Lecture 2: Introduction to Data Stream Sampling

08-28-2025 Lecturer: Ali Vakilian | Scribe: Alessandro Shapiro | Editor: Ali Vakilian

1 The Streaming Model of Computation

When processing massive datasets, we are often limited by the amount of available memory. We cannot store the entire dataset at once, which motivates the design of algorithms that treat data as a **stream**.

In the streaming model, the input consists of N items $e_1, \ldots, e_N$ that arrive sequentially. The algorithm is constrained to use a small amount of memory space, say B bits, where $B \ll N$. The goal is to compute a function or statistic of interest over this stream in one or a few passes while respecting the memory constraint.

This model is relevant in many modern applications. For example, network routers process streams of packets, financial systems analyze streams of transactions, and IoT devices process streams of sensor readings. In all these cases, the volume and velocity of the data make it infeasible to store everything. The challenge is to design algorithms that can extract meaningful information, such as summaries, statistics, or sketches, in a single pass using sublinear space. The key measure of quality is the *trade-off between memory usage, the accuracy of the result*, and the number of passes over the data. In particular, while computing an *exact* answer often requires memory proportional to the data size, the powerful combination of **randomization** and **approximation** allows us to solve many of these problems efficiently.

2 Sampling from a Stream

Random sampling is a fundamental technique in the streaming model. The core idea is to select a *small, random subset* S from a large data stream X and use this sample to estimate a quantity of interest. This approach allows us to analyze the properties of the entire dataset while only storing a tiny fraction of it. The effectiveness of this method depends on three factors:

1. **Sampling Strategy.** How the subset is chosen (e.g., uniformly, with weights).
2. **Sample Size.** The number of samples needed to achieve a desired accuracy.
3. **Estimator.** The function computed on the sample to estimate the true value.

2.1 Estimation and Accuracy

A primary goal when designing an estimator is to ensure its accuracy. A fundamental and highly desirable property in this regard is for an estimator to be **unbiased**. An estimator $\hat{\theta}$ is unbiased if its expected value is equal to the true parameter θ it is trying to estimate, i.e., $\mathbb{E}[\hat{\theta}] = \theta$.

Unbiased estimators are particularly useful and foundational in algorithm design for several key reasons:

- **Intuitive Correctness:** They guarantee that, on average, the estimation process does not systematically over- or underestimate the true value.
- **Analytical Simplicity:** Unbiasedness provides a clean mathematical starting point. For them, proving guarantees often begins by showing that an estimator is unbiased and then bounding its variance.

- **Systematic Improvement:** The accuracy of an unbiased estimator can be reliably improved. By averaging multiple independent copies of the estimator, we can systematically drive down the variance, making the estimate more concentrated around the true mean.

While an unbiased estimator is correct on average, any single estimate might still deviate *significantly* from the true value. Therefore, after establishing unbiasedness, the next crucial step is to analyze its **variance** to provide strong guarantees on its accuracy and reliability.

Variance Reduction by Averaging. A standard technique to reduce variance is to take the average of several *independent* estimates. Suppose we have k *independent and identically distributed (i.i.d.)* estimators $\hat{\theta}_1, \ldots, \hat{\theta}_k$. Let their average be $\hat{\theta}_{\text{avg}} = \frac{1}{k}\sum_{i=1}^{k} \hat{\theta}_i$. By linearity of expectation, the average estimator is still unbiased: $\mathbb{E}[\hat{\theta}_{\text{avg}}] = \theta$. Its variance, however, is reduced by a factor of k.

Lemma 2.1. *If $\hat{\theta}_1, \ldots, \hat{\theta}_k$ are independent random variables with common variance $\sigma^2 = \mathrm{Var}[\hat{\theta}_i]$, then $\mathrm{Var}[\hat{\theta}_{avg}] = \sigma^2/k$.*

Proof. Using the properties of variance and the independence of the estimators:

$$\mathrm{Var}[\hat{\theta}_{\text{avg}}] = \mathrm{Var}\left[\frac{1}{k}\sum_{i=1}^{k}\hat{\theta}_i\right] = \frac{1}{k^2}\,\mathrm{Var}\left[\sum_{i=1}^{k}\hat{\theta}_i\right] = \frac{1}{k^2}\sum_{i=1}^{k}\mathrm{Var}[\hat{\theta}_i] = \frac{1}{k^2}\cdot k\sigma^2 = \frac{\sigma^2}{k}.$$

□

Note that in above lemma we crucially use the fact that $\hat{\theta}_1, \ldots, \hat{\theta}_k$ are *independent* random variables.

Sample Size. With the variance reduced, we can use concentration inequalities to bound the sample size k needed to guarantee that our estimate is within an error ϵ of the true value with probability at least $1 - \delta$.

- Using **Chebyshev's inequality**, we get $\Pr[|\hat{\theta}_{\text{avg}} - \theta| \geq \epsilon] \leq \frac{\mathrm{Var}[\hat{\theta}_{\text{avg}}]}{\epsilon^2} = \frac{\sigma^2}{k\epsilon^2}$. To make this failure probability at most δ, we need $k = O(\frac{\sigma^2}{\epsilon^2\delta})$.
- If the estimator is a sum of independent, bounded random variables, we can use stronger bounds like **Chernoff or Hoeffding**. These give exponential tails, yielding a much better dependency on δ. Typically, the required sample size becomes $k = O(\frac{1}{\epsilon^2}\log\frac{1}{\delta})$.

Takeaway 2.1

To achieve an additive error of at most ϵ with a failure probability of δ, a sample size of $k = O\left(\frac{1}{\epsilon^2}\log\frac{1}{\delta}\right)$ is typically sufficient.

2.2 Reservoir Sampling

While drawing a uniform random sample from a static dataset is typically straightforward, the task becomes non-trivial in a streaming setting. The primary challenge is that the stream's total length is unknown in advance, making it difficult to assign the correct inclusion probability to each item on the fly. **Reservoir sampling** is a classic algorithm that elegantly solves this problem in the streaming model.

Sampling a Single Item ($k = 1$). Suppose we want to maintain a single, uniformly random sample from a stream of unknown length N. The algorithm is as follows: Store the first item. When the t-th item arrives (for $t > 1$), replace the currently stored sample with this new item with probability $1/t$.

Algorithm 1 Reservoir Sampling ($k = 1$)

```
1: Initialize sample to be empty.
2: Initialize counter $t \leftarrow 0$.
3: for each item $x$ in the stream do
4:     $t \leftarrow t + 1$.
5:     Draw a random integer $r$ uniformly from $\{1, \dots, t\}$.
6:     if $r = 1$ then
7:         sample $\leftarrow x$.
8: return sample.
```

Theorem 2.2. *After m items have been processed, the item stored by the single-item reservoir sampling algorithm is a uniform sample from the stream. That is, for any $i \in \{1, \dots, m\}$, the probability that item e_i is the final sample is $1/m$.*

Proof. We prove by induction on the stream length t that after t items, the probability of any item e_i ($i \leq t$) being the sample is $1/t$.

- **Base Case ($t = 1$):** The first item e_1 is stored with probability 1. The claim holds.
- **Inductive Step:** Assume that after t items, for any $i \leq t$, $\Pr[\text{sample is } e_i] = 1/t$. Now consider the $(t+1)$-th item, e_{t+1}. The algorithm replaces the current sample with e_{t+1} with probability $1/(t+1)$. Thus, the probability that e_{t+1} is the sample is $1/(t+1)$.

  For any item e_i with $i \leq t$, it remains the sample only if it was the sample after step t (with probability $1/t$) AND it was not replaced at step $t + 1$ (with probability $1 - 1/(t + 1)$). Therefore,

$$\begin{aligned}\Pr[e_i \text{ is sample at step } t+1] &= \Pr[e_i \text{ is sample at step } t] \cdot \Pr[\text{not replaced}] \\ &= \frac{1}{t} \cdot \left(1 - \frac{1}{t+1}\right) = \frac{1}{t} \cdot \frac{t}{t+1} = \frac{1}{t+1}.\end{aligned}$$

The claim holds for all t. Setting $t = m$ completes the proof. □

Sampling k Items With Replacement. The most straightforward way to obtain a sample of size k is to sample *with replacement*. This is easily accomplished by running k independent instances of the single-item reservoir sampling algorithm in parallel. Each instance processes the same stream and maintains its own uniformly random sample. The final output is the collection of these k items. Because each sampler operates independently, this method is simple to implement, though it allows for the same item to appear multiple times in the final sample set.

Sampling k Items Without Replacement. However, maintaining a uniform sample of k items *without replacement* is more involved. The simple parallel approach is no longer sufficient, as the decision to keep one item is dependent on the others in the sample, and a more coordinated strategy is required.

The algorithm is as follows: Initialize a reservoir S with the first k items of the stream. For each subsequent item e_t (where $t > k$), include it in the reservoir with probability k/t. If e_t is included, it replaces a randomly chosen item from the reservoir.

Algorithm 2 Reservoir Sampling (k items without replacement)

```
1: Initialize a reservoir array $S$ of size $k$ with the first $k$ items from the stream.
2: Initialize a counter $t \leftarrow k$.
3: for each item $x$ in the stream starting from the $(k+1)$-th item do
4:     $t \leftarrow t + 1$.
5:     Draw a random integer $j$ uniformly from $\{1, \ldots, t\}$.
6:     if $j \leq k$ then
7:         $S[j] \leftarrow x$.                ▷ Replace the item at the randomly chosen index $j$.
8: return $S$.
```

Theorem 2.3. *After m items have been processed, the reservoir S contains a uniform random sample of k items from the stream without replacement.*

Proof. We prove by induction on the stream length $t \geq k$ that any item e_i ($i \leq t$) is in the reservoir S with probability k/t.

- **Base Case ($t = k$):** The reservoir contains the first k items, so for $i \leq k$, $\Pr[e_i \in S] = 1 = k/k$. The claim holds.

- **Inductive Step:** Assume that after t items, for any $i \leq t$, $\Pr[e_i \in S] = k/t$. Now consider the $(t+1)$-th item, e_{t+1}. The algorithm includes e_{t+1} in the reservoir with probability $k/(t+1)$. So, $\Pr[e_{t+1} \in S \text{ at step } t+1] = k/(t+1)$.

  For any item e_i with $i \leq t$, it remains in the reservoir at step $t+1$ if it was already in the reservoir AND it was not replaced. This occurs in two mutually exclusive cases:

  1. The new item e_{t+1} is not selected to be in the reservoir (with probability $1 - k/(t+1)$).
  2. The new item e_{t+1} is selected (with probability $k/(t+1)$), but e_i is not the one chosen for eviction (with probability $(k-1)/k$).

  The probability that e_i remains in S, given it was in S at step t, is:

$$\Pr[e_i \text{ survives}] = \left(1 - \frac{k}{t+1}\right) + \left(\frac{k}{t+1} \cdot \frac{k-1}{k}\right) = \frac{t+1-k}{t+1} + \frac{k-1}{t+1} = \frac{t}{t+1}.$$

  By the induction hypothesis, the total probability that e_i is in S after step $t+1$ is:

$$\Pr[e_i \in S \text{ at step } t] \cdot \Pr[e_i \text{ survives}] = \frac{k}{t} \cdot \frac{t}{t+1} = \frac{k}{t+1}.$$

The claim holds for all $t \geq k$. Setting $t = m$ completes the proof. □

2.3 Single-Item Weighted Sampling

This algorithm is a direct extension of the unweighted case, where the inclusion probability is based on the item's proportional weight.

Algorithm 3 Weighted Reservoir Sampling ($k = 1$)

```
1: Initialize sample to be empty.
2: Initialize total weight seen $W \leftarrow 0$.
3: for each item $(x, w)$ in the stream do
4:     $W \leftarrow W + w$.
5:     With probability $w/W$:
6:         sample $\leftarrow x$.
7: return sample.
```

Weighted Sampling (without replacement). Extending this logic to maintain a sample of k items *without replacement* follows a similar structure, with one crucial difference in the replacement step. In the unweighted algorithm, an item is evicted from a uniform random position. To correctly account for importance, the weighted algorithm must instead evict an existing item with probability proportional to its weight. This ensures that items with higher weights have a greater chance of being retained over the course of the stream.

Algorithm 4 Weighted Reservoir Sampling (k items without replacement)

```
1: Initialize a reservoir $S$ with the first $k$ items from the stream.
2: Initialize $W$, the sum of weights of all items seen so far.
3: for each item $(x_t, w_t)$ in the stream starting from the $(k+1)$-th item do
4:     $W \leftarrow W + w_t$.
5:     With probability $w_t/W$, decide to replace an item currently in $S$.
6:     if a replacement is chosen then
7:         Select an item $(x_j, w_j)$ from $S$ to discard with probability proportional to its weight $w_j$.
8:         Replace the discarded item with $(x_t, w_t)$.
9: return the items in $S$.
```

Theorem 2.4. *Let S be the sample of size k returned by the weighted reservoir sampling algorithm on a stream of m items. For any set of k items $\{e_{i_1}, \ldots, e_{i_k}\}$, the probability that S is exactly this set is proportional to the product of their weights.*

Analysis of Weighted Reservoir Sampling ($k = 1$). The proof for the general k-item case is quite involved, but the core inductive logic can be understood by first examining the simpler $k = 1$ case.

Theorem 2.5. *After processing m items, the probability that the final sample is item e_i is proportional to its weight, i.e.,* $\Pr[\textit{sample} = e_i] = w_i/W_m$.

Proof. We will prove by induction on the number of items processed, t. Let the inductive hypothesis, $P(t)$: "After processing t items, for any $i \in \{1, \ldots, t\}$, the probability that the sample is e_i is w_i/W_t."

Base Case ($t = 1$): After the first item arrives, the sample is e_1 with probability 1. The cumulative weight is $W_1 = w_1$. The probability is $w_1/W_1 = 1$. Thus, $P(1)$ holds.

Inductive Step: Assume that $P(t-1)$ is true. That is, after processing $t-1$ items, the probability that the sample is e_i (for any $i < t$) is w_i/W_{t-1}. Now, let's consider when the t-th item, e_t, arrives.

1. **Probability for the new item, e_t:** The new item e_t becomes the sample if it is chosen for replacement, which occurs with probability w_t/W_t. So, for $i = t$, the hypothesis holds.

2. **Probability for a previous item, e_i (where $i < t$):** For an item e_i to be the sample after t items, two independent events must have occurred: (a) It must have been the sample after $t-1$ items. By our hypothesis, this probability is w_i/W_{t-1}, and (b) the new item e_t must *not* have been chosen to replace it, which occurs with probability $1-(w_t/W_t) = (W_t - w_t)/W_t = W_{t-1}/W_t$.

   Therefore, the total probability that e_i is the sample after t items is the product of these probabilities:

$$\begin{aligned}\Pr[\text{sample} = e_i \text{ at step } t] &= \Pr[\text{sample} = e_i \text{ at step } t-1] \times \Pr[\text{not replaced at step } t] \\ &= \left(\frac{w_i}{W_{t-1}}\right) \times \left(\frac{W_{t-1}}{W_t}\right) = \frac{w_i}{W_t}\end{aligned}$$

Hence, the hypothesis $P(t)$ holds for all $i \leq t$. □

The proof for the more general case of sampling k items builds on the same logic.

Proof Sketch of Theorem 2.4. The proof for this theorem also proceeds by induction on the stream length t. However, the state space is much larger, as one must track the probability of every possible k-subset being in the reservoir. The inductive step involves more complex casework, considering the two possibilities for an arbitrary subset at step t: either it contains the new item e_t or it does not. While the algebra is more involved, the core idea of tracking how probabilities are preserved and transferred from one step to the next remains the same. □

2.4 Application: Mean and Median Estimation

Let's consider how to estimate two fundamental statistics of a stream of numbers: the mean and the median.

Mean Estimation. For a stream of N numbers $x_1, \ldots, x_N$, computing the exact mean is simple and requires very little space. We can maintain two variables: a running sum $S = \sum x_i$ and a counter t for the number of items seen. When the stream ends, the mean is simply S/t. This requires only $O(\log N + \log S_{max})$ space, making it straightforward for implementation in streaming.

Median Estimation. Computing the median is significantly harder in a stream. The median is the value that separates the lower half of the data from the higher half, a definition that seems to require sorting the entire dataset. Storing all N items to sort them is infeasible in the streaming model as the length of the stream is assumed to be extremely large.

Optional Reading

Even in an offline setting (with all data available), computing the median in linear time is a classic algorithms problem. The randomized Quickelect algorithm, which is similar to Quicksort, can find the median of n items in expected linear time. However, its worst-case runtime is $O(n^2)$ if it repeatedly chooses bad pivots.

Algorithm 5 Randomized Quickselect

```
1:  function QUICKSELECT(array, $k$)
2:                                          ▷ $k$ is the 0-indexed rank we are looking for
3:      left ← 0
4:      right ← length(array) − 1
5:      while left ≤ right do
6:          pivot ← RANDOMPIVOT(array, left, right)      ▷ Choose a pivot randomly from the range
7:          pivot ← PARTITION(array, left, right, pivot)  ▷ Partition the array around the pivot
8:          if $k$ = pivot then
9:              return array[$k$]                          ▷ Found the $k$-th element
10:         else if $k$ < pivot then
11:             right ← pivot − 1                          ▷ Search in the left part
12:         else
13:             left ← pivot + 1                           ▷ Search in the right part
```

Partition Function. This function takes a sub-array array[left...right] and a chosen pivot. It rearranges the elements, in $O(n)$ time by scanning all items in array, such that all values smaller than the array[pivot] come before it, and all values larger come after it. It returns the *index/rank* of the pivot after the rearrangement is complete, which corresponds to the pivot's correct position in the fully sorted array.

A deterministic linear-time guarantee can be achieved using the Median of Medians algorithm.

The Median of Medians Algorithm. This algorithm guarantees a good pivot for partitioning, ensuring worst-case $O(n)$ performance. It works as follows:

1. Divide the list of N items into groups of 5.
2. Find the median of each group (a constant time operation for 5 elements). This yields $n/5$ medians.
3. Recursively call the algorithm on this list of $n/5$ medians to find their true median. This **median-of-medians** is used as the pivot.
4. Partition the original array around this pivot. The pivot is guaranteed to be "good" in the sense that at least 30% of the elements are smaller and at least 30% are larger.
5. Recurse on the side of the partition that contains the desired median.

Based on the steps provided, the worst-case runtime of the Median of Medians algorithm can be described by the following recurrence relation:

$$T(n) = T(n/5) + T(7n/10) + O(n)$$

Here's where each term comes from:

- $T(n/5)$: This is the cost of the recursive call to find the true median of the $n/5$ group medians.
- $T(7n/10)$: In the worst case, the pivot guarantees a split of at least 30/70. This means the next recursive call will be on a sub-array of at most $7n/10$ elements.

- $O(n)$: This term represents the linear-time work done at each step, which includes dividing the list into groups of size 5, finding the median of each small group, and partitioning the array around the pivot.

It is straightforward to verify that this recurrence resolves to $T(n) = O(n)$. The key insight is that the total size of the subproblems in the recursive step is a fraction of the original size: $n/5 + 7n/10 = 9n/10 < n$. Because the work at each level of recursion decreases geometrically, the sum of work across all levels converges to a linear function of n.

However, the existing algorithms for statistic data require multiple passes and random access over the data to partition and recurse. Therefore, they *cannot be implemented efficiently in streamed data.*

Next session, we will learn how to estimate the median in the streaming setting.