Sampling through Joins

Maksim Levental, Greg Pauloski

CONTENTS

ABSTRACT

In most relational databases joins $\texttt{JOIN}(R_1, R_2)$ between two tables R_1 and R_2 are expensive, especially on large tables (owing to the join having maximum cardinality $|R_1|\times|R_2|$). Often joins are used for the purposes of computing aggregations (e.g. `SUM`, `AVG`, `COUNT`). One potential optimization is to `SAMPLE` the operand tables. Unfortunately, in general, `SAMPLE` doesn't commute with `JOIN`. We aim to study the regimes under which the operations do commute and what are the tradeoffs when they don't.

1 INTRODUCTION

Joins are a fundamental operator in the relational algebra entailed by E.F. Codd's relational model of data [4]; they're used to bridge seemingly unrelated entities and analyze implied relationships. By definition, joins are a subset of the cartesian product of some relations and therefore incur compute cost proportional to that product (i.e. exponential in the number of relations to be joined). In many instances joins are employed as a preprocessing step in order to compute an aggregation function over resulting data, such as a sum, an average, or a count. In such instances it is often acceptable, and occasionally even preferable, to constuct estimates (along with error guarantees) of the functions based on a sample of the entire join; one obvious advantage is in the reduction on the execution time of the aggregation function. Naturally this prompts the question of whether the operand relations in the join can be themselves sampled and whether the join need be constructed at all. Alternatively, when relations are never materialized (e.g. for streaming relations) *online* aggregation, using online sampling, can be performed using similar such sampling techniques.

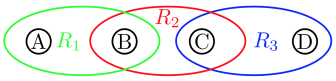
We therefore proceed to study when sampling commutes with joins by reviewing a sampling[1] of the literature on such techniques and then performing some experiments to verify/validate the most distinctive techniques. The rest of the report is organized as follows: section 2 briefly discusses the necessary background on sampling and joins, section 3 reviews several techniques, culminating in a taxonomy that categorizes said techniques according to constraints and performance (see tbl. 1), section 4 describes and reviews our experiments, and finally section 5 concludes with some conjecture about promising research directions.

2 BACKGROUND

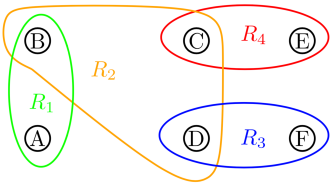
2.1 Joins

Let R_1, R_2 be *relations* on respective *attributes* $(A, B), (A, C)$. The *natural join* $J \coloneqq R_1 \bowtie R_2$ is defined as

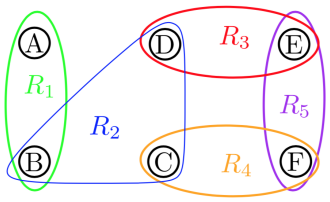
1. Pun intended.



(a) Chain join



(b) Acyclic (or tree/star) join



(c) Cyclic join

Figure 2.1: Join for relations R_1, R_2, R_3, R_4, R_5 [15].

the set of all pairs of tuples in R_1 and R_2 that are equal on their common attribute A

$$J \coloneqq \{(t_1, t_2) \mid t_1.A = t_2.A\, t_1 \in R_1, t_2 \in R_2\}$$

The extension to K relations and multiple common attributes is the natural conjunction. Note that If R_1, R_2 have no common attributes then $R_1 \bowtie R_2 \equiv R_1 \times R_2$, the cartesian product[2] of R_1, R_2. We will also have need of considering joins between relations that have a predicated relationship between some columns; define a θ-join $R_1 \bowtie_{A\theta B} R_2$ of relations $R(A, \dots), R(B, \dots)$ on attributes A, B and some binary operator $\{<, \leq, =, \neq, >, \geq\}$ as all tuples in $R_1 \bowtie R_2$ such that $a\theta b$, for $a \in A, b \in B$ evaluates to true. The particular case of θ being the equality operator $=$ is called an *equijoin*.

We can model K-joins as a hypergraph[3] on the union of all attributes in all R_i; let $\mathcal{A}(R_i)$ be the attributes of R_i and

$$\mathcal{A} \coloneqq \bigcup_{i=1}^{K} \mathcal{A}(R_i)$$

Then a join is a set of vertices corresponding to individual attributes and a hyperedge contains/connects all the attributes of an individual relationship. Using this model we can represent three distinct classes of joins:

- *Chain join*: joins where relations can easily be ordered such that consecutive relations share one attribute (see fig. 2.1a).
- *Acyclic join*: also known as a tree or star join, these are joins where there is no cycle on the join hypergraph (see fig. 2.1b).
- *Cyclic join*: joins where there is a cycle on the join hypergraph (see fig. 2.1c).

2. Sometimes called the *cross join* of R_1, R_2.

3. A generalization of a graph in which an edge can join any number of vertices.

2.2 Sampling

Let R be a relation over some attributes with cardinality $n \coloneqq |R|$. For $0 \leq f \leq 1$, $\texttt{SAMPLE}(R, f)$ is defined to be a uniform random sample S of tuples in R such that $|S| = f \cdot n$. A priori the sampling semantics are unspecified; indeed, there are three distinct interpretations of sampling:

- *Sampling with replacement* (WR): sample $f \cdot n$ tuples uniformly and independently with replacement. The result is bag (multiset) of tuples.
- *Sampling without replacement* (WoR): sample $f \cdot n$ *distinct* tuples, i.e. each successive tuple is sampled from the remaining set of tuples.
- *Coin flip sampling* (CF): for each tuple in R, choose that tuple with probability f (and reject with probability $1 - f$). This essentially produces a draw from a binomial distribution $B(n, f)$ where heads correspond to chosen tuples.

Note we can transform amongst these interpretations for various input, output pairs [2].

Furthermore, sampling can either be *correlated*[4] or *uncorrelated*; we will see that correlated samples lead to higher error rates than uncorrelated samples (see sec. 3.2). We are also particularly interested in *streaming* or *sequential sampling*, which is the act of sampling a relation as it streams by, for example in instances when the relation is produced iteratively by some long running process. We will also have need of *weighted sampling*, wherein elements are sampled with probability proportional to some weight assigned to those elements.

2.3 The join sampling problem

First we make some elementary observations. Let R_1, R_2 be two relations of cardinalities n_1, n_2 respectively and $J \coloneqq R_1 \bowtie R_2$, with $n \coloneqq |J|$, and further suppose R_1, R_2 only have common attribute $A \subseteq D$ for some domain D. For all $v \in D$, let $m_1(v), m_2(v)$ be the frequencies (i.e. quantities) of tuples in R_1, R_2, respectively, for which attribute A takes on value v

$$m_i(v) \coloneqq |\{t \mid v \in D \wedge t \in R_i \wedge t.A = v\}|$$

for $i = 1, 2$. Note that for $v \in D \backslash A$, we have $m_i(v) = 0$. Then

$$\sum_{v \in D} m_i(v) = n_i$$

That is to say, projecting from R_i to A partitions the relation R_i. Clearly

$$n = \sum_{v \in D} m_1(v) \cdot m_2(v)$$

since each tuple in R_1 that contributes to $m_1(v)$ joins with $m_2(v)$ tuples in R_2 and vice-versa.

Consider sampling from relations $R_1(A, B), R_2(B, C)$ defined as such

$$R_1 \coloneqq \{(a_1, b_0), (a_2, b_1), (a_2, b_2), \dots, (a_2, b_K)\}$$
$$R_2 \coloneqq \{(a_2, c_0), (a_1, c_1), (a_1, c_2), \dots, (a_1, c_K)\}$$

4. Inclusion of a member of the sample implies, with some probability, inclusion of some other member.

Observe that $m_1(a_1) = m_2(a_2) = K$ and thus

$$\begin{aligned}|R_1 \bowtie R_2| &= \sum_{v \in D} m_1(v) \cdot m_2(v) \\ &= m_1(a_1) \cdot m_2(a_1) + m_1(a_2) \cdot m_2(a_2) \\ &= K + K = 2K\end{aligned}$$

This shows that there is *skew* on the different attribute values for which the join occurs. Suppose we wish to construct $\texttt{SAMPLE}(R_1 \bowtie R_2, f)$; we should expect that half of the tuples in such a sample will have attribute A equal to a_1 and the other half will have attribute A equal to a_2. Unfortunately, if we attempt to sample each of R_1, R_2 independently and then perform the join we are unlikely to get the correct result; the probability that a uniform sample from R_1 contains the tuple (a_1, b_0) is $1/(K+1)$ and similarly for $(a_2, c_0) \in R_2$. Indeed, with high probability[5]

$$\texttt{SAMPLE}(R_1, f) \bowtie \texttt{SAMPLE}(R_2, f) = \emptyset$$

This implies that the crux of the *join sampling problem* is that, for $\texttt{SAMPLE}(R_1 \bowtie R_2, f)$, each tuple $t_1 \in R_1$ is sampled in direct proportion to the quantity of tuples $t_2 \in R_2$ that join with it, and vice-versa. To wit:

$$\begin{aligned}R_1 \bowtie R_2 = \{&(a_1, b_0, c_1), \ldots, (a_1, b_0, c_K), \\ &(a_2, b_1, c_0), \ldots, (a_2, b_K, c_0)\}\end{aligned} \quad (2.1)$$

Thus, $(a_1, b_0) \in R_1$ is sampled from $R_1 \bowtie R_2$ with probability $1/2$ while the remaining tuples in R_1 are sampled each with probability $1/2K$. That is to say, $\texttt{SAMPLE}(R_1 \bowtie R_2, f)$ corresponds to a weighted sample of R_1 rather than a uniformly random sample. This demonstrates the impossibility of constructing a uniform random sample of $R_1 \bowtie R_2$ by first uniformly sampling each of R_1, R_2.

Theorem 1. *Given uniform random samples* $S_i := \texttt{SAMPLE}(R_i, f_i)$ *with* $f_i < 1$ *it is impossible to construct a uniform random* $\texttt{SAMPLE}(R_1 \bowtie R_2, f)$ *from* S_1, S_2 *for any* $f > 0$.

In fact, even if we bound the skew we cannot hope to achieve such a sample [2]

Theorem 2. *Given a common attribute value* v *for relations* R_1, R_2 *and attribute frequencies* $m_1(v), m_2(v)$, *it is impossible to construct a uniformly random* $\texttt{SAMPLE}(R_1 \bowtie R_2, f)$ *from* S_1, S_2 *(as defined in thm. 1) unless*

$$f_1 \geq \tfrac{f \cdot m_2}{2} \text{ and } f_2 \geq \tfrac{f \cdot m_1}{2} \text{ for } f \leq \tfrac{1}{\max\{m_1, m_2\}}$$

or

$$f_1 \geq \tfrac{1}{2} \text{ and } f_2 \geq \tfrac{1}{2} \text{ for } f \leq \tfrac{1}{\min\{m_1, m_2\}}$$

Therefore, in general, we cannot commute SAMPLE with $\bowtie$. However, this does not preclude the possibility of *non-uniformly* random samples of each of R_1, R_2 being used to construct a uniformly random sample of $R_1 \bowtie R_2$. We shall see that this will be the primary path to resolution of the join sampling problem.

5. $\left(1 - \frac{1}{k+1}\right) \cdot \left(1 - \frac{1}{k+1}\right) = 1 - O\left(\frac{1}{k}\right)$.

2.4 Online aggregation

Consider a SQL query of the form

```
SELECT g, AGG(g(R_1, R_2, ..., R_K))
FROM R_1, R_2, ..., R_K
WHERE θ(R_1, R_2, ..., R_K) AND σ_φ(R_1, R_2, ..., R_K)
GROUP BY g
```

where $\texttt{AGG} \in \{\texttt{SUM}, \texttt{AVG}, \texttt{COUNT}, \ldots\}$, $g(R_1, R_2, \ldots, R_K)$ is an expression that involves any attributes of the relations $R_1, R_2, \ldots, R_K$, and θ is the aforementioned join condition operator, and σ_φ selects tuples that satisfy the condition φ. An effective online aggregation algorithm produces an estimator $\hat{Y}_n$ for $\texttt{AGG}(g)$, at every iteration n, along with confidence intervals (CIs)

$$I_n := \left[\hat{Y}_n - \epsilon_n, \hat{Y}_n + \epsilon_n\right]$$

where ϵ_n, called the *precision*, or *margin of error* is defined as a function of α, the *confidence level*, by

$$P\left(\left|\hat{Y}_n - \texttt{AGG}(g)\right| \leq \epsilon_n\right) \geq \alpha$$

For example, if estimators are derived by using the Central Limit Theorem, then

$$\epsilon_n := \frac{z_p \hat{\sigma}_n}{\sqrt{n}}$$

where z_p is the *z-score*[6] corresponding to a $100p\%$ confidence level. Typically one of ϵ_n, α (but not both) is specified by the user and the algorithm reports the other as it proceeds; the user terminates the query when the unspecified parameter reaches a desired value.

3 Techniques

We consider the problem of constructing $\texttt{SAMPLE}(R_1 \bowtie \cdots \bowtie R_K, f)$ by means of sampling the operand relations R_i. We often restrict ourselves to joins over one common attribute A and discuss necessary adjustments extension to multi-attribute joins. In the proceeding we partition the approaches to resolving the sampling problem according to how much information exists for each of the relations R_i:

- *Frequencies*: complete frequencies (for each possible value) for all relevant attributes.
- *Partial frequencies*: frequencies for only the high frequency values; this is a useful category since it is in fact these values that distort the ultimate sample.
- *Frequency upper bounds*: upper bounds on the frequencies of values of relevant attributes; these are useful as a proxy for complete or partial frequencies.
- *Index*: a means to perform indexed access (as opposed to sequential) on tuples of a relation according to values of some attribute; crucially we require the ability to also evaluate predicates on said attribute.

We therefore produce a classification for strategies according to whether we have statistics (full or partial) and indices for both R_i, neither, or something in between (see tbl. 1).

6. The unique number such that $P(-z_p \leq Z \leq z_p) = p$ for $Z \sim \mathcal{N}(0, 1)$ where $\mathcal{N}(0, 1)$ is the standard Normal distribution.

Table 1: Techniques

Strategy	R_1	R_i	Complexity
Olken sampling [14]	Index	Index and freqs	e.g. $O\left(M\left\|R_1\right\| / \left\|R_1 \bowtie R_2\right\|\right)$ per tuple in result
Stream sampling [2]	—	Index and freqs	$O\left(1\right)$ per tuple in result
Group sampling [2]	—	Frequencies	e.g. $O\left(\alpha_1 \left\|R_1 \bowtie R_2\right\|\right)$
Frequency-partition sampling [2]	—	End-biased freqs	e.g. $O\left(\alpha_2 \left\|R_1 \bowtie R_2\right\|\right)$
Index sampling [2]	—	Index R_i^{hi} and end-biased freqs	e.g. $O\left(\alpha_3 \left\|R_1 \bowtie R_2\right\|\right)$
Universe sampling [8]	—	—	e.g. $O\left(\max\left(\left\|R_1\right\|, \left\|R_2\right\|\right)\right)$
MaxRand join [11]	Index and freqs	Index and freqs	$O\left(\left\|D\right\| \left\|K\right\| + \sum_{i=1}^{K} N\left(1 + \log\left(\frac{\left\|R_i\right\|}{N}\right)\right)\right)$
Ripple join[7]	Index	Index	$O\left(\sqrt{\left\|R_i\right\| / d}\right)$ per tuple in result
Hash ripple join[7]	Index	Index	$O\left(\sqrt{\left\|R_i\right\| / d}\right)$ per tuple in result
Wander join[12]	Index	Index	$O\left(1/2^{K-1}\right)$ per tuple in result for K relations
Upper bound join [1]	Index	Index and upper bounds	$1/W\left(t\right)$ for $t \in R_i$

Algorithm 1 Olken Sampling

Inputs:
$R_1\left(A, \dots\right), R_2\left(A, \dots\right)$
$M \coloneqq \max_{v \in D} m_2\left(v\right)$
$k \coloneqq \lceil f \cdot \left|R_1 \bowtie R_2\right| \rceil$
Output: S, a $\text{SAMPLE}_{WR}\left(R_1 \bowtie R_2, f\right)$
Init:
$S[\cdots] \coloneqq 0$
// sample t_1 from R_1 uniformly
$t_1 \sim U(R_1)$
// sample matching rows in R_2
$t_2 \sim U\left(\{t \mid t \in R_2 \wedge t.A = t_1.A\}\right)$
Begin:
for $i \coloneqq 1$ to k:
// accept in proportion with the
// frequency $m_2\left(t_1.A\right)$
while $m_2\left(t_2.A\right) < U\left(0, M\right)$:
$t_1 \sim U(R_1)$
$t_2 \sim U\left(\{t \mid t \in R_2 \wedge t.A = t_1.A\}\right)$
$S[i] \coloneqq (t_1, t_2)$

Several of the described techniques make use of more general sampling algorithms such as reservoir sampling and rejection sampling. Consult appendix 6.1 for brief descriptions. We relegate techniques that are derivations on a theme (and whose pseudo-code is extensive) to the appendix as well (see appendix 6.2).

3.1 Olken sampling

Olken sampling only applies in the best possible case; for example, when we have random access on R_1 and R_2 and full frequency statistics for R_2. In this case we use rejection sampling (see sec. 6.1.1) to produce tuples in $\text{SAMPLE}_{WR}\left(R_1 \bowtie R_2, f\right)$ by sampling from tuples R_1 in direct proportion to their frequency in R_2 (see alg. 1). With $M \coloneqq \max_{v \in D} m_2\left(v\right)$, for $v \in A \subseteq D$, Olken sampling produces a WR sample of $R_1 \bowtie R_2$ and requires Mn_1/n iterations for each output tuple [2], where $n \coloneqq \left|R_1 \bowtie R_2\right|$ and $n_1 \coloneqq \left|R_1\right|$.

3.1.1 Stream sampling

In the circumstances where we have only streaming access to R_1 we can use weighted stream sampling (see sec. 6.1.4), with weights defined as $w(t) \coloneqq m_2\left(t_1.A\right)$, and join each such tuple with a randomly selected tuple from R_1. See alg. 12. Note that we do constant work per included tuple.

3.1.2 Group sampling

In the case where we further reduce our information to only statistics (no index) for R_2, we can weighted-sample tuples t_i from R_1 and then sample from $t_i \bowtie R_2$. In effect performing a `GROUP BY` (see alg. 13). We perform the second sampling using unweighted sampling with replacement. The cost of this strategy depends on the cardinalities $\left|t_i \bowtie R_2\right|$:

$$\alpha_1 \coloneqq r \times \frac{\sum_{v \in D} m_1\left(v\right) m_2\left(v\right)^2}{\left(\sum_{v \in D} m_1\left(v\right) m_2\left(v\right)\right)^2}$$

Note that this strategy compares very favorably to naive sampling[7] (the only other possible strategy for this set of circumstances).

3.1.3 Frequency-partition sampling

Suppose we now only have an end-biased histogram[8] for R_2. We can then logically partition R_2 into those tuples with high-frequency D^{hi} values and their complement D^{lo} (tuples with low-frequency values) and notice that it's the former that's responsible for the skew that is the core of the sampling problem (see alg. 14). These D^{hi} values function to inflate the join. Therefore we can recover efficiency by taking a hybrid approach: by employing group sample strategy on D^{hi} [9] and naively sampling on D^{lo}. Working to our advantage is the fact that there cannot be too many high-frequency values and that it is precisely this set of values for which maintaining the frequencies is cheap [10].

The remaining issue is how to determine the allocation of the sample to each group: take $k \coloneqq \lceil f \cdot \left|R_1 \bowtie R_2\right| \rceil$ from each subset and then cull in each subset in order to reduce the total quantity [11]. The primary advantage of this algorithm is that it only requires an end-biased histogram

7. Constructing the entire join and then using unweighted sampling on the result.

8. Frequencies for all values that occur l or more times.

9. Thereby saving having to compute the full join for the bulk of the tuples.

10. Since, tautologically, they are observed frequently and can be sketched easily.

11. By CF sampling with p being the relative fractions of tuples in each subset

Algorithm 2 Universe Sampling

Inputs:
$R_1(A,\dots), R_2(A,\dots)$
sample rate f
hash function h
Output: S, a $\texttt{SAMPLE}(R_1 \bowtie R_2, f)$
Init: $S_1, S_2 \coloneqq [\cdots], [\cdots]$
Begin:
// stream R_1
while R_1:
$t_1 = \texttt{next}(R_1)$
if $h(t_1.A) < f$: $S_1 \coloneqq S_1 \cup \{t_1.A\}$
// stream R_2
while R_2:
$t_2 = \texttt{next}(R_2)$
if $h(t_2.A) < f$: $S_2 \coloneqq S_2 \cup \{t_2\}$
$S \coloneqq S_1 \bowtie S_2$

for R_2. The cost incurred by the frequency partition sample strategy is $O(\alpha_2 |R_1 \bowtie R_2|)$ where

$$\alpha_2 \coloneqq \frac{\sum_{v \in D^{lo}} m_1(v)\, m_2(v) + r \times \frac{\sum_{v \in D^{hi}} m_1(v) m_2(v)^2}{\sum_{v \in D^{hi}} m_1(v) m_2(v)}}{\sum_{v \in D} m_1(v)\, m_2(v)}$$

3.1.4 Index sampling

If an index is available for the high-frequency values R_2^{hi}, in addition to just the frequencies themselves, then a more efficient version of frequency partition sample is possible; we can save having to complete the full join

$$\begin{aligned}\left(S_1 \cup R_1^{lo}\right) \bowtie R_2 &\equiv \left(S_1 \cup R_1^{lo}\right) \bowtie \left(R_2^{lo} \cup R_2^{hi}\right) \\ &\equiv \cancel{\left(S_1 \bowtie R_2^{hi}\right)} \cup \left(S_1 \bowtie R_2^{lo}\right) \cup \left(R_1^{lo} \bowtie R_2\right)\end{aligned}$$

and instead uses the same idea as in stream sample[12] to select a random tuple in R_2^{hi} per tuple in S_1. The cost incurred by the Index sample strategy is $O(\alpha_3 |R_1 \bowtie R_2|)$ where

$$\alpha_3 \coloneqq \frac{\sum_{v \in D^{lo}} m_1(v)\, m_2(v) + r}{\sum_{v \in D} m_1(v)\, m_2(v)}$$

3.1.5 Count sampling

Strictly speaking an index for R_2^{hi} isn't necessary and can be replaced by a scan across R_2^{hi} instead. See alg. 15.

3.2 Universe sampling

Given an attribute A and a perfect[13] hash function $h : A \to [0,1]$ we can compute $\texttt{SAMPLE}(R_1 \bowtie R_2, f)$ by hashing [8] tuples $t_1, t_2 \in R_1, R_2$ and rejecting those that fall outside of $[0, f]$ (see alg. 2). This guarantees when $t_1 \in R_1$ is sampled, all matching tuples $t_2 \in R_2$ are also sampled since

$$t_1.A = t_2.A \iff h(t_1.A) = h(t_2.A)$$

Hence, universe sampling produces a uniform sample of $R_1 \bowtie R_2$ (in expectation) since each tuples appears with probability f. Unfortunately, the samples are correlated: if

$$t_1.A = t_2.A = t_1'.A = t_2'.A$$

12. $t_1 \in S_1 \wedge t_2 \sim U\left(\{t \mid t \in R_2^{hi} \wedge t.A = t_1.A\}\right)$
13. Collision free.

then

$$\begin{aligned}(t_1, t_2) \in \texttt{SAMPLE}(R_1 \bowtie R_2, f) \iff \\ (t_1', t_2') \in \texttt{SAMPLE}(R_1 \bowtie R_2, f)\end{aligned}$$

This can lead to poor performance for approximate queries when the frequencies of attributes are highly concentrated. Consider sampling from relations R_1, R_2 each with n identical tuples. The variance of the estimator [3] for the join size

$$\frac{|S_1 \bowtie S_2|}{f} \approx |R_1 \bowtie R_2|$$

is n^4/f while the variance of the same estimator given uniform sampling is n^2/f^2, which is much lower when n is large.

3.3 Correlation based sampling

Olken sampling and its extensions produce uncorrelated samples by enforcing (through various means) that a tuple in R_1 is joined with only one tuple in R_2; this leads to *sample inflation*. Correlated sampling techniques, such as universe sampling, perform better with respect to sample inflation but suffer from poorer error estimates (see section 3.2). Correlation based sampling [11] aims to mitigate the issues of correlated sampling, while preserving the benefits, by maximizing *join randomness*. The join randomness of a sampling technique is the number of different possible samples that can be drawn by the technique given a fixed sample size (and given frequencies of attributes).

To motivate join randomness, consider naive sampling from K relations with common attribute $A \subseteq D$ that takes on $|D|$ distinct values. The maximum number of such samples depends combinatorially on the number drawn from each relation:

$$\#\text{Samplings} = \prod_{j=1}^{|D|} \prod_{k=1}^{K} C_{m_{s_k}(a_j)}^{m_k(a_j)}$$

where $m_k(a_j)$ are frequencies of value a_j in R_k and $m_{s_k}(a_j)$ are the allocations for sample S_k of R_k (C_b^a is the binomial coefficient). For example, suppose R_1, R_2 include an attribute A with two distinct values $\{a_1, a_2\}$ and with frequencies

$$\begin{aligned}m_1(a_1) = m_2(a_1) = 10 \\ m_1(a_2) = m_2(a_2) = 20\end{aligned}$$

If we sample both relations, thereby producing S_1 and S_2, and fix the sample size (the number sampled from either S_1 or S_2) to $M = 30$ then the number of possible samples is

$$C_{m_{s_1}(a_1)}^{10} \times C_{m_{s_2}(a_1)}^{10} \times C_{m_{s_1}(a_2)}^{20} \times C_{m_{s_2}(a_2)}^{20}$$

For various allocations of we see orders of magnitude differences in the number of possible samplings:

$m_{s_1}(a_1)$	$m_{s_2}(a_1)$	$m_{s_1}(a_2)$	$m_{s_2}(a_2)$	# Samplings
5	5	10	10	2.2×10^{15}
3	7	8	12	2.3×10^{14}
2	3	12	13	5.3×10^{13}
1	1	14	14	1.5×10^{11}

Algorithm 3 MaxRand Join

Inputs:
relations $R_1(A,\dots),\dots,R_K(A,\dots)$
$m_{s_k}(a_j)$
Output: S, a SAMPLE$(R_1 \bowtie R_2, f)$
Init:
$S[\cdots] = []$
Begin:
for $i \coloneqq 1$ to K:
// stream
while R_i:
$S[i] \coloneqq [\cdots]$
for $j \coloneqq 1$ to $|D|$:
// independent reservoirs
// for each a_j
$S[i] \coloneqq S[i] \cup$ ReservoirSample$(R_i, m_{s_k}(a_j), a_j)$
$S \coloneqq S[1] \bowtie \cdots \bowtie S[K]$

3.3.1 MaxRand join

In general, the allocation that maximizes the number of possible samplings, per distinct value a_j of attribute A, for sample S_k, for a given total sample size M, is [11]:

$$m_{s_k}(a_j) \coloneqq \text{round}\left(\frac{M \cdot m_k(a_j)}{\sum_{j=1}^{|D|}\sum_{k=1}^{K} m_k(a_j)}\right) \tag{3.1}$$

The MaxRand join algorithm uses this result to produce a maximally random join (see alg. 3). The total cost of MaxRand join is the cost of constructing $m_{s_k}(a_j)$ for $k = 1,\dots,K$ and $j = 1,\dots,|D|$, plus the cost of reservoir sampling each R_i; let

$$N \coloneqq \max_{k,j} m_{s_k}(a_j)$$

then the total cost is

$$O\left(|D|\,|K| + \sum_{i=1}^{K} N\left(1 + \log\left(\frac{|R_i|}{N}\right)\right)\right)$$

3.4 Ripple join

Ripple join [7] belongs to a class of techniques that address the online aggregation problem directly. Thus, ripple join isn't an operand sampling scheme per se but a streaming join algorithm that, by virtue of building the join incrementally, produces a sample of $R_1 \bowtie \cdots \bowtie R_K$. As the full join is approached running estimates for various aggregations converge.

In order to understand ripple join we first need to understand the *streaming nested loops algorithm* [9]. For example, for relations R_1, R_2, with $|R_2| < |R_1|$, streaming nested loops first samples $t_2 \in R_2$ and then R_1 is looped over in search of tuples t_i that satisfy the join condition. Once all such tuples are discovered the running estimate of the aggregation function is updated. Note that for conventional batch processing, a query optimizer would select R_2 as the outer relation, while for streaming processing the reverse is preferable[14]. Despite this optimization, if $|R_1|$ is nontrivial, then wait times for updates can be excessively long. Furthermore, if the aggregation is "insensitive" to the attribute values in R_1 then this leads to poor convergence[15]. Such an example is

$$\texttt{SELECT AVG}\,(R_2.A + R_1.B/10000000)\ \texttt{FROM}\,R_1, R_2$$

where $R_1.B/10000000$ most likely does not move the average on each sampled tuple. Ripple join address both of these issues by adaptively choosing which is the inner relation and which is the outer relation in nested loops (see fig. 3.1) and the *aspect ratios* β_1,β_2 of the join. The aspect ratios β_1,β_2 determine how many times R_2 is sampled per outer loop and how many tuples are selected from R_1 per inner loop.

Using ripple join we can estimate various aggregations by calculating them over the "current" set of sampled tuples. Unsurprisingly, the variance of these estimates depends on the aspect ratios β_i:

$$\hat{\sigma}^2 \coloneqq \sum_{k=1}^{K} \frac{\hat{d}(k)}{\beta_k}$$

where $\hat{d}(k)$ is an estimator for a term $d(k)$ that takes a particular form for each aggregation SUM, COUNT, or AVG [12]. Therefore, in turn, the confidence intervals can be minimized subject to an upper bound c on the product of the aspect ratios[16]:

$$\begin{aligned} \text{minimize} \quad & \sum_{k=1}^{K} \frac{\hat{d}(k)}{\beta_k} \\ \text{such that} \quad & \prod_{k=1}^{K} \beta_k \le c \\ & 1 \le \beta_k \le |R_i| \\ & \beta_k \in \mathbb{N} \end{aligned} \tag{3.2}$$

This is a non-linear integer programming problem which is, in general, NP-hard[17]. Ripple join solves a relaxation such that

$$\beta_k^* = \left(\frac{c}{\prod_{k=1}^{K}\hat{d}(k)}\right)^{1/K} \hat{d}(k)$$

where $\hat{d}(k)$ is the estimate from the previous iteration.

3.4.1 Hash ripple join

In the case of an equijoin we can use hashing to amortize lookup of previously matched tuples; when a new tuple is fetched from either relation it must be combined with all previously matched tuples from the other relation. If previously sampled tuples are hashed on the join column then it is possible to select them efficiently. This implies a reduction in the cost (eq. 3.2) of the $n$sampling steps to only $\sum_{k=1}^{K}\beta_k$. This then implies the optimal aspect ratios are

$$\beta_k^* = \frac{c\sqrt{\hat{d}(k)}}{\sum_{j=1}^{K}\sqrt{\hat{d}(j)}}$$

14. Such that running estimates can be updated more frequently.

15. Since subsequent tuples from R_1 don't contribute new "statistical information".

16. Since the product of the aspect ratios is the number of matches computed and hence I/O.

17. By reduction from minimum vertex cover.

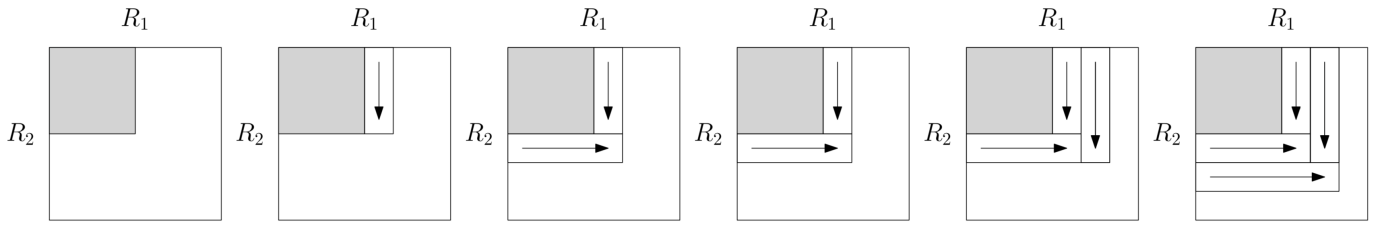


Figure 3.1: Ripple Join for $R_1 \bowtie R_2$ [12].

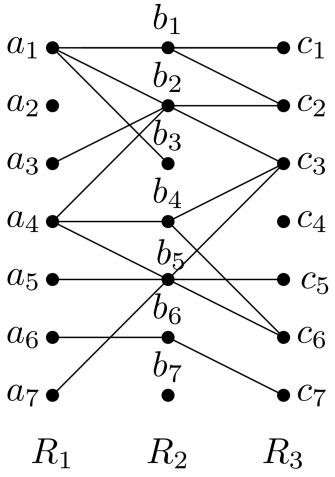


Figure 3.2: Wander join for $R_1(A,B) \bowtie R_2(B,C) \bowtie R_3(C,D)$ [12].

3.5 Wander join

Wander join[12], like ripple join, is a solution to the online aggregation problem. The key idea of wander join is to model a join over K relations as a *join graph*[18] i.e. a K-partite graph where each partition corresponds to a relation and edges between vertices indicate a match on the join condition. Once such a model is adopted one can then further model streaming joins as random walks in the graph. Such random walks can then be used to construct unbiased estimators of various aggregations.

Let $R_1(A,B), R_2(B,C), R_3(C,D)$ be the relations of interest,

$$J \coloneqq R_1(A,B) \bowtie R_2(B,C) \bowtie R_3(C,D)$$

be the join of interest, and $d_{i+1}(t_i)$ be the number neighbors of t_i in relation R_{i+1} i.e the number of tuples in R_{i+1} that join with t_i. A random walk (or path) γ can be sampled from J by picking a tuple (vertex) $t_1 \in R_1$ uniformly at random and then randomly selecting tuples $t_2 \in R_2, t_3 \in R_3$ such that $t_2.B = t_1.B \wedge t_3.C = t_2.C$ (see figure 3.2). Then, given an expression $g(R_1,R_2,R_3)$, we can construct a one point estimate of $\texttt{SUM}(g(R_1,R_2,R_3))$, for example, by

$$X_\gamma \coloneqq \frac{g(\gamma)}{p(\gamma)} \quad p(\gamma) \coloneqq \frac{1}{|R_1|}\frac{1}{d_2(t_1)}\frac{1}{d_3(t_2)}$$

where $p(\gamma)$ is the probability of sampling γ. This estimator is unbiased [10] and therefore the average of many such estimators is also unbiased. That is to say, for a set of sampled paths Γ

$$\hat{\mu}_\Gamma \coloneqq \frac{1}{|\Gamma|}\sum_{\gamma\in\Gamma} X_\gamma \Rightarrow E[\hat{\mu}_\Gamma] = \texttt{SUM}(g(R_1,R_2,R_3))$$

with variance

$$\hat{\sigma}^2_\Gamma \coloneqq \frac{1}{|\Gamma|-1}\sum_{\gamma\in\Gamma}(X_\gamma - \hat{\mu}_\Gamma)^2$$

Furthermore the *walk plan*, i.e. the order of the relations sampled, can be optimized, in terms of estimator variance; by the law of total variance

$$\text{Var}\left(\frac{1}{|\Gamma|}\sum_{\gamma\in\Gamma} X_\gamma\right) = \frac{\text{Var}[X_\gamma]\,E[T]}{t}$$

where T is the running time of a single random walk and t is the total time taken to sample Γ. Thus, by estimating both $\text{Var}[X_\gamma]$ and $E[T]$ from the walks themselves, we choose the walk plan that minimizes the estimator variance.

3.5.1 Wander join for cyclic and acyclic joins

As described, wander join applies to chain joins (see fig. 2.1a). The algorithm can be extended to include merely acyclic joins (see fig. 2.1b) by incorporating "jumps" into the random walk. For example, consider

$$R_1(A,B) \bowtie R_2(B,C,D) \bowtie R_3(C,E) \bowtie R_4(D,F) \bowtie R_5(F,G)$$

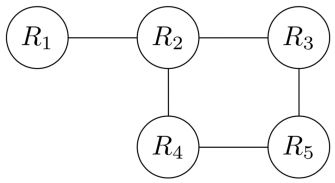
and fix a *walk order* R_1, R_2, R_3, R_4, R_5. On sampling $t_3 \in R_3$ we "jump back" to $t_2 \in R_2$ and sample $t_4 \in R_4$ and so on. Note that different walk orders lead to different execution time and estimator variance. Given relations $R_1, \ldots, R_K$ and a walk order $\left(R_{\lambda(1)}, \ldots, R_{\lambda(K)}\right)$, define $R_{\eta(i)}$ to be the relation corresponding to the "parent" of $R_{\lambda(i)}$. Then for the path $\gamma \coloneqq \left(t_{\lambda(1)}, \ldots, t_{\lambda(K)}\right)$

$$p(\gamma) = \frac{1}{|R_{\lambda(1)}|}\prod_{k=2}^{K}\frac{1}{d_{\lambda(i)}\left(t_{\eta(i)}\right)}$$

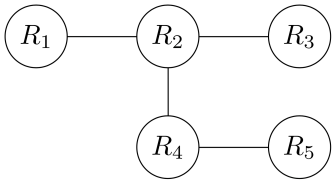
This same adjustment can be extended to work for cyclic joins by first computing a directed spanning tree[19] of the join query graph (see fig. 3.3).

18. Distinct from the join hypergraph.

19. A *directed tree* is a tree in which every edge increases the distance from the root. A directed spanning tree of a graph G is spanning tree of G that is also a directed tree.



(a) Cyclic join.



(b) Spanning tree of cyclic join.

Figure 3.3: Join for $R_1(A,B)$, $R_2(B,C,D)$, $R_3(C,E)$, $R_4(D,F)$, $R_5(F,E)$.

Algorithm 4 Upper Bound Join

Inputs: $R_1, R_2, \ldots, R_K$, $W(t)$
Output: $t \in J$ or reject
Init: $t \coloneqq \perp$
Begin:
  for $i \coloneqq 1$ to K:
    $W' \coloneqq W(t)$
    $W \coloneqq W(t \rtimes R_i)$
    if $U(0,1) > W/W'$:
      reject
    // $t' \to \cdots$ is a lambda
    $w(t') \coloneqq t' \to W(t')/W(t \rtimes R_i)$
    $t \coloneqq \mathtt{BBWR2}(t \rtimes R_i, 1, w(t'))$
  return t

3.6 Upper bound join

For any join $J \coloneqq R_1 \bowtie R_2 \bowtie \cdots \bowtie R_K$ and any $t \in R_i$ define

$$w(t) \coloneqq |t \bowtie R_{i+1} \bowtie \cdots \bowtie R_K|$$

with $w(t) \coloneqq 1$ for $t \in R_K$. Note that every t participates in J proportional to $w(t)$. Thus, if we could approximate $w(t)$ for all $t \in J$ we could then produce an accurate sample of J. Of course, $w(t)$ is not often available but we can make use of a proxy $W(t)$ that has properties

$$\begin{gathered} W(t) \geq w(t) \text{ for all } t \\ W(t) = w(t) = 1 \text{ for all } t \in R_K \\ W(t) \geq W(t \rtimes R_{i+1}) \text{ for all } t \in R_i \text{ with } i < K \end{gathered} \tag{3.3}$$

where $\rtimes$ is a right semi-join[20]. Given such an $W(t)$, we can use alg. 4 to sample J with replacement. The algorithm returns each join result t with probability $1/W(t)$ [15] where

$$W(t) \geq |R_1 \bowtie \cdots \bowtie t \bowtie \cdots \bowtie R_K|$$

Note that employing upper bounds in this way is in effect a generalization of Olken sampling (see sec. 3.1) and frequency partition sampling (see sec. 3.1.3); in the case of frequency partition sampling $W(t) \coloneqq w(t)$. In the case of Olken sampling, let $M_i \coloneqq \max_{v \in D_i} m_i(v)$ be the maximum frequency of the join attribute A_i of relation R_i and for all $t \in R_i$ set

$$W(t) \coloneqq \prod_{j=i+1}^{K} M_i$$

Then $W(t)$ obeys properties 3.3. Essentially this functions as an assumption that every tuple in R_i joins with M_{i+1} tuples in R_{i+1}.

A tighter bound on $|J|$ can be constructed by considering *fractional edge covers* of the join: the fractional edge cover of a join J assigns a weight $u_i \geq 0$ to each R_i, in proportion to the role R_i plays in J, such that for every attribute in A_j in J we have

$$\sum_{i \in I} u_i \geq 1$$

where $I = \{i \mid R_i(A_j, \ldots)\}$. For any fractional edge cover

$$|J| \leq \prod_{i=1}^{K} |R_i|^{u_i}$$

and a tight upper bound [1] can be found by minimizing[21]. Define this tight upper bound AGM and then we can set

$$W(t) \coloneqq \mathrm{AGM}(R_{i+1} \bowtie \cdots \bowtie R_K)$$

for $t \in R_i$.

4 Experiments

We conducted experiments to test how well each of the above techniques reproduces the distribution of a uniform sampling of a K-chain-join $J \coloneqq |R_1 \bowtie \cdots \bowtie R_k|$. To do this we generate $2K$ (such that each R_i has two join attributes) realistic data sets that agree with Benford's law by generating samples from a ratio of χ^2_1 distributions [5]. We then "quantize" each sample to the two leading non-zero digits thereby producing at most 90 different valid join keys. Finally, we produce a benchmark sample of J (by performing the full join and using uniformly random sampling). We compare the benchmark sample against samples generated by each of the techniques using the standard Kolmogorov–Smirnov (KS) test for testing whether two samples are sampled from the same distribution. We iterate across n the number of samples, K the number of relations, and f the fraction sampled, repeating each experiment 100 times and plot the distributions of the KS test statistics d and p-values.

5 Conclusion

References

[1] Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. In *Proceedings of the 2008 49th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '08, pages 739–748, USA, 2008. IEEE Computer Society.

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

[3] Yu Chen and Ke Yi. Two-level sampling for join size estimation. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 759–774, New York, NY, USA, 2017. Association for Computing Machinery.

20. $R_1 \rtimes R_2$ is the set of all tuples in R_2 for which there is a matching tuple in R_1.

21. In practice $\sum_{i=1}^{K} u_i \log(|R_i|)$.

[4] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.

[5] Anton K. Formann. The newcomb-benford law in its relation to some common distributions. *PLOS ONE*, 5(5):1–13, 05 2010.

[6] W. R. Gilks and P. Wild. Adaptive rejection sampling for gibbs sampling. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 41(2):337–348, 1992.

[7] Peter J. Haas and Joseph M. Hellerstein. Ripple joins for online aggregation. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, pages 287–298, New York, NY, USA, 1999. Association for Computing Machinery.

[8] Marios Hadjieleftheriou, Xiaohui Yu, Nick Koudas, and Divesh Srivastava. Hashed samples: Selectivity estimators for set similarity selection queries. *Proc. VLDB Endow.*, 1(1):201–212, August 2008.

[9] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. *SIGMOD Rec.*, 26(2):171–182, June 1997.

[10] D. G. Horvitz and D. J. Thompson. A generalization of sampling without replacement from a finite universe. *Journal of the American Statistical Association*, 47(260):663–685, 1952.

[11] Niranjan Kamat and Arnab Nandi. A unified correlation-based approach to sampling over joins. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*, SSDBM '17, New York, NY, USA, 2017. Association for Computing Machinery.

[12] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. Wander join: Online aggregation via random walks. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 615–629, New York, NY, USA, 2016. Association for Computing Machinery.

[13] Kim-Hung Li. Reservoir-sampling algorithms of time complexity o(n(1 + log(n/n))). *ACM Trans. Math. Softw.*, 20(4):481–493, December 1994.

[14] Frank Olken. *Random sampling from databases*. PhD thesis, University of California, Berkeley, 1993.

[15] Zhuoyue Zhao, Robert Christensen, Feifei Li, Xiao Hu, and Ke Yi. Random sampling over joins revisited. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1525–1539, New York, NY, USA, 2018. Association for Computing Machinery.

6 APPENDIX

6.1 Fundamental sampling algorithms

We briefly describe some fundamental sampling algorithms that will ultimately be employed as blackbox primitives in more sophisticated algorithms.

6.1.1 Rejection sampling

Rejection sampling is a means of generating samples from some *target distribution* $p(x)$ that cannot be sampled directly but for which we can construct a *proposal distribution* $g(x)$ that encompasses the target distribution. It is based on the observation that to sample $p(x)$, one can perform a sampling of the region under the graph of $g(x)$ distribution[22], and reject all such samples that fall outside of the graph of $p(x)$. In the simplest case, the proposal distribution is a uniform distribution $U(a,b) \times U(0,M)$, where $[a,b]$ is the support of $p\,(x)$ and M is the least upper bound of $p(x)$. Then the algorithm is straightforward (see alg. 5). The number of samples until "acceptance" follows a geometric distribution with probability $1/M$ and thus has expected number M. Alternatively, more efficient algorithms exist [6].

22. For probability density function $p(x)$, the set of ordered pairs $\{(x,y) \mid 0 \leq y \leq p(x)\}$.

Algorithm 5 Rejection Sampling

Inputs:
  target distribution $p\,(x)$
  with support $[a,b]$ and max M
Output: (x,y) a draw from p
Init: $U_1 \sim U(a,b)$, $U_2 \sim U(0,M)$
Begin:
  while $p\,(U_1) <$
$p\,(U_2)$: // reject if outside
    $U_1 \sim U(a,b)$
    $U_2 \sim U(0,M)$
  $(x,y) \coloneqq (U_1,U_2)$

Algorithm 6 Naive Reservoir Sampling

Inputs: stream S, sample size k
Output: reservoir A with k samples
Init: for $i \coloneqq 1$ to k: $A[i] \coloneqq \text{next}(S)$
Begin:
  // replace elements with gradually
  // decreasing probability
  while S:
    // randomInteger is inclusive
    $j \coloneqq \text{randomInteger}(1,i)$
    if $j \leq k$:
      $A[j] \coloneqq \text{next}(S)$

6.1.2 Reservoir sampling without replacement

Reservoir sampling selects a simple random sample, without replacement, of K items from a population of unknown size n in a single pass over the items. The simplest such algorithm maintains a *reservoir* of size K and swaps out elements according to the desired sampling probability. See alg. 6. This naive algorithm runs in $O\,(n)$ time since it calls random number generator (RNG) for each element in the stream.

The naive algorithm can be improved upon by instead discarding elements explicitly rather than including elements explicitly. We describe this algorithm constructively [13]:

1) We conceive of naive reservoir sampling as assigning draws u_i from $U\,(0,1)$ to each entry in the S and then selecting the bottom k elements. This proceeds by initially filling the reservoir and then successively replacing the largest element in the reservoir if the u_i associated with s_i is smaller than the largest element in the reservoir.
2) If fact we don't actually need to maintain the set of draws u_i for the entire reservoir, just the largest in the reservoir. Call that that value θ.
3) The u_i value of the "next" s_i to enter the reservoir is actually be distributed $U\,(0,\theta)$.
4) If $X_i \sim U(0,\theta)$ then for $Y = \max\,(X_1,\ldots,X_k)$ we have

$$P\,(Y \leq y) = \left(\frac{y}{\theta}\right)^n$$

by using $P\,(X_i \leq x) = x/\theta$ and by X_i being i.i.d.

Algorithm 7 Optimal Reservoir Sampling

```
Inputs: stream S, sample size k
Output: reservoir A with k samples
Init:
  // initialize the reservoir
  for i ≔ 1 to k: A[i] ≔ next(S)
Begin:
  // note that θ = 1 since A consists
  // of k samples from U(0,1).
  // random() draws from U(0,1)
  θ ≔ exp(log(random())/k)
  while S:
    k ≔ ⌊log(U)/log(1−θ)⌋
    // discard k elements
    while S: next(S)
    // replace random element
    // since we don't need to compare
    A[randomInteger(1,k)] ≔ next(S)
    // update max given that stored
    // element was drawn from U(0,θ)
    θ ≔ θ · exp(log(U)/k)
```

Therefore, by using inverse CDF

$$U = \left(\frac{Y}{\theta}\right)^k \Rightarrow Y \sim \theta U^{1/k} \equiv \theta \exp\left(\frac{\log(U)}{k}\right)$$

5) The number of elements K discarded follows a geometric distribution where the probability of success is θ[23]. Thus, again by using inverse CD

$$U = 1 - (1-\theta)^{K+1} \Rightarrow$$
$$\log(1-U) = (K+1)\log(1-\theta) \Rightarrow$$
$$K = \left\lfloor \frac{\log(U)}{\log(1-\theta)} \right\rfloor$$

where we use the fact that $1-U$ is also distributed $U(0,1)$.

6) We don't actually need to compare against θ but just update it as a precursor to computing K.

Thus, we can more efficiently reservoir sample by taking "jumps" and only querying the RNG for included elements of the stream. See alg. 7. The running time then is $O(k(1+\log(n/k)))$.

6.1.3 Unweighted streaming sampling with replacement

We present two algorithms for unweighted sequential sampling with replacement; we omit proofs that these algorithms in fact uniformly sample[**?**]. The first algorithm `BBU1` fills the output array with copies[24] of successive tuples from streaming relation S. This, in effect, simulates sampling with replacement. The disadvantage of the alg. 8 is that the total size n of the relation is a necessary prerequisite. Algorithm `BBU2` improves on algorithm `BBU1` by eliminating that prerequisite (see alg 9).

23. Recall that the draws u_i are drawn from $U(0,1)$ and for some s_i to enter the reservoir it must be smaller than the current maximum element θ and hence $P(X \leq \theta) = \theta$.

24. Whose quantity depends on a random variable drawn from a binomial distribution.

Algorithm 8 BBU1

```
Inputs:
  stream S
  sample size k
  n ≔ |S|
Output: array A with k samples
Init: x ≔ k, i ≔ 0
Begin:
  while S and x > 0:
    t ≔ next(S)
    X ∼ B(x, 1/(n−i))
    // fill A with X copies of t
    for j ≔ 1 to X:
      A[i+j] ≔ t
    x ≔ x − X
    i ≔ i + 1
```

Algorithm 9 BBU2

```
Inputs:
  stream S
  sample size k
Output: array R with k samples
Init: N ≔ 0, A[1,...,k] = 0
Begin:
  while S:
    t ≔ next(S)
    N ≔ N + 1
    // set A[j] ≔ t with
    // probability 1/N
    for j ≔ 1 to k:
      X ∼ U(0,1)
      if X ≤ 1/N:
        A[j] ≔ t
```

6.1.4 Weighted streaming sampling

The precise semantics of weighted sampling are as such: given a streaming relation S with cardinality n, where each tuple $t \in S$ has associated weight $w(t)$, a weighted, with replacement, sample is produced by drawing $f \cdot n$ tuples with probability[25] proportional to $w(t)$. We can extend algorithms 8, 9 to respect these semantics; see algorithms 10, 11.

6.2 Technique implementations

25. Implying normalization if $\sum_{t \in S} w(t) \neq 1$

Algorithm 10 BBWR1

Inputs:
stream S
sample size k
$n \coloneqq |S|$
weights $w(t)$
Output: array A with k samples
Init: $x \coloneqq k$, $i \coloneqq 0$, $W \coloneqq \sum_{t \in S} w(t)$
Begin:
while S and $x > 0$:
$t \coloneqq \texttt{next}(S)$
$X \sim B\left(x, \frac{w(t)}{W-i}\right)$
// fill A with X copies of t
for $j \coloneqq 1$ to X:
$A[i+j] \coloneqq t$
$x \coloneqq x - X$
$i \coloneqq i + w(t)$

Algorithm 11 BBWR2

Inputs:
stream S
sample size k
weights $w(t)$
Output: array R with k samples
Init: $W \coloneqq 0$, $A[1, \ldots, k] \coloneqq 0$
Begin:
while S:
$t \coloneqq \texttt{next}(S)$
$W \coloneqq W + w(t)$
// set $A[j] \coloneqq t$ with
// probability $w(t)/W$
for $j \coloneqq 1$ to k:
$X \sim U(0,1)$
if $X \leq \frac{w(t)}{W}$:
$A[j] \coloneqq t$

Algorithm 12 Stream Sampling

Inputs:
$R_1(A, \ldots), R_2(A, \ldots)$
$k \coloneqq \lceil f \cdot |R_1 \bowtie R_2| \rceil$
Output: $S \equiv \texttt{SAMPLE}_{WR}(R_1 \bowtie R_2, f)$
Init:
$S[1, \ldots, k] \coloneqq 0$
$w(t) \coloneqq m_2(t.A)$ for $t \in R_1$
Begin:
$S_1 \coloneqq \texttt{BBWR2}(R_1, k, w(t))$
for $i \coloneqq 1$ to k:
$t_1 \coloneqq \texttt{next}(S_1)$
$t_2 \sim U(\{t \mid t \in R_2 \wedge t.A = t_1.A\})$
$S[i] \coloneqq (t_1, t_2)$

Algorithm 13 Group Sampling

Inputs:
$R_1(A, \ldots), R_2(A, \ldots)$
$k \coloneqq \lceil f \cdot |R_1 \bowtie R_2| \rceil$
Output: $S \equiv \texttt{SAMPLE}_{WR}(R_1 \bowtie R_2, f)$
Init:
$S[1, \ldots, k] \coloneqq 0$
$w(t) \coloneqq m_2(t.A)$ for $t \in R_1$
Begin:
$S_1 \coloneqq \texttt{BlackBoxWR2}(R_1, k, w(t))$
for $i \coloneqq 1$ to k:
$t_1 \coloneqq S_1[i]$
$t_2 \coloneqq \texttt{BlackBoxU2}(t_1 \bowtie R_2, 1)$
$S[i] \coloneqq (t_1, t_2)$

Algorithm 14 Frequency Partition Sampling

Inputs:
$R_1(A, \ldots), R_2(A, \ldots)$, $A \subseteq D$
$k \coloneqq \lceil f \cdot |R_1 \bowtie R_2| \rceil$
// low, high frequency values in R_2
D^{lo}, D^{hi}
$R_2^{lo}, R_2^{hi} \coloneqq R_2|_{D^{lo}}, R_2|_{D^{hi}}$
$w_2(t) \coloneqq m_2(t.A)$ for $t \in R_2^{hi}$
Output: $S \equiv \texttt{SAMPLE}_{WR}(R_1 \bowtie R_2, f)$
Begin:
// stream R_1
while R_1:
// partition R_1
$R_1^{lo} \coloneqq R_1|_{D^{lo}}$
$R_1^{hi} \coloneqq R_1|_{D^{hi}}$
// sample but also collect
// stats $w_1(t)$ on R_1^{hi}
$S_1, w_1(t) \coloneqq \texttt{BBWR2}(R_1^{hi}, k, w(t))$
// integrate/combine stats
$w(t) \coloneqq w_1(t) \cup w_2(t)$
// from $w(t)$ you can approximate
$n_{hi} \coloneqq |R_1^{hi} \bowtie R_2^{hi}|$
$R_1^* \coloneqq S_1 \cup R_1^{lo}$
// stream join
while $J^* \coloneqq R_1^* \bowtie R_2$:
$n_{lo} \coloneqq J^*|_{D^{lo}}$ // i.e. $n_{lo} \coloneqq |R_1^{lo} \bowtie R_2^{lo}|$
// partition J^*
$J^{lo} \coloneqq \texttt{BBU2}(J^*|_{D^{lo}}, k)$
// $S_1 \equiv \{s_i\}$ and just
// like in Group-Sample
$J^{hi} \coloneqq \texttt{BBU2}(s_i \bowtie J^*|_{D^{hi}}, k)$
// # of heads and tails
// $p = \frac{n_{hi}}{n_{lo}+n_{hi}}, 1-p = \frac{n_{lo}}{n_{lo}+n_{hi}}$
$k_{lo}, k_{hi} \coloneqq B(k, p)$
$S^{lo} \coloneqq \texttt{BlackBoxUWoR2}(J^{lo}, k_{lo})$
$S^{hi} \coloneqq \texttt{BlackBoxUWoR2}(J^{hi}, k_{hi})$
$S \coloneqq S^{lo} \cup S^{hi}$

Algorithm 15 Count Sampling

Inputs:
  k
  $R_2(A, \dots)$
  $S_1 \subseteq R_1^{hi}$
Output: $S \equiv (S_1 \bowtie R_2^{hi})$
Init:
  H // hash table for counting
Begin:
  while S_1:
    $t \coloneqq \texttt{next}(S_1)$
    // count number of tuples such that
    // $t.A = v$
    $H[t.A] \coloneqq H[t.A] + 1$
  // sample $S_2 \subseteq R_2$ such that
  // the number of tuples t with
  // $t.A \equiv v$ is exactly v
  while $S_2 \coloneqq \texttt{BBWR2}(R_2, H)$:
    $t_1 \coloneqq \texttt{next}(S_2)$
    $t_2 \coloneqq \texttt{BlackBoxWoR}(t_1 \bowtie S_1, 1)$
    $S[i] \coloneqq (t_1, t_2)$