Subset Sampling over Joins*

ARYAN ESMAILPOUR, University of Illinois Chicago, USA
XIAO HU, University of Waterloo, Canada
JINCHAO HUANG, The Chinese University of Hong Kong, Hong Kong SAR
STAVROS SINTOS, University of Illinois Chicago, USA

Subset sampling (also known as Poisson sampling), where the decision to include any specific element in the sample is made independently of all others, is a fundamental primitive in data analytics, enabling efficient approximation by processing representative subsets rather than massive datasets. While sampling from explicit lists is well-understood, modern applications—such as machine learning over relational data—often require sampling from a set defined implicitly by a relational join. In this paper, we study the problem of *subset sampling over joins*: drawing a random subset from the join results, where each join result is included independently with some probability. We address the general setting where the probability is derived from input tuple weights via decomposable functions (e.g., product, sum, min, max). Since the join size can be exponentially larger than the input, the naive approach of materializing all join results to perform subset sampling is computationally infeasible. We propose the first efficient algorithms for subset sampling over acyclic joins: (1) a *static index* for generating multiple (independent) subset samples over joins; (2) a *one-shot* algorithm for generating a single subset sample over joins; (3) a *dynamic index* that can support tuple insertions, while maintaining a one-shot sample or generating multiple (independent) samples. Our techniques achieve near-optimal time and space complexity with respect to the input size and the expected sample size.

CCS Concepts: • **Theory of computation → Sketching and sampling**.

Additional Key Words and Phrases: Join, Subset Sampling, Direct Access

ACM Reference Format:
Aryan Esmailpour, Xiao Hu, Jinchao Huang, and Stavros Sintos. 2026. Subset Sampling over Joins. *Proc. ACM Manag. Data* 4, 2 (PODS), Article 117 (May 2026), 26 pages. https://doi.org/10.1145/3801913

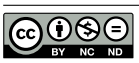
1 Introduction

Sampling is a cornerstone technique in modern data analytics and machine learning, allowing systems to trade a small amount of accuracy for significant gains in performance by processing a representative subset of data rather than the entire dataset [21, 44]. A particularly powerful primitive is *subset sampling* (also known as *Poisson sampling*), where the decision to include any specific element in the sample is made independently of all others, based on a specific probability or weight. This independence is often crucial for theoretical guarantees in randomized algorithms and allows for uncoordinated execution in distributed systems [25, 42].

While subset sampling from explicit lists of items is well-understood, a new challenge arises in modern data pipelines where the dataset of interest is not stored in a single table but is defined

*This work was partially supported by NSF grant IIS-2348919, and NSERC Discovery Grant.

Authors' Contact Information: Aryan Esmailpour, University of Illinois Chicago, Chicago, USA, aesmai2@uic.edu; Xiao Hu, xiaohu@uwaterloo.ca, University of Waterloo, Waterloo, Canada, xiaohu@uwaterloo.ca; Jinchao Huang, jchuang@se.cuhk.edu.hk, The Chinese University of Hong Kong, Shatin, Hong Kong SAR; Stavros Sintos, University of Illinois Chicago, Chicago, USA, stavros@uic.edu.



ACM 2836-6573/2026/5-ART117
https://doi.org/10.1145/3801913

implicitly as the result of a relational join. In this paper, we study the problem of Subset Sampling over Joins: efficiently drawing a random subset from the join results, where every join result is associated with a weight and is included in the sample independently with probability equal to its weight. Crucially, we focus on the standard setting where the weight of a join result is not arbitrary, but is derived from weights assigned to the input tuples via an aggregation function—such as the product, minimum, maximum, or summation of the component weights. A compelling motivation for this problem lies in dataset condensation (or coreset construction) for machine learning over multi-relational data [37, 46].

Example 1.1. Consider a high-stakes scientific computing task—virtual screening in drug discovery—where a researcher wishes to train a graph neural network to predict the efficacy of complex molecular interactions. The training data is defined by joining massive tables of *chemical compounds*, *protein targets*, and *metabolic pathways*. The full set of valid interaction subgraphs (the join result) is combinatorially explosive—often exceeding petabytes in size—making it impossible to materialize or exhaustively simulate. To make training feasible, the system must generate a condensed, representative dataset [6, 27]. Here, input tuples carry crucial domain-specific weights: a compound has a binding affinity score, and a protein target has a clinical relevance score. The importance of a resulting interaction tuple is naturally derived from these components (e.g., the product of affinity and relevance, representing the joint probability of success; or the minimum, representing a bottleneck constraint). By performing subset sampling with these derived weights, we create a manageable, high-quality summary of the chemical space.

However, performing this sampling efficiently is algorithmically difficult due to the implicit nature of the data. A naive approach would be to first compute the join results as well as their aggregated weights (e.g., the product of its components), and then apply the classic subset sampling algorithm. This strategy is prohibitively costly because of the "materialization bottleneck": the size of the join result can be exponentially larger than the input relations [5], which is very costly to compute and store. In this naive process, the algorithm would spend the vast majority of its time generating join results that are immediately discarded by the coin flips, wasting massive amounts of computation and memory. To the best of our knowledge, no prior work has addressed the problem of relational subset sampling except this baseline. In this paper, we aim to fill this gap by providing the first efficient subset sampling algorithms that operate directly on the input relations. By exploiting the join structure of the query and the decomposable nature of the weight functions, we can effectively "skip" over the vast sea of rejected tuples and directly generate the successful samples. Our collective results establish subset sampling over joins as a tractable database primitive, which will pave the way for scalable, interactive analytics over massive multi-relational datasets.

1.1 Problem Definition

Subset Sampling. Given a set of elements $S = \{e_1, e_2, \ldots, e_n\}$ and a function p that assigns each element $e \in S$ a probability $\mathrm{p}(e)$, a *subset sampling* query asks for a *subset sample* of S. A subset sample of S is a random subset $\mathbf{X} \in 2^S$, where each element $e \in S$ is sampled into $\mathbf{X}$ independently with probability $\mathrm{p}(e)$. More formally, the distribution of $\mathbf{X}$ is: $\mathbf{Pr}[\mathbf{X} = Y] = \left(\prod_{e\in Y} \mathrm{p}(e)\right)\left(\prod_{e\in S\setminus Y}(1-\mathrm{p}(e))\right), Y \subseteq S$. We refer to $\Psi = \langle S, \mathrm{p}\rangle$ as a *subset sampling problem instance*. We denote the expected size of $\mathbf{X}$, i.e., $E[|\mathbf{X}|] = \sum_{e\in S} \mathrm{p}(e)$, as μ_Ψ.

(Natural) Joins. Let **att** be a set where each element is called an *attribute*, and **dom** be another set where each element is called a *value*. A *tuple* over a set $U \subseteq$ **att** is a function $\boldsymbol{u} : U \to$ **dom**. For any subset $U' \subseteq U$, define $\boldsymbol{u}[U']$ as the tuple $\boldsymbol{u}'$ over U' such that $\boldsymbol{u}'(A) = \boldsymbol{u}(A)$ for every $A \in U'$. We also call U the support attributes of $\boldsymbol{u}$. A *relation* is a set R of tuples over the same set U of attributes; we call U the *schema* of R, a fact denoted as $\mathrm{schema}(R) = U$. Given a subset

$U \subseteq \text{schema}(R)$, the *projection of R on U*—denoted as $R[U]$—is a relation with schema U defined as $R[U] = \{\text{tuple } \boldsymbol{u} \text{ over } U \mid \exists \text{ tuple } \boldsymbol{v} \in R \text{ s.t. } \boldsymbol{v}[U] = \boldsymbol{u}\}$. We represent a *natural join* (henceforth, simply a "join") as a set Q of relations. We denote the number of relations in Q as $k = |Q|$. The input size of Q is defined as the total number of tuples in Q, denoted as $N = \sum_{R \in Q} |R|$. Define $\text{attset}(Q) = \bigcup_{R \in Q} \text{schema}(R)$. The join result is the following relation over $\text{attset}(Q)$:

$$\text{Join}(Q) = \{\text{tuple } \boldsymbol{u} \text{ over } \text{attset}(Q) \mid \forall R \in Q, \boldsymbol{u}[\text{schema}(R)] \in R\}.$$

The join Q can be characterized by a *schema graph* $G_Q = (V, E)$, which is a hypergraph where each vertex in V is a distinct attribute in $\text{attset}(Q)$, and each edge in E is the schema of a distinct relation in Q. The set E may contain identical edges because two (or more) relations in Q can have the same schema. The term "hyper" suggests that an edge can have more than two attributes. The join Q is *acyclic* [50] if G_Q is acyclic. A hypergraph is acyclic if there exists a tree $\mathcal{T}$, called a *join tree*, whose nodes have a one-to-one correspondence with the relations in Q (i.e., hyperedges in E), such that for any attribute $A \in V$, the set of nodes corresponding to relations containing A forms a connected subtree in $\mathcal{T}$. A join that is not acyclic is called a *cyclic join*.

As a variant of join, the semi-join between R_1 and R_2 is defined as $R_1 \ltimes R_2 = R_1 \bowtie (R_2[\text{schema}(R_1) \cap \text{schema}(R_2)])$, which returns all tuples in R_1 that can be joined with some tuple from R_2. When the second operand is a single tuple $\boldsymbol{v}$, $R_1 \ltimes \boldsymbol{v}$ is a shorthand for $R_1 \ltimes \{\boldsymbol{v}\}$. As a special case, $R \ltimes \emptyset = R$.

Subset Sampling over Joins. Let $Q = \{R_1, R_2, \ldots, R_k\}$ be a join as defined above. Each relation $R_j \in Q$ is associated with a function $\mathrm{p}_j : R_j \to [0, 1]$ that assigns a weight from $[0, 1]$ to each tuple in it. For each join result $\boldsymbol{u} \in \text{Join}(Q)$, its weight is a function $\mathcal{F}$ defined on the weights of these tuples that constitute this join result:

$$\mathrm{p}(\boldsymbol{u}) = \mathcal{F}\left(\mathrm{p}_1(\boldsymbol{u}[\text{schema}(R_1)]), \mathrm{p}_2(\boldsymbol{u}[\text{schema}(R_2)]), \ldots, \mathrm{p}_k(\boldsymbol{u}[\text{schema}(R_k)])\right), \tag{1}$$

where $\mathcal{F}$ can be the MAX, MIN, PRODUCT, or SUM function. It is required that $\mathrm{p}(\boldsymbol{u}) \in [0, 1]$. Given a join instance Q and a set of associated weight functions $\{\mathrm{p}_j\}_{j \in \{1,2,\ldots,k\}}$, a *relational subset sampling* query is a subset sampling query to the subset sampling instance $\Psi = \langle \text{Join}(Q), \mathrm{p} \rangle$, i.e., asks to sample a random subset $\mathbf{X}$ of $\text{Join}(Q)$, where each join result $\boldsymbol{u}$ is sampled into $\mathbf{X}$ independently with probability defined by its weight $\mathrm{p}(\boldsymbol{u})$. More formally, the distribution of $\mathbf{X}$ is

$$\mathbf{Pr}[\mathbf{X} = Y] = \left(\prod_{\boldsymbol{u} \in Y} \mathrm{p}(\boldsymbol{u})\right)\left(\prod_{\boldsymbol{u} \in \text{Join}(Q) \setminus Y} (1 - \mathrm{p}(\boldsymbol{u}))\right), Y \subseteq \text{Join}(Q). \tag{2}$$

We denote the expected size of $\mathbf{X}$, i.e., $E[|\mathbf{X}|] = \sum_{\boldsymbol{u} \in \text{Join}(Q)} \mathrm{p}(\boldsymbol{u})$, as μ_Ψ.

The first problem we study is an *indexing* (data structure) problem, where we wish to have an index that supports drawing multiple independent subset samples from join results efficiently:

> **Problem 1.2** (Indexed Subset Sampling over Joins)**.** Given a join instance Q and a set of associated weight functions $\{\mathrm{p}_i\}_{i \in \{1,2,\ldots,k\}}$, the goal is to construct an index for answering subset sampling queries to the instance $\langle \text{Join}(Q), \mathrm{p} \rangle$. Additionally, the random subsets returned by distinct queries must be independent.

We are interested in the preprocessing time t_p and space usage of the index constructed, as well as the time t_s for answering a relational subset sampling query from the index.

The second problem we study is a *one-shot* version of relational subset sampling where we wish to output one subset sample from join results efficiently:

> **Problem 1.3** (One-shot Subset Sampling over Joins)**.** Given a join instance Q and a set of associated weight functions $\{\mathrm{p}_i\}_{i \in \{1,2,\ldots,k\}}$, the goal is to compute one subset sample from the instance $\langle \text{Join}(Q), \mathrm{p} \rangle$.

In this case, we are interested in the total answering time. Although any solution to Problem 1.2 gives a solution to Problem 1.3 with total time $O(t_p + t_s)$, this may not be efficient enough.

Dynamic Subset Sampling over Joins. We also consider the dynamic setting with tuple insertions[1]. Now, each insertion is a triple $\langle \boldsymbol{u}, R_h, \eta, p\rangle$ for $\eta \in \mathbb{Z}^+$, $p \in [0,1]$ and $R_h \in Q$, indicating that tuple $\boldsymbol{u}$ is inserted into relation R_h at timestamp η with weight p. We follow the set semantics, so inserting a tuple into a relation that already has it has no effect. Consider the stream of input tuples, ordered by their timestamp. Let Q^η be the join defined by the first η tuples of the stream, and set $Q^0 = \emptyset$. Also, let $\{\mathrm{p}_i^\eta\}_{i\in\{1,2,\ldots,k\}}$ be the associated weight functions for Q^η. Each join result $\boldsymbol{u} \in Q^\eta$ has its probability $\mathrm{p}^\eta(\boldsymbol{u})$ defined similar to (1). We use N to denote the length of the stream, which is only used in the analysis. The algorithms will not need the knowledge of N, so they work over an unbounded stream. We have the corresponding versions of both problems in the dynamic setting:

Problem 1.4 (Dynamic Indexed Subset Sampling over Joins)**.** Suppose tuples come in a streaming fashion. The goal is to maintain an index for answering subset sampling queries to the instance $\langle \mathrm{Join}(Q^\eta), \mathrm{p}^\eta\rangle$ for every timestamp $\eta \in \mathbb{Z}^+$. Additionally, the random subsets returned by distinct queries must be independent.

Problem 1.5 (Dynamic One-shot Subset Sampling over Joins)**.** Suppose tuples come in a streaming fashion. The goal is to maintain one subset sample from the instance $\langle \mathrm{Join}(Q^\eta), \mathrm{p}^\eta\rangle$ for every timestamp $\eta \in \mathbb{Z}^+$.

For the dynamic index, we are interested in the maintenance time t_u of the index when a tuple is inserted, and the time t_s for answering a relational subset sampling query from the index. For the dynamic one-shot problem, we care about the total running time. Again, any solution to Problem 1.4 yields a solution to Problem 1.5 with total time $O(t_u \cdot N + t_s)$.

For all versions of the relational subset sampling problem, we study the data complexity [1] and analyze the complexity in terms of the data-dependent quantities (such as input size N and expected size of subset samples μ_Ψ), while taking the schema size of Q (i.e., $|V|$ and $|E|$) as a constant.

Computation Model. We discuss our algorithms in the *real RAM* model of computation [10, 45]. In particular, we assume that the following operations take constant time: (i) accessing a memory location; (ii) generating a random value from the standard uniform distribution $\mathsf{Uniform}(0,1)$; and (iii) performing basic arithmetical operations involving real numbers, such as addition, multiplication, division, comparison, truncation, and evaluating fundamental functions like log. We denote by $\mathsf{Geometric}(p)$ the geometric distribution over $\{0,1,\ldots\}$ with probability mass function $\mathbf{Pr}[X = k] = (1-p)^k p$. We denote by $\mathsf{TruncatedGeometric}(p,n)$ the geometric distribution conditioned on the value being strictly less than n, i.e., with support $\{0,1,\ldots,n-1\}$. Under the RAM model, we can generate a random value x from $\mathsf{Geometric}(p)$ in $O(1)$ time [11] by setting $x = \lfloor \frac{\log \mathsf{Uniform}(0,1)}{\log(1-p)} \rfloor$. Additionally, we can generate a random value x from $\mathsf{TruncatedGeometric}(p,n)$ in $O(1)$ time [11] by setting $x = \lfloor \frac{\log(1-q\cdot\mathsf{Uniform}(0,1))}{\log(1-p)} \rfloor$, where $q = 1-(1-p)^n$.

Math Conventions. For an integer $x \in \mathbb{Z}^+$, the notation $[x]$ denotes the set $\{1,2,\ldots,x\}$, and $[\![x]\!]$ denotes the set $\{0,1,\ldots,x\}$. The notation $\bigsqcup$ denotes the disjoint union of sets. Given an ordered set of elements X and a pair of elements $x, x' \in X$, we use $x \prec x'$ to indicate that x is smaller than x', and $x \preceq x'$ to indicate that x is no larger than x'. We use $\log x$ to denote $\log_2 x$.

[1]In the fully dynamic case with both insertions and deletions, the subset sampling problem (with all weights as 1) is at least as hard as maintaining the join results. For general (more precisely, a non-hierarchical) join query, conditioned on the Online Matrix-Vector (OMv) multiplication conjecture, the update time must be $\Omega(N^{0.5-\epsilon})$ for every $\epsilon > 0$ just to maintain the Boolean answer, when both insertions and deletions are allowed [8]. Even for this Boolean problem, establishing a matching upper bound is still open, except for some specific joins.

To help understanding, we list the main notations used by this paper in Table 2.

1.2 Related Work

Uniform Sampling over Joins. Chaudhuri et al. [14] and Olken [44] first established the uniform sampling over join problem, which returns a uniform sample from the join results. Zhao et al. [52] introduced a linear space index for drawing uniform independent samples from acyclic joins. Later, Chen and Yi [16] investigated the cyclic joins and their results were improved to be conditionally optimal by two independent works [24, 36]. Huang et al. [33] investigated uniform sampling for acyclic joins with selection predicates. Dai et al. [23] proposed a reservoir sampling framework for maintaining uniform samples over joins in streaming settings.

Non-uniform Sampling over Joins. Non-uniform sampling has been extensively studied in the context of online aggregation (OLA). The ripple join [30] and its variants generalize nested-loop joins to incrementally estimate aggregates with confidence intervals. Similarly, wander join [38] uses random walks over the join graph to provide unbiased estimators for aggregates like SUM or COUNT. These approaches differ fundamentally from ours: they are designed to estimate scalar statistics using algorithmic probabilities that minimize variance, whereas our goal is to materialize a concrete subset of independent samples according to user-defined importance weights.

Despite the substantial body of work on sampling over joins, we are not aware of any existing approach that can be adapted to support subset sampling over joins, indicating the need for novel techniques. A straightforward approach to subset sampling over joins is to first materialize the join result and then build a standard index over all output tuples. However, the size of the join result can be orders of magnitude larger than that of the input database, making this approach prohibitively inefficient. More broadly, there is a growing line of work on solving optimization problems on join results without fully materializing the join [2–4, 13, 15, 17, 18, 22, 26, 34, 35, 37, 40, 41, 46–49].

Post-Acceptance Note. Concurrent to our work, a nice work [7] independently studied the same core statistical problem in slightly different settings. Their problem formulation assumes the sampling probability of a join result is dictated by a single attribute originating from a single base relation. In contrast, our problem definition is broader, supporting probabilities derived from the aggregation of tuple weights across multiple relations via decomposable functions (e.g., PRODUCT, MIN, MAX, SUM). Algorithmically, their work achieves an $O(N)$ index construction time and $O(\kappa \log N)$ sampling time (where κ is the sample size), with a focus on practical implementations in main-memory column stores using the Shredded Yannakakis framework.

1.3 Our Results

Our main results are summarized in Table 1. In this paper, we focus on acyclic joins, and all of our algorithms can be extended to cyclic joins using the standard tree decomposition approach [29]. In this transformation, the effective input size scales from N to N^w, where $w \geq 1$ represents the fractional hypertree width (or more complex measures such as #submodular width [39]). Consequently, in our complexity results, linear dependencies on N are replaced by N^w, while logarithmic dependencies (e.g., $\log N$, $\log \log N$) remain unchanged, as the width w is treated as a constant under data complexity. We address three primary scenarios:

- **(Section 3)** For the static setting, we design an index that can be constructed in $O(N \log N \log \log N)$ time and $O(N \log N)$ space, supporting subset sampling queries in expected $O(1 + \mu_\Psi \log N)$ time, where N is the input size and μ_Ψ is the expected sample size.
- **(Section 4)** Then, we present a one-shot algorithm that bypasses index construction to generate a single subset sample in $O(N \log^2 N + \mu_\Psi)$ expected time.

Static	Method	Preprocessing Time	Sampling Time	Space Usage
Index	Baseline	$O(N + \lvert\text{Join}(Q)\rvert)$	$O(1 + \mu_\Psi)$	$O(\lvert\text{Join}(Q)\rvert)$
	Our Algorithm	$O(N \log N \log\log N)$	$O(1 + \mu_\Psi \log N)$	$O(N \log N)$
One-shot	Baseline	—	$O(N + \lvert\text{Join}(Q)\rvert)$	$O(\lvert\text{Join}(Q)\rvert)$
	Our Algorithm	—	$O(N \log^2 N + \mu_\Psi)$	$O(N \log^2 N + \mu_\Psi)$
Dynamic	**Method**	**Update Time**	**Sampling Time**	**Space Usage**
Index	Our Algorithm	$O(\log^3 N \log\log N)$	$O(\mu_\Psi \log N)$	$O(N \log N)$
One-shot	Our Algorithm	—	$O(N \log^3 N \log\log N + \mu_\Psi \log N)$	$O(N \log N)$

Table 1. Complexity of subset sampling on acyclic joins (extendable to cyclic joins with N increased to N^w) for product function. N is the input size, $|\text{Join}(Q)|$ is the join size, and μ_Ψ is the expected output size.

- **(Section 5)** Finally, we extend our framework to the dynamic setting with insertions, by applying and adapting the dynamic direct access index for the acyclic join in [23].

Remark 1. In the main text, we assume the function $\mathcal{F}$ to be the product of the input weights. In Appendix C, we show how other functions can be supported by simply adapting our algorithms.

1.4 Prior Results as Preliminaries

Subset Sampling. The classic subset sampling problem with a given set of elements has been well studied [9, 11, 31, 51]. A naive approach is to iterate through all n elements and flip a biased coin for each, taking $O(n)$ time. So, the goal is to design algorithms with query time proportional to the expected output size μ_Ψ, which can be significantly smaller than the total number of elements. In the dynamic setting, where elements and probabilities can change, any algorithm requires $\Omega(1+\mu_\Psi)$ query time and $\Omega(1)$ update time [51]. In the static setting, [11] provided the first index with an optimal query time of $O(1 + \mu_\Psi)$. However, their index requires $O(\log^2 n)$ update time, which is suboptimal in the dynamic setting. Recently, [9, 31, 51] optimally solved this dynamic problem with indexes that take expected query time $O(1 + \mu_\Psi)$, $O(n)$ space and $O(1)$ update time. [28, 32] explored weighted subset sampling where the probabilities are normalized by the total weight.

Worst-Case Optimal Joins. The AGM bound states that for any join instance Q of size N, the maximum number of join results is $\Theta(N^{\rho^*})$ [5], where ρ^* is the fractional edge covering number of the schema graph G_Q. The worst-case optimal join algorithms can compute any join instance Q of size N within $O(N^{\rho^*})$ time [43].

Direct Access for Joins. The *direct access* problem for joins was first studied by Carmeli et al. [12], which assumes a fixed ordering on the join results in $\text{Join}(Q)$ and asks for an index to return the join result in some specific position. For any acyclic join Q, an index can be built in $O(N)$ time such that each direct access query can be answered in $O(\log N)$ time [12, 52].

2 Classic Subset Sampling Revisited

In this section, we revisit the classic subset sampling problem, which serves as the foundation for our relational algorithms. We rely on the DirectAccess oracle for the input set S (under some fixed ordering[2]). Given an integer $i \in [|S|]$, this oracle returns the i-th element of S in $O(1)$ time.

2.1 Perfect-Uniform Subset Sampling

A subset sampling problem instance $\Psi = \langle S, \mathrm{p}\rangle$ is *perfect-uniform* if $\mathrm{p}(e) = p$ for any $e \in S$, which is also simplified as $\Psi = \langle S, p\rangle$. We have the following folklore result for the perfect-uniform scenario:

[2]The ordering can be arbitrary but must remain consistent across multiple invocations of this oracle.

Lemma 2.1 (Perfect-Uniform Subset Sampling)**.** *For a perfect-uniform subset sampling instance $\Psi = \langle S, p\rangle$, there is an index $\mathcal{D}$ that can be built within $O(|S|)$ space and time, using which each subset sampling query can be answered in $O(1 + \mu_\Psi)$ expected time.*

The index $\mathcal{D}$ consists of an array storing the elements of S, which allows us to retrieve the i-th element in $O(1)$ time. Additionally, we precompute the probability $q = 1 - (1 - p)^{|S|}$ that the resulting sample is non-empty. The construction takes $O(|S|)$ time and space. Given this index, there are two algorithms to achieve the query time complexity. Algorithm 1 represents the standard approach using geometric jumps to skip over rejected elements. Algorithm 2, however, utilizes the precomputed probability q to introduce a preliminary step: it first tosses a coin to decide whether to sample at least one element from S. While both algorithms are optimal for a single instance, Algorithm 2 offers a special benefit when handling multiple perfect-uniform instances. Instead of invoking the sampling procedure for every instance, this structure allows us to first sample the subset of instances that yield non-empty results, and then perform subset sampling only within those. This strategy significantly improves the total time complexity in the batched setting, as will be detailed in Section 2.3.

Algorithm 1: uss-vanilla(S, p)

```
X ← ∅; i ← 0;
while i < |S| do
    i ← i + 1 + Geometric(p);
    Add DirectAccess (S, i) to X;
return X;
```

Algorithm 2: uss-advanced(S, p)

```
X ← ∅; q ← 1 − (1 − p)^|S|;
if Uniform(0, 1) ≤ q then
    i ← 1 + TruncatedGeometric(p, |S|);
    Add DirectAccess (S, i) to X;
    while i < |S| do
        i ← i + 1 + Geometric(p);
        Add DirectAccess (S, i) to X;
return X;
```

2.2 Rejection-based Subset Sampling

The perfect-uniform condition is too restrictive for general applications. We can relax this requirement using a rejection-based strategy without affecting the asymptotic running time. Given a general instance $\Psi = \langle S, \mathrm{p}\rangle$, we define an upper bound $p_{\max} = \max_{e\in S} \mathrm{p}(e)$. We construct the index described in Lemma 2.1 for the bounding perfect-uniform instance $\Psi^+ = \langle S, p_{\max}\rangle$. This preprocessing takes $O(|S|)$ time.

The query procedure first retrieves a subset sample $\mathbf{X}$ from Ψ^+ using the index (via Algorithm 1 or Algorithm 2). Then, for each element $e' \in \mathbf{X}$, we retain it in the final sample with probability $\mathrm{p}(e')/p_{\max}$. The overall runtime is proportional to the size of the intermediate sample $\mathbf{X}$. There are two conditions under which this runtime is efficiently bounded:

Lemma 2.2 (β-uniform Subset Sampling)**.** *Given a subset sampling instance $\Psi = \langle S, \mathrm{p}\rangle$, if $\max_{e\in S} \mathrm{p}(e) \le \beta \cdot \min_{e\in S} \mathrm{p}(e)$ for some parameter $\beta \ge 1$, there is an index $\mathcal{D}$ that can be built within $O(|S|)$ space and time, using which each subset sampling query can be answered in $O(1 + \beta \cdot \mu_\Psi)$ expected time.*

Lemma 2.3 (Light Subset Sampling)**.** *Given a subset sampling instance $\Psi = \langle S, \mathrm{p}\rangle$, if $\max_{e\in S} \mathrm{p}(e) \le 1/|S|$, there is an index $\mathcal{D}$ that can be built within $O(|S|)$ space and time, using which each subset sampling query can be answered in $O(1)$ expected time.*

2.3 Batched Rejection-based Subset Sampling

Consider a general subset sampling problem instance $\Psi = \langle S, \mathrm{p}\rangle$ that is partitioned into a set of m disjoint sub-instances $\{\Psi_i = \langle S_i, \mathrm{p}\rangle\}_{i\in[m]}$, where $S = \bigsqcup_{i\in[m]} S_i$. Each sub-instance has an upper bound p_i^+ on the maximum probability (i.e., $\max_{e\in S_i} \mathrm{p}(e) \le p_i^+$) and is either light or β-uniform.

Algorithm 3: ss-rejected-batch$(\{\Psi_i\}_{i\in[m]})$

Input: Implicit access to the meta-index for q and sub-indexes for $\{\Psi_i\}$.
Output: A subset sample $\mathbf{X}$.

$\mathbf{I} \leftarrow$ **ss-query**(Ψ_{meta}) ; `// Query the meta-index`
foreach $i \in \mathbf{I}$ **do**
  $\mathbf{X}_i \leftarrow \emptyset$;
  $j \leftarrow 1 + \text{TruncatedGeometric}(p_i^+, |S_i|)$; `// Simulate Algorithm 2 using sub-index`
  Add DirectAccess (S_i, j) to $\mathbf{X}_i$;
  while $j < |S_i|$ **do**
    $j \leftarrow j + 1 + \text{Geometric}(p_i^+)$;
    if $j \le |S_i|$ **then** Add DirectAccess (S_i, j) to $\mathbf{X}_i$;
  foreach $e \in \mathbf{X}_i$ **do** Remove e from $\mathbf{X}_i$ with probability $1 - \frac{\mathrm{p}_i(e)}{p_i^+}$;
return $\bigcup_{i\in\mathbf{I}} \mathbf{X}_i$;

A naive approach would be to construct the index from Section 2.2 for each sub-instance and invoke the query procedure for each one. However, this would take $\Omega(m)$ time per query, as we must spend at least $O(1)$ time to check every sub-instance. To improve efficiency, we construct a composite index that supports a two-stage sampling process:

- **Preprocessing phase:** First, for each sub-instance Ψ_i, we build the standard index for $\langle S_i, p_i^+\rangle$ as described in Lemma 2.1. Second, we construct a meta-index to identify which sub-instances are likely to contribute to the final sample. We define a probability function $\mathrm{q} : [m] \to [0, 1]$ where $\mathrm{q}(i) = 1 - (1 - p_i^+)^{|S_i|}$. Note that $\mathrm{q}(i)$ is the probability that sampling from S_i with uniform probability p_i^+ yields a non-empty set. We build an optimal subset sampling index (e.g., [9]) for the meta-instance $\Psi_{\text{meta}} = \langle [m], \mathrm{q}\rangle$. The total preprocessing time is linear in $|S|$.
- **Query phase:** Algorithm 3 proceeds in two stages. First, it queries the meta-index to obtain a random set of indices $\mathbf{I} \subseteq [m]$. Then, only for the selected indices $i \in \mathbf{I}$, it invokes the query procedure on the sub-instance Ψ_i (using Algorithm 2 logic) and performs rejection sampling.

Lemma 2.4 (Batched Subset Sampling Index)**.** *Given a partitioned instance as defined above, there is an index that can be built in $O(|S|)$ time and space. Using this index, Algorithm 3 returns a valid subset sample in $O\left(1 + \sum_{i=1}^m |S_i| \cdot p_i^+\right)$ expected time.*

Proof of Lemma 2.4. The preprocessing phase constructs two levels of indexes: standard subset sampling indexes for each sub-instance Ψ_i (taking $O(\sum |S_i|) = O(|S|)$ time) and a meta-index for $\Psi_{\text{meta}} = \langle [m], \mathrm{q}\rangle$ (taking $O(m) \le O(|S|)$ time). The space complexity is clearly $O(|S|)$.

We now analyze the expected query time of Algorithm 3. The running time consists of two parts:

(1) *Meta-sampling.* Using the meta-index, generating the set of indices $\mathbf{I}$ takes expected time $O(1 + \mu_{\Psi_{\text{meta}}})$, where $\mu_{\Psi_{\text{meta}}} = \sum_{i=1}^m \mathrm{q}(i)$.
(2) *Sub-sampling.* For each selected index $i \in \mathbf{I}$, we generate an intermediate sample $\mathbf{X}_i'$ using the geometric jump procedure (simulating **uss-advanced**). The cost for a specific i is proportional to the size of this intermediate sample. The total expected cost is: $\mathbf{E}[\text{Cost}_{\text{sub}}] = \sum_{i=1}^m \mathrm{q}(i) \cdot O\left(\mathbf{E}[|\mathbf{X}_i'| \mid \mathbf{X}_i' \neq \emptyset]\right)$. Note that $\mathbf{E}[|\mathbf{X}_i'| \mid \mathbf{X}_i' \neq \emptyset] = \frac{\mathbf{E}[|\mathbf{X}_i'|]}{\Pr[\mathbf{X}_i' \neq \emptyset]} = \frac{|S_i| p_i^+}{\mathrm{q}(i)}$. Substituting this back into the summation, we have:

$$\mathbf{E}[\text{Cost}_{\text{sub}}] = O\left(\sum_{i=1}^m \mathrm{q}(i) \cdot \frac{|S_i| p_i^+}{\mathrm{q}(i)}\right) = O\left(\sum_{i=1}^m |S_i| p_i^+\right).$$

Combining both parts, the total expected time is $O(1 + \sum_{i=1}^{m} \mathrm{q}(i) + \sum_{i=1}^{m} |S_i| p_i^+)$. Since $\mathrm{q}(i) \le |S_i| p_i^+$, the total complexity simplifies to $O(1 + \sum_{i=1}^{m} |S_i| \cdot p_i^+)$. □

3 Indexed Subset Sampling over Joins

We start with Problem 1.2 for the join instance Q and the associated weight functions $\{\mathrm{p}_i\}_{i\in[k]}$. We assume tuples in each relation $R_i \in Q$ are in some arbitrary fixed ordering, and relations in Q are also in some arbitrary fixed ordering. All missing proofs of this section are given in Appendix B.

3.1 First Attempt

So far, we first assume that a DirectAccess oracle is available for the input join instance Q (under some fixed ordering), such that it receives an integer $i \in [|\mathrm{Join}(Q)|]$, and returns the i-th element of $\mathrm{Join}(Q)$. Similar to before, the ordering can be arbitrary but must remain consistent across multiple invocations of this oracle. Let $L = \lceil 2\rho^* \log N \rceil$, where ρ^* is the fractional edge covering number of the schema graph G_Q. Note that $2^L \ge |\mathrm{Join}(Q)|$.

Step 1: Partition input relations. For each relation $R_i \in Q$, we partition tuples into $L+1$ sub-relations $R_i^{\langle 0\rangle}, R_i^{\langle 1\rangle}, \ldots, R_i^{\langle L\rangle}$ based on their probabilities, where $R_i^{\langle j\rangle} = \{\boldsymbol{u} \in R_i \mid 2^{-j-1} < \mathrm{p}_i(\boldsymbol{u}) \le 2^{-j}\}$ for $j \in [\![L-1]\!]$ and $R_i^{\langle L\rangle} = \{\boldsymbol{u} \in R_i | \mathrm{p}_i(\boldsymbol{u}) \le 2^{-L}\}$. Then, each combination $\boldsymbol{j} = (j_1, j_2, \ldots, j_k) \in [\![L]\!]^k$ defines a sub-join of Q, denoted as: $Q_{\boldsymbol{j}} = \{R_1^{\langle j_1\rangle}, R_2^{\langle j_2\rangle}, \ldots, R_k^{\langle j_k\rangle}\}$. The join results of $Q_{\boldsymbol{j}}$ are a disjoint partition of the join results of Q, i.e., $\mathrm{Join}(Q) = \bigsqcup_{\boldsymbol{j}\in[\![L]\!]^k} \mathrm{Join}(Q_{\boldsymbol{j}})$. To answer a query to the subset sampling instance $\Psi = \langle \mathrm{Join}(Q), \mathrm{p}\rangle$, we can return the disjoint union of the subsets sampled from all sub-instances $\Psi_{\boldsymbol{j}} = \langle \mathrm{Join}(Q_{\boldsymbol{j}}), \mathrm{p}\rangle$ for $\boldsymbol{j} \in [\![L]\!]^k$. There are $(L+1)^k$ sub-instances in total.

Step 2: Apply subset sampling to all sub-instances. We first point out the following observation on each sub-instance in such a partition:

Lemma 3.1 (Either Light or Near-uniform Sub-instance)**.** *Let $\boldsymbol{j} = (j_1, j_2, \ldots, j_k)$. If $\sum_{i\in[k]} j_i \ge L$, the instance $\Psi_{\boldsymbol{j}}$ is light, and otherwise, it is 2^k-uniform.*

Proof of Lemma 3.1. For the largest probability, we always have $\max_{\boldsymbol{u}\in\mathrm{Join}(Q_{\boldsymbol{j}})} \mathrm{p}(\boldsymbol{u}) \le 2^{-\sum_{i\in[k]} j_i}$. If $\sum_{i\in[k]} j_i \ge L$, we have $\max_{\boldsymbol{u}\in\mathrm{Join}(Q_{\boldsymbol{j}})} \mathrm{p}(\boldsymbol{u}) \le 2^{-\sum_{i\in[k]} j_i} \le 2^{-L} \le 1/|\mathrm{Join}(Q)|$, following our assumption on L. Hence, $\Psi_{\boldsymbol{j}}$ is light. Otherwise, we must have $L \notin \{j_1, j_2, \ldots, j_k\}$. In this case, for the smallest probability, we have $2^{-k-\sum_{i\in[k]} j_i} \le \min_{\boldsymbol{u}\in\mathrm{Join}(Q_{\boldsymbol{j}})} \mathrm{p}(\boldsymbol{u})$. Hence, $\Psi_{\boldsymbol{j}}$ is 2^k-uniform since $\max_{\boldsymbol{u}\in\mathrm{Join}(Q_{\boldsymbol{j}})} \mathrm{p}(\boldsymbol{u}) \le \min_{\boldsymbol{u}\in\mathrm{Join}(Q_{\boldsymbol{j}})} \mathrm{p}(\boldsymbol{u})$. □

To efficiently sample from these sub-instances, we adopt the composite index strategy from Section 2.3. We treat the sub-joins $\{Q_{\boldsymbol{j}}\}_{\boldsymbol{j}\in[\![L]\!]^k}$ as the disjoint sub-instances. We define the meta-probability $\mathrm{q}(\boldsymbol{j}) = 1 - (1 - 2^{-\sum_{i\in[k]} j_i})^{|\mathrm{Join}(Q_{\boldsymbol{j}})|}$ and construct the meta-index for the instance $\langle [\![L]\!]^k, \mathrm{q}\rangle$. This allows us to skip empty or non-selected sub-instances efficiently. To put it formally:

- **Preprocessing phase:** We partition input tuples by weights. For each sub-instance $Q_{\boldsymbol{j}}$, we build a DirectAccess oracle (the sub-index) and compute the join size (e.g., [52]). Additionally, we build the meta-index for q as described in Section 2.3.
- **Query phase:** We invoke Algorithm 3 using the constructed meta-index and sub-indexes. The algorithm first samples a set of indices $\mathbf{I}$ using the meta-index, and then retrieves tuples from the selected sub-instances $Q_{\boldsymbol{j}}$ via their DirectAccess oracles.

Theorem 3.2. *Given an acyclic join instance Q consisting of k relations, and a set of associated weight functions $\{\mathrm{p}_j\}_{R_j\in Q}$, there is an index $\mathcal{D}$ that can be built within $O(N \log^{k-1} N)$ space and time, using which each subset sampling query can be answered in $O(1 + \mu_\Psi \log N)$ expected time.*

Proof of Theorem 3.2. The preprocessing phase constructs the index defined in Lemma 2.4.

(1) *Partitioning and Sub-indexes.* We partition each relation $R_i \in Q$ into $L+1$ sub-relations, defining $(L+1)^k$ sub-instances Q_j. For each sub-instance, we construct a DirectAccess oracle (sub-index). According to [52], building a DirectAccess oracle for an acyclic join takes time linear in its input size. Since each tuple participates in $(L+1)^{k-1}$ sub-instances and $L = O(\log N)$, the total time and space are $O(N \log^{k-1} N)$.
(2) *Meta-index.* We build the meta-index for $\Psi' = \langle [\![L]\!]^k, \mathrm{q} \rangle$. As the universe size is $(L+1)^k = O(\log^k N)$, this index can be built in $O(\log^k N)$ time and space, which is subsumed by the cost of building the sub-indexes.

The query proceeds according to Algorithm 3.

(1) *Meta-sampling.* We query the meta-index to obtain indices $\mathbf{I}$. The expected time is $O(1 + \mu_{\Psi'})$, where $\mu_{\Psi'} = \sum_{\boldsymbol{j}} \mathrm{q}(\boldsymbol{j})$. As shown in the main text, $\mu_{\Psi'} \le \mu_{\Psi} + 1$.
(2) *Sub-sampling.* For each $\boldsymbol{j} \in \mathbf{I}$, we draw samples from $Q_{\boldsymbol{j}}$. Since each sub-instance is either light or 2^k-uniform, the expected cost per selected instance is proportional to the number of samples times the DirectAccess access cost $O(\log N)$. Following the analysis in Lemma 2.4, the total expected cost is $O((1 + \mu_{\Psi}) \log N)$.

Summing these gives the total expected query time $O(1 + \mu_{\Psi} \log N)$. □

Next, we improve upon this initial result by showing an optimized index with much smaller space and preprocessing time, while maintaining the same query time.

3.2 Optimized Index – Framework

The framework in Section 3.1 establishes feasibility but is not space-efficient because it treats every combination of weight buckets $\boldsymbol{j} \in [\![L]\!]^k$ as a distinct sub-instance. We now present an optimized index that reduces space usage to $O(N \log N)$.

Merging Sub-instances by Score. Recall from Lemma 3.1 that the classification of a sub-instance $Q_{\boldsymbol{j}}$ (as either light or near-uniform) depends solely on the sum of indices $\sum_{i=1}^k j_i$. This observation allows us to merge sub-instances. Instead of maintaining $(L+1)^k$ disjoint partitions, we group join results based on their total weight "score."

For any tuple $\boldsymbol{u}$ in a relation R_i, let its *score* be $\phi(\boldsymbol{u}) = \lfloor -\log \mathrm{p}_i(\boldsymbol{u}) \rfloor$. For a join result $\boldsymbol{u} \in \mathrm{Join}(Q)$, its *score* is the sum of component scores: $\bar{\phi}(\boldsymbol{u}) = \sum_{i=1}^k \phi(\boldsymbol{u}[\mathrm{schema}(R_i)])$. We partition the join results $\mathrm{Join}(Q)$ into disjoint buckets based on this score. Let $\mathcal{B}_\ell = \{\boldsymbol{u} \in \mathrm{Join}(Q) \mid \bar{\phi}(\boldsymbol{u}) = \ell\}$ denote the set of join results with score ℓ. We treat these buckets as the disjoint sub-instances required by the batched framework in Section 2.3. Specifically:

(1) **Buckets $\ell < L$:** Each bucket $\mathcal{B}_\ell$ contains join results with probabilities in the range $(2^{-\ell-1}, 2^{-\ell}]$. Thus, $\mathcal{B}_\ell$ forms a 2-uniform sub-instance with upper bound $p^+_\ell = 2^{-\ell}$.
(2) **Tail Bucket $\ell \ge L$:** We merge all join results with score $\ell \ge L$ into a single tail bucket $\mathcal{B}_{\ge L} = \bigsqcup_{\ell \ge L} \mathcal{B}_\ell$. Since any $\boldsymbol{u} \in \mathcal{B}_{\ge L}$ has $\mathrm{p}(\boldsymbol{u}) \le 2^{-L} \le 1/|\mathrm{Join}(Q)|$, this combined bucket is a *light* sub-instance (Lemma 2.3) with upper bound $p^+_{\ge L} = 2^{-L}$.

Algorithm Overview. We apply Algorithm 3 to these $L+1$ sub-instances.

- **Preprocessing phase:** We compute the size $|\mathcal{B}_\ell|$ for each $\ell < L$ to build the meta-index. For the tail bucket $\mathcal{B}_{\ge L}$, we do not maintain an exact count or index; we simply assign it a weight proxy of N^{ρ^*} (an upper bound on join size) for the meta-index, or handle it via a fallback mechanism since its selection probability is negligible.
- **Query phase:** We invoke Algorithm 3 with inputs $\{\langle \mathcal{B}_\ell, \mathrm{p} \rangle \mid \ell < L\}$ and the tail bucket.

- If the meta-sampling selects a bucket $\mathcal{B}_\ell$ with $\ell < L$, we use a specialized DirectAccess oracle (described below) to retrieve a given index in $\mathcal{B}_\ell$ in $O(\log N)$ time.
- If the tail bucket $\mathcal{B}_{\geq L}$ is selected (which happens with very low probability), we materialize the necessary results from $\mathcal{B}_{\geq L}$ on the fly to support the retrieval.

The challenge is to efficiently support DirectAccess for the buckets $\mathcal{B}_\ell$ ($\ell < L$) without materializing them. We next describe how this can be achieved.

Join Tree with Notations. As mentioned, any acyclic join Q has a join tree $\mathcal{T}$ such that (1) there is a one-to-one correspondence between relations in Q and nodes in $\mathcal{T}$, and (2) for each attribute x, the set of nodes containing x forms a connected subtree. With a slight abuse of notation, we also use R_i to denote the corresponding node in $\mathcal{T}$ for relation $R_i \in Q$. For simplicity, we assume the nodes in $\mathcal{T}$ are ordered by the in-order traversal. For node R_i, let C_i be the child nodes of R_i if R_i is not a leaf. For a node $R_j \in C_i$, let $R_{\mathsf{next}(j)}$ be the immediately next child node after R_j in C_i. For the largest child node R_{j^*}, we set $\mathsf{next}(j^*) = \mathsf{null}$. For node R_i, we use $\mathsf{parent}(i)$ to denote the (unique) parent node of R_i if R_i is not the root. Let $\mathsf{key}(i) = \mathsf{schema}(R_i) \cap \mathsf{schema}(R_{\mathsf{parent}(i)})$ be the (common) join attributes between relation R_i and its parent if R_i is not the root. For the root node r, $\mathsf{key}(r) = \emptyset$. For a node R_i, let $\mathcal{T}_i$ be the subtree rooted at node R_i. For an arbitrary child $R_j \in C_i$, we define $\mathcal{T}_i^j$ to be the subtree of $\mathcal{T}_i$ that excludes any subtrees rooted at the child nodes in C_i coming before R_j, i.e., $\mathcal{T}_i^j = \mathcal{T}_i \setminus \cup_{R_{j'} \in C_i : R_{j'} \prec R_j} \mathcal{T}_{j'}$. For completeness, we also define $\mathcal{T}_i^\emptyset = \mathcal{T}_i$.

Data Structure. We first remove all dangling tuples from the database instance. For each relation $R_i \in Q$, we store input tuples in a hash table, so that for any subset of attributes $U \subseteq \mathsf{schema}(R_i)$ and a tuple $t \in \mathbf{dom}(U)$, we can get the list of tuples $R_i \ltimes t$ in $O(1)$ time. We take an arbitrary join tree $\mathcal{T}$ for Q and store the following statistics to augment $\mathcal{T}$.

W-values. Consider any internal node $R_i \in Q$. For each tuple $\boldsymbol{u} \in R_i$, we store L counters $W_{i,\boldsymbol{u}}^j(\ell)$ with $\ell \in [\![L-1]\!]$ specifically for each child node $R_j \in C_i$, where $W_{i,\boldsymbol{u}}^j(\ell)$ stores the number of join results produced by relations in the subtree $\mathcal{T}_i^j$ that is also participated by $\boldsymbol{u}$, with score ℓ, i.e.,

$$W_{i,\boldsymbol{u}}^j(\ell) = \left| \left\{ \boldsymbol{t} \in (\ltimes_{R_h \in \mathcal{T}_i^j} R_h) \ltimes \boldsymbol{u} : \sum_{R_h \in \mathcal{T}_i^j} \phi(\boldsymbol{t}[\mathsf{schema}(R_h)]) = \ell \right\} \right|. \tag{3}$$

Furthermore, we define and store $W_{i,\boldsymbol{u}}^\emptyset(\ell)$, the number of join results produced by relations in the subtree $\mathcal{T}_i$ that is also participated by $\boldsymbol{u}$, with score ℓ, by replacing $\mathcal{T}_i^j$ in (3) with $\mathcal{T}_i^\emptyset$. For completeness, for the largest child node $R_{j^*} \in C_i$, we also define $W_{i,\boldsymbol{u}}^{\mathsf{next}(j^*)}(0) = 1$ and $W_{i,\boldsymbol{u}}^{\mathsf{next}(j^*)}(\ell) = 0$ for every $\ell \in [L]$. Moreover, for each score $\ell \in [\![L-1]\!]$, and each child node $R_j \in C_i$, we build a prefix-sum array on $\langle W_{i,\boldsymbol{u}}^j(\ell) : \boldsymbol{u} \in R_i \rangle$ under the pre-determined ordering of tuples in R_i. For each leaf node $R_i \in Q$, we only store the counter $W_{i,\boldsymbol{u}}^\emptyset(\ell)$ for each tuple $\boldsymbol{u} \in R_i$.

M-values. For a non-root node R_i, we compute the projection of R_i onto $\mathsf{key}(i)$, as $R_i[\mathsf{key}(i)]$. For each tuple $\boldsymbol{v} \in R_i[\mathsf{key}(i)]$, we also compute $M_{i,\boldsymbol{v}}(0), M_{i,\boldsymbol{v}}(1), \ldots, M_{i,\boldsymbol{v}}(L-1)$: for $\ell \in [\![L-1]\!]$, $M_{i,\boldsymbol{v}}(\ell)$ stores the sum of counters $W_{i,\boldsymbol{u}}^\emptyset(\ell)$ over all tuples $\boldsymbol{u} \in R_i \ltimes \boldsymbol{v}$, i.e.,

$$M_{i,\boldsymbol{v}}(\ell) = \sum_{\boldsymbol{u} \in R_i \ltimes \boldsymbol{v}} W_{i,\boldsymbol{u}}^\emptyset(\ell). \tag{4}$$

3.3 Optimized Index – Preprocessing

We will show how to compute the statistics to build the desired index. The complete pseudocode is provided in Algorithm 4. Below, we show the intuition in detail. Our preprocessing phase performs computation in a bottom-up way. We note that M-values are defined on top of W-values, hence for

each node $R_i \in Q$, M-values will be computed according to (4) straightforwardly once W-values are well computed. Below, we focus on computing W-values.

Consider a leaf node R_i. By definition of (3), we have $W^{\emptyset}_{i,\boldsymbol{u}}(\phi(\boldsymbol{u})) = 1$ and $W^{\emptyset}_{i,\boldsymbol{u}}(l) = 0$ for any $l \in [\![L-1]\!] \setminus \{\phi(\boldsymbol{u})\}$. Consider an internal node R_i. Suppose W-values and M-values are well defined for each child $R_j \in C_i$. We next compute W-values for each tuple $\boldsymbol{u} \in R_i$ and each child node $R_j \in C_i$. More specifically, we compute $W^{j}_{i,\boldsymbol{u}}$ in a decreasing ordering of nodes in C_i. Recall that for the largest child node $R_{j^*} \in C_i$, $W^{j^*}_{i,\boldsymbol{u}}(0) = 1$ and $W^{j^*}_{i,\boldsymbol{u}}(\ell) = 0$ for $\ell \in [L-1]$. We next compute $W^{j}_{i,\boldsymbol{u}}(\cdot)$, assuming $W^{\mathsf{next}(j)}_{i,\boldsymbol{u}}(\cdot)$ has been computed. We distinguish the following two cases on the score ℓ (recall that $\ell \in [\![L-1]\!]$):

- **Case 1:** $\ell < \phi(\boldsymbol{u})$. In this case, $W^{j}_{i,\boldsymbol{u}}(\ell) = 0$ since any join result participated by $\boldsymbol{u}$ has its score at least $\phi(\boldsymbol{u})$;
- **Case 2:** $\phi(\boldsymbol{u}) \le \ell < L$**.** First, the Cartesian product of join result in each subtree $\mathcal{T}_{j'}$ rooted at any child node $R_{j'} \in C_i$ coming no earlier than R_j, that can be joined with $\boldsymbol{u}$, is essentially the join result in $\mathcal{T}_i^{\,j}$ that can be joined with $\boldsymbol{u}$. We also need to take the scores into consideration—all possible combinations of $\langle \ell_{j'} \in [\![L-1]\!] : R_{j'} \in C_i, R_j \preceq R_{j'} \rangle$ with $\sum_{R_{j'} \in C_i} \ell_{j'} = \ell - \phi(\boldsymbol{u})$. Each combination will contribute $\prod_{R_{j'} \in C_i : R_j \preceq R_{j'}} M_{j',\boldsymbol{u}[\mathsf{key}(j')]}(\ell_{j'})$ to $W^{j}_{i,\boldsymbol{u}}(\ell)$. However, the number of such combinations is as large as $O(L^k)$, which translates to $O(\log^k N)$ time even for computing a single W-value. A critical observation is that the products of M-values over $R_{j'}$ with $R_j \prec R_{j'}$ is actually captured by $W^{\mathsf{next}(j)}_{i,\boldsymbol{u}}$, where $R_{\mathsf{next}(j)}$ is the immediately next node of R_j in C_i. Putting these observations together, we obtain:

$$
\begin{aligned}
W^{j}_{i,\boldsymbol{u}}(\ell) &= \sum_{\substack{\langle \ell_{j'} \in [\![L-1]\!] : R_{j'} \in C_i, R_j \preceq R_{j'} \rangle : \\ \ell - \phi(\boldsymbol{u}) = \sum_{R_{j'} \in C_i} \ell_{j'}}} \prod_{R_{j'} \in C_i : R_j \preceq R_{j'}} M_{j',\boldsymbol{u}[\mathsf{key}(j')]}(\ell_{j'}) \\
&= \sum_{\substack{(\ell_1,\ell_2) \in [\![\ell - \phi(\boldsymbol{u})]\!] : \\ \ell_1 + \ell_2 = \ell - \phi(\boldsymbol{u})}} M_{j,\boldsymbol{u}[\mathsf{key}(j)]}(\ell_1) \cdot \sum_{\substack{\langle \ell_{j'} \in [\![L-1]\!] : R_{j'} \in C_i, R_j \prec R_{j'} \rangle : \\ \ell_2 = \sum_{R_j \in C_i} \ell_j}} \prod_{R_{j'} \in C_i : R_j \prec R_{j'}} M_{j',\boldsymbol{u}[\mathsf{key}(j')]}(\ell_{j'}) \\
&= \sum_{\substack{(\ell_1,\ell_2) \in [\![\ell - \phi(\boldsymbol{u})]\!] : \\ \ell_1 + \ell_2 = \ell - \phi(\boldsymbol{u})}} M_{j,\boldsymbol{u}[\mathsf{key}(j)]}(\ell_1) \cdot W^{\mathsf{next}(j)}_{i,\boldsymbol{u}}(\ell_2). \qquad (5)
\end{aligned}
$$

Hence $W^{j}_{i,\boldsymbol{u}}(\ell)$ can computed by (5). At that point, the M-values for the child node R_j and the W-values for the subsequent node $R_{\mathsf{next}(j)}$ have already been computed. Note that $W^{\emptyset}_{i,\boldsymbol{u}}(\ell)$ can be computed similarly by (5):

$$
W^{\emptyset}_{i,\boldsymbol{u}}(\ell) = \sum_{\substack{(\ell_1,\ell_2) \in [\![\ell - \phi(\boldsymbol{u})]\!] : \\ \ell_1 + \ell_2 = \ell - \phi(\boldsymbol{u})}} M_{j^\circ,\boldsymbol{u}[\mathsf{key}(j^\circ)]}(\ell_1) \cdot W^{\mathsf{next}(j^\circ)}_{i,\boldsymbol{u}}(\ell_2),
$$

where R_{j° is the first node in C_i.

To speed up the process, let's fix a node $R_i \in Q$, a tuple $\boldsymbol{u} \in R_i$, and a child node $R_j \in C_i$ as an example. Now, we need to compute $W^{\emptyset}_{i,\boldsymbol{u}}(\cdot)$ for L different scores. There is a convolution structure between $W^{j}_{i,\boldsymbol{u}}(\cdot)$, $M_{j,\boldsymbol{u}[\mathsf{key}(j)]}(\cdot)$ and $W^{\mathsf{next}(j)}_{i,\boldsymbol{u}}(\cdot)$, so we apply the Fast Fourier Transform for computing these L scores together. As we will show in the analysis of the preprocessing step, this observation improves the preprocessing time from $O(NL^2)$ to $O(NL \log L)$ (Lemma 3.3).

Algorithm 4: Preprocess$(Q, \mathcal{T})$

Input: Join tree $\mathcal{T}$ for Q, Relations $\{R_i\}_{R_i \in Q}$.

foreach *node* $R_i \in \mathcal{T}$ *in bottom-up order* **do**
  if R_i *is a leaf node* **then**
    foreach *tuple* $\boldsymbol{u} \in R_i$ **do**
      Set $W^{\emptyset}_{i,\boldsymbol{u}}(\phi(\boldsymbol{u})) \leftarrow 1$ and $W^{\emptyset}_{i,\boldsymbol{u}}(\ell) \leftarrow 0$ for $\ell \neq \phi(\boldsymbol{u})$;
      $M_{i,\boldsymbol{u}[\mathsf{key}(i)]} \leftarrow M_{i,\boldsymbol{u}[\mathsf{key}(i)]} + W^{\emptyset}_{i,\boldsymbol{u}}$; `// element-wise vector addition`
  else `//` R_i `is an internal node`
    foreach *tuple* $\boldsymbol{u} \in R_i$ **do**
      Let R_{j^*} be the last child in C_i;
      Set $W^{\mathsf{next}(j^*)}_{i,\boldsymbol{u}}(0) \leftarrow 1$ and $W^{\mathsf{next}(j^*)}_{i,\boldsymbol{u}}(\ell) \leftarrow 0$ for $\ell > 0$;
      foreach *child* $R_j \in C_i$ *in decreasing order* **do**
        Let $R_{\mathsf{next}(j)}$ be the child immediately following R_j;
        foreach *score* $\ell \in [\![L-1]\!]$ **do**
          if $\ell < \phi(\boldsymbol{u})$ **then** $W^{j}_{i,\boldsymbol{u}}(\ell) \leftarrow 0$;
          else $W^{j}_{i,\boldsymbol{u}}(\ell) \leftarrow \sum_{k=0}^{\ell-\phi(\boldsymbol{u})} M_{j,\boldsymbol{u}[\mathsf{key}(j)]}(k) \cdot W^{\mathsf{next}(j)}_{i,\boldsymbol{u}}(\ell - \phi(\boldsymbol{u}) - k)$;
      $M_{i,\boldsymbol{u}[\mathsf{key}(i)]} \leftarrow M_{i,\boldsymbol{u}[\mathsf{key}(i)]} + W^{\emptyset}_{i,\boldsymbol{u}}$; `// element-wise vector addition`
    foreach *child* $R_j \in C_i$ **do**
      foreach $\boldsymbol{v} \in R_i[key(i)]$ *and* $\ell \in [\![L-1]\!]$ **do**
        Build prefix-sum array on $\langle W^{j}_{i,\boldsymbol{u}}(\ell) : \boldsymbol{u} \in R_i \ltimes \boldsymbol{v} \rangle$;

After all W-values are computed, for each internal node $R_i \in Q$ with each child node $R_j \in C_i$, each value $\boldsymbol{v} \in R_i[\mathsf{key}(i)]$, and each score $\ell \in [\![L-1]\!]$, we build a prefix-sum array on $\langle W^{j}_{i,\boldsymbol{u}}(\ell) : \boldsymbol{u} \in R_i \ltimes \boldsymbol{v} \rangle$ under the pre-determined ordering.

Lastly, we compute the bucket size $|\mathcal{B}_\ell|$ for each score $\ell \in [\![L-1]\!]$, which by definition can be rewritten as $|\mathcal{B}_\ell| = \sum_{\boldsymbol{u} \in R_r} W^{\emptyset}_{r,\boldsymbol{u}}(\ell)$ for the root node r.

We briefly describe how to preprocess the data structures (see Appendix B for complete details). We compute W/M values in a bottom-up way. M-values will be computed according to (4) straightforwardly once W-values are well computed. Consider a leaf node R_i, we have $W^{\emptyset}_{i,\boldsymbol{u}}(\phi(\boldsymbol{u})) = 1$ and $W^{\emptyset}_{i,\boldsymbol{u}}(j) = 0$ for any $j \in [\![L-1]\!] \setminus \{\phi(\boldsymbol{u})\}$. Consider an internal node R_i. Suppose W-values and M-values are well defined for each child $R_j \in C_i$. We compute $W^{j}_{i,\boldsymbol{u}}$ in a decreasing ordering of nodes in C_i and follow a recursive structure. If $\ell < \phi(\boldsymbol{u})$, $W^{j}_{i,\boldsymbol{u}}(\ell) = 0$, and otherwise,

$$W^{j}_{i,\boldsymbol{u}}(\ell) = \sum_{(\ell_1,\ell_2) \in [\![\ell - \phi(\boldsymbol{u})]\!] : \ell_1 + \ell_2 = \ell - \phi(\boldsymbol{u})} M_{j,\boldsymbol{u}[\mathsf{key}(j)]}(\ell_1) \cdot W^{\mathsf{next}(j)}_{i,\boldsymbol{u}}(\ell_2). \tag{6}$$

Note that $W^{\emptyset}_{i,\boldsymbol{u}}(\ell)$ can be computed similarly by resorting to the first child node. Importantly, the computation of (6) can be sped up by Fast Fourier Transform [19, 20], and the whole computation only takes $O(NL \log L)$ time. Lastly, the bucket size $|\mathcal{B}_\ell|$ for each score $\ell \in [\![L-1]\!]$ can be rewritten as $|\mathcal{B}_\ell| = \sum_{\boldsymbol{u} \in R_r} W^{\emptyset}_{r,\boldsymbol{u}}(\ell)$ for the root node r.

Lemma 3.3. *Algorithm 4 runs* $O(N \log N \log\log N)$ *time, and uses* $O(N \log N)$ *space.*

Algorithm 5: RecursiveAccess$(i, j, \boldsymbol{v}, \ell, \tau)$

Input: $i, j \in [k]$ such that $R_j \in C_i$, tuple $\boldsymbol{v}$, score $\ell \in [\![L-1]\!]$, integer $\tau \in \mathbb{Z}^+$.

Output: The τ-th tuple in the join result of $\mathcal{T}_i^j$, with score ℓ, that can be joined with $\boldsymbol{v}$.

$\boldsymbol{u} \leftarrow$ the smallest tuple in $R_i \ltimes \boldsymbol{v}$ such that $\sum_{\boldsymbol{u}' \in R_i \ltimes \boldsymbol{v} : \boldsymbol{u}' \preceq \boldsymbol{u}} W^j_{i,\boldsymbol{u}'}(\ell) \geq \tau$;

if R_i *is a leaf node* **then return** $\boldsymbol{u}$;

$\tau \leftarrow \tau - \sum_{\boldsymbol{u}' \in R_i \ltimes \boldsymbol{v} : \boldsymbol{u}' \prec \boldsymbol{u}} W^j_{i,\boldsymbol{u}'}(\ell)$;

$\Phi \leftarrow \{(\ell_1, \ell_2) \in [\![\ell]\!]^2 : \ell_1 + \ell_2 = \ell - \phi(\boldsymbol{u})\}$ in lexicographical ordering by (ℓ_1, ℓ_2) ;

$(\ell_1, \ell_2) \leftarrow$ the smallest pair in Φ that $\sum_{(\ell_1', \ell_2') \in \Phi : (\ell_1', \ell_2') \preceq (\ell_1, \ell_2)} M_{j, \boldsymbol{u}[\mathsf{key}(j)]}(\ell_1') \cdot W^{\mathsf{next}(j)}_{i,\boldsymbol{u}}(\ell_2') \geq \tau$;

$\tau \leftarrow \tau - \sum_{(\ell_1', \ell_2') \in \Phi : (\ell_1', \ell_2') \prec (\ell_1, \ell_2)} M_{j, \boldsymbol{u}[\mathsf{key}(j)]}(\ell_1') \cdot W^{\mathsf{next}(j)}_{i,\boldsymbol{u}}(\ell_2')$;

$\tau_1 \leftarrow \left\lceil \frac{\tau}{W^{\mathsf{next}(j)}_{i,\boldsymbol{u}}(\ell_2)} \right\rceil$, $\tau_2 \leftarrow ((\tau - 1) \bmod W^{\mathsf{next}(j)}_{i,\boldsymbol{u}}(\ell_2)) + 1$;

$\boldsymbol{u}_1 \leftarrow$ **RecursiveAccess**$(j, \emptyset, \boldsymbol{u}, \ell_1, \tau_1)$;

$\boldsymbol{u}_2 \leftarrow$ **RecursiveAccess**$(i, \mathrm{next}(j), \boldsymbol{u}, \ell_2, \tau_2)$;

return $\boldsymbol{u}_1 \bowtie \boldsymbol{u} \bowtie \boldsymbol{u}_2$;

Proof of Lemma 3.3. For any relation R_i, and for every tuple $\boldsymbol{u} \in R_i$, we store $O(|C_i| \cdot L) = O(L)$ different W values. So, in total, the space needed for storing all the W values is $O(NL)$. Moreover, for each $\boldsymbol{v} \in R_i[\mathsf{key}(i)]$, we store $O(L)$ different M values. So, in total, the space needed for storing the M values is $O(NL)$. Hence, the overall space used is $O(NL) = O(N \log N)$.

To compute each W value, we use Equation (6). Calculating each W value by (6), needs a summation of $O(L)$ different terms, leading to an $O(NL^2)$ overall running time for calculating all the W values naively. To speed up the process, let's fix a node $R_i \in Q$, a tuple $\boldsymbol{u} \in R_i$, and a child node $R_j \in C_i$ as an example. Now, we need to compute $W^j_{i,\boldsymbol{u}}(\cdot)$ for L different scores. There is a convolution structure between $W^j_{i,\boldsymbol{u}}(\cdot)$, $M_{j,\boldsymbol{u}[\mathsf{key}(j)]}(\cdot)$ and $W^{\mathsf{next}(j)}_{i,\boldsymbol{u}}(\cdot)$ in Equation (6), so we can apply the Fast Fourier Transform for computing these L scores together. This can improve the $O(L^2)$ computations to $O(L \log L)$. More formally, consider the multiplication of two degree L polynomials:

$$p(x) = M_{j,\boldsymbol{u}[\mathsf{key}(j)]}(L-1)x^{L-1} + M_{j,\boldsymbol{u}[\mathsf{key}(j)]}(L-2)x^{L-2} + \cdots + M_{j,\boldsymbol{u}[\mathsf{key}(j)]}(0)x^0,$$

and,

$$q(x) = W^{\mathsf{next}(j)}_{i,\boldsymbol{u}}(L-1)x^{L-1} + W^{\mathsf{next}(j)}_{i,\boldsymbol{u}}(L-2)x^{L-2} + \cdots + W^{\mathsf{next}(j)}_{i,\boldsymbol{u}}(0)x^0.$$

By Equation (6), notice that for all $\ell \in [\![L-1]\!]$, the value $W^j_{i,\boldsymbol{u}}(\ell)$ is equal to the coefficient of $x^{\ell - \phi(u)}$ in the polynomial $p(x) \cdot q(x)$. Using the Fast Fourier Transform, we can calculate all the coefficients of the degree $2L$ polynomial $p(x) \cdot q(x)$ in $O(L \log L)$ time. Hence, for a fixed node $R_i \in Q$, a tuple $\boldsymbol{u} \in R_i$, and a child node $R_j \in C_i$, we can find all the L different $W^j_{i,\boldsymbol{u}}(\cdot)$ values in $O(L \log L)$ time. Putting everything together, calculating all the W values takes $O(NL \log L) = O(N \log N \log \log N)$. Having W-values, then calculating M-values is straightforward using Equation (4) in $O(NL)$ total time. Therefore, the overall running time of the preprocessing step is $O(N \log N \log \log N)$. □

3.4 Optimized Index – Direct Access

Our goal is to retrieve the τ-th tuple from bucket $\mathcal{B}_\ell$ given score $\ell \in [\![L-1]\!]$ and rank $\tau \in [|\mathcal{B}_\ell|]$. To ensure the "$\tau$-th" tuple is well-defined, we rely on the canonical lexicographic ordering of join

results. This ordering is naturally induced by the fixed traversal order of relations in the join tree $\mathcal{T}$ and the total ordering of tuples within each relation R_i.

We present a general procedure **RecursiveAccess**$(i, j, \boldsymbol{v}, \ell, \tau)$, that returns the τ-th tuple in the join results of the relations in the subtree $\mathcal{T}_i^j$, with score ℓ, that can be joined with $\boldsymbol{v}$ on its support attributes. The original problem can be solved by invoking **RecursiveAccess**$(r, \emptyset, \emptyset, \ell, \tau)$.

Algorithm 5 begins by identifying the specific tuple $\boldsymbol{u} \in R_i$ that participates in the target join result. Line 1 performs a binary search on the prefix-sum stored over $W_{i,*}^j(\ell)$ to locate the first tuple $\boldsymbol{u}$ where the cumulative count of join results in the subtree $\mathcal{T}_j^i$ (with score ℓ) covers the rank threshold τ. Line 3 then subtracts the contribution of tuples preceding $\boldsymbol{u}$ from τ. This reduces the problem to retrieving the (updated) τ-th join result within the subtree $\mathcal{T}_j^i$ specifically involving $\boldsymbol{u}$. We then distinguish two cases. If R_i is a leaf node, we simply return $\boldsymbol{u}$. In the general case, we decompose the retrieval into $\mathcal{T}_j$ and $\mathcal{T}_i^{\text{next}(j)}$ based on the fact that $\mathcal{T}_i^j = \mathcal{T}_j \bigsqcup \mathcal{T}_i^{\text{next}(j)}$. The objective is to determine the local indexes to query within these sub-problems. Recall that $W_{i,\boldsymbol{u}}^j(\ell)$ aggregates disjoint contributions from score pairs (ℓ_1, ℓ_2) (Equation 6). The algorithm iterates through these pairs in lexicographical order (Line 4) to find the specific pair (ℓ_1, ℓ_2) that covers the current rank threshold τ (Line 5). After further adjusting τ to account for preceding pairs (Line 6), we effectively map the remaining rank to indexes τ_1 and τ_2 within the Cartesian product of valid results from $\mathcal{T}_j$ and $\mathcal{T}_i^{\text{next}(j)}$ (Line 7). Finally, we invoke the procedure recursively on $\mathcal{T}_j$ and $\mathcal{T}_i^{\text{next}(j)}$ and combine the returned results.

Correctness. To establish the correctness of Algorithm 5, we show that for DirectAccess (S_ℓ, j) in Algorithm 3, Algorithm 5 returns the τ-th tuple in $\mathcal{B}_\ell$, according to a fixed total ordering of the tuples in each bucket $\mathcal{B}_\ell$. Recall from Section 2 that the actual ordering of the tuples is irrelevant, as long as it is fixed. Since Algorithm 5 is deterministic, it suffices to prove that for any two distinct calls of DirectAccess $(S_{\hat{\ell}}, \hat{\tau})$ and DirectAccess $(S_{\bar{\ell}}, \bar{\tau})$ in Algorithm 3, Algorithm 5 returns two distinct tuples $\hat{\boldsymbol{w}}$ and $\bar{\boldsymbol{w}}$ from Join(Q). This is exactly proved by Lemma B.1 in Appendix B.

Complexity. We will analyze its cost step by step. Line 1 performs a binary search on the prefix-sum tree for N tuples, which takes $O(\log N)$ time. Line 3 can also be efficiently done through the prefix-sum tree in $O(\log N)$ time, since any prefix sum can be decomposed into $O(\log N)$ canonical nodes in this tree. Line 2 takes $O(1)$ time. Lines 5 - 6 take $O(L) = O(\log N)$ time, since there are at most L pairs in Φ to be explored. Line 7 takes $O(1)$ time. Hence, all lines before the recursion take $O(\log N)$ time. As there are at most k^2 different combinations of (i, j), and the recursion is only invoked once for each combination of (i, j), the total number of recursions is $O(k^2)$. As k is a constant, the total running time of Algorithm 5 is $O(\log N)$.

Now, we are ready to show the following theorem for our optimized index:

Theorem 3.4. *Given an acyclic join instance Q of k relations, and a set of associated weight functions $\{\mathrm{p}_j\}_{R_j \in Q}$, there exists an index $\mathcal{D}$ of size $O(N \log N)$ that can be built in $O(N \log N \log\log N)$ time, such that each subset sampling query can be answered in $O(1 + \mu_\Psi \log N)$ expected time.*

Proof of Theorem 3.4. The preprocessing cost follows from Lemma 3.3. We focus on the query answering time. In answering a subset sampling query, whenever some join result needs to be retrieved from $\mathcal{B}_\ell$, we distinguish the following two cases. If $\ell = L$, we evaluate the full join results in $\mathcal{B}_{\geq L}$ (if these join results have not been computed before) to support the direct access. Otherwise, we call the **RecursiveAccess** procedure accordingly. For the first case, the expected cost is $O(1)$ because there are at most $O(N^{\rho^*})$ join results in $\mathcal{B}_{\geq L}$ that takes $O(N^{\rho^*})$ time to materialize [5], and the probability that at least a join result is retrieved from $\mathcal{B}_{\geq L}$ is at most $1 - (1 - p_L^+)^{|\mathcal{B}_{\geq L}|} \leq$

$1-(1-\frac{1}{N^{2\rho^*}})^{N^{\rho^*}} \leq \frac{1}{N^{\rho^*}}$, since $|\mathcal{B}_{\geq L}| \leq N^{\rho^*}$, and $p_L^+ \leq \frac{1}{2^L} \leq \frac{1}{N^{2\rho^*}}$. Following Lemma 2.2 and the runtime of Algorithm 5, the expected query answering time is $O(1+\mu_\Psi \log N)$. □

4 One-shot Subset Sampling over Joins

We observe that the data structure from Theorem 3.4 provides a baseline solution for the one-shot problem. By constructing the index and issuing a single query, we can generate a subset sample in $O(N \log N \log\log N + \mu_\Psi \log N)$ expected time. In this section, we present a specialized algorithm for the one-shot setting that runs in $O(N \log^2 N + \mu_\Psi)$ expected time. This approach improves upon the baseline solution when the expected sample size is large (i.e., $\mu_\Psi \gg N$), as it eliminates the logarithmic factor associated with μ_Ψ.

High-level Idea. We first keep all statistics computed in the preprocessing phase as in Section 3.2. We conceptually partition the join results into buckets. Recall that the high-level idea of answering one subset sampling query is to invoke Algorithm 3 on instances $\{\langle \mathcal{B}_\ell, \mathrm{p}\rangle : \ell \in [\![L-1]\!]\}$ with $p_\ell^+ = 2^{-\ell}$, and $\langle \bigsqcup_{\ell \geq L} \mathcal{B}_\ell, p\rangle$ with $p_L^+ = 2^{-L}$. Whenever a join result needs to be retrieved from $\mathcal{B}_\ell$, if $\ell < L$, we use the DirectAccess oracle to support the retrieval, and otherwise, we compute all the results in $\bigsqcup_{\ell \geq L} \mathcal{B}_\ell$ to support the retrieval. In the first case, whenever the DirectAccess oracle on bucket $\mathcal{B}_\ell$ with $\ell < L$ is invoked, it pays $O(\log N)$ time for each retrieval. To overcome this barrier in the query time, we need to compute additional statistics. Careful inspection of Algorithm 5 reveals that the $O(\log N)$ cost comes from the binary search in Line 1 and the exhaustive search in Line 5. A natural idea is to precompute all these statistics. Consider an arbitrary relation $R_i \in Q$ with an arbitrary child node $R_j \in C_i$. For every score $\ell \in [\![L-1]\!]$, and every tuple $\boldsymbol{v} \in R_i[\mathsf{key}(i)]$, we compute an array $X_{i,j,\boldsymbol{v},\ell}$ of size $|R_i \ltimes \boldsymbol{v}|$, where for every tuple $\boldsymbol{u} \in R_i \ltimes \boldsymbol{v}$,

$$X_{i,j,\boldsymbol{v},\ell}(\boldsymbol{u}) = \sum_{\boldsymbol{u}' \in R_i \ltimes \boldsymbol{v},\, \boldsymbol{u}' \preceq \boldsymbol{u}} W^j_{i,\boldsymbol{u}'}(\ell).$$

Next, for every score $\ell \in [\![L-1]\!]$ and every tuple $\boldsymbol{u} \in R_i$, we compute an array $Y_{i,j,\boldsymbol{u},\ell}$ of size at most L, where for every pair $(\ell_1,\ell_2) \in [\![\ell]\!]^2$ with $\ell_1+\ell_2 = \ell - \phi(\boldsymbol{u})$, in lexicographically sorted order,

$$Y_{i,j,\boldsymbol{u},\ell}(\ell_1,\ell_2) = \sum_{(\ell_1',\ell_2') \in [\![\ell]\!] : (\ell_1',\ell_2') \preceq (\ell_1,\ell_2), \ell_1'+\ell_2' = \ell-\phi(\boldsymbol{u})} M_{j,\boldsymbol{u}[\mathsf{key}(j)]}(\ell_1') \cdot W^{\mathrm{next}(j)}_{i,\boldsymbol{u}}(\ell_2').$$

We note that by constructing X and Y arrays, we do not define a new index for subset sampling over joins. Instead, these statistics are computed when answering one-shot subset sampling query.

Each invocation of Algorithm 5 is identified by a quintuple $(i, j, \boldsymbol{v}, \ell, \tau)$ as its input. The high-level idea is to maintain the list of quintuples $(i, j, \boldsymbol{v}, \ell, \tau)$ on which we will invoke Algorithm 5, and run multiple invocations of Algorithm 5 together by merging certain operations. We group these quintuples by $(i, j, \boldsymbol{v}, \ell)$, where quintuples inside each group only have different τ-values sorted by τ in increasing order. We maintain the set of quintuples $\mathcal{P}_{i,j}$ we should satisfy in each call of Algorithm 5. For a node R_i and $R_j \in C_i \cup \{\emptyset\}$, $\mathcal{P}_{i,j}$ is defined as the parameter set of all invocations of Algorithm 5 on (i, j). Initially, for each request DirectAccess (S_ℓ, τ) issued in Algorithm 3, we add a quintuple $(r, \emptyset, \emptyset, \ell, \tau)$ to $\mathcal{P}_{r,\emptyset}$, where R_r is the root node. Below, we describe a general procedure **BatchRecursiveAccess**$(i, j, \mathcal{P}_{i,j})$. The new algorithm executes the same steps as in Algorithm 5, but it computes all smallest tuples $\boldsymbol{u}$ and pairs (ℓ_1, ℓ_2) for all quintuples simultaneously.

BatchRecursiveAccess$(i, j, \mathcal{P}_{i,j})$**:** This procedure takes as input a node $R_i \in Q$ with a child node $R_j \in C_i$, and the set of quintuples $\mathcal{P}_{i,j}$. The output will be exactly the collection of tuples returned by Algorithm 5 for each invocation parameterized by quintuple $(i, j, \boldsymbol{v}, \ell, \tau) \in \mathcal{P}_{i,j}$. The pseudocode is shown in Algorithm 6.

We first sort $\mathcal{P}_{i,j}$ based on τ using the Radix sort algorithm [20]. Since τ is polynomially bounded in N, using radix sort saves a factor of $\log N$ in the sorting time. Then we group $\mathcal{P}_{i,j}$ by the

Algorithm 6: BatchRecursiveAccess$(i, j, \mathcal{P}_{i,j})$

$\mathcal{U} \leftarrow \emptyset$;
Sort $\mathcal{P}_{i,j}$ by τ using Radix sort;
Group $\mathcal{P}_{i,j}$ further by $(i, j, \boldsymbol{v}, \ell)$ creating groups $\mathcal{P}_{i,j,\boldsymbol{v},\ell}$ using the sorted order by $\mathcal{P}_{i,j}$;
for $\ell \in [\![L-1]\!]$ **do**
  for *each* $(i, j, \boldsymbol{v}, \ell, \tau) \in \mathcal{P}_{i,j,\boldsymbol{v},\ell}$ *in sorted order* **do**
    $\boldsymbol{u} \leftarrow$ first tuple in $X_{i,j,\boldsymbol{v},\ell}$;
    while $X_{i,j,\boldsymbol{v},\ell}(\boldsymbol{u}) < \tau$ **do** $\boldsymbol{u} \leftarrow$ next tuple in $X_{i,j,\boldsymbol{v},\ell}$;
    if R_i *is a leaf node* **then** Add $\boldsymbol{u}$ as $\vec{\boldsymbol{u}}(i, j, \boldsymbol{v}, \ell, \tau)$ to $\mathcal{U}$;
    $\mathcal{P}_{i,j} \leftarrow (\mathcal{P}_{i,j} - \{(i, j, \boldsymbol{v}, \ell, \tau)\}) \cup \{(i, j, \boldsymbol{u}, \ell, \tau - X_{i,j,\boldsymbol{v},\ell}(\mathsf{prev}(\boldsymbol{u})))\}$;
if R_i *is a leaf node* **then return** $\mathcal{U}$;
Sort $\mathcal{P}_{i,j}$ by τ using Radix sort;
Group $\mathcal{P}_{i,j}$ further by $(i, j, \boldsymbol{v}, \ell)$ creating groups $\mathcal{P}_{i,j,\boldsymbol{v},\ell}$ using the sorted order by $\mathcal{P}_{i,j}$.;
$\mathcal{P}_{j,\emptyset} \leftarrow \emptyset, \mathcal{P}_{i,\mathrm{next}(j)} \leftarrow \emptyset$;
for $\ell \in [\![L-1]\!]$ **do**
  for *each* $(i, j, \boldsymbol{u}, \ell, \tau) \in \mathcal{P}_{i,j,\boldsymbol{u},\ell}$ *in sorted order* **do**
    $(\ell_1, \ell_2) \leftarrow$ first pair in $Y_{i,j,\boldsymbol{u},\ell}$;
    while $Y_{i,j,\boldsymbol{v},\ell}((\ell_1, \ell_2)) < \tau$ **do** $(\ell_1, \ell_2) \leftarrow$ the next pair in $Y_{i,j,\boldsymbol{u},\ell}$;
    Compute τ_1, τ_2 based on ℓ_1, ℓ_2 according to Line 7 of Algorithm 5;
    $\mathcal{P}_{j,\emptyset} \leftarrow \mathcal{P}_{j,\emptyset} \cup \{(j, \emptyset, \boldsymbol{u}, \ell_1, \tau_1)\}$;
    $\mathcal{P}_{i,\mathrm{next}(j)} \leftarrow \mathcal{P}_{i,\mathrm{next}(j)} \cup \{(i, \mathrm{next}(j), \boldsymbol{u}, \ell_2, \tau_2)\}$;
$\mathcal{U} \leftarrow \mathcal{U} \cup$ **BatchRecursiveAccess**$(j, \emptyset, \mathcal{P}_{j,\emptyset})$;
$\mathcal{U} \leftarrow \mathcal{U} \cup$ **BatchRecursiveAccess**$(i, \mathrm{next}(j), \mathcal{P}_{i,\mathrm{next}(j)})$;
for $(i, j, \boldsymbol{u}, \ell, \tau) \in \mathcal{P}_{i,j}$ **do**
  $\vec{\boldsymbol{u}}(i, j, \boldsymbol{u}, \ell, \tau) \leftarrow \vec{\boldsymbol{u}}(j, \emptyset, \boldsymbol{u}, \ell_1, \tau_1) \bowtie \boldsymbol{u} \bowtie \vec{\boldsymbol{u}}(i, \mathrm{next}(j), \boldsymbol{u}, \ell_2, \tau_2)$;
  `// ` $\ell_1, \ell_2, \tau_1, \tau_2$ `are computed in Lines 17 and 18 above`
  Add $\vec{\boldsymbol{u}}(i, j, \boldsymbol{u}, \ell, \tau)$ to $\mathcal{U}$;
return $\mathcal{U}$;

different $(i, j, \boldsymbol{v}, \ell)$ creating the groups $\mathcal{P}_{i,j,\boldsymbol{v},\ell}$. We notice that all quintuples in each $\mathcal{P}_{i,j,\boldsymbol{v},\ell}$ are sorted by τ using the sorted ordering of $\mathcal{P}_{i,j}$. The goal is to compute the smallest tuple $\boldsymbol{u}$ for each quintuple, as in Line 1 of Algorithm 5. For every score $\ell \in [\![L-1]\!]$, we consider every quintuple $(i, j, \boldsymbol{v}, \ell, \tau) \in \mathcal{P}_{i,j,\boldsymbol{v},\ell}$ following the sorted order. We traverse table $X_{i,j,\boldsymbol{v},\ell}$ until we get the first tuple $\boldsymbol{u}$ such that $X_{i,j,\boldsymbol{v},\ell}(\boldsymbol{u}) \geq \tau$. We also update the quintuple $(i, j, \boldsymbol{v}, \ell, \tau)$ to $(i, j, \boldsymbol{v}, \ell, \tau - X_{i,j,\boldsymbol{v},\ell}(\mathsf{prev}(\boldsymbol{u})))$ as in Line 3 of Algorithm 5. If R_i is a leaf node we return all smallest tuples $\boldsymbol{u}$ we computed from every quintuple in $\mathcal{P}_{i,j}$. For later reference, we denote the returned tuple $\boldsymbol{u}$ for the quintuple $(i, j, \boldsymbol{v}, \ell, \tau)$ as $\vec{\boldsymbol{u}}(i, j, \boldsymbol{v}, \ell, \tau)$.

Then we proceed in a similar manner to compute the pairs (ℓ_1, ℓ_2). We sort (the updated) $\mathcal{P}_{i,j}$ by τ using the Radix sort algorithm. Then we group $\mathcal{P}_{i,j}$ by the different $(i, j, \boldsymbol{v}, \ell)$ creating the groups $\mathcal{P}_{i,j,\boldsymbol{v},\ell}$. The goal is to compute the smallest pair (ℓ_1, ℓ_2) for each quintuple, as in Line 5 of Algorithm 5. For every score $\ell \in [\![L-1]\!]$ we consider every quintuple $(i, j, \boldsymbol{v}, \ell, \tau) \in \mathcal{P}_{i,j,\boldsymbol{v},\ell}$ following the sorted order. We traverse table $Y_{i,j,\boldsymbol{v},\ell}$ until we get the first pair (ℓ_1, ℓ_2) such that $Y_{i,j,\boldsymbol{v},\ell}(\ell_1, \ell_2) \geq \tau$. We compute τ_1, τ_2 as in Line 7 of Algorithm 5. Furthermore, we add the quintuple $(j, \emptyset, \boldsymbol{u}, \ell_1, \tau_1)$ to

the (initially empty) set $\mathcal{P}_{j,\emptyset}$ and add the quintuple $(i, \text{next}(j), \boldsymbol{u}, \ell_2, \tau_2)$ in the (initially empty) set $\mathcal{P}_{i,\text{next}(j)}$. Notice that these sets are used as input to the next calls **BatchRecursiveAccess**$(j, \emptyset, \mathcal{P}_{j,\emptyset})$ and **BatchRecursiveAccess**$(i, \text{next}(j), \mathcal{P}_{i,\text{next}(j)})$ as in Lines 8 and 9 of Algorithm 5.

Now, we will generate tuples to be returned for quintuples in $\mathcal{P}_{i,j}$. Suppose the sets of tuples returned by the two recursive invocations are available. For every quintuple $(i, j, \boldsymbol{v}, \ell, \tau) \in \mathcal{P}_{i,j}$, let $\vec{\boldsymbol{u}}(j, \emptyset, \boldsymbol{u}, \ell_1, \tau_1)$ be the tuple returned for the quintuple $(j, \emptyset, \boldsymbol{u}, \ell_1, \tau_1) \in \mathcal{P}_{j,\emptyset}$, and $\vec{\boldsymbol{u}}(i, \text{next}(j), \boldsymbol{u}, \ell_2, \tau_2)$ be the tuple returned for the quintuple $(i, \text{next}(j), \boldsymbol{u}, \ell_2, \tau_2) \in \mathcal{P}_{i,\text{next}(j)}$, We simply join tuple $\vec{\boldsymbol{u}}(j, \emptyset, \boldsymbol{u}, \ell_1, \tau_1)$, and tuple $\vec{\boldsymbol{u}}(i, \text{next}(j), \boldsymbol{u}, \ell_2, \tau_2)$ together with tuple $\boldsymbol{u}$ to form the tuple $\vec{\boldsymbol{u}}(i, j, \boldsymbol{u}, \ell, \tau)$. Finally, we will return all tuples for each quintuple $(i, j, \boldsymbol{v}, \ell, \tau) \in \mathcal{P}_{i,j}$.

Theorem 4.1. *For an acyclic join instance Q of size N, and a set of associated weight functions $\{\mathrm{p}_j\}_{R_j \in Q}$, there exists an algorithm that returns one subset sample of the instance $\Psi = \langle \text{Join}(Q), \mathrm{p} \rangle$ in $O\left(\min\left\{N \log N \log\log N + \mu_\Psi \log N,\ N \log^2 N + \mu_\Psi\right\}\right)$ expected time.*

Proof of Theorem 4.1. The correctness follows from that of Algorithm 5, since we exactly simulate the execution of all invocations of Algorithm 5. The data statistics inherited from Theorem 3.4 can be computed in $O(N \log N \log\log N)$ time. All X-arrays can be computed in $O(N \log N)$ time since for every $\boldsymbol{u} \in R_i$ there is a unique $\boldsymbol{v} \in R_i[\text{key}(i)]$ such that $\boldsymbol{u} \in R_i \ltimes \boldsymbol{v}$. All Y-arrays can be computed in $O(N \log^2 N)$ time, since there are $O(N \log N)$ arrays and each array has size $O(\log N)$. Using the precomputed M and W statistics, every value in every Y array is computed in $O(1)$ time.

We fix a table R_i and one of its children R_j in $\mathcal{T}$. Since $|\mathcal{P}_{i,j}| = O(1 + \mu_\Psi)$, Radix sort runs in $O(N + \mu_\Psi)$ expected time. To compute the smallest tuple $\boldsymbol{u}$ for each quintuple in $\mathcal{P}_{i,j,\boldsymbol{v},\ell}$, we iterate over all quintuples in $\mathcal{P}_{i,j,\boldsymbol{v},\ell}$ and traverse $X_{i,j,\boldsymbol{v},\ell}$ once, in the worst case. Since for every $\boldsymbol{u} \in R_i$ there is a unique $\boldsymbol{v}$ with $\boldsymbol{u} \in R_i \ltimes \boldsymbol{v}$ and every quintuple in $\mathcal{P}_{i,j}$ belongs to a unique group $\mathcal{P}_{i,j,\boldsymbol{v},\ell}$, we have $\sum_\ell (|X_{i,j,\boldsymbol{v},\ell}| + |\mathcal{P}_{i,j,\boldsymbol{v},\ell}|) = O(N \log N + \mu_\Psi)$. Similarly, computing all pairs (ℓ_1, ℓ_2) requires a single pass over all quintuples in each $\mathcal{P}_{i,j,\boldsymbol{u},\ell}$ and all Y tables. Since there are $O(N \log N)$ tables Y of size $O(\log N)$, the total cost is $O(N \log^2 N + \mu_\Psi)$ expected time. The number of nodes in $\mathcal{T}$ is $k = O(1)$ and every node has $O(1)$ children, so, overall, the running time of the one-shot algorithm is $O(N \log^2 N + \mu_\Psi)$ expected time. □

5 Dynamic Subset Sampling over Joins

In this section, we move to the dynamic setting, where tuples are inserted one by one. We first adapt the techniques of [23] to directly extend our non-optimal (static) index from Theorem 3.2 to a dynamic variant that maintains an approximate DirectAccess index. We then introduce new techniques and ideas to extend our optimized static index from Theorem 3.4 to the dynamic setting. Throughout, we assume that, for each relation, tuples are ordered by their insertion timestamps.

5.1 Basic Index

The high-level idea is to approximately maintain the static index presented in Section 3. However, as pointed out by [23], even maintaining the join size for line-3 join $Q = \{R_1(X, Y), R_2(Y, Z), R_3(Z, W)\}$ requires at least $\Omega(\sqrt{N})$ time for each insertion, which is too costly in the streaming setting. Hence, we follow the approach proposed by [23] by maintaining an approximate DirectAccess index more efficiently while not sacrificing the overall sampling efficiency asymptotically. We first define the delta query $\Delta\,\text{Join}(Q, \boldsymbol{u})$ as the set of new join results generated when tuple $\boldsymbol{u}$ is inserted into a relation of Q. Formally, if $\boldsymbol{u}$ is inserted into R_i, $\Delta\,\text{Join}(Q, \boldsymbol{u}) = \text{Join}(Q \setminus \{R_i\} \cup \{R_i \cup \{\boldsymbol{u}\}\}) \setminus \text{Join}(Q)$.

Theorem 5.1 ([23])**.** *Given an initially empty acyclic join Q, we can maintain an index $\mathcal{L}$ on Q that supports the following operations:*

- *When a tuple t is added to Q, $\mathcal{L}$ can be updated in amortized $O(\log N)$ time.*

- *The index defines an array* $J \supseteq \mathrm{Join}(Q)$ *where the tuples in* $\mathrm{Join}(Q)$ *are the real tuples and the others are dummy. The index can return* $|J|$ *in* $O(1)$ *time. For any given* $j \in [|J|]$*, it returns* $J[j]$ *in* $O(\log N)$ *time. Furthermore,* J *is guaranteed to be* λ*-dense*[3] *for some constant probability* $0 < \gamma \leq 1$.
- *The above also holds when* $\mathrm{Join}(Q)$ *is replaced by the delta query* $\Delta\,\mathrm{Join}(Q, \boldsymbol{u})$ *for any* $\boldsymbol{u} \notin Q$.

where N *is the total number of insertions, but the index does not need the knowledge of* N *in advance.*

Using Theorem 5.1, we derive a simple dynamic algorithm by adapting the framework from Section 3.1. As in Section 3.1, we define $(L+1)^k$ sub-instances $Q_{\boldsymbol{j}}$ for $\boldsymbol{j} \in [\![L]\!]^k$. In the dynamic setting, we maintain a dynamic index $\mathcal{L}_{\boldsymbol{j}}$ (as described in Theorem 5.1) for *each* sub-instance $Q_{\boldsymbol{j}}$. When a tuple t is inserted into relation R_i, we first determine its bucket index a such that $t \in R_i^{\langle a \rangle}$. This tuple t participates in $(L+1)^{k-1}$ sub-instances. We update the corresponding $(L+1)^{k-1}$ indexes. Since each update takes $O(\log N)$ time amortized, the total amortized update time is $O(L^{k-1} \log N)$. To answer a subset sampling query, we use the batched rejection sampling strategy. For each sub-instance $Q_{\boldsymbol{j}}$, the index $\mathcal{L}_{\boldsymbol{j}}$ provides the size of the implicit array $|J_{\boldsymbol{j}}|$. We use $|J_{\boldsymbol{j}}|$ as an upper bound on the size of the sub-join to define the meta-sampling probability $\mathrm{q}(\boldsymbol{j})$. Suppose a sub-instance $Q_{\boldsymbol{j}}$ is selected and the algorithm needs to retrieve the tuple $u = J_{\boldsymbol{j}}[z]$. If u is a dummy tuple (which happens with probability at most $1-\gamma$) we reject it. If u is real, we proceed with the standard rejection check based on its weight. Since γ is a constant, the expected query time remains asymptotically the same as the static case.

Corollary 5.2. *Given an acyclic join instance* Q *consisting of* k *relations, and a set of associated weight functions* $\{\mathrm{p}_j\}_{R_j \in Q}$*, we can maintain a dynamic index under tuple insertions using* $O(N \log^{k-1} N)$ *space, with amortized update time* $O(\log^k N)$*, while supporting answering the subset sampling query in* $O(1 + \mu_\Psi \log N)$ *expected time, where* N *is the total number of insertions.*

5.2 Optimized Dynamic Index for Subset Sampling over Joins

To improve both space and update time, we adapt the optimized index from Section 3.2 to the dynamic setting. This adaptation is nontrivial, since the index in Section 3.2 is inherently static and therefore requires new ideas. Rather than maintaining disjoint indexes for every bucket combination, we maintain the aggregated statistics (the W and M values) directly. Because maintaining exact counts is too costly, we instead store *approximate* statistics, rounded up to the next power of 2.

For each relation $R_i \in Q$, each child node $R_j \in C_i$, and each tuple $\boldsymbol{u} \in R_i$, we maintain an upper bound $\tilde{W}^j_{i,\boldsymbol{u}}(\ell)$ on $W^j_{i,\boldsymbol{u}}(\ell)$. Similarly, for each non-root node R_i, tuple $\boldsymbol{v} \in R_i[\mathrm{key}(i)]$, and score $\ell \in [\![L-1]\!]$, we maintain $\hat{M}_{i,\boldsymbol{v}}(\ell)$ as the sum of the approximate counters:

$$\hat{M}_{i,\boldsymbol{v}}(\ell) = \sum_{\boldsymbol{u} \in R_i \ltimes \boldsymbol{v}} \tilde{W}^{\emptyset}_{i,\boldsymbol{u}}(\ell), \tag{7}$$

and use it to compute the upper bound $\tilde{M}_{i,\boldsymbol{v}}(\ell) = 2^{\lceil \log \hat{M}_{i,\boldsymbol{v}}(\ell) \rceil}$. Consider a leaf node R_i. As $C_i = \emptyset$, we have $\tilde{W}^{\emptyset}_{i,\boldsymbol{u}}(\ell) = 1$ if $\ell = \phi(\boldsymbol{u})$, and $\tilde{W}^{\emptyset}_{i,\boldsymbol{u}}(\ell) = 0$ otherwise. Consider an internal node R_i. Suppose $\tilde{W}$ and $\tilde{M}$-values have been computed for each child node $R_j \in C_i$. We compute $\tilde{W}^j_{i,\boldsymbol{u}}$ in a decreasing ordering of nodes in C_i. Let $R_{\mathrm{next}(j)} \in C_i$ be the immediate next sibling of R_j.

$$\tilde{W}^j_{i,\boldsymbol{u}}(\ell) = \sum_{(\ell_1,\ell_2) \in [\![\ell - \phi(\boldsymbol{u})]\!] : \ell_1 + \ell_2 = \ell - \phi(\boldsymbol{u})} \tilde{M}_{j,\boldsymbol{u}[\mathrm{key}(j)]}(\ell_1) \cdot \tilde{W}^{\mathrm{next}(j)}_{i,\boldsymbol{u}}(\ell_2). \tag{8}$$

Also, we define $\tilde{W}^{\mathrm{next}(j^*)}_{i,\boldsymbol{u}}(0) = 1$ and $\tilde{W}^{\mathrm{next}(j^*)}_{i,\boldsymbol{u}}(\ell) = 0$ for $\ell > 0$ for the largest child node $R_{j^*} \in C_i$.

[3] A stream $S = \langle x_1, x_2, \ldots, x_n \rangle$ is λ*-dense* for $0 < \lambda \leq 1$, if $r_i \geq \lambda \cdot (i-1)$ for all i, where r_i is the number of real (non-dummy) items in the first $i-1$ items.

Algorithm 7: Update$(i, \boldsymbol{v}, \ell, \Delta)$

Input: $i \in [k]$, tuple $\boldsymbol{v} \in \mathbf{dom}(\mathsf{key}(i))$, score ℓ, and $\tilde{W}^{\emptyset}_{i,\boldsymbol{u}}(\ell)$ increased by Δ for $\boldsymbol{u} \in R_i \ltimes \boldsymbol{v}$.

Update the prefix-sum tree built for $\{\tilde{W}^{\emptyset}_{i,\boldsymbol{u}'}(\ell) : \boldsymbol{u}' \in R_i, \boldsymbol{u}'[\mathsf{key}(i)] = \boldsymbol{u}[\mathsf{key}(i)]\}$;
$\hat{M}_{i,\boldsymbol{v}}(\ell) \leftarrow \hat{M}_{i,\boldsymbol{v}}(\ell) + \Delta$;
$\tilde{M}_{i,\boldsymbol{v}}(\ell) \leftarrow 2^{\lceil \log \hat{M}_{i,\boldsymbol{v}}(\ell) \rceil}$;
if $\tilde{M}_{i,\boldsymbol{v}}(\ell)$ *changes and parent*$(i) \neq$ null **then**
  $p \leftarrow \mathsf{parent}(i)$;
  foreach *tuple* $\boldsymbol{u} \in R_p \ltimes \boldsymbol{v}$ **do**
    Compute $\tilde{W}^{j}_{p,\boldsymbol{u}}(\ell')$ for each score $\ell' \in [\![L-1]\!]$ and $R_j \in C_p$ according to (8);
    foreach $\ell' \in [\![L-1]\!]$ **do**
      if $\tilde{W}^{\emptyset}_{p,\boldsymbol{u}}(\ell')$ *is increased (by* Δ'*)* **then** **Update**$(p, \boldsymbol{u}[\mathsf{key}(p)], \ell', \Delta')$;

We point out that $\tilde{W}^{j}_{i,\boldsymbol{u}}(\ell)$ (resp. $\tilde{M}_{i,\boldsymbol{v}}(\ell)$) is a constant-approximation of $W^{j}_{i,\boldsymbol{u}}(\ell)$ (resp. $M_{i,\boldsymbol{v}}(\ell)$). For each score $\ell \in [\![L-1]\!]$, and each child node $R_j \in C_i$, we maintain a dynamic prefix-sum tree on the values $\tilde{W}^{j}_{i,\boldsymbol{u}}(\ell)$ over all tuples $\boldsymbol{u} \in R_i$ under the pre-determined ordering. This allows us to support point updates and prefix sum queries in $O(\log N)$ time.

Update Procedure. When a new tuple $\boldsymbol{u}$ is inserted into relation R_i, we first initialize its local statistics. If R_i is a leaf node, $\tilde{W}^{\emptyset}_{i,\boldsymbol{u}}(\ell) = 1$ if $\ell = \phi(\boldsymbol{u})$ and $\tilde{W}^{\emptyset}_{i,\boldsymbol{u}}(\ell) = 0$ otherwise. If R_i is an internal node, we compute $\tilde{W}^{\emptyset}_{i,\boldsymbol{u}}(\ell)$ for all score $\ell \in [\![L-1]\!]$ via Equation (8), which takes $O(L \log L)$ time using FFT. After initializing $\boldsymbol{u}$, we must update the statistics in the parent node. Specifically, inserting $\boldsymbol{u}$ increases $\hat{M}_{i,\boldsymbol{v}}(\ell) = \sum_{\boldsymbol{u}' \in R_i \ltimes \boldsymbol{v}} \tilde{W}^{\emptyset}_{i,\boldsymbol{u}'}(\ell)$ for $\boldsymbol{v} = \boldsymbol{u}[\mathsf{key}(i)]$. We update $\hat{M}_{i,\boldsymbol{v}}(\ell)$ and check if $\tilde{M}_{i,\boldsymbol{v}}(\ell)$ changes. If $\tilde{M}_{i,\boldsymbol{v}}(\ell)$ changes, we must propagate this change to the parent relation $R_{\mathsf{parent}(i)}$. This propagation is described in Algorithm 7. When a tuple $\boldsymbol{u}$ is inserted into a leaf R_i, we just invoke Update$(i, \boldsymbol{u}[\mathsf{key}(i)], \phi(\boldsymbol{u}), 1)$. If it is inserted in an inner node we invoke Update$(i, \boldsymbol{u}[\mathsf{key}(i)], \ell, \tilde{W}^{\emptyset}_{i,\boldsymbol{u}}(\ell))$ for every $\ell \in [\![L-1]\!]$.

Complexity Analysis. The update cost is dominated by the propagation of changes. Note that $\tilde{M}_{i,\boldsymbol{v}}(\ell)$ is an upper bound that increases at most $O(\log N)$ times for any fixed $i, \boldsymbol{v}, \ell$, since $\tilde{M}_{i,\boldsymbol{v}}(\ell)$ only increases if $\hat{M}_{i,\boldsymbol{v}}(\ell)$ gets doubled. When $\tilde{M}_{i,\boldsymbol{v}}(\ell)$ changes, we perform convolution operations for tuples in the parent node that can be joined with $\boldsymbol{v}$. Since $L = O(\log N)$, the convolution takes $O(L \log L)$ time. We show that the total number of times $\tilde{M}$ values change across the entire stream of length N is bounded as $O(NL \log N)$. So, the amortized update time is $O(L^2 \log N \log L) = O(\log^3 N \log \log N)$. The space complexity remains $O(NL) = O(N \log N)$.

Query Procedure. The query procedure follows the optimized strategy from Section 3.2, invoking Algorithm 3 on the score buckets $\mathcal{B}_\ell$ (for $\ell < L$) and the tail bucket $\mathcal{B}_{\geq L}$. To retrieve the τ-th tuple from a selected bucket, we employ a modified version of Algorithm 5 that traverses the approximate statistics $\tilde{W}$ and $\tilde{M}$. As discussed in Section 5.1, since these statistics are upper bounds, the index implicitly defines a superset of the join results containing "dummy" tuples. If the retrieved tuple is a dummy, we reject it. Since $\tilde{W}$ is a constant-factor approximation of W, the probability of drawing a real tuple is at least 2^{-c} for some constant c. Consequently, sampling χ tuples takes $O(\chi)$ expected time, and the overall expected query time remains $O(1 + \mu_\Psi \log N)$.

Theorem 5.3. *Given an acyclic join instance Q consisting of k relations, and a set of associated weight functions $\{\mathrm{p}_j\}_{R_j \in Q}$, we can maintain a dynamic index under tuple insertions using $O(N \log N)$ space,*

with amortized update time of $O(\log^3 N \log\log N)$, while supporting answering the subset sampling query in $O(1 + \mu_\Psi \log N)$ expected time, where N is the total number of insertions.

Proof of Theorem 5.3. We first show that all values of $\tilde{W}$, $\hat{M}$, and $\tilde{M}$ are correctly updated. Wlog, assume that all values of $\tilde{W}$, $\tilde{M}$, and $\hat{M}$ are correct in the subtree $\mathcal{T}_i$ up to the moment when Update$(i, \boldsymbol{v}, \ell, \Delta)$ is invoked for the first time at a node R_i located at level h of $\mathcal{T}$. Let Δ denote the correct increment to $\hat{M}_{i,\boldsymbol{v}}(\ell)$. We distinguish two cases. If $\tilde{M}_{i,\boldsymbol{v}}(\ell)$ does not change, then by Equation (8), no value $\tilde{W}$ needs to be updated in any ancestor of R_i. Since $\tilde{W}$ remains unchanged, all corresponding $\hat{M}$ values on the ancestors of R_i also remain unchanged. Therefore, no further updates are required. Otherwise, suppose that $\tilde{M}_{i,\boldsymbol{v}}(\ell)$ changes. We then recompute all values $\tilde{W}^j_{p,\boldsymbol{u}}(\ell')$ according to Equation (8) (Algorithm 7, line 7). If a value $\tilde{W}^j_{p,\boldsymbol{u}}(\ell')$ with $j \neq \emptyset$ increases, then no further $\tilde{W}$ or $\hat{M}$ values in the ancestors of R_i need to be updated. If a value $\tilde{W}^\emptyset_{p,\boldsymbol{u}}(\ell')$ does not increase, again no changes are required in the ancestors. Finally, if $\tilde{W}^\emptyset_{p,\boldsymbol{u}}(\ell')$ increases by a nonzero amount Δ', then by Equation (7), the value $\hat{M}_{i,\boldsymbol{v}}(\ell')$ must be increased by Δ'. Consequently, the recursive call **Update**$(p, \boldsymbol{u}[\text{key}(p)], \ell', \Delta')$ is triggered, and $\hat{M}_{i,\boldsymbol{v}}(\ell')$ is updated at line 2 of Algorithm 7. Applying the argument recursively establishes the correctness of the entire process.

We next show that the amortized update time is $O(\log^3 N \log\log N)$. The key observation is that for each relation R_i and its parent R_p, we update the values $\tilde{W}^j_{p,\boldsymbol{u}}(\ell')$ if and only if $\tilde{M}_{i,\boldsymbol{v}}(\ell)$ changes, where $\boldsymbol{u} \in R_p \ltimes \boldsymbol{v}$ (note that for each $\boldsymbol{u}$ there exists a unique $\boldsymbol{v}$ such that $\boldsymbol{u} \in R_p \ltimes \boldsymbol{v}$). For any fixed $i, \boldsymbol{v}$, and ℓ, the value $\tilde{M}_{i,\boldsymbol{v}}(\ell)$ changes at most $O(\log N)$ times. Whenever an update occurs, we recompute all values $\tilde{W}^j_{p,\boldsymbol{u}}(\ell')$ for every $\ell' \in [\![L - 1]\!]$ using the FFT algorithm, which takes $O(L \log L)$ time. Since there are L possible scores, for fixed i and $\boldsymbol{v}$ there are $O(L)$ distinct values $\tilde{M}_{i,\boldsymbol{v}}(\ell)$. Hence, the values $\tilde{W}^j_{p,\boldsymbol{u}}(\ell')$ are updated at most $O(L \log N)$ times in total, each costing $O(L \log L)$ time. This yields an amortized update time of $O(\log^3 N \log\log N)$. Finally, after every N insertions, we rebuild the dynamic index from scratch and update $N \leftarrow 2N$. This rebuilding step preserves the same amortized update bound of $O(\log^3 N \log\log N)$.

By definition, there exists a constant $c^\star$, depending only on $|\mathcal{T}|$, such that $W^j_{i,\boldsymbol{u}} \leq \tilde{W}^j_{i,\boldsymbol{u}} \leq c^\star W^j_{i,\boldsymbol{u}}$ and $M_{i,\boldsymbol{u}} \leq \tilde{M}_{i,\boldsymbol{u}} \leq c^\star M_{i,\boldsymbol{u}}$, for all i, j, and $\boldsymbol{u}$. Therefore, the index implicitly represents a superset of the true join results that may include dummy tuples. Since the number of non-dummy tuples is a constant fraction of this superset, the probability of sampling a non-dummy tuple using Algorithm 5 is at least 2^{-c}, where c is a constant depending only on $|\mathcal{T}|$, similarly to [23]. Consequently, the expected query time remains $O(1 + \mu_\Psi \log N)$. □

This index can be used to generate a one-shot subset sample with the following running time:

Corollary 5.4. *Given an acyclic join instance Q consisting of k relations, and a set of associated weight functions $\{\mathrm{p}_j\}_{R_j \in Q}$, we can maintain a one-shot subset sample in $O(N \log^3 N \log\log N + \mu_\Psi \log N)$ expected time, where N is the total number of insertions.*

6 Conclusion

We presented the first efficient framework for subset sampling over joins without full materialization. By exploiting the join structure of the query and decomposability of weight functions, our algorithms achieve near-optimal complexity for static indexing, one-shot sampling and dynamic maintenance under insertions. We aim to extend our theoretical framework to support a broader class of weight definitions. While this paper addressed standard aggregation functions, efficient sampling under complex, non-monotonic, or holistic aggregation functions remains open. Furthermore, we plan to bridge the gap between theory and practice by implementing and evaluating our indexes on real-world machine learning workloads over relational data.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases*, volume 8. Addison-Wesley Reading, 1995.

[2] M. Abo-Khamis, S. Im, B. Moseley, K. Pruhs, and A. Samadian. A relational gradient descent algorithm for support vector machine training. In *Symposium on Algorithmic Principles of Computer Systems (APOCS)*, pages 100–113. SIAM, 2021.

[3] P. K. Agarwal, A. Esmailpour, X. Hu, S. Sintos, and J. Yang. Computing a well-representative summary of conjunctive query results. *Proceedings of the ACM on Management of Data*, 2(5):1–27, 2024.

[4] M. Arenas, T. C. Merkl, R. Pichler, and C. Riveros. Towards tractability of the diversity of query answers: Ultrametrics to the rescue. *Proceedings of the ACM on Management of Data*, 2(5):1–26, 2024.

[5] A. Atserias, M. Grohe, and D. Marx. Size Bounds and Query Plans for Relational Joins. *SIAM J. of Comp.*, 42(4): 1737–1767, 2013.

[6] O. Bachem, M. Lucic, and A. Krause. Practical coreset constructions for machine learning. *arXiv preprint arXiv:1703.06476*, 2017.

[7] L. Bekkers, F. Neven, L. Pantelis, and S. Vansummeren. Poisson sampling over acyclic joins. *Proceedings of the ACM on Management of Data*, 4(2), 2026.

[8] C. Berkholz, J. Keppeler, and N. Schweikardt. Answering conjunctive queries under updates. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '17, page 303–318, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450341981.

[9] S. Bhattacharya, P. Kiss, A. Sidford, and D. Wajc. Near-Optimal Dynamic Rounding of Fractional Matchings in Bipartite Graphs. In *STOC*, pages 59–70. ACM, 2024.

[10] A. Borodin and J. I. Munro. *The computational complexity of algebraic and numeric problems*, volume 1. Elsevier, 1975.

[11] K. Bringmann and K. Panagiotou. Efficient sampling methods for discrete distributions. In *ICALP*, pages 133–144, 2012.

[12] N. Carmeli, S. Zeevi, C. Berkholz, B. Kimelfeld, and N. Schweikardt. Answering (unions of) conjunctive queries using random access and random-order enumeration. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 393–409, 2020.

[13] N. Carmeli, N. Tziavelis, W. Gatterbauer, B. Kimelfeld, and M. Riedewald. Tractable orders for direct access to ranked answers of conjunctive queries. *ACM Transactions on Database Systems*, 48(1):1–45, 2023.

[14] S. Chaudhuri, R. Motwani, and V. Narasayya. On random sampling over joins. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, page 263–274, New York, NY, USA, 1999. Association for Computing Machinery. ISBN 1581130848.

[15] J. Chen, Q. Yang, R. Huang, and H. Ding. Coresets for relational data and the applications. *Advances in Neural Information Processing Systems*, 35:434–448, 2022.

[16] Y. Chen and K. Yi. Random Sampling and Size Estimation Over Cyclic Joins. In C. Lutz and J. C. Jung, editors, *23rd International Conference on Database Theory (ICDT 2020)*, volume 155 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 7:1–7:18, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-139-9.

[17] Z. Cheng and N. Koudas. Nonlinear models over normalized data. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1574–1577. IEEE, 2019.

[18] Z. Cheng, N. Koudas, Z. Zhang, and X. Yu. Efficient construction of nonlinear models over normalized data. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 1140–1151. IEEE, 2021.

[19] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[20] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2022.

[21] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends® in Databases*, 4(1–3):1–294, 2012.

[22] R. Curtin, B. Moseley, H. Ngo, X. Nguyen, D. Olteanu, and M. Schleich. Rk-means: Fast clustering for relational data. In *International Conference on Artificial Intelligence and Statistics*, pages 2742–2752. PMLR, 2020.

[23] B. Dai, X. Hu, and K. Yi. Reservoir sampling over joins. In *SIGMOD*, pages 1–26, 2024.

[24] S. Deng, S. Lu, and Y. Tao. On join sampling and the hardness of combinatorial output-sensitive join algorithms. In *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '23, page 99–111, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701276.

[25] N. Duffield, C. Lund, and M. Thorup. Priority sampling for estimation of arbitrary subset sums. *JACM*, 54(6):32, 2007.

[26] A. Esmailpour and S. Sintos. Improved approximation algorithms for relational clustering. *Proceedings of the ACM on Management of Data*, 2(5):1–27, 2024.

[27] D. Feldman and M. Langberg. A unified framework for approximating and clustering data. In *STOC*, pages 569–578, 2011.

[28] J. Gan, S. W. Umboh, H. Wang, A. Wirth, and Z. Zhang. Optimal dynamic parameterized subset sampling. *Proceedings of the ACM on Management of Data*, 2(5):1–26, 2024.
[29] G. Gottlob, G. Greco, F. Scarcello, et al. Treewidth and hypertree width. *Tractability: Practical Approaches to Hard Problems*, 1:20, 2014.
[30] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, page 287–298, New York, NY, USA, 1999. Association for Computing Machinery. ISBN 1581130848.
[31] J. Huang and S. Wang. Subset Sampling and Its Extensions. *arXiv preprint arXiv:2003.01075*, 2023.
[32] J. Huang and S. Wang. Dips: Optimal dynamic index for poisson πps sampling. In *SIGKDD*, pages 520–531, 2025.
[33] J. Huang, Y. Tao, and S. Wang. Acyclic join sampling under selections: Dichotomy, union sampling, and enumeration. In *ICDT*, 2026.
[34] A. Kara, M. Nikolic, D. Olteanu, and H. Zhang. F-ivm: analytics over relational databases under updates. *The VLDB Journal*, 33(4):903–929, 2024.
[35] M. A. Khamis, H. Q. Ngo, X. Nguyen, D. Olteanu, and M. Schleich. Ac/dc: in-database learning thunderstruck. In *Proceedings of the second workshop on data management for end-to-end machine learning*, pages 1–10, 2018.
[36] K. Kim, J. Ha, G. Fletcher, and W.-S. Han. Guaranteeing the $\tilde{O}$(agm/out) runtime for uniform sampling and size estimation over joins. In *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '23, page 113–125, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701276.
[37] A. Kumar, J. Naughton, and J. M. Patel. Learning generalized linear models over normalized data. In *SIGMOD*, pages 1969–1984, 2015.
[38] F. Li, B. Wu, K. Yi, and Z. Zhao. Wander join: Online aggregation via random walks. In *Proceedings of the 2016 International Conference on Management of Data*, pages 615–629, 2016.
[39] D. Marx. Tractable hypergraph properties for constraint satisfaction and conjunctive queries. *Journal of the ACM (JACM)*, 60(6):1–51, 2013.
[40] T. C. Merkl, R. Pichler, and S. Skritek. Diversity of answers to conjunctive queries. *Logical Methods in Computer Science*, 21, 2025.
[41] B. Moseley, K. Pruhs, A. Samadian, and Y. Wang. Relational algorithms for k-means clustering. In *International Colloquium on Automata, Languages, and Programming*, 2021.
[42] R. Motwani and P. Raghavan. *Randomized algorithms*. Cambridge University Press, 1995.
[43] H. Q. Ngo. Worst-case optimal join algorithms: Techniques, results, and open problems. In *PODS*, pages 111–124. ACM, 2018.
[44] F. Olken. *Random sampling from databases*. PhD thesis, University of California, Berkeley, 1993.
[45] F. P. Preparata and M. I. Shamos. Computational geometry. texts and monographs in computer science. *Berlin, Springer-Verlag*, 1985.
[46] M. Schleich, D. Olteanu, and R. Ciucanu. Learning linear regression models over factorized joins. In *SIGMOD*, pages 3–18, 2016.
[47] M. Schleich, D. Olteanu, M. Abo-Khamis, H. Q. Ngo, and X. Nguyen. Learning models over relational data: A brief tutorial. In *Scalable Uncertainty Management: 13th International Conference, SUM 2019*, pages 423–432. Springer, 2019.
[48] V. Surianarayanan, N. Kumar, and S. Sintos. Clustering with set outliers and applications in relational clustering. *Proc. ACM Manag. Data*, 3(5), 2025. doi: 10.1145/3767712.
[49] K. Yang, Y. Gao, L. Liang, B. Yao, S. Wen, and G. Chen. Towards factorized svm with gaussian kernels over normalized data. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1453–1464. IEEE, 2020.
[50] M. Yannakakis. Algorithms for acyclic database schemes. In *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*, VLDB '81, page 82–94. VLDB Endowment, 1981.
[51] L. Yi, H. Wang, and Z. Wei. Optimal Dynamic Subset Sampling: Theory and Applications. In *SIGKDD*, pages 3116–3127. ACM, 2023.
[52] Z. Zhao, R. Christensen, F. Li, X. Hu, and K. Yi. Random sampling over joins revisited. In *SIGMOD*, 2018.

A Full Notation Table

Please find our main notations in Table 2.

	Notation	Description
General	S	A set of elements.
	p	A probability function assigning a probability to each element in S.
	$\Psi = \langle S, \mathrm{p} \rangle$	A subset sampling problem instance.
	$\mathbf{X}$	A random subset sample from a subset sampling instance.
	μ_Ψ	The expected size of a subset sample $\mathbf{X}$ from Ψ.
	$[x]$	The set of integers $\{1, 2, \ldots, x\}$.
	$[\![x]\!]$	The set of integers $\{0, 1, 2, \ldots, x\}$.
	DirectAccess	A Direct Access oracle.
Subset Sampling over Joins	$\mathbf{att}, \mathbf{dom}$	The set of all attributes and the domain of all values.
	$\boldsymbol{u}, \boldsymbol{v}$	Tuples.
	R_i	The i-th relation in a join query.
	$\mathrm{schema}(R_i)$	The schema (set of attributes) of relation R_i.
	$Q = \{R_1, \ldots, R_k\}$	A join query consisting of k relations.
	N	The input size of Q, i.e., $\sum_{i=1}^{k} \lvert R_i \rvert$.
	$\mathrm{Join}(Q)$	The set of result tuples from the join query Q.
	$G = (V, E)$	The schema hypergraph of a join query Q.
	ρ^*	The fractional edge covering number of schema graph G.
	$\mathrm{p}_i(\boldsymbol{u})$	The probability associated with a tuple $\boldsymbol{u} \in R_i$.
	$\mathrm{p}(\boldsymbol{u})$	Prob. of join result $\boldsymbol{u}$, defined as $\prod_{i=1}^{k} \mathrm{p}_i(\boldsymbol{u}[\mathrm{schema}(R_i)])$.
	L	The number of partitions/buckets, defined as $\lceil \rho^* \log N \rceil$.
	$\phi(\boldsymbol{u})$	The score of a tuple $\boldsymbol{u}$.
	$\mathcal{B}_\ell$	The set (bucket) of join results with score ℓ.
Optimized Index	$\mathcal{T}$	Join tree for an acyclic join.
	$\mathcal{T}_i$	The subtree of $\mathcal{T}$ rooted at relation R_i.
	C_i	The set of child nodes of relation R_i in $\mathcal{T}$.
	$\mathcal{T}_i^j$	The partial subtree of $\mathcal{T}_i$ excluding children preceding R_j.
	$\mathrm{key}(i)$	Common join attributes between R_i and its parent in $\mathcal{T}$.
	$W_{i,\boldsymbol{u}}^j(\ell)$	# of join results with score ℓ in $\mathcal{T}_i^j$ involving $\boldsymbol{u}$.
	$M_{i,\boldsymbol{v}}(\ell)$	An aggregated count of W-values for tuples projecting to $\boldsymbol{v}$.
Dynamic	η	A timestamp in the streaming setting.
	Q^η	The join defined by the first η tuples in the stream.
	p^η	The weight functions associated with Q^η.
	$\tilde{W}_{i,\boldsymbol{u}}^j(\ell)$	An approximated upper bound of $W_{i,\boldsymbol{u}}^j(\ell)$.
	$\tilde{M}_{i,\boldsymbol{v}}(\ell)$	An approximated upper bound of $M_{i,\boldsymbol{v}}(\ell)$.

Table 2. Table of Notations

B Missing Materials in Section 3

Formally, we prove the correctness of Algorithm 5 using the following lemma.

Lemma B.1. *Let DirectAccess $(S_{\hat{\ell}}, \hat{\tau})$ and DirectAccess $(S_{\bar{\ell}}, \bar{\tau})$ be two distinct calls in Algorithm 3. Let $\hat{\boldsymbol{w}} \in \text{Join}(Q)$ be the tuple returned by Algorithm 5 on input $(S_{\hat{\ell}}, \hat{\tau})$, and let $\bar{\boldsymbol{w}} \in \text{Join}(Q)$ be the tuple returned on input $(S_{\bar{\ell}}, \bar{\tau})$. Then $\hat{\boldsymbol{w}} \neq \bar{\boldsymbol{w}}$.*

Proof. We assume that all the W and M-values are correctly computed. Moreover, $\hat{\ell} \neq \bar{\ell}$. We will show that $\hat{\boldsymbol{w}} \in \mathcal{B}_{\hat{\ell}}$ and $\bar{\boldsymbol{w}} \in \mathcal{B}_{\bar{\ell}}$. Consider any call of DirectAccess $(S_{\ell'}, \tau')$ in Algorithm 3. We claim that Algorithm 5 always returns a join result from $\mathcal{B}_{\ell'}$. We prove this by strong induction on the height of the join tree $\mathcal{T}$. If the join tree has a single leaf node R_i (height 1), then in Line 1 the algorithm only considers tuples of R_i with score ℓ', so the returned tuple belongs to $\mathcal{B}_{\ell'}$. Assume that the claim holds for all join trees of height at most h. Let R_i be the root of a join tree of height $h+1$. In Line 4, the algorithm considers pairs (ℓ_1, ℓ_2) such that $\ell_1 + \ell_2 + \phi(\boldsymbol{u}) = \ell'$. By the induction hypothesis, we have $\phi(\boldsymbol{u}_1) = \ell_1$ and $\phi(\boldsymbol{u}_2) = \ell_2$, and therefore $\phi(\boldsymbol{u}_1 \bowtie \boldsymbol{u} \bowtie \boldsymbol{u}_2) = \ell'$.

Applying this argument to DirectAccess $(S_{\hat{\ell}}, \hat{\tau})$ and DirectAccess $(S_{\bar{\ell}}, \bar{\tau})$, we obtain $\hat{\boldsymbol{w}} \in \mathcal{B}_{\hat{\ell}}$ and $\bar{\boldsymbol{w}} \in \mathcal{B}_{\bar{\ell}}$. Since $\hat{\ell} \neq \bar{\ell}$, we have $\mathcal{B}_{\hat{\ell}} \cap \mathcal{B}_{\bar{\ell}} = \emptyset$, and thus $\hat{\boldsymbol{w}} \neq \bar{\boldsymbol{w}}$. We now assume that $\hat{\ell} = \bar{\ell} = \ell^*$ and $\hat{\tau} \neq \bar{\tau}$. Wlog, suppose that $\hat{\tau} > \bar{\tau}$. We prove the claim by strong induction on the height of the join tree. If the join tree consists of a single leaf node R_i (height 1), let $\hat{\boldsymbol{u}}$ (resp., $\bar{\boldsymbol{u}}$) be the smallest tuple found in Line 1 for the pair $(\hat{\ell}, \hat{\tau})$ (resp., $(\bar{\ell}, \bar{\tau})$). For every $\boldsymbol{u}' \in R_i$, the value $W^j_{i,\boldsymbol{u}'}(\ell^*)$ is equal to 1 if $\phi(\boldsymbol{u}') = \ell^*$ and 0 otherwise. Hence, the sum $\sum_{\boldsymbol{u}' \in R_i : \boldsymbol{u}' \preceq \boldsymbol{u}} W^j_{i,\boldsymbol{u}'}(\ell^*)$ increases by one for each new qualifying tuple. Since $\hat{\tau} \neq \bar{\tau}$, we conclude that $\hat{\boldsymbol{u}} \neq \bar{\boldsymbol{u}}$ and therefore $\hat{\boldsymbol{w}} \neq \bar{\boldsymbol{w}}$.

Assume that the claim holds for all join trees of height at most h. Let R_i be the root of a join tree of height $h+1$. If $\hat{\boldsymbol{u}} \neq \bar{\boldsymbol{u}}$ in Line 1, then clearly $\hat{\boldsymbol{w}} \neq \bar{\boldsymbol{w}}$. Hence, we assume that $\hat{\boldsymbol{u}} = \bar{\boldsymbol{u}} = \boldsymbol{u}$. The sum $\sum_{\boldsymbol{u}' \in R_i : \boldsymbol{u}' \preceq \boldsymbol{u}} W^j_{i,\boldsymbol{u}'}(\ell^*)$ is identical for both instances, so $\hat{\tau}$ and $\bar{\tau}$ are reduced by the same amount in Line 2. By a slight abuse of notation, we again denote the updated values by $\hat{\tau}$ and $\bar{\tau}$. Let $(\hat{\ell}_1, \hat{\ell}_2)$ and $(\bar{\ell}_1, \bar{\ell}_2)$ be the smallest pairs found in Line 5 for the inputs $(\hat{\ell}, \hat{\tau})$ and $(\bar{\ell}, \bar{\tau})$, respectively. If $\hat{\ell}_1 \neq \bar{\ell}_1$ or $\hat{\ell}_2 \neq \bar{\ell}_2$, then by the first part of the proof we immediately obtain $\hat{\boldsymbol{w}} \neq \bar{\boldsymbol{w}}$. Hence, we assume that $\hat{\ell}_1 = \bar{\ell}_1 = \ell_1$ and $\hat{\ell}_2 = \bar{\ell}_2 = \ell_2$. The sum $\sum_{(\ell'_1, \ell'_2) \in \Phi : (\ell'_1, \ell'_2) \prec (\ell_1, \ell_2)} M_{j, \boldsymbol{u}[\text{key}(j)]}(\ell'_1) \cdot W^{\text{next}(j)}_{i,\boldsymbol{u}}(\ell'_2)$ is identical for both instances, so $\hat{\tau}$ and $\bar{\tau}$ are again reduced by the same amount in Line 6.

We now reach the final stage of the algorithm, where the smallest tuples and score pairs coincide, but $\hat{\tau} > \bar{\tau}$. We compute $\hat{\tau}_1, \hat{\tau}_2$ and $\bar{\tau}_1, \bar{\tau}_2$ as in Lines 7 and 8. The denominator $W^{\text{next}(j)}_{i,\boldsymbol{u}}(\ell_2)$ in Line 7 is the same in both executions. If $\hat{\tau}_1 \neq \bar{\tau}_1$, then by the induction hypothesis $\hat{\boldsymbol{u}}_1 \neq \bar{\boldsymbol{u}}_1$, which implies $\hat{\boldsymbol{w}} \neq \bar{\boldsymbol{w}}$. Otherwise, assume that $\hat{\tau}_1 = \bar{\tau}_1 = \tau_1$. There are two cases. If neither $\hat{\tau}$ nor $\bar{\tau}$ is divisible by $W^{\text{next}(j)}_{i,\boldsymbol{u}}(\ell_2)$, then $\hat{\tau}_2 = ((\hat{\tau} - 1) \bmod W^{\text{next}(j)}_{i,\boldsymbol{u}}(\ell_2)) + 1 = \hat{\tau} \bmod W^{\text{next}(j)}_{i,\boldsymbol{u}}(\ell_2) > 0$ and $\bar{\tau}_2 = ((\bar{\tau} - 1) \bmod W^{\text{next}(j)}_{i,\boldsymbol{u}}(\ell_2)) + 1 = \bar{\tau} \bmod W^{\text{next}(j)}_{i,\boldsymbol{u}}(\ell_2) > 0$. Since $\hat{\tau} > \bar{\tau}$, it follows that $\hat{\tau}_2 > \bar{\tau}_2$, and by the induction hypothesis $\hat{\boldsymbol{u}}_2 \neq \bar{\boldsymbol{u}}_2$, again implying $\hat{\boldsymbol{w}} \neq \bar{\boldsymbol{w}}$. In the second case, $\hat{\tau}$ is divisible by $W^{\text{next}(j)}_{i,\boldsymbol{u}}(\ell_2)$ but $\bar{\tau}$ is not. Then $\hat{\tau}_2 = ((\hat{\tau} - 1) \bmod W^{\text{next}(j)}_{i,\boldsymbol{u}}(\ell_2)) + 1 = W^{\text{next}(j)}_{i,\boldsymbol{u}}(\ell_2)$ while $\bar{\tau}_2 = ((\bar{\tau} - 1) \bmod W^{\text{next}(j)}_{i,\boldsymbol{u}}(\ell_2)) + 1 = \bar{\tau} \bmod W^{\text{next}(j)}_{i,\boldsymbol{u}}(\ell_2) < W^{\text{next}(j)}_{i,\boldsymbol{u}}(\ell_2)$. Hence $\hat{\tau}_2 > \bar{\tau}_2$, and by the induction hypothesis $\hat{\boldsymbol{u}}_2 \neq \bar{\boldsymbol{u}}_2$, which concludes that $\hat{\boldsymbol{w}} \neq \bar{\boldsymbol{w}}$. □

C Discussion On Other Functions

We next discuss the extension to other aggregation functions. For our algorithms, the results can be extended to support MIN, MAX, and SUM. We will show how to adapt the high-level idea in Section 3.1 to these functions.

MIN and MAX. Without loss of generality, we consider the function MIN. We assume that a DirectAccess oracle is available for the input join Q (under some fixed ordering), such that it receives an integer $i \in [|\text{Join}(Q)|]$, and returns the i-th element of $\text{Join}(Q)$. Similar to before, the ordering can be arbitrary but must remain consistent across multiple invocations of this oracle. Let

$L = \lceil 2\rho^* \log N \rceil$. The first step of partitioning join result is exactly the same as before. The second step of applying subset sampling to all sub-instances also follows. Let $\boldsymbol{j} = (j_1, j_2, \ldots, j_k)$. Every join result in the sub-instance $Q_{\boldsymbol{j}}$ has the probability $\min_{i\in[k]} 2^{-j_i} = 2^{-\max_{i\in[k]} j_i}$ to be sampled. We first point out the following observation on each sub-instance in such a partition:

Lemma C.1 (Either Light or Near-uniform Sub-instance)**.** *Let $\boldsymbol{j} = (j_1, j_2, \ldots, j_k)$. If $\max_{i\in[k]} j_i \geq L$, the instance $\Psi_{\boldsymbol{j}}$ is light, and otherwise, it is 2-uniform.*

Intuitively, if $\max_{i\in[k]} j_i \geq L$, every join result has its probability at most $\frac{1}{|\text{Join}(Q)|}$, so by definition this sub-instance is light. Otherwise, $\max_{i\in[k]} j_i < L$. Now, the ratio between the maximum and the minimum probability is at most 2, so by definition this sub-instance is 2-uniform. Hence,

- **Preprocessing phase:** We partition input tuples by weights, as well as the join instance Q. For each sub-instance $Q_{\boldsymbol{j}}$ with $\boldsymbol{j} \in [\![L]\!]^k$, we build DirectAccess oracles and compute the join size.
- **Query phase:** We invoke Algorithm 3 with input $\{\langle Q_{\boldsymbol{j}}, p\rangle : \boldsymbol{j} \in [\![L]\!]^k\}$ with $p^+_{\boldsymbol{j}} = 2^{-\max_{i\in[k]} j_i}$. Whenever a join result is sampled from $Q_{\boldsymbol{j}}$, we use the DirectAccess oracle to retrieve it.

For the optimized index, for every $\boldsymbol{u}$ in R_i we define the score $\phi(\boldsymbol{u}) = \lfloor -\log \mathrm{p}_i(\boldsymbol{u}) \rfloor$. For a join result $\boldsymbol{u} \in \text{Join}(Q)$, its score is $\bar{\phi}(\boldsymbol{u}) = \max_{i=1}^{k} \phi(\boldsymbol{u}[\text{schema}(R_i)])$. The buckets $\mathcal{B}_\ell = \{\boldsymbol{u} \in \text{Join}(Q) \mid \bar{\phi}(\boldsymbol{u}) = \ell\}$ are defined in a similar way, as for the product function. We note that for any $\boldsymbol{u} \in \mathcal{B}_\ell$, $2^{-\ell-1} < \mathrm{p}(\boldsymbol{u}) \leq 2^{-\ell}$, hence the ratio between the maximum and minimum probability in each bucket $\mathcal{B}_\ell$ is at most 2. The index is similar to the one we constructed in Section 3.2. During the query phase, we run an algorithm similar to Algorithm 5. In Steps 4 and 5 the number of pairs we try is not $L = O(\log N)$, as in the case of the product function, but $O(L^2) = O(\log^2 N)$. Additionally, we do not apply the FFT in the preprocessing phase. Overall, our results extend to the MIN (and MAX) function with the same guarantees up to a $\log N$ factor.

SUM. Similarly, our indexing framework naturally extends to the SUM function. Interestingly, SUM can be handled in exactly the same way as MAX: we define identical scores and buckets for the SUM function as those used for the MAX function. Namely, for every $\boldsymbol{u}$ in R_i we define the score $\phi(\boldsymbol{u}) = \lfloor -\log \mathrm{p}_i(\boldsymbol{u}) \rfloor$. For a join result $\boldsymbol{u} \in \text{Join}(Q)$, its score is $\bar{\phi}(\boldsymbol{u}) = \min_{i=1}^{k} \phi(\boldsymbol{u}[\text{schema}(R_i)])$. For every ℓ we define the bucket $\mathcal{B}_\ell = \{\boldsymbol{u} \in \text{Join}(Q) \mid \bar{\phi}(\boldsymbol{u}) = \ell\}$. We note that for any $\boldsymbol{u} \in \mathcal{B}_\ell$, $2^{-\ell-1} < \mathrm{p}(\boldsymbol{u}) \leq k \cdot 2^{-\ell}$, hence the ratio between the maximum and minimum probability in each bucket $\mathcal{B}_\ell$ is at most $2k = O(1)$. The index is similar to the one we constructed in Section 3.2. During the query phase, we run an algorithm similar to Algorithm 5. In Steps 4 and 5 the number of pairs we try is not $L = O(\log N)$, as in the case of the product function, but $O(L^2) = O(\log^2 N)$. Additionally, we do not apply the FFT in the preprocessing phase. Overall, our results extend to the SUM function with the same guarantees up to a $\log N$ factor.

Received December 2025; accepted February 2026