Communication-Optimal Parallel Reservoir Sampling

Christian Winter[1] Moritz Sichert[2] Altan Birler[3] Thomas Neumann[4] Alfons Kemper[5]

Abstract: When evaluating complex analytical queries on high-velocity data streams, many systems cannot run those queries on all elements of a stream. Sampling is a widely used method to reduce the system load by replacing the input with a representative yet manageable subset. For unbounded data, reservoir sampling generates a fixed-size uniform sample independent of the input cardinality. However, the collection of reservoir samples itself can already be a bottleneck for high-velocity data.

In this paper, we introduce a technique that allows fully parallelizing reservoir sampling for many-core architectures. Our approach relies on the efficient combination of thread-local samples taken over chunks of the input without necessitating communication during the sampling phase and with minimal communication when merging. We show how our efficient merge guarantees uniform random samples while allowing data to be distributed over worker threads arbitrarily. Our analysis of this approach within the Umbra database system demonstrates linear scaling along the available threads and the ability to sustain high-velocity workloads.

Keywords: Reservoir Sampling; Parallel Sampling; Stream Processing

1 Introduction

With the widespread deployment of cheap connected sensors and the increased use of off-premise systems, there is an ever-growing need to analyze the log and data streams generated by these sensors and systems remotely. Due to this data's high volume and high velocity, analyzing all generated entries and values is often infeasible. Therefore, many analyses are performed on a reduced version of the data. Some applications rely on aggregates over windows or subsets of the data, e.g., monitoring average temperature readings in a given area. Others, e.g., for analyzing log streams, apply highly selective filters to reduce the input cardinality, showing only relevant error messages and surrounding entries. However, for some applications, representative and unfiltered data points are desirable. Sampling is employed to reduce the data stream to a manageable size for analytics systems.

By drawing a uniform sample from a stream of data points, each point of the stream has an equal probability of being part of the resulting sample, and thus the result is representative of the entire stream. Sampling is utilized in a wide range of applications, e.g., for workload

[1] Technische Universität München winterch@in.tum.de
[2] Technische Universität München moritz.sichert@tum.de
[3] Technische Universität München altan.birler@tum.de
[4] Technische Universität München neumann@in.tum.de
[5] Technische Universität München kemper@in.tum.de

doi:10.18420/BTW2023-27

statistics [BRN20, LN90], machine learning [Sc22, Sc21, Sc15], and big data [Ma20]. While some sampling algorithms take a variable-sized subset of the input, e.g., selecting $x\%$ of the arriving tuples at random, their resulting sample size can still be infeasible for analyzing high-velocity unbounded data streams. Therefore, we will focus on those algorithms that produce a fixed-sized sample independent of the input size, i.e., reservoir sampling.

While reservoir sampling produces a uniform random sample in a single pass over the input in $O(n(1 + \log(N/n)))$ [Li94], the sheer volume of data can lead to bottlenecks during the sampling phase. In the past, the growth in data could be compensated by the performance improvements of new hardware. However, with Moore's law coming to its end, a single core can often no longer keep up. Therefore, many solutions for stream processing and analysis, such as dedicated stream processing engines [Za16, Ze20, Ca15] and in-database stream-processing approaches [Wi20], have focused on processing incoming streams in a parallel and distributed manner.

In this paper, we describe a mechanism allowing fully parallel reservoir sampling without communication between sampling threads. By keeping thread-local samples over chunks of the input, we can construct a complete sample of size n in parallel from p worker samples using only $2p + n$ messages in an efficient k-way merge. The low message overhead is especially beneficial in distributed environments, where communication takes place using comparably slow and expensive network connections. Further, we describe an $O(p + n\log(p))$ merge strategy optimized for small sample sizes and show that it can guarantee uniform random samples, independent of how input elements are assigned to workers. By constraining all communication to our merge stage, our approach can scale almost linearly in many-core machines. Implementing our approach within the code-generating Umbra database system [NF20], we demonstrate that our approach is applicable in real-world systems. Using this implementation, we evaluate our novel merge strategy against a merge strategy based on a hypergeometric distribution in many-core applications for different sample sizes, showing that our proposed merge is beneficial for small sample sizes. Overall, our communication optimal reservoir sampling can scale linearly along the number of available workers and sustain a sampling throughput of more than 300 million tuples per second per thread, independent of the distribution used in the merge.

The remainder of the paper is structured as follows: We introduce relevant concepts and algorithms to our approach in Sect. 2 and discuss related work. In Sect. 3, we present the design and implementation of both of our contributions and prove the correctness of our novel merge strategy. To demonstrate our approach's applicability and performance, we evaluate its scalability and throughput in Sect. 4 before concluding in Sect. 5.

2 Background and Related Work

Approximate results are often deemed acceptable to gain instant query response times for analytics over large volumes of data. Simple random samples of fixed size are a reliable tool to reduce the costs of computing approximate statistics [SA22]. We will argue why this tool is particularly interesting and discuss related work in constructing such samples.

When using a random sample, a small random subset of the data is picked and processed instead of the entire data, significantly improving query response times. Other statistical tools, such as histograms [Co12] and distinct count sketches [FN19], are useful for approximate statistics. However, they are inflexible as the filters that they can evaluate are limited. A histogram can only evaluate simple predicates, such as one (or few) dimensional ranges. Samples, on the other hand, can evaluate arbitrary predicates, as they are smaller representatives of the entire data set. In the absence of complex filters, histograms can provide useful upper bounds, while samples can only provide probabilistic estimates. However, with large enough samples, the variance of the estimates a sample provides is relatively low and can be relied upon. Additionally, with complex filters, a histogram's upper bounds can be too high to be useful, as it can only consider simple range predicates.

There are many ways to pick a random sample. For computing unbiased statistics, simple random samples without replacement are a good fit as every subset of the data is selected with equal probability. Tillé [Ti11] describes the theoretical background of simple random sampling and computation of statistics from a simple random sample. Ting [Ti21] provides efficient implementations for a range of algorithms for sampling without replacement. We focus on samples of fixed size. In contrast to Bernoulli sampling, where every tuple is picked with independent probability θ, fixed-size samples do not grow alongside the input data size. One might assume a larger sample would be a better fit for a larger data set. This is true for predicates whose selectivity decrease with increasing data set size, such as filters for fixed timespan on a data set that grows over time. However, for predicates of constant selectivity, the size of the data set has little to no effect on the quality of the sample. We will try to build a simplified intuition as to why and refer the reader to the detailed theoretical analysis by Moerkotte and Hertzschuch [MH20] for further details.

Given a sample of size n, a data set of size $N = \rho n$, and a predicate of constant selectivity σ, we want to estimate the number of matches $K = \sigma N = \sigma n \rho$ where ρ is the ratio of the size of the data set to the size of the sample. This task is common in cardinality estimation within database systems [He21]. As a strategy, we evaluate the predicate on the sample and count the number of matches k, which is a random variable from the hypergeometric distribution $HG(\rho n, \sigma \rho n, n)$. We use the simple estimator $\hat{K} = k\rho$ as our estimate of $K = \sigma n \rho$. As a simple cost metric, we define the expected relative mean squared error of our estimate as:

$$\text{rMSE} = \mathrm{E}\left[\left(\frac{\hat{K} - K}{K}\right)^2\right] = \mathrm{E}\left[\left(\frac{k\rho - \sigma n \rho}{\sigma n \rho}\right)^2\right] = \frac{1}{\sigma n}\,\mathrm{Var}\,[k] = \frac{1-\sigma}{\sigma}\frac{\rho - 1}{\rho n - 1} \approx \frac{1-\sigma}{\sigma}\frac{1}{n} \quad (1)$$

Assuming ρ is relatively large, it disappears from our error estimate, implying that the relative size of the sample has little to no effect on the accuracy of this estimate. On the contrary, the predicate's selectivity and the sample's absolute size directly influence the error. An intuitive explanation of this result is that we can assume that the data set from which we are sampling is itself a random sample drawn from an infinite distribution. Resampling from this *intermediate sample* simply means that we construct a smaller sample of the original data set. The size of the intermediate sample has only a small effect on the final sample's contents. So, given requirements on the error and information on the selectivity of predicates, it makes sense to pick a *fixed* sample size rather than having sample sizes adapt to the data set size as adapting the size of a sample is a costly operation requiring that the sample be rebuilt.

The optimal way to compute a simple random sample depends on various factors. Our proposed approach focuses on concurrently sampling from many parallel data streams in environments where communication costs are high (we are optimal in the number of communications), and memory usage is not a limiting factor. Due to these assumptions, our approach is flexible and an excellent fit for distributed environments. There are simpler algorithms for when the size of the data set is known beforehand: The draw-by-draw procedure iterates over the sample and picks tuples one by one with potentially high costs from random accesses into the data [Ti11]. The sequential selection-rejection method described by Fan et al. [FMR62] instead iterates over the data to produce the sample. Vitter [Vi84, Vi87] further improves the selection rejection method by computing skip lengths; rather than iterating over the data one tuple at a time, one can probabilistically generate the number of tuples to skip before the next tuple is selected, significantly reducing the number of random number generations. This approach is parallelized by Sanders et al. [Sa16] by distributing the task of random sampling among different workers. Chickering et al. [CRM07] describe merging parallel reservoir samples using a hypergeometric distribution, which we utilize for larger sample sizes, and offer proof that the resulting sample is still uniformly random. However, they send all local samples to a centralized coordinator for the merge, leading to communication overhead.

To maintain a sample of size n in a single pass over a data stream, a set of more than n values, the so-called reservoir, needs to be processed to guarantee a simple random sample at any point during processing. All procedures maintaining a simple random sample in a single pass over a data stream are variants of the reservoir sampling algorithm defined by Vitter [Vi85]. Reservoir sampling iterates over input tuples and selects them for the sample with a probability proportional to the number of tuples seen so far. Vitter [Vi85] improves on this by probabilistically generating skips, a contiguous amount of tuples not contained in the sample, thus reducing the costs for generating random numbers from $O(N)$ to $O(n(1+\log(N/n)))$. Li [Li94] improves on the approach by Vitter by proposing a simpler distribution to generate skips. These sequential approaches only support sampling data from a single stream. Hübschle-Schneider and Sanders [HS20] also parallelize reservoir sampling by independently collecting multiple reservoirs and merging them afterward. However, their approach needs to maintain a distributed priority queue, which incurs additional communication costs but potentially reduces the sizes of their independent samples. For situations where memory is scarce, Tirthapura and Woodruff [TW11] maintain a single sample at a central coordinator. Their approach has optimal communication complexity for the centralized sample setting. Birler et al. [BRN20] reduce the communication costs of Tirthapura and Woodruff's approach by accepting temporary imperfections in the central sample that are eventually corrected. Our approach, in contrast, is communication optimal among all possible distributed reservoir sampling algorithms. For this property, we accept a slight increase in per-stream memory consumption, which is acceptable in analytical workloads as our local samples are fixed-sized and small compared to all the other data the streams need to maintain.

The performance characteristics of the various approaches can be analyzed by looking at communication costs, total processing costs, and total memory use. These three metrics are influenced by the utilized sampling approach, the sample size, the data size, and the number of workers. Shared-sample-based approaches [BRN20, TW11] benefit from low memory use and total processing costs. Thus, they are well applicable to low communication cost environments, such as a single machine with a single CPU socket. However, in settings with high communication costs, such as manycore machines with multiple sockets or distributed networks, distributed sampling approaches [HS20] are beneficial as they sacrifice memory and some computation to cut down on inter-worker communication. These trade-offs are necessary as Tirthapura and Woodruff [TW11] prove that communication costs may not be optimal when only one shared sample exists. In our approach, where we focus on minimizing communication costs, we must maintain a full sample per worker. Otherwise, we must incur additional communication or provide weaker guarantees, such as a probability of failure [Sa16].

3 Approach

Having outlined the background in sampling, we can describe our merge-based parallel reservoir sampling technique and the novel post-sampling merge strategy optimized for small sample sizes. Our approach aims to draw a sample of n tuples uniformly random in parallel from an input of N tuples using p workers. Each worker i draws a thread-local reservoir sample of size n out of the N_i tuples assigned to it, denoted as $s_{i,1} \dots s_{i,n}$, using algorithm L [Li94]. This sample is merged into a global sample on demand. We assume no prior knowledge about the workload, only the reservoir size n has to be known, and we materialize only tuples selected for a local reservoir. Throughout this section, we will assume a work-stealing, morsel-based [Le14] distribution of input chunks to workers, as this is the parallelization strategy of our system Umbra. However, our approach is applicable to any input distribution strategy. Our contributions are twofold. First, we detail the communication-optimal merge process, which improves upon prior work independent of the reservoir size and merge strategy used. Subsequently, we discuss our novel merge strategy optimized for small reservoir sizes.

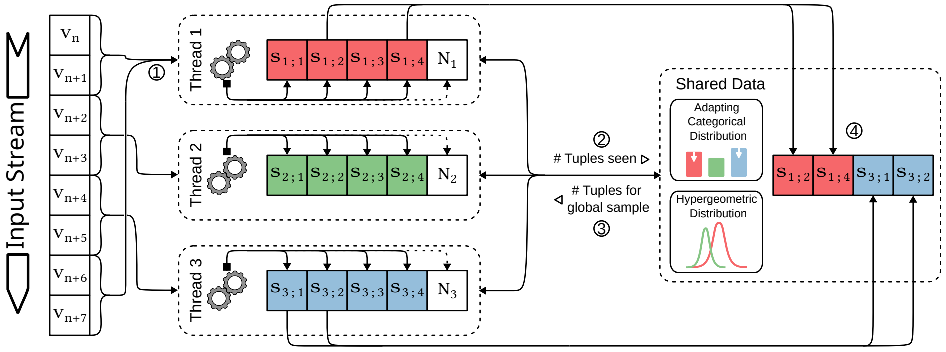
Input Stream
v_n
v_{n+1}
v_{n+2}
v_{n+3}
v_{n+4}
v_{n+5}
v_{n+6}
v_{n+7}
①
Thread 1
Thread 2
Thread 3
$s_{1;1}$ $s_{1;2}$ $s_{1;3}$ $s_{1;4}$ N_1
$s_{2;1}$ $s_{2;2}$ $s_{2;3}$ $s_{2;4}$ N_2
$s_{3;1}$ $s_{3;2}$ $s_{3;3}$ $s_{3;4}$ N_3
②
# Tuples seen
# Tuples for global sample
③
Shared Data
Adapting Categorical Distribution
Hypergeometric Distribution
④
$s_{1;2}$ $s_{1;4}$ $s_{3;1}$ $s_{3;2}$

Fig. 1: Overview of the sampling process with three workers consisting of ① thread-local sampling, ② the transfer of local cardinalities, ③ determining thread shares in the global sample, and ④ the transfer of the resulting sample tuples.

3.1 Communication-Optimal Process

Our approach consists of two phases, the sampling and the merge phase, as shown in Fig. 1. In the sampling phase ①, all threads create local reservoirs of size n over chunks of the input. While sampling, workers fetch arriving chunks independently from one another, ensuring that each chunk is assigned to exactly one worker. Reservoir sampling guarantees that all local samples are uniformly random at any point, with each tuple having a probability of $\frac{n}{N_i}$ to be contained in the corresponding thread-local sample. To generate the whole sample, each worker first reports the cardinality of all chunks it processed, N_i, to a coordinator ②.

This coordinator can be an external worker or one of the sampling workers. First, the coordinator determines the share in tuples that each thread-local sample has in the global sample using either the hypergeometric distribution, or the proposed merge strategy outlined in detail below. Then, the coordinator notifies each thread of the number of tuples it has to choose for the global sample ③, which in turn selects the desired amount uniformly at random from their sample and reports it to the global sample ④. In contrast to prior work [CRM07] sharing all local samples with the coordinator in pn messages, our approach needs at most $2p + n$ messages: 2 per worker to communicate the local cardinality and the number of tuples to contribute to the global sample, and n to send the selected tuples.

3.2 Merge Strategy for Small Reservoirs

Conceptually, our approach for small reservoirs relies on iteratively evaluating and updating a categorical distribution over all threads for each position of the final sample to determine which thread to select for this reservoir slot. In contrast to strategies based on the hypergeometric distribution, the categorical distribution has to be evaluated per sample slot and

Algorithm 1 Calculating per-thread share of global sample

```
 1: function CALCULATETHREADSHARE(localCardinalities[])
 2:     $globalCardinality \leftarrow \text{sum}(localCardinalities)$
 3:     $fenwickTree \leftarrow \text{FenwickTree::build}(localCardinalities)$
 4:     $samplesPerThread \leftarrow []$
 5:     $slot \leftarrow 0$
 6:     while $slot < sampleSize$ and $globalCardinality > 0$ do
 7:         $selectedTuple \leftarrow \text{pickRandom}(0,\ globalCardinality - 1))$
 8:         $selectedThread \leftarrow fenwickTree.\text{rank}(selectedTuple)$
 9:         $samplesPerThread[selectedThread] \leftarrow samplesPerThread[selectedThread] + 1$
10:         $fenwickTree.\text{add}(selectedTuple,\ -1)$
11:         $globalCardinality \leftarrow globalCardinality - 1$
12:         $slot \leftarrow slot + 1$
13:     return $samplesPerThread$
```

not per thread. While this is too costly for large sample sizes, it avoids the computationally expensive hypergeometric distribution. Each thread i is selected with a probability of $\frac{N_i}{N}$ for the first slot. As we sample without replacement, we decrease the cardinality of the selected thread N_i and the global cardinality N by 1 for the next draw. All threads j with $j \neq i$ have the probability $\frac{N_j}{N-1}$ of being selected for the next reservoir slot, the selected thread has a probability of $\frac{N_i-1}{N-1}$. We repeat this until we have selected the source for all n spaces in the sample, decreasing N and N_i for the selected i at every draw. Note that while conceptually drawing slot by slot, we do not care about the order of the sample or the actual slot to select from. This allows us to only track the number of tuples per thread.

Algorithm 1 shows our implementation. In the first step, we calculate the global cardinality from the thread-local information and build a Fenwick tree over the local cardinalities (Line 3). Fenwick trees, as described in [Fe94], allow efficient operations over prefix sums while requiring only linear space. Building a Fenwick tree is possible in linear time, whereas rank and update operations have logarithmic runtime. Following the setup, we can pick the shares for each thread. For this, we first generate a random number r in the range of 0 to N (Line 7). For this, we first generate a random integer $r \in [0, N)$ (Line 7). From this value, we pick the corresponding thread by using the prefix-sums stored in the Fenwick tree. For $r \in [0, N_1)$, we pick thread 1, for $r \in [N_1, N_2)$, thread 2, and so forth. Mapping r to a thread this way is possible in $O(\log(p))$ using the rank operation of the Fenwick tree (Line 8). Then, we update the cardinality of the selected thread and the global cardinality (Lines 10 and 11) for the next draw from the updated categorical distribution. We repeat the draw and distribution update until we have either selected every tuple or filled every slot in the final sample.

Our algorithm does not require floating point arithmetic, allowing a fast and exact evaluation. The Fenwick tree construction dominates the setup step with a runtime of $O(p)$. The loop of Line 6 is evaluated n times, requiring $O(\log(p))$ to update the Fenwick tree, resulting in an overall runtime complexity of $O(p + n\log(p))$.

3.3 Proof

To show that the merge strategy outlined above does not change the resulting samples' probability, we show that it selects tuples from local reservoirs equal to the hypergeometric distribution. For this, we will use the probability mass function $P(X = k)$ where X denotes the number of tuples selected from thread i. For our proof, we use the fact that the positions for which i is selected are irrelevant. Consider first the case where all k selections of i happen in the first k draws, followed by $n - k$ draws of $j \neq i$. This results in the probability

$$P(\text{first } k \text{ from } i) = \frac{N_i}{N} \times \frac{N_i - 1}{N - 1} \times \cdots \times \frac{N_i - k + 1}{N - k + 1} \times \frac{N - N_i}{N - k} \times \frac{N - N_i - 1}{N - k - 1} \times \cdots \times \frac{N - N_i - n + k + 1}{N - n + 1} \tag{2}$$

$$= \frac{N_i \times (N_i - 1) \times \cdots \times (N_i - k + 1) \times (N - N_i) \times (N - N_i - 1) \times \cdots \times (N - N_i - n + k + 1)}{N \times (N - 1) \times \cdots \times (N - n + 1)}. \tag{3}$$

It is clear that, through the commutative property of the product, the draws for i, N_i to $N_i - k + 1$, can be moved to any of the draws N to $N - n + 1$ without changing the resulting probability. For the full $P(X = k)$, we additionally need to select the k positions for our i draws, resulting in the probability

$$P(X = k) = \frac{N_i \times (N_i - 1) \times \cdots \times (N_i - k + 1) \times (N - N_i) \times (N - N_i - 1) \times \cdots \times (N - N_i - n + k + 1)}{N \times (N - 1) \times \cdots \times (N - n + 1)} \times \binom{n}{k}. \tag{4}$$

Using $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ and $x \times (x - 1) \times \cdots \times (x - m + 1) = \frac{x!}{(x-m)!}$ we get

$$P(X = k) = \frac{(N_i \times (N_i - 1) \times \cdots \times (N_i - k + 1)) \times ((N - N_i) \times (N - N_i - 1) \times \cdots \times (N - N_i - n + k + 1))}{N \times (N - 1) \times \cdots \times (N - n + 1)} \times \binom{n}{k} \tag{5}$$

$$= \frac{N_i!}{(N_i - k)!} \times \frac{(N - N_i) \times (N - N_i - 1) \times \cdots \times (N - N_i - n + k + 1)}{N \times (N - 1) \times \cdots \times (N - n + 1)} \times \binom{n}{k} \tag{6}$$

$$= \frac{N_i!}{(N_i - k)!} \times \frac{(N - N_i)!}{(N - N_i - n + k)!} \times \frac{1}{N \times (N - 1) \times \cdots \times (N - n + 1)} \times \binom{n}{k} \tag{7}$$

$$= \frac{N_i!}{(N_i - k)!} \times \frac{(N - N_i)!}{(N - N_i - n + k)!} \times \frac{(N - n)!}{N!} \times \frac{n!}{k!(n - k)!} \tag{8}$$

$$= \frac{N_i!}{k!(N_i - k)!} \times \frac{(N - N_i)!}{(N - N_i - (n - k))! \times (n - k)!} \times \frac{(N - n)!n!}{N!} \tag{9}$$

$$= \frac{\binom{N_i}{k} \times \binom{N - N_i}{n - k}}{\binom{N}{n}} \tag{10}$$

which is the probability mass function of the hypergeometric distribution.

4 Evaluation

Having outlined the implementation of our fully parallel communication-optimal reservoir sampling approach, we demonstrate its performance, focussing on scalability and the impact of our proposed merge strategy for small sample sizes. The experiments are conducted using an implementation within Umbra on a server equipped with 2 AMD EPYC™ 7713 CPUs with 64 cores each and 1 TiB of main memory. To reduce the impact of IO bottlenecks, we generate all data using the PostgreSQL-derived `generate_series`[6] command. All results reported are based on averages over 9 runs, each sampling $N = 100$ billion records.

[6] `https://www.postgresql.org/docs/current/functions-srf.html`

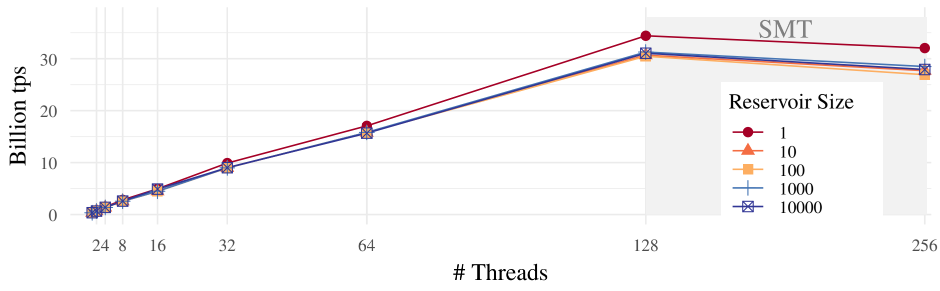

Fig. 2: Total sampling throughput with an increasing number of threads: Our communication-optimal sampling approach scales nearly linearly to more than 30 billion tuples per second.

4.1 Scalability and Performance

In the first experiment, we investigate the scalability of our approach. As we require no communication between threads during the sampling phase, we expect the sampling phase to scale perfectly along the thread count. Additionally, the runtime of the merge phase is independent of the input size, so we expect it to amortize for large data sets. We measure the total throughput of our implementation with different reservoir sizes and an increasing number of threads and report the results in Fig. 2. As expected, the throughput of processed tuples scales nearly linearly with the number of threads. For a reservoir size of 1, our implementation can process up to 35 billion tuples per second when using all 128 physical cores. The experiment samples 8 B integers, so in total, our system processes up to 280 GB of data per second.

Our approach requires each thread to collect a full-size local sample. For this reason, the memory usage of sampling increases linearly with the number of threads. Therefore, we expect higher sampling overhead and lower throughput for increasing sample sizes. However, the results show that the overhead is manageable even for larger sample sizes. For example, even when collecting a sample of size 10000, our implementation can still process over 30 billion tuples per second.

4.2 Merge Strategy Comparison

Our approach requires communication between threads only in the merge phase. We want to show that our merge strategy, which employs a categorical distribution, can be more efficient than a hypergeometric distribution while producing equivalent results. To evaluate our strategy, we compare the runtime of the merge phase for both distributions. Fig. 3 shows the relative speedup of our merge strategy with varying numbers of threads and sample sizes. Note that the merge phase itself always runs single-threaded on a coordinator node.

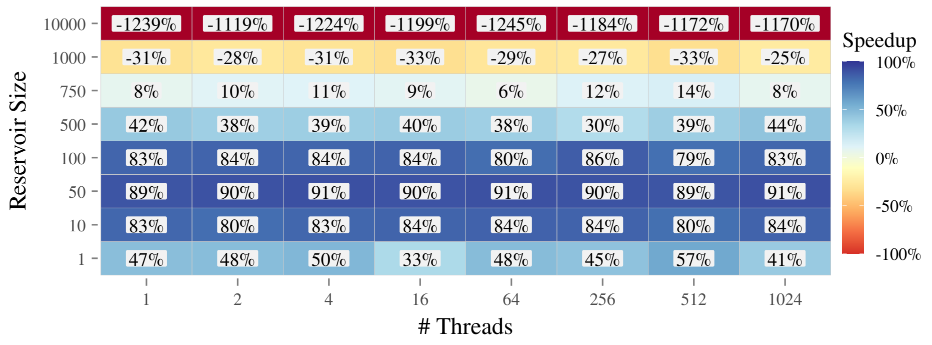

Fig. 3: Speedup of our merge strategy based on a categorical distribution over using a hypergeometric distribution: For sample sizes below 1000, our approach achieves up to 91% speedup independent of the number of threads.

The number of threads in the figure refers to the number of locally collected samples to be merged.

For sample sizes up to 750, our approach consistently outperforms using the hypergeometric-distribution-based merge. The main reason for the efficiency of our merge strategy is that it uses no floating-point operations. However, we need to perform two operations on the Fenwick tree for every element in the sample, while the merge phase using a hypergeometric distribution only generates one value from the distribution for every thread, independent of the sample size. Therefore, our merge strategy is inefficient for larger sample sizes. The experiment further shows that our approach's speedup is generally independent of the number of threads: Our strategy consistently achieves similar speedup across all sample sizes, even for several hundred threads.

5 Conclusion

In this paper, we introduced a new communication-optimal parallel reservoir sampling technique, requiring only $2p + n$ messages for environments with p workers and a sample size of n. Our technique relies on an efficient merge of thread-local reservoir samples, each taken over arbitrarily distributed chunks of the input. In addition, we described a novel merge strategy optimized for small sample sizes and provide proof that this strategy is statistically equal to the hypergeometric distribution. With our implementation of communication-optimal parallel reservoir sampling in the Umbra database system, we evaluated its overall performance, and both merge strategies. Achieving more than 370 million samples per thread, we showed near-linear scale up to 128 workers and a clear advantage of our optimized merge for samples smaller than 1000 tuples.

Bibliography

[BRN20] Birler, Altan; Radke, Bernhard; Neumann, Thomas: Concurrent online sampling for all, for free. In: DaMoN. ACM, pp. 5:1–5:8, 2020.

[Ca15] Carbone, Paris; Katsifodimos, Asterios; Ewen, Stephan; Markl, Volker; Haridi, Seif; Tzoumas, Kostas: Apache Flink™: Stream and Batch Processing in a Single Engine. IEEE Data Eng. Bull., 38(4):28–38, 2015.

[Co12] Cormode, Graham; Garofalakis, Minos N.; Haas, Peter J.; Jermaine, Chris: Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches. Found. Trends Databases, 4(1-3):1–294, 2012.

[CRM07] Chickering, David M; Roy, Ashis K; Meek, Christopher A: , Distributed reservoir sampling for web applications, December 11 2007. US Patent 7,308,447.

[Fe94] Fenwick, Peter M.: A New Data Structure for Cumulative Frequency Tables. Softw. Pract. Exp., 24(3):327–336, 1994.

[FMR62] Fan, C. T.; Muller, Mervin E.; Rezucha, Ivan: Development of Sampling Plans by Using Sequential (Item by Item) Selection Techniques and Digital Computers. Journal of the American Statistical Association, 57(298):387–402, 1962.

[FN19] Freitag, Michael J.; Neumann, Thomas: Every Row Counts: Combining Sketches and Sampling for Accurate Group-By Result Estimates. In: CIDR. www.cidrdb.org, 2019.

[He21] Hertzschuch, Axel; Moerkotte, Guido; Lehner, Wolfgang; May, Norman; Wolf, Florian; Fricke, Lars: Small Selectivities Matter: Lifting the Burden of Empty Samples. In: SIGMOD Conference. ACM, pp. 697–709, 2021.

[HS20] Hübschle-Schneider, Lorenz; Sanders, Peter: Communication-Efficient Weighted Reservoir Sampling from Fully Distributed Data Streams. In: SPAA. ACM, pp. 543–545, 2020.

[Le14] Leis, Viktor; Boncz, Peter A.; Kemper, Alfons; Neumann, Thomas: Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In: SIGMOD Conference. ACM, pp. 743–754, 2014.

[Li94] Li, Kim-Hung: Reservoir-Sampling Algorithms of Time Complexity O(n(1 + log(N/n))). ACM Trans. Math. Softw., 20(4):481–493, 1994.

[LN90] Lipton, Richard J.; Naughton, Jeffrey F.: Query Size Estimation by Adaptive Sampling. In: PODS. ACM Press, pp. 40–46, 1990.

[Ma20] Mahmud, Mohammad Sultan; Huang, Joshua Zhexue; Salloum, Salman; Emara, Tamer Z.; Sadatdiynov, Kuanishbay: A survey of data partitioning and sampling methods to support big data analysis. Big Data Min. Anal., 3(2):85–101, 2020.

[MH20] Moerkotte, Guido; Hertzschuch, Axel: alpha to omega: the G(r)eek Alphabet of Sampling. In: CIDR. www.cidrdb.org, 2020.

[NF20] Neumann, Thomas; Freitag, Michael J.: Umbra: A Disk-Based System with In-Memory Performance. In: CIDR. www.cidrdb.org, 2020.

[Sa16] Sanders, Peter; Lamm, Sebastian; Hübschle-Schneider, Lorenz; Schrade, Emanuel; Dachsbacher, Carsten: Efficient Random Sampling - Parallel, Vectorized, Cache-Efficient, and Online. CoRR, abs/1610.05141, 2016.

[SA22] Sanca, Viktor; Ailamaki, Anastasia: Sampling-Based AQP in Modern Analytical Engines. In: DaMoN. ACM, pp. 4:1–4:8, 2022.

[Sc15] Schelter, Sebastian; Soto, Juan; Markl, Volker; Burdick, Douglas; Reinwald, Berthold; Evfimievski, Alexandre V.: Efficient sample generation for scalable meta learning. In: ICDE. IEEE Computer Society, pp. 1191–1202, 2015.

[Sc21] Schüle, Maximilian E.; Lang, Harald; Springer, Maximilian; Kemper, Alfons; Neumann, Thomas; Günnemann, Stephan: In-Database Machine Learning with SQL on GPUs. In: SSDBM. ACM, pp. 25–36, 2021.

[Sc22] Schüle, Maximilian E.; Lang, Harald; Springer, Maximilian; Kemper, Alfons; Neumann, Thomas; Günnemann, Stephan: Recursive SQL and GPU-support for in-database machine learning. Distributed Parallel Databases, 40(2):205–259, 2022.

[Ti11] Tillé, Yves: Sampling Algorithms. In: International Encyclopedia of Statistical Science, pp. 1273–1274. Springer, 2011.

[Ti21] Ting, Daniel: Simple, Optimal Algorithms for Random Sampling Without Replacement. CoRR, abs/2104.05091, 2021.

[TW11] Tirthapura, Srikanta; Woodruff, David P.: Optimal Random Sampling from Distributed Streams Revisited. In: DISC. volume 6950 of Lecture Notes in Computer Science. Springer, pp. 283–297, 2011.

[Vi84] Vitter, Jeffrey Scott: Faster Methods for Random Sampling. Commun. ACM, 27(7):703–718, 1984.

[Vi85] Vitter, Jeffrey Scott: Random Sampling with a Reservoir. ACM Trans. Math. Softw., 11(1):37–57, 1985.

[Vi87] Vitter, Jeffrey Scott: An efficient algorithm for sequential random sampling. ACM Trans. Math. Softw., 13(1):58–67, 1987.

[Wi20] Winter, Christian; Schmidt, Tobias; Neumann, Thomas; Kemper, Alfons: Meet Me Halfway: Split Maintenance of Continuous Views. Proc. VLDB Endow., 13(11):2620–2633, 2020.

[Za16] Zaharia, Matei; Xin, Reynold S.; Wendell, Patrick; Das, Tathagata; Armbrust, Michael; Dave, Ankur; Meng, Xiangrui; Rosen, Josh; Venkataraman, Shivaram; Franklin, Michael J.; Ghodsi, Ali; Gonzalez, Joseph; Shenker, Scott; Stoica, Ion: Apache Spark: a unified engine for big data processing. Commun. ACM, 59(11):56–65, 2016.

[Ze20] Zeuch, Steffen; Chaudhary, Ankit; Monte, Bonaventura Del; Gavriilidis, Haralampos; Giouroukis, Dimitrios; Grulich, Philipp M.; Breß, Sebastian; Traub, Jonas; Markl, Volker: The NebulaStream Platform for Data and Application Management in the Internet of Things. In: CIDR. www.cidrdb.org, 2020.