Streaming Basics: One-pass Data Processing with Tiny Memory

Contents

1 Introduction

Data streaming refers to processing a (potentially infinite) sequence of data items one at a time, typically using very limited memory. Unlike traditional batch processing, where the entire dataset is available for random access, streaming algorithms process each item on the fly and often cannot store the entire stream. This model is motivated by big data scenarios where data arrives continuously at high velocity (e.g., network traffic, sensor readings, social media feeds) and storing all raw data is impractical or impossible. Streaming algorithms aim for *one-pass* (or few-pass) processing with *sublinear* memory (often polylogarithmic in the data size) and provide approximate answers with provable error bounds.

Key differences from batch processing include:

- **Memory constraint:** Streaming algorithms use memory much smaller than the stream size (often $O(\text{poly}(\log n, 1/\epsilon))$).
- **One-pass processing:** Each data item is examined only once (or a small number of times) in arrival order.
- **Approximation:** Exact answers (e.g., exact count of distinct items) are usually impossible under tight memory bounds; instead, streaming methods provide *approximate* answers with provable error guarantees.
- **Real-time:** Streaming algorithms typically process items fast enough to keep up with the input rate, enabling real-time analytics and decision-making.

Example scenario: Monitoring the most frequent IP addresses in a high-speed network requires processing millions of packets per second. A streaming algorithm would update frequency estimates on the fly, using small memory, instead of storing all packets for later analysis.

2 Streaming Model and Memory Constraints

Streaming Model

A *data stream* is a sequence of items $\{x_1, x_2, \dots\}$ arriving one by one. A streaming algorithm processes each x_i in order, typically maintaining a small summary or "sketch" of seen data. The goal is to answer queries (e.g., frequency counts, distinct elements) using this compact summary. Often we assume x_i belongs to a large domain (size n) and the stream length N may be large or unbounded.

In the streaming model, the algorithm is given a stream of updates of the form (i, v), meaning item i appears (or its count changes by v). Two common models are:

- *Insert-only (cash-register) model:* Each update adds (or increments by 1) an element count. The stream only increases counts.
- *Turnstile (dynamic) model:* Updates can add or subtract (counts may go up and down), allowing item counts to be decremented (but usually counts stay nonnegative).

Memory is measured relative to stream length N or domain size n. Typical streaming algorithms use $O(\text{polylog}(n) + \text{poly}(1/\epsilon))$ space, which is sublinear in N and n. For some problems (e.g., exact distinct count), $\Omega(n)$ space is needed; thus streaming algorithms settle for approximation. One often expresses error in terms of parameters (ϵ, δ): an algorithm with (ϵ, δ)-guarantee returns a result within a factor of $(1 \pm \epsilon)$ of the truth with probability $1 - \delta$.

3 Sampling Techniques

Random sampling is a basic strategy in streaming. The goal is to maintain a uniform random sample of k items from the stream.

3.1 Reservoir Sampling

Reservoir sampling is a classic algorithm to sample k items uniformly from a stream of unknown length. Algorithm 1 outlines the basic method:

Algorithm 1 Reservoir Sampling ($k = 1$ shown; generalizes easily)

```
1: **Input:** Stream of items $x_1, x_2, \ldots, x_N$, sample size $k$
2: Initialize a "reservoir" array $R[1..k]$
3: **for** $i = 1$ **to** $k$ **do**
4:     $R[i] \leftarrow x_i$                              ▷ Fill reservoir with first $k$ items
5: **end for**
6: **for** $i = k + 1$ **to** $N$ **do**
7:     $j \leftarrow$ random integer in $[1, i]$
8:     **if** $j \leq k$ **then**
9:         $R[j] \leftarrow x_i$                          ▷ Replace random element in reservoir
10:    **end if**
11: **end for**
12: **return** $R$                                      ▷ Uniform random sample of $k$ items
```

The idea is that each incoming item x_i (for $i > k$) is included in the reservoir with probability k/i, ensuring that at all times each item seen so far is equally likely to be in R. A short Python snippet:

```python
import random
def reservoir_sample(stream, k):
    R = []
    for i, x in enumerate(stream, start=1):
        if i <= k:
            R.append(x)
        else:
            j = random.randint(1, i)
            if j <= k:
                R[j-1] = x
    return R
```

3.2 Other Sampling Approaches

In addition to reservoir sampling for uniform sampling, there are sampling techniques for other purposes:

- *Weighted sampling:* If items have weights, one can sample proportionally to weight (e.g., using priority sampling or weighted reservoir).
- *Min-wise hashing:* To estimate set similarity or distinct counts, one can sample items by their minimum hash values (important for locality-sensitive hashing).

4 Sketch-Based Frequency Estimation (Count-Min Sketch)

Streaming algorithms often use *sketches*: compact data structures that summarize frequencies or other statistics. The Count-Min Sketch (CMS) is a widely-used sketch for approximate frequency counts.

Count-Min Sketch

Count-Min Sketch maintains a $d \times w$ array of counters and d independent hash functions h_i. To update with an item x, we increment the counter at each row i at column $h_i(x)$. To query the frequency of x, take $\min_i \text{counter}[i, h_i(x)]$. With parameters $w = \lceil e/\epsilon \rceil$ and $d = \lceil \ln(1/\delta) \rceil$, the estimate $\hat{f}(x)$ satisfies $f(x) \le \hat{f}(x) \le f(x) + \epsilon N$ with probability $1 - \delta$.

Operations: Initialize d hash functions $h_1, \dots, h_d$ each mapping items to $\{0, \dots, w-1\}$. For each stream update of item x (count increment of 1), do:

$$\text{for } i = 1 \dots d : \ \text{count}[i][\,h_i(x)\,] += \ 1.$$

To estimate frequency of x, compute $\min_i \text{count}[i][h_i(x)]$.

Guarantee: The error in the estimate is at most ϵN (additive), with probability $1 - \delta$. The space used is $O(\frac{1}{\epsilon} \log \frac{1}{\delta})$. For example, setting $\epsilon = 0.01$, $\delta = 0.01$ yields a sketch of size about $O(100 \times \log 100) \approx O(460)$ counters.

A Python-style implementation sketch:

```python
import math, random
class CountMinSketch:
    def __init__(self, epsilon, delta):
        self.w = math.ceil(math.e/epsilon)
        self.d = math.ceil(math.log(1/delta))
        self.table = [[0]*self.w for _ in range(self.d)]
        # Initialize d pairwise-independent hash functions
        self.hashes = [(random.randrange(1«31), random.randrange(1«31))
                       for _ in range(self.d)]
    def _hash(self, x, i):
        a,b = self.hashes[i]
        return (a*hash(x)+b) % self.w
    def update(self, x, count=1):
        for i in range(self.d):
            j = self._hash(x, i)
            self.table[i][j] += count
    def query(self, x):
        return min(self.table[i][self._hash(x,i)] for i in range(self.d))
```

This class approximates item frequencies. For example, one can track word frequencies in a text stream, or network packet flows, using a CMS to answer queries like how many times did IP address A appear?, with controllable error bounds.

5 Distinct Elements and the HyperLogLog Sketch

Counting the number of distinct elements (cardinality) in a stream is another key task. Exact counting requires memory linear in the number of distinct items. The HyperLogLog (HLL) algorithm provides a probabilistic estimate of the number of distinct elements using very little memory.

HyperLogLog (Cardinality Estimation)

HLL uses $m = 2^p$ small registers (memory $O(m)$) initialized to 0. Each stream element x is hashed to a w-bit value. Split this hash into a p-bit index (register) and the remaining bits. Let $\rho(x)$ be the position of the leftmost 1-bit (counting leading zeros) in the remaining bits. Update: $\text{register}[idx] = \max(\text{register}[idx], \rho(x))$. After processing the stream, the estimate is

$$E = \alpha_m \, m^2 \Big/ \sum_{j=1}^{m} 2^{-\text{register}[j]},$$

where α_m is a bias-correction constant. The standard error of HLL is about $1.04/\sqrt{m}$.

Algorithm outline:

1. Choose a hash function that maps items to, say, 32 or 64-bit integers.
2. Use the first p bits of the hash as an index j into an array of $m = 2^p$ registers.
3. Let the remaining bits (to the right) be a value w. Compute $\rho(w)$ = number of leading zeros in w plus 1.
4. Update register $M[j] = \max(M[j], \rho(w))$.

After the stream, compute the harmonic mean of $2^{-M[j]}$ across registers and apply the formula above to get the cardinality estimate.

A simplified Python sketch:

```
import math
class HyperLogLog:
    def __init__(self, p=14):
        self.p = p
        self.m = 1 << p
        self.registers = [0]*self.m
    def _rho(self, w, max_width):
        # Count leading zeros in integer w (up to max_width bits)
        if w == 0:
            return max_width
        rho = 1
        while w & (1 << (max_width-1)) == 0:
            rho += 1
            w <<= 1
        return rho
    def add(self, x):
        x_hash = hash(x)  # in practice, use a stable 64-bit hash
        j = x_hash >> (64-self.p)              # first p bits
        w = (x_hash << self.p) & ((1<<64)-1)   # remaining bits
```

```
            self.registers[j] = max(self.registers[j],
                                    self._rho(w, 64-self.p))
    def estimate(self):
        # Compute harmonic mean of 2^{-registers[j]}
        Z = sum([2.0**(-v) for v in self.registers])
        A = 0.7213/(1 + 1.079/self.m)  # bias-correction (empirical)
        return (A * self.m * self.m) / Z
```

This implementation is simplified. In practice, one uses better hashing and handles edge cases (small or large cardinalities). But the key point is that HyperLogLog uses only $O(m)$ registers and provides a *relative* error $\approx 1.04/\sqrt{m}$. For example, with $m = 16384$ ($p = 14$), one gets error about 1%, yet can count distinct values in a stream of billions using only tens of kilobytes.

6 Sliding Window Models

Often one cares about recent items in the stream rather than the entire history. Sliding window models restrict the computation to the last N items (or last T time units). There are two common types:

- **Count-based window:** Only the W most recent elements are considered at any time.
- **Time-based window:** Only elements from the last Δt time window are considered.

A naive approach is to store all items in the window (e.g. in a queue) and update counts as items expire. This costs $O(W)$ memory. Streaming algorithms aim to approximate window queries in sublinear space.

Exponential Histograms (for counting): To approximate the count of 1's in the last W updates (bitstream), one can use *exponential histograms*. The idea is to partition the last W items into exponentially growing buckets and keep summary counts, ensuring $O(\log W)$ space while approximating the sum with small error.

Sliding Window Key Idea

For sliding windows, one can use hierarchies of buckets or sketches to summarize recent data. Each bucket might cover 2^k items, and we merge or drop buckets as new data arrives. This yields $O(\log W)$ or $O((1/\epsilon)\log W)$ space algorithms for various queries.

Simple window counter: If memory allows, one can store a queue of the last W items and maintain a running count. As each new item enters, update summary; when items exit the window, decrement summary. This exact method uses $O(W)$ space. Approximate methods (like exponential histograms or wavelet-based summaries) reduce space to $O(\log W)$ with a guaranteed $(1 + \epsilon)$-approximation of counts.

Example: To maintain a sliding window count of distinct items or frequencies, one can combine a sliding window queue with sketches like CMS or HLL: keep updating the sketch with new items, and to handle removals when items exit the window, either subtract (for linear sketches) or rebuild periodically. More advanced techniques (like *landmark windows* or *tumbling windows*) are used in stream processing systems to batch data over intervals.

7 Frequency Estimation and Heavy Hitters

A frequent task is identifying *heavy hitters*: items whose frequency in the stream exceeds a threshold (often a fraction ϕ of the total count). Related tasks include estimating item frequencies or finding the top-k frequent items.

7.1 Misra-Gries and Space-Saving Algorithms

Algorithms like Misra-Gries (MG) or Space-Saving maintain a small summary of candidate heavy hitters. The Space-Saving algorithm (Metwally et al., 2005) is popular. It keeps at most k counters; each counter stores an item, its count estimate, and an error bound. When a new item arrives:

1. If the item is already stored, increment its counter.
2. Else if fewer than k counters are used, create a new counter with count=1.
3. Otherwise, find the counter with smallest count $c_{\min}$, replace its item with the new item, and set its count to $c_{\min} + 1$ (keeping track of the error).

This guarantees that any item with true frequency $> N/(k+1)$ will appear in the table. Pseudocode outline:

Algorithm 2 Space-Saving Algorithm (Heavy Hitters)

```
Input: stream items, max counters $k$
Initialize empty table $T$ (up to $k$ entries)
for each stream item $x$ do
    if $x$ has an entry in $T$ then
        increment $T[x]$.count
    else if $|T| < k$ then
        insert $T[x] = \{\text{count} = 1, \text{error} = 0\}$
    else
        let $y$ = key with smallest count in $T$
        $\{c_{\min}, e_{\min}\} \leftarrow T[y]$
        remove $T[y]$, insert $T[x] = \{\text{count} = c_{\min} + 1, \text{error} = c_{\min}\}$
    end if
end for
return top items from $T$ sorted by count
```

A simple Python sketch (using a dictionary) is shown in Appendix A. The output of this process is a set of estimated counts for candidate heavy hitters. One can then filter items whose estimated count exceeds the threshold ϕN. The memory used is $O(k)$ counters, and the algorithm runs in $O(1)$ time per update (amortized, if a priority queue is used to find the minimum).

7.2 Other Methods for Heavy Hitters

Count-Min Sketch can also find heavy hitters: by querying each item in a candidate set (or by combining CMS with a heap to track candidates). The Misra-Gries algorithm (a simpler variant of Space-Saving) uses k counters and decrements all counters when an untracked item arrives and counters are full. Both methods ensure that any true heavy hitter (frequency $> \phi N$) is reported, with some possible false positives whose error is bounded by ϵN for appropriate parameters.

8 Stream Joins and Relational Operations

Streaming data often needs to be joined or aggregated. A *stream join* takes two (or more) input streams and outputs a combined stream of paired items. For example, joining a stream of click events with a user information stream on user ID.

Windowed Stream Join: Typically joins are performed over sliding windows. For instance, one can join two streams A and B by keeping recent items of each in a hash table (window) and, for each incoming item, probing the hash table of the other stream for matches. Pseudocode for a simple symmetric hash join:

Algorithm 3 Sliding-Window Hash Join of Streams A and B

```
Maintain hash tables $H_A, H_B$ of recent items (within window)
for each new tuple $a$ in stream A do
    Insert $a$ into $H_A$
    for each matching tuple $b$ in $H_B$ on join key do
        Output $(a, b)$
    end for
end for
for each new tuple $b$ in stream B do
    Insert $b$ into $H_B$
    for each matching tuple $a$ in $H_A$ on join key do
        Output $(a, b)$
    end for
end for
```

In practice, time-based or count-based eviction is used to remove old tuples from H_A and H_B. Join algorithms must handle out-of-order arrivals and large window sizes; often a tradeoff between latency and memory is made. More advanced joins use indexes or repartitioning of streams. Stream joins are supported in complex event processing (CEP) systems and streaming databases.

9 Distributed Streaming Systems

In real-world systems, streaming data may be processed in a distributed environment for scalability and fault tolerance. Key ideas include:

- **Partitioning:** Stream data is partitioned (sharded) across multiple worker nodes (e.g., by key hashing) so that each node processes a subset of the data.
- **Parallel algorithms:** Each worker can run local streaming algorithms (e.g., local sketches) on its partition. The results (sketches or partial aggregates) may be periodically merged.
- **Stateful stream processing:** Frameworks like Apache Flink, Spark Streaming, and Kafka Streams manage state (e.g., counts, windows) across distributed nodes. They provide exactly-once processing semantics and handle scaling and failures.
- **Sketch merging:** Many sketches (CMS, HLL, Bloom filters) are mergeable: e.g., adding corresponding counters of two Count-Min Sketches yields a sketch for the union of their data. This is useful in a distributed setting to combine results from substreams.

Modern streaming platforms (e.g., Apache Storm, Samza, Flink) offer built-in operators for common tasks (windowing, aggregations, joins) and allow custom streaming algorithms. For example, one can implement a distributed heavy-hitter detection by having each node run a Space-Saving algorithm, and then combine or reconcile the results centrally.

10 Real-World Applications

Streaming algorithms are used across domains:

- **Networking:** Detecting heavy-hitters (e.g., top talkers in network traffic) for load balancing or anomaly detection. Monitoring packet flows in routers with sketches.
- **Databases/Analytics:** Approximate query processing on data streams (e.g., real-time SQL queries on streaming data), continuous analytics dashboards. Stream processing engines (Flink, Spark Streaming) support windowed aggregations and joins on data streams.
- **Finance:** Monitoring financial transactions or stock tick streams for fraud detection or trend analysis. Estimating cardinalities (unique transactions) in real time.
- **Sensors/IoT:** Aggregating sensor readings (e.g., temperature, pressure) in real time, often on devices with very limited memory. Summarizing telemetry data using sketches.
- **Online Analytics:** Calculating metrics like page views, unique visitors, or trending topics on social media streams with sketches and counters.

11 Conclusion

Streaming algorithms enable real-time analysis of massive data flows using limited memory. By leveraging randomness and approximation, tasks like frequency estimation, distinct counting, and heavy-hitter detection can be performed in one pass over the data. Key techniques include sampling (e.g., reservoir sampling), sketches (Count-Min, HyperLogLog), and specialized algorithms (Misra-Gries, Space-Saving, Exponential Histograms). In practice, streaming computations are implemented in distributed stream processing systems, allowing scalable and fault-tolerant processing.

This document covered the fundamentals of the streaming model, essential algorithms, and practical considerations. For further reading, classic references include Muthukrishnan's survey on data streams, and recent textbook or survey chapters on streaming algorithms. The field is evolving rapidly, with connections to machine learning and real-time systems continuing to grow.

A Python Code Samples for Streaming Algorithms

Count-Min Sketch (illustration):

```python
import math, random
class CountMinSketch:
    def __init__(self, epsilon, delta):
        self.w = math.ceil(math.e/epsilon)
        self.d = math.ceil(math.log(1/delta))
        self.table = [[0]*self.w for _ in range(self.d)]
        self.hashes = [(random.randrange(1«31), random.randrange(1«31))
```

```
                   for _ in range(self.d)]
    def _hash(self, x, i):
        a,b = self.hashes[i]
        return (a*hash(x)+b) % self.w
    def update(self, x, count=1):
        for i in range(self.d):
            self.table[i][self._hash(x,i)] += count
    def query(self, x):
        return min(self.table[i][self._hash(x,i)] for i in range(self.d))
```

Space-Saving (Heavy Hitters) algorithm:

```
import heapq
class SpaceSaving:
    def __init__(self, k):
        self.k = k
        self.counters = {}  # item -> count
        self.minheap = []   # (count, item)
    def update(self, x):
        if x in self.counters:
            self.counters[x] += 1
        elif len(self.counters) < self.k:
            self.counters[x] = 1
            heapq.heappush(self.minheap, (1, x))
        else:
            # replace the min-count item
            c_min, y = heapq.heappop(self.minheap)
            del self.counters[y]
            self.counters[x] = c_min + 1
            heapq.heappush(self.minheap, (c_min+1, x))
    def topk(self):
        return sorted(self.counters.items(), key=lambda item: -item[1])
```

Reservoir Sampling:

```
import random
def reservoir_sample(stream, k):
    R = []
    for i, x in enumerate(stream, start=1):
        if i <= k:
            R.append(x)
        else:
            j = random.randint(1, i)
            if j <= k:
                R[j-1] = x
    return R
```

HyperLogLog (simplified illustration):

```
import math
class HyperLogLog:
    def __init__(self, p=14):
        self.p = p
        self.m = 1 « p
        self.registers = [0] * self.m
    def _rho(self, w, max_bits):
        if w == 0: return max_bits
        rho = 1
        while (w & (1 « (max_bits-1))) == 0:
            rho += 1
            w «= 1
        return rho
    def add(self, x):
        x_hash = hash(x) & ((1«64)-1)
        j = x_hash » (64-self.p)
        w = x_hash & ((1«(64-self.p))-1)
        self.registers[j] = max(self.registers[j], self._rho(w, 64-self.p))
    def estimate(self):
        Z = sum([2.0**(-v) for v in self.registers])
        A = 0.7213/(1 + 1.079/self.m)
        return (A * self.m * self.m) / Z
```

B Further Reading

- A. Cormode and S. Muthukrishnan, *Data Stream Algorithms and Applications*, *Foundations and Trends in Theoretical Computer Science*, 2005.
- P. Flajolet et al., "HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm," *Analysis of Algorithms*, 2007.
- G. Cormode and S. Muthukrishnan, "An improved data stream summary: the Count-Min sketch and its applications," *J. Algorithms*, 2005.
- M. Datar et al., "Maintaining stream statistics over sliding windows," *SIAM J. Comput.*, 2002.
- V. Vitter, "Random sampling with a reservoir," *ACM Trans. Math. Software*, 1985.
- J. S. Vitter, "Random sampling with a reservoir," *ACM Transactions on Mathematical Software*, 1985.
- A. Metwally, D. Agrawal, and A. El Abbadi, "Efficient computation of frequent and top-k elements in data streams," *ICDT*, 2005.
- C. G. Nevill-Manning, D. W. Fisher, "A general sense of the grammar of streams," *PODS*, 1997.
- G. H. L. Fletcher, "Approximate distinct counting algorithms in streams," *XX*, 2020.