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, i.e.

$$\texttt{SAMPLE}\left(\texttt{JOIN}(R_1, R_2)\right) \neq \texttt{JOIN}\left(\texttt{SAMPLE}\left(R_1\right), \texttt{SAMPLE}\left(R_2\right)\right)$$

We aim to study the regimes under which the operations do commute and what are the tradeoffs when they don't.

Sampling through Joins

Maksim Levental, Greg Pauloski

Contents

1 Introduction

words about why sampling joins is important e.g. online aggregation problem.

1.1 Definitions

Let $R(A_i)$, be a relation over some attributes A_i, with cardinality $n := |R|$. For $0 \leq f \leq 1$, $\texttt{SAMPLE}(R, f)$ is defined to be a uniform random sample of tuples in R. This produces a subset $R' \subseteq R$ with $|R'| = f \cdot n$. 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.
- *Independent Coin Flip* (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

- Given a WR sampling process we can transform to WoR by performing duplicate removal (*deduplication*).
- Given a CF sampling process we can transform to WoR by CF sampling with $f' = f + \varepsilon$ and then WoR sampling the CF sample down to a f fraction [2].
- Given WoR sampling process we can transform to WR by sampling with replacement from the WoR sample.
- Given a WR or WoR sampling process **it is impossible** to produce a CF sample since with CF there is a small, but nonzero, probability that the sample is empty (which is impossible for both WR and WoR given $f > 0$).

Another dimension of sampling is whether the samples are *correlated*[1] or *uncorrelated*; we will see that correlated samples lead to higher error rates than uncorrelated samples (see sec. 2.2). 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 where the relation is never materialized (as in when it's produced 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.

Joins are defined according to Codd's relational model [3]; the *natural join* $R \bowtie S$ of two relations R, S is defined as the set of all pairs of tuples in R and S that are equal on their common attributes. If R, S have no common attributes then

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

$R \bowtie S \equiv R \times S$, the cartesian product[2] of R, S. We will also have need of considering joins between relations that have a predicated relationship between some colums; define a θ-join $R \bowtie_{A\theta B} S$ of relations $R(A, \dots), S(B, \dots)$ on attributes A, B and some binary operator $\{<, \leq, =, \neq, >, \geq\}$ as all tuples in $R \bowtie S$ 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*.

1.2 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 $J := R_1 \bowtie R_2$, with $n := |J|$, and further suppose R_1, R_2 only have in common attribute $A \subseteq D$ for some domain D. For all $v \in D$, let $m_1(v), m_2(v)$ be the quantities (frequencies) of tuples in R_1, R_2, respectively, for which attribute A takes on value v[3]. Then

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

for $i = 1, 2$. 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

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

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. Then, suppose we wish to construct $\texttt{SAMPLE}(R_1 \bowtie R_2, f)$; we should expect that half 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[4]

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

The crux of the join sampling problem (and it's potential solution) is that, for a sample $\texttt{SAMPLE}(R_1 \bowtie R_2, f)$, 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:

$$\begin{aligned} R_1 \bowtie R_2 = \{&(a_1, b_0, c_1), \dots, (a_1, b_0, c_K), \\ &(a_2, b_1, c_0), \dots, (a_2, b_K, c_0)\} \end{aligned} \quad (1.1)$$

2. Sometimes called the *cross join* of R, S.
3. For $v \in D \backslash A$, i.e. $v \in D$ but $v \notin A$, we have $m_i(v) = 0$.
4. $\left(1 - \frac{1}{k+1}\right) \cdot \left(1 - \frac{1}{k+1}\right) = 1 - O\left(\frac{1}{k}\right)$.

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 uniform 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 sampling [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 uniform 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

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

In effect, this shows that we cannot commute $\texttt{SAMPLE}$ with $\bowtie$. However, this does not preclude the possibility of *non-uniform* random samples of R_1, R_2 being used to construct a uniform random sample of $R_1 \bowtie R_2$. Though, even with such non-uniform sampling there are minimum lower bounds on what fractions of operand relations need to be sampled:

Theorem 3. *To produce* $\texttt{SAMPLE}(R_1 \bowtie R_2, f)$ *from* S_1, S_2 *it's the case that* f_1, f_2 *need to be such that* $f_1 \times f_2 \geq f$.

1.3 Online aggregation for joins

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 $\texttt{AGG} \in \{\texttt{SUM}, \texttt{AVG}, \texttt{COUNT}, \dots\}$, $g(R_1, R_2, \dots, R_K)$ is an expression that involves any attributes of the relations $R_1, R_2, \dots, R_K$, θ is the aforementioned join condition operator, and σ_φ *selects* tuples that satisfy the predicate 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*[5] 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.

2 Techniques

As in previous sections we consider the problem of constructing a uniform random $\texttt{SAMPLE}\left(R_1 \bowtie R_2, f\right)$ (for various sampling semantics) by means of sampling the constituent relations R_1, R_2. We also restrict ourselves to joins over one common attribute $A \subseteq D$; where the extension to more complicated joins isn't obvious we discuss necessary adjustments. In the proceeding we partition the approaches to resolving the sampling problem according to how much information exists for each of the relations R_1, R_2:

- *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.
- *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 have a classification for strategies according to whether we have statistics (full or partial) and indices for both R_1, R_2, neither, or some in between (see tbl. 1).

2.1 Olken sample

Olken sampling only applies in the best possible case; 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. 4.1.1) to produce tuples in $\texttt{SAMPLE}_{WR}\left(R_1 \bowtie R_2, f\right)$ by sampling from tuples R_1 in direct proportion to their frequency in R_2 (see alg. 1). For $M := \max_{v\in D} m_2(v)$ for $v \in A \subseteq D$, Olken sampling produces a with-replacement sample of $R_1 \bowtie R_2$ and requires Mn_1/n iterations for each output tuple [2], where $n := |R_1 \bowtie R_2|$ and $n_1 := |R_1|$.

2.1.1 Stream sample

In the circumstances where we have only streaming access to R_1 we can use weighted streaming sampling (see sec. 4.1.4) with weights for $t \in R_1$ defined as $w(t) := m_2(t.A)$ and join each such tuple with a randomly selected tuple from R_1. See alg. 10. Note that we do constant work per ultimately included tuple.

2.1.2 Group sample

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`. We perform the second sampling using unweighted sampling with replacement (see sec. 4.1.3). The cost of this strategy depends on the cardinalities $|t_i \bowtie R_2|$:

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

Note that in the case of a foreign-key join (where $m_2(v) \in \{0, 1\}$) and this strategy compares very favorably to naive sampling[6] (the only other possible strategy for this set of circumstances).

2.1.3 Frequency-partition sample

Suppose we now only have an end-biased histogram[7] 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. These D^{hi} values also inflate the join between. Therefore we can recover efficiency by taking a hybrid approach; employing Group-Sample strategy on D^{hi} (thereby saving having to compute the full join for the bulk of the tuples) and naive 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 (since, tautologically, they are observed frequently and can be sketched easily).

The remaining issue is how to determine the allocation of the sample to each group/strategy: take $k := \lceil f \cdot |R_1 \bowtie R_2| \rceil$ from each subset and then cull in each subset in order to reduce the total quantity (by CF sampling with p being the relative fractions of tuples in each subset). See alg. 12. The key benefit of this algorithm is that only requires an end-biased histogram for R_2. The

Algorithm 1 Olken Sampling

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

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

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

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

Table 1: Sampling techniques

Strategy	R_1	R_2	Complexity
Olken sample [16]	Index	Index and frequencies	$O\left(M\left\|R_1\right\| / \left\|R_1 \bowtie R_2\right\|\right)$ per tuple in result
Stream sample [2]	—	Index and frequencies	$O\left(1\right)$ per tuple in result
Group sample [2]	—	Frequencies	$O\left(\alpha_1\left\|R_1 \bowtie R_2\right\|\right)$
Frequency-partition sample [2]	—	End-biased frequencies	$O\left(\alpha_2\left\|R_1 \bowtie R_2\right\|\right)$
Index sample [2]	—	Index R_2^{hi} and end-biased frequencies	$O\left(\alpha_3\left\|R_1 \bowtie R_2\right\|\right)$
Ripple join[6]	Index	Index	$O\left(\sqrt{\left\|R_1\right\|/d}\right)$ per tuple in result
Hash ripple join[6]	Index	Index	$O\left(\sqrt{\left\|R_1\right\|/d}\right)$ per tuple in result
Wander join[14]	Index	Index	$O\left(1/2^{K-1}\right)$ per tuple in result for K relations
MaxRand join [11]	Index and frequencies	Index and frequencies	$O\left(\left\|D\right\|\left\|K\right\| + \sum_{i=1}^{K} N\left(1+\log\left(\frac{\left\|R_i\right\|}{N}\right)\right)\right)$

cost incurred by the Frequency-partition Sample strategy is $O\left(\alpha_2\left|R_1 \bowtie R_2\right|\right)$ where

$$\alpha_2 := \frac{\sum_{v\in D^{lo}} m_1\left(v\right) m_2\left(v\right) + r \times \frac{\sum_{v\in D^{hi}} m_1(v)m_2(v)^2}{\sum_{v\in D^{hi}} m_1(v)m_2(v)}}{\sum_{v\in D} m_1\left(v\right) m_2\left(v\right)}$$

2.1.4 Index sample

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[8] to select a random tuple in R_2^{hi} per tuple in S_1. The cost incurred by the Index sample strategy is $O\left(\alpha_3\left|R_1 \bowtie R_2\right|\right)$ where

$$\alpha_3 := \frac{\sum_{v\in D^{lo}} m_1\left(v\right) m_2\left(v\right) + r}{\sum_{v\in D} m_1\left(v\right) m_2\left(v\right)}$$

2.1.5 Count sample

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. 13.

2.2 Correlation-based sampling

Olken sample 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 perform better with respect to sample inflation but suffer from poorer error estimates (relative to uncorrelated sampling). Correlation-based samping aims to mitigate the issues of correlated sampling, while preserving the benefits, by maximizing *join randomness* [11]; 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 frequencies of attributes).

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

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

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

where $m_k\left(a_j\right)$ are frequencies in R_k and $ms_k\left(a_j\right)$ 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 $|D| := 2$ distinct values $\{a_1, a_2\}$ and frequencies

$$m_1(a_1) = m_2(a_1) = 10$$
$$m_1(a_2) = m_2(a_2) = 20$$

If we sample both relations producing S_1, 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

$$C_{ms_1(a_1)}^{10} \times C_{ms_2(a_1)}^{10} \times C_{ms_1(a_2)}^{20} \times C_{ms_2(a_2)}^{20}$$

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

$ms_1\left(a_1\right)$	$ms_2\left(a_1\right)$	$ms_1\left(a_2\right)$	$ms_2\left(a_2\right)$	# 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}

2.2.1 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]:

$$ms_k\left(a_j\right) := \text{round}\left(\frac{M \cdot m_k\left(a_j\right)}{\sum_{j=1}^{|D|}\sum_{k=1}^{K} m_k\left(a_j\right)}\right) \tag{2.1}$$

Using this result we use the MaxRand join algorithm to produce a maximally random join (see alg. 2). The total cost of MaxRand join is the cost of constructing $ms_k\left(a_j\right)$ for $k = 1, \ldots, K$ and $j = 1, \ldots, |D|$ plus the cost of reservoir sampling each R_i; let

$$N := \max_{k,j} ms_k\left(a_j\right)$$

then the total cost is

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

2.3 Ripple join

Ripple join [6] 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

Algorithm 2 MaxRand Join

```
Inputs:
  relations $R_1(A,\dots),\dots,R_K(A,\dots)$
  $M$ sample size
  $ms_k(a_j)$ defined by eqn. 2.1
  $M \coloneqq \lceil f \cdot |R_1 \bowtie \cdots \bowtie R_K| \rceil$
Output: $S \equiv \texttt{SAMPLE}(R_1 \bowtie \cdots \bowtie R_K, f)$
Init:
  $S[1,\dots,K] = []$
Begin:
  for $i \coloneqq 1$ to $K$:
    // stream
    while $R_i$ :
      $S[i] \coloneqq []$
      for $j \coloneqq 1$ to $|D|$:
        // independent reservoirs
        // for each $a_j$
        $S[i] \coloneqq S[i] \cup$
              ReservoirSample$(R_i, ms_k(a_j), a_j)$
  $S \coloneqq S[1] \bowtie \cdots \bowtie S[K]$
```

join algorithm that by virtue of building the join incrementally produces a sample of $R_1 \bowtie R_2$; 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 [7]. For relations $|R_1| < |R_2|$ streaming nested-loops first samples $t_1 \in R_1$ and then R_2 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 is updated. Note that for conventional batch processing, a query optimizer would select R_1 as the outer relation, while for streaming processing the reverse is preferrable so that running estimates can be updated more frequently. 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 because subsequent tuples from R_1 don't yield new "statistical information"; for example in

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

Ripple join address both of these issues by adaptively choosing which is the inner relation and which is the outer relation (see fig. 2.1) and the *aspect ratios* β_1,β_2 of the join i.e. 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 (see [14]). Therefore, in turn, the confidence intervals can be minimized subject to an upper bound c on the the product of the aspect ratios (since the product of the aspect ratios is the number of matches computed and hence I/O):

$$\begin{aligned} \text{minimize } & \sum_{k=1}^{K} \frac{\hat{d}(k)}{\beta_k} \\ \text{such that } & \prod_{k=1}^{K} \beta_k \le c \\ & 1 \le \beta_k \le |R_i| \\ & \beta_k \in \mathbb{N} \end{aligned} \tag{2.2}$$

This is a non-linear integer programming problem which is, in general NP-hard[9]. 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.

2.3.1 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 tuples are hashed on the join column (here is where we need equijoin) then it is possible to identify them efficiently. This implies a reduction in the cost (eq. 2.2) of $n$sampling steps to $\sum_{k=1}^{K} \beta_k \le c$. This then implies the optimal aspect ratios are

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

2.4 Wander join

Wander join[14], 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* 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 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 2.2). Then, given an expression $g(R_1, R_2, R_3)$, we can estimate for $\texttt{SUM}(g(R_1, R_2, R_3))$, for example, by

$$X_\gamma \coloneqq \frac{g(\gamma)}{p(\gamma)}$$

9. By reduction from minimum vertex cover.

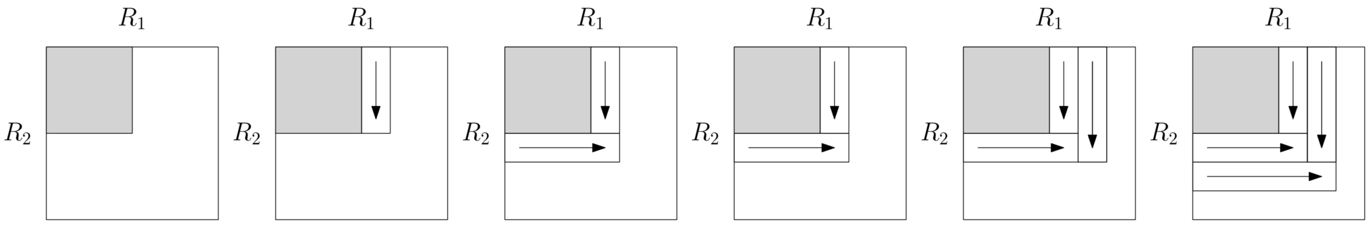


Figure 2.1: Ripple Join for $R_1 \bowtie R_2$ [14].

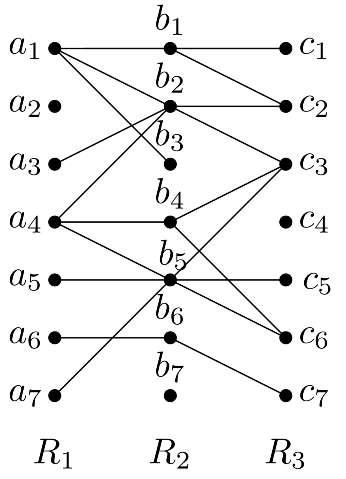
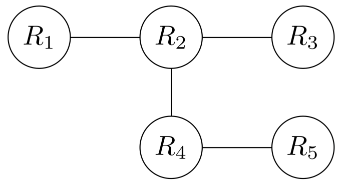
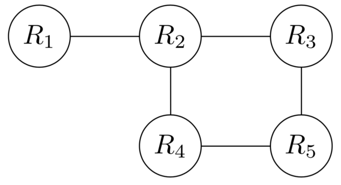

a_1 a_2 a_3 a_4 a_5 a_6 a_7
b_1 b_2 b_3 b_4 b_5 b_6 b_7
c_1 c_2 c_3 c_4 c_5 c_6 c_7
R_1 R_2 R_3

Figure 2.2: Wander join for $R_1(A,B) \bowtie R_2(B,C) \bowtie R_3(C,D)$ [14].



(a) Acyclic join



(b) Cyclic join

Figure 2.3: Join for $R_1(A,B)$, $R_2(B,C,D)$, $R_3(C,E)$, $R_4(D,F)$, $R_5(F,E)$.

where $p(\gamma)$ is the probability of sampling γ

$$p(\gamma) := \frac{1}{|R_1|}\frac{1}{d_2(t_1)}\frac{1}{d_3(t_2)}$$

This estimator is unbiased [8] and therefore the average of many such estimators is also unbiased. That is to say, for a set of sampled paths Γ

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

with variance

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

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

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

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 $Var[X_\gamma], E[T]$ from the walks themselves, the walk plan that minimizes the estimator variance.

2.4.1 Wander join for cyclic and acyclic joins

As described, wander join applies to *chain joins* (joins where relations can easily be ordered such that consecutive relations share one attribute). The algorithm can be extended to include merely *acyclic joins* by incorporating "jumps" into the random walk. For example (see fig. 2.3a), 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 $(R_{\lambda(1)}, \ldots, R_{\lambda(K)})$, define $R_{\eta(i)}$ to be the relation corresponding to the "parent" of $R_{\lambda(i)}$. Then for the path $\gamma := (t_{\lambda(1)}, \ldots, t_{\lambda(K)})$

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

This same adjustment can be extended to work for cyclic joins by first computing a directed spanning tree[10] of the *join query graph* (see fig. 2.3b).

3 SKETCHES

4 APPENDIX

4.1 Stream sampling algorithms

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

10. 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.

Algorithm 3 Rejection Sampling

```
Inputs:
  target distribution $p$
  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 4 Naive Reservoir Sampling

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

4.1.1 *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[11], 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 and M is the least upperbound of $p(x)$. Then the algorithm is straightforward (see alg. 3). The number of samples until "acceptance" follows a geometric distribution with probability $1/M$ and thus has expected number M. Alternative, more efficient, algorithms exist [5].

4.1.2 *Reservoir sampling without replacement*

Reservoir sampling algorithms select 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. 4. 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 [15]:

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) 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 $Y = \max(X_1,\dots,X_k)$ we have
$$P(Y \le y) = \left(\frac{y}{\theta}\right)^n$$
by using $P(X_i \le x) = x/\theta$ and i.i.d. 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[12]. Thus, again by using inverse CD
$$\begin{gathered} U = 1-(1-\theta)^{K+1} \Rightarrow \\ \log(1-U) = (K+1)\log(1-\theta) \Rightarrow \\ K = \left\lfloor \frac{\log(U)}{\log(1-\theta)} \right\rfloor \end{gathered}$$
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. 5. The running time then is $O(k(1+\log(n/k)))$.

4.1.3 *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 `Black-Box U1` fills the output array with copies (whose quantity depends on a random variable drawn from a binomial distribuiton) of successive tuples from streaming relation S. See alg 6. This in effect simulates sampling with replacement. The disadvantage of the alg 6 is that the total size n of the relation is a necessary prerequisite. Algorithm `Black-Box U2` improves on algorithm `Black-Box U1` by eliminating that prerequisite (see alg 7).

4.1.4 *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

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

12. 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 $P(X \le \theta) = \theta$.

Algorithm 5 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 $\theta = 1$ since $A$ consists
  // of $k$ samples from $U(0,1)$.
  // random() draws from $U(0,1)$
  $\theta :=$ exp$\left(\frac{\log(\texttt{random}())}{k}\right)$
  while $S$:
    $k := \left\lfloor \frac{\log(U)}{\log(1-\theta)} \right\rfloor$
    // discard $k$ elements
    while $S$: next($S$)
    // replace random element
    // since we don't need to compare
    $A[\texttt{randomInteger}(1,k)] :=$ next($S$)
    // update max given that stored
    // element was drawn from $U(0,\theta)$
    $\theta := \theta \cdot \exp\left(\frac{\log(U)}{k}\right)$
```

Algorithm 6 Black-Box U1

```
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($S$)
    $X \sim B\left(x, \frac{1}{n-i}\right)$
    // fill $A$ with $X$ copies of $t$
    for $j := 1$ to $X$:
      $A[i+j] := t$
    $x := x - X$
    $i := i + 1$
```

Algorithm 7 Black-Box U2

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

Algorithm 8 Black-Box WR1

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

Algorithm 9 Black-Box WR2

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

with probability proportional $w(t)$ (implying normalization if $\sum_{t \in S} w(t) \neq 1$). We can extend algorithms 6, 7 to respect these semantics; see algorithms 8, 9.

Algorithm 10 Stream Sampling

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

Algorithm 11 Group Sampling

```
Inputs:
  $R_1(A,\dots), R_2(A,\dots)$
  $k \coloneqq \lceil f \cdot |R_1 \bowtie R_2| \rceil$
Output: $S \equiv \texttt{SAMPLE}_{WR}(R_1 \bowtie R_2, f)$
Init:
  $S[1,\dots,k] \coloneqq 0$
  $w(t) \coloneqq m_2(t.A)$ 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 12 Frequency Partition Sampling

```
Inputs:
  $R_1(A,\dots), R_2(A,\dots),\ A \subseteq D$
  $k \coloneqq \lceil f \cdot |R_1 \bowtie R_2| \rceil$
  // 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 $R_1^{hi}$
    $S_1, w_1(t) \coloneqq \texttt{BlackBoxWR2}\left(R_1^{hi}, k, w(t)\right)$
  // integrate/combine stats
  $w(t) \coloneqq w_1(t) \cup w_2(t)$
  // from $w(t)$ you can approximate
  $n_{hi} \coloneqq \left|R_1^{hi} \bowtie R_2^{hi}\right|$
  $R_1^* \coloneqq S_1 \cup R_1^{lo}$
  // stream join
  while $J^* \coloneqq R_1^* \bowtie R_2$:
    $n_{lo} \coloneqq J^*|_{D^{lo}}$ // i.e. $n_{lo} \coloneqq \left|R_1^{lo} \bowtie R_2^{lo}\right|$
    // partition $J^*$
    $J^{lo} \coloneqq \texttt{BlackBoxU2}\left(J^*|_{D^{lo}}, k\right)$
    // $S_1 \equiv \{s_i\}$ and just
    // like in Group-Sample
    $J^{hi} \coloneqq \texttt{BlackBoxU2}\left(s_i \bowtie J^*|_{D^{hi}}, k\right)$
  // # of heads and tails
  // $1 - p = \frac{n_{lo}}{n_{lo}+n_{hi}}$
  $k_{lo}, k_{hi} \coloneqq B\left(k, p = \frac{n_{hi}}{n_{lo}+n_{hi}}\right)$
  $S^{lo} \coloneqq \texttt{BlackBoxUWoR2}\left(J^{lo}, k_{lo}\right)$
  $S^{hi} \coloneqq \texttt{BlackBoxUWoR2}\left(J^{hi}, k_{hi}\right)$
  $S \coloneqq S^{lo} \cup S^{hi}$
```

Algorithm 13 Count Sampling

```
Inputs:
  $k$
  $R_2(A,\dots)$
  $S_1 \subseteq R_1^{hi}$
Output: $S \equiv \left(S_1 \bowtie R_2^{hi}\right)$
Init:
  $H$ // hashtable 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 $S_2 \coloneqq \texttt{BlackBoxWR2}(R_2, H)$:
    $t_1 \coloneqq \texttt{next}(S_2)$
    $t_2 \coloneqq \texttt{BlackBoxWoR}(t_1 \bowtie S_1, 1)$
    $S[i] \coloneqq (t_1, t_2)$
```

4.2 Relation sampling algorithms

5

References

[1] Surajit Chaudhuri, Bolin Ding, and Srikanth Kandula. Approximate query processing: No silver bullet. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 511–519, New York, NY, USA, 2017. Association for Computing Machinery.

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

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

[4] Graham Cormode, Minos Garofalakis, Peter J. Haas, and Chris Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and TrendsÂ® in Databases*, 4(1-3):1–294, 2011.

[5] 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.

[6] 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.

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

[8] 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.

[9] Dawei Huang, Dong Young Yoon, Seth Pettie, and Barzan Mozafari. Joins on samples: A theoretical guide for practitioners. *Proc. VLDB Endow.*, 13(4):547–560, December 2019.

[10] Niranjan Kamat and Arnab Nandi. Perfect and maximum randomness in stratified sampling over joins. *CoRR*, abs/1601.05118, 2016.

[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] Srikanth Kandula, Anil Shanbhag, Aleksandar Vitorovic, Matthaios Olma, Robert Grandl, Surajit Chaudhuri, and Bolin Ding. Quickr: Lazily approximating complex ad-hoc queries in big data clusters. In *Proceedings of the ACM SIGMOD International*

Conference on Management of Data (SIGMOD 2016). ACM - Association for Computing Machinery, June 2016.

[13] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. Learning to optimize join queries with deep reinforcement learning, 2019.

[14] 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.

[15] 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.

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

[17] Do Le Quoc, Istemi Ekin Akkus, Pramod Bhatotia, Spyros Blanas, Ruichuan Chen, Christof Fetzer, and Thorsten Strufe. Approxjoin: Approximate distributed joins. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, pages 426–438, New York, NY, USA, 2018. Association for Computing Machinery.

[18] Todd L. Veldhuizen. Leapfrog triejoin: A simple, worst-case optimal join algorithm, 2014.

[19] 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.