information Security and Cryptology – ICISC 2012*. 202–216.

[28] Feng Han, Lan Zhang, Hanwen Feng, Weiran Liu, and Xiangyang Li. 2022. Scape: Scalable Collaborative Analytics System on Private Database with Malicious Security. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 1740–1753.

[29] Srikanth Kandula, Anil Shanbhag, Aleksandar Vitorovic, Matthaios Olma, Robert Grandl, Surajit Chaudhuri, and Bolin Ding. 2016. Quickr: Lazily Approximating Complex AdHoc Queries in BigData Clusters. In *Proceedings of the 2016 International Conference on Management of Data*. 631–646.

[30] Marcel Keller. 2020. MP-SPDZ: A Versatile Framework for Multi-Party Computation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1575–1590. https://doi.org/10.1145/3372297.3417872

[31] Simeon Krastnikov, Florian Kerschbaum, and Douglas Stebila. 2020. Efficient Oblivious Database Joins. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2132–2145.

[32] Richard E. Ladner and Michael J. Fischer. 1980. Parallel Prefix Computation. *J. ACM* 27, 4 (1980), 831–838.

[33] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. 2016. Wander Join: Online Aggregation via Random Walks. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 615–629. https://doi.org/10.1145/2882903.2915235

[34] Qiyao Luo, Yilei Wang, Wei Dong, and Ke Yi. 2024. Secure Query Processing with Linear Complexity. arXiv:2403.13492 [cs.CR]

[35] Qiyao Luo, Yilei Wang, Ke Yi, Sheng Wang, and Feifei Li. 2023. Secure Sampling for Approximate Multi-party Query Processing. *Proc. ACM Manag. Data* 1, 3, Article 219 (Nov. 2023), 27 pages. https://doi.org/10.1145/3617339

[36] Payman Mohassel and Peter Rindal. 2018. ABY3: A Mixed Protocol Framework for Machine Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 35–52. https://doi.org/10.1145/3243734.3243760

[37] 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. https://doi.org/10.1145/3372297.3423358

[38] Payman Mohassel and Yupeng Zhang. 2017. SecureML: A System for Scalable Privacy-Preserving Machine Learning . In *2017 IEEE Symposium on Security and Privacy (SP)*. 19–38. https://doi.org/10.1109/SP.2017.12

[39] Hung Q. Ngo. 2018. Worst-Case Optimal Join Algorithms: Techniques, Results, and Open Problems. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 111–124. https://doi.org/10.1145/3196959.3196990

[40] Frank Olken and Doron Rotem. 1995. Random sampling from databases: a survey. *Statistics and Computing* 5, 1 (March 1995), 25–42. https://doi.org/10.1007/BF00140664

[41] Yongjoo Park, Barzan Mozafari, Joseph Sorenson, and Junhao Wang. 2018. VerdictDB: Universalizing Approximate Query Processing. In *Proceedings of the 2018 International Conference on Management of Data*. Association for Computing Machinery, New York, NY, USA, 1461–1476.

[42] Benny Pinkas, Thomas Schneider, Oleksandr Tkachenko, and Avishay Yanai. 2019. Efficient Circuit-Based PSI with Linear Communication. In *Advances in Cryptology – EUROCRYPT 2019: 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19–23, 2019, Proceedings, Part III*. 122–153. https://doi.org/10.1007/978-3-030-17659-4_5

[43] Rishabh Poddar, Sukrit Kalra, Avishay Yanai, Ryan Deng, Raluca Ada Popa, and Joseph M. Hellerstein. 2021. Senate: A Maliciously-Secure MPC Platform for Collaborative Analytics. In *Proceedings of the 30th Conference on USENIX Security Symposium*.

[44] Sajin Sasy and Olga Ohrimenko. 2019. Oblivious Sampling Algorithms for Private Data Analysis. In *Advances in Neural Information Processing Systems*, Vol. 32.

[45] Adithya Vadapalli, Ryan Henry, and Ian Goldberg. 2023. DUORAM: A bandwidth-efficient distributed ORAM for 2- and 3-party computation. In *Proceedings of the 32nd USENIX Conference on Security Symposium*. Article 219, 18 pages.

[46] Abraham Waksman. 1968. A Permutation Network. *J. ACM* (1968), 159–163.

[47] Xiao Wang, Hubert Chan, and Elaine Shi. 2015. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 850–861. https://doi.org/10.1145/2810103.2813634

[48] Yilei Wang and Ke Yi. 2021. Secure Yannakakis: Join-Aggregate Queries over Private Data. In *Proceedings of the 2021 International Conference on Management of Data*. 1969–1981. https://doi.org/10.1145/3448016.3452808

[49] Yilei Wang and Ke Yi. 2022. Query Evaluation by Circuits. In *Proceedings of the 41st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. Association for Computing Machinery, 67–78.

[50] Mihalis Yannakakis. 1981. Algorithms for acyclic database schemes. In *VLDB*, Vol. 81. 82–94.

[51] Andrew C Yao. 1982. Protocols for secure computations. In *23rd Annual Symposium on Foundations of Computer Science*. IEEE, 160–164.

[52] Zhuoyue Zhao, Robert Christensen, Feifei Li, Xiao Hu, and Ke Yi. 2018. Random Sampling over Joins Revisited. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1525–1539. https://doi.org/10.1145/3183713.3183739

[53] Peizhao Zhou, Xiaojie Guo, Pinzhi Chen, Tong Li, Siyi Lv, and Zheli Liu. 2024. Shortcut: Making MPC-based Collaborative Analytics Efficient on Dynamic Databases. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 854–868. https://doi.org/10.1145/3658644.3690314