Utrecht University - Centrum voor
Wiskunde en Informatica

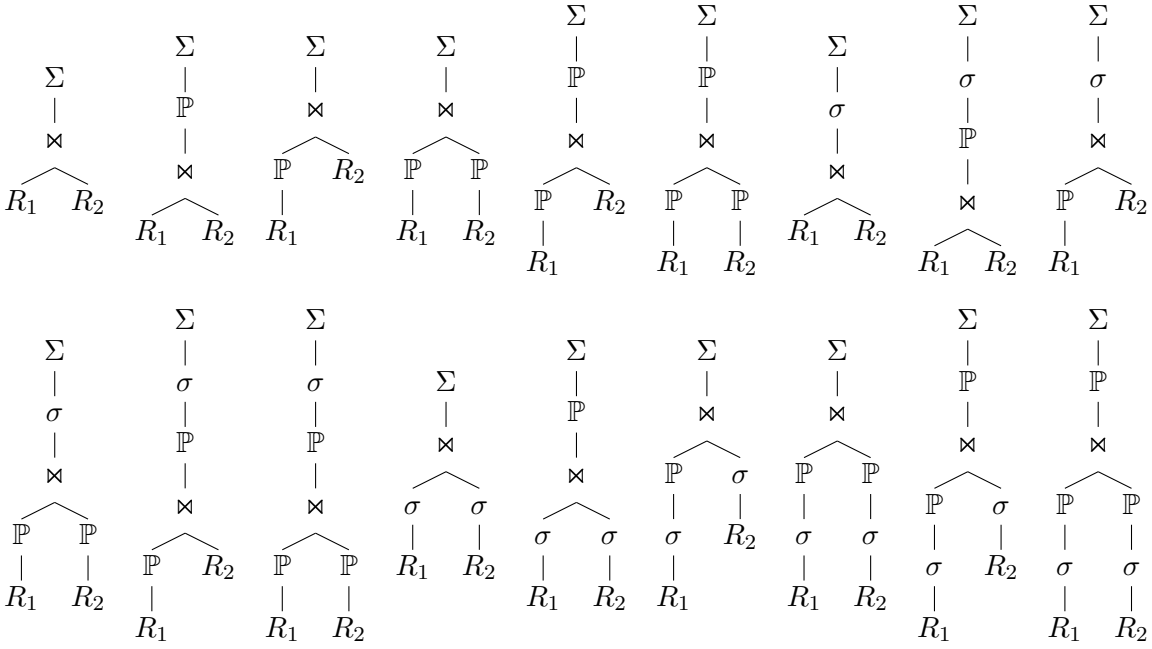
Masters Thesis

Estimating Aggregations over Joins

Author:
Abe Wits

Daily Supervisors:
Hannes Mühleisen (CWI)
Lefteris Sidirourgos (CWI)

UU Supervisors:
Tristan van Leeuwen (UU Math)
Hans Philippi (UU CS)



CWI database Architectures
UU Departement of Mathematics - UU Departement of Computer Science

December 1, 2016

Contents

Chapter 1

Introduction

Computer scientists and mathematicians study closely related topics. However, there is a fundamental conflict between computer science and mathematics. In computer science, the emphasis is typically put on producing results that work well in a computational setting. Meanwhile, in mathematics, the focus is usually on rigid proofs. A computer scientist may think that formal proofs are unnecessary. A mathematician may think that the implementation is an easy or unimportant problem. In many cases, these fields will use a different language, which is tailored to their own goals, adding to misapprehension between the fields.

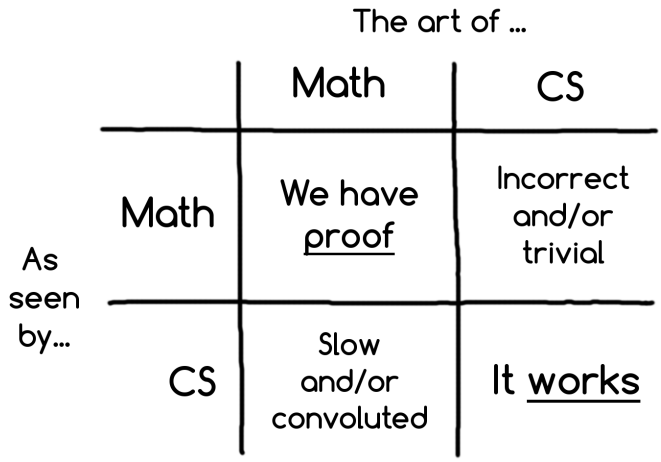


FIGURE 1.1: Comparing the art of Mathematics with the art of Computer Science

We will try to bring computer science and mathematics closer to each other, by keeping an eye on the theoretical and practical aspects, and by describing it in a consistent notation. Specifically, we will combine mathematical theory on sampling and statistics with Approximate Query Processing (AQP). AQP is the field that speeds up the answering of queries on databases by using approximations. Some sampling theory is already used in this field. However, this does not include all state of the art sampling techniques. In Chapter 2 we review the theoretical framework of sampling and describe some popular sampling techniques and their complexity.

The main goal of this thesis is to efficiently approximate the aggregation over a join. More formally, the goal is to approximate the following logical query plan:

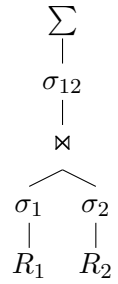


FIGURE 1.2: The target query

In Figure 1.2, R_1 and R_2 represent two relations, and σ, $\bowtie$ and $\sum$ represent operators from a relational algebra. The symbol σ denotes a filter, and is known as `SELECT` in SQL. The natural join is represented by $\bowtie$, which combines rows that have common attributes. The operator $\sum$ computes the sum over some attribute, and is also known as `SUM` in SQL. Estimating the sum by using samples has been studied extensively in AQP, see Section 3.2. Instead of taking a sum directly, we first take a sample and then we estimate the total sum from this sample. Schematically, we can describe this approximation as follows:

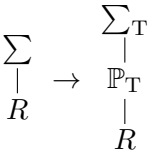


FIGURE 1.3: Approximating the sum

Here $\mathbb{P}_{\mathrm{T}}$ denotes the sampling operator. T denotes its type, which we choose to be uniform for now. $\sum_{\mathrm{T}}$ is an adjusted sum operator $\sum R \approx \frac{|R|}{m} \sum \mathbb{P}_{\mathrm{T}}(R)$. Here m is the sample size. We can apply this technique to approximate the aggregation over a join:

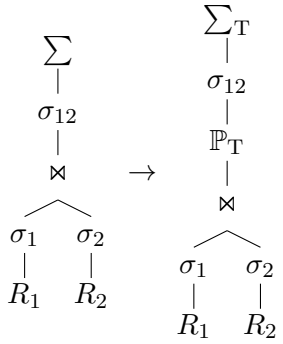


FIGURE 1.4: Approximating the sum in the target query

This allows for some speedup. However, we have not sped up the join operator, which dominates the runtime, so the speedup is relatively small. It is possible to obtain a uniform sample of the join result without calculating the join explicitly. This can be done by using the Stream Sample Join algorithm [7], which we will discuss in Section 3.1. Stream Sample Join "pushes" the sampling operator through the join. This is the resulting updated logical query plan:

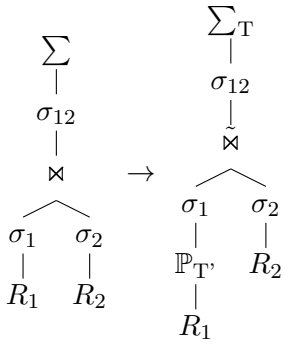


FIGURE 1.5: Approximating the sum and pushing down the sampling operator in the target query

Here $\tilde{\bowtie}$ and $\mathbb{P}_{\mathrm{T}'}$ are an adjusted join and an adjusted sampling operator (see Section 3.1 for details). They make sure that the output is a mathematically correct uniform sample of the join result. This logical query plan combines the state of the art techniques to approximate the query result. However, the state of the art can be improved in two ways.

1. We can choose T stratified or weighted (see Figure 1.6) to improve the runtime and/or quality of the estimated sum $\sum R \approx \sum_{\mathrm{T}}(S \subset R)$. All element have equal probability to be selected by a uniform sampling operator. In a stratified sample, this probability can differ for groups of elements. In a weighted sample all elements have their own probability. By choosing a higher selection probability for "interesting" (groups of) elements, the aggregation can be estimated more accurately with a smaller sample. See Section 3.2 for more information.

2. The adjusted sampling operator, $\mathbb{P}_{\mathrm{T}'}(R)$, takes $\mathcal{O}(|R|)$ time. This can be sped up. This can be done by using indices, by taking a heuristic sample, or by forcing $\mathbb{P}_{\mathrm{T}'}$ to be uniform. See Section 4.1 for a full description.

There is an intricate interplay between these two optimisations (see Section 4.2). A fast sampling operator might produce a sample with weights that do not suit the aggregation. Obtaining a weighted sample that allows efficient aggregation may not be possible, if certain statistics on R_1 have not been pre processed. We discuss the results of our experiments and their implementation in Chapter 5. Finally, in Chapter 6 we draw our conclusions and discuss potential future work.

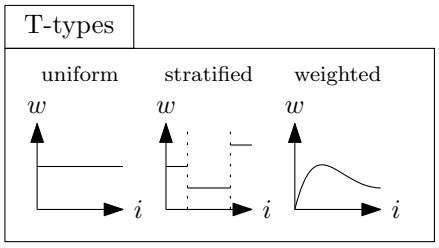


FIGURE 1.6: Overview of the different T-types for the operators $\mathbb{P}_{\mathrm{T}'}$ and $\sum_{\mathrm{T}}$.

Chapter 2

Preliminaries

In this chapter we review some essential techniques that are needed to read this thesis. Some proofs are included to familiarize the reader with notation and the type of reasoning behind sampling algorithms. A good understanding of these proofs will help the reader to comprehend why certain sampling techniques have a certain complexity, and what it means for a sampling algorithm to be correct in a formal sense. A motivation for including this section is that some work on sampling in AQP shows a lack of interest for the runtime of sampling. For example, one paper [7] describes slow algorithms for with-replacement uniform sampling (Black-Box U1 and Black-Box U2), and only refers to a much faster algorithm [27] without mentioning the complexity. Another paper [16] seems to imply that Weighted Sampling can be done with a runtime linear in the size of a sample, which is incorrect (see Section 2.1.2).

2.1 Sampling

Let us start at the beginning: What is a sample?

> *"A sample is a subset of a population that is obtained through some process, possibly random selection or selection based on a certain set of criteria, for the purposes of investigating the properties of the underlying parent population."*
>
> –Eric Weisstein

Taking the top cookie from a jar to see if you would buy the whole jar fits this definition. However, we will not discuss how to select cookies in this thesis. Instead we focus on certain types of samples only. We will always sample from some list of numeric values (floating point, real or integer). The selection criterion will always be precisely defined and based on randomness. And we investigate aggregations (the sum, the average, the minimum, etc) over the underlying parent population. The desired sample size is chosen before sampling, and a sample of exactly the desired size is always taken.

We distinguish two kinds of sampling; sampling with replacement and sampling without replacement. When sampling with replacement, one element in the parent population could be selected multiple times. When sampling without replacement, an element in the parent population can be put into the sample only once.

2.1.1 Uniform sampling

A uniform sample takes elements from the parent population uniform randomly; every element has the same selection probability. Given some relation R with n elements, we want to construct a size r uniform sample $S = \{S_1, S_2, \cdots, S_r\} \subset R = \{R_1, R_2, \cdots, R_n\}$ without replacement. By definition, S is a uniform sample (without replacement) of R if and only if elements in S are drawn independently and $\mathbb{P}(R_j \in S) = \frac{r}{n}$ for all j. Here $\mathbb{P}(*)$ denotes a probability. If a sampling procedure yields a sample with slightly different probabilities, the procedure is *biased*. If elements of S are not obtained independently, this induces correlation. To illustrate why correlation is bad; suppose we draw R_i as the first element of S uniform randomly, and then just take $R_{i+1}, R_{i+2}, \cdots$ sequentially to obtain the rest of S. This sample *does* have the property that $\mathbb{P}(R_j \in S) = \frac{r}{n}$ for all j. However, it usually is not a "good" sample! For example, if the elements of R are sorted by value, the sequential sample contains a small range of similar values, which is not representative for R.

To illustrate how a uniform sample might be taken, we will now describe reservoir sampling [27, 12], a well known algorithm for taking uniform samples without replacement. The reservoir sampling algorithm for sampling with replacement is very similar.

input: $R = \{R_1, R_2, \cdots, R_n\}$
output: $S = \{S_1, S_2, \cdots, S_r\}$, a uniform sample of R w/o repl.
Initialize S; $S_1 \leftarrow R_1, S_2 \leftarrow R_2, \cdots, S_r \leftarrow R_r$
for $i \in \{r+1, r+2, \cdots, n\}$ **do**
  Do with probability $\frac{r}{i}$
    Let j uniform random in $\{1, 2, \cdots, r\}$
    $S_j \leftarrow R_i$.
  End
end

Algorithm 1: Uniform reservoir sampling without jumps

We prove that this approach yields a uniform sample (without replacement) using induction.

- Induction basis. When $n = r$, the algorithm will simply yield the sample $S = R$. This is a uniform sample without replacement (in fact, it is the *only* possible sample).
- Induction hypothesis. At iteration $i = n-1$ we have a uniform sample of $\{R_1, R_2, \cdots, R_{n-1}\}$
- Induction step. From the induction hypothesis we know $\mathbb{P}(R_i \in S) = \frac{r}{n-1}$ at iteration $n-1$. In iteration n we replace some S_j by R_n with probability $\frac{r}{i} = \frac{r}{n}$, thus $\mathbb{P}(R_n \in S) = \frac{r}{n}$. For any $\ell < n$, the probability that $R_\ell \in S$ was $\frac{r}{n-1}$ before this replacement, and becomes:

$$\begin{aligned}\frac{r}{n-1}\mathbb{P}(S_i = R_\ell \text{ not replaced by } R_n) &= \frac{r}{n-1}\left(\mathbb{P}(R_n \notin S) + \mathbb{P}(R_n \in S)\frac{r-1}{r}\right)\\ &= \frac{r}{n-1}\left(\left(1-\frac{r}{n}\right) + \frac{r}{n}\frac{r-1}{r}\right)\\ &= \frac{r}{n-1}\left(\frac{n-r}{n} + \frac{r-1}{n}\right)\\ &= \frac{r}{n-1}\left(\frac{n-1}{n}\right) = \frac{r}{n}\end{aligned}$$

We conclude that S is a uniform sample without replacement in R $\square$

This algorithm yields a uniform sample without replacement in $\mathcal{O}(n)$ time using $\mathcal{O}(r)$ memory. However, this requires us to compute a random variate for every element, which is a time intensive operation in practice. It can be sped up by using exponential jumps; instead of doing a random experiment for every element in R to check if it should replace an element in S, we do a random experiment to immediately jump to the next element in R that replaces an element in S. The difficulty lies with choosing the distribution of the jumps in the correct way.

input: $R = \{R_1, R_2, \cdots, R_n\}$
output: $S = \{S_1, S_2, \cdots, S_r\}$, a uniform sample of R w/o repl.
Initialize S; $S_1 \leftarrow R_1, S_2 \leftarrow R_2, \cdots, S_r \leftarrow R_r$
Set $i \leftarrow r + \text{jump}(r)$
while $i < n$ **do**
  Let j uniform random in $\{1, 2, \cdots, r\}$
  $S_j \leftarrow R_i$.
  $i \leftarrow i + \text{jump}(i)$
end

Algorithm 2: Uniform reservoir sampling with Jumps

Since we must jump forward at least one element:

$$\mathbb{P}(\text{jump}(i) = 0) = 0$$

From uniform reservoir sampling without jumps we know that:

$$\mathbb{P}(\text{jump}(i) = 1) = \frac{r}{i+1}$$

From the above statements and probability theory we can deduce:

$$\begin{aligned}\mathbb{P}(\text{jump}(i) = k) &= P\left(\text{jump}(i) \geq k\right)\frac{r}{i+k}\\ &= \left(\prod_{\ell=1}^{k-1}\left(1 - \mathbb{P}(\text{jump}(i) = \ell)\right)\right)\frac{r}{i+k}\end{aligned}$$

This jump distribution can be precomputed (partially), or it can be (closely) approximated using a geometric distribution [27]. By computing a random variate of the jump distribution in $\mathcal{O}(1)$ time, the time complexity of uniform reservoir sampling is linear in the number of replacements in S; the expected number of replacements is $\sum_{i=r+1}^{n} \frac{r}{i} = \mathcal{O}(r \log \frac{n}{r})$.

Uniform samples with replacement can be taken in $\mathcal{O}(r)$ time by repeatedly (r times) generating a random index i between 1 and n, and adding R_i to S. While the big-$\mathcal{O}$ complexity is a useful tool to asses runtime efficiency, hidden constant factors can be crucial in practice. In this case, the realities of the actual computing hardware play an important role. Repeatedly generating a random index in R will lead to random access to the back-end storage device. If the storage device is main memory, the difference in access time is not gigantic, but for a classic hard drive random reads take much more time per read than sequential reads [11]. Reservoir sampling keeps the sampling process sequential, which has several benefits:

- It may save disk reads if sampled elements are adjacent
- It can be applied even if R streams by only once
- n does not have to be known apriori

2.1.2 Weighted sampling

A weighted sample is a sample where different elements can have different selection probability. As before, we will denote the sample by S and the parent population by R. The weights of $R = (R_1, R_2, \cdots, R_n)$ are denoted by the $w = (w_1, w_2, \cdots, w_n)$. From these weights selection probabilities can be computed:

sampling type	**selection probability**
with replacement	$\forall_i \forall_j \mathbb{P}(R_i = S_j) = \frac{w_i}{\sum_{i=1}^{n} w_i}$
without replacement	$\forall_i \mathbb{P}(R_i \in S) = \frac{\lvert S \rvert}{\lvert R \rvert}$

While sampling with replacement, the selection probabilities remain unchanged. However, while sampling without replacement, the weight of the sampled elements is removed. If, for example, element k has been selected, the remaining probabilities are:

$$\mathbb{P}(\text{element } R_i \text{ is drawn}) = p_i = \mathbb{1}(i \neq k) \frac{w_i}{-w_k + \sum_{j=1}^{n} w_j}$$

Here $\mathbb{1}(*)$ is the indicator function, which takes the value 1 if its argument is true, and which takes the value 0 if its argument does not hold.

Reservoir sampling can be adopted to take weighted samples [12]. To reduce the number of random variates that has to be drawn, exponential jumps can be used in a way very similar to the uniform sampling case. Instead of jumping over an amount of elements, this jumps over a certain weight. Hence, to determine the number of elements to skip, the sum over the weights still has to be determined. As a result, weighted reservoir sampling

with exponential jumps remains a $\mathcal{O}(n)$ time, $\mathcal{O}(r)$ memory algorithm. This linear time complexity limits the usability of weighted reservoir sampling on very big data. On such data taking a uniform sample in $\mathcal{O}(r)$ time is still feasible.

The $\mathcal{O}(n)$ time complexity of reservoir sampling is not bad; without special indices, all weighted sampling algorithms need $\Omega(n)$ time ($\Omega(n)$ means asymptotically worse or equal to $\mathcal{O}(n)$ time). This can be made intuitive by looking at a corner case; suppose $p_j = 0.9$ and $p_{i\neq j} = \frac{0.1}{n-1}$. p is normalised and, on average, a weighted sample with replacement should contain R_j with probability at least 90%. Thus any good weighted sampling algorithm should know where R_j is, requiring at least one linear pass over all the weights, taking $\Omega(n)$ time. In this sense, reservoir sampling has an optimal time complexity. However, it is possible to device a faster algorithm if we approximate the weighted samples heuristically; we introduce a novel (and the first) Heuristic Weighted Sampling algorithm in Section 4.1.3.

With the help of auxiliary data structures, we can obtain samples (with replacement) with a better time complexity. Suppose we have the normalized Cumulative Distribution Function (CDF) of p, which can be defined as $C_i = \frac{\sum_{j=1}^{i} w_j}{\sum_{j=1}^{n} w_j}$. To sample one element a random double x is taken in $[0, 1[$. Let i the biggest index such that $w_i \leq x$; observe that $\mathbb{P}(i = j) = C_j - C_{j-1} = p_j$, so this approach does indeed yield an element with the correct selection probability. The index i can be found using a binary search in the CDF in $\mathcal{O}(\log(n))$ time, hence the total time complexity of this algorithm is $\mathcal{O}(r \log(n))$. This algorithm can be adopted to take samples without replacement by simply redrawing an element if it was already present in S (a data structure for detecting duplicates is required). As long as the total weight of S remains small relative to the total weight of R, this approach is efficient.

We cannot expect that a normalized CDF is always available; it takes linear storage and needs to be precomputed in linear time. The storage overhead is not always acceptable. Precomputing CDFs for intermediate query results (for example after a filter) is not feasible without a priori knowledge; the number of possible intermediate query results is practically infinite.

Chapter 3

State of the Art

In this chapter we will discuss the current state of the art techniques to sample through joins (Section 3.1) and to estimate aggregates from samples (Section 3.2). By combining these techniques, we can efficiently approximate aggregations over joins. We will discuss how to combine these techniques, and improvements of Sampling Through Join techniques in Chapter 4.

3.1 Sampling Through Joins

In this section we will discuss how to obtain a sample over a join efficiently, using some of the current state of the art techniques. In Section 4.1 we extend and improve some of these techniques.

Let us formalize the problem setting. See Figure 3.1 for an overview of the notation used in this section. Suppose we have two relations, $R_1(A, B)$ and $R_2(A, C)$. We want to compute a (uniform) sample (with replacement) over $J = R_1 \bowtie R_2$ efficiently. We want this sample to have size $\alpha|J| \in \mathbb{N}$ for some $\alpha \in [0, 1]$. R_1 and R_2 join over attribute A, which attains values $a_1, a_2, a_3, \cdots$. The number of tuples $t \in R_j$ with $t.A = a_i$ is denoted with $m_j(a_i)$ for $j \in \{1, 2\}$.

The most basic approach is to compute the full join and sample from that. This can be sped up by "pushing down" the sampling operator. With "pushing the sampling operator down through a join" (or "sample through a join" for short), we refer to moving the sampling operator down in the logical query plan so the sampling is performed before the join is executed. This reduces the size of the input of the join, which can result in a big speedup. For the same reason, filters are often pushed down joins.

It is not efficient to simply push down uniform sampling through a Join operator. An example (based on an example by Chaudhuri et al [7]); suppose $R_1.A = \{1, 1, 1, 1, 1, 2, 1\}$ and $R_2.A = \{2, 2, 2, 2, 2, 2, 2, 2, 2, 2\}$. Note that all elements of $R_1 \bowtie_A R_2$ are the result of joining just one tuple of R_1 with all the tuples of R_2. Hence if we were to instead join $S_1 \subset R_1$ with $S_2 \subset R_2$, the result will likely be empty. In this section we will describe some other ways of pushing sampling through a join.

The techniques described strive to achieve several goals.

- Create truly uniform samples (no bias, no correlation)
- Be computationally efficient (in terms of time and memory)
- Depend on as few indices/statistics as possible

Constructing an index is more expensive than the direct execution of an exact weighted sampling algorithm. To avoid this problem, it is possible

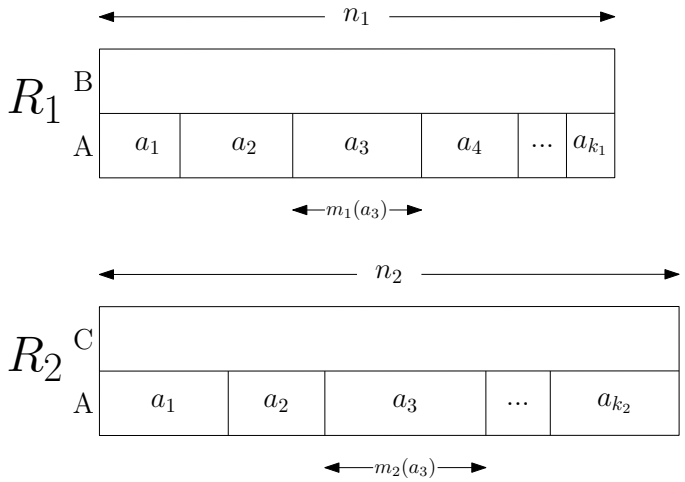


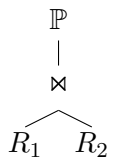
FIGURE 3.1: The notation used in the section on sampling through joins, which is based on the notation used by Chaudhuri et al [7]. In this picture, R_1 and R_2 are stratified by values of A to illustrate the meaning of m_1 and m_2. Whenever R_1 and/or R_2 is assumed to be stratified, this is mentioned explicitly.

to pre-process the data to construct and store indices. We do not know which attributes are relevant in advance, so it may be necessary to store many indices. However, we do want to depend on as few indices/statistics as possible, since there are some common cases where pre-processing is not possible:

- Intermediate query steps; it is hard to predict what the intermediate query results will be, let alone index them.
- Big datasets and/or small memory; we cannot assume there is enough space available to store indices/statistics.
- Updates; if the data is changed, updating the indices can be time consuming
- Streaming setting; in a streaming setting the data must be processed in one pass.

3.1.1 Baseline sample join

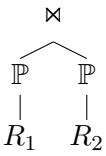
The simplest way of sampling over a join is just performing the join and then sampling from the result.



This technique is statistically sound but can be rather inefficient; it will take $\mathcal{O}(|R_1 \bowtie R_2|)$ time, which could be as much as $\Theta(n_1 * n_2)$. If we were interested in a uniform sample containing only 1% of the tuples in J, we will "throw away" 99% of our hard work in the sampling step. Can we do better?

3.1.2 "Fast" baseline sample join

We could consider pushing down the sampling operator through the join to both sides:



Here $\mathbb{P}$ is the sampling operator. Would this improve the runtime, since we will have to join two much smaller sets? The answer is no, it will not always improve the runtime; if only very few combinations $t_1 \in R_1$ and $t_2 \in R_2$ join, we could end up requiring very large samples in R_1 and R_2 to get any output, which is especially inefficient if sampling with replacement is used. However, if the join key is reasonably dense, we can expect that most tuples in the sample will join, and this approach would be efficient. But there are bigger problems with this approach; bias and correlation.

First, we will discuss the bias. It is induced by the (implicit) projection performed by the join operator. One output tuple $t = (t_1 \in R_1) \bowtie (t_2 \in R_2)$ could originate from different pairs of t_1 and t_2. This can be the case even if R_1 and R_2 contain no duplicates, because the fields that differentiate tuples can be projected out. The probability that a certain $t \in (R_1 \bowtie R_2)$ occur will differ depending on the number of ways it could be created, causing bias. This problem would affect all the sample join techniques we study in this thesis. The solution is simple; we can assume that duplicates are not removed from the join result. In most cases duplicates are very rare, so the influence of this solution on the results is limited. To us, the benefits of avoiding bias outweigh the possibility of duplicates, and hence we will assume duplicates are not removed in the rest of this thesis.

Now let us discuss the correlation. This turns out to be the real issue of "Fast" baseline sample join; correlation can pose significant practical problems when trying to estimate aggregates from a sample. To illustrate this issue, suppose we take a size 1 sample $S_1 = \{t\} \subset R_1$. By joining it with $S_2 \subset R_2$ we obtain $S_1 \bowtie S_2$, which is a sample in $R_1 \bowtie R_2$. However, all tuples in S_1 and S_2 have the same value for the A and B attribute, $t.A$ and $t.B$. $S_1 \bowtie S_2$ resides entirely in the stratum of R_2 with $R_2.A = t.A$. If the C and A attribute of R_2 are not independent, the estimate of the sum over $(R_1 \bowtie R_2).C$ using $S_1 \bowtie S_2$ will be (severely) biased. The problem persists if larger samples in R_1 and R_2 are used, since a weaker (but still significant) correlation between tuples in $S_1 \bowtie S_2$ will be present. There is no way of solving this issue without drastically altering the algorithm.

3.1.3 Stream Sample Join

In this section, we will first discuss Olken sample join [24], the algorithm that inspired Stream Sample Join [7]. After that we will discuss Stream Sample Join here briefly, a more technical description can be found in Section 4.1.1.

Can we push sampling down a join operation without inducing correlation? Olken has shown that this is indeed possible. His algorithm, the Olken sample join algorithm can be classified as a rejection sampling scheme. We take tuples from R_1 uniform randomly. Suppose t_1 is such a tuple, with $t_1.A = a_i$. Then we accept the tuple t_1 with probability $\frac{m_2(a_i)}{M}$, where

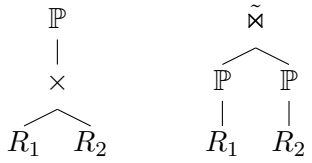


FIGURE 3.2: The left and right LQP have the same result, while the right LQP is much more efficient.

$M = \max_j (m_2(a_j))$, and otherwise reject it. If it is accepted, we select t_2 uniform randomly from $\{t \in R_2 \mid t.A = a_i\}$, the set of tuples in R_2 that could join with t_1, and output $t_1 \bowtie t_2$.

We need to efficiently select random elements from $\{t \in R_2 \mid t.A = a_i\}$. This is an easy task if R_2 is stratified by A values. To obtain $m_2(a_i)$ efficiently, we also need statistics on R_2. Finally we need to be able to take a uniform random sample from R_1. The size of this sample depends on the number of tuples that are rejected.

Is there a way to avoid rejecting part of our sample? Yes! The resulting algorithm is called Strategy Stream-Sample in the original paper [7], we will refer to it as Stream Sample Join. Instead of rejecting tuples sampled from R_1 to fix the distribution of the output tuples, we weigh the sample on R_1. Tuple t_1 should have weight $m_2(t_1.A)$ to attain the correct output distribution (so we need statistics on R_2 to obtain weights). To get such a weighted sample from R_1 we need $\mathcal{O}(n_1)$ time. Then this t_1 is joined with t_2, which is sampled uniform randomly from $\{t \in R_2 \mid t.A = a_i\}$, the set of tuples in R_2 that could join with t_1. We need an index on R_2 to be able to do this efficiently. Then $t_1 \bowtie t_2$ is added to the result.

3.1.4 Mini-Join

The Mini-Join [16] operator (denoted $\tilde{\bowtie}$) allows us to push sampling down a Cartesian product, see Figure 3.2. It can also be used to simplify the description of Stream Sample Join, see Section 4.1.1.

Suppose we have $S_1 \subset R_1$ and $S_2 \subset R_2$, uniform samples. The Mini-Join operator $\tilde{\bowtie}$ takes tuples $s_1 \in S_1$ and $s_2 \in S_2$ uniform randomly, without replacement, outputting $s_1 \bowtie s_2$, until either S_1 or S_2 is empty. The result is a true uniform sample of $R_1 \times R_2$, the Cartesian product of R_1 and R_2.

3.1.5 Additional sample join algorithms

Kamat and Nandi describe some sample join algorithms for situations with specific (un)availability of statistics/indices [16].

First there is Group Sample Join. This is a variation of the Stream Sample Join that does not require availability of an index on R_2. As a result, it is slower than Stream Sample Join. Basically, a weighted sample S_1 from R_1 of size r is constructed with weights $m_2(t_1.A)$ (hence statistics on R_2 are required). This sample is joined with R_2, the result is $S_2 = S_1 \bowtie R_2$ `GROUP BY` S_1. One tuple is selected from every group of S_2, the resulting set of tuples is S, the desired size r sample of J.

Then there is "StratJoin_Both", a stratified sample join algorithm. It assumes both R_1 and R_2 are stratified (as in Figure 3.1). The algorithm; For stratum i (with value a_i) produce uniform samples $S_1^i \subset \{t_1 \in R_1 | t_1.A = a_i\}$

and similarly S_2^i, both of size $f \cdot m_1(a_i) \cdot m_2(a_i)$. Here f is the sampling fraction, a constant. Use the Mini-Join operator to combine S_1^i and S_2^i for each i. The result, $\cup_i S_1^i \tilde{\bowtie} S_2^i$, is a uniform sample in J.

Finally they introduce StratJoin_Overall, which can be seen as an improvement of StratJoin_Both. The idea of StratJoin_Overall; depending on the sampling fraction, we sample or use the complete stratum; if the size of the fraction to be sampled is bigger than 1 we do not sample, in order to avoid sample inflation (sampling fraction bigger than one). Three cases are considered:

- both strata have sampling fraction less than 1
  $\implies$ use StratJoin_Both
- only one stratum has sampling fraction less than 1
  $\implies$ use Stream Sample Join (Kamat and Nandi refer to Stream Sample Join as StratJoin_NN)
- neither stratum has sampling fraction less than 1
  $\implies$ use baseline sample join

3.2 Sampling For Aggregation

We will discuss how to estimate an aggregation given that we have some way of obtaining a weighted sample, for example from a join result as explained in Section 3.1 or 4.1. First we will motivate why it is a good idea to use samples to estimate aggregations. Then we will describe the exact problem setting, and finally we will discuss possible approaches and their properties.

First the motivation; why do we want to use sampling to estimate aggregates? Aggregation functions, like `SUM`, `MIN`, `MAX` or `AVG`, can be computed in linear time. However, if there is a lot of data, such a linear scan may be too time intensive. It is possible to estimate the aggregation, by instead looking at part of the data only. It turns out that such approximate aggregations are very efficient; it is possible to reduce the runtime a lot while maintaining reasonably accuracy. In this section we will discuss ways to estimate aggregations from different types of samples. For each type of sample, we discuss a way to estimate the aggregate and some ways of estimating the error of the estimate.

Now for the problem setting. Suppose we have a weighted without replacement sample S in R with weights w. We want to obtain an estimate of the aggregation as quick as possible, and with help of as few statistics as possible. In this section, we will only discuss the `SUM` aggregation. If statistics are used, we assume them to be precomputed, to avoid online linear scans. We prefer statistics that can be computed in $\mathcal{O}(n) = \mathcal{O}(|R|)$ time. Depending on the setting, storing $\mathcal{O}(n)$ size statistics may or may not be possible. In the rest of this section, we will show how to estimate aggregation given a uniform, stratified or weighted sample.

For each type of sampling, we will discuss known error bounds. Ultimately, we would like to obtain estimates of query results with a confidence interval. A (σ, ϵ) confidence interval is defined by the property that the absolute error of the estimate is less than ϵ for at least a fraction σ of the random trials. If we have some estimate $\hat{\sum} R$ for the exact sum aggregate

$\sum R$, we can write this property like this:

$$\mathbb{P}\left(\left|\hat{\sum} R - \sum R\right| < \epsilon\right) > \sigma$$

If the resulting query is not just one aggregate, but a set of aggregates (for example after a groupby), it can be preferable to use a distribution precision measure [10] instead of a confidence interval.

3.2.1 Aggregation through uniform sampling

In the special case that S is a uniform sample, the following estimator can be used:

$$\sum R \approx \frac{|R|}{|S|} \sum S$$

This is an unbiased estimate [8]; the expected value $\mathbb{E}(\frac{|R|}{|S|} \sum S)$ equals the correct value $\sum R$. The speed of convergence depends on the distribution of the data. In the worst case, we have an extremely large outlier, and we need to sample almost all of the data to get a reasonable estimate of the sum. If we know the values are in $[\min, \max]$, a (σ, ϵ) confidence interval for the sum can be obtained using a uniform sample [5, 31] of size

$$\frac{|R|^2(\max - \min)^2}{2\epsilon^2} \log\left(\frac{2}{1-\sigma}\right).$$

This result can be obtained using Hoeffding bounds.

It is also possible to estimate the variance of the estimated sum [8].

$$\hat{\text{var}}\left(\frac{|R|}{|S|} \sum S\right) = \frac{|R|(|R| - |S|)}{|S|(|S| - 1)} \sum_i (S_i - \text{avg}(S))^2$$

This is an unbiased estimate of the variance, which has a variance of its own; we will not analyse this "higher order" variance. The variance gives us a way to obtain an approximate confidence interval of our estimate. Assuming that R is normally distributed (a strong assumption!), at a significance level σ we have that approximately:

$$\sum R \in \left[\frac{|R|}{|S|} \sum S \pm t_{(1-\sigma)/2} \sqrt{\hat{\text{var}}\left(\frac{|R|}{|S|} \sum S\right)}\right]$$

Here t_α is the tail probability of the student's t-distribution with $|S| - 1$ degrees of freedom. This approximation can still be used if R is not exactly normally distributed, but it will be less accurate (again, we do not analyse higher order variance). Agarwal et al [1] have compared properties of the above ways of obtaining confidence intervals, and provide more methods to estimate confidence intervals.

3.2.2 Aggregation through stratified sampling

Stratified sampling for aggregation is a well known variance-reduction technique [8, 21, 31], that improves upon estimation of aggregation through uniform sampling. It is an approach that allows for computationally efficient

application in practice. The main idea is that there are "interesting" and "uninteresting" regions in the data. A sample that focusses on the interesting parts of the data allows for higher precision estimates of the aggregation.

To this effect, the data is partitioned based on the values of the aggregation attribute. In the world of sampling, the partitioning is called the stratification, and the partitions are called strata. Creating this stratification is usually considered to be a preprocessing step. A uniform sample is taken within each stratum to estimate the aggregation value for each stratum. The sampling fractions of the uniform sampling steps are based on some measure for "interestingness" each stratum. For example, the variance within the strata and the average impact of each stratum's value range on the aggregation. As a result we can achieve a better approximation with similar sample-size, or achieve a similar-quality approximation with a lower sample-size. The total aggregation is estimated from these stratum-estimates.

Consider, for example, estimating the sum of a certain attribute that is distributed according to the exponential distribution. This distribution has large outliers, so a small number of values will have a big impact on the results; if we select too many outliers, our estimate will be too big, if we select too few, our estimate will be too small. By sampling a larger fraction in the strata containing the outliers, the precision of our estimate can be greatly improved. By sampling less from clusters of similar values, the size of the sample is greatly reduced, improving performance.

As an example, consider the stratified estimator of the sum of a relation:

$$\sum R \approx \sum_{i=1}^{\#\text{strata}} |R_i| \text{avg}(S_i)$$

Here the R_i are a partition of R (i.e. $R = \cup_{i=1}^{\#\text{strata}} R_i$ and $\forall_{i \neq j} R_i \cap R_j = \emptyset$). $S_i \subset R_i$ is a uniform sample. Note that this estimator is unbiased; its expected value is exactly $\sum R$. The estimated variance of the stratified estimator is [8]:

$$\hat{\text{var}}\left(\sum_{i=1}^{\#\text{strata}} |R_i| \text{avg}(S_i)\right) = \sum_{i=1}^{\#\text{strata}} |R_i|^2 \hat{\text{var}}(\text{avg}(S_i))$$

Here the variance contribution of each stratum stems from the variance of uniform sampling. Obviously, one can apply the expressions derived in Section 3.2.1:

$$\hat{\text{var}}\left(\text{avg}(S_i)\right) = \hat{\text{var}}\left(\frac{1}{|S_i|}\sum S_i\right) = \frac{1}{|R_i|^2}\hat{\text{var}}\left(\frac{|R_i|}{|S_i|}\sum S_i\right)$$

To get the most out of the stratified estimator, we have to choose the desired sampling size within the strata ($|S_i|$) in a "good" way. This problem has been studied extensively, for example by Chaudhuri et al [6] and Yan et al [31]. However, existing optimization techniques spend $\Omega(|R|)$ time, which is not acceptable if we are to choose our weight function online. Hence we will focus on finding one or more stratifications offline that are efficiently applicable for a variety of queries.

3.2.3 Aggregation through weighted sampling

Suppose we do not stratify the data, but rather directly estimate the aggregation result using the weighted sample S. In this section s_i denote elements of S. The stratified approach (Section 3.2.2) is a special case with a piecewise constant weight distribution. The additional degrees of freedom by no longer using strata should allow us to do better, but they also complicate the problem. For sampling without replacement the Horvitz-Thompson estimator can be used:

$$\sum R \approx \sum_{i=1}^{|S|} \frac{s_i}{\pi_i}$$

Here $\pi_i = \mathbb{P}(R_{s_i} \in S)$ is the probability that an element in R ends up in the sample. If we use sampling with replacement, we could use the Hansen-Hurwitz estimator instead. It is a special case of the Horvitz-Thompson estimator.

$$\sum R \approx \frac{1}{|S|} \sum_{i=1}^{|S|} \frac{s_i}{p_i}$$

Here p_i denote the normalized weights $p_i = \frac{w_i}{\sum_i w_i}$. These estimators are very similar, especially if the sampling fraction is low enough to make duplicates in the sample with replacement unlikely. Sampling without replacement yields slightly higher precision estimates, but is more difficult to analyse. In fact, in Section 4.1.3 we could only find a good way of estimating the error of Heuristic Weighted Sampling by assuming sampling with replacement. Since we want samples to be small for fast performance, collisions are unlikely (unless the distribution has some very high weight elements). Hence we focus on sampling with replacement here.

The Hansen-Hurwitz estimator works with any weight distribution. Choosing a weight distribution that allows for fast convergence is the name of the game. If possible, one should choose the weights as close to the values as possible; for the special case $w_i = cs_i$, where c an arbitrary constant, the variance of this estimator drops to zero. However, obtaining the normalization factor $\sum_i w_i$ is as complex as obtaining an aggregation (it *is* an aggregation)! Hence we need to pre-compute it. Pre-computing only makes sense if we can use the same weight distribution for many different aggregations; otherwise we might as well pre-compute the aggregation offline.

From a theoretical point of view, combining a uniform sample with a linearly weighted sample is an (almost) optimal strategy, requiring a size $\mathcal{O}(n^{1/3})$ sample up to polylogarithmic factor [22]. "Up to polylogarithmic" factor means that we ignore $\log(n)^k$ factors, so this is $\mathcal{O}(\sqrt[3]{n}\log(n)^k)$ for some constant k. With linearly weighted sampling, we refer to a sample with replacement and with weights set to the values of the aggregation column. As opposed to the Hansen-Hurwitz estimator, this method by Motwani et al does not require $\sum_i w_i$. The result by Motwani et al is a truly theoretical one, in the sense that it is not directly suitable for practical implementation. However, the potential benefits of the improved asymptotic complexity are a good motivation for future study of their sampling algorithm.

Motwani et al also describe a method to obtain an (ϵ, σ) confidence interval using only linearly weighted samples. In this case, a sample with

$$O\left(\sqrt{n}\epsilon^{-\frac{7}{2}} \log n \left(\log \frac{1}{\delta} + \log \frac{1}{\epsilon} + \log \log n\right)\right)$$

tuples is needed. Even though these results for linearly weighted sampling are specific to sum aggregations, we expect that techniques based on importance sampling could be used in a more general setting. Importance sampling is a sampling technique that allows approximate integration of an unknown continuous function efficiently and with a rigid error bound. It puts emphasis on parts of the integration domain that influence the outcome the most, to faster obtain a good estimate. Perhaps, this technique can be adjusted to estimate aggregations over discrete data. The Non-Parametric [28] variant [32, 23] is probably better suited for such an adjustment, since it already assumes the distribution is represented by a discrete set of values.

Chapter 4

Approximate Aggregation over Joins

Now that we know the state of the art techniques for pushing sampling down joins (Section 3.1), and estimating aggregation from samples (Section 3.2), we can go beyond the state of the art. We will combine the techniques described in Chapter 3 and extend them to efficiently approximate aggregation over joins.

4.1 Sampling Through Joins

In Section 3.1 we discussed current state of the art techniques for taking a sample over a join efficiently. All these techniques strive to produce uniform random samples. However, if we drop this constraint, we can benefit in two major ways. First of all we can "steer" the distribution of a sample to suit a certain aggregation. How this improves the aggregation quality is discussed in Section 4.2. Secondly, it turns out that a sample can be taken in a much more time efficient way for certain weight distributions. A theoretical comparison of runtimes of different approaches is presented in the example setting in Section 4.1.5.

4.1.1 (Heuristic) Stream Sample Join

A high level overview of Stream Sample Join [7] can be found in Section 3.1.3. In this section we will describe it in more detail, and prove its correctness and complexity. Stream Sample Join has been summarized by Kamat and Nandi [16] as follows:

1. Obtain a with-replacement sample S_1 of R_1 where the sampling weight $w(t_1)$ for a tuple t_1 is set to $m_2(t_1.A)$.

2. For each tuple from S_1 in a streaming fashion do:

   (a) sample a random tuple t_2 from amongst all tuples $t \in R_2$ that satisfy $t.A = t_1.A$

   (b) Output $t_1 \bowtie t_2$

In step 1, it is assumed that $m_2(t_1.A)$ can be obtained efficiently, so some statistics on R_2 should be available. Step 2 could written as $S_1 \tilde{\bowtie} R_2$, see Section 3.1.4. We will now prove that the result is a uniform sample of $|R_1 \bowtie R_2|$ (if duplicate elements are not removed from the join).

Proof: Let $j = (j_1 \bowtie j_2) \in (R_1 \bowtie R_2)$ an arbitrary tuple. The probability that j is selected by the Stream Sample Join algorithm is equal to the probability that we first select j_1 (this probability is $\alpha^{-1} m_2(j_1.A)$ with normalisation constant $\alpha = \sum_{j_1 \in R_1} m_2(j_1.A) = |R_1 \bowtie R_2|$) times the probability that we then select j_2 (this probability is $1/m_2(j_1.A)$). Thus the total probability that j is selected is $\alpha^{-1} m_2(j_1.A) * 1/m_2(j_1.A) = \alpha^{-1} = 1/|R_1 \bowtie R_2|$. Since all elements in the sample produced by Stream Sample Join are chosen independently, we also know that there is no correlation. We conclude that Stream Sample Join produces a uniform sample of $R_1 \bowtie R_2$.

□

The complexity of Stream Sample Join is $\mathcal{O}(\max(|R_1|, m)$, where $m = f*|R_1 \bowtie R_2|$ is the output size and f is the desired sampling fraction. Assuming we cannot speedup the weighted sampling using statistics on the weight distribution, taking a weighted sampling using weights $w(t_1) = m_2(t_1.A)$ will take $\mathcal{O}(|R_1|)$ time. However, it is possible to obtain heuristic weighted samples without memory-intensive statistics in $\mathcal{O}(m^2 \frac{w_{\max}}{w_{\min}})$ time (see Section 4.1.3). The resulting algorithm, Heuristic Stream Sample Join, has a total complexity $\mathcal{O}(\max(m^2 \frac{w_{\max}}{w_{\min}}, m)) = \mathcal{O}(m^2 \frac{w_{\max}}{w_{\min}})$. Here $w_{\max} = \max_i w_i$ and $w_{\min} = \min_i w_i$.

4.1.2 (Heuristic) Weighted Stream Sample Join

Stream Sample Join is a great algorithm to compute uniform samples. However, for many applications, we would prefer an algorithm that can take weighted samples. One such application is estimation of aggregates. As discussed in Section 3.2, stratified and weighted samples can yield better estimates of the sum than equally sized uniform samples.

In this section we extend Stream Sample Join to compute (heuristic) weighted samples. This yields two novel algorithms, Weighted Stream Sample Join (WS-join) and weighted Heuristic Stream Sample Join (HWS-join). In Section 4.1.4 we introduce a third novel algorithm, uniform sample Stream Sample Join (US-join). See Figure 4.3 for a visual summary of HWS-join, WS-join and US-join.

Stream Sample Join can be adjusted to produce weighted output; we can adjust the probability that tuples are selected by tinkering with the weights in the weighted sampling step. This may be useful if we want to aggregate over a sample of the join-result; as discussed in Section 3.2 this may reduce the size of the sample needed, hence producing a result faster or with smaller uncertainty.

We can set the weights to $w(t_1) = h(t_1)m_2(t_1)$. By following the correctness proof of Stream Sample Join in Section 4.1.1 we will show that Weighted Stream Sample Join yields a weighted sample of $R_1 \bowtie R_2$ with weights $h(t_1)$.

Proof: Let $j = (j_1 \bowtie j_2) \in (R_1 \bowtie R_2)$ an arbitrary tuple. The probability that j is selected by the Weighted Stream Sample Join algorithm is equal to the probability that we first select j_1 (this probability is $h(t_1)m_2(j_1.A)$ up to normalisation constant) times the probability that we then select j_2 (this probability is $1/m_2(j_1.A)$). Thus the total probability that j is selected is $h(t_1)m_2(j_1.A) * 1/m_2(j_1.A) = h(t_1)$ up to normalisation. Since all

elements in the sample produced by Weighted Stream Sample Join are chosen independently, we also know that there is no correlation. We conclude that Weighted Stream Sample Join produces a correctly weighted sample of $R_1 \bowtie R_2$.

□

The output weights h can be chosen based on attributes in R_1 and the join attribute; in the notation we are using this are the attributes A and B. By using Heuristic Weighted Sampling instead of weighted sampling, we also have an approximate version of Weighted Stream Sample Join.

It is possible to extend the algorithm, so our choice of output weights h may also depend on the attributes unique to R_2 (attribute C in the notation used). However, some additional preprocessing steps are required. $h(t)$ can be split; $h \sim h_1(t_1)h_2(t.C)$. To obtain one output tuple, we first obtain t_1 by sampling from R_1 using weight function h_1, and some arbitrary ("gauge") function g. Then we join this tuple with t_2, one of the matching tuples in R_2. Instead of choosing t_2 uniformly among all $t_2 \in R_2$ with $t_2.A = t_1.A$, we weigh this choice using $h_2(t_2.C)$ (this gives rise to a weighted generalization of the minijoin operation, which we will denote as $\tilde{\bowtie}_{h_2}$). The selection probability of some tuple t will be:

$$\begin{aligned}\mathbb{P}(t) &= \mathbb{P}(t.A = t_1.A \wedge t.B = t_1.B)\mathbb{P}(t.C = t_2.C) \\ &= \frac{h_1(t_1)g(t_1)}{\sum_{\hat{t}_1 \in R_1} h_1(\hat{t}_1)g(\hat{t}_1)} \frac{h_2(t_2.C)}{\sum_{\hat{t}_2 \in \{\tilde{t}_2 \in R_2 | \tilde{t}_2.A = t_1.A\}} h_2(\hat{t}_2.C)}\end{aligned}$$

The normalization of h_2 depends on $t_1.A$. To produce a sample according to h we have to adjust for this by setting $g(t_1) = \gamma \sum_{\hat{t}_2 \in \{\tilde{t}_2 \in R_2 | \tilde{t}_2.A = t_1.A\}} h_2(\hat{t}_2.C)$ for some constant γ. The total selection probability of t becomes:

$$\begin{aligned}\mathbb{P}(t) &= \frac{h_1(t_1)\gamma \sum_{\hat{t}_2 \in \{\tilde{t}_2 \in R_2 | \tilde{t}_2.A = t_1.A\}} h_2(\hat{t}_2.C)}{\sum_{\hat{t}_1 \in R_1} h_1(\hat{t}_1)g(\hat{t}_1)} \frac{h_2(t_2.C)}{\sum_{\hat{t}_2 \in \{\tilde{t}_2 \in R_2 | \tilde{t}_2.A = t_1.A\}} h_2(\hat{t}_2.C)} \\ &= \frac{h_1(t_1)\gamma}{\sum_{\hat{t}_1 \in R_1} h_1(\hat{t}_1)g(\hat{t}_1)} h_2(t_2.C) \\ &= \frac{\gamma}{\mu} h_1(t_1)h_2(t_2.C)\end{aligned}$$

Here $\mu = \sum_{\hat{t}_1 \in R_1} h_1(\hat{t}_1)g(\hat{t}_1)$ is a constant. We conclude that

$$\mathbb{P}(t) \sim h(t).$$

Adjusting the weight using attributes unique to R_2 (i.e. attribute C) will influence the time complexity of weighted (heuristic) Stream Sample Join. The CDF of the weight distribution over R_2, can be computed using $\mathcal{O}(n_2)$ memory and $\mathcal{O}(n_2)$ time. Assuming this is done apriori, we can ignore this runtime. The time complexities are as follows:

	Exact	**Heuristic**
h **without** C	$\mathcal{O}(\max(n_1, m))$	$\mathcal{O}(\max(m^2 \frac{w_{\max}}{w_{\min}}, m))$
h **with** C	$\mathcal{O}(\max(n_1, m \log n_2))$	$\mathcal{O}(\max(m^2 \frac{w_{\max}}{w_{\min}}, m \log n_2))$

Under the mild assumptions that $n_1 > m$ and $m^2 \frac{w_{\max}}{w_{\min}} > m$ this can be simplified:

	Exact	**Heuristic**
h **without** C	$\mathcal{O}(n_1)$	$\mathcal{O}(m^2 \frac{w_{\max}}{w_{\min}})$
h **with** C	$\mathcal{O}(n_1 + m \log n_2)$	$\mathcal{O}(m^2 \frac{w_{\max}}{w_{\min}} + m \log n_2)$

Here $J = R_1 \bowtie R_2$. When h depends on C, the additional factor $\log n_2$ is a result of the weighted sampling in a stratum of R_2 after t_1 is selected, by binary search in the CDF.

input:

- Two relations, $R_1(A, B)$, $R_2(A, C)$
- A weight function $h = h_1 h_2$, where $h_1 : R_1 \to \mathbb{R}_{\geq 0}$, $h_2 : R_2 \to \mathbb{R}_{\geq 0}$
- The desired output size m

output: $S \subset R_1 \bowtie R_2$, a h-weighted sample

Precompute $g(A) = \sum_{\hat{t}_2 \in \{\tilde{t}_2 \in R_2 | \tilde{t}_2.A = A\}} h_2(\hat{t}_2.C)$
Set $w_1(t_1) = h_1(t_1) g(t_1.A)$ (calculation of w_1 can be done lazily)
Obtain $S_1 \subset R_1$, a size m w/ repl. sample with weights w_1
for $t_1 \in S_1$ **do**
  Obtain $t_2 \in S_2$, with weights h_2
  Add $t_1 \bowtie t_2$ to the output S
end

Algorithm 3: Weighted (Heuristic) Stream Sample Join algorithm. If $S_1 \subset R_1$ is obtained using a weighted sampling algorithm, the result is exact. If it is obtained using Heuristic Weighted Sampling, it is a heuristic algorithm.

4.1.3 Faster than linear Heuristic Weighted Sampling

The Stream Sample Join algorithm depends on weighted sampling (with replacement). A weighted sampling algorithm creates a size m sample S of some relation R, given some weight function $w : R \to \mathbb{R}_{\geq 0}$. An element $x \in R$ is selected with probability $p(x) = \frac{w(x)}{\sum_{x \in R} w(x)}$. Notice p is a probability distribution.

Our goal is to provide a faster than linear ($o(n)$) algorithm, given that we only need a sample that approximately follows the weighting function, and given some statistics of w that can be precomputed. We postulate that improvements are possible if the amount of rare/high weight elements is somehow limited. The general idea is that we can first take a uniform sample $U \subset R$ of size k using a known $\mathcal{O}(k)$ uniform sampling algorithm (for example reservoir sampling with exponential jumps [12]), and then take a weighted sample $S \subset U$ using a standard $\mathcal{O}(k)$ weighted sampling algorithm. See Figure 4.1 for an overview of the notation.

We have to choose $k = |U|$ big enough, so that our heuristic weighted

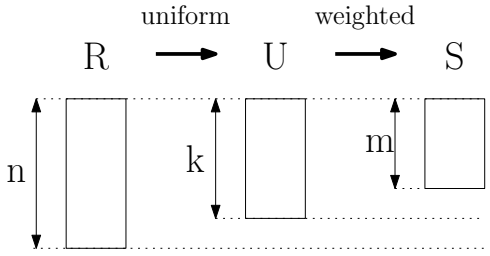


FIGURE 4.1: An overview of the Heuristic Weighted Sampling Algorithm. U is a uniform sample without replacement of R, and S is a weighted sample without replacement of U. S is an heuristic weighted sample of R.

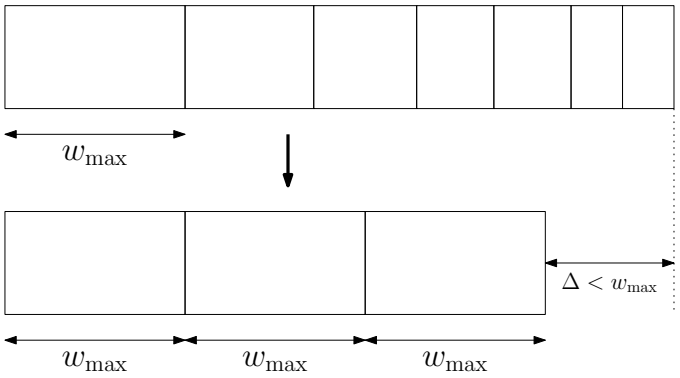


FIGURE 4.2: Reducing the Surname Problem to the Birthday Problem. $w_{\max} \equiv \max_{x_i \in R} w(x_i)$

samples resemble true weighted samples. Ideally the quality of the approximation could be quantified by proving that for some choice of k, and some basic statistics of the weight function, we know that the probability-distribution of samples selected by our approximate algorithm is within a factor $(1 \pm \epsilon)$ of the correct probability-distribution with probability at least σ. However, we were not able to design such a proof, and we leave this for future research.

Hence we choose to focus on some practical and necessary (but not necessarily sufficient) constraints on k. In Section 5.2 we will empirically test if these constraints are sufficient in practice. If U is too small...

1. ...the probability that duplicates (some x_i which is selected multiple times) occur increases

2. ...the similarity between the distribution of weights in U and the distribution of weights in R decreases

First we will discuss how to make duplicates unlikely. Or rather; given some U, how big can $m = |S|$ be, so a weighted sample of size m will not contain duplicates with probability at least 50%. This problem is also known as the Surname Problem, which can also be seen as a weighted version of the Birthday Problem [19]. The known solutions depend on the distribution of the weight function, and are not easily adopted to estimate the size of U in $o(n)$ time. Hence we will instead provide a bound on the proper size of U by reducing the Surname Problem to the Birthday Problem.

We can change any instance of the Surname Problem to an instance of the Birthday Problem with an equal or higher probability of duplicates by

replacing the bins by $\lfloor \frac{\sum_i w_i}{w_{\max}} \rfloor \leq \lfloor \frac{kw_{\min}}{w_{\max}} \rfloor$ bins of size $w_{\max} \equiv \max_{x_i \in R} w(x_i)$, see Figure 4.2. For the birthday problem it is known that samples of size $\sqrt{2q \ln\left(\frac{1}{1-\sigma}\right)}$ out of q bins, will contain duplicates with probability σ (approximately [20]). Hence we can avoid duplicates at significance level σ (with probability $1-\sigma$), by choosing $\sqrt{2\frac{kw_{\min}}{w_{\max}} \ln\left(\frac{1}{1-\sigma}\right)} \approx m$ which we can rewrite as $k \approx \frac{1}{2}m^2 \frac{w_{\max}}{w_{\min}} \ln\left(\frac{1}{1-\sigma}\right)$. Hence choosing $k \geq \frac{1}{2}m^2 \frac{w_{\max}}{w_{\min}} \ln\left(\frac{1}{1-\sigma}\right)$ should avoid duplicates at significance level at least σ.

Now for the second constraint; we want the distribution of weights in R to be similar to the distribution of weights in U. It is difficult to guarantee this without knowing much about the distribution of weights, but we would at least like the weight density of U to be similar to the weight density of R. Given that we know the normalisation of the weight function $\sum_i w_i$, this is easy to check while constructing U:

$$\frac{n}{|U|} \sum_{x \in U} \frac{w(x)}{\sum_i w_i} \approx 1$$

To enforce this constraint in practice, we need to introduce $\delta \in \mathbb{R}_{\geq 0}$, a constant that controls how close the weight density in U should approximate the weight density of R:

$$\frac{n}{|U|} \sum_{x \in U} \frac{w(x)}{\sum_i w_i} \in [1-\delta, 1+\delta]$$

It is not trivial to choose an optimal value of δ. The term $\frac{w_{\max}}{w_{\min}}$ in the sample size already accounts for skew in the data by avoiding duplicates; the purpose of δ and $\frac{w_{\max}}{w_{\min}}$ overlap partially. However, if we are unlucky and select a non-representative sample, δ does add some guarantee that a fair fraction of outliers is present in the sample. In a way, these "unlucky" cases are already covered by the certainty σ in the confidence interval. It is difficult to provide effective rigid bounds on the impact of δ on the size and the quality of the sample. At preliminary experiments, we did observe that large values of δ have no effect, while small values of δ cause large fluctuations in the sample size (and thus the runtime), complicating analysis. For which value δ crosses over from a little to a lot of influence depends on the data. Because of the complex behavior of δ, and because of the already huge amount of parameters, we do not consider δ in Section 5.2, where we evaluate the quality of HWS numerically.

4.1.4 Uniform Stream Sample Join

It is possible to choose the output distribution in a way that allows for a massive speedup of WS-join! Suppose we join a uniform sample in R_1 with R_2 to approximate $R_1 \bowtie R_2$, or equivalently, we set the desired output distribution of the Weighted Stream Sample Join algorithm to $h(t_1) = m_2(t_1)^{-1}$. In this case the weights used in the Weighted Sample step will simplify to uniform weights, $w(t_1) = h(t_1)m_2(t_1) = m_2(t_1)^{-1}m_2(t_1) = 1$. We will refer to this approach as US-join. This case is different than WS-join and HWS-join for two reasons. First of all, it is faster than WS-join and HWS-join

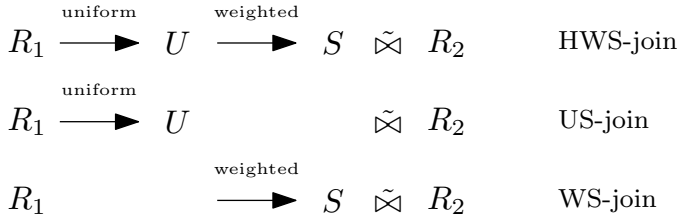


FIGURE 4.3: Three possible sample-join algorithms.

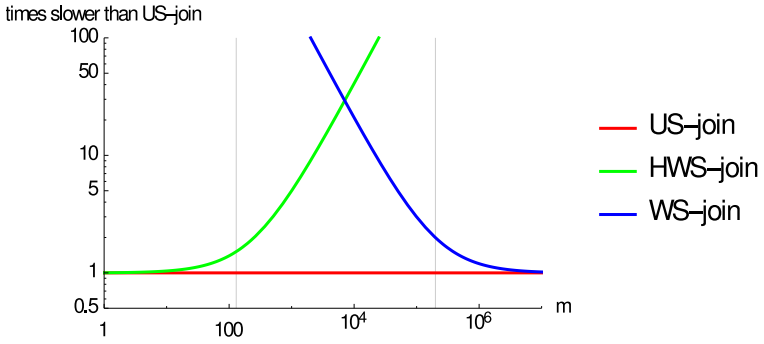


FIGURE 4.4: Log-log plot of relative time needed for HWS-join and WS-join (speed of US-join is set to 1) versus the output size m. R_1 is not indexed, $n_1 = 10^8$, $\frac{w_{\max}}{w_{\min}} = 2$ and $T_2 = 500T_1$. The vertical lines are the crossover points, for $\alpha = 0.7$ and $\beta = 0.5$.

since $w(t_1) = 1$. These weights allow us to use (exact) uniform sampling algorithm which has a time complexity linear in the sample size. Secondly, it is less flexible than WS-join ans HWS-join. $h(t_1) = m_2(t_1)^{-1}$ is fixed, so it is not possible to tweak the output distribution for a certain aggregation.

4.1.5 Theoretical comparison

In this section we compare the theoretical runtime of HWS-join, US-join and WS-join (see Figure 4.3 for an overview of these algorithms). We assume a uniform random tuple in R_2 with a given join-attribute value can be found efficiently, since we need this for Stream Sample Join to work efficiently. We will discuss two settings; the case where R_1 is not indexed and the case where R_1 is indexed using a CDF.

We take differences in access speed to disks into account. This can be relevant because access speeds can differ wildly. For example if one of the relations is in cold storage (magnetic tapes or optic discs) and the other relation is on hard drives. Or, if one of the relation is stored on a hard drive, and the other is in main memory. The ratio of the read speeds is important for the relative performance of US-join, HWS-join and WS-join.

The constants T_1 and T_2 denote the times needed to do a random access read of a tuple from R_1 and R_2 respectively. T_1 and T_2 are expressed in units of T_1^{seq}, the (average) time needed to read one tuple from R_1 during a sequential scan. Typically $T_1^{\text{seq}} \ll T_1$ because of cache effects and (on classic hard drives) seek time and rotational latency.

R_1 is not indexed

We will now compare the complexity of different sample-join algorithms given that no index on R_1 is available.

$$T_{\text{HWS-join}}^{R_2 \text{ indexed}} = \mathcal{O}\left(m^2 \frac{w_{\max}}{w_{\min}} T_1 + mT_2\right)$$

The first term, $\mathcal{O}\left(m^2 \frac{w_{\max}}{w_{\min}} T_1\right)$, is the time needed to do obtain a Heuristic Weighted Sample of size m. This is done by first obtaining U, requiring $\mathcal{O}\left(m^2 \frac{w_{\max}}{w_{\min}}\right)$ random reads in R_1, taking T_1 time per read. This term dominates the time needed to obtain a size m weighted sample S from U. Finally we need $\mathcal{O}(m)$ random reads in R_2 to compute the minijoin, which takes T_2 time per read since R_2 in indexed. While the $\mathcal{O}(mT_2)$ term for the minijoin remains the same for US-join and WS-join, the time needed obtain $S \subset R_1$ does vary. In the case of the US-join, S can be obtained using m random reads in R_1.

$$T_{\text{US-join}}^{R_2 \text{ indexed}} = \mathcal{O}\left(m(T_1 + T_2)\right)$$

In the case of the WS-join, S can be obtained by sequentially reading through all of R_1 by using reservoir sampling.

$$T_{\text{WS-join}}^{R_2 \text{ indexed}} = \mathcal{O}\left(n_1 + mT_2\right)$$

See Figure 4.4 for a plot comparing these runtimes. We also take the difference in quality of the different algorithms into account, without specifying the metric. For this purpose, we use the constant α to denote that a size αm sample using HWS-join is "as good" as a size m sample using US-join. In a similar manner, we can compare WS-join with US-join. WS-join can yield the same "quality" sample as US-join with sample size βm. Independent of the precise measure used, we know $\beta \leq 1$, since uniform sampling is a special case of weighted sampling. Also, one would expect $\beta < \alpha$, since HWS is an approximation of WS. The experimental results in Section 5.2 suggest that β is a modest constant factor smaller than α in most settings.

HWS-join is the most efficient for small values of m, US-join is the most efficient for intermediate values of m and WS-join is the most efficient for big values of m. Given a fixed sample size, WS-join can yield the highest quality aggregation estimates, followed by HWS-join, and the lowest quality (biggest error) estimate is produced by US-join.

We can estimate which kind of sample-join will perform best using the theoretical runtimes. HWS-join will be faster than US-join if:

$$m \in \mathcal{O}\left(\frac{w_{\min}}{w_{\max}} \frac{T_1 + (1-\alpha)T_2}{T_1 \alpha^2}\right) \subset \mathcal{O}\left(\frac{T_2}{T_1}\right)$$

We can conclude that HWS-join is the most efficient only if random access in R_2 is much slower than random access in R_1, or in other words, if $\frac{T_2}{T_1} \gg 1$, the weights do not have a high w-ratio $\frac{w_{\max}}{w_{\min}}$ and m is relatively small. This means that WS-join will be faster than US-join if:

$$m \in \Omega\left(\frac{n_1 T_1^{\text{seq}}}{T_1 + (1-\beta)T_2}\right) \subset \Omega\left(n_1 \frac{T_1^{\text{seq}}}{T_1 + T_2}\right)$$

Here T_1^{seq} is the time it takes to access one tuple in R_1 given that we do this in a sequential manner (thus $T_1^{\text{seq}} < T_1$). We conclude that WS-join is the most efficient if m is "not small" compared to n_1, so scanning through R_1 sequentially to obtain a weighted sample is not much slower than taking a uniform sample (using a lot of random access) instead.

R_1 is indexed

In this section, we will assume that R_1 is indexed with a CDF. As a result we can take a size m weighted sample in $\mathcal{O}(T_1 m \log n_1)$ time. Other indices that speed up weighted sampling exist. For example, a binary tree with cumulative probabilities [30]. To our knowledge, no indices exist that can provide weighted samples in less than $\mathcal{O}(T_1 m \log n_1)$ time.

Only the time complexity of WS-join improves if a CDF on R_1 is available, because it is the only algorithm that computes a weighted sample in R_1 directly. The CDF cannot be used to speedup uniform sampling. Storing (additional) indices to speedup uniform sampling can yield at most a constant speedup factor, since we have to read all m elements in S in any sample-join algorithm.

$$\begin{aligned}
T_{\text{HWS-join}}^{R_1 \text{ and } R_2 \text{ indexed}} &= \mathcal{O}\left(m^2 \frac{w_{\max}}{w_{\min}} T_1 + m T_2\right) \\
T_{\text{US-join}}^{R_1 \text{ and } R_2 \text{ indexed}} &= \mathcal{O}\left(m(T_1 + T_2)\right) \\
T_{\text{WS-join}}^{R_1 \text{ and } R_2 \text{ indexed}} &= \mathcal{O}\left(m(T_1 \log n_1 + T_2)\right)
\end{aligned}$$

If we assume $T_1 = T_2$, we get that US-join is more efficient whenever $n_1 > e^{\frac{2}{\beta}-1}$. For $\beta > 0.2$, US-join will usually be more efficient (it is faster for all relations with $n_1 \gtrsim 10^4$). For $\beta < 0.1$, WS-join will usually be more efficient (it is faster for all relations with $n_1 \lesssim 2 * 10^8$, this bound increases rapidly for smaller β).

Example setting: R_1 in main memory, R_2 on HDD

To get a feeling for typical values of T_1, T_2 and T_1^{seq}, we have done some benchmarks [29] on some consumer hardware. We assume R_1 is in main memory, while R_2 is on a hard disk drive (HDD). The data used was 6.4GB of randomized data. This is big enough to avoid CPU cache effects (our L3 is approximately 8MB) and HDD cache effects (with a cache of roughly 30MB), and in our case small enough to avoid issues with NUMA (non uniform memory access) memory banks. To limit the influence of page caching, it was cleared between runs. The number of tuples read was kept small enough to avoid hitting cached parts of the data too often. Overhead of auxiliary code (time measurements, for loop, etc) was taken into account. This yielded the following results:

Read Type	Var	Latency	Latency ($T_1^{\mathbf{seq}}$)
Sequential RAM	T_1^{seq}	5.2 ns	1.
Random RAM	T_1	57. ns	11.
Sequential HDD	T_2^{seq}	2.3 $* 10^6$ ns	4.5 $* 10^5$
Random HDD	T_2	9.0 $* 10^6$ ns	1.7 $* 10^6$

We can compare the different sample-join algorithms using the above latency measurements. We assume $\alpha = 0.7$, $\beta = 0.6$, $\frac{w_{\max}}{w_{\min}} = 2$ and $n_1 = 10^{11}$. One column in R_1 consisting of 10^{11} 32-bit values takes 0.4TB, which fits in main memory on modern machines. Assuming R_1 is not indexed, the ranges for m are:

Sample-join	Approximate optimal range of m
HWS-join	$m < 4.9 * 10^4$
US-join	$4.9 * 10^4 < m < 1.4 * 10^5$
WS-join	$1.4 * 10^5 < m$

4.2 Combining Sample-Join with Aggregation

In this section we will discuss estimation of aggregates over a join. First we introduce the problem setting, then we discuss some baseline solutions, and finally we describe some novel approaches based on our new sample-join algorithms, US-join, WS-join and HWS-join. See Figure 4.5 for a generic overview of the overall strategy of aggregation over sampling (with a filter).

We will use the notation of Section 3.1 as described in Figure 3.1. It is assumed that computing and storing statistics on R_2 (in memory) is feasible. We assume that we have $\mathcal{O}(n_1)$ time and $\mathcal{O}(1)$ memory available to preprocess R_1 offline, so some statistics on R_1 are available. Our goal is to estimate an aggregation over $R_1 \bowtie R_2$ with precision ϵ and certainty σ. The $\mathcal{O}(1)$ memory limit does not allow the beautiful Wander Join technique by Li et al [17], which relies on indices.

Selections are pushed down as far as possible. σ_{12} filters on predicates that depend on attributes in both R_1 and R_2. σ_1 and σ_2 contain predicates over attributes in R_1 or R_2 only respectively. To achieve target precision, the selectivity of σ_1, σ_2 and σ_{12} have to be taken into consideration when determining the sample size of $\mathbb{P}_{\mathrm{T}}$.

4.2.1 Baseline strategies

Full Join + uniform aggregation

The Baseline solution is full execution of the join followed by an aggregation. Notice that the join is the computationally most intensive step, and this approach will take $\mathcal{O}(|R_1 \bowtie R_2|)$ time. If the join result is small this approach is reasonably efficient (in terms of runtime), however, $R_1 \bowtie R_2$ could be as big as $n_1 n_2$.

Stream Sample Join + uniform aggregation

Stream Sample Join (which is the current state of the art sample-join algorithm) can be combined with a uniform aggregate estimator. Even though

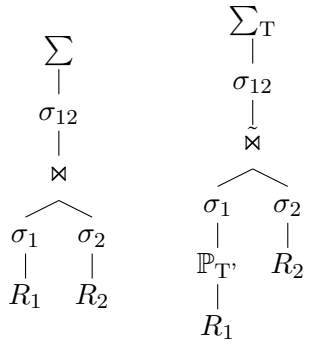


FIGURE 4.5: Logical Query Plan of aggregation over join with selection. The classic approach is on the left. The sampling approach is on the right. T is the type of sampling.

all ingredients of this approach have been known for quite some time, the combination has not been studied yet. The Stream Sample Join has the biggest contribution to the runtime, thus the total complexity is:

$$\mathcal{O}(\max(f|R_1 \ltimes R_2|, |R_1|))$$

4.2.2 Novel strategies

By following one of the three sample-join algorithms (see Figure 4.3) by a stratified aggregation (as described in Section 3.2) we can estimate the sum over a join without executing a full join. There is an intricate interplay between sample-join and the Aggregation step; we have to tune the amount of time spent in each step, and how to choose weights in the intermediate Weighted Heuristic Sample. What choices are optimal depends on the problem size, value distributions, aggregation type and target precision.

WS-join + aggregation

Stream Sample Join can run in $\mathcal{O}(f|R_1 \ltimes R_2| \log n_1)$ if the CDF (Cumulative Distribution Function) of w (the weights on R_1) is available. Here f is the selection fraction; the time complexity of WS-join is linear in its output size. The weighted sampling step does a binary search over this CDF to obtain one random element in $\mathcal{O}(\log n_1)$ time. However, storing a linear-size auxiliary data structure of R_1 is not always feasible. It is possible to use a different sampling algorithm that does not need a CDF. A good choice would be the A-ExpJ algorithm [12], a reservoir sampling algorithm with exponential jumps. The exponential jumps vastly reduce the number of required random variates, but A-ExpJ does requires $\mathcal{O}(n_1)$ time in total.

Stream Sample Join [7] is the special case of WS-join where we take the output-weights uniform. In many cases, the weights of WS-join can be chosen to outperform Stream Sample Join, see Figure 5.5 for such a case.

As briefly discussed in Section 3.2.3, it is possible to use stratified sampling techniques by choosing the weight function piecewise constant. To do this, R_1 has to be grouped into strata, and counts of R_1 within these strata have to be stored. To support aggregation over different attributes, we need more than one stratification. We cannot store the strata explicitly (by reordering R_1 or storing IDs of elements) without inducing $\mathcal{O}(n_1)$

memory overhead. Just storing the counts for all attributes would take $\mathcal{O}(\#\text{attributes}(R_1)\#\text{strata}(R_1))$, which can be chosen to be much less than n_1. Because of the complexity of this approach, we leave full exploration and implementation as future work.

HWS-join + aggregation

"HWS-join + aggregation" is the WS-Join approach where the weighted sampling step is replaced by Heuristic Weighted Sampling. This provides better time complexity, or avoids storing a size $\mathcal{O}(n_1)$ index on R_1. We can use the same choices for output-distribution as discussed in Section 4.2.2, however the Heuristic Weighted Sampling step adds more uncertainty. We can adjust the output size of the algorithm to correct for this additional uncertainty.

US-join + aggregation

The weight distribution $f(t_1) = m_2(t_1)^{-1}$ only depends on the join attribute. Hence we can use the stratified aggregation techniques described in Section 3.2.2. In this case the strata correspond to the distinct values of the join attribute in R_2. Thus the number of strata is at most n_2. We need to know the counts of the strata to be able to perform the aggregation.

Chapter 5

Experiments

5.1 Data generation

To make sure that our algorithms work well in practice, we need to try them on a wide range of inputs. It is not possible to consider all inputs, since there are infinitely many. However, it is possible to generate data with certain properties. We prefer easy to describe properties that have an impact on the performance that can be parametrized. We can analyze the performance of our algorithms under different (extreme) conditions by varying the parameters. This allows us to give some guarantees on the performance of our algorithms. In this section, we will first identify four properties that span a wide range of inputs, and describe measures for these properties. Then we will discuss ways to generate data with certain properties.

Weight ratio

The weight ratio is defined as the maximum weight divided by the minimum weight. This quantity appears in the required minimum sample size of Heuristic Weighted Sampling (HWS), and hence is interesting to control. A weight ratio equal to 1 corresponds to constant data. The weight ratio of a distribution can be set to any value by applying a linear transformation. Such a transformation preserves most other properties of the data.

Sparsity

The sparsity denotes how many different values the data contains, and it can be measured as follows:

$$sparsity(S) = \frac{\|\{s \in S\}\|}{\|S\|} = \frac{\#\text{ of unique values in } S}{\|S\|}$$

The sparsity is especially important when generating columns that will be joined together; by drawing from a small set of distinct values, many tuples will join, and the join result will be big. Or if we use a very low sparsity, the probability of any collisions is small, and the join result could be (close to) empty.

Skew

Skew or skewness is a measure for the "shape" of the distribution. At a low skew, all values are close to each other. At a high skew, there will be a few outliers with a value that is quite different from the rest of the data. The exact definition of skew varies in AQP literature. The two most common definitions:

- Skew is the exponent s in the Zipf Distribution [7, 4, 6, 31, 16]. The Zipf distribution has PDF $f_s(k) = \frac{k^{-s}}{\zeta(s)}$, where $\zeta(s) = \sum_{k=1}^{\infty} \frac{1}{s^k}$.
- Skew is some undefined measure. High skew indicates that the data is "asymmetric" and has "a long tail" and that uniform sampling performs poorly [22, 9]

We will refer to the second definition as *effective skew*, and to the first definition as *Zipf-skew*. Zipf-skew has a concrete definition, which is helpful when doing experiments. Effective skew is more general, and leaves the choice of a precise measure of skewness to the reader. There exist many measures for skewness [3, 18]. The Zipf-skew is a measure/parametrization of effective skew for data following the Zipf distribution. We will also describe a different, more general measure of effective skew, which we call bias.

Bias

In order to compare the quality of aggregation over uniform samples to aggregation over weighted samples, we would like to generate data that will make uniform samples misrepresent the data with high probability. A high bias should denote that the plugin estimator with uniform sampling yields imprecise results. It is known that data with big outliers has this property; the amount of outliers present in the uniform sample varies wildly because of their rarity. At the same time, they influence the estimated aggregate a lot. However, we would like to use a more objective measure for the compatibility of data with uniform sampling techniques. For this purpose we can use the measure of variance of the mean estimator over uniform samples [8]:

$$bias(S) \equiv variance(\bar{y}) = E(\bar{y} - \bar{Y})^2 = \frac{\hat{\sigma}^2}{n}(1 - f)$$

Here $y \subset S$ is the uniform sample of size n, $\bar{y}$ denotes its average, which should approximate $\bar{Y}$, the true average. The dataset has total size N, and $\hat{\sigma}^2 = \frac{\sum_{i=1}^{N}(y_i - \bar{Y})^2}{N-1}$. f denotes the sampling fraction, n/N.

$$bias(S) = \frac{\hat{\sigma}^2}{n}(1 - f) = \frac{\left(\frac{\sum_{i=1}^{N}(y_i - \bar{Y})^2}{N-1}\right)}{n}(1 - f) = \frac{\sum_{i=1}^{N}(y_i - \bar{Y})^2}{(N-1)n}(1 - f)$$

For different aggregations different measures could be used.

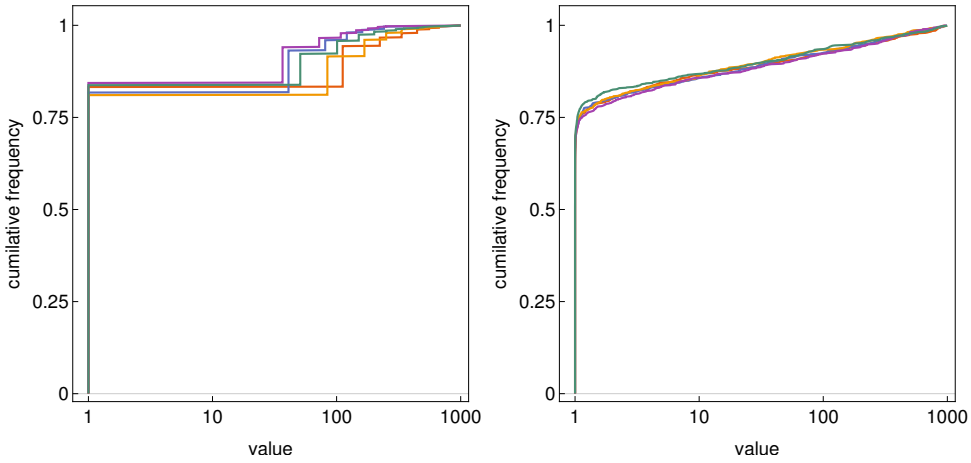


FIGURE 5.1: Empirical CDF of five datasets generated by Algorithm 4 (left) and Algorithm 5 (right). $n = 1000$, weightratio $= 1000$, (Zipf-)skew $= 2$ (left) and (Polynomial-)skew $= 32$ (right)

5.1.1 Generate data given skew and weight ratio

We can generate data using the Zipf distribution and then apply a linear transformation to enforce a certain weight ratio.

input: Desired parameters (Zipf-)skew, weight ratio $\in \mathbb{R}$ and size $n \in \mathbb{N}$
output: R, the generated data.
`// Take a size` n `sample of the Zipf distribution`
$R \leftarrow \text{sample}\left(\frac{k^{-\text{skew}}}{\zeta(\text{skew})}, n\right) \in \mathbb{N}^n$
`// Scale` R `to attain the desired weight ratio`
$R \leftarrow R - \min(R)$
$R \leftarrow R / \max(R)$
$R \leftarrow R * (\text{weightratio} - 1) + 1$

Algorithm 4: Generating data with skew and weight ratio using the Zipf distribution

There is a downside to this approach; the rescaling depends on the maximum value in R, which varies wildly for the Zipf distribution. Hence the shape of the distribution will vary given fixed parameters. See Figure 5.1. To produce more stable output, we can use a different distribution instead. We try to produce similar results in an efficient and continuous manner by taking uniform random $u_i \in [0, 1]$, and taking these to the power *skew*, $R_i = u_i^{\text{skew}}$. Since we do not use the Zipf distribution anymore, the meaning of "skew" has changed; we will refer to the exponent of u_i^{skew} as *Polynomial-skew* if the skewness measure can not be deduced from the context.

The CDF of this function is $F(x) = x^{1/\text{skew}}$ and the corresponding PDF is $(1 - \frac{1}{\text{skew}})x^{-1/\text{skew}}$. See Figure 5.2 for a plot of the CDF for different skew and weight ratio.

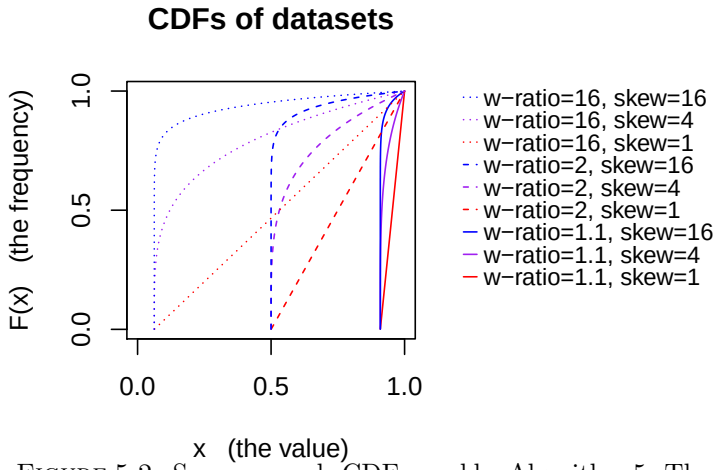


FIGURE 5.2: Some example CDFs used by Algorithm 5. The x-axis is expressed in units of w-ratio. The distribution with (Polynomial-)skew $= 1$ tends to the uniform distribution as w-ratio tends to infinity.

input: Desired parameters (Polynomial-)skew, weightratio $\in \mathbb{R}$ and size $n \in \mathbb{N}$
output: R, the generated data.
`// Take a size` n `sample of a continuous distribution`
$R \leftarrow \text{sample}\left((1 - \frac{1}{\text{skew}})x^{-1/\text{skew}}, n\right) \in \mathbb{R}^n$
`// Scale` R `to attain the desired weight ratio`
$R \leftarrow R * (\text{weightratio} - 1) + 1$

Algorithm 5: Generating data with (Polynomial-)skew and weight ratio using a continuous distribution

We would like to control sparsity as well, so the data can be used to test joins with different output sizes. This could be done by (partially) rounding the data. However, the sparsity would then influence the effective skew as well, especially at low sparsity (few unique values). The effect on the effective skew is not straightforward. To make sure that our parameters still represent the properties of the data we want to describe, we switch from skew to bias as measure of effective skew. The resulting method is described in the following section.

5.1.2 Generate data given sparsity, bias and weight ratio

We have tried several ways of generating a dataset parametrized by sparsity, bias and weight ratio. It turns out that tuning common discrete distributions to suit this purpose can be rather cumbersome. There are also some very "artificial" ways of generating such a dataset, for example by first choosing the distribution of unique values and then adding duplicate values to achieve some target bias. We finally choose to generate our data by first sampling from a polynomial distribution (this way we can choose the bias) and then

rounding the results (this will enforce some sparsity). The resulting data has a "natural" distribution.
The polynomial distribution we choose has the following CDF:

$$f(x) = x^{\frac{1}{\lambda}}$$

Here λ is the Polynomial-skew, a parametrization of effective skew. Suppose $\tilde{S} = sample(n, f(x))$, a sample of size n following this distribution; $\mathbb{P}(s \leq x) = f(x)$. We find our final sample S by applying $g : \mathbb{R} \to \mathbb{Q}$ to all elements of S:

$$g(x) = \frac{\lfloor \alpha x \rfloor}{\alpha}$$

So if we enumerate elements of $\hat{S} = \{\hat{S}_i\}_{i \in [1,2,\cdots,N]}$, then $S = \{g(\hat{S}_i)\}_{i \in [1,2,\cdots,N]}$.

We want to find λ and α so that the resulting data has some target *bias* and *sparsity*. First we tried to express λ and α in terms of *bias* and *sparsity* algebraically. We identified relevant ranges of values of λ and α, and observed that *sparsity* will behave in a roughly linear fashion with respect to β and μ if $\alpha = 2^\beta$ and $\lambda = 3^\mu$, while the *bias* behaves in an exponential manner. This "linearization" will make it easier to find some optimal set of parameters. Note that this problem can be written as a minimization problem where we score a solution using the quadratic distance between the target and measured *bias* and *sparsity*.

After trying out some existing solvers, we concluded that they did not exhibit the properties needed to find the solution of our problem. Our minimization problem exhibits several difficult properties:

- it is multidimensional: we have two input parameters and two target measures (these targets can be combined into one objective function)
- it is very non-uniform: some regions of α and λ will yield little change in the target measures, while there may be rapid change in other regions. We partially solved this problem by parametrizing using β and μ instead.
- it is computationally expensive: to evaluate the score function, we have to create a sample and measure its *bias* and *sparsity*, which takes $\mathcal{O}(n)$ time, with a reasonably big constant because of the random number generation
- it is random: different generated samples will have different *bias* and *sparsity*

Gradient descent like methods are infeasible, since they will try to estimate the gradient by evaluating the objective function at some x and $x + \Delta$ for some small Δ, which does not work well because of the randomness of this objective function. Grid search methods are infeasible, since they will evaluate the objective function many times; the $\mathcal{O}(n)$ time complexity of our objective function is very prohibitive. Hence we implemented a simulated annealing solver [14]. The simulated annealing algorithm starts with some initial state β and μ, and randomly varies them, accepting changes for the worse at random depending on the temperature. The size of the random changes depends on the temperature as well (smaller changes at lower temperatures). Note that due to variance in *bias* and *sparsity* given some β and

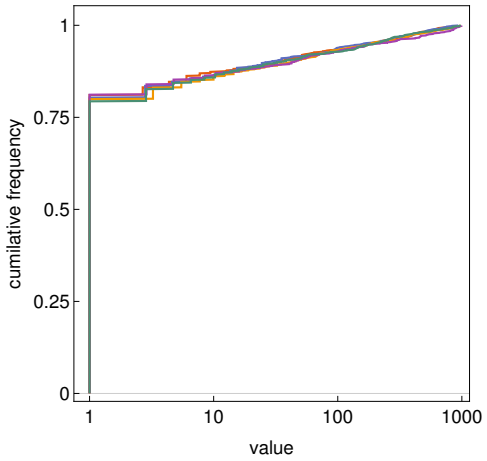


FIGURE 5.3: Empirical CDF of five datasets generated by the algorithm described in Section 5.1.2. $n = 1000$, weightratio $= 1000$, bias $= 16$ and sparsity $= 1/10$.

μ, we cannot improve β and μ by much after some point. Hence we stop the simulated annealing after some reasonable β and μ have been found, and generate multiple samples with these parameters and select the sample with the best score.

With this technique we are able to find datasets with *sparsity* and *bias* within 0.1% range of the desired *sparsity* and *bias* in roughly $n/1000$ seconds, which is good enough for our purposes. Note that this approach can easily be adapted to optimize for different notions of *bias*, which is useful since different aggregation operators may require different sparsity measures.

The distribution of the data is much more stable than that of the scaled Zipf-distribution (Algorithm 4) as can be seen by comparing Figure 5.3 with Figure 5.1.

5.2 Assessing the Quality of Heuristic Weighted Sampling

First we will describe a measure for the quality of a heuristic weighted sample, and describe a way of computing this measure (Section 5.2.1). Then we discuss how we have assessed the quality of HWS on a representative set of relations in a computationally feasible fashion (Section 5.2.2). Finally, we use the results of our experiments to describe the properties of HWS (Section 5.2.3). The experimental results in this section suggest that HWS yields samples that are very similar to true weighted samples for a reasonable intermediate sample size k. This implies that the difference between the α and β factor defined in Section 4.1.5 is reasonably small.

5.2.1 Theory behind the experiments

We assess the probability that a sample is selected through Heuristic Weighted Sampling, and compare this probability to the desired (true) selection probability. In other words; our goal is to determine the error

$$\epsilon_w = \sup_{s \subset R}(\epsilon_w(s)) = \sup_{s \subset R}(|\mathbb{P}_{\text{ws}}(s) - \mathbb{P}_{\text{aws}}(s)|)$$

up to some significance level σ. Note that this error will depend on the weight distribution on R. One might estimate these probabilities by using direct simulation; for some given w, repeatedly sample using (heuristic) weighted sampling, and compare the frequencies. However, this is not computationally feasible for all but the smallest relations, since we will have to repeat the experiment $\mathcal{O}(n^k)$ times to get reasonable statistics. In this section we provide a computationally more efficient way of estimating the selection probabilities.

In this section we will use sampling with replacement everywhere. This is reasonable since Heuristic Weighted Sampling can only work well if there is a low probability of collisions. In this section S and U denote random variables, while we denote the random variates with s and u. For the probability $\mathbb{P}(S = s)$ we use the shorthand $\mathbb{P}(s)$, for similar cases we use a similar shorthand. First, consider the true selection probability of some sample s:

$$\mathbb{P}_{\text{ws}}(s) = \#\text{perms}(s) \prod_{i=1}^{m} \frac{w(s_i)}{w_R}$$

Where $w_R \equiv \sum_{i=1}^{n} w(R_i)$. s is an unordered set. Hence we need to account for its permutations. $\#\text{perms}(s)$ is the number of different orderings of s. We count duplicates by id, not by value. In general we can count the number of orderings as follows:

$$\#\text{perms}(s) = \frac{\|s\|!}{\prod_{r \in R} \#\text{occurrences}(r \in s)!}$$

Here $\#\text{occurrences}(r \in s) = \|\{r = x | x \in s\}\|.$, and $0! = 1$.

The probability that s is selected using Heuristic Weighted Sampling can be expressed as follows:

$$\mathbb{P}_{\text{aws}}(s) = \sum_{u \subset R} \mathbb{P}(s|u)\mathbb{P}(u)$$

Notice that $\mathbb{P}(u) = \#\text{perms}(u)\frac{1}{n}^k$, since u is an (unordered) uniform sample with replacement. Now consider the probability that s is selected given that u was selected, $\mathbb{P}(s|u)$:

$$\mathbb{P}(s|u) = \#\text{perms}(s)\mathbb{1}(s \subset u) \prod_{i=1}^{m} \mathbb{P}(s_i \in u)$$

Here $\mathbb{1}(s \subset u)$ is the indicator function, which is 1 if $s \subset u$ and 0 otherwise.

$$\mathbb{P}(s_i \in u) = \frac{w(s_i)\#\text{occurrences}(s_i \in u)}{w_u}$$

Where $w_u \equiv \sum_{i=1}^{m} w(u_i)$. We now have all ingredients to calculate $\mathbb{P}_{\text{aws}}(s)$ directly by looping over all possible u. This is, however, too computationally expensive. Instead we would like to estimate $\mathbb{P}_{\text{aws}}(s)$ by evaluating random u (this is a Monte Carlo approach). However, most u do not contain s and do not contribute to $\mathbb{P}_{\text{aws}}(s)$. Hence we instead evaluate random $\bar{u}$, such that $u = \bar{u} \cup s$ (here $\|\bar{u}\| = k - m$). Let $\widetilde{u}$ be the ordered version of $\bar{u}$. Drawing random ordered sets is easier than drawing unordered sets. We rewrite $\mathbb{P}_{\text{aws}}(s)$:

$$\mathbb{P}_{\text{aws}}(s) \approx n^{k-m} \text{avg}_{\widetilde{u}} \frac{\mathbb{P}(s|\bar{u} \cup s)\mathbb{P}(\bar{u} \cup s)}{\#\text{perms}(\bar{u})}$$

In this formula n^{k-m} is the total number of possible $\bar{u}$. The term $\#\text{perms}(\bar{u})$ allows us to convert between the ordered and unordered probabilities. Some components of the above formulas are quite sensitive to over and/or underflow. To avoid these issues we have rewritten all equations in logarithmic form.

5.2.2 Practical setup of the experiments

The properties of the weights w that have a big impact on performance of HWS are the skew and w-ratio. We expect that HWS will perform better (produce better results and/or require smaller U) for lower weight ratios, since a low w-ratio (close to 1) limits the relative size of outliers in the weight function. The skew parametrizes the fraction of high outliers (indirectly); we expect HWS to perform worse if outliers are rare, since we typically need a bigger U to capture a representative fraction of the outliers. Hence we can generate the weights by using Algorithm 5 from Section 5.1.1. All code used in the experiments can be found online [29].

Given some relation R and its weight function w, we compare the selection probability using WS with the selection probability using HWS by evaluating $\mathbb{P}_{\text{ws}}(s)$ for random s and by approximating $\mathbb{P}_{\text{aws}}(s)$ using the method described in Section 5.2.1. We obtain ϵ_w at significance level σ by doing this for n_{outer} different s, and taking the $\lceil \sigma * n_{\text{outer}} \rceil$-st biggest measured $\epsilon_w(s)$. Notice that the quality of this estimate of the error will depend on both the number of U used in the calculation of $\mathbb{P}_{\text{aws}}(s)$ (n_{inner}) and the number of s considered (n_{outer}). We assume convergence of the Monte Carlo method of order $1/\sqrt{n}$, and choose n_{inner} and n_{outer} sufficiently large accordingly. For $m = 100$ we used $n_{\text{inner}} = n_{\text{outer}} = 1000$. For bigger values of m, it is not computationally feasible to calculate a confidence interval this way. Instead, n_{outer} is chosen smaller (~ 10 points) to obtain ϵ with an estimated uncertainty based on the variance in the results.

To summarize, experiments were parametrized using the following variables:

Variable	Derived?	Description
n	N	size of R
k	N	size of U
m	N	size of S
skew	N	$w \sim \text{rand}([0,1])^{\text{skew}}$
w-ratio	N	$w_{\max}/w_{\min}$
n_{outer}	N	number of S tried
n_{inner}	N	number of U tried
k-factor	Y	$k/(w\_\text{ratio} * m^2)$
n-ratio	Y	n/k

5.2.3 Experimental results

The experimental results are summarized in Figure 5.4. In each plot, we vary one parameter (while keeping the rest of the parameters constant) to show its influence on the error in the selection probability. In this section we will discuss these parameters one by one. But first some general remarks on the experiments.

To obtain confidence intervals, very time intensive simulations have to be run. The theory behind these experiments is described in Section 5.2.1. The algorithm iterates over $n_{\text{inner}} * n_{\text{outer}} * k\_\text{factor} * w\_\text{ratio} * m^2$ elements while obtaining S from U. To obtain reasonable $\sigma = 0.99$ confidence intervals, we need $n_{\text{inner}} = n_{\text{outer}} = 1000$. We chose $m = 100$ for most experiments, which is big enough for practical application but small enough to keep the number of iterations down. For these values of m, n_{inner} and n_{outer}, the number of iterations needed is $10^{10} * k\_\text{factor} * w\_\text{ratio}$. As long as the k-factor and w_ratio are relatively small, the runtime can be kept reasonable.

The experiments were executed on the SciLens cluster, on four machines boasting 1TB memory and 96 logical cores each. We implemented code to find confidence intervals in the R language, but we wrote the performance critical code (the kernel) in C++. The code was (embarrassingly) parallelized. The most time intensive simulations took 2 weeks on one machine.

Sample size m

In the top left graph in Figure 5.4 it is shown that the size of m does not have a big influence on the error ϵ, given that we keep the k-factor (and other parameters) fixed. This is evidence that our heuristic for letting the size of U increase quadratically with m to keep the error constant is a good approximation (at least for small values of m). The other plots were all made with $m = 100$, which is the largest value we could use while still computing reasonable $\sigma = 0.99$ confidence intervals. We did run experiments with bigger m, which only provided a rough estimate of ϵ. The results indicated that HWS behaves similar when taking bigger samples.

Skew

The influence of the skew on ϵ depends on the w-ratio. This behavior can be seen in the top middle and top right graphs in Figure 5.4. For small w-ratio, the influence of skew is also small, as is to be expected. For larger w-ratio, we observed that ϵ will first increase with skew and later decrease with skew.

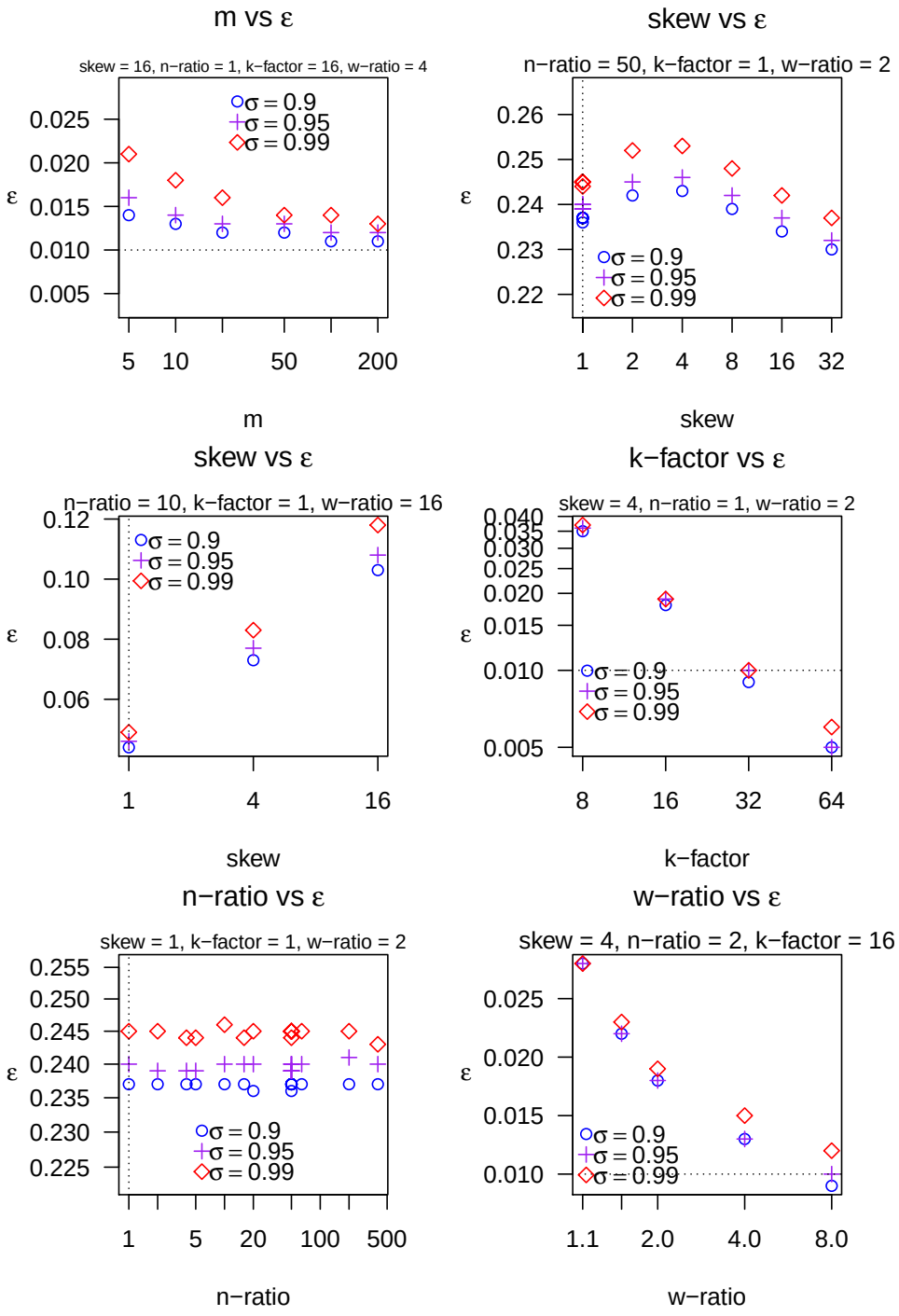


FIGURE 5.4: Measuring ϵ, the relative error in selection probability of a sample using HWS with respect to the selection probability of the same sample using WS. All plots are logarithmic in the x-axis, only k-factor vs ϵ is plotted with a logarithmic y-axis. The top row tries to capture the influence of the skew and w-ratio of the data on ϵ. The bottom row shows the influence of other variables on ϵ. In all of these plots, $m = 100$.

The skew for which ϵ attains its maximum value increases with the w-ratio. The initial increase of ϵ (for small values of skew) can be explained by the fact that U is less likely to include a representative fraction of outliers in R as the skew increases. The decrease of ϵ for big values of skew can be explained by the overall flattening of the distribution from which R is drawn; it is likely that true weighted sampling will yield samples without any large outliers if they are too rare, and hence, HWS will produce more realistic samples.

k-factor

The size of the k-factor influences ϵ in a very direct and predictable way (see the bottom left graph in Figure 5.4). Based on our experiments, the rate of convergence is $\mathcal{O}(\frac{1}{k\_\text{factor}})$. As the size of $|U| = k\_\text{factor} * w\_\text{ratio} * m^2$ increases, ϵ decreases. Based on the heuristics in Section 4.1.3, $k\_\text{factor} = 1$ should be big enough to sample $S \subset U$ without over-representing small weights at significance level $\sigma \approx 0.86$ (given that a fair fraction of big weights is present in U). For $\sigma = 0.9$, in most cases $k\_\text{factor} = 1$ yields $\epsilon \approx 0.20 \pm 0.07$. Other variables do also influence ϵ, as can be seen in Figure 5.4. Redefining the k-factor to take these other variables into account in a more precise way, so a fixed k-factor yields a (nearly) constant (and predictable) ϵ, is a future goal.

n-ratio

The ratio n/k (or the n-ratio) does not influence the quality of samples. This is illustrated in the lower middle plot in Figure 5.4. This plot shows this property for one specific skew, w-ratio and k-factor. However, we tested this for a multitude of parameter values and it never influences ϵ. We conclude that the complexity of HWS is independent of n in practice.

w-ratio

The amplitude of the influence of the w-ratio (defined as $\frac{w_\text{max}}{w_\text{min}}$) on ϵ depends on the skew. However, all other things being equal, a bigger w-ratio yields a smaller ϵ. The reason for this is that the size of U will depend linearly on the w-ratio (given that we have a constant k-factor) in an attempt to correct for the bigger required sample size if there is a bigger w-ratio. The fact that ϵ reduces at bigger w-ratios means that we are over-correcting. Notice that rate with which ϵ reduces is much smaller than $\mathcal{O}(\frac{1}{w\_\text{ratio}})$, as we would expect if higher w-ratio would only influence the size of U; this means that we do need (part of) the correction.

5.3 Comparison of Sample-Join Algorithms

In this section we compare the runtime and quality of different sample-join algorithms.

5.3.1 Comparing the quality

In this section, we will compare the quality of Stream Sample Join with that of WS-Join experimentally. See Figure 5.5 for the results. In this experiment, the skew of $R_2.C$ is varied, while everything else remains constant. This way

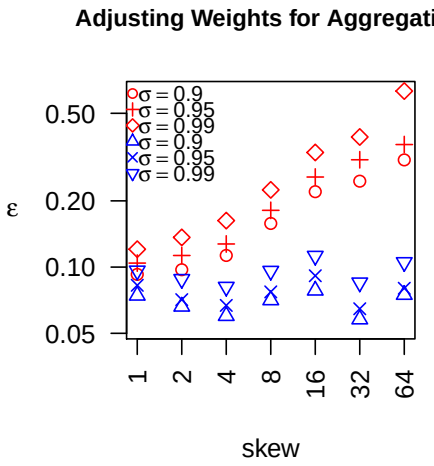


FIGURE 5.5: Log-log plot of skew in column C vs ϵ. Here ϵ is the relative error of an estimation of the sum over column C, using a weighted sample-join. The upper/red points were created using classic Stream Sample Join with aggregation. The bottom/blue points was created using WS-Join, with weights linear in C. Note that our approach is much more robust under skew.

we can test the robustness of aggregation over Stream Sample Join and WS-join with respect to skew in the aggregation column. The weight ratio of $R_2.C$ is fixed at 1024. $R_1.A$ and $R_2.A$ are chosen uniform randomly in $[0, 100]$. The relation R_1 contains $n_1 = 10^7$ rows, R_2 contains $n_2 = 10^3$ rows and the sample size is set to $m = 10^3$. WS-join is configured to produce a weighted sample with weight proportional to the C value.

Note that the full join result contains approximately

$$\frac{n_1 * n_2}{\#\text{unique values in } R_1.A \text{ and } R_2.A} = \frac{10^7 * 10^3}{100} = 10^8$$

tuples. While our sample size is very small ($m = 10^3$ is 0.001% of the data), the error remains in the order of 10% with significance 0.99 even if the aggregation column is extremely skewed, if we use WS-join. While the aggregation quality of Stream Sample Join is good if there is little skew in the data, the error increases drastically with skew.

While the runtime of WS-Join is similar to that of Stream Sample Join (see section 4.1.5), we observe that WS-Join is much more robust under skew.

5.3.2 Comparing the run-time

In Section 4.1.5, we have compared the run-times of the sample-join algorithms in a theoretical setting. In this section, we confirm that our theoretical run-time model is correct, and study the performance of our sample-join algorithms in a realistic setting.

We compare US-join, WS-join and HWS-join. WS-join is tested twice; one implementation uses reservoir sampling with exponential jumps, the other uses reservoirs sampling without exponential jumps [12]. These algorithms are tested for different sample sizes, and for different read speeds T_1

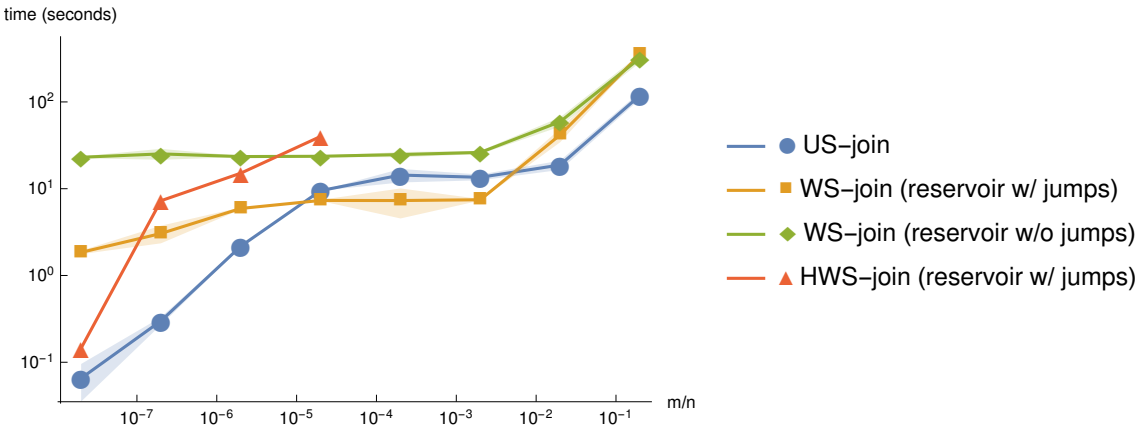


FIGURE 5.6: R_1 on HDD, R_2 in memory – log-log plot of runtime vs relative sample size. $n_1 = 2 * 10^8$ and $n_2 = 2000$. Experiments were run 5 times, the shaded areas mark the standard deviation.

and T_2 to R_1 and R_2 (T_1 and T_2 are the time needed to do a random read in R_1 and R_2 respectively). The read speeds are controlled by storing R_1 and R_2 on either a hard disk drive or in main memory. In this system the difference in non-sequential read speed is roughly $T_{\text{HDD}} \approx 1.6 * 10^5 T_{\text{memory}}$, more differences in reading behavior between HDD and main memory are discussed in Section 4.1.5).

R_1 took 200MB on the hard drive, which is much more than the 30MB cache in the hard drive, avoiding caching effects. Further more, the Linux page file and CPU-cache were flushed in between experiments. We will now discuss the results of our experiments for four different settings.

R_1 on HDD, R_2 in memory

See Figure 5.6. As expected, computation of n_1 random variates is the dominating component of the runtime of WS-join without exponential jumps. For WS-join with exponential jumps, the time needed for random number generation becomes the most time consuming component for large m/n. For small m/n, the time required to sequentially read all of R_1 from disk dominates the runtime. US-join starts out very fast for small sampling fractions, because it can avoid reading all of R_1. However, when the average distance between random reads is less than or equal to the size of blocks fetched by the hard drive, US-join will read all of R_1 as well, and it loses its advantage.

From a theoretical point of view, we know that HWS-join can only be competitive if T_1 is much smaller than T_2. Since T_1 is much bigger than T_2, obtaining a uniform sample of size $m^2 \frac{w_{\max}}{w_{\min}}$ is the dominating run time component of the HWS-join algorithm. Assuming $\frac{w_{\max}}{w_{\min}} = 1$ (the best case), HWS-join is a factor m slower than US-join, and it becomes slower than WS-join with exponential jumps for $m \approx 50$. This experiment confirms that the runtime is problematic for all but the smallest samples. In this experiment (and all the run-time comparisons in this section) HWS-join is not evaluated for any m such that $\frac{w_{\max}}{w_{\min}} m^2 \geq n_1$; once we sample all of R_1 in our intermediate sample U, it is always more efficient to use WS-join.

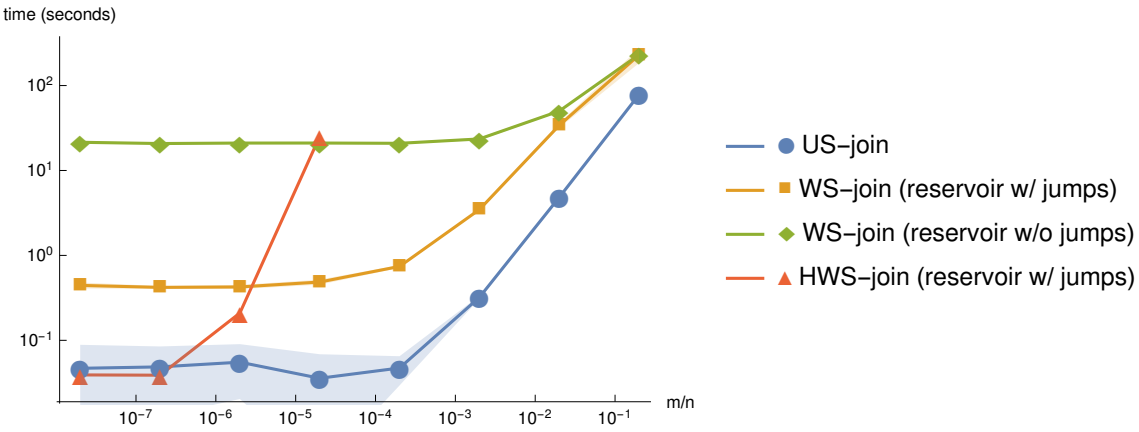


FIGURE 5.7: R_1 in memory, R_2 on HDD – log-log plot of runtime vs relative sample size. $n_1 = 2 * 10^8$ and $n_2 = 2000$. Experiments were run 5 times, the shaded areas mark the standard deviation.

R_1 in memory, R_2 on HDD

See Figure 5.7. This case is similar to the setting explored theoretically in Section 4.1.5. We will compare the different approaches by assuming that;

- A size αm sample obtained using US-join is as good for estimating aggregations as a size m sample obtained through HWS-join for a constant α
- A size βm sample obtained using US-join is as good for estimating aggregations as a size m sample obtained through WS-join for a constant β

Here $\beta \leq 1$ and $\beta < \alpha$. We will assume that $\alpha = 0.7$ and $\beta = 0.5$, but the analysis can easily be adopted for other values of α and β.

Since random access in R_2 is much more expensive than random access in R_1, joining the sample in R_1 with R_2 (using the mini-join) will be the bottleneck if we do not read all of R_1 (US-join and HWS-join) and m is small. Because of this it can be efficient to join (using mini-join) only the "important" part of the sample in R_1 with R_2; that is precisely what HWS-join does. HWS-join is the most efficient choice for m smaller than ~ 4000. US-join is the most efficient choice for m in between ~ 4000 and n_1.

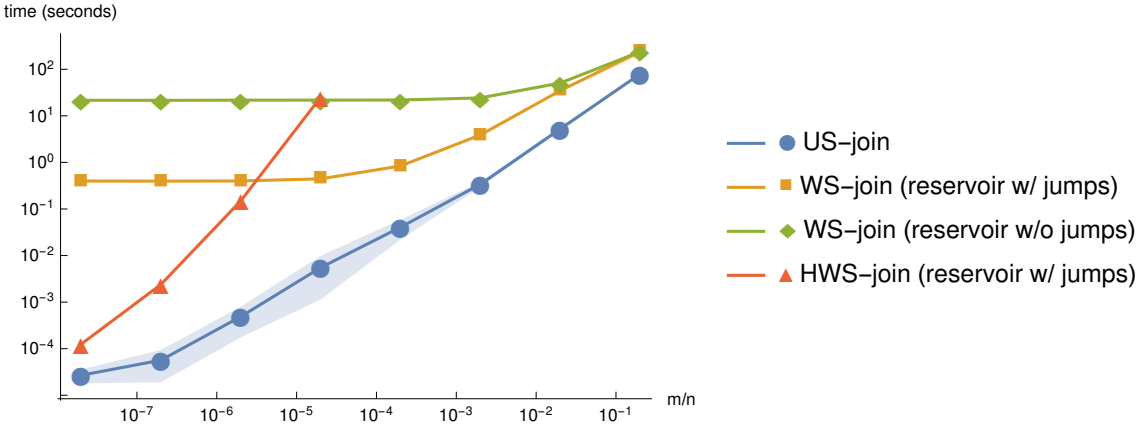


FIGURE 5.8: R_1 and R_2 in main memory – log-log plot of runtime vs relative sample size. $n_1 = 2 * 10^8$ and $n_2 = 2000$. Experiments were run 5 times, the shaded areas mark the standard deviation. One outlier in the leftmost point of US-join, with a runtime more than a factor 250 bigger than the other four measurements, was removed from the data.

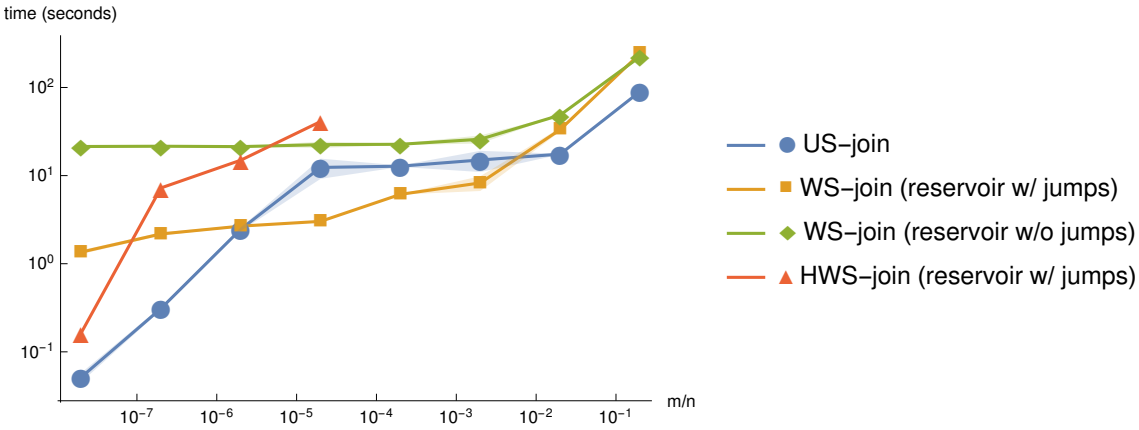


FIGURE 5.9: R_1 and R_2 on HDD – log-log plot of runtime vs relative sample size. $n_1 = 2 * 10^8$ and $n_2 = 2000$. Experiments were run 5 times, the shaded areas mark the standard deviation.

R_1 and R_2 in main memory

See Figure 5.8. This setting is very similar to the setting with R_1 in memory and R_2 on HDD. The difference is that the mini-join with R_2 is much faster when R_2 is in main memory. Because of this, the cost of the mini-join does not dominate the runtime of US-join and HWS-join for small m. Hence US-join and HWS-join perform even better relative to the runtime of WS-join for small m. Another effect is that HWS-join is almost a factor m slower than US-join (for all m), which renders it relatively useless.

R_1 and R_2 on HDD

See Figure 5.9. If all data is stored on a hard disk drive, sequentially scanning becomes relatively efficient. For m larger than ~ 1000 WS-join is the most efficient option. For smaller m, US-join is competitive. However, since the

read speeds in R_1 are as fast as the read speeds in R_2, HWS-join is roughly a factor m slower than US-join.

Conclusions from the run-time comparison

On conclusion is that the theoretical framework presented in section 4.1.5 does describe the overall behavior of the runtime of US-join, WS-join and HWS-join accurately. However this framework can only *estimate* which algorithm will be best in what case. How well the algorithms perform in reality does depend on practical details. We have compared the different sample-join approaches for one specific setting, but because of the multitude of measurements it is easy to loose sight of the bigger picture. In Figure 5.10 we have summarized a small part (only one sample size) of the results presented in this section, using a bar chart. The differences between the different sample-join algorithms seem less profound when presented on a logarithmic (rather than linear) scale. We conclude that US-join and HWS-join can provide a big improvement in runtime compared to the state of the art, depending on the setting.

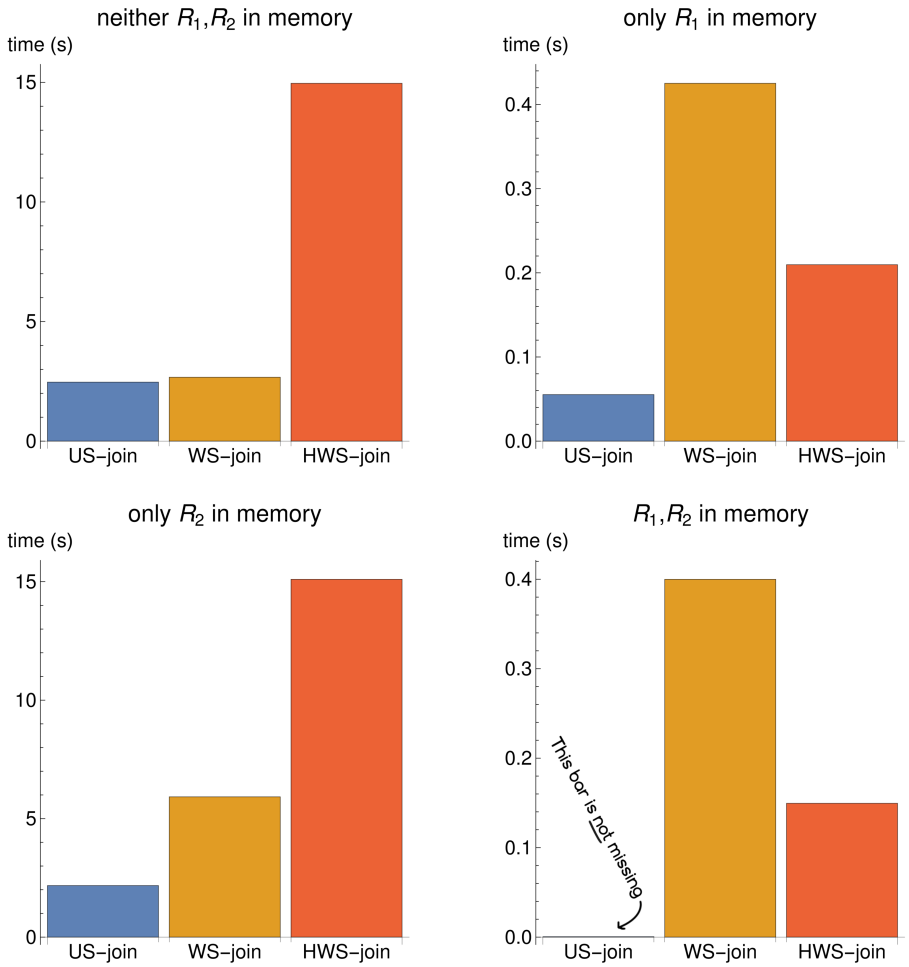


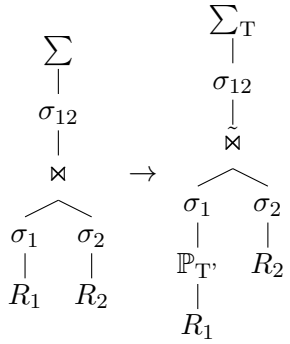
FIGURE 5.10: Lower is better. The yellow bar has the same speed as the current state of the art. Comparison of the runtimes of US-join, WS-join (with exponential jumps) and HWS-join. The bar that is *not* missing (US-join with R_1 and R_2 in memory) has height 0.00049 seconds (a factor 819 speedup over the state of the art). The sample size is $m = 400$, the sizes of R_1 and R_2 are $n_1 = 2 * 10^8$ and $n_2 = 2000$ respectively.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

The main goal of this thesis was to efficiently approximate aggregations over joins.



We have shown that it is indeed possible to do this using state of the art AQP techniques. We have shown three new strategies that improve on the current state of the art aggregation over Stream Sample Join:

1. We have extended Stream Sample Join to produce weighted samples (WS-join), without increasing the runtime. These weighted samples can be used to improve the precision of the aggregation, and are robust against skew in the aggregation column.

2. We have shown that the special case of Weighted Stream Sample Join where $\mathbb{P}_{\mathrm{T}'}$ is the uniform sampling operator (US-join) can be executed in time linear in the sample size, which is asymptotically faster than Stream Sample Join. The aggregation operator can be corrected for the resulting non-uniformity of the joined sample.

3. The weighted sampling operator $\mathbb{P}_{\mathrm{T}'}$ can be replaced by Heuristic Weighted Sampling (HWS-join). We have shown that this yields high quality samples, and that it is faster than WS-join and US-join in some cases.

It will depend on the exact setting which of the above strategies is the most efficient, however, we are always able to improve on Stream Sample Join, the current state of the art.

6.2 Future Work

In this section we discuss some of the possible continuations of the work presented in this thesis.

6.2.1 Experimenting with stratified sampling over WS-join

We combine Stratified Sampling (Section 3.2.2) with WS-join in Section 4.2. However, when using WS-join (see Section 4.1.2) for stratified sampling, we cannot choose the amount of tuples sampled from each stratum (without making big changes to the algorithm). As a result, there is a subtle difference between traditional Stratified Sampling, and the approach to Stratified Sampling we need. We use a weight function with constant weights within each stratum; we do not control the exact amount of tuples that is sampled from a stratum, but rather the expected amount of tuples that is sampled from a stratum. This subtle difference influences the theory slightly. However, we can still test the quality of estimates using stratified sampling experimentally. We expect that state of the art strata selection heuristics [31] can be applied without big adjustments. These experiments would be a good topic for future work.

6.2.2 Extending estimation of aggregation over joins for different aggregation functions

We have focussed on the sum aggregation, since this has been extensively studied. It is relatively easy to extend our approach to support aggregations based on the sum over some algebraic UDFs (User Defined Functions); we can add a column with the UDF applied to each element, and use this as our aggregation column. Some of the techniques described in this thesis may also be applicable to `min` and `max`, but this is less straight forward.

6.2.3 Determining the quality difference between US-join and Stream Sample Join

The US-join algorithm produces weighted samples with weights depending on the strata sizes in R_2. We expect that the aggregation quality of US-join differs a little from the aggregation quality of Stream Sample Join, but that the massive difference in run-time makes up for this. A rigorous evaluation of the quality of aggregations obtained using US-join in a practical setting could provide a strong argument in favor of US-join.

6.2.4 Data generation

The problem of generating data with certain properties is very common. We developed a data generator for the join setting (see section 5.1). In literature, data is usually generated by parametrizing the desired properties in the model that generates data. When properties are complicated and not independent, finding a model that can generate correct data is a difficult task. Our data generator reverses this approach. The relationship between input parameters and desired properties is never modeled explicitly. Instead, we use a black box solver to find a dataset with the desired properties. Finding such a solver is complicated by the multidimensionality of the solution space, the randomness of the problem, and the large time required to evaluate the quality measure for one set of input parameters. We propose a Simulated Annealing solver, and show that this approach can be effectively used to generate data with given sparsity, bias and weight ratio for the join setting.

Our data generation scheme could be released as a tool for other researchers. The code can already be found online [29], but has to be rewritten and tested extensively to guarantee good portability, stability and performance. Documentation has to be added. And it should be integrated with one or more popular languages such as R and Python for ease of use.

6.2.5 Sample synopsis

Samples and their distribution could be used as a novel synopsis type akin to wavelets and histograms [9]. The idea is to propagate a sample and its distribution through a logical query plan. This has been done in the field of Probabilistic Databases [26, 15, 13], by associating some "probability of existence" to every tuple. We instead keep a random sample of tuples, and store a (small) data structure describing the distribution of this sample. In this thesis we have focussed on doing this for just one join followed by an aggregation. In the future we can try to extend this so multiple query operations can follow each other. Ultimately, sample synopsis could be integrated into a practical approximate query engine such as BlinkDB [2] or SciBORQ [25].

Some of the results in this thesis show that this approach has potential:

- Existing weighted aggregation techniques can be applied; they are more efficient than uniform aggregation (Section 3.2)
- This synopsis type can be pushed down joins efficiently under some mild assumptions (Section 4.1)

However, much more work is required to settle on one (nearly) optimal synopsis type, and to fully explore its properties and compare it to other synopsis types.

Acknowledgements

I would like to thank my daily supervisors, Lefteris and Hannes from CWI, for their help and advice with the writing of this thesis. I would also like to thank Hans and Tristan from the UU, for reading about, listening to and grading my thesis. Thanks to Mark and Jeemijn for reading drafts of my thesis and providing feedback. Thanks to my office mates Mark, Benno and Thibault, and my former office mates Duc, Mrunal, Pedro and Bo for the distraction. Thanks to Mark, Till, Tim and Dean for the table tennis sessions. Thanks to Tom for his help with memory mappings. Thanks to Jan-Willem and Tom for their parallel-algorithm related distractions. The experiments in this thesis would not have been possible without the computational power provided by the SciLens cluster.

Bibliography

[1] Sameer Agarwal, Henry Milner, Ariel Kleiner, Ameet Talwalkar, Michael Jordan, Samuel Madden, Barzan Mozafari, and Ion Stoica. Knowing when you're wrong: Building fast and reliable approximate query processing systems. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 481–492, New York, NY, USA, 2014. ACM.

[2] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. Blinkdb: Queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 29–42, New York, NY, USA, 2013. ACM.

[3] Barry C Arnold and Richard A Groeneveld. Measuring skewness with respect to the mode. *The American Statistician*, 49(1):34–38, 1995.

[4] Brian Babcock, Surajit Chaudhuri, and Gautam Das. Dynamic sample selection for approximate query processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 539–550. ACM, 2003.

[5] Ran Canetti, Guy Even, and Oded Goldreich. Lower bounds for sampling algorithms for estimating the average. *Information Processing Letters*, 53(1):17–25, 1995.

[6] Surajit Chaudhuri, Gautam Das, and Vivek Narasayya. Optimized stratified sampling for approximate query processing. *ACM Trans. Database Syst.*, 32(2), June 2007.

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

[8] William G Cochran. *Sampling techniques.* John Wiley, 1953.

[9] Graham Cormode, Minos Garofalakis, Peter J Haas, and Chris Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases*, 4(1–3):1–294, 2012.

[10] Bolin Ding, Silu Huang, Surajit Chaudhuri, Kaushik Chakrabarti, and Chi Wang. Sample+ seek: Approximating aggregates with distribution precision guarantee. In *Proceedings of the 2016 International Conference on Management of Data*, pages 679–694. ACM, 2016.

[11] Jonathan Dursi. On random vs. streaming i/o performance; or seek(), and you shall find - eventually. `https://simpsonlab.github.io/2015/05/19/io-performance/`, 2015. [Online; accessed: 2016-11-15].

[12] Pavlos S Efraimidis and Paul G Spirakis. Weighted random sampling with a reservoir. *Information Processing Letters*, 97(5):181–185, 2006.

[13] Robert Fink, Jiewen Huang, and Dan Olteanu. Anytime approximation in probabilistic databases. *The VLDB Journal*, 22(6):823–848, 2013.

[14] Chii-Ruey Hwang. Simulated annealing: theory and applications. *Acta Applicandae Mathematicae*, 12(1):108–111, 1988.

[15] Ravi Jampani, Fei Xu, Mingxi Wu, Luis Leopoldo Perez, Christopher Jermaine, and Peter J Haas. Mcdb: a monte carlo approach to managing uncertain data. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 687–700. ACM, 2008.

[16] Niranjan Kamat and Arnab Nandi. Perfect and maximum randomness in stratified sampling over joins. *arXiv preprint arXiv:1601.05118*, 2016.

[17] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. Wander join: Online aggregation for joins. In *ACM SIGMOD International Conference on Management of Data, San Francisco, USA*, page 1, 2016.

[18] HL MacGillivray. Skewness and asymmetry: measures and orderings. *The Annals of Statistics*, pages 994–1011, 1986.

[19] Shigeru Mase. Approximations to the birthday problem with unequal occurrence probabilities and their application to the surname problem in japan. *Annals of the Institute of Statistical Mathematics*, 44(3):479–499, 1992.

[20] Frank H Mathis. A generalized birthday problem. *SIAM Review*, 33(2):265–270, 1991.

[21] Xiangrui Meng. Scalable simple random sampling and stratified sampling. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 531–539, 2013.

[22] Rajeev Motwani, Rina Panigrahy, and Ying Xu. Estimating sum by weighted sampling. In *International Colloquium on Automata, Languages, and Programming*, pages 53–64. Springer, 2007.

[23] Jan C. Neddermeyer. Computationally efficient nonparametric importance sampling. *Journal of the American Statistical Association*, 104(486):788–802, 2009.

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

[25] Lefteris Sidirourgos, Martin L Kersten, Peter A Boncz, et al. Sciborq: Scientific data management with bounds on runtime and quality. In *CIDR*, volume 11, pages 296–301, 2011.

[26] Dan Suciu, Dan Olteanu, Christopher Ré, and Christoph Koch. Probabilistic databases. *Synthesis Lectures on Data Management*, 3(2):1–180, 2011.

[27] Jeffrey S Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57, 1985.

[28] L. Wasserman. *All of Nonparametric Statistics*. Springer Texts in Statistics. Springer New York, 2010.

[29] Abe Wits. Repository - estimating aggregations over joins. `https://github.com/usewits/MastersThesis`, 2016. [Online].

[30] Chak-Kuen Wong and Malcolm C. Easton. An efficient method for weighted sampling without replacement. *SIAM Journal on Computing*, 9(1):111–113, 1980.

[31] Ying Yan, Liang Jeff Chen, and Zheng Zhang. Error-bounded sampling for analytics on big sparse data. *Proceedings of the VLDB Endowment*, 7(13):1508–1519, 2014.

[32] Ping Zhang. Nonparametric importance sampling. *Journal of the American Statistical Association*, 91(435):1245–1253, 1996.