Random Sampling and Size Estimation over Cyclic Joins

Yu Chen and Ke Yi
Hong Kong University of Science and Technology
(ychenbh, yike)@cse.ust.hk

Abstract

Computing joins is expensive, and often unnecessary when the output size is large. In 1999, Chaudhuri et al. [7] posed the problem of random sampling over joins as a potentially effective approach to avoiding computing the join in full, while obtaining important statistical information about the join results. Unfortunately, no significant progress has been made in the last 20 years, except for the case of acyclic joins. In this paper, we present the first non-trivial result on sampling over cyclic joins. We show that after a linear-time preprocessing step, a join result can be drawn uniformly at random in expected time $O(\text{IN}^\rho/\text{OUT})$, where IN^ρ is known as the AGM bound of the join and OUT is its output size. This result holds for all joins on binary relations, as well as certain joins on relations of higher arity. We further show how this algorithm immediately leads to a join size estimation algorithm with the same running time.

2012 ACM Subject Classification General and reference → General literature; General and reference

Keywords and phrases Random sampling, join size estimation

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

A join query can be modeled as a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, with $|\mathcal{V}| = n$ vertices and $m = |\mathcal{E}|$ hyperedges. Each vertex models an attribute, and for each hyperedge $F \in \mathcal{E}$, there is a relation R_F on attribute set F. We use $Q := \bowtie_{F\in\mathcal{E}} R_F$ to denote the set of join results. In this paper, we consider the data complexity of query evaluation, i.e., the running time of the algorithms will be measured by the total input size $\text{IN} = \sum_{F\in\mathcal{E}} |R_F|$ and the output size $\text{OUT} = |Q|$, while assuming n and m are constants.

Algorithms for computing Q have been extensively studied. It is well known that Q can be computed in $O(\text{IN}^\rho)$ time, where ρ is the optimal solution of the following linear program [3, 13, 14, 16]:

$$\begin{aligned} &\min_{x_F, F\in\mathcal{E}} \log_{\text{IN}} |R_F| \cdot x_F \\ \text{s.t.} &\sum_{F, v\in F} x_F \geq 1, \text{ for every } v \in \mathcal{V}, \\ &0 \leq x_F \leq 1, \text{ for every } F \in \mathcal{E}. \end{aligned} \tag{1}$$

This is worst-case optimal since OUT can be as large as $\Theta(\text{IN}^\rho)$ on some instances. Alternatively, output-sensitive algorithms are known that compute Q in $O(\text{IN}^w + \text{OUT})$ time, where w is certain notion of *width* of the hypergraph $\mathcal{H}$ [10, 11]. But this is still very expensive for highly cyclic queries, for which w is close to ρ, or when OUT is large. Indeed, computing multi-way joins is still the bottleneck in query evaluation in modern database systems.

One key observation made in as early as 1999 [7] is that rarely are full join results required by the user. In almost all use cases, the join results are aggregated and presented to the user in a succinct form. The aggregation function can be as simple as a count or a sum, a random sample, or an arbitrary UDF. Thus, the intriguing question is, can we

compute the final query result without computing the join in full? To make the problem well defined and amenable to theoretical investigation, in this paper we consider two particular aggregation functions: random sampling and (approximate) counting. These are arguably the two most basic aggregates, which other more complicated aggregations can be based upon. For example, sum can be considered as a weighted version of the count, while many UDFs (e..g, median and quantiles) can be computed approximated from a random sample in lieu of full data.

1.1 Previous results on random sampling over joins

The starting observation from the pioneering work of Chaudhuri et al. [7] is that the sampling operator cannot be pushed down through a join operator, i.e, $\text{sample}(R_1) \bowtie \text{sample}(R_2) \neq \text{sample}(R_1 \bowtie R_2)$. To see this, consider a binary join $R_1(A, B) \bowtie R_2(B, C)$, with $R_1 = \{(a_1, b_1), (a_1, b_2), \ldots, (a_N, b_2)\}$ and $R_2 = \{(b_1, c_1), (b_1, c_2), \ldots, (b_1, c_N), (b_2, c_1)\}$. Note that the join size is $\text{OUT} = 2N$. Thus, to be able to sample a join result uniformly at random from these $2N$ join results, one has to *non-uniformly* sample tuples from either R_1 or R_2. In particular, the (a_1, b_1) in R_1 must be sampled with a probability that is N times larger than the other tuples in R_1, because it joins with N tuples in R_2. More formally, Chaudhuri et al. [7] prove that, without precomputing some auxiliary information from the data, drawing a sample from $R_1 \bowtie R_2$ requires at least $\Omega(\text{IN})$ time. In view of this negative result, they first collect the frequency information from R_2, i.e., $|\sigma_{B=b}R_2|$ for each distinct b, and build a weighted sampling data structure on R_1 using these frequencies as weights. One can use the "alias method" [6] to build a weighted sampling structure in linear time, which supports drawing a weighted sample in $O(1)$ time. After drawing a weighted sample t_1 from R_1, we randomly draw a tuple t_2 from $R_2 \ltimes t_1$[1], which can be done in constant time if there is an index on R_2. Note that an index can also be built in linear time. Therefore, the formal result of Chaudhuri et al. [7] is that a data structure can be built in linear time, which allows one to draw a random sample from $R_1 \bowtie R_2$ in $O(1)$ time.

At the same SIGMOD conference with Chaudhuri et al. [7], Acharya et al. [1] studied the problem of random sampling over multi-way joins. However, their algorithm only works for a very special type of joins with foreign-key constraints, which imply that there is a one-to-one correspondence between the join results and tuples in the largest table. Thus, random sampling from the join reduces to sampling from a single table, which is trivial.

This problem then stayed dormant for almost 20 years, until Zhao et al. [19] designed an algorithm to sample from multi-way acyclic joins. They show that, on any acyclic join, a data structure can be build in linear time that allows one to draw a random sample from the join results in constant time. Their observation is that, on a multi-way join Q, one should sample a tuple t with probability proportional to $|Q_t|$, where $Q_t = \bowtie_{F \in \mathcal{E}} (R_F \ltimes t)$ is the *residual query* of t. Note that on the binary join $R_1(A, B) \bowtie R_2(B, C)$, the size of the residual query of some tuple $t = (a, b) \in R_1$ is exactly the frequency of b in R_2. Then, the idea is to do so recursively, with the algorithm of Chaudhuri et al. [7] becoming the base case. Finally, they make use of the (probably folklore) result that, for an acyclic join [18], all the residual query sizes can be computed in $O(\text{IN})$ time. Then they organize these residual query sizes in an appropriate data structure so that a sample can be drawn in $O(1)$ time. They also adapt their algorithm to cyclic queries, but there is no formal guarantee on its performance.

[1] We use $R \ltimes t$ as an abbreviation for $R \ltimes \{t\}$.

1.2 Previous results on join size estimation

For an acyclic query Q, it is well known that $\text{OUT} = |Q|$ can be computed in $O(\text{IN})$ time. But the problem looks very difficult for cyclic queries. Even for the simplest cyclic query, the triangle query $Q_\triangle = R_{\{A,B\}} \bowtie R_{\{B,C\}} \bowtie R_{\{A,C\}}$, we currently do not have any algorithm that can compute OUT faster than $O(\text{IN}^\rho)$ time[2], i.e., counting seems to be as difficult as computing the full join results. On general cyclic queries, the $O(\text{IN}^w + \text{OUT})$-time output-sensitive join algorithm above can be modified so as to compute OUT in $O(\text{IN}^w)$ time, but this is still very expensive for highly cyclic queries.

In most applications, we do not need an accurate OUT, while a reasonable estimate would be good enough. In the database literature, this is known as the "query size estimation" problem, and has been extensively studied. However, most results in this area are heuristics without formal guarantees on the accuracy of the estimate.

The problem of estimating $|Q_\triangle|$, i.e., counting triangles in a graph, has received particular attention. A commonly used model in the algorithms community is the *property testing* model, which assumes that the graph has already been preprocessed into some standard graph data structure (e.g., adjacent lists with hash tables), so that one can perform the following operations in constant time: randomly sampling a vertex, randomly sampling an edge, returning the degree of a vertex, returning the i-th neighbor of a vertex, and testing if an edge exist between two vertices. In this model, Eden et al. [8] designed an $\tilde{O}\left(\frac{\text{IN}^{3/2}}{\text{OUT}}\right)$-time[3] algorithm that returns a constant-factor approximation of OUT with constant probability, and showed that this is optimal. Their algorithm has been later extended to counting length-k cycles and size-k cliques for any constant k, and the running time becomes $\tilde{O}\left(\frac{\text{IN}^{k/2}}{\text{OUT}}\right)$ [4, 9]. Very recently, Assadi et al. [2] extended this algorithm to computing a constant-factor approximation of the join size of any query on binary relations in time of $\tilde{O}\left(\frac{\text{IN}^\rho}{\text{OUT}}\right)$. They also proved a matching lower bound, which actually holds for the problem of distinguishing between an input with no join result and one with OUT join results. So it holds for both the sampling problem and the join size estimation problem. However, their algorithm only works for the join size estimation problem, not sampling.

Unfortunately, the aforementioned algorithms perform very badly in practice, despite their theoretical optimality. For the triangle counting problem, one of the most practically efficient algorithms is *wedge sampling* [15], which departs from the property testing model slightly. A *wedge* is just a length-2 path. The basic idea of wedge sampling is to first uniformly sample a wedge, and then check if the wedge is closed, i.e., forms a triangle. The standard property testing model does not allow one to uniformly sample a wedge in constant time, but this can be easily supported by building a weighted sampling structure on all the vertices, where the weight of each vertex v is $d(v)(d(v)-1)/2$, which is exactly the number of wedges centered at v. This weighted sampling data structure can be built cheaply in a linear-time preprocessing step. Extending the idea of wedge sampling, MOSS-5 [17] is an algorithm that can count any pattern with up to 5 vertices. Instead of sampling wedges, MOSS-5 samples spanning trees of up to 5 vertices, and then checks if the other required edges are present. The same algorithm

[2] On the triangle query, the optimal solution to the linear program (1) is either $x_1 = x_2 = x_3 = 1/2$, or $x_1 = x_2 = 1, x_3 = 0$ (ignoring the other two symmetric cases). In the former case, $O(\text{IN}^\rho) = O(\text{IN}^{3/2})$; in the latter case, $O(\text{IN}^\rho) = O(|R_1| \cdot |R_2|)$.

[3] The bound stated in [8] has an extra term, since they did not use edge sampling, which is sometimes not included in the standard property testing model. When edge sampling is allowed, the bound simplifies to $\tilde{O}\left(\frac{\text{IN}^{3/2}}{\text{OUT}}\right)$.

also applies to the join size estimation problem on joins over binary relations involving up to 5 attributes, but it does not have any theoretical guarantees. Wander Join [12] is an effective approach to answering join-aggregate queries approximately, with join size estimation as a special case. It returns non-uniform samples from the join results, and de-biases them using the Horvitz-Thompson estimator. The estimator is unbiased, but there is no guarantee on its error.

1.3 Our results

In this paper, we present the first nontrivial algorithm for random sampling over arbitrary cyclic queries. More formally, we show how to construct a data structure in linear time (the size of the data structure is thus necessarily no more than linear), so that a sample can be drawn uniformly at random from the join results in $O\left(\frac{\text{IN}^\rho}{\text{OUT}}\right)$ time in expectation, for the class of *sequenceable* queries. The precise definition of sequenceable queries is a bit technical; please see Section 2.5 for details. They include all queries on binary relations, as well as certain queries on relations of higher arity. For non-sequenceable queries, the running time for drawing a sample is $O\left(\frac{\text{IN}^{\rho+1}}{\text{OUT}}\right)$. These results hold for both full join queries and join-project queries. Prior to this work, the only solution to this problem with formal guarantees is to either precompute the full join results, which has $O(\text{IN}^\rho)$ preprocessing and storage cost[4], and $O(1)$ sampling cost, or compute the full join at sampling time, which has no preprocessing cost but $O(\text{IN}^\rho)$ sampling cost.

We also adapt our algorithm to solve the join size estimation problem. We show that after drawing a constant number of samples, a constant-factor approximation to the join size can be obtained with constant probability. This matches the recent result of Assadi et al. [2] on joins over binary relations[5]. Compared with [2], our result is different in the following aspects: (1) We have to build some additional data structures (still in linear time), while only standard graph representations are needed in [2], so their result in stronger in this respect. (2) Our algorithm supports random sampling from the join results, with join size estimation as a simple corollary, while [2] does not support random sampling. (3) Our algorithm supports certain queries on relations of higher arity, while [2] only supports binary relations. (4) The algorithm of [2] has some hidden logarithmic factors, while ours does not. (5) Unlike the algorithm [2] which is of only theoretical interest, our algorithm is actually very practical. We conducted some experiments in Section A, showing that our algorithm is competitive with the best known heuristics on the join size estimation problem.

The $O\left(\frac{\text{IN}^\rho}{\text{OUT}}\right)$ bound may not look attractive when OUT is small. In particular, if $\text{OUT} = O(1)$, our algorithm is no better than computing the join results in full. However, improving this can be very difficult. Note that when $\text{OUT} = O(1)$, random sampling from the join results is the same as finding all the results. Even for the triangle query, the best algorithm to date still takes $O(\text{IN}^{1.408})$ time when $\text{OUT} = O(1)$ [5], using the highly impractical fast matrix multiplication algorithm. This is only slightly better than $O(\text{IN}^\rho) = O(\text{IN}^{1.5})$.

[4] By combining the acyclic join sampling algorithm [19] with the generalized tree decomposition framework [10], the preprocessing cost can be driven down to $O(\text{IN}^{\text{fhw}})$, where fhw is the *fractional hypertree width* of the query, but $\rho = \text{fhw}$ for highly cyclic queries.

[5] In fact, our work was done independent of [2], which we just came to know before submitting this paper. Also, our approach is completely different from [2].

2 Random Sampling over Cyclic Queries

2.1 Overview of approach

Order the vertices (attributes) in $\mathcal{V}$ arbitrarily as $v_1, v_2, \ldots, v_n$. Let $\mathrm{dom}(v_i)$ be the domain of attribute v_i. For any $X \subseteq \mathcal{V}$, let $\mathcal{E}_X = \{F \in \mathcal{E} \mid F \cap X \neq \emptyset\}$. We use I to denote the attribute set $\{v_1, v_2, \ldots, v_i\}$ and J denotes $\{v_{i+1}, \ldots, v_n\}$, for $i = 1, \ldots, n$. Define $I = \emptyset$ when $i = 0$. For a tuple t on attributes I, define the residual query on t as:

$$Q_t = \bowtie_{F \in \mathcal{E}_J} \pi_J(R_F \ltimes t). \tag{2}$$

Our starting point is Generic-Join [14], an elegant worst-case optimal join algorithm. A particular version of the algorithm is shown in Algorithm 1 . In the algorithm description, $\langle\rangle$ denotes the empty tuple; (t, y) denotes the concatenation of t and y, where t is a tuple on attributes I, and $y \in \mathrm{dom}(v_{i+1})$.

Algorithm 1: Generic-Join(i, t):

```
// t is a tuple on I = {v_1, ..., v_i} and t ∈ π_I R_F for all F ∈ E_I
if i = n then return {⟨⟩}
Q_t ← ∅
L_t ← ⋂_{F ∈ E_{v_{i+1}}} π_{v_{i+1}}(R_F ⋉ t)
foreach y ∈ L_t do
    Q_(t,y) ← Generic-Join(i + 1, (t, y))
    Q_t ← Q_t ∪ {y} × Q_(t,y)
return Q_t
```

Let $(x_F, F \in \mathcal{E})$ be the optimal solution to linear program (1). It is known that the total running time of the Generic-Join algorithm is bounded by the AGM bound [3] of the query:

$$\mathsf{AGM}(Q) = \prod_{F \in \mathcal{E}} |R_F|^{x_F}.$$

Furthermore, the time spent on each recursive call Generic-Join(i, t) is bounded by the AGM bound on the residual query Q_t (define $0^0 = 0$):

$$\mathsf{AGM}(Q_t) = \prod_{F \in \mathcal{E}_J} |\pi_J(R_F \ltimes t)|^{x_F} = \prod_{F \in \mathcal{E}_J} |R_F \ltimes t|^{x_F}. \tag{3}$$

Unfolding the recursion, the execution process of the Generic-Join algorithm forms a tree $\mathcal{T}$. The root node of $\mathcal{T}$ corresponds to the initial call Generic-Join$(0, \langle\rangle)$; every node on the i-th level of $\mathcal{T}$ corresponds to a tuple t on attribute $I = (v_1, \ldots, v_i)$; a leaf node on level n corresponds to a join result. This tree has exactly OUT leaves. Below, we will not differentiate between a node in the tree and its corresponding tuple t.

If we know the residual query size $|Q_t|$ for every t, then this algorithm immediately yields a random sampling algorithm: At each node t of $\mathcal{T}$, instead of exploring all its children (t, y) for $y \in L_t$, we just sample one of them with probability proportional to its subtree size, i.e., sample (t, y) with probability $|Q_{(t,y)}|/|Q_t|$. This way, we will reach every leaf with equal probability. In fact, this is exactly the basic idea of the random sampling algorithm over acyclic queries [19].

For cyclic queries, unfortunately, there is no efficient way to compute all the residual query sizes $|Q_t|$. Our idea is to assume that each node t had a subtree size of $\mathsf{AGM}(Q_t)$,

and perform the sampling using these subtree size upper bounds. This can be equivalently viewed as adding "rejection nodes" at various places of $\mathcal{T}$, such that each node t has exactly $\mathsf{AGM}(Q_t)$ leaves below, which include $|Q_t|$ "accept nodes", which correspond to true join results, and $\mathsf{AGM}(Q_t) - |Q_t|$ rejection nodes, which correspond to failed sampling paths.

More precisely, to sample a join result, we start at the root of $\mathcal{T}$. At each node t, we randomly sample a child $y \in L_t$ with probability $\mathsf{AGM}(Q_{(t,y)})/\mathsf{AGM}(Q_t)$, and reject with probability $1 - \sum_{y \in L_t}(\mathsf{AGM}(Q_{(t,y)})/\mathsf{AGM}(Q_t))$. Note that the sum of the sampling probabilities of all children $y \in L_t$ will not exceed 1, due to the query decomposition lemma [14], which states that

$$\sum_{y \in L_t} \mathsf{AGM}(Q_{(t,y)}) \leq \mathsf{AGM}(Q_t). \tag{4}$$

Finally, the probability to reach any leaf $t = (y_1, y_2, \ldots, y_n) \in Q$ is

$$\frac{\mathsf{AGM}(Q_{(y_1)})}{\mathsf{AGM}(Q)} \cdot \frac{\mathsf{AGM}(Q_{(y_1,y_2)})}{\mathsf{AGM}(Q_{(y_1)})} \cdot \cdots \cdot \frac{\mathsf{AGM}(Q_{(y_1,\ldots,y_n)})}{\mathsf{AGM}(Q_{(y_1,\ldots,y_{n-1})})} = \frac{1}{\mathsf{AGM}(Q)},$$

i.e., all join results are uniformly sampled. The probability to successfully reach a leaf node is $\frac{\mathrm{OUT}}{\mathsf{AGM}(Q)}$, so it takes $O\left(\frac{\mathsf{AGM}(Q)}{\mathrm{OUT}}\right) = O\left(\frac{\mathrm{IN}^\rho}{\mathrm{OUT}}\right)$ attempts in expectation to draw a sample, as desired. Note that when Generic-Join is used as a sampling algorithm, there is no need to materialize the whole tree $\mathcal{T}$; only one root-to-leaf path needs to be explored.

However, to achieve a running time of $O\left(\frac{\mathrm{IN}^\rho}{\mathrm{OUT}}\right)$, we only have constant time for each attempt. This means that we need to perform the sampling of a child $y \in L_t$ in constant time, which poses the two main technical difficulties that we must resolve in the rest of the paper: (1) How to avoid computing L_t, which would take super-constant time, and (2) how to build appropriate weighted sampling data structures so as to sample a y with probability $\mathsf{AGM}(Q_{(t,y)})/\mathsf{AGM}(Q_t)$ in constant time. We resolve these two difficulties in the rest of this section.

2.2 Avoid computing L_t

First, we order the attributes as $\mathcal{V} = \{v_1, \ldots, v_n\}$ in a way such that for any $2 \leq j \leq n$ there is an $1 \leq i \leq j-1$ with $\{v_i, v_j\} \subseteq F$ for some $F \in \mathcal{E}$. Here, we assume that the query is connected; otherwise, we can just sample a result from each connected component $Q_1, \ldots, Q_k$ and return their concatenation. The sampling time would be

$$\frac{\mathsf{AGM}(Q_1)}{|Q_1|} + \cdots + \frac{\mathsf{AGM}(Q_k)}{|Q_k|} \leq \frac{\mathsf{AGM}(Q_1)}{|Q_1|} \cdot \cdots \cdot \frac{\mathsf{AGM}(Q_k)}{|Q_k|} = \frac{\mathsf{AGM}(Q)}{\mathrm{OUT}}.$$

We will also assume that there are no relations of arity 1. Such relations can be easily removed in a preprocessing step: Suppose there is an arity-1 relation $R_{\{v_i\}}$. We just replace every other relation R_F with $R_F \ltimes R_{\{v_i\}}$, and then we remove $R_{\{v_i\}}$.

Suppose t is a tuple on $I = \{v_1, \ldots, v_i\}$. Computing L_t requires computing the set intersection of $\pi_{v_{i+1}}(R_F \ltimes t)$ for $F \in \mathcal{E}_{\{v_{i+1}\}}$, which we cannot afford. Instead, we take one of these sets, chosen as follows:

$$F_t^* = \begin{cases} \arg\min_{F \in \mathcal{E}_{\{v_1\}}} |R_F|, & \text{if } i = 0; \\ \arg\min_{F \in \mathcal{E}_{\{v_{i+1}\}} \cap \mathcal{E}_I} |R_F \ltimes t|, & \text{if } i \geq 1. \end{cases} \tag{5}$$

Then we use

$$L'_t = \pi_{v_{i+1}}(R_{F_t^*} \ltimes t) \tag{6}$$

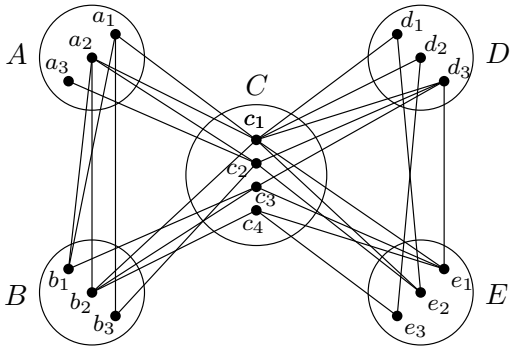


Figure 1 A database instance: The big circles represent attributes, the vertices inside a circle represent values in the domain of that attribute. The edges between two attributes, say, A, B represent tuples in the relation R_{AB}.

instead of L_t as the children of t in $\mathcal{T}$. Note that we always have $L'_t \supseteq L_t$.

Because every L'_t depends on only one particular relation, they are readily available from standard index structures (e.g., hash tables). More precisely, we build an index on each relation R_F, such that for any tuple t, the index returns the list $\pi_v(R_F \ltimes t)$, as well as its size, for any $v \in F$. This list is also known as the *neighbor list* of t in F on v, and its size the *degree* of $\pi_F t$ in F on v. In fact, the Generic-Join algorithm requires exactly the same index. Using these indexes, L'_t can be found in $O(1)$ time for any t on $I = \{v_1, \ldots, v_i\}$. For $i = 0$, F^*_t is a fixed relation, and L'_t is simply $\pi_{v_1} R_{F^*_t}$. For $i \geq 1$, F^*_t is one of the relations in $\mathcal{E}_{\{v_{i+1}\}}$. For any $F \in \mathcal{E}_{\{v_{i+1}\}}$, the neighbor list $\pi_{v_{i+1}}(R_F \ltimes t)$ is available from the index. By comparing their sizes, we can determine F^*_t and L'_t in $O(1)$ time. Finally, it is easy to see that the total size of all the neighbor lists is linear.

2.3 The sampling algorithm

Replacing L_t with L'_t, together with the discussion in Section 2.1, we obtain the Generic-Join-Sample algorithm, as shown in Algorithm 2. In addition, we need to check the validity of t in line 2. This is because we now sample from L'_t, which is a superset of L_t, so the parent call cannot guarantee its validity as in Algorithm 1.

Algorithm 2: Generic-Join-Sample(i, t):

1 // t is a tuple on $I = \{v_1, \ldots, v_i\}$
2 **if** $t \notin \pi_I R_F$ *for any* $F \in \mathcal{E}_I$ **then** reject
3 **if** $i = n$ **then return** t
4 Set F^*_t and L'_t as in (5) and (6)
5 $q_t \leftarrow \sum_{y \in L'_t} \mathsf{AGM}(Q_{(t,y)})$
6 $y \leftarrow$ a random sample from L'_t with probability $\mathsf{AGM}(Q_{(t,y)})/q_t$
7 With probability $q_t/\mathsf{AGM}(Q_t)$ **return** Generic-Join-Sample$(i + 1, (t, y))$
8 **else** reject

Note that $\mathsf{AGM}(Q_t)$ is never zero in line 7. This is because in line 2, we reject t if $|R_F \ltimes t| = 0$ for any $F \in \mathcal{E}_I$, which implies that $|R_F \ltimes t| > 0$ for any $F \in \mathcal{E}_J$ (assuming the input relations are all nonempty).

Except line 5–6, all other operations in Algorithm 2 take $O(1)$ time using the indexes. Before describing how to implement line 5–6 efficiently, we first see an example.

An example

We illustrate the Generic-Join-Sample algorithm on the query $R_{\{A,B\}} \bowtie R_{\{B,C\}} \bowtie R_{\{A,C\}} \bowtie R_{\{C,D\}} \bowtie R_{\{C,E\}} \bowtie R_{\{D,E\}}$, with the database instance shown in Figure 1. For simplicity, we will write e.g. $\{A, B\}$ as AB, then $R_{\{A,B\}}$ is written as R_{AB}. Similarly, we write $x_{AB}, x_{BC}, \ldots$ as the optimal solution to (1). Suppose we order the attributes as A, B, C, D, E. We start the algorithm by calling Generic-Join-Sample$(0, \langle\rangle)$. When $i = 0$, F_t^* is always AB, since $|\pi_{v_1} R_{AB}| = 2$ and $|\pi_{v_1} R_{AC}| = 3$. So $L'_t = \{a_1, a_2\}$. We sample each $y \in L'_t$ with probability

$$\begin{aligned}\frac{\mathsf{AGM}(Q_y)}{\mathsf{AGM}(Q)} &= \frac{|R_{AB} \ltimes y|^{x_{AB}} |R_{AC} \ltimes y|^{x_{AC}} |R_{BC}|^{x_{BC}} |R_{CD}|^{x_{CD}} |R_{CE}|^{x_{CE}} |R_{DE}|^{x_{DE}}}{|R_{AB}|^{x_{AB}} |R_{AC}|^{x_{AC}} |R_{BC}|^{x_{BC}} |R_{CD}|^{x_{CD}} |R_{CE}|^{x_{CE}} |R_{DE}|^{x_{DE}}} \\ &= \frac{|R_{AB} \ltimes y|^{x_{AB}} |R_{AC} \ltimes y|^{x_{AC}}}{|R_{AB}|^{x_{AB}} |R_{AC}|^{x_{AC}}}.\end{aligned}$$

So a_1 and a_2 are sampled with probabilities $(\frac{2}{4})^{x_{AB}} (\frac{1}{4})^{x_{AC}}$ and $(\frac{2}{4})^{x_{AB}} (\frac{2}{4})^{x_{AC}}$, respectively, and reject otherwise. Suppose a_2 is sampled. Then we call Generic-Join-Sample$(1, \langle a_2 \rangle)$. When $i = 1$, F_t^* can only be AB and $L'_t = \{b_1, b_2\}$. Canceling the common terms as above, we sample each $y \in L'_t$ with probability

$$\frac{\mathsf{AGM}(Q_{(t,y)})}{\mathsf{AGM}(Q_t)} = \frac{|R_{BC} \ltimes (t, y)|^{x_{BC}}}{|R_{AB} \ltimes t|^{x_{AB}} |R_{BC}|^{x_{BC}}} = \frac{|R_{BC} \ltimes y|^{x_{BC}}}{|R_{AB} \ltimes t|^{x_{AB}} |R_{BC}|^{x_{BC}}}.$$

So b_1 and b_2 are sampled with probabilities $(\frac{1}{2})^{x_{AB}} (\frac{1}{5})^{x_{BC}}$ and $(\frac{1}{2})^{x_{AB}} (\frac{3}{5})^{x_{BC}}$, respectively. Suppose b_2 is sampled.

Then we call Generic-Join-Sample$(2, \langle a_2, b_2 \rangle)$, which is the most interesting step. When $i = 2$, F_t^* is either AC or BC, depending on t. With $t = \langle a_2, b_2 \rangle$, we take $F_t^* = AC$ and thus $L'_t = \{c_1, c_2\}$. (If we had sampled b_1 from B in the previous step, we would take $F_t^* = BC$ and $L'_t = \{c_3\}$.) Then we sample each $y \in L'_t$ with probability

$$\begin{aligned}\frac{\mathsf{AGM}(Q_{(t,y)})}{\mathsf{AGM}(Q_t)} &= \frac{|R_{CD} \ltimes (t, y)|^{x_{CD}} |R_{CE} \ltimes (t, y)|^{x_{CE}}}{|R_{AC} \ltimes t|^{x_{AC}} |R_{BC} \ltimes t|^{x_{BC}} |R_{CD}|^{x_{CD}} |R_{CE}|^{x_{CE}}} \\ &= \frac{|R_{CD} \ltimes y|^{x_{CD}} |R_{CE} \ltimes y|^{x_{CE}}}{|R_{AC} \ltimes t|^{x_{AC}} |R_{BC} \ltimes t|^{x_{BC}} |R_{CD}|^{x_{CD}} |R_{CE}|^{x_{CE}}}.\end{aligned}$$

So c_1 and c_2 are sampled with probabilities $(\frac{1}{2})^{x_{AC}} (\frac{1}{3})^{x_{BC}} (\frac{3}{5})^{x_{CD}} (\frac{1}{5})^{x_{CE}}$ and $(\frac{1}{2})^{x_{AC}} (\frac{1}{3})^{x_{BC}} (\frac{1}{5})^{x_{CD}} (\frac{1}{5})^{x_{CE}}$, respectively. Note that $\mathsf{AGM}(Q_{\langle a_2, b_2, c_2 \rangle}) \neq 0$, although $\langle a_2, b_2, c_2 \rangle$ is not part of any valid join result. This is because the residual query Q_t is defined (see definition (2)) only over relations containing at least one free variable (i.e., attributes not appearing in t). This is exactly where we depart from Generic-Join: In Generic-Join, c_2 is not in L_t because it does not join with $t = \langle a_2, b_2 \rangle$ in R_{BC}. More precisely, $L_t = \pi_C(R_{AB} \ltimes t) \cap \pi_C(R_{BC} \ltimes t)$, but L'_t is only the smaller of the two sets. In general, L'_t is a superset of L_t, but as argued before, we cannot afford to compute this set intersection during sampling time, so can only sample from L'_t. In particular, this means that c_2 also has a chance to be sampled at this step, but it will be rejected immediately in the next recursive call, in line 2 of Algorithm 2. Note that this line is not needed in the Generic-Join algorithm, because every $y \in L_t$ is guaranteed to join with t in every relation. Although we have avoided computing L'_t, one immediate concern is that whether the sum of the sampling probabilities $\frac{\mathsf{AGM}(Q_{(t,y)})}{\mathsf{AGM}(Q_t)}$ over all $y \in L'_t$ would still be at most 1, as the query decomposition lemma only guarantees so when summed over L_t. We show in the next subsection that this is indeed still the case, which can be actually considered as a stronger version of the query decomposition lemma.

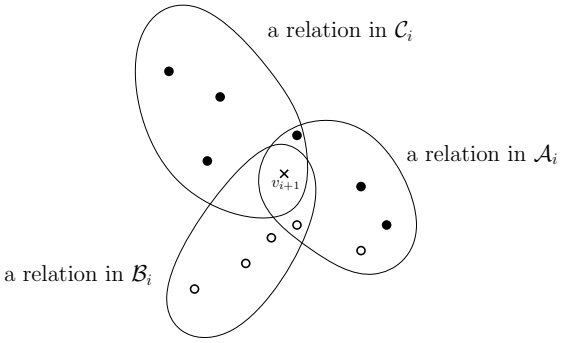


Figure 2 Three types of relations in $\mathcal{E}_{\{v_{i+1}\}}$. The attributes in I are represented as solid disks and attributes in J' are represented as hollow circles.

Let us finish the example. As mentioned above, if c_2 is sampled, it will be rejected in the next step and we will start over. Now suppose c_1 is sampled. Then we move on to Generic-Join-Sample$(3, \langle a_2, b_2, c_1\rangle)$. When $i = 3$, F_t^* can only be CD, and $L'_t = \{d_1, d_2, d_3\}$. Each of them will be sampled with the same probability $(\frac{1}{3})^{x_{CD}}(\frac{1}{3})^{x_{DE}}$. Suppose d_3 is sampled, we move on to Generic-Join-Sample$(4, \langle a_2, b_2, c_1, d_3\rangle)$. Then $F_t^* = DE$, $L'_t = \{e_1\}$. e_1 will be sampled with probability $(\frac{1}{2})^{x_{CE}}(\frac{1}{1})^{x_{DE}}$. Finally, we call Generic-Join-Sample$(5, \langle a_2, b_2, c_1, d_3, e_1\rangle)$, which checks that e_1 joins with c_1 in R_{CE}, and then returns $\langle a_2, b_2, c_1, d_3, e_1\rangle$ as a sampled join result.

2.4 Correctness

In each step of the Generic-Join-Sample algorithm, we sample from some L'_t that is a superset of L_t, so the algorithm can reach every valid join result. In addition, each $y \in L'_t$ is still sampled with probability $\mathsf{AGM}(Q_{(t,y)})/\mathsf{AGM}(Q_t)$, so the uniformity argument in Section 2.1 that every join result is sampled with probability $1/\mathsf{AGM}(Q)$ is not affected. To prove the correctness of the algorithm, it only remains to show that $q_t = \sum_{y\in L'_t} \mathsf{AGM}(Q_{(t,y)}) \le \mathsf{AGM}(Q_t)$, so that line 7 of Algorithm 2 is well defined.

▶ **Lemma 1.** *For any $i = 0, 1, \ldots, n$ and any tuple t on attributes $I = (v_1, v_2, \ldots, v_i)$, let F_t^* and L'_t be defined as in (5) and (6). Then*

$$\sum_{y\in L'_t} \mathsf{AGM}(Q_{(t,y)}) \le \mathsf{AGM}(Q_t). \tag{7}$$

As mentioned, if L'_t is replaced by L_t in (7), this is just the query decomposition lemma. However, since L'_t is a superset of L_t, this requires another proof. Before giving the proof, we first cancel out the common factors on both sides on (7). The remaining factors are all on relations in $\mathcal{E}_{\{v_{i+1}\}}$. Denote $I \cup \{v_{i+1}\}$ as I' and $J \setminus \{v_{i+1}\}$ as J'. We partition $\mathcal{E}_{\{v_{i+1}\}}$ into the following three types:

1. $\mathcal{A}_i = \mathcal{E}_{\{v_{i+1}\}} \cap \mathcal{E}_I \cap \mathcal{E}_{J'}$.
2. $\mathcal{B}_i = \mathcal{E}_{\{v_{i+1}\}} \setminus \mathcal{E}_I$.
3. $\mathcal{C}_i = \mathcal{E}_{\{v_{i+1}\}} \setminus \mathcal{E}_{J'}$.

Please see Figure 2 for an illustration of these three types of relations. Note that $\mathcal{A}_i, \mathcal{B}_i, \mathcal{C}_i$ depend on the particular ordering of the attributes, and these three types of relations will also play an important role in characterizing the class of queries we can sample from efficiently. Now F_t^* can be equivalently defined as $F_t^* = \arg\min_{F\in\mathcal{A}_i\cup\mathcal{C}_i} |\pi_{v_{i+1}}(R_F \ltimes t)|$.

Note that $Q_{(t,y)}$ does not involve any relation in $\mathcal{C}_i$, so (7) can be simplified to the following:

$$\sum_{y\in L'_t}\prod_{F\in\mathcal{A}_i\cup\mathcal{B}_i}|R_F\ltimes(t,y)|^{x_F}\leq\prod_{F\in\mathcal{A}_i\cup\mathcal{B}_i\cup\mathcal{C}_i}|R_F\ltimes t|^{x_F}. \tag{8}$$

Since only the relations in $\mathcal{A}_i,\mathcal{B}_i,\mathcal{C}_i$ are relevant, we introduce the notation

$$\mathsf{AGM}'(Q_{(t,y)})=\prod_{F\in\mathcal{A}_i\cup\mathcal{B}_i}|R_F\ltimes(t,y)|^{x_F},$$

$$\mathsf{AGM}'(Q_t)=\prod_{F\in\mathcal{A}_i\cup\mathcal{B}_i\cup\mathcal{C}_i}|R_F\ltimes t|^{x_F}.$$

Define $q'_t=\sum_{y\in L'_t}\mathsf{AGM}'(Q_{(t,y)})$. Then to prove Lemma 1, we just need to prove

$$q'_t\leq\mathsf{AGM}'(Q_t) \tag{9}$$

for all $i\geq 1$.

Proof. We first consider the case when $i=0$ and $t=\langle\rangle$. In this case, $L_t=\bigcap_{F\in\mathcal{E}_{\{v_1\}}}\pi_{v_1}R_F$. Observe that for any $y\notin L_t$, there is some $F\in\mathcal{E}_{\{v_1\}}$ such that $R_F\ltimes y=\emptyset$. Also, F cannot be $\{v_1\}$ since we assumed that there are no relations of arity 1. Thus, $\mathsf{AGM}(Q_{(t,y)})=\mathsf{AGM}(Q_y)=0$ for any $y\notin L_t$. So $\sum_{y\in L'_t}\mathsf{AGM}(Q_{(t,y)})=\sum_{y\in L_t}\mathsf{AGM}(Q_{(t,y)})\leq\mathsf{AGM}(Q_t)$, following directly from the query decomposition lemma (4). Next we show the case for $i\geq 1$.

In the following discussion, we fix an arbitrary $i\in[1,n]$. If $|R_F\ltimes t|=0$ for some $F\in\mathcal{A}_i\cup\mathcal{B}_i\cup\mathcal{C}_i$, then (7) clearly holds because the both sides of (7) become 0. (In fact, the Generic-Join-Sample algorithm will never reach this case—such a t would be rejected in line 2.) Below, we assume $|R_F\ltimes t|\geq 1$ for every $F\in\mathcal{A}_i\cup\mathcal{B}_i\cup\mathcal{C}_i$.

Let $x_{\mathcal{A}_i\mathcal{B}_i}=\sum_{F\in\mathcal{A}_i\cup\mathcal{B}_i}x_F$ and $x_{\mathcal{C}_i}=\sum_{F\in\mathcal{C}_i}x_F$. Consider the following three cases.

1. $0<x_{\mathcal{A}_i\mathcal{B}_i}<1$. In this case, we have

$$\begin{aligned}q'_t&=\sum_{y\in L'_t}\prod_{F\in\mathcal{A}_i\cup\mathcal{B}_i}|R_F\ltimes(t,y)|^{x_F}\\&=\sum_{y\in L'_t}\left(1\cdot\prod_{F\in\mathcal{A}_i\cup\mathcal{B}_i}|R_F\ltimes(t,y)|^{x_F}\right)\\&\leq\left(\sum_{y\in L'_t}1^{\frac{1}{1-x_{\mathcal{A}_i\mathcal{B}_i}}}\right)^{1-x_{\mathcal{A}_i\mathcal{B}_i}}\cdot\left(\sum_{y\in L'_t}\prod_{F\in\mathcal{A}_i\cup\mathcal{B}_i}|R_F\ltimes(t,y)|^{\frac{x_F}{x_{\mathcal{A}_i\mathcal{B}_i}}}\right)^{x_{\mathcal{A}_i\mathcal{B}_i}},\end{aligned} \tag{10}$$

where the last inequality is due to Hölder's inequality. Applying Hölder's inequality again on the term in the second parentheses, we have

$$\sum_{y\in L'_t}\prod_{F\in\mathcal{A}_i\cup\mathcal{B}_i}|R_F\ltimes(t,y)|^{\frac{x_F}{x_{\mathcal{A}_i\mathcal{B}_i}}}\leq\prod_{F\in\mathcal{A}_i\cup\mathcal{B}_i}\left(\sum_{y\in L'_t}|R_F\ltimes(t,y)|\right)^{\frac{x_F}{x_{\mathcal{A}_i\mathcal{B}_i}}}\leq\prod_{F\in\mathcal{A}_i\cup\mathcal{B}_i}|R_F\ltimes t|^{\frac{x_F}{x_{\mathcal{A}_i\mathcal{B}_i}}}. \tag{11}$$

Meanwhile, the term in the first parentheses of (10) is

$$\sum_{y\in L'_t}1^{\frac{1}{1-x_{\mathcal{A}_i\mathcal{B}_i}}}=|L'_t|=|\pi_{v_{i+1}}(R_{F^*_t}\ltimes t)|. \tag{12}$$

Substituting (11) and (12) into (10), we obtain

$$q'_t \le |\pi_{v_{i+1}}(R_{F_t^*} \ltimes t)|^{1-x_{\mathcal{A}_i\mathcal{B}_i}} \cdot \prod_{F\in\mathcal{A}_i\cup\mathcal{B}_i} |R_F \ltimes t|^{x_F}. \tag{13}$$

Note that when $x_{\mathcal{A}_i\mathcal{B}_i} < 1$, $\mathcal{C}_i$ cannot be empty. By the definition of F_t^*, we have

$$|\pi_{v_{i+1}}(R_{F_t^*} \ltimes t)| \le |\pi_{v_{i+1}}(R_F \ltimes t)| \le |R_F \ltimes t|$$

for any $F \in \mathcal{C}_i$. Also, since v_{i+1} is covered by $\mathcal{A}_i \cup \mathcal{B}_i \cup \mathcal{C}_i$, $x_{\mathcal{A}_i\mathcal{B}_i} + x_{\mathcal{C}_i} \ge 1$. Applying these to (13), we obtain

$$\begin{aligned} q'_t &\le |\pi_{v_{i+1}}(R_{F_t^*} \ltimes t)|^{x_{\mathcal{C}_i}} \cdot \prod_{F\in\mathcal{A}_i\cup\mathcal{B}_i} |R_F \ltimes t|^{x_F} \\ &= \prod_{F\in\mathcal{C}_i} |\pi_{v_{i+1}}(R_{F_t^*} \ltimes t)|^{x_F} \cdot \prod_{F\in\mathcal{A}_i\cup\mathcal{B}_i} |R_F \ltimes t|^{x_F} \\ &\le \prod_{F\in\mathcal{C}_i} |R_F \ltimes t|^{x_F} \cdot \prod_{F\in\mathcal{A}_i\cup\mathcal{B}_i} |R_F \ltimes t|^{x_F} = \mathsf{AGM}'(Q_t). \end{aligned}$$

2. $x_{\mathcal{A}_i\mathcal{B}_i} = 0$. Then $x_{\mathcal{C}_i} \ge 1$. Recall that we define $0^0 = 0$. Thus,

$$q'_t \le \sum_{y\in L'_t} 1 = |R_{F_t^*} \ltimes t| \le |R_{F_t^*} \ltimes t|^{x_{\mathcal{C}_i}} = \prod_{F\in\mathcal{C}_i} |R_{F_t^*} \ltimes t|^{x_F} \le \prod_{F\in\mathcal{C}_i} |R_F \ltimes t|^{x_F}, \tag{14}$$

where the last inequality follows from the definition of F_t^* and the observation that $|\pi_{v_{i+1}}(R_F \ltimes t)| = |R_F \ltimes t|$ for any $F \in \mathcal{C}_i$. If $\mathcal{A}_i \cup \mathcal{B}_i = \emptyset$, then (14) is the same as the RHS of (8). Otherwise, recall that we have assumed $|R_F \ltimes t| \ge 1$ for every $F \in \mathcal{A}_i \cup \mathcal{B}_i$, so (14) must be no more than $\mathsf{AGM}'(Q_t)$.

3. $x_{\mathcal{A}_i\mathcal{B}_i} \ge 1$. In this case, we apply Hölder's inequality directly on q'_t:

$$q'_t \le \prod_{F\in\mathcal{A}_i\cup\mathcal{B}_i} \left(\sum_{y\in L'_t} |R_F \ltimes (t,y)|\right)^{x_F} \le \prod_{F\in\mathcal{A}_i\cup\mathcal{B}_i} |R_F \ltimes t|^{x_F}. \tag{15}$$

If $\mathcal{C}_i = \emptyset$, then (15) is the same as the RHS of (8). Otherwise, recall that we have assumed $|R_F \ltimes t| \ge 1$ for every $F \in \mathcal{C}_i$, so (15) must be no more than $\mathsf{AGM}'(Q_t)$.

◀

2.5 Weighted sampling

It remains to show how to perform the sampling step in line 5–7 of the Generic-Join-Sample algorithm. Recall from Section 2.2 that L'_t is just one of the neighbor lists, which are all available in the index, and we can find the right one in $O(1)$ time during each sampling step. However, since we do weighted sampling, a simple list is not enough. If we can compute all the sampling probabilities in advance, then we can build a weighted sampling data structure [6] on every neighbor list, which can then support drawing a weighted sample in $O(1)$ time when we conduct the sampling. The total preprocessing time will be linear since the total size of all the neighbor lists is linear, and the weighted sampling data structure can also be built in linear time [6].

Lines 5–7 of the Generic-Join-Sample algorithm sample each $y \in L'_t$ with probability $p_{(t,y)} = \frac{\mathsf{AGM}(Q_{(t,y)})}{\mathsf{AGM}(Q_t)}$. Canceling out the common factors on the numerator and the denominator,

this can be simplified as (using the notation $\mathsf{AGM}'(Q_{(t,y)})$, $\mathsf{AGM}'(Q_t)$, and q'_t introduced after Lemma 1)

$$p_{(t,y)} = \frac{\mathsf{AGM}'(Q_{(t,y)})}{\mathsf{AGM}'(Q_t)}.$$

To sample a $y \in L'_t$ in constant time according to $p_{(t,y)}$, we perform rejection sampling in the following two steps:

(1) Sampling: sample a $y \in L'_t$ with probability $\frac{\mathsf{AGM}'(Q_{(t,y)})}{q'_t}$.
(2) Rejection: keep the sample with probability $\frac{q'_t}{\mathsf{AGM}'(Q_t)}$.

Note that $\mathsf{AGM}'(Q_t)$ can be computed in constant time by looking up $|R_F \ltimes t|$ from the index, so it only remains to describe how to do step (1) and compute q'_t. For each i, the corresponding sets $\mathcal{A}_i$, $\mathcal{B}_i$, $\mathcal{C}_i$ must be in the following there cases.

The case with $\mathcal{A}_i = \emptyset$

In this case, $\mathsf{AGM}'(Q_{(t,y)}) = \prod_{F\in\mathcal{B}_i} |R_F \ltimes (t,y)|^{x_F} = \prod_{F\in\mathcal{B}_i} |R_F \ltimes y|^{x_F}$; please also refer to the example in Section 2.3, where we made the same simplification. Note that this does not depend on t. Thus, we can precompute all $\mathsf{AGM}'(Q_{(t,y)})$ and construct a weighted sampling structure on every neighbor list $\pi_{v_{i+1}}(R_F \ltimes t)$ to support sampling step (1) in constant time. We also store the value of q'_t together with the list to do the rejection sampling (2) in constant time.

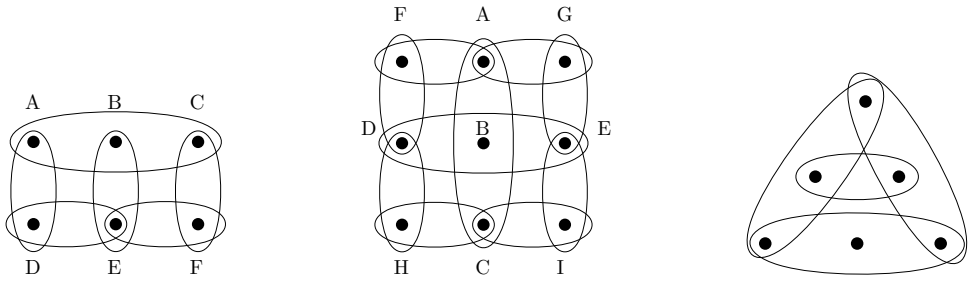
Going back to the example in Section 2.3, to support Generic-Join-Sample$(2,t)$ for any possible t, we precompute $\mathsf{AGM}'(Q_{(t,y)})$ for every $y \in \mathrm{dom}(C)$: $\mathsf{AGM}'(Q_{(t,c_1)}) = 3^{x_{CD}}2^{x_{CE}}$, $\mathsf{AGM}'(Q_{(t,c_2)}) = \mathsf{AGM}'(Q_{(t,c_3)}) = 1^{x_{CD}}1^{x_{CE}}$, $\mathsf{AGM}'(Q_{(y,c_4)}) = 0$. Note that they do not actually depend on t. Using these as weights, we build a weighted sampling structure on each neighbor list of R_{AC} and R_{BC}, i.e., $\pi_C(R_{AC} \ltimes \langle A = a\rangle)$ for every $a \in \mathrm{dom}(A)$ and $\pi_C(R_{BC} \ltimes \langle B = b\rangle)$ for every $b \in \mathrm{dom}(B)$. Note that the total size of these neighbor list is $O(|R_{AC}| + |R_{BC}|)$. Then we store q_t with each neighbor list. For example, with the list $\pi_C(R_{AC} \ltimes \langle A = a_2\rangle)$, we store $p_t = \mathsf{AGM}'(Q_{(t,c_1)}) + \mathsf{AGM}'(Q_{(t,c_2)})$. During the call to Generic-Join-Sample$(2,t)$, we first decide F^*_t, and then use the neighbor list $\pi_C(R_{F^*_t} \ltimes t)$ to perform the sampling.

The case with $|\mathcal{A}_i| = 1$ and $\mathcal{C}_i = \emptyset$

When type-$\mathcal{A}$ relations are present, $\mathsf{AGM}'(Q_{(t,y)})$ will depend on t, which creates complications. However, when there is just one type-$\mathcal{A}$ relation (call it A) and no type-$\mathcal{C}$ relations, then we must have $F^*_t = A$. In this case, we are sampling a $y \in L'_t = \pi_{v_{i+1}}(R_A \ltimes t)$ with weights $\mathsf{AGM}'(Q_{(t,y)}) = |R_A \ltimes (t,y)|^{x_A} \prod_{F\in\mathcal{B}_i} |R_F \ltimes y|^{x_F}$. Although this depends on t, the key observation is that it only depends on the attributes of t that are included in $F^*_t = A$. Thus, we can precompute all the $\mathsf{AGM}'(Q_{(t,y)})$ and construct a weighted sampling structure on each neighbor list $\pi_{v_{i+1}}(R_A \ltimes t)$ to support sampling step (1) in constant time. We also store the value of q_t together with the list to do the rejection sampling (2) in constant time.

Other cases

Unfortunately, for other cases, we do not currently know how to preprocess the weights $\mathsf{AGM}'(Q_{(t,y)})$ in linear time and support constant-time sampling. Nevertheless, we can always compute all the $\mathsf{AGM}'(Q_{(t,y)})$ for $y \in L'_t$ on-the-fly, as well as q_t, which takes $O(|L'_y|) = O(\mathrm{IN})$



(a) Sequenceable in order: A, B, C, D, E, F

(b) Sequenceable in order: A, B, C, D, E, F, G, H, I

(c) Not sequenceable.

Figure 3 Sequenceable and non-sequenceable queries.

time. This means that each sampling attempt will take $O(\text{IN})$ time as opposed to constant time.

We call a query Q *sequenceable* if there is an ordering of the attributes such that for every i, there is either $\mathcal{A}_i = \emptyset$, or $|\mathcal{A}_i| = 1$ and $\mathcal{C}_i = \emptyset$ for every i. Note that if Q is disconnected, Q is sequenceable iff every connected component is sequenceable. We arrive at the main result of this paper:

▶ **Theorem 2.** *Given a sequenceable join query, after a linear-preprocessing step, the Generic-Join-Sample algorithm returns a join result uniformly at random in expected time* $O\left(\frac{\text{IN}^{\rho}}{\text{OUT}}\right)$. *For a non-sequenceable query, it returns a sample in expected time* $O\left(\frac{\text{IN}^{\rho+1}}{\text{OUT}}\right)$.

As a type-$\mathcal{A}$ relation must have at least 3 attributes, any query over binary relations is sequenceable. In addition, Figure 3 shows two sequenceable queries of higher arity, as well as a query that is not sequenceable. Indeed, the definition of sequenceable queries is a rather technical one, which follows from the two cases above that we know how to handle efficiently. Whether more general queries can be handled remains an interesting open problem.

3 Sampling from Join-Project Queries

In this section, we extend our sampling algorithm to *join-project queries* $\pi_O(Q)$ (a.k.a. *conjunctive queries*), where $O \subseteq \mathcal{V}$ is the set of output attributes. Let $\bar{O} = \mathcal{V} - O$. The algorithm consists of two simple steps: (1) Use Generic-Join-Sample to sample a join result t from $Q_O = \bowtie_{F \in \mathcal{E}_O} (\pi_O R_F)$. Note that it is possible that Q_O is disconnected; in which case we take a sample from each connected component, as described at the beginning of Section 2.2. (2) Check if $Q_{\bar{O}}(t) = \bowtie_{F \in \mathcal{E}_{\bar{O}}} (R_F \ltimes t)$ is empty. If not, we return t as a sampled result, otherwise we repeat.

The Correctness of the algorithm is straightforward: Observe that $\pi_O(Q) \subseteq Q_O$. The Generic-Join-Sample algorithm returns a sample t from Q_O uniformly at random. Then we return it iff $t \in \pi_O(Q)$, so every $t \in \pi_O(Q)$ has equal probability to be returned. Next we analyze its running time.

We will assume that Q_O is sequenceable. The analysis for the case when Q_O is not sequenceable is similar. We iterate steps (1) and (2) until a t sampled from Q_O is in $\pi_O(Q)$, which happens with probability $p = \frac{|\pi_O(Q)|}{|Q_O|}$. Let X_i denote the running time of the i-th iteration. Note that $X_i = 0$ if the i-th iteration does not take place. The total expected running time is thus $\sum_{i \geq 1} \mathsf{E}[X_i]$. Conditioned on the i-th iteration taking place, step (1)

takes time $O\left(\frac{\mathsf{AGM}(Q_O)}{|Q_O|}\right) = O\left(\frac{\mathsf{AGM}(Q)}{|Q_O|}\right)$ in expectation. Step (2) takes time $O(\mathsf{AGM}(Q_{\bar{O}}(t))$, using any worst-case optimal join algorithm, e.g., Generic-Join. Because each $t \in Q_O$ is sampled with probability $1/|Q_O|$, the expected running time of step (2) is (the big-Oh of)

$$\sum_{t \in Q_O} \frac{1}{|Q_O|} \cdot \mathsf{AGM}(Q_{\bar{O}}(t)) = \frac{\sum_{t \in Q_O} \mathsf{AGM}(Q_{\bar{O}}(t))}{|Q_O|} \leq \frac{\mathsf{AGM}(Q)}{|Q_O|},$$

where the last inequality follows from the query decomposition lemma. Thus, we have $\mathsf{E}[X_i \mid \text{the } i\text{-th iteration takes place}] = O\left(\frac{\mathsf{AGM}(Q)}{|Q_O|}\right)$. Since the i-th iteration takes place with probability $(1-p)^{i-1}$, we have $\mathsf{E}[X_i] = (1-p)^{i-1} \cdot O\left(\frac{\mathsf{AGM}(Q)}{|Q_O|}\right)$. Thus, the total expected running time is

$$\sum_{i \geq 1} (1-p)^{i-1} \cdot O\left(\frac{\mathsf{AGM}(Q)}{|Q_O|}\right) = \frac{1}{p} \cdot O\left(\frac{\mathsf{AGM}(Q)}{|Q_O|}\right) = \frac{|Q_O|}{|\pi_O(Q)|} \cdot O\left(\frac{\mathsf{AGM}(Q)}{|Q_O|}\right) = \frac{\mathsf{AGM}(Q)}{|\pi_O(Q)|}.$$

▶ **Theorem 3.** *Given a join-project query $\pi_O(Q)$, after a linear-preprocessing step, we can return a query result uniformly at random in expected time $O\left(\frac{\mathrm{IN}^\rho}{\mathrm{OUT}}\right)$ if Q_O is sequenceable, and $O\left(\frac{\mathrm{IN}^{\rho+1}}{\mathrm{OUT}}\right)$ time otherwise.*

Note that Theorem 3 degenerates into Theorem 2 when $O = \mathcal{V}$.

4 Join Size Estimation

Because the Generic-Join-Sample algorithm succeeds in returning a random sample with probability exactly $\frac{\mathrm{OUT}}{\mathrm{IN}^\rho}$, this can be turned into a join size estimation algorithm using standard techniques. More precisely, we simply make k attempts, and see how many of them succeed. Suppose X out of the k attempts succeed, then we return $\mathrm{IN}^\rho \cdot \frac{X}{k}$ as an estimator of OUT.

It is obvious that this estimator is unbiased. Its variance is

$$\left(\frac{\mathrm{IN}^\rho}{k}\right)^2 \mathsf{Var}[X] \leq \left(\frac{\mathrm{IN}^\rho}{k}\right)^2 \cdot k \cdot \frac{\mathrm{OUT}}{\mathrm{IN}^\rho} = \frac{\mathrm{IN}^\rho \cdot \mathrm{OUT}}{k}.$$

To obtain a constant-factor approximation with constant probability, it is sufficient to make this variance smaller than $\mathrm{OUT}^2/4$, and it takes $k = O\left(\frac{\mathrm{IN}^\rho}{\mathrm{OUT}}\right)$ attempts to achieve so. Since each attempt takes $O(1)$ time (assuming Q is sequenceable), this means that we can obtain a constant-factor approximation of OUT in $O\left(\frac{\mathrm{IN}^\rho}{\mathrm{OUT}}\right)$ time.

However, a technical issue is that k depends on OUT, which is exactly the value we want to estimate. In [2], the standard technique of making repeated guesses for OUT by a binary search is used, which results in a logarithmic-factor increase in the running time. For our algorithm, a simpler strategy can be deployed: We simply keep repeating the attempts until c samples have been successfully obtained, where c is some constant. Note that the total running time is still $O\left(\frac{\mathrm{IN}^\rho}{\mathrm{OUT}}\right)$ in expectation. Below, we show that it returns a constant-factor approximation of OUT with constant probability.

Note that in this version of the algorithm, k becomes a random variable. We need to show that with at least constant probability, we stop the algorithm with, say, $\frac{c}{2} \cdot \frac{\mathrm{IN}^\rho}{\mathrm{OUT}} \leq k \leq 2c \cdot \frac{\mathrm{IN}^\rho}{\mathrm{OUT}}$. For $k \geq 2c \cdot \frac{\mathrm{IN}^\rho}{\mathrm{OUT}}$ to happen, we must have collected less than c samples when $2c \cdot \frac{\mathrm{IN}^\rho}{\mathrm{OUT}}$ attempts have been made. We know that in expectation, we should have collected $2c \cdot \frac{\mathrm{IN}^\rho}{\mathrm{OUT}} \cdot \frac{\mathrm{OUT}}{\mathrm{IN}^\rho} = 2c$ so far. As each attempt is independent, we can use the Chernoff inequality to bound

$$\Pr\left[k \geq 2c \cdot \frac{\mathrm{IN}^\rho}{\mathrm{OUT}}\right] \leq e^{-\frac{(\frac{1}{2})^2 \cdot 2c}{2}} = e^{-\frac{c}{4}},$$

which can be made as small as possible by choose c large enough. Using a similar argument, we can show that $k \leq \frac{c}{2} \cdot \frac{\text{IN}^\rho}{\text{OUT}}$ also happens with a constant probability small enough. Then by a union bound, $\frac{c}{2} \cdot \frac{\text{IN}^\rho}{\text{OUT}} \leq k \leq 2c \cdot \frac{\text{IN}^\rho}{\text{OUT}}$ happens with at least a constant probability.

Finally, standard techniques can be applied to boost the accuracy and success probability to achieve an (ϵ, δ) guarantee. We omit the detailed proof of the following result. This result also holds for join-project queries.

▶ **Theorem 4.** *Given a join-project query $\pi_O(Q)$, after a linear-preprocessing step, we can return a $(1+\epsilon)$-approximation of* OUT *with probability at least $1-\delta$. The running time is $O\left(\frac{1}{\epsilon^2}\log\frac{1}{\delta} \cdot \frac{\text{IN}^\rho}{\text{OUT}}\right)$ if Q_O is sequenceable, and $O\left(\frac{1}{\epsilon^2}\log\frac{1}{\delta} \cdot \frac{\text{IN}^{\rho+1}}{\text{OUT}}\right)$ otherwise.*

A more practical algorithm

If the goal is just to estimate the join size, not uniformly sampling, we can further improve the efficiency of the algorithm, by simply omitting rejection step (2) in Section 2.5. This means that a $y \in L'_t$ is always sampled and there is no rejection step in line 7 of the Generic-Join-Sample algorithm (it may still be rejected in line 2 of the next recursive call, though). This results in non-uniform samples, i.e., the sampling probability of a tuple t in each attempt, denoted as p_t, will be different for different t. We can no longer use the simple estimator as above. Instead, after each sampling attempt, we return the following Horvitz-Thompson estimator:

$$\tilde{X} = \begin{cases} \frac{1}{p_t}, & \text{if the attempt successfully returns } t; \\ 0, & \text{otherwise.} \end{cases}$$

Note that p_t can be computed as t is sampled, which is simply the product of the sampling probabilities used in sampling step (1) in Section 2.5. It can be easily shown that $\mathsf{E}[\tilde{X}] = \text{OUT}$. Thus, we repeat the attempts and return the average of the above estimator.

The variance of the estimator is $\mathsf{Var}[\tilde{X}] = \sum_{t \in Q} \frac{1}{p_t} - \text{OUT}^2$. Since skipping the rejection step only increases p_t, we have $p_t \geq \frac{1}{\mathsf{AGM}(Q)}$ for every $t \in Q$. Thus, $\mathsf{Var}[\tilde{X}]$ is always no larger than $O(\text{IN}^\rho \cdot \text{OUT})$, and the same theoretical guarantee from above applies. In practice, $\mathsf{Var}[\tilde{X}]$ can be much smaller, as demonstrated by the experimental results shown in the appendix.

5 Open Questions

The obvious open problem is if it is possible to improve the sampling time to $O(\text{IN}^\rho/\text{OUT})$ for non-sequenceable joins. Another intriguing question is whether the bound $O(\text{IN}^\rho/\text{OUT})$ is optimal. Note that the $\Omega(\text{IN}^\rho/\text{OUT})$ lower bound [2] assumes that the algorithm can only access the database through standard operations, such as sampling a tuple, looking up the degree, sampling a neighbor, etc. Because our algorithm builds auxiliary data structures (still in linear time though), this lower bound does not apply.

References

1 Swarup Acharya, Phillip B Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. Join synopses for approximate query answering. In *Proc. ACM SIGMOD International Conference on Management of Data*, 1999.

2 Sepehr Assadi, Mikhail Kapralov, and Sanjeev Khanna. A simple sublinear-time algorithm for counting arbitrary subgraphs via edge sampling. In *Proc. Innovations in Theoretical Computer Science*, 2019.

3 Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. *SIAM Journal on Computing*, 42(4):1737–1767, 2013.

4 S. K. Bera and A. Chakrabarti. Towards tighter space bounds for counting triangles and other substructures in graph streams. In *Symposium on Theoretical Aspects of Computer Science*, 2017.

5 Andreas Björklund, Rasmus Pagh, Virginia V. Williams, and Uri Zwick. Listing triangles. In *Proc. International Colloquium on Automata, Languages, and Programming*, 2014.

6 P. Bratley, B. L. Fox, and L. E. Schrage. *A Guide to Simulation*. Springer Verlag, 1983.

7 Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. On random sampling over joins. In *Proc. ACM SIGMOD International Conference on Management of Data*, 1999.

8 T. Eden, A. Levi, D. Ron, and C. Seshadhri. Approximately counting triangles in sublinear time. In *Proc. IEEE Symposium on Foundations of Computer Science*, 2015.

9 Talya Eden, Dana Ron, and C. Seshadhri. On approximating the number of k-cliques in sublinear time. In *Proc. ACM Symposium on Theory of Computing*, 2018.

10 G. Gottlob, M. Grohe, N. Musliu, M. Samer, and F. Scarcello. Hypertree decompositions: structure, algorithms, and applications. In *Lecture Notes in Computer Science*, volume 3787, pages 1–15. Springer, 2005.

11 Mahmoud Abo Khamis, Hung Q. Ngo, and Dan Suciu. What do shannon-type inequalities, submodular width, and disjunctive datalog have to do with one another? In *Proc. ACM Symposium on Principles of Database Systems*, 2017.

12 Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. Wander join: Online aggregation via random walks. In *Proc. ACM SIGMOD International Conference on Management of Data*, 2016.

13 Hung Q Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms. In *Proc. ACM Symposium on Principles of Database Systems*, pages 37–48, 2012.

14 Hung Q Ngo, Christopher Ré, and Atri Rudra. Skew strikes back: New developments in the theory of join algorithms. *ACM SIGMOD Record*, 42(4):5–16, 2014.

15 C Seshadhri, Ali Pinar, and Tamara G Kolda. Triadic measures on graphs: the power of wedge sampling. In *Proc. SIAM International Conference on Data Mining*, 2013.

16 Todd Veldhuizen. Leapfrog triejoin: A simple, worst-case optimal join algorithm. In *Proc. International Conference on Database Theory*, 2014.

17 Pinghui Wang, Junzhou Zhao, Xiangliang Zhang, Zhenguo Li, Jiefeng Cheng, John CS Lui, Don Towsley, Jing Tao, and Xiaohong Guan. Moss-5: A fast method of approximating counts of 5-node graphlets in large graphs. *IEEE Transactions on Knowledge and Data Engineering*, 2017.

18 Mihalis Yannakakis. Algorithms for acyclic database schemes. In *Proc. International Conference on Very Large Data Bases*, pages 82–94, 1981.

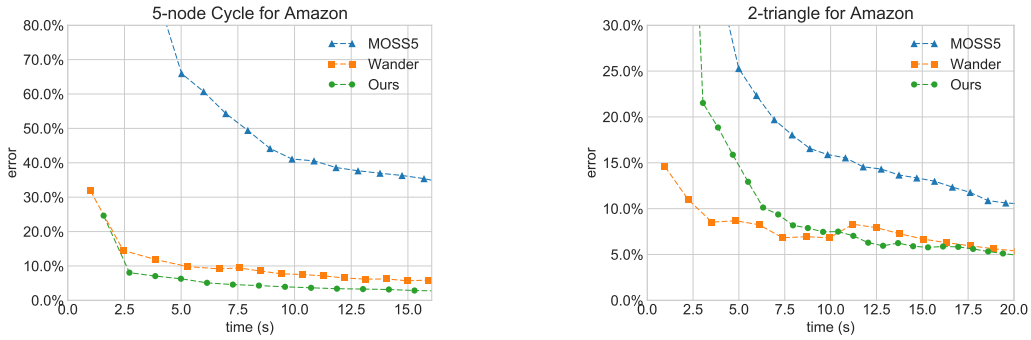
19 Zhuoyue Zhao, Robert Christensen, Feifei Li, Xiao Hu, and Ke Yi. Random sampling over joins revisited. In *Proc. ACM SIGMOD International Conference on Management of Data*, 2018.

A Experiments

After skipping the rejection sampling step, the join size estimation algorithm described above is very practical. Here we report some preliminary experimental results, comparing it with Wander Join [12] and MOSS-5 [17], two best heuristic algorithms for estimating the size of a cyclic join. We have also implemented and tested the other algorithm [2] with the same $O\left(\frac{\mathrm{IN}^\rho}{\mathrm{OUT}}\right)$ running time guarantee, but its performance is far worse than the other three. We used 4 real-world graph data sets and tested 2 cyclic queries as self-joins on each graph: a length-5 cycle and the two-triangle query as shown in Figure 1.

We used a machine with an Intel Xeon E5-2650 v4 2.20GHz CPU and 256G main memory for our experiments. To see how fast the estimator converges, when running an algorithm, we collect the estimated counts reported by the algorithm at regular intervals, say, every 1 second. Then we repeat the algorithm 100 times and compute the relative RMSE (root-mean-square error) to the true count at each time interval. We ran all algorithms in single-thread mode, but all algorithms can be easily parallelized as they all repeatedly take independent samples.

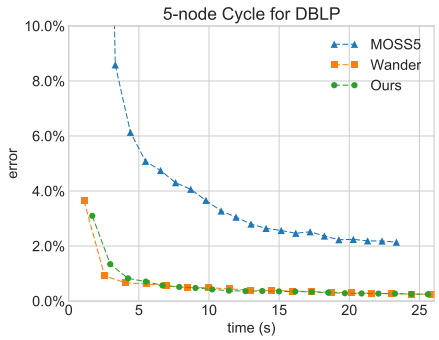
The experimental results are given in Figures 4–7. Note that the preprocessing time is included, which is why the curves do not start from time 0. We see that Wander Join has the shortest preprocessing time, as it only needs hash tables to be built. Our algorithm needs more preprocessing time to build weighted sampling data structures. MOSS-5 needs the longest preprocessing time to prepare the weighted sampling structures for all spanning trees. After preprocessing, the estimates returned by all algorithms converge to the true value as they are all unbiased estimators. The convergence rates are different, though. While Wander Join and MOSS-5 seem to behave differently on different data sets, our algorithm consistently performs on par with the better of the two, probably due to the theoretical guarantee it enjoys. On the other hand, the other $O\left(\frac{\mathrm{IN}^\rho}{\mathrm{OUT}}\right)$ algorithm [2] performs extremely poorly in the experiments, with no reasonable estimates returned after 60 seconds.



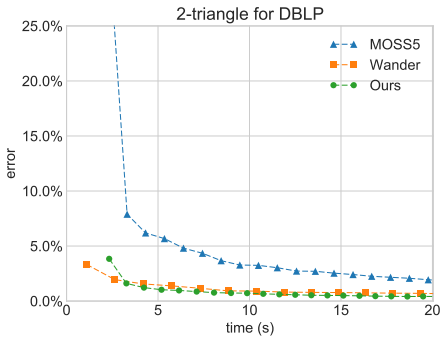
(a) Query: Cycle-5

(b) Query: 2-triangle

Figure 4 Experimental results on `amazon`.

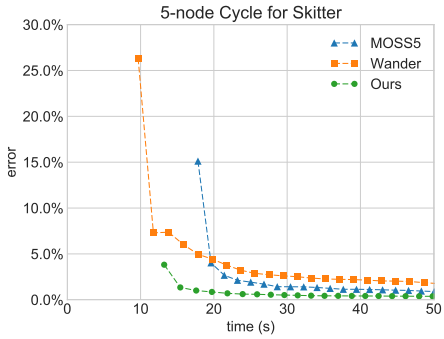


(a) Query: Cycle-5

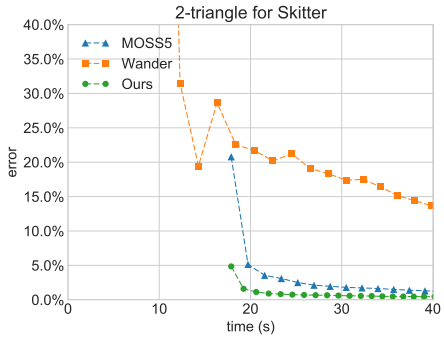


(b) Query: 2-triangle

Figure 5 Experimental results on `dblp`.

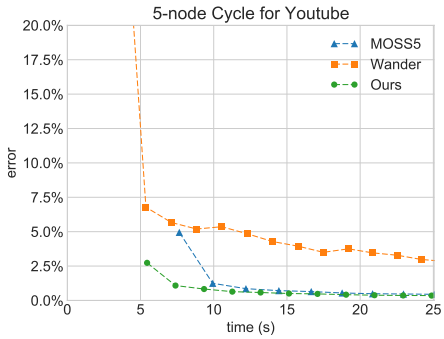


(a) Query: Cycle-5

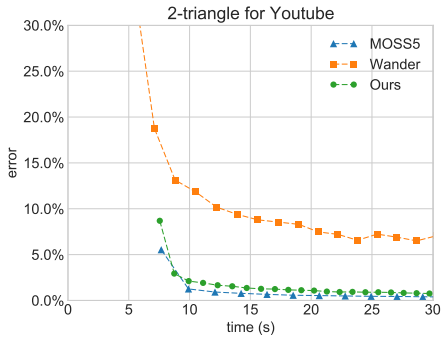


(b) Query: 2-triangle

Figure 6 Experimental results on `skitter`.



(a) Query: Cycle-5



(b) Query: 2-triangle

Figure 7 Experimental results on `youtube`.