juliacon

StreamSampling.jl: Efficient Sampling from Data Streams in Julia

Adriano Meligrana[1]

[1]Sapienza University of Rome

ABSTRACT

StreamSampling.jl is a Julia library designed to provide general and efficient methods for sampling from data streams in a single pass, even when the total number of items is unknown. In this paper, we describe the capabilities of the library and its advantages over traditional sampling procedures, such as maintaining a small, constant memory footprint and avoiding the need to fully materialize the stream in memory. Furthermore, we provide empirical benchmarks comparing online sampling methods against standard approaches, demonstrating performance and memory improvements.

Keywords

Julia, Reservoir Sampling, Sequential Sampling, Online Sampling, Data Streams

1. Introduction

Random sampling from data streams is a fundamental operation in data analysis, particularly when dealing with massive datasets that exceed available memory or continuous streams of indeterminate length. The methods covered here fall under the umbrella of *online sampling*, i.e. algorithms that process stream elements sequentially in a single pass, which can be broadly divided into two subcategories: reservoir sampling and sequential sampling [16, 17].
Reservoir sampling algorithms maintain a sample of a fixed size K, which is dynamically updated as stream elements are processed. They guarantee that, at any point during the process, the collected sample is representative of the portion of the stream seen so far. This makes them ideal for unbounded, continuous streams where the total population size N is unknown [16].
Sequential sampling algorithms, by contrast, are typically used when either the total number of elements in the stream (N) or the exact total weight (W_N) is known in advance [17, 13]. These methods return an ordered sample of the stream, without keeping track of previous sampled elements. A sequential algorithm can compute, from a single random variate, how many elements to skip over before the next selection, without the need to keep a reservoir. This can make sequential methods more efficient than reservoir methods when prior knowledge of N or W_N is available.
`StreamSampling.jl` provides a comprehensive native Julia suite of algorithms covering both categories. However, a fundamental advantage of reservoir methods over sequential methods is that they maintain a representative sample in memory at all times, even when only a portion of the N elements have been processed, a property that sequential methods lack by design.

*Email: `adriano.meligrana@diag.uniroma1.it`

2. Statement of Need

Online sampling techniques have been implemented across a range of programming languages and frameworks, typically addressing specific use cases rather than providing a unified suite of algorithms. For instance, eBay's `tsv-utils` provide the `tsv-sample` command-line tool, which supports simple random sampling and weighted reservoir sampling over tabular streams, as well as Bernoulli sampling and distinct sampling for streaming scenarios. In the POSIX environment, the GNU Coreutils `shuf` binary [9] implements unweighted reservoir sampling when the `-n` option is used to extract a fixed number of lines from potentially unbounded inputs, without needing to know the total line count in advance.
In the Big Data domain, Apache DataFu [15] provides User-Defined Functions (UDFs) for Apache Pig, including both unweighted (`ReservoirSample`) and weighted (`WeightedReservoirSample`) reservoir sampling routines. These implementations are designed for the Hadoop/Pig ecosystem and are not directly usable outside of it. Similarly, in Python, the River library [11] focuses on online machine learning and includes sampling utilities primarily targeted at handling class imbalance in streaming classification and regression tasks, rather than general-purpose random sampling. Finally, the Clojure library `bigmlcom/sampling` most closely resembles `StreamSampling.jl` in terms of scope. However, it does not support the full range of reservoir and sequential sampling variations offered by `StreamSampling.jl`. Furthermore, the project appears to be inactive, with its last commit dating back seven years.
Compared to these alternatives, `StreamSampling.jl` provides a comprehensive suite of sampling methodologies directly within the Julia programming language [3], covering both reservoir and sequential paradigms, with and without replacement, and with or without weights. This makes it the only package, to the author's knowledge, that offers such breadth in a single, self-contained library. Furthermore, its design follows Julia's standard iterable protocol and is compatible with the `OnlineStats.jl` [6] API, enabling excellent integration with existing Julia data pipelines.

3. Research Impact Statement

Although `StreamSampling.jl` is a recent package, we believe the work already shows credible near-term scholarly significance. First, the benchmark results reported in Section 6 show that stream-native sampling can reduce both runtime and memory consumption relative to `StatsBase.sample` by avoiding full materialization of the population. These gains make iterator-based sampling practical in settings where population-based methods incur substantial allocation overhead.

Second, the out-of-core experiment in Section 7 demonstrates usefulness beyond synthetic iterator benchmarks. On a 100 GB Arrow dataset, the stream-based methods remain feasible at scales where the chunked `StatsBase.sample` baseline is slower and eventually fails due to out-of-memory errors.
Third, the package serves as a software vehicle for recent research results, since its implementation of `AlgWRSWRSKIP` is tied to a recent publication in the field [10].
Taken together, these results provide evidence that the library already transfers current data-stream sampling research into practical Julia workflows and is positioned for near-term use in statistical computing and large-scale data analysis.

4. Implemented Methods

`StreamSampling.jl` categorizes its implemented methods along three main axes: the underlying logic (reservoir vs. sequential), the sampling scheme (with vs. without replacement), and the inclusion probabilities (weighted vs. unweighted).
For reservoir sampling without replacement, `AlgR` and `AlgL` [16] are provided, whereas `AlgRSWRSKIP` [12] covers the with-replacement counterpart. For weighted reservoir sampling, `AlgARes` and `AlgAExpJ` [8] handle the without-replacement case, and `AlgWRSWRSKIP` [10] is used for the with-replacement case. In the sequential sampling domain, methods like `AlgD` [17] and `AlgHiddenShuffle` [13] are employed for unweighted without-replacement sampling. `AlgORDSWR` [2] is used for sequential sampling with replacement, and `AlgORDWSWR` [14] for weighted sequential sampling with replacement.
Notably absent from this set is a weighted sequential method without replacement. This is a fundamental mathematical impossibility when the only prior information over the unobserved elements is the total remaining weight of the stream W_N, which is what it is usually assumed to be known for weighted sequential sampling algorithms. In sampling with replacement, every draw is independent and the inclusion probability of any item, that is, the probability that it will appear in the final sample is proportional to its weight over W_N, so W_N alone is sufficient. In sampling without replacement, however, selecting an item alters the inclusion probabilities of all subsequent items. To correctly compute the inclusion probability of the next item, one needs to know the individual weights of all remaining elements in the stream, not merely their sum: different configurations of individual weights can yield the same total W_N but require different inclusion probabilities. Knowing only W_N therefore leaves the per-element inclusion probabilities undetermined, making it impossible to draw a correctly weighted sample without replacement in a sequential single pass.

Method	Ref.	Type	Replacement	Weighted	Time	Space
AlgR	[16]	Reservoir	Without	False	$\mathcal{O}(N)$	$\mathcal{O}(K)$
AlgL	[16]	Reservoir	Without	False	$\mathcal{O}(K\log(N/K))$	$\mathcal{O}(K)$
AlgRSWRSKIP	[12]	Reservoir	With	False	$\mathcal{O}(K\log N)$	$\mathcal{O}(K)$
AlgARes	[8]	Reservoir	Without	True	$\mathcal{O}(N)$	$\mathcal{O}(K)$
AlgAExpJ	[8]	Reservoir	Without	True	$\mathcal{O}(K\log(N/K))^*$	$\mathcal{O}(K)$
AlgWRSWRSKIP	[10]	Reservoir	With	True	$\mathcal{O}(K\log W_N)$	$\mathcal{O}(K)$
AlgD	[17]	Sequential	Without	False	$\mathcal{O}(K)$	$\mathcal{O}(1)$
AlgHiddenShuffle	[13]	Sequential	Without	False	$\mathcal{O}(K)$	$\mathcal{O}(1)$
AlgORDSWR	[2]	Sequential	With	False	$\mathcal{O}(K)$	$\mathcal{O}(1)$
AlgORDWSWR	[14]	Sequential	With	True	$\mathcal{O}(K)$	$\mathcal{O}(1)$

Table 1. : Sampling algorithms implemented in `StreamSampling.jl` (N=population size, K=sample size, W_N=total weight). *The expected $\mathcal{O}(K\log(N/K))$ complexity for `AlgAExpJ` holds when weights are independent and identically distributed (i.i.d.) draws [8].

Table 1 summarizes the algorithms implemented in the package along with their time and space complexities.

It is important to clarify that the *Time* complexity metric strictly evaluates the running time in terms of the number of random variates that must be generated to obtain a sample of size K from a (possibly weighted) population of size N.

5. Software Design

The software design has two goals: integration with existing Julia streaming workflows and efficient support for the different sampling algorithms implemented by the package. For reservoir samplers, this led to an interface aligned with `OnlineStats.jl` [6]. Rather than introducing a package-specific stateful API, reservoir samplers support `fit!`, `value`, and `merge!`. This makes them composable with existing online and streaming workflows. The use of `fit!` and `value` for reservoir sampling has previously been proposed in [7], while mergeable reservoirs have been discussed in [5]. The latter operation is particularly important for parallel and distributed sampling workflows.
Reservoir samplers naturally fit the `OnlineStats.jl` model because they maintain a state that is updated as observations arrive and can later be queried or merged. Sequential samplers, however, have a different computational structure and therefore follow Julia's iterator protocol instead.
The implementation also balances performance with flexibility by providing most reservoir samplers in both mutable and immutable forms. These variants are generated from a shared definition using `HybridStructs.jl`. The mutable variants are easier to work with for workflows in which surrounding code requires storing references of the sampler object. Empirical testing showed that the immutable variants are generally more performant, and they are therefore retained for cases where more speed is required.
The internal data structures to store the sample are chosen to match the operations required by each algorithm. Weighted sampling without replacement, as implemented by `AlgARes` and `AlgAExpJ`, repeatedly compares candidate priorities with the current minimum retained priority to choose if an element needs to be sampled. For this reason, the sample is stored in a binary heap, as is usually recommended [8, 7]. This allows the current top-K priorities to be maintained efficiently and also simplifies merging, at the cost of the ordering overhead introduced by the heap. In contrast, the unweighted algorithms and the weighted algorithms with replacement maintain fixed sample slots in flat arrays. This simpler representation allows these algorithms to outperform the heap-based implementations, as shown in Figure 1.

5.1 Package Interface

The interface of `StreamSampling.jl` relies on two main samplers, each made for its sampling domain: `ReservoirSampler` and `SequentialSampler`. The package also provides `itsample`, which, similarly to `StatsBase.sample`, returns an `Array`, but, by using stream methods, it can be applied to any iterator.

5.1.1 Reservoir Samplers. As previously described, the `ReservoirSampler` API exposes three main core operations:

—`fit!(sampler, item, [weight])`: Processes a new element from the stream and updates the sample accordingly.
—`value(sampler)`: Returns the current in-memory sample.
—`merge!(sampler1, sampler2, ...)`: Combines multiple reservoir samplers, each typically operating on a different partition of the stream, into a single statistically consistent reservoir.

5.1.2 Sequential Samplers. Instead of maintaining an in-memory reservoir, `SequentialSampler` wraps an input iterable together

with the required population size (or total weight) and conforms to Julia's standard `iterate` protocol. Rather than processing each element via `fit!`, the sampler computes skip lengths on the fly and emits selected elements directly as the iterator is consumed, without any intermediate collection.
For parallel workflows, `SequentialSampler` provides a `combine(samples, weights)` function. Similarly to `merge!`, `combine` merges samples taken from different partitions into a single globally correct sample by re-weighting each local sample proportionally to its partition's share of the total weight, ensuring that the final output has the correct global inclusion probabilities.

5.1.3 Iterator-Based Sampling Interface. `itsample` provides a convenience layer that dispatches between stream-oriented and length-aware implementations using iterator traits. When `Base.IteratorSize(iter)` is `Base.SizeUnknown`, it selects reservoir methods that remain valid on a one-pass stream. When the iterable has a known length, it can instead switch to sequential methods that exploit that additional information. Users who need exact control over the selected algorithm or who need to update the sample incrementally should instantiate `ReservoirSampler` or `SequentialSampler` directly.

5.2 Example Usage

The following minimal examples illustrate typical usage of the interface. Notice that, in all cases, the iterator is never materialized in memory:

```julia
using StreamSampling

stream = 1:10^8
K = 10
alg = AlgL()
sampler = ReservoirSampler{Int}(K, alg)
for x in stream
    fit!(sampler, x) # O(1) per element
end

# returns a vector of length 10
sample = value(sampler)
```

Code 1: Reservoir sampling without replacement of $K = 10$ elements from an iterator of unknown size.

```julia
using StreamSampling

stream = 1:10^8
K, N = 10, 10^8
alg = AlgD()
sampler = SequentialSampler{Int}(stream, K, N, alg)
for x in sampler
    println(x) # sampled element emitted on the fly
end
```

Code 2: Sequential sampling without replacement of $K = 10$ elements when the population size N is known.

```julia
using StreamSampling

# returns a vector of length 10
itsample(1:10^8, 10)
```

Code 3: Sampling without replacement of $K = 10$ elements with the `itsample` convenience layer.

The reservoir sampler maintains a buffer of exactly K elements throughout and is valid even when N is unknown. The sequential sampler holds no buffer at all; it skips directly to each selected element using the precomputed population size N.

6. Benchmarks

Several benchmarks were conducted to evaluate the performance of the reservoir and sequential sampling implementations[1]. All benchmarks were run on a machine with an AMD Ryzen 5 5600H (6 cores) and 16GB of RAM, running Ubuntu 24.04 LTS and Julia 1.12.
To evaluate the algorithms' core computational performance, we benchmarked them on a simple iterator, drawing samples of sizes ranging from 0.01% to 10% of a generator producing integers between 1 and 10^8. In this setup, the baseline population method uses `StatsBase.sample`, which requires fully materializing the iterator into an array before sampling. This imposes a significant memory footprint and allocation overhead.
The reservoir and sequential methods provided by the package bypass collecting the iterator into memory entirely, drastically reducing allocations. While reservoir algorithms naturally handle iterators of unknown length in a single pass, sequential sampling algorithms require the total weight or population size to be known in advance to calculate inclusion probabilities. If this total is unknown, a prior pass over the iterator is required, so a two-passes method is also included in the comparison.
Figure 1 summarizes the results of the benchmarks. Each benchmark was run for at least 20 seconds of wall time, recording the median execution time and memory allocations across four sampling scenarios: unweighted/weighted and with/without replacement:

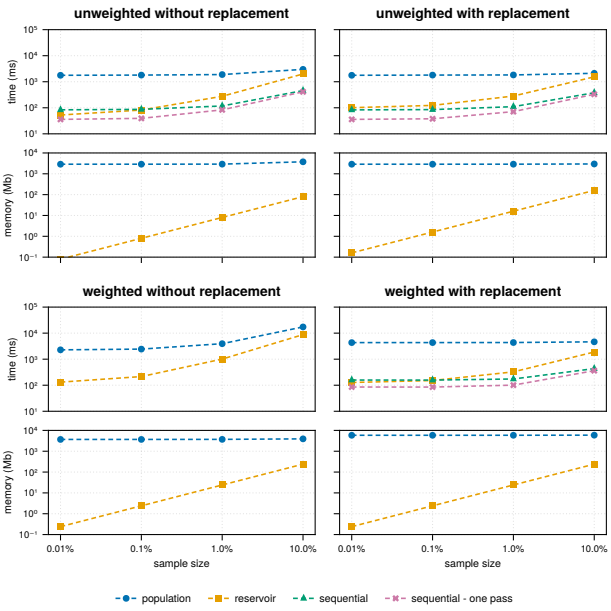

Fig. 1: Comparison of median execution time (ms) and memory allocation (MB) for sampling varying sizes (10^4 to 10^7 elements) from an iterator of 10^8 elements.

The population baseline forces full iterator materialization, resulting in the highest memory usage, while no allocation is performed

by sequential algorithms so no line is shown for those. The reservoir methods can be seen to operate instead within $\mathcal{O}(K)$ memory. As expected by previous discussions, reservoir and sequential methods are much faster than population-based methods for small sample sizes. The one pass sequential variation demonstrates the time savings achieved when the total population weight is known in advance, allowing the algorithm to avoid a secondary traversal of the iterator.

7. Applications

The algorithms provided by `StreamSampling.jl` can be applied across various data-intensive domains including approximate query processing, online log monitoring, querying massive network analytics feeds, and sensor data sub-sampling where unbounded observations naturally prohibit offline computational constraints [5, 18]. A direct application is efficient sampling from persistent data[1]. To demonstrate the usefulness of the algorithms presented earlier in this setting, we performed an experiment to sample from 100 GB of weighted tabular data stored on disk in the Arrow format (a columnar binary format enabling efficient I/O without full deserialization). As a baseline, a chunking approach which loads and samples each chunk with `StatsBase.sample` and eventually recombines the partial samples is provided. Reservoir techniques process the elements in a continuous single pass which can allow them to outperform the chunking approach. Sequential methods are similarly applicable; however, they require two passes, one to compute the aggregate total weight W_N, and a second pass to extract the sample.

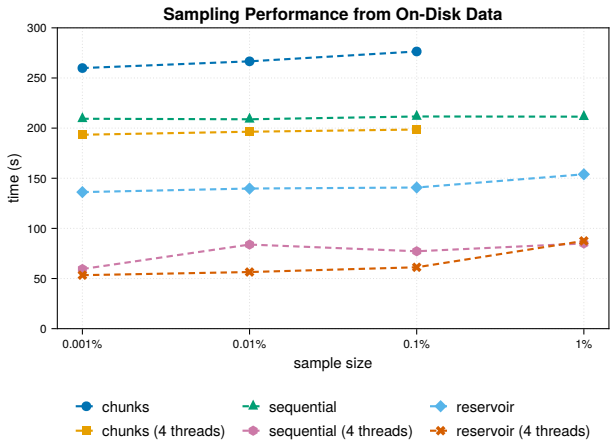

Fig. 2: Wall-clock time for weighted sampling with replacement from 100 GB of Arrow tabular data on disk, as a function of sample size K (shown as a percentage of the total row count $N \approx 3.1 \times 10^9$). The chunks baseline crashed at the largest sample size.

The out-of-core experiment was conducted on the same machine and in the same environment described earlier for the iterator benchmarks. In addition, it should be noted that a 512 GB KIOXIA KXG60ZNV SSD was used as persistent storage. File I/O was performed via `Arrow.jl` and OS page-cache effects were mitigated by dropping the page cache between runs. Execution times are single-trial measurements.

[1]The benchmarks and applications described are available at https://github.com/JuliaDynamics/StreamSampling.jl/tree/main/benchmark.

Figure 2 presents execution times for extracting a weighted sample with replacement from the 100 GB Arrow file with the different methods.
The baseline chunked `StatsBase.sample` strategy exhibits the highest execution times across all tested configurations. The single-threaded chunks approach requires between 260 and 280 seconds, while the 4-thread version improves this to approximately 195 seconds. Furthermore, both chunk configurations fail due to out-of-memory errors before completing the 1% sample size extraction.
In contrast, the reservoir and sequential methods (which use `AlgWRSWRSKIP` and `AlgORDWSWR` respectively) successfully process the largest sample sizes while requiring substantially lower execution times. For single-threaded execution, the reservoir method is the most efficient, operating in the 135–155 second range and outperforming the single-threaded sequential method, which remains flat at approximately 210 seconds. However, the sequential method demonstrates superior parallel scaling in comparison to the reservoir method.

8. Conclusion

`StreamSampling.jl` introduces a comprehensive suite of algorithms covering both reservoir and sequential paradigms, filling a gap in the Julia ecosystem. The ability to sample from any Julia iterator without materializing it in memory extends the reach of random sampling to contexts such as lazy data pipelines and large on-disk datasets. Future work may aim to extend the library to support other sampling algorithms. This includes implementing methods where the sample size is only guaranteed in expectation, such as Bernoulli and Poisson sampling. Furthermore, we plan to introduce sliding window sampling techniques [4] as well as stratified reservoir algorithms [1]. Finally, we aim to integrate the package algorithms into the broader Julia ecosystem, specifically by improving the poly-algorithm implementation of `StatsBase.sample` to dispatch to these methods under favorable circumstances.

9. AI Usage Disclosure

During the preparation of this work, the author used some AI tools for the purpose of proofreading and improving readability. On the software side, these have also been used to add tests to improve code coverage. The author reviewed and edited the content as needed and takes full responsibility for the final content of the publication.

10. References

[1] Mohammed Al-Kateb and Byung Suk Lee. Stratified reservoir sampling over heterogeneous data streams. In *Scientific and Statistical Database Management*, pages 621–639. Springer Berlin Heidelberg, 2010. doi:10.1007/978-3-642-13818-8.

[2] Jon Louis Bentley and James B. Saxe. Generating sorted lists of random numbers. *ACM Transactions on Mathematical Software*, 6(3):359–364, 1980. doi:10.1145/355900.355907.

[3] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017. doi:10.1137/141000671. https://doi.org/10.1137/141000671.

[4] Vladimir Braverman, Rafail Ostrovsky, and Carlo Zaniolo. Optimal sampling from sliding windows. *Journal of Computer and System Sciences*, 78(1):260–272, 2012.

doi:10.1016/j.jcss.2011.04.004. JCSS Knowledge Representation and Reasoning.

[5] 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, 2011. doi:10.1561/1900000004. https://www.emerald.com/ftdbs/article-pdf/4/1-3/1/11024120/1900000004en.pdf.

[6] Josh Day and Hua Zhou. Onlinestats.jl: A julia package for statistics on data streams. *Journal of Open Source Software*, 5(46):1816, 2020. doi:10.21105/joss.01816.

[7] Pavlos S Efraimidis. Weighted random sampling over data streams. In *Algorithms, Probability, Networks, and Games: Scientific Papers and Essays Dedicated to Paul G. Spirakis on the Occasion of His 60th Birthday*, pages 183–195. Springer International Publishing, 2015. doi:10.1007/978-3-319-24024-4.

[8] Pavlos S. Efraimidis and Paul G. Spirakis. Weighted random sampling with a reservoir. *Information Processing Letters*, 97(5):181–185, 2006. doi:10.1016/j.ipl.2005.11.003.

[9] David MacKenzie et al. gnu coreutils. *Free Software Foundation, Inc*, 2022, 1994.

[10] Adriano Meligrana and Adriano Fazzone. Weighted reservoir sampling with replacement from data streams. In *Proceedings of the ACM Web Conference 2026*, WWW '26, page 8789–8792, New York, NY, USA, 2026. Association for Computing Machinery. doi:10.1145/3774904.3792966.

[11] Jacob Montiel, Max Halford, Saulo Martiello Mastelini, Geoffrey Bolmier, Raphaël Sourty, Robin Vaysse, Adil Zouitine, Heitor Murilo Gomes, Jesse Read, Talel Abdessalem, and Albert Bifet. River: machine learning for streaming data in python. *Journal of Machine Learning Research*, 22:1–8, 2021. doi:10.48550/arXiv.2012.04740.

[12] Byung-Hoon Park, George Ostrouchov, and Nagiza F. Samatova. Sampling streaming data with replacement. *Computational Statistics & Data Analysis*, 52(2):750–762, 2007. doi:10.1016/j.csda.2007.03.010.

[13] Michael Shekelyan and Graham Cormode. Sequential random sampling revisited: Hidden shuffle method. In *Proceedings of The 24th International Conference on Artificial Intelligence and Statistics*, volume 130 of *Proceedings of Machine Learning Research*, pages 3628–3636. PMLR, 2021.

[14] Michal Startek. An asymptotically optimal, online algorithm for weighted random sampling with replacement. *CoRR*, abs/1611.00532, 2016. arXiv:1611.00532.

[15] Roshan Sumbaly, Jay Kreps, and Sam Shah. The big data ecosystem at linkedin. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, page 1125–1134. Association for Computing Machinery, 2013. doi:10.1145/2463676.2463707.

[16] Jeffrey S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 11(1):37–57, 1985. doi:10.1145/3147.3165.

[17] Jeffrey Scott Vitter. An efficient algorithm for sequential random sampling. *ACM Transactions on Mathematical Software*, 13(1):58–67, 1987. doi:10.1145/23002.23003.

[18] Joel Wolfrath and Abhishek Chandra. Efficient transmission and reconstruction of dependent data streams via edge sampling. In *2022 IEEE International Conference on Cloud Engineering (IC2E)*, pages 47–57, 2022. doi:10.1109/IC2E55432.2022.00013.