Random Sampling over Joins Revisited

Zhuoyue Zhao [1], Robert Christensen [1], Feifei Li [1], Xiao Hu [2], Ke Yi [2]

[1]University of Utah [2]Hong Kong University of Science and Technology

{zyzhao,robertc,lifeifei}@cs.utah.edu {xhuam,yike}@cse.ust.hk

ABSTRACT

Joins are expensive, especially on large data and/or multiple relations. One promising approach in mitigating their high costs is to just return a simple random sample of the full join results, which is sufficient for many tasks. Indeed, in as early as 1999, Chaudhuri et al. posed the problem of sampling over joins as a fundamental challenge in large database systems. They also pointed out a fundamental barrier for this problem, that the sampling operator *cannot* be pushed through a join, i.e., sample($R \bowtie S$) $\neq$ sample(R) $\bowtie$ sample(S). To overcome this barrier, they used precomputed statistics to guide the sampling process, but only showed how this works for two-relation joins.

This paper revisits this classic problem for *both acyclic and cyclic multi-way joins*. We build upon the idea of Chaudhuri et al., but extend it in several nontrivial directions. First, we propose a general framework for random sampling over multi-way joins, which includes the algorithm of Chaudhuri et al. as a special case. Second, we explore several ways to instantiate this framework, depending on what prior information is available about the underlying data, and offer different tradeoffs between sample generation latency and throughput. We analyze the properties of different instantiations and evaluate them against the baseline methods; the results clearly demonstrate the superiority of our new techniques.

ACM Reference Format:

Zhuoyue Zhao [1], Robert Christensen [1], Feifei Li [1], Xiao Hu [2], Ke Yi [2] . 2018. Random Sampling over Joins Revisited. In *Proceedings of Management of Data (SIGMOD '18).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Join is a fundamental operator in relational database systems. Many queries and analytical workloads rely on join operations to link records from two or more relations. Many queries on unstructured or semi-structured data, such as subgraph pattern matching in graph databases, can also be formulated as relational joins. On the other hand, joins are expensive, especially over large amounts of data and/or multiple relations (known as multi-way joins).

However, an important observation is that many applications do not require the full join results. Instead, a random sample of the join results often suffices [2, 8]. Examples include estimating aggregates like `COUNT, SUM, AVG`, more sophisticated analytical tasks like medians and quantiles, kernel density estimation, statistical inference, clustering, regression, classification, etc. Essentially, any analytical task that depends on the data *holistically* can still work with a sample in lieu of the full data[1]. In fact, the entire literature of statistics is about how to effectively make use of a sample and quantify the accuracy of a particular sampling procedure.

Among the many sampling methods, a *simple random sample* is the most widely used and studied in the statistics literature. In a simple random sample of size k, each element in the underlying population is picked *with equal probability*, and the procedure is *repeated k times independently*. One can also distinguish between sampling *with replacement* and *without replacement*. In the latter case, a sampled element is removed from the population after it is sampled, so that the sample must contain k distinct elements.

While simple random sampling is easy when the underlying population can be directly accessed (e.g., stored in an array), it becomes highly nontrivial when it is given implicitly, such as the results of a join. Indeed, at SIGMOD'99, two prominent papers [2, 8] simultaneously posed the problem of random sampling over joins as a fundamental challenge to the community, while offering some partial solutions. Acharya et al. [2] observed that when the join consists of only foreign-key joins that follow a special structure, then the join results map to just one relation, which makes the problem much easier (more details given in Section 2.3).

A fundamental challenge for the problem, as pointed out in [8], is that the sampling operation *cannot* be pushed down through a join operator, i.e., sample(R) $\bowtie$ sample(S) $\neq$ sample($R \bowtie S$). Each joined pair of tuples in sample(R) $\bowtie$ sample(S) is uniformly chosen from the full join results, but there is strong correlation among the pairs. In fact, the series of work on *ripple join* [11, 13, 14, 16] for online aggregation exactly studies how to use a such a *non-independent sample* for estimating simple aggregates like COUNT, SUM, AVG, which is non-trivial and can be costly. Furthermore, it is still not clear how to use non-independent samples for all the other not-so-simple analytical tasks mentioned above. Interestingly, the recent wander join algorithm [22] returns *independent but non-uniform samples* from the join, which again, can only be used for estimating aggregates due to *non-uniformity*.

Another important motivation for us to revisit this classical problem is the recent trend to combine relational operations and machine learning into a unified system, such as Spark [5]. In addition to the obvious benefit of avoiding data transfers between different systems, a greater advantage of a unified system is the potential performance improvement for both the relational processing (in particular, joins) and the machine learning task. Recent works on learning over joins

[1]Notable exceptions include `MAX` and `MIN`.

SIGMOD '18, June 2018, Houston, TX, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00

https://doi.org/10.1145/nnnnnnn.nnnnnnn

[21, 32] present a promising first step towards this direction. Their approach is based on factorized computation, thus only applies to specific machine learning models such as linear regression. On the other hand, we believe that sampling over joins provides a more general solution to this problem. Indeed, there is a well established theory in statistical machine learning that relates the sample size needed to train a model with the model's inherent complexity and expressive power (i.e., VC dimension). Thus, when the training is to be done on the join results, which can be very large, taking a random sample over the join and training the model with the sample can bring great savings for both computing the join and training the model, while incurring a small and bounded loss in accuracy. Note that, however, this theory requires the sample to be uniform and independent, so neither ripple join nor wander join can be used.

To obtain uniform and independent samples, Chaudhuri et al. [8] proposed two methods. The first one, which attributes to Olken [25], rejects samples from sample(R) $\bowtie$ sample(S) with appropriate probabilities. The second method takes samples from R non-uniformly, guided by the statistical information on S. However, they only considered the problem over two-relation joins; extension to joins over 3 or more relations was left as future work but was never finished.

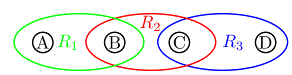
Our contribution. This work revisits this classic yet important problem. We design a general join sampling framework that incorporates both Chaudhuri et al.'s method and Olken's method as special cases, but extend their methods in nontrivial ways to handle *arbitrary multi-way joins, acyclic or cyclic, and with selection predicates*. In doing so, we leverage on new results on join size upper bounds [6, 18, 22], which results in significant improvements in sampling efficiency. Our sampling framework can be instantiated in various ways, depending on what prior information is available about the underlying data, and offers different tradeoffs between sample production latency and throughput. Specifically, this paper makes the following contributions:

- We design a general join sampling framework (Section 3) that can be combined with any join size upper bound method. Any instantiation of the framework *always returns uniform and independent samples from the full join results*, as we will show later in Theorems 3.1, 5.1, and 5.2, but may differ in sampling efficiency.
- We show that existing algorithms on join sampling are special cases of our framework, and present new instantiations of the framework to process multi-way joins (Section 4).
- We show how to extend our framework to process general multi-way joins, acyclic or cyclic, with or without selection predicates (in Section 5).
- We perform extensive experimental evaluations using both TPC-H benchmark and a large social graph data set to investigate the performance of the proposed approach, and compare our methods with other baselines. The results in Section 6 have clearly demonstrated the efficiency and scalability of our approach, and its superiority over other methods.
- We present insights to various instantiations based on their algorithmic properties and empirical performance in Section 7, and a few useful extensions (e.g., condition join, group-by, update) and the limitations of our approach.

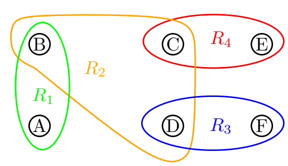
Section 2 provides the background of our study, Section 8 reviews the related works, and the paper is concluded in Section 9.

R_i	An input relation.
$A_i, A, B, C, \ldots$	The attributes.
dom(A)	The domain of attribute A.
t, t', t_i	Tuples.
$\mathscr{A}$	The set of all attributes.
$\mathscr{A}(R_i)$	The attributes of R_i.
$\mathscr{H}$	A join query (represented as a hyper-graph).
$\mathscr{H}(I)$	The instance hyper-graph on instance I.
J	The set of join results.
$d_A(v, R)$	Frequency of v on attribute A in R.
$M_A(R)$	Maximum frequency of any value on A in R.
r_0	The root tuple that joins with all tuples in R_1.
$w(t)$	The join size of t with all relations below.
$w(t, R)$	The join size of t with R and all relations below R.
$W(t), W(t, R)$	Upper bounds on $w(t)$ and $w(t, R)$.
$w(R), W(R)$	$\sum_{t \in R} w(t)$, $\sum_{t \in R} W(t)$.
$R_i^1, R_i^2, \ldots$	The child relations of R_i.
$\mathscr{H}_s$	The skeleton query (must be acyclic).
$\mathscr{H}_r$	The residual query.
M_r	The largest number of tuples in $\mathscr{H}_r$ that can join with any join result in $\mathscr{H}_s$.

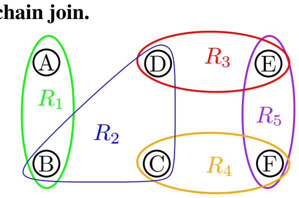
Table 1: Notation used in the paper.



(a) A chain join.



(b) A star join.



(c) A cyclic join.

Figure 1: Several joins represented as hyper-graphs.

2 PRELIMINARIES

2.1 Problem definition: Join as a Hyper-Graph

Let $\mathscr{A}$ be the set of attributes in a database, and let $\mathscr{A}(R_i)$ be the set of attributes in relation R_i. A multi-way join query $R_1 \bowtie \cdots \bowtie R_n$ can be represented as a hyper-graph $\mathscr{H} = (\mathscr{A}, \{\mathscr{A}(R_i), i = 1, \ldots n\})$ where each vertex represents an attribute and each hyper-edge contains all the attributes of a relation. Some visual examples of hyper-graphs for various joins are shown in Figure 1.

For each attribute $A_i \in \mathscr{A}$, let dom(A) be its domain from which values are drawn. We sometimes use the notation such as $R_i(A, B)$ to denote $\mathscr{A}(R_i) = \{A, B\}$. For simplicity of presentation, we assume that any two relations join on at most one common attribute[2].

Each tuple t in R_i assigns each attribute $A \in \mathscr{A}(R_i)$ to a value $a \in$ dom(A). A join result τ is thus any assignment of values to all the attributes in $\mathscr{A}$, such that for each R_i, there exists a tuple $t \in R_i$ that is consistent with τ, i.e., $\pi_{\mathscr{A}(R_i)} \tau = t$. Figure 2 shows an example of the chain join in Figure 1a on a particular instance.

[2] When this is not the case, a "composite" attribute can be introduced to combine multiple attributes into one.

R_1

A	B
1	2
2	2
3	6
4	7

R_2

B	C
2	18
5	18
6	26
6	31
7	32

R_3

C	D
18	101
18	102
26	103
31	104

$R_1 \bowtie R_2 \bowtie R_3$

A	B	C	D
1	2	18	101
1	2	18	102
2	2	18	101
2	2	18	102
3	6	26	103
3	6	31	104

Figure 2: Relations R_1, R_2, R_3 with the join $R_1 \bowtie R_2 \bowtie R_3$.

Given a particular database instance I, we can also view each distinct attribute value as a vertex, and each tuple as a hyper-edge, which contains all the values in this tuple. We call this (hyper-)graph the *instance (hyper-)graph*, denoted $\mathscr{H}(I)$. Thus, for the case of the 3-relation chain join in Figure 2, a join result τ corresponds to a length-3 path in the join graph that starts from some value in dom(A) and ends at some value in dom(D).

For a general join query $\mathscr{H}$, a join result is a subgraph of $\mathscr{H}(I)$ that contains exactly one value in each dom(A_i) for all $A_i \in \mathscr{A}$. *The problem of join sampling* is to return each such subgraph with probability $1/|J|$, where $J = R_1 \bowtie \cdots \bowtie R_n$ represents all join results.

Returning just one sampled result is usually not enough, and one would like to generate sampled results continuously until a certain desired sample size k is reached. Join sampling requires that these samples are totally independent.

2.2 Sampling two-relation joins

We first review the existing algorithms for sampling two-relation joins. Consider $R_1(A,B) \bowtie R_2(B,C)$, where B is the common join attribute of the two relations. For any value $v \in \text{dom}(B)$, let $d_B(v,R_i)$ be the frequency of v on attribute B in R_i, i.e., $d_B(v,R_i) = |t : t \in R_i, \pi_B(t) = v|$. Let $M_B(R_i) = \max_v d_B(v,R_i)$.

The first algorithm was due to Olken [25]. His algorithm first samples a tuple t_1 uniformly from R_1, and then uniformly samples a tuple t_2 from R_2 from all those tuples that join with t_1, i.e., $\pi_B(t_2) = \pi_B(t_1) = v$. Finally, it outputs the join result $t_1 \bowtie t_2$ with probability $d_B(v,R_2)/M_B(R_2)$, and rejects it otherwise.

Olken's algorithm may have a high rejection rate if $M_B(R_2)$ is much larger than typical value frequencies in R_2. The observation of Chaudhuri et al. [8] is that, if the frequency information $d_B(v,R_2)$'s are known, then the sampling efficiency can be greatly improved by sampling a tuple $t_1 \in R_1$ with $\pi_B(t_1) = v$ with *probability proportional* to $d_B(v,R_2)$. After such a tuple $t_1 \in R_1$ is sampled, we then simply randomly pick some $t_2 \in R_2$ that joins with t_1, and return the join result without rejection. In the case that full frequency information is not available, they used a hybrid strategy, which handles high-frequency values (for which frequencies are assumed to be available) with their method, and handles low-frequency values using Olken's method.

Both Olken's algorithm and Chaudhuri et al.'s algorithm can be implemented if indexes are available on the join attribute B. If not, a full scan on both relations is needed.

2.3 Sampling multi-way foreign-key joins

At the same SIGMOD conference with Chaudhuri et al.'s algorithm, Acharya et al. [2] actually studied the problem of sampling over multi-way joins. However, their algorithm only works for *acyclic foreign-key joins with a single source relation*, where R_n is called the *source relation* in [2]. In the hyper-graph terminology, such a join must satisfy the following requirements (possibly after relabeling the relations):

(1) for any $i < j$ with $\mathscr{A}(R_i) \cap \mathscr{A}(R_j) \neq \emptyset$, their common attribute is a primary key in R_i and a foreign key in R_j; and

(2) for any $i \leq n-1$, there exists a $j > i$ such that $\mathscr{A}(R_i) \cap \mathscr{A}(R_j) \neq \emptyset$.

A strong implication of these two requirements is that there is a one-to-one mapping between R_n and the join results, so R_n is called the *source* relation. Hence, sampling from the join results reduces to just sampling from R_n.

For example, the database instance shown in Figure 2 does not meet these requirement, even after any relabeling of the relations. Sampling such a join cannot be reduced to sampling any single relation, and this will be the key challenge to tackle in this paper.

3 A JOIN SAMPLING FRAMEWORK

To tackle the problem of sampling over multi-way joins, we first give a general join sampling framework that incorporates both Olken's algorithm and Chaudhuri et al.'s algorithm as special cases. In Section 4, we show several instantiations of this framework, leveraging the latest join size bounds developed in the literature.

We will first consider chain joins; extensions to other types of joins are given in Section 5. To ease the presentation, we introduce an R_0, which has only a single "root" tuple r_0, and it joins with all tuples in R_1. Furthermore, for each R_i and each tuple $t \in R_i$, define

$$w(t) = |t \bowtie R_{i+1} \bowtie R_{i+2} \bowtie \cdots \bowtie R_n|.$$

Thus, $w(r_0)$ is just the full join size; for any $t \in R_n$, $w(t) = 1$.

Let $W(t)$ be an upper bound on $w(t)$. In fact, all instantiations of this framework only differ in how the $W(t)$'s are computed. The basic idea of this sampling framework is that a tuple t should be sampled using weight $w(t)$. But $w(t)$ is often not available, so we use $W(t)$ as a proxy, but then have to reject the sample with an appropriate probability. To simplify notation, we use the shorthand $w(R) = \sum_{t \in R} w(t)$ and $W(R) = \sum_{t \in R} W(t)$, where R can be any set of tuples. We maintain the following invariants on the $W(t)$'s:

$$W(t) \geq w(t), \quad \text{for any } t; \tag{1}$$

$$W(t) = w(t) = 1, \quad \text{for any } t \in R_n; \tag{2}$$

$$W(t) \geq W(t \ltimes R_{i+1}), \quad \text{for any } t \in R_i, 0 \leq i \leq n-1. \tag{3}$$

Note that the semi-join $t \ltimes R_{i+1}$ represents the set of all tuples in R_{i+1} that join with t. In fact, invariants (2) and (3) imply (1), but we prefer to keeping it for clarity. The algorithm is described below.

Below we show that each join result will be returned by Algorithm 1 with equal probability, under Invariants (1–3). Thus, invoking it multiple times will return a random sample with replacement. If sampling without replacement is desired, we can simply reject a sample if it has already been sampled.

THEOREM 3.1. *On a chain join, Algorithm 1 returns each join result in J with probability $1/W(r_0)$, where $W(r_0) \geq w(r_o) = |J|$ is an upper bound value on the join size before the algorithm runs.*

PROOF. We will show by induction that, for any i, every partial join result $(r_0, t_1, \ldots, t_i) \in R_0 \bowtie R_1 \bowtie \cdots \bowtie R_i$ is sampled with probability $W(t_i)/W(r_0)$. Then, taking $i = n$ and applying Invariant (2) would imply the theorem.

Algorithm 1: Sampling over a chain join

Input: $R_1, \cdots R_n, W(t)$ for t_0 and any $t \in R_i$, $i \in [1,n]$
Output: A tuple sampled from $R_1 \bowtie \cdots \bowtie R_n$ or reject

```
1  t ← r_0;
2  S ← (r_0);
3  for i = 1, ..., n do
4      W'(t) ← W(t);
5      W(t) ← W(t ⋉ R_i);
6      reject with probability 1 − W(t ⋉ R_i)/W'(t) ;
7      t ← a random tuple t' ∈ (t ⋉ R_i) with probability
         W(t')/W(t ⋉ R_i);
8      add t to S;
9  end
10 report S;
```

The base case $i=0$ is trivial. Assuming this holds for i, and we will show that it also holds for $i+1$. Consider any $(r_0, t_1, \ldots, t_{i-1}, t_i) \in R_0 \bowtie R_1 \bowtie \cdots \bowtie R_i$. By the induction hypothesis, we know that $(r_0, t_1, \ldots, t_{i-1})$ is sampled with probability $W(t_{i-1})/W(r_0)$. The i-th step of the algorithm *does not* reject the sample with probability $W(t_{i-1} \ltimes R_i)/W(t_{i-1})$. Conditioned upon not rejecting, t_i is sampled with probability $W(t_i)/W(t_{i-1} \ltimes R_i)$. Thus, the probability that $(r_0, t_1, \ldots, t_i)$ is sampled is

$$\frac{W(t_{i-1})}{W(r_0)} \cdot \frac{W(t_{i-1} \ltimes R_i)}{W(t_{i-1})} \cdot \frac{W(t_i)}{W(t_{i-1} \ltimes R_i)} = \frac{W(t_i)}{W(r_0)}. \qquad \square$$

Note line 5 of Algorithm 1 updates $W(t)$, thus making Invariant (3) tight at t. This has no effect on the current invocation of the algorithm, but will be helpful in future invocations when more samples are needed.

By Theorem 3.1, the sampling efficiency (i.e., the probability that it successfully returns a sample) is $|J|/W(r_0)$, so we want to make $W(r_0)$ as close to $|J|$ as possible. The algorithm is thus *self-improving* in the sense that, each invocation (including those that reject a partial sample) tightens some of the $W(t)$'s. Eventually, when Invariant (3) is tight at all tuples, we would have $W(r_0) = |J|$, hence 100% sampling efficiency.

Remarks. Note that instead of having all $W(t)$'s, for t_0 and any $t \in R_i$ ($i \in [1,n]$), as an input to Algorithm 1, the algorithm can also take any method that is able to derive an upper bound $W(t)$ for any given tuple t to initialize a $W(t)$ on the fly, and it does so only for those t's that are encountered during the sampling process. Regardless of the choice of $W(t)$, Algorithm 1 always returns independent samples as well. The reason is that an accepted sample returned by Algorithm 1 is always uniform regardless of the history of samples. By Bayes' theorem and induction, the joint probability of a set of samples is the product of their individual probabilities, which means the samples are mutually independent.

Both Olken's and Chaudhuri et al.'s algorithms are special cases of Algorithm 1 when it degenerates into the case $n=2$: Olken's method corresponds to setting $W(t_1) = M_B(R_2)$ for all $t_1 \in R_1$, assuming $R_1 = AB$ and $R_2 = BC$ (recall that $W(t_2)$ must be 1 for all $t_2 \in R_2$). Chaudhuri et al.'s algorithm sets $W(t_1) = w(t_1) = |t_1 \ltimes R_2|$ for any $t_1 \in R_1$. Thus, it has 100% sampling efficiency, but needs the full frequency information on the join attribute from R_2.

4 FRAMEWORK INSTANTIATIONS

As mentioned, different instantiations of Algorithm 1 only differ in how the join size upper bounds, namely the $W(t)$'s, are computed. There is a tradeoff between latency and throughput: tighter upper bounds are more costly to set up, but once in place, can generate samples more efficiently. On the other hand, looser upper bounds are easier to compute, but lead to low sampling efficiency (due to potentially higher rejection rates). Olken's algorithm and Chaudhuri et al.'s algorithm actually take the two extremes on this tradeoff. Olken's algorithm has almost zero setup cost by requiring only the max degree information but very low sampling efficiency. Chaudhuri et al.'s algorithm, on the other hand, uses the tightest upper bounds that lead to 100% sampling efficiency, but requires full frequency information on R_2.

In this section, we first show how these two extreme methods generalize to an n-relation chain join, and then explore new methods that strike a better tradeoff between upper bound computation and sampling efficiency (also for n-relation chain join). We discuss how to extend to other types of joins in Section 5.

4.1 Generalizing Olken's algorithm

Generalizing Olken' algorithm in our framework is straightforward. For each relation $R_i(A_i, A_{i+1}), i = 2, \ldots, n$, let $M_{A_i}(R_i) = \max_v d_{A_i}(v, R_i)$, the maximum frequency on attribute A_i. Note that these basic statistical information is often kept by the database already. Then, for $i = 1, \ldots, n-1$, we set $W(t)$'s as follows.

$$W(t) = \prod_{j=i+1}^{n} M_{A_j}(R_j)$$

for all $t \in R_i$, and set $W(r_0) = W(R_1)$. Essentially, we assume that every tuple in R_i joins with $M_{A_{i+1}}(R_{i+1})$ tuples in R_{i+1}. Invariants (1–3) can be easily verified. We will later refer to the generalized Olken's algorithm as Extended Olken (EO) algorithm.

Note that when instantiating Algorithm 1, there is no need to explicitly store all the $W(t)$'s. In the case of Olken's instantiation, all tuples in one relation share the same initial value. Only when Algorithm 1 makes improvements over some of the $W(t)$'s do we need to maintain them explicitly.

4.2 Generalizing Chaudhuri et al.'s algorithm

Chaudhuri et al.'s algorithm sets $W(t) = w(t)$ for all t. In the case of a 2-relation join, $w(t)$ is simply the value frequency in R_2. For an n-relation chain join, each $w(t)$ is a sub-join size. To compute all the $w(t)$'s, we recall the (hyper-)graph representation of a join and the instance graph $\mathcal{H}(I)$. Our observation is that, for a tuple $t \in R_i$, the partial join $t \bowtie R_{i+1} \bowtie \cdots \bowtie R_n$ is just the set of paths starting from the edge t going all the way to R_n, and $w(t)$ is the number of such paths.

While enumerating all the paths (i.e., computing the full join) can be expensive, computing all the counts can be done using dynamic programming in linear time $O(|R_1| + \cdots + |R_n|)$. This dynamic programming formulation is based on *backward tracing* from R_n to R_1 using Invariants (2) and (3), except that $\geq$ is replaced with $=$ in Invariant (3). Hence, its cost is linear to the total size of all relations. We will later refer to the generalized Chaudhuri et al.'s algorithm as Exact Weight (EW) algorithm.

4.3 Other join size upper bounds

There is a revitalized interest in the database theory community in upper bounding a join size. The generality of our sampling framework allows any of them to be used.

The most well-known result is the AGM bound [6]. On a general query, it requires solving a linear program and we refer the readers to [6] and the better-written survey paper [24] for details. However, in the case of chain joins, the AGM bound is simple and we include it here for completeness:

$$|R_1 \bowtie \cdots \bowtie R_n| \le \min_{I:\{1,n\}\subseteq I, I\cap\{i,i+1\}\neq\emptyset, i=2,\ldots,n-2} \prod_{i\in I} |R_i|.$$

To plug the AGM bound into the sampling framework, we initialize each $W(t) = \mathrm{AGM}(R_{i+1} \bowtie \cdots \bowtie R_n)$ for all $t \in R_i$. The *query decomposition lemma* [24] ensures that Invariant (3) is established using the AGM bounds to initialize the $W(t)$'s.

Note that the AGM bound and Olken's bound are in general not comparable. If the maximum frequencies are small, then the Olken's bound may be smaller. However, in the worst case, the maximum frequency can be as large as the relation size, i.e., $M_A(R_i) = |R_i|$, and Olken's bound can be much larger (almost quadratically) than the AGM bound. In fact, the AGM bound is the optimal bound on the join size when only the relation sizes are given.

Based on the observation above, another natural idea is to handle the frequent and infrequent values separately, if partial frequency information is available. For example, the heavy hitters may have already been collected by the database system. We illustrate this idea on bounding the size of $R_1(A_1,A_2) \bowtie R_2(A_2,A_3) \bowtie R_3(A_3,A_4)$. For a threshold h, we say a tuple t in R_i is *heavy* if $|t' : t' \in R_i, \pi_{A_i}(t') = \pi_{A_i}(t)| \ge h$, otherwise *light*. Let R_i^H be the set of heavy tuples in R_i, and R_i^L the set of light tuples. The join $R_1 \bowtie R_2 \bowtie R_3$ can be decomposed into 4 subjoins:

$$R_1 \bowtie R_2^{X_2} \bowtie R_3^{X_3},$$

where each X_i can be either H or L. We upper bound the size of each subjoin as follows. (1) For $R_1 \bowtie R_2 \bowtie R_3^H$ (which actually includes 2 of the 4 subjoins), we use the AGM bound, i.e., $|R_1|\cdot|R_3^H|$. (2) For $R_1 \bowtie R_2^H \bowtie R_3^L$, we bound it as $|R_1|\cdot|R_2^H|\cdot h$. (3) For $R_1 \bowtie R_2^L \bowtie R_3^L$, we bound it as $|R_1|\cdot h^2$. (Note that we leverage the generalized Olken's algorithm in the latter two cases).

As a concrete example, suppose $|R_1| = |R_2| = |R_3| = 10^4$. In both R_2 and R_3, there is one frequent value that appears in 100 tuples, while the rest of values each appear in less than 10 tuples. So we have $h = 10, |R_2^H| = |R_3^H| = 100$. Then Olken's bound is $|R_1|\cdot 100 \cdot 100 = 10^8$. The AGM bound is $|R_1|\cdot|R_3| = 10^8$. The improved bound is

$$\begin{aligned}&|R_1|\cdot|R_3^H| + |R_1|\cdot|R_2^H|\cdot h + |R_1|\cdot h^2\\ =&10^6 + 10^7 + 10^6.\end{aligned}$$

4.4 Wander join as initialization

Recently, Li et al. [22] proposed a random walk based algorithm for join size estimation. To estimate $|J|$, their algorithm performs a number of random walks on $\mathscr{H}(I)$ from r_0 to R_n. These random walks generate non-uniform but independent samples on J, and still can be used for join size estimation. Our observation is that, each random walk starting from r_0 not only can be used to estimate $|J| = w(r_0)$, but also each intermediate join size $w(t) = |t \bowtie R_{i+1} \bowtie \cdots \bowtie R_n|$ where $t \in R_i$ is a tuple on the path of this random walk.

To improve upon Olken's loose bounds but also avoid the high cost of running the full dynamic programming as in the generalized Chaudhuri's algorithm, we start by performing a number of random walks starting from r_0. Since the estimation on $w(t)$ is based on the central limit theorem, which requires a minimum sample size to hold, these random walks will allow us to obtain estimates on $w(t)$ only for a subset of tuples *with sufficient number of walks*.

More precisely, we set a threshold of θ, and for any tuple t such that at least θ random walks have passed through, we compute a confidence interval on $w(t)$, and set $W(t)$ to be the upper bound of this confidence interval using the wander join estimator [22]. Since these upper bounds are computed probabilistically, there is a small chance that Invariant (3) may not hold for some t's. In this case we raise $W(t)$ in a bottom-up fashion, so as to restore Invariant (3) for all the $W(t)$'s. For any tuple t that has not seen enough random walks, we set $W(t) =$ *unknown*.

Next, we start executing Algorithm 1. Whenever we reach a tuple t such that there are some $t' \in t \ltimes R_i$ with $W(t') =$ *unknown*, we run the dynamic programming algorithm to compute $W(t') = w(t')$ for each such t'. Note that here, we will run the dynamic program top-down, computing and memorizing only those $W(u)$'s that are needed for computing $w(t')$. Factually, they are exactly those tuples from $R_{i+1}, \cdots, R_n$ that participate in the join $t' \bowtie R_{i+1} \bowtie \cdots \bowtie R_n$. The cost of this DP exploration is linear to the number of such tuples (which is typically much smaller than $w(t')$).

Another observation is that Algorithm 1 also performs random walks from r_0 to find samples, though it is guided by the $W(t)$'s. Nevertheless, they still form random walks and the estimator provided in [22] can still be used over these random walks, so that as Algorithm 1 returns samples, they can still continuously contribute to the shrinking of the confidence intervals at all tuples that the random walks have passed (which leads to tighter upper bounds $W(t)$'s as more samples being returned).

5 OTHER TYPES OF JOINS

5.1 Acyclic join queries

In this section, we show how to generalize our sampling algorithm to acyclic join queries. We organize the relations in a tree structure such that R_1 is the root. As before, we also add an R_0 which contains only one tuple r_0 that joins with every tuple in R_1. For every non-leaf relation R_i, let $R_i^1, R_i^2, \ldots$ be the child relations of R_i. We use the notation $R_i \prec R_j$ to denote that R_i is an ancestor of R_j in the tree.

For any R_i and any tuple $t \in R_i$, the definition of $w(t)$ changes to

$$w(t) = |t \bowtie (\bowtie_{j:R_i \prec R_j} R_j)|.$$

If R_i is not a leaf relation, we define $w(t, R_i^k)$ for a child R_i^k of R_i:

$$w(t, R_i^k) = |t \bowtie R_i^k \bowtie (\bowtie_{j:R_i^k \prec R_j} R_j)|.$$

The basic idea in extending our sampling algorithm from chain joins to acyclic queries is that, whenever we reach a relation that has more than one child below (e.g., R_2 in Figure 1b), we branch the sampling process into the two subtrees. This means that a tuple in R_i might have multiple children in the R_i^k's, which explains why we also need to introduce $w(t, R_i^k)$, which is the subjoin size with t but only restricted to one of its subtrees.

Similarly, we introduce join size upper bounds $W(t)$ and $W(t, R_i^k)$ to assist sampling. Invariants (1–3) now become:

$$W(t) \ge w(t), \quad \forall t; \tag{4}$$

$$W(t, R_i^k) \geq w(t, R_i^k), \quad \forall t \in R_i, R_i \text{ is a non-leaf;} \tag{5}$$

$$W(t) = 1, \quad \forall t \in R_i, R_i \text{ is a leaf;} \tag{6}$$

$$W(t) \geq \prod_k W(t, R_i^k), \quad \forall t \in R_i, R_i \text{ is a non-leaf;} \tag{7}$$

$$W(t, R_i^k) \geq W(t \ltimes R_i^k), \quad \forall t \in R_i, R_i \text{ is a non-leaf.} \tag{8}$$

While 4 of the 5 invariants are natural extensions from chain joins, invariant (7) is new. Intuitively, since different branches are independent, any join result from one branch can be combined with any join result from another branch via t, resulting in the multiplication of the join sizes from different branches.

Algorithm 2: ACYCLIC-SAMPLE(t, R_i)

1 $W'(t, R_i) \leftarrow W(t, R_i)$;
2 $W(t, R_i) \leftarrow W(t \ltimes R_i)$;
3 reject with probability $1 - W(t \ltimes R_i)/W'(t, R_i)$;
4 $t \leftarrow$ a random tuple $t' \in (t \ltimes R_i)$ with probability $W(t')/W(t \ltimes R_i)$;
5 $W'(t) \leftarrow W(t)$;
6 $W(t) \leftarrow \prod_k W(t, R_i^k)$;
7 reject with probability $1 - \left(\prod_k W(t, R_i^k)\right)/W'(t)$;
8 add t to S;
9 **if** *R_i is a leaf* **then return**;
10 **foreach** *child relation R_i^k of R_i* **do**
11 ACYCLIC-SAMPLE(t, R_i^k);
12 **end**

Algorithm 1 now becomes a recursive algorithm as shown in Algorithm 2, with the initial call being ACYCLIC-SAMPLE(r_0, R_1). It reports S as a sample upon termination, unless the sample is rejected. Similar to the chain join case, line 2 and line 6 in Algorithm 2 do not affect the correctness, but iteratively tighten the join size upper bounds, hence improving the sampling efficiency over time. The following theorem establishes its correctness, and its proof is found in Appendix A.

THEOREM 5.1. *On any acyclic join, Algorithm 2 returns each join result in J with probability $1/W(r_0)$, where $W(r_0)$ is the value before the algorithm runs.*

To initialize the join size upper bounds, all methods discussed in Section 4 can be carried over, though some care has to be taken. In particular, the AGM bound for an acyclic query is not as clean as the chain join case; please refer to [6] for details. Olken's upper bounds remain as the product of the maximum frequencies in each of the relations in a subtree. Chaudhuri et al.'s instantiation requires exact join sizes, which can still be computed using dynamic programming. In fact, the dynamic programming recurrence is the same backward-tracing idea as that in chain join, but now uses invariants (6-8) (replacing $\leq$ with $=$ in (7) and (8)).

5.2 Cyclic queries

In this section, we extend our sampling algorithm to cyclic queries. We break up all the cycles in the query hyper-graph by removing a subset of relations so that the query becomes a connected, acyclic query. For example, for the cyclic query in Figure 1c, one may remove any one of R_3, R_4, or R_5. We call the removed relations the *residual query*, denoted $\mathscr{H}_r$, and the main acyclic query the *skeleton query*, denoted $\mathscr{H}_s$. Note that for very complicated cyclic queries, e.g., when the query graph is a complete graph with ≥ 5 attributes and ≥ 10 relations, the residual query may be larger than the skeleton query. But for most queries that arise in practice, the residual query tends to be small. We define:

$$M_r = \max_{v_i \in \mathrm{dom}(A_i)} |t : t \in \mathscr{H}_r, \pi_{A_i}(t) = v_i, \text{for all } A_i \in \mathscr{A}(\mathscr{H}_s) \cap \mathscr{A}(\mathscr{H}_r)|,$$

i.e., the largest number of tuples in $\mathscr{H}_r$ once their common attributes with A_s have been fixed to particular values. Note that this is equivalently the largest number of tuples in $\mathscr{H}_r$ that can join with any join result from $\mathscr{H}_s$. In some cases (e.g., the triangle query below), the value of M_r can be trivially obtained. Otherwise, we can evaluate the residual query $\mathscr{H}_r$ in full to compute M_r. The sampling algorithm for cyclic queries is then given below.

Algorithm 3: CYCLIC-SAMPLE$(\mathscr{H}_s, \mathscr{H}_r)$

$t \leftarrow$ a sample from $\mathscr{H}_s$ using Algorithm 2;
reject with probability $1 - |t \ltimes \mathscr{H}_r|/M_r$;
return a uniform sample from $t \ltimes \mathscr{H}_r$;

THEOREM 5.2. *Algorithm 3 returns each join result in J with probability $\frac{1}{M_r \cdot |\mathscr{H}_s|}$, conditioned on a uniform sample being obtained from $\mathscr{H}_s$.*

PROOF. Consider any join result of the whole query. Those tuples in $\mathscr{H}_s$, using the conditionality, is sampled with probability $1/|\mathscr{H}_s|$. Then the algorithm decides if the sampling will continue with probability $|t \ltimes \mathscr{H}_r|/M_r$, and if so, samples remaining tuples with probability $|t \ltimes \mathscr{H}_r|$. Thus, the overall sampling probability is

$$\frac{1}{|\mathscr{H}_s|} \cdot \frac{|t \ltimes \mathscr{H}_r|}{M_r} \cdot \frac{1}{|t \ltimes \mathscr{H}_r|} = \frac{1}{M_r \cdot |\mathscr{H}_s|}. \qquad \square$$

An example. We illustrate the algorithm on the simplest cyclic join, the *triangle query* $R_1(A, B) \bowtie R_2(B, C) \bowtie R_3(A, C)$. We break the query into $\mathscr{H}_s = R_1 \bowtie R_2$ and $\mathscr{H}_r = R_3$. Note that in this case, $M_r = 1$ trivially, assuming the relational algebra semantics that a relation cannot contain duplicated tuples. This might not be the case if R_3 contained another attribute, say D, or bag semantics is used. If so, we can scan R_3 once to compute M_r.

For the skeleton query, if we use Chaudhuri et al.'s instantiation, the algorithm will first sample a tuple t_1 in R_1 with probability proportional to the number of tuples it joins in R_2, and then pick one of these tuples in R_2 uniformly at random, say t_2. Then, the algorithm computes $(t_1, t_2) \ltimes R_3$, and finds the only tuple $t_3 \in R_3$ that joins with both t_1 and t_2, if it exists. Since $M = 1$ in this case, the algorithm simply returns (t_1, t_2, t_3) if t_3 exists, and rejects otherwise.

Remarks. Interestingly, on a triangle query, our algorithm coincides with the *wedge sampling* algorithm [33], which is one of the best algorithms for sampling triangles from large graphs. Similarly, our algorithm degenerates into the *path sampling* algorithm [17] on sampling subgraph patterns with 4 vertices. Note that such a subgraph pattern query is a special case of the multi-way join problem, where there are 4 attributes in total and each relation contains exactly two attributes. *Essentially, our algorithm generalizes these graph sampling algorithms to arbitrary hyper-graphs.*

Another issue is how to decompose the query into the skeleton query and the residual query. This does not affect the correctness of the algorithm, but decides its sampling efficiency. Theorem 5.2 suggests that we should decompose the query such that $M_r \cdot |\mathscr{H}_s|$ is minimized. We can always compute M_r by evaluating the residual query, which is usually small, but computing the join size of $\mathscr{H}_s$ can be expensive. However, an estimated join size of $\mathscr{H}_s$ is sufficient for making a good enough decision (as in query optimization), and we can use any existing join size estimation technique, depending on whether indexes and/or prior statistics are available, for example, using wander join [22].

5.3 Selection predicates

There are two ways to support selection predicates in our sampling algorithms. First, we can simply push the predicates down to the relations. More precisely, we filter each relation with the predicates (by scan or index probe) and feed the filtered relations to our sampling algorithm. In fact, for highly selective predicates, this may still be the best choice. In particular, in the version of our algorithm where we compute $W(t) = w(t)$ as the exact subjoin sizes using dynamic programming, since we need to spend linear time in the input size to set up the sampling framework, it is always beneficial to first filter the relations to reduce their sizes before running the dynamic program.

In other instantiations where we aim at lower setup costs, or when the selection predicates are not very selective, it is more effective to enforce the selection predicates during the sampling process. More precisely, we replace every $t \ltimes R_i$ with $\sigma_\varphi(t \ltimes R_i)$ in Algorithm 1 and 2, where φ is the selection predicate on $\mathscr{A}(R_i)$, if any. Note that there is no need to change Algorithm 3 since the skeleton query already includes all the attributes.

6 EXPERIMENTS

The objective of this work is to design scalable and efficient methods to generate a simple random sample from a join. Once a simple random sample is obtained from a dataset (in our case, the join results), many different and useful analytical tasks can be carried out using the set of samples obtained. It is possible to design specific methods to approximate certain specific analytical functions *without using a simple random sample* (e.g., ripple join [14] or wander join [22] for online aggregations for SUM and COUNT), but these methods cannot be generalized to more sophisticated tasks such as regression and classification. A simple random sample, on the other hand, is much more broadly applicable as shown in Section 1. Hence, the focus of our experimental evaluation is to understand the performance of different methods in producing *uniform and independent samples* from various joins, rather than how one may use such a simple random sample for different analytical purposes, which are well understood subjects in many different domains and not the focus of our study.

6.1 Experimental setup

We have instantiated the proposed sampling framework with a few alternatives as presented in the paper. In particular, we have explored the following methods:

- Full Join (FJ): the baseline method of computing the full join results. Since we focus on natural joins (equi-joins), we use hash join as the underlying method (using the optimized query plan returned by PostgreSQL). The times for full join we report below only include the time for PostgreSQL to run the join to completion. We also tested FJ on a commercial database system, and the trend is similar.
- Extended Olken (EO): the hybrid approach, that was introduced in Section 4.3, which combines the AGM bound and *the generalization of the original Olken's algorithm* [25].
- Exact Weight (EW): the dynamic programming approach that computes the exact weight $w(t)$'s for every tuple in a join hyper-graph instance. This is *a generalization of the original Chaudhuri el al.'s algorithm* [8], as discussed in Section 4.2 for chain joins and in Section 5.1 for arbitrary joins.
- Online Exploration (OE): the combination of using the online aggregation (e.g., via wander join) for tuples with sufficient number of walks, and the on-demand exploration of a join subtree rooted a tuple t in the join hyper-graph instance via EW (only for that subtree), as introduced in Section 4.4.
- Reverse Sampling (RS): the method that is *only useful for multi-way foreign-key joins* introduced by [2] and reviewed in Section 2.3, which does sampling from only R_n and traverse backwards to R_1, using a sampled record from R_n, to produce a sample of the join.

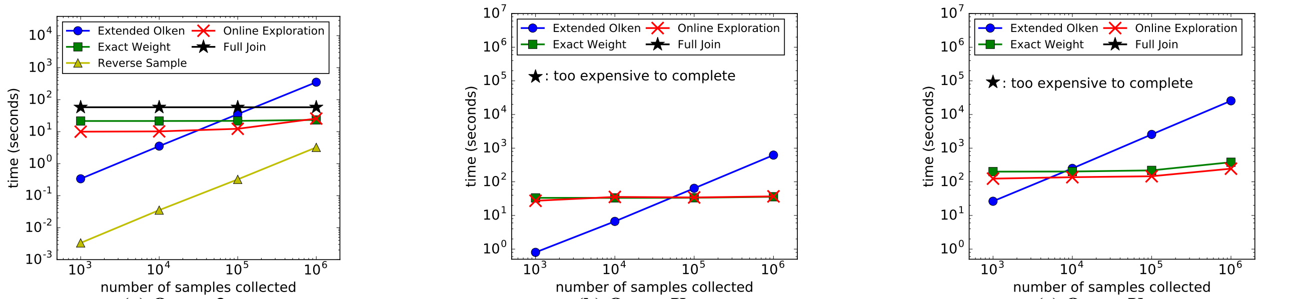
For all methods, we report the total running time to retrieve a specific number of samples. The total running time does not include the building times of all the indexes such as B-Trees, which is a common cost of all methods, since they all require indexes on the join attributes.

Note that even though we only discussed the online exploration method for multi-way chain joins, since wander join works for any join topology [22], same idea can be easily used for any join queries, by simply using the updated version of EW as discussed in Section 5.1 to explore a join subtree on demand. Furthermore, while random samples are continuously returned by our sampling framework, the explored path for producing that sample can be viewed as a random walk, which increases the counts on the number of random walks for any tuple on that path, and improves the wander join estimates for the subtree joins rooted at those tuples.

Lastly, as shown in Section 5.2 and Section 5.3, any method instantiated from our join sampling framework, i.e., EO, EW, and OE, can be made to support cyclic queries and selection predicates easily. Note that there are many ways of "breaking" a cycle by removing a relation and producing the skeleton query $\mathscr{H}_s$ and the residual query $\mathscr{H}_r$ respectively. We followed the discussion at the end of Section 5.2 to optimize our choice and select the best strategy to cut a cycle. This optimization *is included in the cost of EO, EW, and OE* when they are applied for cyclic join queries.

We implemented all methods in C++ and performed experiments on a machine running Ubuntu 14.04 with E5-2609 processor at 2.4GHz. All experiments were performed with data completely residing in memory; disk IOs only occurred during data loading.

TPC-H Dataset. Our first data set consist of data generated using the standard TPC-H benchmark. We modified the generator so that it can add some skewness to the join degrees of the data, by generating the number of lineitems per order according to a Zipfian(1) distribution. By default we use a scale factor of 10 (roughly 10GB), but adjust the scale factor from 1 to 40 in our experiments to show how the algorithms perform and scale with various sizes of data. We explore three join queries over TPC-H data. Appendix B shows the three queries in SQL.



(a) Query 3. **(b) Query X.** **(c) Query Y.**

Figure 3: TPC-H sampling collection time for scale factor 10.

- Query 3 (Q3): is a *multi-way foreign key chain join* that joins tables *customer*, *orders*, and *lineitem* on the primary and foreign keys between them. The cardinality of this join is exactly the size of the *lineitem* table.
- Query X (QX): is a *general multi-way chain join* which joins *nation*, *supplier*, *customer*, *orders*, and *lineitem*. It is a *chain-join*, but is no longer a foreign-key join, hence, the cardinality of this chain join is much larger then any of the tables.
- Query Y (QY): is a *multi-way cyclic join query* that joins *supplier*, *customer1*, *orders1*, *lineitem1*, *lineitem2*, *orders2*, *customer2*, which joins back to the *supplier* table. The join size of this cyclic join query is also very large compared to the size of the input tables.

Social graph data. The second data set is a collection of twitter friendship links and user profiles used in [7]. The data set contains a "celebrity" table with those twitter users with more $> 10,000$ followers, denoted as the *popular-user*, and another table with all twitter users in the data set, denoted as the *twitter-user*. A record on each table represents a friendship link that has a user id as source and another user id as destination. We performed the following three queries on this data set:

- Query T (QT): a *triangle join* between popular-user, twitter-user, and twitter-user.
- Query S (QS): a *square join* between popular-user, twitter-user, twitter-user, and twitter-user.
- Query F (QF): a *snowflake join* between two popular-user tables, and two twitter-user tables, which results in a *tree topology* for the underlying join hyper-graph.

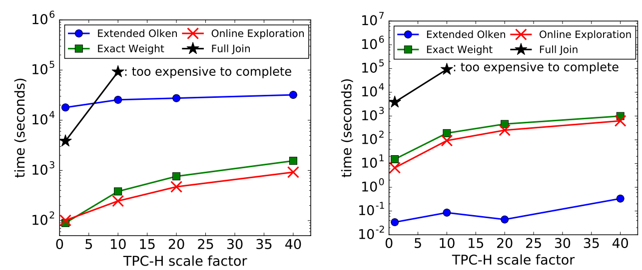
The popular-user table has 165 million records in 3.1GB, and the twitter-user table has 1 billion records in 19GB. We have also down-sampled the twitter-user table to get different sizes of this table for scalability tests. Appendix C shows the queries in SQL.

6.2 TPC-H experiments

Sampling efficiency. We run each of the algorithms over the TPC-H data set with the default scale factor of 10 (roughly 10GB), to test how fast they can collect 10^3, 10^4, 10^5, and 10^6 samples. For Q3 we also apply the reverse sampling method (RS), which cannot be applied to QX and QY. The results of this experiment is shown in Figure 3. Note that both the x-axis and y-axis are in logarithmic scale. Since Q3 is a simple multi-way foreign-key chain join, as expected, in this case RS has the best performance as shown in Figure 3a. Also because that it it a simple multi-way foreign-key chain join, FJ is fairly efficient too. Nevertheless, both EW and OE show good performance, and EO is efficient for producing smaller number of samples, but as the required number of samples continues to rise, its low sampling efficiency (due to high rejection rate) leads to much more cost than other methods.

QX is a general multi-way chain join (i.e., some join attributes are no longer on a primary-key, foreign-key relationship), hence, the join size is much larger than the input relation sizes. As a result, FJ becomes too expensive to complete in a day, as shown in Figure 3b. EO exhibits a similar pattern as that in Q3 experiment. OE and EW in this case show a similar performance; both are able to produce 10^6 samples in 35 seconds.

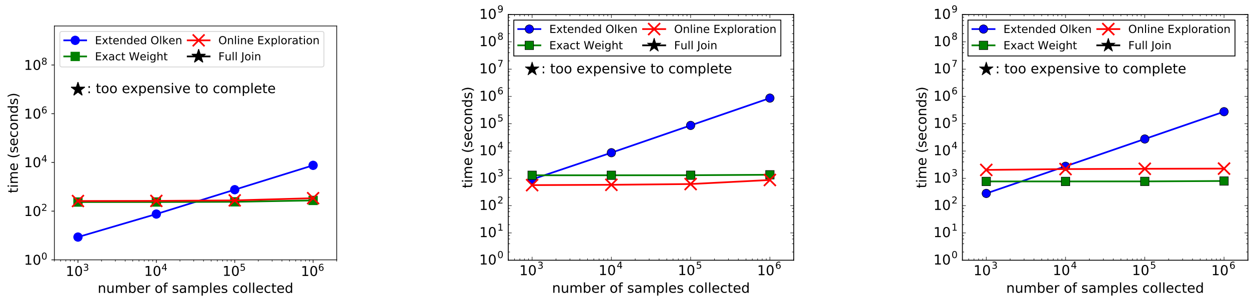
Lastly, QY is a cyclic query and not all join conditions are on a primary-key, foreign-key relationship, which makes FJ very expensive. EO still shows the similar patter but becomes much more expensive when it has to return larger number of samples as indicated in Figure 3c, due to the fact that its rejection rate is typically higher for a cyclic query (than an acyclic query). And in this case, OE has always outperformed EW. Since the number of relations in the join is more than that in QX, EW has become more expensive. It takes OE 245 seconds to produce 10^6 samples, while EW has to spend 381 seconds. Both OE and EW are several orders of magnitude more efficient than FJ for both QX and QY.



(a) Time to collect 10^6 samples. **(b) Time to collect first sample.**

Figure 5: TPC-H scalability tests for Query Y.

Scalability. We use QY to test the scalability of different methods, when the database is generated with different scale factors (that changes from 1 to 40, or roughly from 1GB to 40GB of data). We ask each method to collect 10^6 samples, and the results are shown in Figure 5a. FJ is too expensive to complete in all but the smallest case as data becomes large. On the other hand, methods instantiated from our framework have exhibited very good scalability. EO mostly only depends on the join topology (AGM bound) and the max degree information, which is relatively stable even when the data size grows (since the data distribution is the same). Both OE and EW scale roughly linearly to the increase of input table sizes, which is as expected from their analysis. But EW is strictly linear to the total

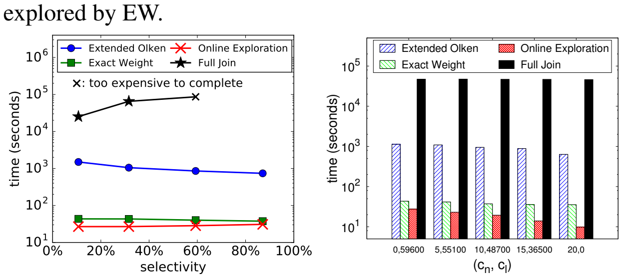


(a) Query T. (b) Query S. (c) Query F.

Figure 4: Social graph collection times for the full dataset.

input relation sizes, which is not necessarily the case for OE due to its online exploration nature. Hence, OE has the best performance and the performance gap between OE and EW enlarges as relations get bigger. For scale factor 40, OE has outperformed EW by almost 2 times.

We also measure the time for each algorithm to report the first sample and show the results in Figure 5b. Not surprisingly, EO is the fastest to report the first sample due to its low initialization overhead, whereas OE is faster than EW in reporting its first sample because of its online exploration. EW is the slowest to report the first sample, since it can only start reporting samples after all $w(t)$'s are computed by running its dynamic programming formulation. But once EW starts reporting samples, it enjoys the best sampling efficiency with no rejection, whereas OE will always reject some, and EO has the highest rejection rate. Note that the rejection rate of OE is actually extremely low, due to the accurate upper bound estimate returned by wander join estimators, as well as the exact weights for the subtrees explored by EW.



(a) One selection predicate. (b) Two selection predicates.

Figure 6: QX with selection predicates

Selection predicates. To show the effectiveness of various methods when selection predicates exist, we measure the time to collect 10^6 samples when 1 or 2 selection predicates are added to QX.

In the first experiment (shown in Figure 6a), we only add a selection predicate *l_extendedprice* $\geq c_l$ on table *lineitem* and vary the selectivity of the query (the ratio between the number of tuples in the query with the selection predicate and that without it). FJ is still very expensive and cannot complete when the selectivity gets larger than 50%. EO enjoys a better sampling efficiency when the query becomes less selective since rejection rate is lower in that case. On the other hand, both OE and EW adapt well to queries with different selectivity.

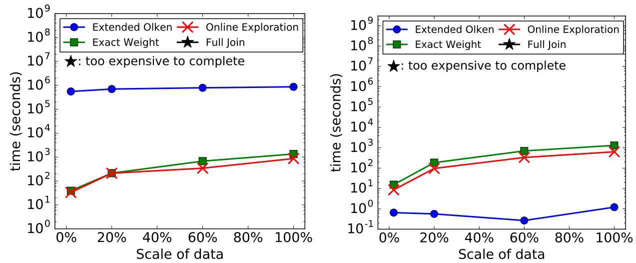
In the second experiment (shown in Figure 6b), we add two selection predicates *n_nationkey* $\geq c_n$ on *nation* and *l_extendedprice* $\geq c_l$ on *lineitem*. We fix the selectivity at roughly 20% and measure the time to collect 10^6 samples using different methods. FJ still is several orders of magnitude more expensive than other three sampling methods. EO, EW and OE all take shorter time to get the samples when the predicate on nation key gets more selective. This is because we start sample from the nation table and fewer tuples in the first table will lead to fewer rejections. Note that in this case, OE is consistently faster than EW due to its low initialization cost and they both run at least one order of magnitude faster than EO.

6.3 Social graph experiments

Figures 4a, 4b and 4c show the running time for different sampling algorithms to collect 10^3, 10^4, 10^5 and 10^6 samples from the triangle query (QT), square query (QS) and the tree-structured query (QF) on the full social graph dataset.

The running time of EW does not increase a lot when the total number of samples increases because it needs to build the weights table before it could produce samples but runs extremely fast once that's done. OE pays a much smaller cost during the warm up phase before it can produce samples. When it starts producing samples, it builds the weights on demand so its initial sampling rate is smaller than EW but increases over time. When the total number of samples needed is small, which of OE and EW runs faster depends on the join structure and input sizes. Query S has 3 large tables while Query T and F only have 2, so the dynamic programming (DP) in EW on Query S will be much more expensive than the other two. As a result, OE runs faster than EW on Query S. Query T is only a two-way join after we break the cycle by removing a relation. The cost for OE to explore a large portion of the join graph is roughly the same as the cost for EW to perform the DP. On Query F, since the join is acyclic and there are only 2 large tables, the DP cost of EW is actually smaller than the exploration cost of OE, and thus DP runs faster than OE. However, the total running time will be similar once a sufficiently large number of samples is needed. EO has a constant sampling rate due to its static structure but the sampling rate is much smaller than both EW's and OE's and will be slower than the other two even if only a moderate number of samples are required. The full join never finishes on all the three queries because of the large input and output sizes.

Figure 7a shows the time to get one million samples from the square query (QS) on the down-sampled social graph of varying sizes. As the data size increases, the running time of EW and OE algorithms will increase roughly linearly while EO increases logarithmically. The reason for EW and OE to have linear scalability is that the most of the time is spent on the DP for both EW and OE (OE uses EW to explore subtrees under certain levels), which



(a) Time to collect 10^6 samples. (b) Time to collect first sample.

Figure 7: Social graph scalability tests for query S.

scales linearly. The time of EO increases logarithmically because of the increase in the index size. However, EO is still at least 3 orders of magnitude slower than the other two due to its extremely high rejection rate. The full join still fails to run to complete even on the smallest scale of data.

Figure 7b shows the time to get the first sample. EO algorithm gets the first sample much faster than EW and OE because it does not need to pay the cost of building weights or warming up. But as long as the number of samples needed is sufficiently large, the total running time of EO is much longer than EW and OE.

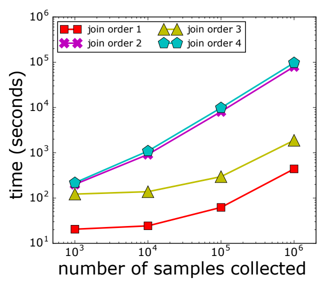


Figure 8: Time to get 10^6 sample in different join orders for QS.

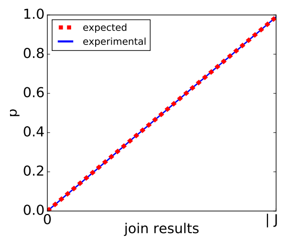


Figure 9: KS test of QX.

Figure 8 shows the time for OE to collect different number of samples using different join orders for QS, and it shows that the time using join order 1 is the smallest, because join order 1 has the smallest number of join results, which is picked up by our optimization.

6.4 Kolmogorov-Smirnov (KS) test

Recall that our sampling framework *guarantees to return a true simple random sample as long as the upper bounds $W(t)$'s used are indeed upper bounds for subtrees rooted at tuples t encountered in the sampling process,* which is the case in all of our instantiations for different join topology, with the only exception that wander join may return an estimate that's smaller than the actually join size for a subtree with very small probability. This only happens in theory with literally negligible probability since we use the upper limit of a wander join estimate (i.e., its estimate + its confidence interval). In practice, this never occurred in all our experiments.

Nevertheless, we use the well-known KS test to verify that the samples returned by each method based on our sampling framework are indeed uniform (clearly they are independent samples from the construction of our sampling methods). To do so, we collected 1 million samples from QX, an acyclic query, and the twitter triangle query QT, a cyclic query, and tested that the cumulative distribution of the samples returned followed the expected uniform distribution using the Kolmogorov-Smirnov test (K-S test).

Table 2: K-S values, where n = 1 million.

	Sampling Algorithm	Experimental d
Query X	Extended Olken	0.0003276
	Exact Weight	0.000774
	Online Exploration	0.000357
Query T	Extended Olken	0.00130
	Exact Weight	0.00131
	Online Exploration	0.00155

For each query, we collected 1 million samples from the join results, and assigned the sampled tuples to a collection M. We sort $M = (m_1, m_2, \ldots, m_n)$ in the same order that the full join algorithm would order the tuples in $J = (t_1, t_2, \ldots, t_{|J|})$. Let $O_J(m_i) = k$, where $t_k = m_i$, then M is sorted such that $O_J(m_i) > O_J(m_k)$ for all $k > i$.

If the samples reported from our algorithms are uniform, given any $m_i \in M$ the proportion $\frac{i}{|M|}$ will be approximately equal to the proportion $\frac{O_J(m_i)}{|J|}$. By comparing the progression of these proportions as a cumulative probability distribution while analyzing the entirety of M, we can compare how close our samples match with the expected uniform distribution.

The largest difference between these the expected cumulative distribution and the experimental cumulative distribution is d, the K-S score. The smaller the value of d the closer the experimental results match with the expected results (the uniform distribution). Furthermore, we calculated that when $n = 1$ million, we can accept the hypothesis that the distribution is uniform at the $\alpha = 0.01$ significant level if $d < 0.00163$.

We have generated a plot showing the experimental results along with the expected uniform distribution when samples are collected from QX using the OE in Figure 9.

The experimental value of d from the two queries with each of the sampling algorithms is shown in Table 2. All experiments pass the K-S test at the $\alpha = 0.01$ significance level.

6.5 Where to break a cycle

An important consideration when sampling from cyclic queries is how effective different ways of breaking the cycle are. We implemented the algorithm as described in Section 5.2, but to see the effectiveness of this strategy we manually configured different ways to cut the cycle and measured the sample rate for different ways to cut the cycle. Two ways exist to break cycles: one is to break by changing one join condition to a selection condition; the other is to remove a relation and check if the appropriate edge exists in the removed relation using a compound index. While the second option is better, it can not always be used, as described in Section 5.2.

We use our optimization algorithm based on wander join to estimate the size of each of the different ways to order a cycle join as an acyclic join. A table of the join estimation for orientation is shown in table 3 for Query Y. Clearly, some of the orientations are enormously different then other orientations, and the optimal strategy is to cut the cycle that leads to the smallest intermediate join size for the resulting acyclic query. Our optimization strategy as presented in Section 5.2 is always able to pick up the optimal cut.

Table 3: $|J|$ of different options to break a cycle.

option	$|J|$ adding condition	$|J|$ breaking relation
1	5.853×10^{17}	1.53×10^{14}
2	4.532×10^{18}	1.45×10^{14}
3	4.765×10^{17}	4.27×10^{14}
4	7.127×10^{12}	7.57×10^{10}
5	1.133×10^{13}	— invalid —
6	4.534×10^{17}	9.24×10^{10}
7	4.534×10^{18}	— same as 1 —

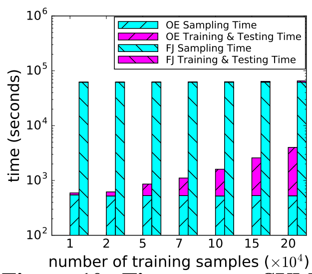


Figure 10: Time to run SVM on sampled data

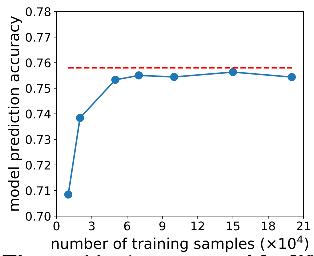


Figure 11: Accuracy with different sample sizes

6.6 Sampling joins for machine learning

As mentioned in Section 1, one useful application enabled by doing random sampling over joins is to combine join processing and machine learning in a unified way while improving on both. Recent works on learning over joins [21, 32] only apply to gradient descent with the linear regression model. They cannot be applied to more general models such as SVM. On the other hand, classical statistical learning theory states that the sample size needed to train a model up to a (δ,ε)-error is $\Theta(\frac{D+\ln(1/\delta)}{\varepsilon})$, where D is the VC-dimension of the model [34]. This holds for any classification model, thus sampling over join provides a more general solution to the problem.

In this subsection, we conduct a case study to train a SVM binary classifier to predict the return status field in the join results of 7 TPC-H tables. We modified the TPC-H generator so that each customer has a higher item return rate if the item is shipped to them during a randomly selected half of the whole period of time. The item return rate is also higher for those items with price in the top 1%. The full join results of this join have around 17 trillion records, from which 14 features have been selected to train the model. The details of the TPC-H data generator and the join query are provided in Appendix D. We sample the join with sample sizes ranging from 10K to 200K and train the SVM model using the sample . We also independently sampled 100K tuples to use as the test dataset.

Figure 11 shows how the accuracy changes when the sample size changes. As sample sizes increase, the accuracy first has a sharp increase and then converges to a constant value when the sample size is big enough. In this particular case, the accuracy converges to about 76%. This agrees well with the VC theory, which states that the sample size is proportional to $1/\varepsilon$. Note that the remaining 24% error is the inherent error of the model itself (as indicated by the dashed red line in Figure 11), i.e., even it is trained on all data. The ε in VC the theory denotes the additional error introduced by the sampling, which is already very close to 0 when the sample size has reached around 60K.

In terms of efficiency, we run the Online Exploration instantiation of our sampling framework to retrieve samples of varying sizes. In comparison, we also run the full join and directly sample from the full join results. Figure 10 shows the comparison of the two methods in terms of sampling, training and testing time. Online Exploration runs at least 2 orders of magnitude faster than the full join approach. Note that this is only due to the difference in sampling time; the training & testing time is the same for the two methods on the same sample size (but look differently due to the log scale of the figure, and almost invisible for the full join method).

7 REMARKS AND EXTENSIONS

The experimental results have clearly demonstrated the effectiveness, the efficiency, the scalability, and the extensibility of the proposed join sampling framework. Given that many join queries are extremely expensive, as shown in our experiments, supporting random sampling over arbitrary join queries is extremely important.

Among the three instantiations of our sampling framework, EO enjoys the least overhead in terms of initialization since it only needs the AGM bound (which depends on only the join topology and relations' sizes) and the max degree information (or the heavy hitter frequency threshold). Hence, it is the fastest to produce samples. But it has the worst sampling efficiency, due to loose upper bounds on join size estimation that lead to high rejection rates.

EW is effective when relations in the join have small or moderate sizes, since it uses a dynamic programming formulation to calculate all $w(t)$'s exactly, which has a linear cost to the sum of the sizes of all relations from the join. But this becomes expensive if one or more relations have a large table size. Furthermore, it's the slowest to start producing samples (but once that starts, it enjoys the best sampling efficiency due to zero rejection rate).

OE in most cases is an improvement to EW, since it's a hybrid method that combines online aggregation estimates (on the join size of any join subtree) and the on-demand exploration of $w(t)$'s for some join subtrees using EW. Hence, it enjoys a low rejection rate and a fairly quick response rate to start producing samples. It also enjoys good scalability as the sizes of some relations in the join start to increase, due to its property of doing online, on-demand EW explorations for only a small fraction of join subtrees.

Both EW and OE have easily outperformed FJ and EO in all cases, especially when the join size is much larger than input relations, and/or the join topology is complex.

Extensions and limitations. There are a few extensions and limitations of our proposed join sampling framework. *First*, it supports natural join or equi-join with selection conditions, it can be extended to support θ-joins by treating θ-join conditions as additional selection predicates, but the extension is non-trivial in some cases and deserves a full investigation.

Second, group-by and projection without de-duplication can be easily supported by adding the group-by and projection operators on top of the sampling operator. For a group-by query, we can perform group-by on the samples. When a group contains a small number of records, there could be few samples in that group. In such a case, we can convert the group-by clause to a selection predicate and sample independently in each group. Projection without de-duplication, which is the default in SQL, can be done by performing the projection on the samples. However, projection with de-duplication could change the cardinality of join results, and thus cannot be performed on the sample directly, which requires further investigation.

Third, as with most random sampling techniques, our random sampling framework relies on efficient random access to the underlying data. Hence, we do require index on the join attribute when going from one relation to another. This means that our framework is mostly useful for an in-memory database where index structures are often built and maintained for join attributes.

When some of these indexes are missing, i.e., no index on a join attribute A in some relation R, our sampling framework can build an index over A before initiate the sampling process. This adds only a linear cost only to the size of R, and the overall sampling cost will still be much smaller than computing the full join.

For disk-resident data, sampling efficiency will drop significantly due to its random access. However, in that case, the full join also becomes much more expensive.

Lastly, our sampling framework can handle updates. In particular, AGM bound is not affected by data updates; max degree and heavy hitters can be efficiently maintained under updates. Hence, the EO method needs no change. It will be expensive to re-calculate all $w(t)$'s in EW using a full dynamic programming whenever data receive updates, but the observation is that we only need to update the weights ($w(t)$'s), using the same backward propagation in our dynamic programming formulation, for tuples along the subtree affected by an update. Hence, EW only needs some minor adjustments to cope with updates. For the OE method, since it relies on EW and online aggregation technique like wander join, we only need to adapt wander join to support updates. This is possible by remembering the number of successful and failed walks passing through a tuple t, and adjusting these numbers when an update happens to t's children in the join hyper graph instance. Nevertheless, supporting updates is still non-trivial and we leave the exploration of this topic as an interesting future work.

Note that previous results on sampling over join by Chaudhuri et al. [8] (EW is a generalization of it) and using the Olken's method [25] (EO is an extension of it) also suffer from the same limitations.

8 RELATED WORK

Join sampling is a fundamental yet challenging problem that was explored in the pioneer work by Chaudhuri et al. in SIGMOD 1999 [8]. A special case of sampling from a foreign-key join was also studied in the same conference by [2]. Before that, random sampling from a single database table was studied by Olken and others in a series of work that were summarized by Olken's PhD thesis [25]. In fact, Olken's technique was explored by Chaudhuri et al. in their study. We have reviewed these works in Section 2, and they do not support random sampling over arbitrary multi-way joins that may or may not be cyclic. Some recent studies use stratified sampling to solve this problem [19] by pushing down sampling through a join operator, but no longer guarantees to always return random samples.

Our join sampling framework leverages the recent developments on deriving upper bound of the join size for a join query. To that end, AGM bound is a recent result that yields a upper bound for arbitrary joins [6] which we have used in one of the instantiations of our join sampling framework. We have also used the max degree of a table to help derive a tighter upper bound than the AGM bound, where the degree of a tuple from a table is defined as the frequency of its join attribute value in that table. Similar principle was also explored by [18] to get better estimates for a join size given degree (frequency) information, albeit in a more theoretical setting.

Another approach to obtain a join size upper bound is to use the idea of *online aggregation* [15, 22, 26, 28, 38]. In particular, we can use online aggregation over joins [14, 22, 23] to estimate a confidence interval for the join size (by using count(*) in the join query); taking the upper bound of this confidence interval provide an upper bound for the join size with high probability. There are many other works concerning join size cardinality estimation [4, 6, 12, 29, 31, 36, 41]; our join sampling framework may employ any of such techniques to instantiate a new method as long as it can efficiently generate an upper bound of a join size for any partial join from the join hyper graph with high probability.

Note that various sampling techniques are used to develop online and approximate aggregations [10, 14, 22, 27, 37, 39], however, in the context of join queries, represented by ripple join [14] and wander join [22], they generate either *uniform but non-independent* or *independent but non-uniform samples* respectively. But random sampling requires *uniform and independent samples*.

Sampling is an useful construct towards building interactive and approximate systems, e.g., Aqua [1], DBO [30], BlinkDB [3], Quickr [20], G-OLA [42], ABS [43], and many others. But none of these systems are able to support random sampling over joins.

There are also a revitalized interest in designing worst-case optimal join algorithms in various settings [9, 35], but typically the join size can be very large except for foreign-key key joins and random sampling will be much more efficient and effective than computing full joins, as shown in our experimental evaluations.

Lastly, random sampling is a fundamental machinery that is useful towards numerous analytical tasks, such as sophisticated statistics like medians, quantiles, inference, clustering, classification, kernel density estimate, etc. Many of these require *independent and uniform* samples in order to construct a good and scalable approximation or enable a rigorous and principal analysis to bound the performance. Random sampling is also useful towards query optimization [40].

9 CONCLUSION

This paper revisits the classic and important problem of join sampling. A general join sampling framework is proposed that incorporates existing studies as a special case, and can be instantiated with different methods that produce an upper bound on a join size. We show that the proposed framework is very flexible and general which can handle arbitrary multi-way acyclic and cyclic queries with or without selection predicates. Extensive experiments have demonstrated the efficiency and effectiveness of the proposed methods. Interesting future work directions include integrating the proposed work into a database engine, and extending our study and results to work with more complex queries such as θ-joins and joins that involve nested query blocks.

10 ACKNOWLEDGMENTS

Feifei Li, Zhuoyue Zhao and Robert Christensen are supported in part by NSF grants 1251019, 1302663, 1443046, 1619287 and NSFC grant 61729202. Xiao Hu and Ke Yi are supported by HKRGC under grants GRF-16211614, GRF-16200415, and GRF-16202317. The authors greatly appreciate the valuable feedback provided by the anonymous SIGMOD reviewers.

REFERENCES

[1] Swarup Acharya, Phillip B. Gibbons, and Viswanath Poosala. 1999. Aqua: A Fast Decision Support Systems Using Approximate Query Answers. In *VLDB*. 754–757.

[2] 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*. 275–286.

[3] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 2013. BlinkDB: queries with bounded errors and bounded response times on very large data. In *EuroSys*. 29–42.

[4] Noga Alon, Phillip B. Gibbons, Yossi Matias, and Mario Szegedy. 1999. Tracking join and self-join sizes in limited storage. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (PODS '99)*. ACM, New York, NY, USA, 10–20. https://doi.org/10.1145/303976.303978

[5] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proc. ACM SIGMOD International Conference on Management of Data*.

[6] Albert Attsertia, Martin Grohe, and Daniel Marx. 2013. Size bounds and query plans for relational joins. *SIAM J. Comput.* 42, 4 (2013), 1737–1767.

[7] Meeyoung Cha, Hamed Haddadi, Fabrício Benevenuto, and P. Krishna Gummadi. 2010. Measuring User Influence in Twitter: The Million Follower Fallacy. In *ICWSM*.

[8] Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. 1999. On Random Sampling over Joins. In *Proc. ACM SIGMOD International Conference on Management of Data*.

[9] Shumo Chu, Magdalena Balazinska, and Dan Suciu. 2015. From Theory to Practice: Efficient Join Query Evaluation in a Parallel Database System. In *SIGMOD*. 63–78.

[10] Bolin Ding, Silu Huang, Surajit Chaudhuri, Kaushik Chakrabarti, and Chi Wang. 2016. Sample + Seek: Approximating Aggregates with Distribution Precision Guarantee. In *SIGMOD*. 679–694.

[11] Alin Dobra, Chris Jermaine, Florin Rusu, and Fei Xu. 2009. Turbo Charging Estimate Convergence in DBO. In *Proc. International Conference on Very Large Data Bases*.

[12] Sumit Ganguly, Phillip B. Gibbons, Yossi Matias, and Abraham Silberschatz. 1996. Bifocal Sampling for Skew-Resistant Join Size Estimation. In *SIGMOD*. 271–281.

[13] P. J. Haas. 1997. Large-Sample and Deterministic Confidence Intervals for Online Aggregation. In *Proc. Ninth Intl. Conf. Scientific and Statistical Database Management*.

[14] P. J. Haas and J. M. Hellerstein. 1999. Ripple Joins for Online Aggregation. In *Proc. ACM SIGMOD International Conference on Management of Data*. 287–298.

[15] J. M. Hellerstein, P. J. Haas, and H. J. Wang. 1997. Online Aggregation. In *Proc. ACM SIGMOD International Conference on Management of Data*.

[16] Christopher Jermaine, Subramanian Arumugam, Abhijit Pol, and Alin Dobra. 2007. Scalable Approximate Query Processing With The DBO Engine. In *Proc. ACM SIGMOD International Conference on Management of Data*.

[17] Madhav Jha, C. Seshadhri, and Ali Pinar. 2015. Path Sampling: A Fast and Provable Method for Estimating 4-Vertex Subgraph Counts. In *WWW*.

[18] Manas Joglekar and Christopher Re. 2016. It's all a matter of degree: Using degree information to optimize multiway joins. In *ICDT*.

[19] Niranjan Kamat and Arnab Nandi. 2016. Perfect and Maximum Randomness in Stratified Sampling over Joins. *CoRR* abs/1601.05118 (2016).

[20] Srikanth Kandula, Anil Shanbhag, Aleksandar Vitorovic, Matthaios Olma, Robert Grandl, Surajit Chaudhuri, and Bolin Ding. 2016. Quickr: Lazily Approximating Complex AdHoc Queries in BigData Clusters. In *SIGMOD*. 631–646.

[21] Arun Kumar, Jeffrey Naughton, and Jignesh M. Patel. 2015. Learning Generalized Linear Models Over Normalized Data. In *SIGMOD*.

[22] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. 2016. Wander Join: Online Aggregation via Random Walks. In *SIGMOD*. 615–629.

[23] Gang Luo, Curt J. Ellmann, Peter J. Haas, and Jeffrey F. Naughton. 2002. A Scalable Hash Ripple Join Algorithm. In *Proc. ACM SIGMOD International Conference on Management of Data*.

[24] Hung Q Ngo, Christopher Ré, and Atri Rudra. 2013. Skew Strikes Back: New Developments in the Theory of Join Algorithms. (2013). arXiv:1310.3314v2

[25] F. Olken. 1993. *Random Sampling from Databases*. Ph.D. Dissertation. University of California at Berkeley.

[26] Niketan Pansare, Vinayak R. Borkar, Chris Jermaine, and Tyson Condie. 2011. Online Aggregation for Large MapReduce Jobs. In *Proceedings of the VLDB Endowment*, Vol. 4.

[27] Chengjie Qin and Florin Rusu. 2013. Sampling Estimators for Parallel Online Aggregation. In *BNCOD*. 204–217.

[28] Chengjie Qin and Florin Rusu. 2014. PF-OLA: a high-performance framework for parallel online aggregation. *Distributed and Parallel Databases* 32, 3 (2014), 337–375.

[29] Florin Rusu and Alin Dobra. 2008. Sketches for size of join estimation. *ACM TODS* 33, 3 (2008).

[30] Florin Rusu, Fei Xu, Luis Leopoldo Perez, Mingxi Wu, Ravi Jampani, Chris Jermaine, and Alin Dobra. 2008. The DBO database system. In *SIGMOD*. 1223–1226.

[31] Florin Rusu, Zixuan Zhuang, Mingxi Wu, and Chris Jermaine. 2015. Workload-Driven Antijoin Cardinality Estimation. *ACM Trans. Database Syst.* 40, 3 (2015), 16.

[32] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. 2016. Learning Linear Regression Models over Factorized Joins. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*.

[33] C Seshadhri, Ali Pinar, and Tamara G Kolda. 2013. Triadic measures on graphs: the power of wedge sampling. In *SDM*.

[34] V. N. Vapnik and A. Y. Chervonenkis. 1971. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and its Applications* 16 (1971), 264–280.

[35] Todd L. Veldhuizen. 2012. Leapfrog Triejoin: a worst-case optimal join algorithm. *CoRR* abs/1210.0481 (2012).

[36] David Vengerov, Andre Cavalheiro Menck, and Mohamed Zait. 2015. Join Size Estimation Subject to Filter Conditions. In *Proc. International Conference on Very Large Data Bases*.

[37] Lu Wang, Robert Christensen, Feifei Li, and Ke Yi. 2016. Spatial Online Sampling and Aggregation. In *Proc. International Conference on Very Large Data Bases*.

[38] Sai Wu, Shouxu Jiang, Beng Chin Ooi, and Kian-Lee Tan. 2009. Distributed Online Aggregation. *PVLDB* 2, 1 (2009), 443–454.

[39] Sai Wu, Beng Chin Ooi, and Kian-Lee Tan. 2010. Continuous sampling for online aggregation over multiple queries. In *SIGMOD*. 651–662.

[40] Wentao Wu, Jeffrey F. Naughton, and Harneet Singh. 2016. Sampling-Based Query Re-Optimization. In *SIGMOD*. 1721–1736.

[41] Feng Yu, Wen-Chi Hou, Cheng Luo, Dunren Che, and Mengxia Zhu. 2013. CS2: a new database synopsis for query estimation. In *SIGMOD*. 469–480.

[42] Kai Zeng, Sameer Agarwal, Ankur Dave, Michael Armbrust, and Ion Stoica. 2015. G-OLA: Generalized On-Line Aggregation for Interactive Analysis on Big Data. In *SIGMOD*. 913–918.

[43] Kai Zeng, Shi Gao, Jiaqi Gu, Barzan Mozafari, and Carlo Zaniolo. 2014. ABS: a system for scalable approximate queries with accuracy guarantees. In *SIGMOD*. 1067–1070.

A PROOF OF THEOREM 2

PROOF. A *partial tree* P is a subset of the relations defined recursively as follows:

(1) $\{R_0\}$ is a partial tree.

(2) For any partial tree P and any $R_i \in P$, $P \cup \{R_i^k$ for all $k\}$ is a partial tree. Let ∂P be the *boundary* of P, i.e., it consists of all relations in P whose children (if there are any) do not belong to P.

A set of tuples $T(P)$ (resp. $T(\partial P)$) are called the *partial join result* of P (resp. ∂P) if $T(P)$ (resp. $T(\partial P)$) contains exactly one tuple from each $R_i \in P$ (resp. ∂P).

We will show that for any partial tree P, any of its partial join results $T(P)$ is sampled by the algorithm with probability $\left(\prod_{t\in T(\partial P)} w(t)\right)/W(r_0)$. Then taking P to be the whole tree and applying Invariant (6) prove the theorem.

The proof is by induction on the size of P. The base case when P contains only R_0 is trivial. Now assume that the claim holds for any partial tree strictly contained in P. We remove all the children of one relation, say R_i, from P to obtain a smaller partial tree P' . By the induction hypothesis, we know that the tuples in $T(P')$ have been sampled with probability $\left(\prod_{t\in T(\partial P')} w(t)\right)/W(r_0)$.

To go from $T(P')$ to $T(P)$, the algorithm must succeed in obtaining the corresponding tuple in each R_i^k. Let $t_i \in R_i$ be the tuple sampled in R_i and $t_i^k \in R_i^k$ the tuple sampled in R_i^k. The algorithm first decides if it should move onto the children of R_i with probability $\left(\prod_k W(t_i, R_i^k)\right)/W(t_i)$ in line 7. If so, it makes a recursive call with ACYCLIC-SAMPLE(t_i, R_i^k) for each R_i^k. Inside this call, the algorithm first decides whether to sample from R_i^k with probability $W(t_i \ltimes R_i^k)/W(t_i, R_i^k)$ in line 3, and then samples t_i^k with probability $W(t_i^k)/W(t_i \ltimes R_i^k)$ in line 4. Thus, the probability that $T(P)$ are all sampled is

$$
\begin{aligned}
&\frac{\prod_{t\in T(\partial P')} W(t)}{W(r_0)} \cdot \frac{\prod_k W(t_i, R_i^k)}{W(t_i)} \\
&\qquad \cdot \prod_k \left(\frac{W(t_i \ltimes R_i^k)}{W(t_i, R_i^k)} \cdot \frac{W(t_i^k)}{W(t_i \ltimes R_i^k)} \right) \\
&= \frac{\prod_{t\in T(\partial P')} W(t)}{W(r_0)} \cdot \frac{\prod_k W(t_i^k)}{W(t_i)} \\
&= \frac{\prod_{t\in T(\partial P)} W(t)}{W(r_0)}. \quad \square
\end{aligned}
$$

□

B TPC-H QUERIES

Q3 - Lineitems joined with the orders they associated with and the customers who placed the orders:

```
SELECT c_custkey, o_orderkey, l_linenumber
FROM customer, orders, lineitem
WHERE   c_custkey = o_custkey
    AND l_orderkey = o_orderkey;
```

QX - Suppliers and customers in the same nations with the purchase history of the customers.

```
SELECT nationkey, s_suppkey, c_custkey,
       o_orderkey, l_linenumber
FROM nation, supplier, customer,
     orders, lineitem
WHERE   n_nationkey = s_nationkey
    AND s_nationkey = c_nationkey
    AND c_custkey = o_custkey
    AND o_orderkey = l_orderkey;
```

QY - Suppliers that are in the same nation as a pair of customers in the same nation that has once ordered the same items:

```
SELECT l1.l_linenumber, o1.o_orderkey, c1.c_custkey,
       l2.l_linenumber, o2.o_orderkey, s_suppkey,
       c2.c_custkey
FROM lineitem l1, orders o1, customer c1,
     lineitem l2, orders o2, customer c2, supplier s
WHERE   l1.l_orderkey = o1.o_orderkey
    AND o1.o_custkey = c1.c_custkey
    AND l1.l_partkey = l2.l_partkey
    AND l2.l_orderkey = o2.o_orderkey
    AND o2.o_custkey = c2.c_custkey
    AND c1.c_nationkey = s.s_nationkey
    AND s.s_nationkey = c2.c_nationkey;
```

C SOCIAL GRAPH QUERIES

QT - Triangles in a twitter follower graph:

```
SELECT * FROM
popuar-user A, twitter-user B, twitter-user C
WHERE   A.dst = B.src
    AND B.dst = C.src
    AND C.dst = A.src;
```

QS - Squares in a twitter follower graph:

```
SELECT *
FROM popuar-user A, twitter-user B,
     twitter-user C, twitter-user D
WHERE   A.dst = B.src
    AND B.dst = C.src
    AND C.dst = D.src
    AND D.dst = A.src;
```

QF - Two users followed by a popular user and another user who is followed by a popular user followed by the first popular user:

```
SELECT *
FROM popular-user A, twitter-user B,
     twitter-user C, popular-user D
WHERE   A.src = B.src
    AND C.dst = A.src
    AND C.src = D.src;
```

D TRAINING DATA GENERATOR AND JOIN QUERY FOR THE LEARNING EXPERIMENTS

The training data used in Section 6.6 is generated by a modified TPC-H data generator. In the lineitem table of TPC-H benchmark, there is a return_flag field which indicates whether a delivered item has been returned by the customer who placed the order. In the original benchmark, this field follows a uniform distribution if the item has been delivered. We modified the distribution of the return flag as follows so that it is related to customer, shipping date and the price of the item.

First, all items have a 10% probability to be returned. Then, we split the whole period of shipping dates into 13 200-day windows (0-12) with the last one being shorter than 200 days (because of the range of the date values). Each customer chooses the set of the odd-numbered shipping date windows with probability $1/2$ and the other half with probability $1/2$. If the item is shipped within the chosen windows, they are returned with an additional 50% chance. This is to simulate the case that often time a customer's return rate exhibits strong degree of temporal locality.

Finally, if the price after discount for an item is among roughly the top 1% of all the items (l_extendedprice * (1 - l_discount) / 1000 >= 85), it is returned with additional 30% chance. This is to simulate the scenario that expensive orders experience a higher rate of return.

The query for extracting features to train the SVM model is shown below. In order to predict the return_flag of an item ℓ_1 shipped to a customer c, we want to look at another item ℓ_2 shipped to the same customer c and include the return_flag of ℓ_2 as a feature. We also include the difference between the shipping dates of the two items (ℓ_1 and ℓ_2), as well as the quantity and price after discount of both items. In addition, we include the region of the supplier of the item, the supplier's account balance, the total price and the priority of the order as the features.

Note that a machine learning algorithm has no way of knowing what features are important for making an effective prediction. Hence, it cannot "magically" choose only those features that will influence the return_flag values used in our generator (i.e., difference of shipping dates and item price). Rather, it has to make use of a set of seemingly useful features as shown in our query to train its model (e.g., it is possible that customers from certain region have a higher return rate that others; it is also possible that return rate is correlated with order quantity.

```
SELECT
    l1.l_returnflag, n_regionkey, s_acctbal,
    l1.l_quantity, l1.l_extendedprice, l1.l_discount,
    l1.l_shipdate, o1.o_totalprice, o1.o_orderpriority,
    l2.l_quantity, l2.l_extendedprice, l2.l_discount,
    l2.l_returnflag, l2.l_shipdate
```

```
FROM nation, supplier, lineitem l1, orders o1,
     customer, orders o2, lineitem l2
WHERE   s_nationkey = n_nationkey
    AND s_suppkey = l1.l_suppkey
    AND l1.l_orderkey = o1.o_orderkey
    AND o1.o_custkey = c_custkey
    AND c_custkey = o2.o_custkey
    AND o2.o_orderkey = l2.l_orderkey;
```

E ADDITIONAL EXPERIMENTS

Table 4: Random sampling acceptance rate comparison

	Exact Weight	Online Exploration	Extended Olken
Q3	1.00	1.00	0.0092
QX	1.00	1.00	0.0084
QY	0.071	0.071	0.00033
QT	0.15	0.15	0.00078
QS	0.13	0.0076	0.000043
QF	1.00	1.00	0.000031

Table 4 shows the acceptance rate of the three instantiations of our sampling framework on all the six queries. As shown in the table, Exact Weight and Online Exploration have the highest acceptance rates since the upper bounds are quite tight. Extended Olken, on the other hand, usually has an acceptance rate of 1 or 2 magnitude lower because of the loose upper bounds set by the smaller one of AGM bound and the product of maximum degrees in each table.