UNIT V

Algorithms for Streaming and Big Data

Introduction

Modern data science systems deal with continuous, massive, fast-moving data known as data streams.
Examples include:

- Website logs
- Sensor data
- Clickstream data
- Telemetry data

Traditional algorithms fail because:

- Data size is too large
- Data arrives continuously
- Storing all data is impossible

Streaming and Big Data algorithms are designed to work with limited memory, one pass, and approximate answers.

Data Stream Model

Introduction

The Data Stream Model is a computational model used to process continuous, high-speed, and massive data that arrives sequentially over time.

In this model:

- Data elements arrive one by one
- The total data size may be infinite
- Storing the entire data is not feasible

Hence, algorithms must process data in real time using limited memory.

Why Data Stream Model is Needed

Traditional algorithms assume:

- All data fits in memory

- Multiple passes over data are possible

✕ These assumptions fail for:

- Web logs
- Sensor data
- Network traffic
- Clickstream data

✓ The data stream model addresses these challenges.

Key Characteristics

- Single or few passes over data
- Limited memory
- Fast processing
- Approximate answers
- Order-sensitive

Formal Definition

A data stream is a sequence:

$$x_1, x_2, x_3, \dots, x_n$$

where:

- Each element must be processed immediately
- Past elements cannot be revisited

Constraints in Data Stream Model

1. **Memory constraint** – cannot store entire stream
2. **Time constraint** – each element processed quickly
3. **Pass constraint** – usually one-pass

6. Types of Data Streams

1. **Insert-only stream**
   (elements only added)
2. **Turnstile stream**
   (elements added and removed)
3. **Sliding window stream**
   (only recent data considered)

Approximation in Data Streams

Exact answers are often impossible.

Data stream algorithms provide:

- Approximate results
- Error bounds
- High probability correctness

Common Data Stream Algorithms

Algorithm	Purpose
Count-Min Sketch	Frequency estimation
Reservoir Sampling	Random sampling
Flajolet–Martin	Distinct counting
Misra–Gries	Frequent elements
Approximate Quantiles	Percentile estimation

Applications

- Log processing
- Clickstream analysis
- Network monitoring
- IoT telemetry
- Financial tick data

Advantages

✓ Scalable
✓ Memory efficient
✓ Real-time processing

12. Limitations

✕ Approximate results
✕ Algorithm design complexity
✕ Sensitive to parameter tuning

One-Pass Algorithms

Meaning

A one-pass algorithm:

- Reads each data element only once
- Does not revisit previous elements

Why One-Pass?

- Streams cannot be stored
- Multiple passes are infeasible

Examples

- Counting approximate frequencies
- Finding maximum/minimum
- Estimating distinct elements

Advantages

✓ Low memory
✓ Fast processing
✓ Real-time capability

Count-Min Sketch (CMS)

Introduction

The Count–Min Sketch is a probabilistic data stream algorithm used to estimate the frequency of elements in massive data streams using very small memory.

It is widely used when:

- The stream is too large to store
- Exact frequency counting is infeasible
- Approximate answers with error bounds are acceptable

Problem Statement

Given a data stream:

$$x_1, x_2, x_3, \dots, x_n$$

Estimate the frequency of any element $x$using:

- One pass
- Limited memory

Key Idea

- Use multiple hash functions
- Maintain a 2D array of counters
- Hash collisions may occur, but taking the minimum reduces error

Data Structure

Structure

- A table of size $d \times w$
- d= number of hash functions (rows)
- w= number of counters per row (columns)

Each row has an independent hash function.

Working of Count–Min Sketch

Step 1: Initialization

- Initialize all counters to **0**

Step 2: Update (Insert Element)

For each incoming element x:

1. Apply each hash function $h_i(x)$
2. Increment the counter at position $(i, h_i(x))$

Step 3: Query (Estimate Frequency)

To estimate frequency of element x:

$$\text{freq}(x) = \min_{i=1}^{d} \text{table}[i][h_i(x)]$$

Why Take the Minimum?

- Collisions can only increase counts
- The minimum value gives the closest estimate to the true frequency

Example

Stream

a, b, a, c, a, b

Estimated Frequency

Element	Actual	Estimated
a	3	3
b	2	2
c	1	1

(Approximation depends on hash collisions)

Error Guarantee

With appropriate parameters:

$$\text{Estimated frequency} \leq \text{True frequency} + \epsilon N$$

With probability:

$$1 - \delta$$

Where:

- N= total stream size
- ϵ= error factor
- δ= failure probability

Parameter Selection

$$w = \lceil e/\epsilon \rceil$$
$$d = \lceil \ln(1/\delta) \rceil$$

Advantages

✓ One-pass algorithm
✓ Very memory efficient
✓ Fast updates and queries
✓ Simple implementation

Limitations

✕ Overestimates frequency
✕ Approximate results
✕ Depends on hash quality

Applications

- Network traffic monitoring
- Log processing
- Query frequency estimation
- Clickstream analysis
- Heavy hitter detection

Reservoir Sampling

What is Reservoir Sampling?

Reservoir Sampling is a randomized algorithm used to select k random items from a stream of unknown size n (or very large n) in a single pass.

- **Key Idea**: You don't know the total number of items in advance, so you can't store everything to sample later. Reservoir sampling allows you to maintain a representative sample while reading the data sequentially.
- **Applications:**
    - Selecting random logs from a large log file.
    - Randomly picking users from a stream of events.
    - Big data analytics when storing all data is impossible.

Problem Setup

- Input: A stream of items $S = s_1, s_2, s_3, \dots, s_n$
- Goal: Pick k items uniformly at random from n, without knowing n in advance.
- Output: A random sample of k items, each with probability $\frac{k}{n}$of being included.

Algorithm (for k = 1)

Let's start with the simplest case: pick **1 item** from a stream.

1. Initialize res = first item of the stream.
2. For the i-th item (i ≥ 2):
    - Pick it with probability $\frac{1}{i}$.
    - If chosen, replace res with the current item.
3. Continue until the end of the stream.
4. res will now be **1 item chosen uniformly at random** from the stream.

Why it works: Each item has a probability $1/i$of replacing the current item. By induction, every item has equal probability $1/n$at the end.

Algorithm (for k > 1)

If we want **k items**:

1. Initialize an array reservoir[1..k] with the first k items of the stream.

2. For the i-th item (i > k):
    - Pick a random index j from 1 to i.
    - If $j \leq k$, replace reservoir[j] with the i-th item.
3. Continue until the end of the stream.
4. The reservoir array now contains **k random items**, each with equal probability k/n.

Example

Suppose you want **k = 2** samples from the stream [10, 20, 30, 40].

1. Take first two items: reservoir = [10, 20].
2. Item 3 (30):
    - Pick random $j \in [1,3]$. Suppose j=2 → replace reservoir[2] → [10, 30].
3. Item 4 (40):
    - Pick random $j \in [1,4]$. Suppose j=3 → do nothing (j>k).
4. Final reservoir = [10, 30] (one of the possible combinations with equal probability).

Properties

- **Single-pass algorithm** – suitable for streaming data.
- **Memory efficient** – only need to store k items.
- **Uniform probability** – each item has exactly k/n chance of being selected.

Use Cases

1. **Big Data / Streaming**
    - Sampling logs or events when n is huge.
2. **Machine Learning**
    - Creating a random subset for training without loading all data.
3. **Online Systems**
    - Randomly selecting users for A/B testing from a live event stream.

Approximate Quantiles

What are Quantiles?

Quantiles are values that divide a dataset into equal-sized intervals. They help summarize the distribution of data.

- Median = 0.5-quantile (divides data into two equal parts)
- Quartiles = divide data into 4 equal parts
- Deciles = divide data into 10 equal parts
- Percentiles = divide data into 100 equal parts

Formally:

- The φ-quantile ($0 \leq \varphi \leq 1$) is the value x such that φ fraction of data $\leq$ x.

Why Approximate Quantiles?

In big data or streaming data, computing exact quantiles is costly because:

- Data size n may be huge or infinite (streaming data).
- Sorting the whole data to get exact quantiles is impractical.

Solution: Use approximate quantiles algorithms that:

- Provide a value close to the true quantile
- Use limited memory
- Work in one pass over the data

Problem Statement

Given:

- Stream of n numbers: x1, x2, ..., xn
- Quantile φ ($0 \leq \varphi \leq 1$)

Goal: Find a value v such that the rank of v is approximately φ*n, i.e.,

$$(\varphi - \varepsilon) * n \leq rank(v) \leq (\varphi + \varepsilon) * n$$

Where ε is the tolerance (error bound).

Basic Approaches

Sorting (Exact)

- Store all data → sort → pick quantiles
- Memory & time intensive → O(n log n)
- Not feasible for streams

Reservoir Sampling + Sorting (Approximate)

- Maintain a random sample of the stream (size k)
- Sort the sample → pick φ-quantile from sample
- Works well if k << n
- Memory-efficient, but approximation depends on sample size

GK Algorithm (Greenwald-Khanna)

One of the most popular streaming algorithms for approximate quantiles:

- Maintains a summary of the stream: list of tuples (value, g, δ)
  - value = observed number
  - g = gap (how many elements between previous tuple and this one)
  - δ = allowable error in rank
- Ensures that the rank of any value can be estimated within ε*n
- Memory usage: O(1/ε * log(ε*n)) → very efficient

Q-digest (for integers)

- Works for integer streams
- Uses tree-based summaries
- Merges counts to compress data while maintaining approximate ranks

Example (Approximate Median)

Stream: [3, 1, 4, 1, 5, 9, 2, 6]
Goal: Approximate 0.5-quantile (median)

- Keep a **sample of size 4** → [3, 1, 5, 2]
- Sort sample → [1, 2, 3, 5]

- Median ≈ 2.5 → close to exact median 3

As the stream grows, the approximation becomes closer to the true quantile.

Properties of Approximate Quantiles

- Memory-efficient → only store summary or sample
- Single-pass / streaming-friendly
- ε-approximation guarantees that the error in rank is bounded
- Can compute multiple quantiles simultaneously

Applications

1. **Big Data Analytics:** Percentiles of response times, log data, or transactions
2. **Monitoring Systems:** Latency monitoring → approximate 95th percentile
3. **Databases:** Fast percentile queries without full sorting
4. **Machine Learning:** Feature binning, normalization

Frequent Elements (Heavy Hitters)

What are Frequent Elements (Heavy Hitters)?

Frequent elements (or heavy hitters) are items in a data stream or dataset that appear more than a certain threshold.

Formally:

- Given a stream of elements: S = s1, s2, ..., sn
- A frequency threshold φ (0 < φ ≤ 1)
- Heavy hitters = items that appear more than φ*n times in the stream

Example:

- Stream: [a, b, a, c, a, b, d]
- Threshold φ = 0.3 → 0.3 * 7 = 2.1
- Heavy hitters: a (appears 3 times, > 2.1)

Why Heavy Hitters?

- In big data or streaming, storing all items to count frequencies is impractical.
- Applications:
  - Detecting popular search queries in real-time
  - Monitoring network traffic for frequently accessed IPs
  - Identifying hot items in recommendation systems

Challenges in Streams

1. **Unknown size (n)** → cannot compute exact frequency threshold in advance
2. **Memory limitation** → cannot store all distinct elements
3. **Single-pass requirement** → must process each element **once**

Algorithms for Frequent Elements

Misra-Gries Algorithm (Counter-Based)

Idea: Keep **k counters** to track potential heavy hitters.

- Initialize $k = 1/\varphi$ counters
- For each incoming element x:
  1. If x is already in counters → increment its count
  2. Else if counter has empty slot → add x with count = 1
  3. Else → decrement **all counters by 1**
- At the end, counters **may contain all heavy hitters**
- Final pass: check actual frequencies of candidates

Memory: $O(1/\varphi)$ → very efficient

Example:

- Stream: [a, b, a, c, a, b, d]
- $\varphi = 0.3 \rightarrow k = 3$ counters
- Final counters: {a:2, b:1, c:0} → heavy hitter = a

Count-Min Sketch (Approximate, Hash-Based)

- Uses hash functions and a 2D array to approximate counts

- For each element, increment counters in hash-based rows
- Query approximate count → error bounded by ε*n
- Very memory-efficient → works well for high-speed streams

Space-Saving Algorithm

- Similar to Misra-Gries but always replaces the smallest counter if a new element arrives
- Often more accurate than Misra-Gries

Properties

- **Single-pass algorithms** → suitable for streams
- **Memory-efficient** → track only potential heavy hitters, not all elements
- **Approximate counts** → some algorithms give approximate frequency with error bounds

Applications

1. **Networking:** Detect heavy IPs causing congestion
2. **Web analytics:** Find trending search terms or hashtags
3. **E-commerce:** Identify frequently sold products
4. **Fraud detection:** Detect unusual transaction patterns

Scalable MapReduce-like Algorithm Design

What is MapReduce?

A programming model for processing huge datasets in parallel.

Phases

1. **Map** – process input data and produce key-value pairs
2. **Shuffle** – group values by key
3. **Reduce** – aggregate results

Why MapReduce?

- Handles petabytes of data
- Fault tolerant
- Scalable

Examples

- Word count
- Log aggregation
- Click analysis

Applications

1.Log Processing

What it is:

- Systems and applications generate logs continuously: errors, user actions, server metrics.
- Logs are often high-volume and unbounded, so storing all logs is impractical.

How streaming algorithms help:

Algorithm	Application in Logs
Reservoir Sampling	Select a random subset of log entries for inspection or debugging without storing the entire log.
Approximate Quantiles	Monitor response times, request sizes, or latency. For example, approximate 95th percentile response time helps detect performance bottlenecks.
Frequent Elements (Heavy Hitters)	Identify the most frequent error codes or IPs causing failures in the system.

Example:

- Detect top 10 most frequent 500-error pages in a web server log in real-time.

2. Clickstream Analysis

What it is:

- Clickstream = sequence of user actions (clicks, page visits) on a website or app.

- Helps understand user behavior and improve UX or personalization.

How streaming algorithms help:

Algorithm	**Application in Clickstreams**
Reservoir Sampling	Randomly select user sessions for A/B testing or behavior analysis.
Approximate Quantiles	Measure time spent on pages or scroll depth: approximate median or 90th percentile helps identify anomalies.
Frequent Elements (Heavy Hitters)	Track the most clicked pages, buttons, or search queries in real-time.

Example:

- Detect trending products or pages that users are clicking most frequently today.

3. Telemetry / Sensor Data

What it is:

- Telemetry = continuous stream of measurements from devices (IoT, satellites, vehicles).
- Data is high-velocity, high-volume, and often unbounded.

How streaming algorithms help:

Algorithm	**Application in Telemetry**
Reservoir Sampling	Sample sensor readings to detect unusual events without storing all raw data.
Approximate Quantiles	Monitor temperature, pressure, or latency metrics: approximate percentiles help detect anomalies.
Frequent Elements (Heavy Hitters)	Detect sensors that frequently report unusual values, or most common events in a time window.

Example:

- In a smart city, detect the roads with the most frequent traffic congestion events in real-time using