Poisson Sampling over Acyclic Joins

LIESE BEKKERS, UHasselt, Data Science Institute, Belgium
FRANK NEVEN, UHasselt, Data Science Institute, Belgium
LORRENS PANTELIS, UHasselt, Data Science Institute, Belgium
STIJN VANSUMMEREN, UHasselt, Data Science Institute, Belgium

We introduce the problem of Poisson sampling over joins: compute a sample of the result of a join query by conceptually performing a Bernoulli trial for each join tuple, using a non-uniform and tuple-specific probability. We propose an algorithm for Poisson sampling over acyclic joins that is nearly instance-optimal, running in time $O(|db| + k \log |db|)$ where $|db|$ is the size of the input database, and k is the size of the resulting sample. Our algorithm hinges on two building blocks: (1) The construction of a random-access index that allows, given a number i, to randomly access the i-th join tuple without fully materializing the (possibly large) join result; (2) The probing of this index to construct the result sample. We study the engineering trade-offs required to make both components practical, focusing on their implementation in column stores, and identify best-performing alternatives for both. Our experiments show that this pair of alternatives significantly outperforms the repeated-Bernoulli-trial algorithm for Poisson sampling while also demonstrating that the random-access index by itself can be used to competively implement Yannakakis' acyclic join processing algorithm when no sampling is required. This shows that, as far a query engine design is concerned, it is possible to adopt a common basis for both classical acyclic join processing and Poisson sampling, both without regret compared to classical join and sampling algorithms.

CCS Concepts: • **Theory of computation** → *Database query processing and optimization (theory)*; *Database query languages (principles)*; **Sketching and sampling**; • **Information systems** → Main memory engines; **Query optimization**; *Join algorithms.*

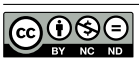
Additional Key Words and Phrases: Poisson sampling, acyclic joins, Yannakakis, semijoin, join, database, query processing, nested relational model

ACM Reference Format:
Liese Bekkers, Frank Neven, Lorrens Pantelis, and Stijn Vansummeren. 2026. Poisson Sampling over Acyclic Joins. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 224 (June 2026), 26 pages. https://doi.org/10.1145/3802101

1 Introduction

Drawing a fixed-size sample from the result of a query has numerous interesting applications including online aggregation [1, 16, 26], large-scale analytics [38] and query optimization in general. Formally, the sampling problem is usually phrased as follows: given a query Q, input database db and desired sample size k, draw k tuples uniformly at random from $Q(db)$, without replacement. A naive solution is to first materialize $Q(db)$ in a table, and then randomly access the table to perform the sampling. However, this naive method is inefficient when Q is a join query since it requires to compute the full join result—which can be orders of magnitude larger than both the input database db and the sample size k—and hence wastes time computing tuples that do not contribute to the

Authors' Contact Information: Liese Bekkers, UHasselt, Data Science Institute, Hasselt, Belgium, liese.bekkers@uhasselt.be; Frank Neven, UHasselt, Data Science Institute, Hasselt, Belgium, frank.neven@uhasselt.be; Lorrens Pantelis, UHasselt, Data Science Institute, Hasselt, Belgium, lorrens.pantelis@student.uhasselt.be; Stijn Vansummeren, UHasselt, Data Science Institute, Hasselt, Belgium, stijn.vansummeren@uhasselt.be.



ACM 2836-6573/2026/6-ART224
https://doi.org/10.1145/3802101

desired output. Alternative techniques that avoid computing the full join exist and are typically based on designing an index structure that can be used to guide the sampling process [1, 8, 38]. Specifically, for *acyclic joins* (formally defined in Section 2) index structures have been proposed that can be computed in $O(|db|)$ time, which can then be used to sample a single result tuple in $O(\log |db|)$ time—yielding overall complexity $O(|db| + k \log |db|)$ [7, 10].

In this paper, instead of drawing a fixed-size (uniform) sample, which requires each output tuple to be sampled with the same uniform probability, we study the following more general problem, which we call *Poisson sampling*. In this problem, each join tuple output by Q specifies the (not necessarily uniform) probability with which it is to be included in the sample, and the sample is to be taken by conceptually doing a Bernoulli trial for each tuple with its specified probability.

Poisson sampling has applications in Markov-chain based simulations [6, 22]. In particular, we are motivated by our efforts of designing EpiQL, a declarative language and accompanying high-performance data engine that focuses on simulation of discrete agent-based infectious disease models [18, 31]. Such models are used by epidemiologists to predict the evolution of transmissible diseases such as measles, influenza, or covid in various situations.

Example 1.1. Infectious disease models simulate contact events between individuals in a population in order to calculate, at each simulation timestep, which subset of the population is susceptible to infection, which subset is infectious, which has recovered from the infection, and so on. In EpiQL, the following conjunctive query rules can be used to define the transition from the `Susceptible` *to* `Infected` *disease state. Other rules (not shown) compute from the* `Infected` *population the subpopulation that becomes* `Infectious`*, and which can hence cause further spread in later simulation steps.*

$$\begin{aligned}\texttt{Infected}(\texttt{per1}) \leftarrow\ & \texttt{Susceptible}(\texttt{per1}), \texttt{Infectious}(\texttt{per2}),\\ & \texttt{Contact}(\texttt{per1}, \texttt{per2}), \mathit{Bernoulli}(p_t)\\ \texttt{Contact}(\texttt{per1}, \texttt{per2}) \leftarrow\ & \texttt{Person}(\texttt{per1}, \texttt{age1}, \texttt{pool}),\\ & \texttt{Person}(\texttt{per2}, \texttt{age2}, \texttt{pool}),\\ & \texttt{ContactProb}(\texttt{pool}, \texttt{age1}, \texttt{age2}, \texttt{prob}),\\ & \mathit{Bernoulli}(\texttt{prob})\end{aligned}$$

The first query states that susceptible person `per1` *may become infected with probability p_t if there was a contact event with infectious person* `per2`*. Disease models simulate contact events by dividing the population into* contact pools *(e.g., family households, schools, workplaces) and modeling contact within a pool according to statistics derived from real-world diary studies [19, 30]. In line with this, the second query above uses a relation* `ContactProb(pool, age1, age2, prob)` *that indicates that in* `pool` *there is a* `prob` *chance that two people aged* `age1` *and* `age2` *meet, and a relation* `Person(pers, age, pool)` *listing each individual's age and pool memberships. In particular, the* `Contact` *query joins the* `Person` *and* `ContactProb` *relations. From the resulting join, each tuple is independently sampled with probability* `prob`*. The sampled tuples are then projected onto the participating persons. We note that both conjunctive queries are acyclic.*

Similar to the case for fixed-size sampling, it is important to design algorithms for Poisson sampling *without* first materializing the full join result. To illustrate, in EpiQL we found that, even for a small country with a population of $\pm 10^7$ people using `ContactProb` data collected from real-world diary studies [19, 30], the full join output size of the contact query is $\pm 10^{10}$ whereas the expected sample size is only $\pm 10^8$, two orders of magnitude less.

In this paper, we design algorithms for Poisson sampling over acyclic joins that avoid materialization of the full join result and which are nearly instance-optimal from an asymptotic complexity viewpoint. Furthermore, we study the engineering trade-offs required to make these algorithms practical in column stores. Our contributions are as follows:

(1) We introduce the problem of Poisson sampling over join queries, which generalizes fixed-size sampling.

(2) For acyclic joins, we show that Poisson sampling can be solved in the same time as fixed-size sampling: $O(|db| + k \log |db|)$, where $|db|$ is the size of the input database, and k is the size of the resulting sample (Theorem 5.1). From an asymptotic complexity viewpoint, this is *instance-optimal* up to a $\log |db|$ factor since any correct algorithm will need to read the input and produce the sample. Conceptually, we follow an Index-and-Probe strategy: *(i)* we construct a random-access index for a join query Q that avoids materializing the full result $Q(db)$ while still allowing efficient retrieval of the j-th join tuple for each position j; *(ii)* we determine the sequence of tuple positions in $Q(db)$ that define the output sample (referred to as position sampling); *(iii)* we generate the sample by repeatedly probing this index.

(3) Because asymptotic complexity may hide crucial constant factors, we next investigate the engineering trade-offs involved in making this conceptual algorithm efficient in practice, focusing on its implementation in column stores.

(a) *Random-access index construction.* For acyclic joins it is known that a random-access index can be constructed in $O(|db|)$ time that allows random access to a single tuple in $O(\log |db|)$ time [5, 7]. Note that, crucially, the query result $Q(db)$ may be much larger than $|db|$. Constructing a random-access index is closely tied to Yannakakis' seminal algorithm (YA) for processing acyclic joins [34], but with additional logic to ensure $O(\log |db|)$ access time. There has recently been extensive interest in implementing YA in a manner that is also competitive with traditional binary join algorithms for inputs with no or little dangling join tuples—situations where YA has traditionally been slower than binary joins [3, 20, 24, 29, 33, 37]. We observe that the column store implementation approach of Bekkers et al [3], which is called Shredded Yannakakis (SYA), already provides a linear-time-constructable random-access index. This implementation is based on a *chained shredded representation* (CSR) but has access time $O(\log |db| + d)$ instead of $O(\log |db|)$ where d is the largest join-degree in the input database. To recover a $O(\log |db|)$ access time, we implement an *unchained shredded representation* (USR) within the shredded Yannakakis framework which lifts the concept of a random-access index structure as proposed by Carmeli et al [7] to column stores. However, we empirically find that CSR is faster to build and, suprisingly, also faster to probe which causes CSR-based sampling to outperform USR-based sampling.

(b) *Position sampling.* We distinguish between the uniform and non-uniform cases. For uniform sampling, we compare three strategies for generating the sequence of probe positions. We demonstrate that these have complementary advantages depending on the specific sampling probability that is being used. Based on this observation, we design a hybrid position-sequence generation algorithm that dynamically adapts to the observed data distribution. For the non-uniform case, we reduce the problem to a series of uniform sampling steps over groups of tuples sharing the same sampling probability, applying the hybrid method to each group. Further details are provided in Section 5.

(4) We implement all methods inside of Apache Datafusion, a high-performance main-memory-based columnar query engine written in Rust, and experimentally compare these methods based on established join query benchmarks over real-world data as well as the disease transmission use case. We find that:

(a) The combination of CSR with our hybrid position-sampling method is most efficient across all benchmarks, even though CSR has worse asymptotic complexity than USR.

(b) Compared to the naive approach that first materializes the join result and then performs a Bernoulli trial per join tuple, our method is up to 6.08x faster.

(c) Finally, we observe that CSR can also be used for normal join processing (without sampling), and there we find that CSR is competitive with USR on a wide variety of benchmarks. This illustrates that, as far as query engine design is concerned, it suffices to adopt a *single* strategy for implementing Yannakakis without regret in a column store,[1] one based on CSR, as this allows for both normal join processing and sampling. We believe that this insight is relevant because we know of no current column store that implements the known efficient fixed-size sampling algorithms introduced in [7, 10, 38], presumably because it requires intricate changes to the engine's internals. By adopting CSR we achieve two goals at once: enable efficient sampling and ensure robust acyclic join processing.

Organization. Section 2 defines the Poisson sampling problem, and Section 3 outlines our approach and background. Sections 4 and 5 resp. cover indexing and probing. Section 6 presents experiments, Section 7 discusses related work, and Section 8 concludes.

2 Problem Statement

For a natural number $l > 0$, we denote the set $\{1, \ldots, l\}$ by $[l]$. We are interested in processing *Poisson sampling queries*, which are queries of the following form:

$$Q = \beta_y \left(R_1(\overline{x}_1) \bowtie \cdots \bowtie R_l(\overline{x}_l)\right). \tag{1}$$

Here, $l \geq 1$; each R_i is a (not necessarily distinct) relation symbol; each $\overline{x}_i$ is a set of pairwise distinct attributes that denotes the schema of R_i, for $i \in [l]$; and $y \in \overline{x}_1 \cup \cdots \cup \overline{x}_l$. Expressions of the form $R_i(\overline{x}_i)$ are called *atoms*. The expression $R_1(\overline{x}_1) \bowtie \cdots \bowtie R_l(\overline{x}_l)$ is called the *(full) join query* underlying Q, which we denote by $\hat{Q}$.

Throughout the paper, we adopt a bag-based semantics for relations and queries. Formally, tuple t over $\overline{x}_1 \cup \cdots \cup \overline{x}_l$ occurs in the join result of $\hat{Q}(db)$ of $\hat{Q}$ on database db if for every $i \in [l]$ the tuple $t[\overline{x}_i]$ (i.e., t projected on $\overline{x}_i$), occurs with multiplicity $m_i > 0$ in input relation R_i. The result multiplicity of t in the full join is then $m_1 \times \cdots \times m_l$.

The operator β has the following semantics. If the values of the y-attribute in $\hat{Q}(db)$ are in the range $[0, 1]$, then β_y performs a Bernoulli trial with probability $t[y]$ for each tuple $t \in \hat{Q}(db)$, keeping the tuple if the trial succeeds and discarding it otherwise. The output of β_y is hence non-deterministic, as it depends on the stochastic random choices made. In other words, β is a randomized relational operator. In statistics, a process where each element of a population is subjected to an independent Bernoulli trial that determines whether the element becomes part of a sample is called *Poisson sampling*. We hence refer to β as the *Poisson sampling operator*. For simplicity, we assume throughout the paper that the y-values are always in the range $[0, 1]$.

Example 2.1. The rule defining `Contact(per1, per2)` *in Example 1.1 can be formalized as the Poisson sampling query*

$$\begin{aligned} Q_c = \beta_{\texttt{prob}} \big(&\texttt{Person(per1, age1, pool)} \\ &\bowtie \texttt{Person(per2, age2, pool)} \\ &\bowtie \texttt{ContactProb(pool, age1, age2, prob)}\big). \end{aligned}$$

Acyclicity. Throughout the paper, we focus on Poisson sampling queries for which $\hat{Q}$ is acyclic. A join query $\hat{Q}$ is *acyclic* if it admits a join tree [2, 14]. A *join tree for* $\hat{Q}$ is a rooted undirected tree J in which each node is an atom of $\hat{Q}$. To be correct under bag semantics, it is required that each atom in $\hat{Q}$ appears exactly as many times in J as it does in $\hat{Q}$. Join trees are required to satisfy the

[1] That is, in a manner that is competitive with traditional binary join algorithms for *all* kind of inputs and not just for inputs with many dangling join tuples for which YA is known to be efficient.

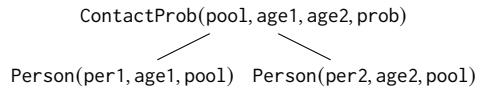


Fig. 1. A join tree for the join query of Example 2.1.

connectedness property: for every attribute x, all the atoms containing x form a connected subtree of J. To illustrate, Figure 1 shows a join tree for the join query in Example 2.1. The triangle query $R(x, y) \bowtie S(y, z) \bowtie T(z, x)$ is the prototypical example of a join query that is *cyclic*, i.e., not acyclic. Checking whether a join query is acyclic and constructing a join tree if it exists can be done in linear time w.r.t. the size of the query by means of the GYO algorithm [15, 27, 35].

Processing. We are interested in the efficient processing of acyclic Poisson sampling queries. Just like a deterministic query may have multiple algorithms (i.e., physical query plans) that implement the query—differing in their efficiency—a Poisson sampling query may have multiple randomized algorithms implementing it.

The naive way to process $Q(db)$ is to first materialize the full join $\hat{Q}(db)$, and then iterate over the elements t of the resulting bag—flipping a coin with probability $t[y]$ to see if t needs to be included in the output. This naive method, which we will refer to as *Materialize-and-Scan (M&S for short)*, clearly has complexity $\Omega(|db| + |\hat{Q}(db)|)$. Note that $|\hat{Q}(db)|$ may be much larger than the size of the returned sample. We are interested in developing practical alternate algorithms of lower complexity.

In what follows, we analyze our algorithms in the RAM model of computation with the unit cost model. We assume that that drawing random values from the uniform distribution takes constant time; that building hash tables is in linear time; and that probing a hash tables takes constant time. We consider the query itself fixed, and hence focus on data complexity.

Discussion. While we restrict y to be a single attribute in (1), our approach naturally extends to the setting where the sampling probability is computed from a number of attributes $\overline{y}$ that all belong to the same relation R_i. Also note that we allow the relation symbols R_i to be equal. Hence, our approach also applies to self-joins.

Our approach also applies to queries with so-called free-connex projection. We denote bag-based projection by π_A and set-based projection by $\delta\pi_A$, where δ is the duplicate elimination operator. Specifically, consider queries of the modified form

$$Q = \beta_y\left(\pi_A\left(\hat{Q}\right)\right) \quad \text{or} \quad Q = \beta_y\left(\delta\pi_A\left(\hat{Q}\right)\right), \tag{2}$$

where $\hat{Q} = R_1(\overline{x}_1) \bowtie \cdots \bowtie R_l(\overline{x}_l)$, $A \subseteq \bigcup_{i\in[l]} \overline{x}_i$, and $y \in A$. In Section 5, we show that our approach and complexity results extend to such queries when $\pi_A(\hat{Q})$ is *free-connex acyclic*, which is defined as follows. Let Q' be the full join query obtained by extending $\hat{Q}$ with an additional atom having A as its variables. Then $\pi_A(\hat{Q})$ is *free-connex acyclic* if both $\hat{Q}$ and Q' are acyclic.

3 Solution Overview

Let $\hat{Q}$ be a join query. A *random-access* index for $\hat{Q}$ on input db is a data structure idx that is equipped with a method GET. The index is free to fix an order on the tuples in $\hat{Q}(db)$. Given positions $pos = [i_1, \ldots, i_k]$ with $0 \le i_j < |Q(db)|$ for every j, $idx.\text{GET}(pos)$ returns the bag $\{\!\{t_{i_1}, \ldots, t_{i_k}\}\!\}$, where t_{i_j} is the $(i_j + 1)$-th tuple of $\hat{Q}(db)$ in the chosen order. The *access time* refers to the time required to construct this result, given pos.

To efficiently process a Poisson sampling query Q on database db we will adopt the following *Index-and-Probe* (I&P) strategy:

(1) Compute random-access index idx of $\hat{Q}$ on db.
(2) Determine $pos = [i_1, \ldots, i_k]$, the sequence of tuple positions in $Q(db)$ that define the output sample.
(3) Return $idx.\textsc{get}(pos)$.

The sequence pos computed in step (2) is called the *probe sequence*. Determining this sequence is called *position sampling*.

Clearly, the running time of I&P is determined by (1) the index construction time; (2) the probe sequence construction time; and (3) the random access time.

We base index construction on the approach recently taken by Bekkers et al. [3] for processing acyclic joins. Concretely, Bekkers et al. show that acyclic joins can be expressed in the *Nested Semijoin Algebra* (NSA for short) as a sequence of *nested* semijoin operations, followed by a single flatten operation. Logically, nested semijoin operations produce *nested relations*, i.e., relations where a tuples' attribute value may contain an entire relation instead of a scalar value as is typically the case. The flatten operator turns a nested relation back into an ordinary flat relation. While nested semijoin and flatten are logical operators, Bekkers et al. propose a specific physical representation of nested relations in main-memory column stores, based on so-called query shredding [3, 11]. The implementation of nested semijoin and flatten based on this representation always evaluates a sequence of nested semijoins followed by a flatten in $O(|db| + |\hat{Q}(db)|)$ time, recovering Yannakakis' seminal result that acyclic joins can be evaluated instance-optimally. What is more, this way of implementing Yannakakis is competitive with classical binary joins even when the input relations have no or only few dangling tuples—when traditional Yannakakis implementations are observably slower [3].

In Section 4, we will show that the shredded representation of Bekkers et al., which we will refer to as the *chained shredded representation* (CSR) already provides a random-access index for $\hat{Q}$. In other words, we can see nested semijoins as a logical operator that physically constructs a random-access index structure. In Section 5, we will further show how this representation can be used for efficient position sampling.

Previously, Carmeli et al. [7] have shown that acyclic joins admit a random-access index with single-tuple access time $O(\log |db|)$. As we will see, the access time of the CSR index is asymptotically more expensive, and therefore not optimal. We therefore also consider in Section 4 a second representation, called the *unchained shredded representation* (USR) that engineers the idea of the random-access index structure of Carmeli et al [7] in column stores using the shredded Yannakakis framework.

Before turning to describe these index structures, however, we introduce the nested semijoin algebra of [3], focusing on the nested semijoin ($\ltimes_\nu$) and flatten (μ^*) operators. We will use doubly curly braces $\{\!\{\ldots\}\!\}$ to denote bags as well as bag comprehension.

Schemes and Nested Relations. Just like flat relations have flat schemes, nested relations have nested schemes. We refer to the attributes that appear in the scheme of classical flat relations as *flat* attributes. A *(nested) scheme* is a finite set X, like $\{x, \{y\}, \{u, \{v\}\}\}$, that consists of flat attributes (x in this case) and other schemes (i.e., $\{y\}$ and $\{u, \{v\}\}$). No flat attribute is allowed to occur more than once, so $\{x, \{y\}, \{u, \{x\}\}\}$ is not a valid scheme. Schemes are also called *nested* attributes. We range over flat attributes by lowercase letters ($x, y, \ldots$); over schemes by uppercase letters (X, Y, $\ldots$), both from the end of the alphabet; and over finite sets of flat attributes by $\overline{x}$. We write $attr(X)$ for the set of all flat attributes occurring somewhere in X (either directly or in some inner nested scheme); and $sch(X)$ for the set of all schemes occurring in X, including X itself (again, either directly or in some inner nested scheme). So for $X = \{x, \{y\}, \{u, \{v\}\}\}$, $attr(X) = \{x, y, u, v\}$ and $sch(X) = \{\{y\}, \{u, \{v\}\}, \{v\}, X\}$.

Nested relations and nested tuples are defined mutually recursively as follows. A *relation* over a scheme X is a finite bag of tuples over X. Here, a *tuple over* X is a mapping t on X such that $t(x)$ is a scalar data value for each flat attribute $x \in X$, and $t(Y)$ is a non-empty relation over Y for each nested attribute $Y \in X$. Note that if X is *flat*, i.e., if X consists of flat attributes only, then this definition of a relation over X coincides with the usual one. We call R a *flat relation* in that case. We restrict inner nested relations to be non-empty as in this paper we always start from flat relations and the operators that we consider will never introduce empty inner nested relations. We write $R\colon X$ and $t\colon X$ to denote that R is a relation (resp. t is a tuple) over scheme X. We write $|R|$ to denote the total number of tuples in R. Note that this only refers to the number of tuples in the outer-most bag of R, and does not say anything about the cardinality of the inner-nested relations appearing in those tuples. Figure 2b shows a nested relation with cardinality 4 and scheme $\{x, y, p, \{u, a\}\}$.

We adopt the following notation on tuples. If $s\colon X$ and $t\colon Y$ are tuples over disjoint schemes then $s \uplus t$ denotes their concatenation, which is a tuple over $X \cup Y$. Furthermore, if $Z \subseteq X$ then $s[Z]$ denotes the restriction (i.e., projection) of s to the attributes in Z. The standard relational operators π and σ are extended to nested relations in the obvious manner. For example, $\pi_Z(R) = \{\!\{t[Z] \mid t \in R\}\!\}$ and $\sigma_{\overline{y}=s}(R) = \{\!\{t \in R \mid t[\overline{y}] = s\}\!\}$.

Nested Semijoins and Flatten. The *nested semijoin* operator $\ltimes_\nu$ takes two arguments, $R\colon X$ and $S\colon Y$.[2] For a valid application of the nested semijoin operator, it is required that $X \cap Y$ contains only flat attributes, and that $X \cup Y$ is again a scheme. Hence, $X = \{x, y, \{v\}\}$ and $Y = \{y, \{u, w\}\}$ is allowed. But $X = \{x, y, \{v\}\}$ and $Y = \{y, \{x, u\}\}$ is not since $X \cup Y$ contains x multiple times.

Let $\overline{z} = X \cap Y$ be the flat attributes shared between X and Y, and let $Z = Y \setminus X$. Observe that for $\overline{z}$-tuple t, $\sigma_{\overline{z}=t}(S)$ is the subbag of S containing those tuples having t as join key. Then $R \ltimes_\nu S$ extends the tuples in R with one nested attribute, Z, containing the non-empty subbag of tuples in S that join:

$$R \ltimes_\nu S \stackrel{\text{def}}{=} \{\!\{r \uplus \{Z \mapsto \pi_Z(\sigma_{\overline{z}=r[\overline{z}]}(S)\} \mid r \in R, \sigma_{\overline{z}=r[\overline{z}]}(S) \neq \emptyset\}\!\}. \tag{3}$$

To illustrate, Figure 2b shows the result of $R \ltimes_\nu S$ on the relations R and S shown in Figure 2a where $\overline{z} = \{x\}$ and $Z = \{u, a\}$.

The flatten operator μ^* converts a nested relation $N\colon X$ or nested tuple $t\colon X$ into a flat relation with scheme $attr(X)$. Specifically, assume that X consists of flat attributes $\overline{x}$ and nested attributes $Y_1, \ldots, Y_m$. Then

$$\mu^*(t) \stackrel{\text{def}}{=} \{\!\{t[\overline{x}] \uplus t_1 \uplus \cdots \uplus t_m \mid t_j \in \mu^*\left(t(Y_j)\right), j \in [m]\}\!\}; \tag{4}$$

$$\mu^*(N) \stackrel{\text{def}}{=} \bigcup \{\!\{\mu^*(t) \mid t \in N\}\!\}. \tag{5}$$

That is, $\mu^*(t)$ pairs the flat attributes of t with all combinations of tuples obtained by flattening the nested relations $t(Y_1), \ldots, t(Y_m)$ and $\mu^*(N)$ is the bag union of flattening all of its tuples. Note that when t and N are flat, i.e, $m = 0$ then $\mu^*(t) = \{\!\{t\}\!\}$ and $\mu^*(N) = N$.

To illustrate, reconsider Figure 2a. When we flatten the nested relation in Figure 2b we obtain the same relation as the one that results from $R \bowtie S$. When we flatten the nested relation in Figure 2c we obtain $R \bowtie S \bowtie T$.

Weights. The *weight* of a nested relation R (resp. tuple r) is the total number of tuples produced when flattening R (resp r), i.e. $weight(R) = |\mu^*(R)|$ (resp. $weight(r) = |\mu^*(r)|$).

NSA. While Bekkers et al. also consider additional operators on nested relations, for the purpose of this paper we define the Nested Semijoin Algebra (NSA) to be all expressions that are built from flat relation symbols using valid application of $\ltimes_\nu$ and μ^*. Such an expression is called *2-phase*

[2] In [3], the nested semijoin operator is defined as the composition of two other operators, $\triangleright\!\!\!\triangleright$ and γ; here we combined them into the single operator $\ltimes_\nu$ for convenience.

	R			S			T	
	x	y	p	u	a	x	v	y
0	x_1	y_1	p_1	u_1	a_1	x_1	v_1	y_4
1	x_1	y_2	p_2	u_1	a_1	x_2	v_2	y_2
2	x_4	y_3	p_3	u_2	a_1	x_1	v_3	y_1
3	x_2	y_1	p_4	u_3	a_2	x_1	v_4	y_2
4	x_2	y_2	p_5	u_3	a_2	x_3	v_5	y_1
5	x_4	y_3	p_6	u_4	a_3	x_2	v_6	y_2

(a) Input relations

x	y	p	$\{u, a\}$
x_1	y_1	p_1	$u_1\ a_1$ $u_2\ a_1$ $u_3\ a_2$
x_1	y_2	p_2	$u_1\ a_1$ $u_2\ a_1$ $u_3\ a_2$
x_2	y_1	p_4	$u_1\ a_1$ $u_4\ a_3$
x_2	y_2	p_5	$u_1\ a_1$ $u_4\ a_3$

(b) N_1

x	y	p	$\{u, a\}$	$\{v\}$
x_1	y_1	p_1	$u_1\ a_1$ $u_2\ a_1$ $u_3\ a_2$	v_3 v_5
x_1	y_2	p_2	$u_1\ a_1$ $u_2\ a_1$ $u_3\ a_2$	v_2 v_4 v_6
x_2	y_1	p_4	$u_1\ a_1$ $u_4\ a_3$	v_3 v_5
x_2	y_2	p_5	$u_1\ a_1$ $u_4\ a_3$	v_2 v_4 v_6

(c) N_2

$\Sigma(\{x, y, p, \{u, a\}, \{v\}\})$							$\Sigma(\{u, a\})$			$\Sigma(\{v\})$		
x	y	p	$\{u, a\}$		$\{v\}$		pref	u	a	nxt	v	nxt
			hd	w	hd	w						
x_1	y_1	p_1	3	3	4	2	6	0 u_1	a_1	-1	0 v_1	-1
x_1	y_2	p_2	3	3	5	3	15	1 u_1	a_1	-1	1 v_2	-1
x_2	y_1	p_4	5	2	4	2	19	2 u_2	a_1	0	2 v_3	-1
x_2	y_2	p_5	5	2	5	3	25	3 u_3	a_2	2	3 v_4	1
								4 u_3	a_2	-1	4 v_5	2
								5 u_4	a_3	1	5 v_6	3

(d) Chained shredded representation of N_2

$\Sigma(\{x, y, p, \{u, a\}, \{v\}\})$									$\Sigma(\{u, a\})$				$\Sigma(\{v\})$			
x	y	p	$\{u, a\}$			$\{v\}$			pref	u	a	perm	pref	v	perm	pref
			start	len	w	start	len	w								
x_1	y_1	p_1	0	3	3	0	2	2	6	0 u_1	a_1	0	1	0 v_1	2	1
x_1	y_2	p_2	0	3	3	2	3	3	15	1 u_1	a_1	2	2	1 v_2	4	2
x_2	y_1	p_4	3	2	2	0	2	2	19	2 u_2	a_1	3	3	2 v_3	1	1
x_2	y_2	p_5	3	2	2	2	3	3	25	3 u_3	a_2	1	1	3 v_4	3	2
										4 u_3	a_2	5	2	4 v_5	5	3
										5 u_4	a_3	4	1	5 v_6	0	1

(e) Unchained shredded representation of N_2

Fig. 2. Illustration of nested relations and nested semijoins with $N_2 = (R(x, y, p) \ltimes_\nu S(u, a, x)) \ltimes_\nu T(v, y)$ and $N_1 = R(x, y, p) \ltimes_\nu S(u, a, x)$.

(2NSA) if it is of the form $\mu^*(E)$ with E itself not containing any further application of μ^*. We call E *μ^*-free* in this case. Bekkers et al. observe that from a join tree J for an acyclic join query $\hat{Q}$ we may compute a 2NSA expression $\mu^*(E)$ as follows. First, construct the μ^*-free expression E by traversing J bottom up. For leaf nodes, E is simply the node's atom. For interior nodes, E is the result $(\ldots (R(\overline{x}) \ltimes_\nu E_1) \ltimes_\nu \ldots) \ltimes_\nu E_n$ of taking the semijoin of node's atom $R(\overline{x})$ with the expressions recursively obtained for each of the children. Finally, add μ^* to the expression obtained for the root. For later use we strengthen this observation as follows.

Proposition 3.1. *For every acyclic join query $\hat{Q}$ and any attribute y of $\hat{Q}$ we can compute an equivalent two-phase NSA expression $\mu^*(E)$ such that y is a flat attribute in the output scheme of E.*

Proof. First, compute a join tree J of $\hat{Q}$. Pick any node in J that mentions y, and reroot J into a join tree J' that has this node as the root. Then the above-mentioned procedure applied to J' yields the desired 2NSA expression. □

4 Shredded Random-Access Indexing

Columnar storage. We assume that we are working in main memory, and that a flat relation $R(x_1, \ldots, x_n)$ is physically represented as a tuple $\mathsf{R} = (\mathsf{R}.x_1, \ldots, \mathsf{R}.x_n)$ of vectors $\mathsf{R}.x_i$, all of length $|R|$. It is understood that values at the same offset in these vectors encode a complete tuple, i.e., $R = \{\!\{(\mathsf{R}.x_1[i], \ldots, \mathsf{R}.x_n[i]) \mid 0 \leq i < |R|\}\!\}$. In particular, it is possible to refer to tuples positionally, i.e., the tuple at offset 0 in R, the tuple at offset 1, and so on. We will refer to R as a *physical relation*, and denote the number of tuples in R by $|\mathsf{R}|$. If $0 \leq i < |\mathsf{R}|$ and $\overline{y} = y_1, \ldots, y_k$ is a subset of $\{x_1, \ldots, x_n\}$, then we write $\mathsf{R}[i](\overline{y})$ for the tuple $(\mathsf{R}.y_1[i], \ldots, \mathsf{R}.y_k[i])$. A *position vector for* R is a vector of natural numbers, all between 0 and $|\mathsf{R}| - 1$.

Shredded representations. *Shredding* [9, 11, 32] refers to the representation of a nested relation by means of a collection of flat relations. In our setting, the latter will be physical relations, as the ability to refer to the individual columns and individual positions of tuples is important for the efficiency of processing the representation. Since we will consider two distinct shredded representations, we first define the concept generically, and then introduce the concrete representations. Generically, a *shredded representation* of a nested relation $R\colon X$ is a collection Σ of physical (flat) relations, one physical relation $\Sigma(Y)$ for every $Y \in sch(X)$. Each $\Sigma(Y)$ contains columns for at least all flat attributes of Y, but also has additional columns to encode the hierarchical relationships of the nested tuples in R. We note that when X is a flat scheme, $sch(X) = \{X\}$, and Σ hence consists then only of a single flat table, namely $\Sigma(X)$, which is a physical representation of R. Define the *size* $|\Sigma|$ of a shredded representation to be the total number of tuples that it contains, in all of its flat relations.

4.1 Chained Shredding

We next introduce the shredding representation of Bekkers et al [3], referred to as the *chained shredded representation* (CSR). Let $N\colon X$ be a nested relation. A CSR Σ of N has one physical relation $\Sigma(Y)$ for every $Y \in sch(X)$. This relation contains one column for each flat attribute in Y. Moreover, for every nested attribute $Z \in Y$, $\Sigma(Y)$ has two extra columns hd_Z and w_Z, both of which hold position vectors. Finally, if Y is a strict subscheme of X, i.e. $Y \neq X$, then $\Sigma(Y)$ also has a column nxt. To illustrate, Figure 2d shows a CSR for the nested relation in Figure 2c.

In a CSR, the nxt column is used to encode a linked list of tuples: for all offsets $0 \leq i < |\Sigma(Y)|$, if $\Sigma(Y).\mathsf{nxt}[i] < 0$ then the tuple at offset i in $\Sigma(Y)$ is the final tuple in the list; otherwise its successor in the list is the tuple at offset $\Sigma(Y).\mathsf{nxt}[i]$. Hence, nxt is used to chain tuples together, see Figure 2d.

Chained shredding works as follows: every tuple $t \in N$ is represented by exactly one tuple in $\Sigma(X)$. Let i be the index of the tuple in $\Sigma(X)$ representing t. For every flat $x \in X$ we have $t(x) = \mathsf{R}.x[i]$. For every nested attribute $Y \in X$, $\Sigma(X).\mathsf{w}_Y[i]$ stores the weight of $t(Y)$. Furthermore, $\Sigma(X).\mathsf{hd}_Y[i]$ stores the head index of the linked list of tuples in $\Sigma(Y)$ that together represent the tuples occurring in $t(Y)$. Note that the tuples in $t(Y)$ may themselves contain further nested relations, and the shredding hence proceeds recursively.

Example 4.1. To illustrate, Figure 2d shows a CSR for the nested relation $N_2 : X$ of Figure 2c. Here, $X = \{x, y, p, \{u, a\}, \{v\}\}$. The i-th tuple in N_2 is represented by the i-th tuple in $\Sigma(X)$. When we look at the first tuple of N_2 we see that $t(\{v\}) = \{\!\{v_3, v_5\}\!\}$. In Σ, this bag is encoded by the linked list in $\Sigma(\{v\}).\mathsf{nxt}$ starting at offset $\mathsf{hd} = 4$.

It is important to note that multiple tuples in Σ may repeat the same hd_Y value; this effectively means that these tuples share the same nested relation in the Y-value. See the $\{u, a\}$-value of the first and second tuple of Figure 2c for an illustration. This sharing is important to ensure that,

```
def CSR-GROUP(Σ_S):
  nxt = [0, ..., 0] # |Σ_S(X_S)| times
  h = {} # maps keys -> (pos, weight)
  for 0 ≤ i < |Σ_S(X_S)| :
    key = Σ_S(X_S)[i](z̄); w = weight_of(Σ_S, X_S, i)
    if h.contains(key) :
      (j, prev_w) = h[key]
      nxt[i] = j
      h[key] = (i, prev_w + w)
    else
      h[key] = (i, w)
  return (h, nxt)
```

Fig. 3. CSR grouping algorithm.

starting from a number of flat relations we can build a nested relation resulting from their semijoin whose physical representation size is linear in the input size.

In what follows, we assume a method `weight_of` on CSRs that, given a CSR Σ and one if its schemes Y and an offset $0 \leq i < |\Sigma(Y)|$, returns the weight of the nested tuple represented at offset i in $\Sigma(Y)$. This is straightforward to compute: when Y is flat, the weight is always one; otherwise it is the product of the weights of the nested attributes.

Building the representation. Bekkers et al [3] show that, given CSRs Σ_R and Σ_S for nested relations $R\colon X_R$ and $S\colon X_S$, respectively, we can construct a CSR Σ for $R \ltimes_\nu S$ in time linear in the size of Σ_R and Σ_S as follows. The process is akin to a hash join. Let $\overline{z} = X_R \cap X_S$ be the join attributes and $Z = X_S \setminus X_R$.

(1) First, group S on the join attributes $\overline{z}$ by computing a pair (h, nxt) consisting of (i) a hash table h that maps the join keys occurring in S to pairs of the form (i, w), and (ii) a position vector nxt, encoding a collection of linked lists. For every join key k, if we start traversing nxt in a linked-list fashion starting at offset i, then we traverse the offsets of all tuples in S that have join key k. Moreover, w is the sum of weights of these tuples. Figure 3 shows the pseudo-code to compute (h, nxt) in a single pass over Σ_S.
(2) Second, drop all join columns from $\Sigma_S(X_S)$, and add the nxt vector computed in the previous step as an additional column.
(3) Use $\Sigma_R(X_R)$ to obtain $\Sigma(X_R \cup \{Z\})$ by looping over the tuples of $\Sigma_R(X_R)$, probing the hash table h to see if a joining tuple in $\Sigma_S(Y)$ exists and if so, use the (i, w) value stored in h to populate the hd_Z and w_Z-values. Tuples without joining tuples in $\Sigma_S(Y)$ are dropped.
(4) Now, Σ_R, Σ_S and $\Sigma(X_R \cup \{Z\})$ together form the CSR Σ for $R \ltimes_\nu S$.

The interested reader is invited to apply this procedure on the input relations of Figure 2a to obtain CSR for first $R \ltimes_\nu S$ and then $(R \ltimes_\nu S) \ltimes_\nu T$. Initially, Σ_R consists only of R itself, and similarly for Σ_S and Σ_T.

Random access. Given CSR Σ for nested relation N, we can turn Σ into a random-access index for $\mu^*(N)$. That is, given an offset $0 \leq i < |\mu^*(N)|$, we can extract the $(i+1)$-th tuple of $\mu^*(N)$ directly from Σ (i.e., directly from the representation of N) without first needing to materialize $\mu^*(N)$.

Conceptually, we order the tuples of $\mu^*(N)$ as follows: the tuple t of N that occurs first in Σ produces the first $\mathit{weight}(t)$ tuples of $\mu^*(N)$, the tuple t that occurs second in Σ produces the next $\mathit{weight}(t)$ tuples, and so on. Here, each tuple t produces $\mathit{weight}(t)$ flat tuples, which cf. Equation (4), are obtained by combining the flat attribute values of t with the result of recursively flattening the nested attributes $Y_1, \ldots, Y_m$ of X. We order the tuples of $\mu^*(t)$ itself as follows. Let $0 \leq i < \mathit{weight}(t)$.

```
1: def CSR-GET(Σ, X, i):
2:   result = {} # result tuple, initially empty
3:   find smallest j in 0..|Σ(X)| s.t. i < pref[j]
4:   CSR-SUB(Σ,X, j, i − pref[j − 1]) # assume pref[-1] = 0
5:   return result
6: def CSR-SUB(Σ, X, j, i):
7:   result = result⊎{a ↦ Σ(X).a[j] | a ∈ X}
8:   for each nested attr Y ∈ X :
9:     (j′, w) = Σ(X)[j](hd_Y, w_Y)
10:    i′ = i mod w; i = i div w
11:    curr_weight = weight_of(Σ, Y, j′)
12:    while j′ ≥ 0 and i′ ≥ curr_weight :
13:      i′ = i′ − curr_weight
14:      j′ = Σ(Y).nxt[j′]
15:      curr_weight = weight_of(Σ, Y, j′)
16:    CSR-SUB(Σ, Y, j′, i′)
```

Fig. 4. CSR random access.

To be able to unambiguously identify which combination of the tuples in $\mu^*\left(t(Y_j)\right)$ produces the tuple of $\mu^*(t)$ at offset i, we view i as a number in a mixed-radix numeral system where the bases are the weights of the $t(Y_j)$. That is, let w_j abbreviate $weight(t(Y_j))$ and let $(i_1, \ldots, i_m)$ be offsets of tuples in $(\mu^*(t(Y_1)), \ldots, \mu^*(t(Y_m)))$, respectively. These tuples together will produce the tuple of $\mu^*(t)$ at offset

$$i = i_1 + i_2 w_1 + i_3 w_1 w_2 + \cdots + i_m w_1 \ldots w_{m-1}$$

We note that, given i it is straightforward to compute the offsets i_j inductively as follows

$$i_1 = i \bmod w_1 \qquad q_1 = i \operatorname{div} w_1 \tag{6}$$

$$i_j = (q_{j-1} \bmod w_j) \qquad q_j = (q_{j-1} \operatorname{div} w_j) \qquad j > 1 \tag{7}$$

Then the tuple of $\mu^*(t)$ at offset i is obtained by combining the i_j-th tuple of $\mu^*(t(Y_j))$ for every $j \in [m]$.

Example 4.2. Consider Figure 2c, which shows a nested relation with scheme $X = \{x, y, p, Y_1, Y_2\}$ where $Y_1 = \{u, a\}$ and $Y_2 = \{v\}$. Let t be its first nested tuple, which will produce $3 \times 2 = 6$ tuples when flattened. To identify the tuple that will be produced at offset $i = 4$ (i.e., the 5-th tuple), we compute

$$i_1 = 4 \bmod 3 = 1 \quad q_1 = 4 \operatorname{div} 3 = 1 \quad i_2 = (1 \bmod 2) = 1$$

Since Y_1 and Y_2 are flat schemes, $\mu^*(t(Y_1)) = t(Y_1)$ and $\mu^*(t(Y_2)) = t(Y_2)$. As such, i_1 and i_2 give offsets directly in $t(Y_1)$ and $t(Y_2)$, which are represented at $\Sigma(Y_1)$ and $\Sigma(Y_2)$ respectively. In other words, the 5-th tuple obtained while flattening t has $u = u_2$, $a = a_1$, $v = v_5$.

Given this order, we turn Σ into a random-access index by first adding one extra column to $\Sigma(X)$: the *prefix vector* pref. This vector contains, for every offset $0 \le i < |N|$, the sum of the weights of the nested tuples up and including offset i. To illustrate, Figure 2d shows this prefix vector in blue. The prefix vector can clearly be computed in linear time.

Given an offset i into $\mu^*(N)$, we can access the tuple at offset i directly from the CSR Σ of N as shown in Figure 4:

(1) Allocate space to store the result tuple.
(2) Use binary search to locate the smallest index $0 \le j < |\Sigma(X)|$ for which $i < \text{pref}[j]$. This takes time $O(\log |\Sigma|)$ and identifies the nested tuple that produces the tuple at offset i when flattened.

(3) Call CSR-SUB to focus on the nested tuple t represented at offset j in $\Sigma(X)$, and to construct from t the tuple that is produced at offset $(i - \mathsf{pref}[j-1])$ when flattening t.
(4) CSR-SUB does this by first copying the values of all flat attributes, which are at offset j in $\Sigma(X)$. It remains to get the correct values for the nested attributes. This is done by iterating through all nested attributes Y. For each Y, we compute the offset i' of the tuple that needs to be retrieved from $\mu^*(t(Y))$ by means of the mixed-radix indexing scheme (equation (6)). Then, starting at $\Sigma(Y).\mathsf{hd}_Y[j]$, we iterate through Σ_Y.nxt's linked list of tuples, keeping track of how many tuples these would produce when flattened. As soon as this number exceeds i' we have found the correct position, and we call CSR-SUB recursively to populate the result values for $attr(Y)$.

CSR-SUB will be called h times, where $h = |\, sch(X)|$ is the number of subschemes of X. Each call it linearly traverses a linked list. The overall complexity is $O(\log|\Sigma(X)| + h \times d)$ where d is the maximum length of any of the linked lists that we need to traverse. When Σ_N was constructed using nested semijoins starting from flat relations, then it is not difficult to see that $|\Sigma(X)| \leq |db|$ and that d can be at most the degree of any join key in db, which is formally defined as follows.

Definition 4.3. Assume that $R\colon \overline{x}$ is a flat relation, let $\overline{y} \subseteq \overline{x}$ and let t be a $\overline{y}$-tuple. The degree of t in R is the number of times that t occurs in $\pi_y(R)$. In other words, it is the cardinality of $\sigma_{\overline{y}=t}(R)$. The degree of $\overline{y}$ in R, denoted $deg_{\overline{y}}(R)$ is the maximum degree of any y-tuple in R. The *degree* of a join query $\hat{Q} = R_1(\overline{x}_1) \bowtie \cdots \bowtie R_k(\overline{x}_k)$ in a database db is defined as

$$deg_{\hat{Q}}(db) \stackrel{\text{def}}{=} \max\{deg_{\overline{x}_i \cap \overline{x}_j}(R_i) \mid i \neq j, \overline{x}_i \cap \overline{x}_j \neq \emptyset\}$$

We may hence conclude:

Proposition 4.4. *Using nested semijoins it is possible to construct in $O(|db|)$ time a CSR that can be used as a random-access index for an acyclic join query $\hat{Q}$, with access time $O(\log|db| + deg_{\hat{Q}}(db))$.*

PROOF. Since $\hat{Q}$ is acyclic, we can compute $\hat{Q}(db)$ by means of 2NSA expression $\mu^*(E)$, where E consists of nested semijoins only. By our earlier reasoning we can build a CSR for $N = E(db)$ in time linear in db. We can then compute the prefix vector also in linear time. This CSR + prefix vector allows random access into $\mu^*(N)$ with access time $O(\log|N| + h \times deg_Q(db))$. The result follows by observing that necessarily $|N| = O(|db|)$ and that h is constant in data complexity. □

Caching optimization. Above, we focused on random access for a single tuple offset i. When given a sequence $pos = [i_1, \ldots, i_k]$ of such positions to access randomly in bulk, we can apply the following *caching optimization*. During random access to the output tuple at position i_ℓ, we traverse the linked list starting at $\Sigma(Y).\mathsf{nxt}[j']$. We cache the position (i.e., a pointer) at which this traversal terminates. If the subsequent random access to the output tuple at position $i_{\ell+1}$ requires traversing the same linked list and the desired item appears further along that list, we resume the traversal from the cached pointer rather than restarting from the head of the list. This strategy avoids unnecessary repeated traversals from the beginning of the linked list. Pseudocode is given in the full paper version [4].

4.2 Unchained Shredding

The *unchained shredded representation* (USR) of a nested relation $N\colon X$ differs from CSR in that it stores the offsets, and the prefix vector, of tuples in the same nested attribute value consecutively rather than chaining them together. This allows binary rather than linear search during random access, at the expense of more complicated index construction.

To that end, a USR uses the following extra columns. Every $\Sigma(Y)$ has a column pref. Additionally, for every nested subscheme $Z \in Y$, $\Sigma(Y)$ has three extra columns, start_Z, len_Z, and w_Z. Finally, if

$Y \neq X$ then Σ_Y also has a column perm. To illustrate, Figure 2e shows a USR for the nested relation in Fig 2c.

Unchained shredding works as follows: every tuple $t \in N$ is represented by exactly one tuple in $\Sigma(X)$. Let i be the representations' offset. Then $t(x) = \Sigma(X).x[i]$ for every flat $x \in X$. For every nested attribute $Y \in X$ we have that $\mathit{weight}(t(Y)) = \Sigma(X).\mathrm{w}_Y[i]$. Moreover, if $j = \Sigma(X).\mathrm{start}_Y[i]$ and $l = \Sigma(X).\mathrm{len}_Y[i]$, then $\Sigma(Y).\mathrm{perm}[j : l]$ contains the offsets of all the tuples in $\Sigma(Y)$ that together represent the tuples occurring in $t(Y)$. Here, $\mathrm{perm}[j : l]$ denotes the slice of perm starting at j and ending at $j + l - 1$. Finally, $\Sigma(Y).\mathrm{pref}[j : l]$ is the prefix sum of the weights of those tuples, see Figure 2e. For X itself, $\Sigma_R(X).\mathrm{pref}$ contains the prefix sum of all tuples.

Discussion. A USR is an implementation, in the shredded framework, of the random-access index proposed in [7], with the following difference. A USR uses the slice columns $\mathrm{start}_Z, \mathrm{len}_Z$ to index into a permutation vector perm to encode the offsets of elements of inner nested attributes Z. By contrast, Carmeli et al. leave open how these elements are to be represented exactly, suggesting implementation wise to use an additional index data structure on Z and probing this index during random access. This is wasteful since when building the USR we already have the correct positional information, avoiding extra index space and associated costs.

Building the representation. Similar to the chained case, a USR representing $R \ltimes_\nu S$ can be built in linear time given USR Σ_R and Σ_S for $R\colon X_R$ and $S\colon X_S$, respectively. The procedure is entirely similar to the building of a *CSR* for $R\colon X_R$ and $S\colon X_S$ described earlier, but differs in the algorithm used to group S by the join key. We hence focus next only on the grouping aspect.

Specifically, to group S on the join attributes $\overline{z}$, we now construct a triple $(h, \mathrm{perm}, \mathrm{pref})$ consisting of (i) a hash table h that maps the join keys occurring in S to triples of the form (i, l, w), (ii) a position vector perm, and (iii) a prefix vector pref. For every join key k, $\mathrm{perm}[i : l]$ contains the positions of all tuples having join key k, and $\mathrm{pref}[i : l]$ is the prefix sum of their weights, in the order that the tuples are specified in $\mathrm{perm}[i : l]$. The triple $(h, \mathrm{perm}, \mathrm{pref})$ is then used in the probing phase to construct the necessary columns of the USR representation.

Compared to Algorithm 3, which describes the CSR grouping, USR grouping needs to make two hashing passes over S to construct $(h, \mathrm{perm}, \mathrm{pref})$. In the first pass we count, per join key k, the number of tuples l with that join key. Using this information, we can determine for each join key the pair (i, l) so that the positions of all tuples with join key k can be stored in the slice $\mathrm{perm}[i : l]$. A second hashing pass through the data then actually stores the positions of the tuples with join key k in this location, and computes the prefix vector.

Given that USR requires an extra hashing pass, USR building is expected to be slower than CSR building even though they share the same asymptotic complexity. We return to this in Section 6.3.

Random access. The fact that the positions and prefix vector are stored consecutively per join key aids in random access complexity. The USR random access procedure, shown in Algorithm 5 is similar to the CSR access procedure. However, the fact that each nested attribute (and not only the top-most scheme X) has a prefix vector allows us to perform binary search at every level, obviating the need to iterate linearly through the nested sets. The overall complexity is therefore $O(\log|\Sigma(X)| + h \times \log(d))$ where d is the maximum cardinality of any nested set and $h = |\mathit{sch}(X)|$ is the number of times that USR-SUB is called.

Proposition 4.5. *Using nested semijoins it is possible to construct in $O(|db|)$ time a USR that can be used as a random-access index for acyclic join query $\hat{Q}$, with access time $O(\log|db|)$.*

Optimization. The same optimization used for CSR *bulk* random access can also be applied for USR. Specifically, for every nested attribute Y, we cache the result of the binary search so that, if

```
1: def USR-GET(Σ, i):
2:   result = {} # result tuple, initially empty
3:   perm = [1, 2, ..., |Σ(X)|]
4:   USR-SUB(Σ, X, i, Σ(X).pref, perm)
5:   return result
6: def USR-SUB(Σ, X, i,pref, perm):
7:   find smallest j in 0..|pref| s.t. i < pref[j]
8:   i = i − pref[j − 1] # assume pref[-1] = 0
9:   j = perm[j]
10:  result = result ⊎{a ↦ Σ(X).a[j] | a ∈ X}
11:  for each nested attr Y ∈ X :
12:    w = weight_of(Σ,Y, j); i′ = i mod w; i = i div w
13:    (s, l) = Σ(X)[j](start_Y, len_Y)
14:    USR-SUB(Σ,Y, i′, Σ(Y).pref[s : l], Σ(Y).perm[s : l])
```

Fig. 5. USR Access

the subsequent binary search for Y is performed over the same search space, the search can resume from the point where the previous one has ended. Pseudocode is given in the full paper version [4].

5 Position sampling

We next describe how to construct $pos = [i_1, \ldots, i_k]$, the sequence of tuple offsets that defines the output sample, and which we use to probe to the random-access index. To that end, we assume given a nested relation N and a shredded random-access index Σ for N. Throughout the section, let $n = |\mu^*(N)|$. Note that we can obtain n from Σ in constant time as it is the last element of the pref vector of $\Sigma(X)$ in both CSR and USR. To get insight into the position sampling process, we first discuss how to draw *pos* in the simplified setting when all tuples have the same probability. Using this insight, we turn to the general setting where tuples have non-uniform probabilities.

Uniform. Assume that all tuples in $\mu^*(N)$ have the same fixed probability $p \in [0, 1]$. Then the sample space I to draw *pos* from is $I = \{0, \ldots, n-1\}$. The *uniform position sampling problem for p and n* asks to construct $pos \subseteq I$ such that each element $i \in I$ is included in *pos* with probability p. Observe that the expected size of *pos* follows a Binomial distribution with parameters n and p, and hence equals np.

The naive approach is to perform, for each $i \in I$, an independent Bernoulli trial with probability p, and to include i in *pos* whenever the trial succeeds. This method, which we call Bern, requires $\Theta(n)$ time, irrespective of p and irrespective of the produced position vector length k. Consequently, Bern is suboptimal when p is small, since it still performs a Bernoulli trial for every flat output tuple, even though the expected sample size is only np.

Alternatively we can also construct *pos* by repeatedly sampling from the geometric distribution. Figure 6 shows the pseudocode of this approach, denoted Geo. Let X be a stochastic variable that follows a geometric distribution with probability p, denoted $X \sim \text{Geometric}(p)$. The value of X represents the number of independent Bernoulli(p) trials that result in failure before the first success occurs, i.e, $Pr[X = i] = p(1-p)^i$ for $i = 0, 1, \ldots$. Function DrawGeo shows how to draw from the geometric distribution in constant time. Intuitively, we sample from the inverse of the exponential distribution and apply truncation to map the exponential distribution to the geometric distribution, see also [12]. In Geo, successive calls to DrawGeo determine the gaps between sampled positions until the running index exceeds n. The complexity of Geo is $O(k)$ with k the size of the sample; thus the expected complexity of Geo is $O(np)$. We hence expect Geo to outperform Bern for smaller sampling probabilities. However, we will experimentally show in

```
def DrawGeo(p):
  u ~ Uniform(0, 1)
  return ⌊ln(u)/ln(1 − p)⌋
def Geo(p, n):
  if p = 0 return [] :
  else if p = 1 return [0, 1, 2, . . . , n − 1] :
  else:
    result = []; i = DrawGeom(p)
    while i < n :
      result = result++[i] # append i to the vector result
      j = DrawGeom(p); i = i + 1 + j
    return result
```

Fig. 6. Geo algorithm

Section 6.1 that for larger probabilities, Geo is actually slower than Bern. For this reason, we also define the hybrid method Hybrid which uses Geo when p is smaller than a fixed threshold and Bern otherwise. We experimentally determine the threshold to be $p = 0.5$ in Section 6.1.

Finally, it is also possible to construct *pos* by first drawing k from Binomial(n, p), the binomial distribution with parameters n and p, and subsequently drawing a subset of size k from $\{0, \ldots, n-1\}$. The first step is $O(n)$ in the worst case, but has expected runtime $O(n \times \min(p, 1-p))$ [23]. The second step can be done in $O(k)$ time [7]. We denote this approach Binom in what follows.

Non-uniform. We next discuss the non-uniform case, assuming that the attribute y that contains the probability that each tuple should be sampled with is a flat attribute of N. We may do so without loss of generality by Proposition 3.1.

Every nested tuple t of N specifies in attribute y the sampling probability of all tuples in $\mu^*(t)$, of which there are $weight(t)$ in total. Hence, conceptually we can reduce the construction of *pos* in the non-uniform case to $|N|$ uniform position sampling problems—one for each $t \in N$. It suffices to iterate over the nested tuples $t \in N$, constructing a position vector pos_t for the uniform position sampling problem with probability $t(y)$ and length $weight(t)$, and concatenating these vectors—making sure to update offsets to take into account the positions sampled for previous nested tuples. Implementation-wise, we simply iterate over the tuples of $\Sigma(X)$.

In what follows, when M is one of the uniform position sampling methods, i.e., $M \in \{\text{Bern}, \text{Geo}, \text{Hybrid}, \text{Binom}\}$, then we denote by PT$M$ this non-uniform position sampling method.

Asymptotic complexity of Poisson Sampling. We close this section by establishing the asymptotic complexity of Poisson sampling over acyclic joins.

Theorem 5.1. *Poisson sampling over acyclic joins, as well as over free-connex projections of such joins, can be solved in time $O(|db| + k \log |db|)$ in data complexity, where $|db|$ is the size of the input database, and k the size of the resulting sample.*

Proof. Let $Q = \beta_y\,(R_1(\overline{x}_1) \bowtie \cdots \bowtie R_m(\overline{x}_m))$ be an acyclic Poisson sampling query. By Proposition 3.1 we can compute a 2NSA expression $\mu^*(E)$ equivalent to $\hat{Q}$, such that y is a flat attribute of E.

Given E, we can compute in $O(|db|)$ time a USR Σ of the nested relation $N \stackrel{\text{def}}{=} E(db)$, which supports single-tuple random access into $\mu^*(N) = \hat{Q}(db)$ in $O(\log |db|)$ time. Given Σ we can use PTGeo to compute a probe sequence *pos* of length k in $O(|N| + k) = O(|db| + k)$ time, and use this to produce the resulting sample in $O(k \log |db|)$ time by probing the index. Summing up the complexity of each individual step yields the claimed complexity.

For Poisson sampling queries of the form $\beta_y(\delta\pi_A(\hat{Q}))$ with $\pi_A(\hat{Q})$ free-connex acyclic we reason as follows. Carmeli et al. [7] observe that for every free-connex acyclic conjunctive query $\delta\pi_A(\hat{Q})$ and database D, one can compute in linear time a full acyclic join query Q' and a database D' such that $\delta\pi_A(\hat{Q})(D) = Q'(D')$. In other words, processing $\beta(\delta\pi_A(\hat{Q}))$ on D is equivalent to processing $\beta(Q')$ on D' and our techniques apply to the latter. Since there is only a linear time overhead to obtain the latter from the former, our complexity results transfer to sampling over free-connex queries where the projection is set-based. For bag-based projection, our complexity results trivially hold because $\beta_y(\pi_A(\hat{Q})) = \pi_A(\beta_y(\hat{Q}))$. □

6 Experimental Evaluation

Implementation. We start from the Shredded Yannakakis (SYA) implementation of [3] which is implemented in Apache DataFusion, a Rust-based in-memory columnar query engine. We extend this implementation by (1) adding the CSR access method to SYA's existing CSR implementation; (2) adding the unchained USR variant; and (3) adding the position sampling methods.

Benchmarks. We evaluate our methods on two sets of sampling queries: (1) uniform sampling queries of the form $\beta_p(\hat{Q})$ where p is a *fixed uniform* probability $p \in [0, 1]$; and (2) true non-uniform Poisson sampling queries $\beta_y(\hat{Q})$. The former allows to gain insight into the trade-offs across different fixed probabilities p; while the latter allows to confirm these insights for Poisson sampling.

The full join queries $\hat{Q}$ are drawn from the join order benchmark JOB [25] and STATS-CEB [17] which contain synthetic queries evaluated on real-world data. For Poisson sampling, we additionally add the real-world benchmark query Q_c from Example 1.1, which simulates contact patterns in infectious disease transmission scenarios and which is executed on real-world contact probability data [19, 30] modeling the population of Belgium, consisting of 1.1×10^7 individuals.

Both JOB and STATS-CEB consist of SQL queries that perform base table filters and equijoins, followed by a single aggregation. We omit the final aggregation step to obtain the full join queries, all of which are acyclic. Since Datafusion's standard binary join method will serve as one of our baselines, we remove queries for which the obtained full join fails to execute in Datafusion, leaving 112 JOB queries and 142 STATS-CEB queries.

All of these queries are used for the uniform sampling experiments. For Poisson sampling, however, we also need an attribute y that specifies the sampling probability. Because neither JOB nor STATS-CEB contain a probability attribute, we generate such an attribute as follows. Relation *Title* is central in the query graph of the JOB workload [25], and this relation also appears in all JOB queries. Therefore, we add probability attribute y to *Title*. For STATS-CEB, no single relation appears in all queries. We therefore attach probability attribute y to the relation that occurs most frequently: *Users*, present in 113 out of 142 queries. We populate attribute y in both *Title* and *Users* according to three different distributions, denoted `low`, `medium`, and `high` respectively:

- Beta($a = 2, b = 10$) with $\mathbb{E}[y] \approx 0.167$ and $\mathrm{Var}[y] \approx 0.011$
- Normal($\mu = 0.5, \sigma = 0.2$) with $\mathbb{E}[y] = 0.5$ and $\mathrm{Var}[y] = 0.04$
- Beta($a = 10, b = 2$) with $\mathbb{E}[y] \approx 0.833$ and $\mathrm{Var}[y] \approx 0.011$

Table 1 summarizes the number of queries in each benchmark as well as the distribution of full join sizes in each setup. (For Q_c the benchmark has only one query, hence the distribution has only one measurement.)

Setup. All experiments are conducted on a Ubuntu 22.04.4 LTS machine using a single thread with an Intel Core i7-11800 CPU and 32GB of RAM. All runtimes are averages of 5 different runs. We apply caching optimization, as described in Section 4, when probing a chained index because it

Benchmark	Setup	#Queries	Full join output size			
			min	avg	median	max
JOB	uniform	112	0	1.39×10^5	4.44×10^2	3.71×10^6
	poisson					
STATS-CEB	uniform	142	2.00×10^2	5.75×10^7	1.48×10^6	2.26×10^9
	poisson	113	2.00×10^2	3.98×10^7	1.46×10^6	5.93×10^8
Q_c	poisson	1	1.32×10^{10}			

Table 1. Overview of benchmark queries.

consistently improves performance on our benchmarks. In contrast, for unchained index probing, we found that the overhead of caching may slightly outweigh its benefits, as the search spaces are extremely small (see Section 6.3). We therefore disabled unchained caching, unless explicitly specified otherwise. A detailed experimental analysis of caching is provided in the full paper version [4].

Baseline. We consider three possible M&S implementations as potential baselines, differing in how they materialize the full join:

- M-BJ: using a sequence of binary hashjoins.
- M-CSYA: using CSR-based SYA, i.e., by means of CSR-based nested semijoins followed by a flatten.
- M-USYA: the USR variant of the previous method.

A per-tuple Bernoulli trial is always performed to create the sample.

Prefix M indicates that these methods fully materialize the join. Since Apache Datafusion lacks a join order optimizer, we use DuckDB to generate binary join plans for M-BJ and then apply the cost-based rewriting technique of [3] to obtain the corresponding nested semijoin + flatten plans that are used in the other methods. Bekkers et al [3] have shown that computing full joins using chained SYA is instance-optimal, and therefore faster and more robust than using binary joins. On our benchmarks, we observe that M-CSYA is on average 91.4ms faster than M-BJ for JOB and 690ms for STATS-CEB. This hence confirms the findings of [3]. Furthermore, we find the M-CSYA is faster than M-USYA on JOB (21.4ms on avg.), while being competitive for STATS-CEB (median runtimes differ only 0.09ms). A more detailed comparison of chained and unchained variants is deferred to Section 6.3. Given that M-CSYA is significantly faster than M-BJ and faster or competitive with M-USYA, we fix it as the baseline to compare *I&P* against in what follows.

6.1 Uniform Sampling

Position sampling. We first analyze the efficiency of the uniform position sampling methods Bern, Geo, and Binom described in Section 5. Their expected runtimes are $O(n)$, $O(np)$, and $O(n \times \min(p, 1-p) + np)$, respectively. Bern should be outperformed by Geo and Binom for smaller sampling probabilities. Figure 7 confirms this hypothesis: the smaller p, the greater the improvement of Geo and Binom over Bern. However, for larger values of p, Bern becomes consistently faster. Binom follows the same trend as Geo, but is slightly slower due to higher constant-factor overhead. We hence discard Binom from further discussion and experiments.

A closer look reveals that, although the expected runtime of Bern is independent of p, the observed runtime actually increases with p up to approximately $p = 0.5$, and decreases for larger values of p. We attribute this non-monotonic behavior to the effects of *branch prediction*. Bern makes repeated random choices that depend on p. When $p \approx 0.5$, each branch is essentially unpredictable,

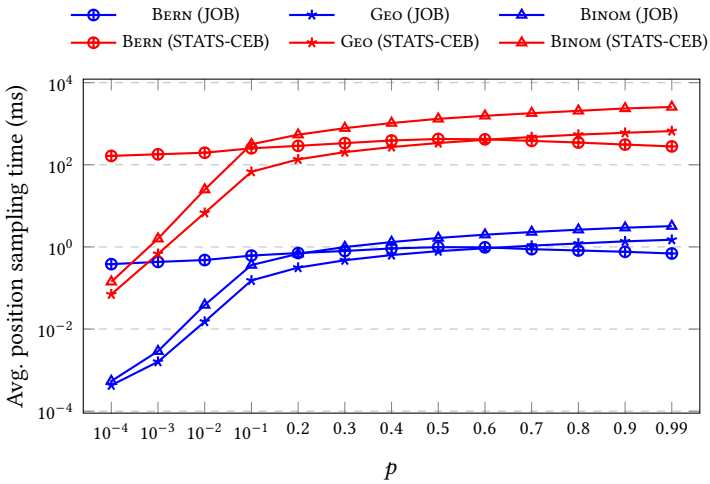


Fig. 7. Position sampling efficiency in function of p.

causing frequent mispredictions, which slows execution. In contrast, when p deviates from 0.5, the branch outcome becomes more predictable, reducing misprediction rates and improving runtime.

For large sampling probabilities, Geo requires nearly as many iterations as Bern. As a Geo iteration is more costly than performing a single lightweight Bernoulli trial, the constant overhead of Geo dominates, making Bern more efficient for high values of p.

End-to-end runtimes. We next compare the end-to-end runtimes of I&P methods for uniform sampling. We adopt the following naming scheme: for an index type $T \in \{\text{U(SR)}, \text{C(SR)}\}$ and uniform position sampling method $M \in \{\text{Geo}, \text{Bern}\}$, let I$_T$-$M$ denote the I&P algorithm obtained by using T and M as index and position sampling method, respectively.

Since Geo outperforms Bern for $p \leq 0.5$, we focus on I-Geo when $p \leq 0.5$ and on I-Bern otherwise.

Figure 8a shows that for JOB, chained I&P is faster than the M-SYA baseline across all sampling probabilities p, whereas unchained I&P is consistently slower. The differences remain small: averaged over all queries, the relative speedup of chained I&P over M-CSYA remains close to 1 for all p, reaching a maximum of 1.0056× at $p = 0.0001$. This is because the total runtime is dominated (82%) by reading input relations from disk and base table filtering, which is the same for all methods. Moreover, as shown in Figure 8a, the remaining 18% of the runtime is dominated by index construction, which is independent of the sampling probability. Hence the small difference between the three approaches across different values of p. Figure 8a also shows that chained sampling is faster than unchained due to the dominating but faster index building phase.

For STATS-CEB, the results in Figure 8b show more pronounced differences. Chained I&P remains the fastest method up to $p \leq 0.8$ and achieves substantially larger speedups over the baseline than for JOB, with a maximum speedup (averaged over all queries) of 38.79x at $p = 0.0001$. Because STATS-CEB has larger full join sizes (see Table 1), the overhead of materializing join tuples that are not part of the sampled output becomes more significant. This is confirmed in Figure 8b, showing that position sampling and index probing (for *I&P*) as well as flattening all full join tuples (given an index) and performing Bernoulli trails (for *M&S*) contribute more to the total runtime compared to JOB. Note that the index construction time is plotted as well, but so small that it is not visible.

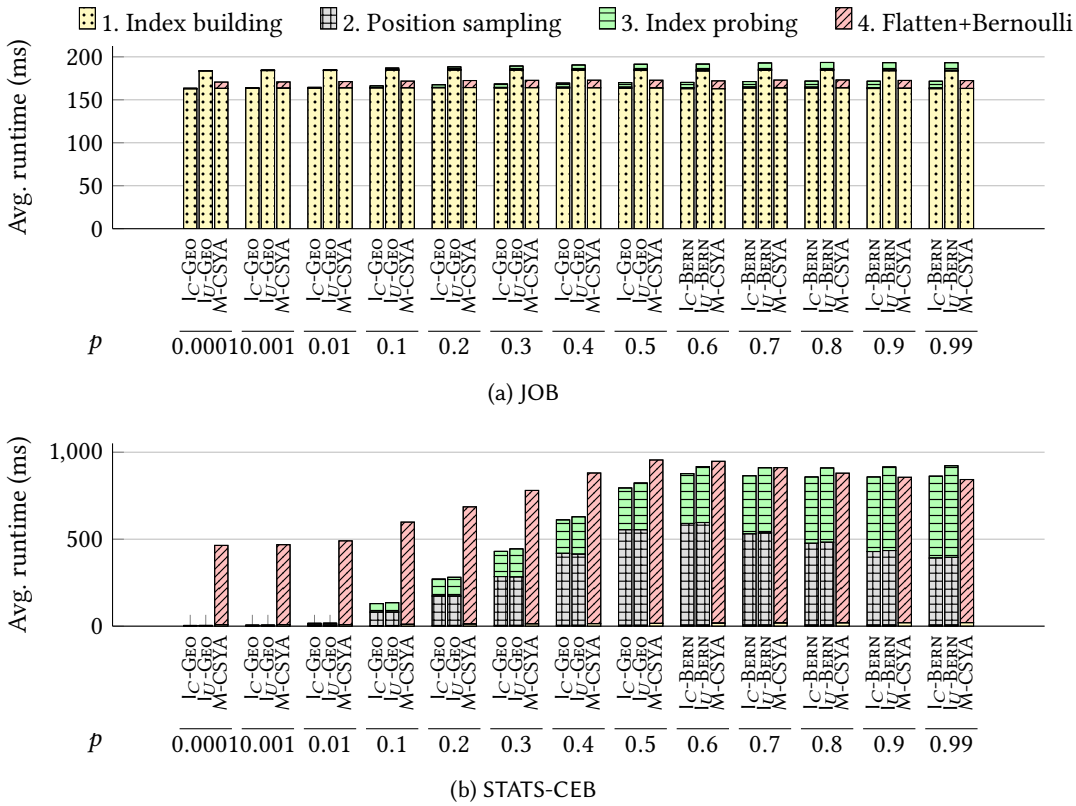


Fig. 8. A breakdown of the total runtime (excl. input reading and base table filtering) for each sampling probability p. Runtimes are averaged over all queries. For *I&P* methods, the remaining execution time consists of (1) index building, (2) position sampling and (3) index probing. For M-CSYA, the runtime is split into two phases: (1) index building, and (4) using the index to output the full join result (flatten) and performing a Bernoulli trial for each full join output tuple.

Moreover, chained *I&P* is faster than the unchained variant due to the faster index probing phase. This is in contrast to JOB, where chained *I&P* was faster due to faster index construction.

For STATS-CEB, chained I&P becomes slightly slower than M-CSYA for $p \geq 0.9$, which was not the case for JOB. To understand why, we note that the M-CSYA baseline uses a flatten operation that is highly optimized for sequential memory access [3]. In contrast, the index-based methods cannot exploit such optimizations, as they must assume random access and incur additional overhead during GET . These costs grow significant when the output is large—as in STATS-CEB, but not in JOB—explaining why index-based methods become slower for STATS-CEB at high sampling probabilities while remaining faster for JOB.

Conclusion. In summary, the choice of position sampling method should depend on the sampling probability. For small p, GEO has clear advantages due to its lower expected number of iterations, whereas for large p, the simpler control flow of BERN results in better performance. When considering end-to-end runtimes, chained I&P is preferable over unchained I&P, significantly outperforming full materialization for moderate and low sampling probabilities, confirming that avoiding full joins is particularly effective when only a fraction of the result is needed.

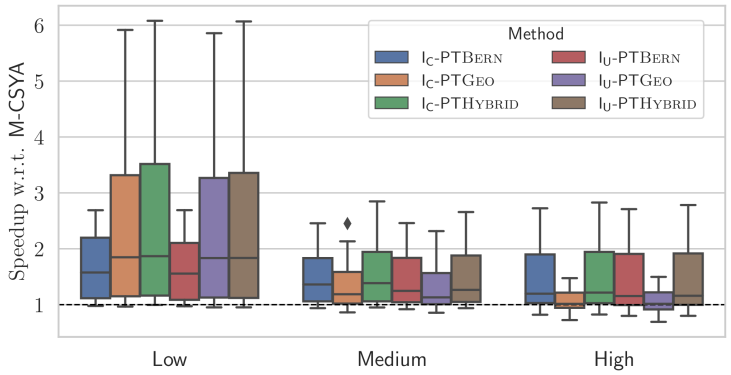


Fig. 9. Poisson sampling speedups w.r.t. M-CSYA on STATS-CEB for `low`, `medium`, and `high` sampling probabilities.

6.2 Poisson Sampling

We next turn to I&P methods for non-uniform Poisson sampling, adopting the following naming scheme: by I_T-M we denote the I&P algorithm for Poisson sampling obtained by using $T \in \{\text{U(SR)}, \text{C(SR)}\}$ and $M \in \{\text{PTGeo}, \text{PTBern}, \text{PTHybrid}\}$ as index and position sampling method, respectively. We omit the subscript when referring to both chained and unchained variants at the same time. Note that the non-uniform position sampling methods require the probability attribute y to appear as a flat attribute at the root of the shredded representation. We therefore rewrite the plans from Section 6.1 accordingly (see Proposition 5.1).

We observed in Section 6.1 that Geo outperforms Bern when $p \leq 0.5$. Based on this observation, we configure Hybrid to use Geo when $p \leq 0.5$, and Bern otherwise.

STATS-CEB. Figure 9 shows the relative end-to-end speedups obtained on STATS-CEB by the I&P methods compared to the M&S baseline M-CSYA. Overall, we observe that the chained variants perform slightly better than their unchained counterparts. For `medium` and `high` sampling probabilities, I-PTGeo performs the worst, and only for `low` probabilities it is competitive with I-PTHybrid. This aligns with the findings from Section 6.1, where we observed that Geo is slower than Bern (and hence Hybrid) when $p > 0.5$. I-PTBern becomes increasingly competitive with I-PTHybrid as the sampling probability grows, since both methods perform the same operations when $p > 0.5$.

Overall, I_C-PTHybrid performs consistently best, when looking at the distribution of the observed speedups in both relative and absolute terms. These are (min/avg/max):

- (0.995/2.39/6.08) x relative, and (-4.7×10^{-5}/0.4/5.54) s absolute for `low`,
- (0.95/1.54/2.85) x and (−0.0017/0.32/5.25) s for `medium`, and
- (0.82/1.49/2.83) x and (−0.007/0.28/5.72) s for `high` probabilities.

For comparison, I_U-PTHybrid achieves (min/avg/max) speedups of

- (0.95/2.34/6.07) x and (−0.0006/0.40/5.53) s for `low`,
- (0.94/1.49/2.66) x and (−0.0021/0.29/5.12) s for `medium`, and
- (0.80/1.45/2.78) x and (−0.8021/0.24/5.65) s for `high` probabilities.

We conclude that on STATS-CEB, the chained and unchained variants are competitive, with chained slightly more robust w.r.t. absolute regressions for `high` probabilities.

JOB. Table 2 reports on the distribution of the *absolute* end-to-end time differences obtained on JOB compared to the M-CSYA baseline. Positive numbers are improvements; negative numbers are regressions. Best-performing method is highlighted in bold.

	min			avg			max		
p	Geo	Bern	Hybrid	Geo	Bern	Hybrid	Geo	Bern	Hybrid
low	**-13.0**	-13.3	-13.3	**7.3**	6.4	7.2	102.8	102.0	**103.8**
medium	-13.1	**-8.2**	-12.6	4.6	4.3	**4.8**	93.5	**99.8**	97.4
high	-28.4	**-13.3**	-14.6	1.8	2.5	**2.6**	97.8	98.2	**98.9**

(a) Chained

	min			avg			max		
p	Geo	Bern	Hybrid	Geo	Bern	Hybrid	Geo	Bern	Hybrid
low	-2207.7	-2218.3	-2222.2	-108.7	-109.5	-109.4	33.5	37.4	33.6
medium	-2254.5	-2253.1	-2258.1	-112.1	-112.0	-112.0	38.1	37.2	38.5
high	-2283.4	-2274.2	-2276.9	-115.3	-114.3	-114.2	45.0	44.9	44.3

(b) Unchained

Table 2. Minimum, average, and maximum absolute Poisson sampling speedups (ms) on JOB.

We observe that chained methods behave more robust than their unchained counterparts: the largest regressions (in the min column) are significantly smaller for chained compared to unchained (100 times or more). Furthermore, on average (in the avg column) queries exhibit a slight chained speedup but still an unchained regression. Finally, the largest chained speedups are approximately three times that of the unchained speedups.

Overall, the speedups and regressions exhibited by chained methods are only marginal in terms of end-to-end runtime—on the order of only a few ms on average to a hundred ms at most, while total runtimes are in the range of seconds. By contrast the maximum regression exhibited by the unchained methods are on the same order (seconds), hence significant, while speedups are outperformed by the chained methods.

As far as the choice of position sampling method is concerned we see that there is little absolute difference between the three methods for the same index type. While I-PTHybrid is not always the best method in absolute terms, it is always competitive with the best method—differing only a few ms. This is consistent with our earlier observations.

EpiQL. Figure 10 reports total runtimes on Q_c for varying population sizes, where each size corresponds to a subset of the Belgian population. Method M-BJ runs out of memory for the full population size. In contrast, the instance-optimality of Shredded Yannakakis allowed us to materialize the full join output for 11M people within available memory. For a population size of 1M individuals, M-CSYA is 4.2x (or 130.1s) faster than M-BJ.

Among I&P methods, the unchained variants are consistently outperformed by their chained counterparts. The best-performing I&P method across all population sizes is I_C-PTHybrid, which achieves an end-to-end runtime improvement of 415.6s (5.3x) compared to the M-CSYA baseline for 11M individuals. For comparison, I_U-PTHybrid is only 382.4s (3.9x) faster than the baseline. These results demonstrate that avoiding full join materialization can be highly effective—especially in real-world Monte Carlo simulations where such queries must be executed repeatedly and efficiency gains accumulate.

We next compare the runtimes of the position sampling methods in isolation. Hybrid is the most efficient, closely followed by Geo, while Bern performs worst. For a population size of 11M individuals, position sampling using Geo is 34 s (or 3.4x) faster than Bern, and Hybrid is even 36 s (or 3.7x) faster than Bern. The explanation is twofold. First, the probability of two people having

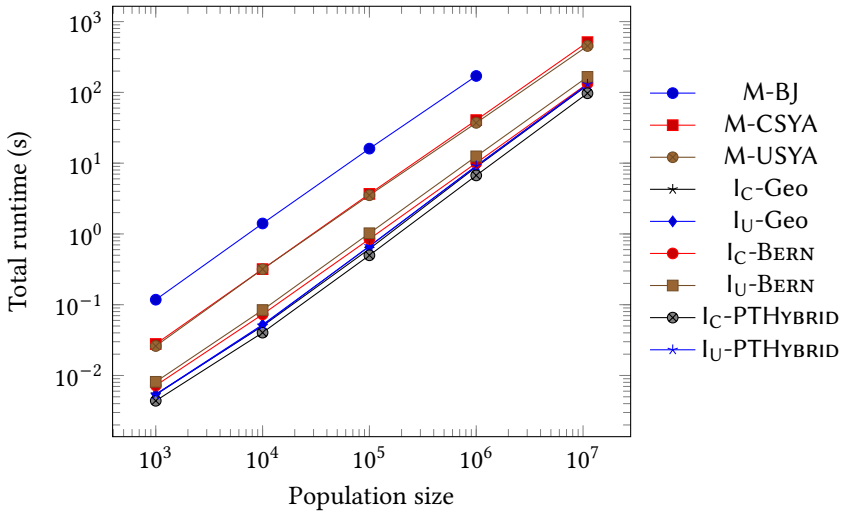


Fig. 10. Total runtime on Q_c.

	d						average		median		maximum	
Benchmark	min	25%	50%	75%	95%	max	chained	unchained	chained	unchained	chained	unchained
JOB (uniform)	1	1	2	4	12	342381	**4.30**	4.76	0.54	**0.51**	**168.28**	193.40
JOB (poisson)	1	2	2	6	9	15	6.53	**6.07**	**0.46**	**0.46**	**207.13**	214.53
STATS-CEB (uniform)	1	3	5	7	21	55	**202.57**	226.31	**1.90**	**1.90**	**18866.44**	20584.97
STATS-CEB (poisson)	2	4	11	15	59	137	**129.91**	149.76	**1.90**	2.14	**4902.71**	6222.82
Q_c (poisson)	$d = 50$						**4.4×10^3 (chained)**		21.1×10^3 (unchained)			

Table 3. Time in ms to probe a chained vs. unchained index.

Benchmark	Method	min	avg	med	max
JOB	chained SYA	18.83	1211.04	1270.75	3028.94
JOB	unchained SYA	19.28	1232.21	1279.84	3113.61
JOB	binary join	125.41	1304.88	1295.94	3164.28
STATS-CEB	chained SYA	1.286	212.59	14.87	5249.68
STATS-CEB	unchained SYA	1.227	186.35	14.78	4728.56
STATS-CEB	binary join	1.551	1178.65	26.47	46024.57

Table 4. End-to-end full join runtimes (ms).

contact is in practice very low. In particular, the average sampling probability is only 2.4%. Second, on 11M people the output of Q_c is 1.3×10^{10}. This large full join size in combination with the small sampling probabilities accounts for the observed differences between position sampling methods.

Conclusion. Across all benchmarks, I_C-PTHybrid consistently achieves the best performance among the Poisson sampling methods even though its asymptotic complexity is worse than that of I_U-PTHybrid.

6.3 Chained vs. unchained

Index building. We next compare USR and CSR in more depth. We begin by comparing the index-building time. Recall that USR construction requires two hashing passes, and is therefore

expected to be slower than CSR construction. This is confirmed in our experiments: for JOB, the (min/avg/max) CSR build-time speedups compared to USR are (0.91/1.3/2.17)x or (−9.9/69/131.3)ms; for STATS-CEB they are (0.82/1.02/1.36)x or (−8.1/0.14/3.7)ms. For Q_c with population size 11M, CSR building is 1.17x (12.8s) faster.

Index probing. Table 3 reports the avg/med/max probe times for both index types, along with a distribution of the largest join degree d over the entire set of queries. Uniform runtimes are aggregated across all sampling probabilities. We observe that CSR probing is often faster than USR probing. This is because d remains low for most of the queries, and for such d the binary search is empirically slower than linear search. One JOB query exhibits an extremely high max degree d (=342381). However, its full join output size is extremely low (=10), hence there is not much to be probed.

Full join processing. We next analyze the end-to-end runtimes of computing full joins using USR-based SYA versus CSR-based SYA. Once the index is built, these methods use the flatten operator (μ^*) to produce the join output instead of probing the index. In Table 4 we observe that on JOB, computing full joins using CSR is faster due to the dominating but faster index construction time. However, for STATS-CEB, computing full joins using CSR is actually slower. We observe that the flatten operation for USR is on average slightly faster than for CSR. We observe the same for Q_c; the unchained variant computes the full join 1.16x faster (60.7s) than the chained variant. A detailed analysis of the flatten operator is provided in the full paper version [4]. When considering *median* runtimes of both the flatten operator and the end-to-end time, however, the chained and unchained variants are competitive.

Chained or unchained? When are unchained methods preferable over the chained variants? To answer this question, we conducted an additional experiment on synthetic data, focusing on the binary join $\beta_p\,(S(x,y) \bowtie T(y,z))$ with T appearing in the right-hand side of the corresponding nested semijoin and $p \in [10^{-4}, 10^{-1}, 0.5, 0.9]$. We consider a setup that allows us to modulate the degree d, as this controls the theoretical complexity difference between CSR and USR probing. We consider three scenarios, fixing the join output size O to be 10^5, 10^6, and 10^7, respectively. Per fixed value of O and p, the number of probed tuples, which is a rough measure for the amount of work that needs to be done, hence remains constant. For each output size $O = 10^o$, we consider all configurations of inputs (S, T) such that $|S| = 10^s$ and $\deg_y(T) = 10^{o-s}$ for $1 \le s < o$, ensuring that each S-tuple joins with $\deg_y(T)$ tuples in T to get the desired output size. S is populated such that each key value appears only once in S, and hence $O = |T|$. Because each tuple in S joins with the same number of tuples in T, the maximum join degree d equals $\deg_y(T)$. Tuples in T are randomly permuted to ensure that tuples with the same y-value are not consecutive in the chained representation. We use Hybrid for position sampling. We enable caching for both CSR and USR as it consistently improves performance for both on this experiment.

Detailed measurements for all configurations and probabilities are given in the full paper version [4]; here we summarize the main trends. We observe that for $O = |T| = 10^5$ the CSR nxt array is small enough to fit in L2 CPU cache and for this reason, CSR probing (as well as total execution time) is faster than USR probing for all p, even for large degree $d = 10^4$. When $O = |T|$ increases to 10^6, CSR probing becomes slower than USR probing for all p, even for low degree $d = 10^1$. However, the total CSR runtime remains faster than USR for all join degrees (up to 10^5) due to the faster build time. When $O = |T|$ further increases to 10^7, CSR probing remains slower than USR probing but also the total runtime becomes slower for the majority of configurations. However, there remain configurations and values of p where CSR's faster index construction time results in lower total runtime for CSR.

In conclusion, this synthetic experiment shows that there are cases where the theoretically faster USR probe time is also experimentally faster, even when the degree is relatively small. However, this does not necessarily translate to a faster overall runtime, due to the interaction with (i) CPU caching effects; (ii) the index construction time; and (iii) the sampling probability. On our real-world benchmarks, chained I&P methods are more robust compared to their unchained counterparts. Since both variants are competitive for full join materialization we recommend the chained index as the preferred choice for both sampling and full join computation.

7 Related Work

There is extensive work on *uniform sampling* over acyclic joins [1, 7, 8, 10, 38], differing in the subclasses of acyclic joins that are supported, from binary joins [8], over foreign-key joins [1] to arbitrary acyclic joins [7, 10, 38] as well as the specific kind of uniform sampling considered—from arbitrary uniform sampling [1, 8, 38], over fixed-size sampling [7] to fixed-size sampling in a streaming context [10]. We consider arbitrary Poisson sampling without fixing a sample size.

Existing works also differ in whether sampling is performed with replacement [1, 8, 38] or, like our work, without replacement [7, 10]. Any sampling-with-replacement algorithm can be adapted to sampling without replacement by rejecting duplicates; however, as the sample size grows, repeated resampling leads to increasing rejection overhead, making this approach inefficient compared to methods designed for sampling without replacement [7]. The fixed-size sampling approach taken by Carmeli et al [7] is the closest to our work. However, we consider non-uniform sampling with a focus on implementation in column stores and show that from a non-asymptotic viewpoint the theoretically best method is not the practically most efficient. Prior work on *subset sampling* [21, 28, 36]—where each element of a set has its own sampling probability—addresses a simpler setting, as it does not involve the challenge of avoiding materializing full joins.

In parallel to our own work, Esmailpour et al. [13] have recently also studied Poisson sampling over acyclic joins. Compared to our work, they focus on a theoretical perspective, also considering the setting where the sampling probability is not in a single relation, but computed from attributes from multiple relations. By contrast, we focus on the practical implementation in column stores, limiting the probability to come from attributes of a single relation.

8 Conclusions

We introduced the problem of Poisson sampling over acyclic joins, where each join tuple specifies the probability with which it is to be included in the sample. This setting generalizes classical uniform sampling. We presented a nearly instance-optimal algorithm based on the Index-and-Probe paradigm. We have shown how to efficiently implement this approach in column stores by building upon the SYA algorithm from [3]. Additionally, we compared two different index structures, CSR and USR. While USR achieves the theoretically optimal access time, we found empirically that CSR offers the best end-to-end performance. Since CSR and USR are competitive for computing full joins, it is possible to adopt SYA with CSR as a uniform basis for both sampling and join processing.

Acknowledgments

Liese Bekkers and Stijn Vansummeren were supported by the Bijzonder Onderzoeksfonds (BOF) of Hasselt University (Belgium) under Grants No. BOF22DOC07 and BOF20ZAP02. This research was further supported by Research Foundation Flanders (FWO) under Grant No. G0B9623N.

References

[1] Swarup Acharya, Phillip B. Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. 1999. Join Synopses for Approximate Query Answering. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June*

1-3, 1999, Philadelphia, Pennsylvania, USA, Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh (Eds.). ACM Press, 275–286. doi:10.1145/304182.304207

[2] Catriel Beeri, Ronald Fagin, David Maier, Alberto O. Mendelzon, Jeffrey D. Ullman, and Mihalis Yannakakis. 1981. Properties of Acyclic Database Schemes. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing, May 11-13, 1981, Milwaukee, Wisconsin, USA*. ACM, 355–362. doi:10.1145/800076.802489

[3] Liese Bekkers, Frank Neven, Stijn Vansummeren, and Yisu Remy Wang. 2025. Instance-Optimal Acyclic Join Processing Without Regret: Engineering the Yannakakis Algorithm in Column Stores. *Proc. VLDB Endow.* 18, 8 (2025), 2413–2426. https://www.vldb.org/pvldb/vol18/p2413-vansummeren.pdf

[4] Liese Bekkers, Lorrens Pantelis, Frank Neven, and Stijn Vansummeren. 2026. *Poisson Sampling over Acyclic Joins.* Technical Report. Full paper version, available at https://arxiv.org/abs/2603.10982.

[5] Johann Brault-Baron. 2012. A Negative Conjunctive Query is Easy if and only if it is Beta-Acyclic. In *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France (LIPIcs, Vol. 16)*, Patrick Cégielski and Arnaud Durand (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 137–151. doi:10.4230/LIPICS.CSL.2012.137

[6] Zhuhua Cai, Zografoula Vagena, Luis Perez, Subramanian Arumugam, Peter J. Haas, and Christopher Jermaine. 2013. Simulation of Database-Valued Markov Chains Using SimSQL. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 637–648. doi:10.1145/2463676.2465283

[7] Nofar Carmeli, Shai Zeevi, Christoph Berkholz, Alessio Conte, Benny Kimelfeld, and Nicole Schweikardt. 2022. Answering (Unions of) Conjunctive Queries using Random Access and Random-Order Enumeration. *ACM Trans. Database Syst.* 47, 3 (2022), 9:1–9:49. doi:10.1145/3531055

[8] Surajit Chaudhuri, Rajeev Motwani, and Vivek R. Narasayya. 1999. On Random Sampling over Joins. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA*, Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh (Eds.). ACM Press, 263–274. doi:10.1145/304182.304206

[9] James Cheney, Sam Lindley, and Philip Wadler. 2014. Query shredding: efficient relational evaluation of queries over nested multisets. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu (Eds.). ACM, 1027–1038. doi:10.1145/2588555.2612186

[10] Binyang Dai, Xiao Hu, and Ke Yi. 2025. Reservoir Sampling over Joins. *SIGMOD Rec.* 54, 1 (2025), 70–78. doi:10.1145/3733620.3733635

[11] Jan Van den Bussche. 2001. Simulation of the nested relational algebra by the flat relational algebra, with an application to the complexity of evaluating powerset algebra expressions. *Theor. Comput. Sci.* 254, 1-2 (2001), 363–377. doi:10.1016/S0304-3975(99)00301-1

[12] Luc Devroye. 1986. *Non-Uniform Random Variate Generation.* Springer. doi:10.1007/978-1-4613-8643-8

[13] Aryan Esmailpour, Xiao Hu, Jinchao Huang, and Stavros Sintos. 2026. Subset Sampling over Joins. *Proc. ACM Manag. Data* 4, 2 (PODS), Article 117 (2026). doi:10.1145/3801913

[14] Ronald Fagin. 1983. Degrees of Acyclicity for Hypergraphs and Relational Database Schemes. *J. ACM* 30, 3 (1983), 514–550. doi:10.1145/2402.322390

[15] M. H. Graham. 1979. *On the universal relation.* Technical Report. University of Toronto, Toronto, Ontario, Canada.

[16] Peter J. Haas and Joseph M. Hellerstein. 1999. Ripple Joins for Online Aggregation. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA*, Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh (Eds.). ACM Press, 287–298. doi:10.1145/304182.304208

[17] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, Zhengping Qian, Jingren Zhou, Jiangneng Li, and Bin Cui. 2021. Cardinality Estimation in DBMS: A Comprehensive Benchmark Evaluation. *Proc. VLDB Endow.* 15, 4 (2021), 752–765. doi:10.14778/3503585.3503586

[18] Thang Hoang, Pietro Coletti, Alessia Melegaro, Jacco Wallinga, Carlos G. Grijalva, W. John Edmunds, Philippe Beutels, and Niel Hens. 2019. A Systematic Review of Social Contact Surveys to Inform Transmission Models of Close-contact Infections. *Epidemiology* 30, 5 (September 2019), 723–736. doi:10.1097/EDE.0000000000001047

[19] Thang Van Hoang, Pietro Coletti, Yimer Wasihun Kifle, Kim Van Kerckhove, Sarah Vercruysse, Lander Willem, Philippe Beutels, and Niel Hens. 2021. Close contact infection dynamics over time: insights from a second large-scale social contact survey in Flanders, Belgium, in 2010-2011. *BMC Infectious Diseases* 21, 274 (2021). doi:10.1186/s12879-021-05949-4

[20] Zeyuan Hu, Yisu Remy Wang, and Daniel P Miranker. 2024. TreeTracker Join: Simple, Optimal, Fast. *arXiv preprint arXiv:2403.01631* (2024).

[21] Jinchao Huang and Sibo Wang. 2023. Subset Sampling and Its Extensions. arXiv:2307.11585 [cs.DS] https://arxiv.org/abs/2307.11585

[22] Ravi Jampani, Fei Xu, Mingxi Wu, Luis Perez, Chris Jermaine, and Peter J. Haas. 2011. The Monte Carlo Database System: Stochastic Analysis Close to the Data. *ACM Trans. Database Syst.* 36, 3 (Aug. 2011), 18:1–18:41. doi:10.1145/2000824.2000828

[23] Voratas Kachitvichyanukul and Bruce W. Schmeiser. 1988. Binomial Random Variate Generation. *Commun. ACM* 31, 2 (1988), 216–222. doi:10.1145/42372.42381

[24] Matthias Lanzinger, Reinhard Pichler, and Alexander Selzer. 2025. Avoiding Materialisation for Guarded Aggregate Queries. *Proc. VLDB Endow.* 18, 5 (Aug. 2025), 1398–1411. doi:10.14778/3718057.3718068

[25] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215. doi:10.14778/2850583.2850594

[26] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. 2019. Wander Join and XDB: Online Aggregation via Random Walks. *ACM Trans. Database Syst.* 44, 1 (2019), 2:1–2:41. doi:10.1145/3284551

[27] Robert Endre Tarjan and Mihalis Yannakakis. 1984. Simple Linear-Time Algorithms to Test Chordality of Graphs, Test Acyclicity of Hypergraphs, and Selectively Reduce Acyclic Hypergraphs. *SIAM J. Comput.* 13, 3 (1984), 566–579. doi:10.1137/0213035

[28] Alastair J. Walker. 1977. An Efficient Method for Generating Discrete Random Variables with General Distributions. *ACM Trans. Math. Softw.* 3, 3 (Sept. 1977), 253–256. doi:10.1145/355744.355749

[29] Qichen Wang, Bingnan Chen, Binyang Dai, Ke Yi, Feifei Li, and Liang Lin. 2025. Yannakakis+: Practical Acyclic Query Evaluation with Theoretical Guarantees. *Proc. ACM Manag. Data* 3, 3, Article 235 (June 2025), 28 pages. doi:10.1145/3725423

[30] Lander Willem, Kim Van Kerckhove, Dennis L. Chao, Niel Hens, and Philippe Beutels. 2012. A nice day for an infection? Weather conditions and social contact patterns relevant to influenza transmission. *PLoS ONE* 7, 11 (2012), e48695. doi:10.1371/journal.pone.0048695 Epub 2012 Nov 14.

[31] Lander Willem, Frederik Verelst, Joke Bilcke, Niel Hens, and Philippe Beutels. 2017. Lessons from a decade of individual-based models for infectious disease transmission: a systematic review (2006–2015). *BMC Infectious Diseases* 17, 612 (2017). doi:10.1186/s12879-017-2699-8

[32] Limsoon Wong. 1993. Normal Forms and Conservative Properties for Query Languages over Collection Types. In *Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 25-28, 1993, Washington, DC, USA*, Catriel Beeri (Ed.). ACM Press, 26–36. doi:10.1145/153850.153853

[33] Yifei Yang, Hangdong Zhao, Xiangyao Yu, and Paraschos Koutris. 2024. Predicate Transfer: Efficient Pre-Filtering on Multi-Join Queries. In *14th Conference on Innovative Data Systems Research, CIDR 2024, Chaminade, HI, USA, January 14-17, 2024*. www.cidrdb.org. https://www.cidrdb.org/cidr2024/papers/p22-yang.pdf

[34] Mihalis Yannakakis. 1981. Algorithms for Acyclic Database Schemes. In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*. IEEE Computer Society, 82–94.

[35] C. T. Yu and M. Z. Ozsoyoglu. 1979. An algorithm for tree-query membership of a distributed query. In *The IEEE Computer Society's Third International Computer Software and Applications Conference, COMPSAC 1979, 6-8 November, 1979, Chicago, Illinois, USA*. IEEE, 306–312. doi:10.1109/CMPSAC.1979.762509

[36] Fangyuan Zhang, Mengxu Jiang, and Sibo Wang. 2023. Efficient Dynamic Weighted Set Sampling and Its Extension. *Proc. VLDB Endow.* 17, 1 (2023), 15–27. doi:10.14778/3617838.3617840

[37] Junyi Zhao, Kai Su, Yifei Yang, Xiangyao Yu, Paraschos Koutris, and Huanchen Zhang. 2025. Debunking the Myth of Join Ordering: Toward Robust SQL Analytics. *Proc. ACM Manag. Data* 3, 3, Article 146 (June 2025), 28 pages. doi:10.1145/3725283

[38] Zhuoyue Zhao, Robert Christensen, Feifei Li, Xiao Hu, and Ke Yi. 2018. Random Sampling over Joins Revisited. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 1525–1539. doi:10.1145/3183713.3183739

Received October 2025; revised January 2026; accepted February 2026