Sampling with SQL

Tom Moertel
blog.moertel.com

2024-08-23

Contents

Sampling is one of the most powerful tools you can wield to extract meaning from large datasets. It lets you reduce a massive pile of data into a small yet representative dataset that's fast and easy to use.

If you know how to take samples using SQL, the ubiquitous query language, you'll be able to take samples anywhere. No dataset will be beyond your reach!

In this post, we'll look at some clever algorithms for taking samples. These algorithms are fast and easily translated into SQL.

First, however, I'll note that many database systems have some built-in support for taking samples. For example, some SQL dialects support a `TABLESAMPLE` clause. If your system has built-in support—and it does what you need—using it will usually be your best option.

Often, though, the built-in support is limited to simple cases. Let's consider some realistic scenarios that are more challenging:

- We want to be able to take samples *with* and *without* replacement.
- We want to take *weighted* samples in which each item in the input dataset is selected with probability in proportion to its corresponding weight.
- We want to support the full range of weights we might expect to see in a FAANG-sized dataset, say between 0 to 10^{20} for frequency distributions

(e.g., clicks or impressions or RPC events) and between 0 to 1 with values as small as 10^{-20} for normalized probability distributions. In other words, weights are non-negative numbers, possibly very large or very small.
- We want to take deterministic samples. This property lets us take repeatable samples and, in some cases, helps query planners produce faster queries.

Sampling without replacement: the A-ES algorithm in SQL

In 2006, Pavlos S. Efraimidis and Paul G. Spirakis published a one-pass algorithm for drawing a weighted random sample, without replacement, from a population of weighted items. It's quite simple:

Given a population V indexed by $i = 1 \ldots n$ and having weights w_i:

1. For each v_i in V, let $u_i = \text{random}(0, 1)$ and $k_i = u_i^{1/w_i}$.
2. Select the m items with the largest keys k_i.

That algorithm has a straightforward implementation in SQL:

```
SELECT *
FROM Population
WHERE weight > 0
ORDER BY -LN(1.0 - RANDOM()) / weight
LIMIT 100  -- Sample size.
```

You'll note that we changed the ordering logic a bit. A straight translation would have been

```
ORDER BY POW(RANDOM(), 1.0 / weight) DESC
```

Our translation

```
ORDER BY -LN(1.0 - RANDOM()) / weight
```

is more numerically stable and also helps to show the connection between the algorithm and the fascinating theory of Poisson processes. This connection makes it easier to understand how the algorithm works. More on that in a moment.

Numerical stability and other tweaks

First, the numerical stability claim. Assume that our SQL system uses IEEE double-precision floating-point values under the hood. When the weights are large, say on the order of $w_i = 10^{17}$, it doesn't matter what the random value u_i is. The corresponding sort key $k_i = u_i^{1/w_i}$ will almost always be 1.0. Consider the interval $0.01 \leq u_i \leq 1$, representing 99% of the possible random values u_i. This entire interval gets mapped to 1.0 when $w_i = 10^{17}$:

```
# Python.
>>> w_i = 1e17
>>> math.pow(0.01, 1.0/w_i) == math.pow(1.0, 1.0/w_i) == 1.0
True
```

Likewise, when weights are small, say $w_i = 10^{-17}$, the corresponding sort key will almost always be zero. Consider the interval $0 \leq u_i \leq 0.99$, representing 99% of the possible random values u_i:

```
>>> w_i = 1e-17
>>> math.pow(0.0, 1.0/w_i) == math.pow(0.99, 1.0/w_i) == 0.0
True
```

For very large (or small) weights, then, the straightforward implementation doesn't work. The wanted sort ordering is destroyed when very large (or small) powers cause what should be distinct sort keys to collapse into indistinguishable fixed points.

Fortunately, logarithms are order-preserving transformations, so sorting by $\ln(u_i^{1/w_i}) = \ln(u_i)/w_i$ produces the same ordering as sorting by u_i^{1/w_i} when we're using mathematically pure real numbers. But the log-transformed version is much more stable when using floating-point numbers. Distinct random inputs u_i now produce reliably distinct sort keys k_i, even when the input weights w_i are very large or very small:

```
>>> [math.log(u_i) / 1e17 for u_i in (0.01, 0.010001, 0.99, 0.990001)]
[-4.605170185988091e-17, -4.605070190987759e-17,
 -1.005033585350145e-19, -1.0049325753001471e-19]

>>> [math.log(u_i) / 1e-17 for u_i in (0.01, 0.010001, 0.99, 0.990001)]
[-4.605170185988091e+17, -4.605070190987758e+17,
 -1005033585350145.0, -1004932575300147.1]
```

As a final tweak, we negate the sort keys so that instead of sorting by u_i^{1/w_i} **descending**, as in the original algorithm, we do an equivalent sort by $-\ln(u_i)/w_i$ **ascending**. Note the leading minus sign. The rationale for flipping the sign will become apparent when we discuss Poisson processes in the next section.

One last numerical subtlety. Why do we generate random numbers with the expression `1.0 - RANDOM()` instead of just `RANDOM()`? Since most implementations of `RANDOM()`, such as the PCG implementation used by DuckDB, return a floating-point value in the semi-closed range $[0, 1)$, they can theoretically return zero. And we don't want to take the logarithm of zero. So we instead use `1.0 - RANDOM()` to generate a random number in the semi-closed range $(0, 1]$, which excludes zero.

Does this algorithm actually work?

First, what do we mean by *work*? In this case, we'll say that we want the algorithm to produce samples that are equivalent to a succession of random draws, each draw removing an item from the population, and each draw fair with respect to the population that remains *at the time of the draw*. Because the population grows slightly smaller with each draw, the draws in the sample are not independent and identically distributed (iid). In practice, however, when

your samples are very small compared to your population of interest, and your weights are such that it's unlikely that iid draws would draw the same item more than once, you can generally pretend that the draws are iid and get away with it. (When you can't get away with it, you can use a reweighting scheme to extract unbiased estimates anyway. This subject is worth its own post, so I won't say more here.)

Anyway, it's not obvious that assigning a random number u_i to every row i, then sorting rows on $-\ln(u_i)/w_i$, and finally taking the top m rows is a recipe that would result in a sample that has the wanted properties. But it does.

The clearest way to understand what's going on is to first take a detour through the fascinating theory of Poisson processes. In short, **a Poisson process with rate λ is a sequence of arrivals such that the times between successive arrivals are all independent, exponentially distributed random variables with rate λ.**

Poisson processes have some important (and useful!) properties:

1. *They are memoryless.* No matter what has previously happened, the time until the next arrival from a Poisson process with rate λ is exponentially distributed with rate λ.
2. *They can be merged.* If you have two Poisson processes with rates λ_1 and λ_2, the arrivals from both processes form a combined Poisson process with rate $\lambda_1 + \lambda_2$. This holds for any number of processes.
3. *They win races in proportion to their rates.* In a race between the very next arrival from a Poisson process with rate λ_1 and the very next arrival from a Poisson process with rate λ_2, the probability that the first process will win the race is $\lambda_1/(\lambda_1 + \lambda_2)$.

Now that we know the basics of Poisson processes, there's just one more tidbit we need:

- *The uniform–exponential bridge.* If X is a random variable having a uniform distribution between zero and one, then $-\ln(X)/\lambda$ has an exponential distribution with rate λ.

With the uniform–exponential bridge in mind, we can begin to see what the algorithm is doing when it assigns every row a key $k_i = -\ln(u_i)/w_i$ and sorts the population by that key. It's running a race over all the rows in the population! In this race, each row arrives at the finish line at a time that's an exponentially distributed random variable with a rate corresponding to the row's weight w_i. The first m arrivals form the sample.

To prove that this race does the sampling that we want, we will show that it is equivalent to a succession of one-row draws, each draw being fair with respect to the population that remains at the time of the draw. Let the population's total weight be w, and consider an arbitrary row i with weight w_i. The algorithm will assign it an exponentially distributed random variable with rate w_i, which corresponds to the very next arrival from a Poisson process with the same rate.

Now consider all rows except i. They too correspond to Poisson processes with rates equal to their weights. And we can merge them into a combined process with rate $\sum_{j \neq i} w_j = w - w_i$.

Now, using the rule about Poisson races, we know that row i, represented by a process with rate $\lambda_1 = w_i$, will win the race against those other rows, represented by a combined process with rate $\lambda_2 = w - w_i$, with probability

$$\frac{\lambda_1}{\lambda_1 + \lambda_2} = \frac{w_i}{w_i + (w - w_i)} = \frac{w_i}{w}.$$

And since we chose row i arbitrarily, the same argument applies to all rows. Thus every row's probability of being drawn is equal to its weight in proportion to the population's total weight. This proves that running a "race of exponentials" lets us perform one fair draw from a population.

But, after we've drawn one row, what's left but a new, slightly smaller population? And can't we run a new race on this slightly smaller population to correctly draw another row?

We can. And, since Poisson processes are memoryless, we do not have to generate new arrival times to run this new race. We can reuse the existing arrival times because the arrivals that have already happened have no effect on later arrivals. Thus the next row we draw using the leftover arrival times will be another fair draw.

We can repeat this argument to show that successive rows are chosen fairly in relation to the population that remains at the time of each draw. Thus algorithm A-ES selects a sample of size m by making m successive draws, each fair with respect to its remaining population. And that's the proof.

Tricks for faster samples

Most large analytical datasets will be stored in a column-oriented storage format, such as Parquet. When reading from such datasets, you typically only have to pay for the columns you read. (By "pay", I mean wait for the query engine to do its work, but if you're running your query on some tech company's cloud, you may actually pay in currency too.)

For example, if your dataset contains a table having 100 columns but you need only four of them, the query engine will usually only read those four columns. In row-oriented data stores, by contrast, you'll generally have to decode entire rows, even if you only want four out of the 100 values in each row. Additionally, most column-oriented stores support some kind of filter pushdown, allowing the storage engine to skip rows when a filtering expression evaluates to false. These two properties—pay for what you read and filter pushdown—are ones we can exploit when taking samples.

Say we have a Population table with billions of rows and around 100 columns. How can we efficiently take a weighted sample of 1000 rows?

We could use the basic sampling formulation, as discussed earlier:

```
SELECT *
FROM Population
WHERE weight > 0
ORDER BY -LN(1.0 - RANDOM()) / weight
LIMIT 1000  -- Sample size.
```

But think about what the query engine must do to execute this query. It must read and decode all 100 columns for all of those billions of rows so that it may pass those rows into the sort/limit logic (typically implemented as a `TOP_N` operation) to determine which rows to keep for the sample. Even though the sample will keep only 0.00001% of those rows, you'll have to pay to read the entire Population table!

A much faster approach is to only read the columns we need to determine *which* rows are in the sample. Say our table has a primary key `pk` that uniquely identifies each row. The following variant on our sampling formulation returns only the primary keys needed to identify the rows in the sample:

```
SELECT pk
FROM Population
WHERE weight > 0
ORDER BY -LN(1.0 - RANDOM()) / weight
LIMIT 1000  -- Sample size.
```

This variant only forces the query engine to read two columns: `pk` and `weight`: Yes, it still must read those two columns for the billions of rows in the table, but those columns contain small values and can be scanned quickly. After all, that's what column-oriented stores are designed to do well. The point is that we're not paying to read about 100 additional columns whose values we're just going to throw away 99.99999% of the time.

One we have identified the rows in our sample, we can run a second query to pull in the full set of wanted columns for just those rows.

Adding determinism

Our sampling algorithm depends on randomization. If we run our algorithm twice with the same inputs, we'll get different results each time. Often, that nondeterminism is exactly what we want.

But sometimes it isn't. Sometimes, it's useful to be able to *control* the dice rolls that the algorithm depends on. For example, sometimes it's useful to be able to repeat a sample. Or *almost* repeat a sample.

To allow us to control the nature of the randomization used when we take

samples, we must replace calls to `RANDOM` with a deterministic pseudorandom function. One common approach is to hash a primary key and then map the hashed value to a number in the range $[0, 1)$. The following DuckDB macro `pseudorandom_uniform` will do exactly that:

```
-- Returns a pseudorandom fp64 number in the range [0, 1). The number
-- is determined by the given `key`, `seed` string, and integer `index`.
CREATE MACRO pseudorandom_uniform(key, seed, index)
AS (
  (HASH(key || seed || index) >> 11) * POW(2.0, -53)
);
```

We can vary the `seed` and `index` parameters to generate independent random values for the same `key`. For example, if I fix the `seed` to "demo-seed-20240601" and generate random numbers for the `key` "key123" over the `index` values $1, 2, \ldots, 10$, I get 10 fresh random numbers:

```
SELECT
  pseudorandom_uniform('key123', 'demo-seed-20240601', i) AS u_key123
FROM RANGE(1, 11) AS t(i);
```

Table 1: Ten random numbers for the key "key123" and seed "demo-seed-20240601".

i	*u_key123*
1	0.9592606495318252
2	0.6309411348395693
3	0.5673207749533353
4	0.11182926321927167
5	0.3375806483238627
6	0.12881607107157678
7	0.6993372364353198
8	0.94031652266991
9	0.17893798791559323
10	0.6903126337753016

To take deterministic samples, we just replace calls to `RANDOM()` with calls to our function `pseudorandom_uniform()`.

Now that we can take deterministic samples, we can do even more useful things! For example, we can take samples *with* replacement.

Sampling with replacement

Earlier, we proved that the A-ES algorithm allows us to take a sample without replacement as a series of successive draws, each draw removing an item from the population, and each draw fair with respect to the population that remains at

the time of the draw. But what if we wanted to take a sample *with replacement*? A sample with replacement requires us to return each item to the population as it is selected so that every selection is fair with respect to the *original* population, and individual items may be selected more than once.

Can we efficiently implement sampling with replacement in SQL? Yes! But it's a little trickier. (I haven't found this algorithm published anywhere; please let me know if you have. It took me some effort to create, but I wouldn't be surprised if it's already known.)

Think back to our correctness proof for the A-ES algorithm. For each row i having a weight w_i, the algorithm imagined a corresponding Poisson process with rate $\lambda_i = w_i$ and represented the row by the very next arrival from that process. That arrival would occur at time $k_i = -\ln(u_i)/w_i$, where u_i is a uniformly distributed random number in the range $(0, 1]$. Then the algorithm sorted all rows by their k_i values and took the first m arrivals as the sample.

With one minor tweak to this algorithm, we can take a sample with replacement. That tweak is to consider not just the *very next* arrival from each row's Poisson process but *all* arrivals. Let $k_{i,j}$ denote the jth arrival from row i's process. Since we know that in a Poisson process the times between successive arrivals are exponentially distributed random variables, we can take the running sum over interarrival times to give us the needed arrival times. That is, $k_{i,j} = \sum_{r=1}^{j} -\ln(u_{i,r})/w_i$, where $u_{i,r}$ represents the rth uniformly distributed random variable for row i.

One minor problem with this tweaked algorithm is that a Poisson process generates a theoretically infinite series of arrivals. Creating an infinite series for each row and then sorting the arrivals from all of these series is intractable.

Fortunately, we can avoid this problem! Think about how the A-ES algorithm for taking a sample *without* replacement relates to our proposed intractable algorithm for taking a sample *with* replacement. We could describe the *without* algorithm in terms of the *with* algorithm like so: Prepare to take a sample *with* replacement, but then ignore all arrivals $k_{i,j}$ for $j > 1$; the remaining arrivals must be of the form $k_{i,1}$, where i indicates the corresponding row. Then take the first m of these remaining arrivals as your sample, as before.

Now think about going the other way, from having a *without*-replacement sample and needing to construct a corresponding *with*-replacement sample. Let S be the set of rows sampled *without* replacement. We know these rows were represented by a corresponding set of arrival times $k_{i,1}$ for i in S. We also know that, had the sample been taken *with* replacement, the race would have included some additional arrival times $k_{i,j}$ for $j > 1$ that could have displaced some of the winning rows in S. But, crucially, we also know that if S_* represents the set of rows in the corresponding sample *with* replacement, then S_* must be contained within S. This claim follows from the fact that if any arrival $k_{i,j}$ for $j > 1$ does displace a row among the first m arrivals, the $j > 1$ requirement implies that

the displacing row is a duplicate of some row i that arrived earlier in the sample at time $k_{i,1}$; thus, displacement cannot introduce a new row from outside of S.

Therefore, if we have a sample *without* replacement, we can construct a sample *with* replacement from its rows. *We can ignore all other rows in the population.* This makes the problem much more approachable.

So now we can see an algorithm taking shape for taking a sample *with* replacement of size m:

1. First, take an m-sized sample S *without* replacement using the efficient A-ES algorithm.
2. For each sampled row i in S, generate m arrivals $k_{i,j}$ for $j = 1, 2, \ldots, m$ using the same pseudorandom universe that was used to generate S.
3. Take the first m arrivals as the sample.

You may have noticed that step 2 of this algorithm requires us to create m arrivals for each of the m rows in S. This step thus requires $O(m^2)$ time. When m is large, this time can be prohibitive.

Fortunately, we can use probability theory to reduce this cost to $O(m)$. The idea is that if row i is expected to occur in the sample n_i times, it is very unlikely to occur more than $c \cdot n_i$ times when $c \geq 2$. So we don't need to generate a full m arrivals for each row i in S; we can get away with generating $m_i = \lceil c \cdot n_i \rceil$ arrivals instead, for a suitably large c to assuage our personal level of paranoia.

Here's a sample implementation as a DuckDB macro:

```
-- Takes a weighted sample with replacement from a table.
--
-- Args:
--  population_table: The table to sample from. It must have a `pk` column
--    of unique primary keys and a `weight` column of non-negative weights.
--  seed: A string that determines the pseudorandom universe in which the
--    sample is taken. Samples taken with distinct seeds are independent.
--    If you wish to repeat a sample, reuse the sample's seed.
--  sample_size: The number of rows to include in the sample. This value
--    may be larger than the number of rows in the `population_table`.
--
-- Returns a sample of rows from the `population_table`.
CREATE MACRO sample_with_replacement(population_table, seed, sample_size)
AS TABLE (
  WITH
    -- First, take a sample *without* replacement of the wanted size.
    SampleWithoutReplacement AS (
      SELECT *
      FROM query_table(population_table::varchar)
      WHERE weight > 0
      ORDER BY -LN(pseudorandom_uniform(pk, seed, 1)) / weight
```

```
    LIMIT sample_size
  ),
  -- Compute the total weight over the sample.
  SampleWithoutReplacementTotals AS (
    SELECT SUM(weight) AS weight
    FROM SampleWithoutReplacement
  ),
  -- Generate a series of arrivals for each row in the sample.
  SampleWithReplacementArrivals AS (
    SELECT
      S.*,
      SUM(-LN(pseudorandom_uniform(pk, seed, trial_index)) / S.weight)
        OVER (PARTITION BY pk ORDER BY trial_index)
        AS rws_sort_key
    FROM SampleWithoutReplacement AS S
    CROSS JOIN SampleWithoutReplacementTotals AS T
    CROSS JOIN
      UNNEST(
        RANGE(1, CAST(2.0 * sample_size * S.weight / T.weight + 2 AS INT)))
      AS I(trial_index)
  )
-- Form the sample *with* replacement from the first `sample_size` arrivals.
SELECT * EXCLUDE (rws_sort_key)
FROM SampleWithReplacementArrivals
ORDER BY rws_sort_key
LIMIT sample_size
);
```

Example of sampling with replacement

As an example of when we might want to sample with replacement instead of without, consider the following population table `ThreeToOne` that represents the possible outcomes of tossing a biased coin:

Table 2: `ThreeToOne` population table.

pk	*weight*
heads	3
tails	1

For this biased coin, "heads" is 3 times as likely as "tails." We can simulate flipping this coin 10 times by sampling 10 rows from the `ThreeToOne` population table with replacement:

```
SELECT pk
FROM sample_with_replacement(ThreeToOne, 'test-seed-20240601', 10);
```

Table 3: Results of drawing a 10-row sample with replacement from `ThreeToOne`.

pk
heads
heads
heads
tails
tails
heads
heads
heads
tails
heads

In this sample, we got 7 heads and 3 tails. On average, we would expect about 7.5 heads in each sample of size 10, so our observed sample is close to our expectations.

But maybe we just got lucky. As a stronger test of our SQL sampling logic, let's take 10,000 samples of size 40 and look at the count of heads across all of the samples. We would expect this count to have a Binomial($size = 40$, $p = 3/4$) distribution. To compare our observed results to the expected distribution, I'll compute the empirical distribution of the results and plot that over the expected distribution. As you can see in the figure below, the observed distribution closely matches the expected distribution.

Conclusion

Sampling is a powerful tool. And with the SQL logic we've just discussed, you can take fast, easy samples from virtually any dataset, no matter how large. And you can take those samples with or without replacement.

What makes it all work is a clever connection to the theory of Poisson processes. Those processes are memoryless and mergeable, and their arrivals win races in proportion to their rates. These properties are exactly what we need to run races that let us take samples.

Beyond what we've discussed in this article, there are further ways we can exploit these properties. For example, as a performance optimization, we can predict the arrival time t of the final row in a sample. Then we can augment our SQL sampling logic with a pushdown filter that eliminates population rows with arrival times greater than $c \cdot t$ for some constant c. This filtering happens before `ORDER/LIMIT` processing and can greatly speed queries by eliminating more than 99.99% of rows early on, before they are even fully read on systems that support "late materialization."

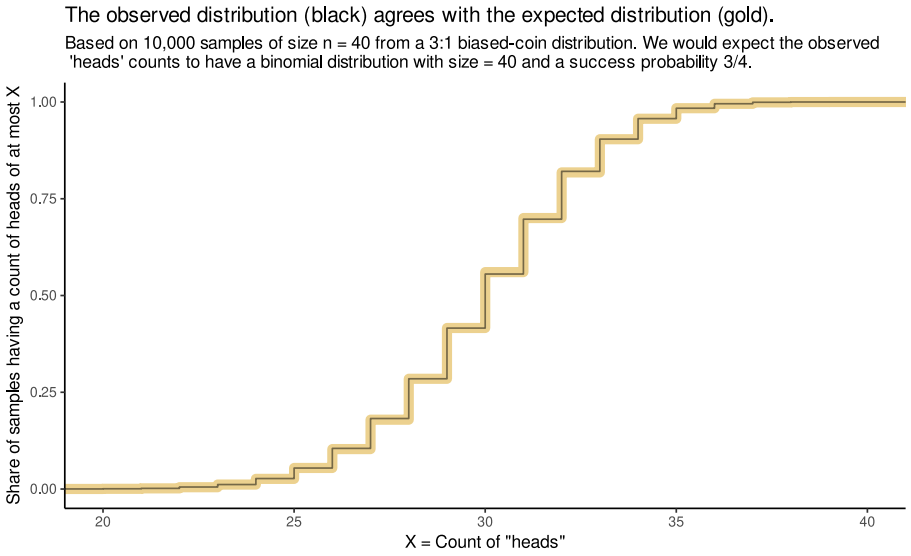

Figure 1: When we take 10,000 independent samples of size $n = 40$ from a 3:1 biased-coin distribution, we find that the count of "heads" over the samples agrees with the expected Binomial($size = 40$, $p = 3/4$) distribution.

But this article is already too long, so I'll stop here for now.

References

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