Algorithms for Data Science: Data Streams

Pablo Mollá Chárlez

October 5, 2024

Contents

1 Introduction to Data Streams

Data streams are sequences of data that arrive continuously and in-real time, requiring immediate processing or analysis. Unlike traditional databases that assume datasets are available in their entirety (offline), data streams are often available only online, meaning the data arrives incrementally and is not stored permanently. This is common in scenarios like:

- **Twitter status update**: New tweets are constantly being posted in real-time.
- **Queries on search engines**: Devices submit new queries continuously.
- **Data from sensor networks**: Devices such as weather sensors or IoT devices generate real-time data streams.
- **Telephone calls of IP packets on the Internet**: Both involve data that must be processed instantly.
- **High-speed trading data**: Financial markets generate streams of transactions data at a rapid pace.

1.1 Data Stream Model

A stream of data can be represented as an infinite sequence of items $S = \{i_1, i_2, \ldots, i_k, \ldots\}$. Data streams are characterized by being:

- **Infinite**: The streams can grow indefinitely as new data arrives.
- **Non-stationary**: The statistical properties of the stream can change over time.

The primary challenge with data streams is how to ask queries on the stream in 2 forms:

1. **Standing Queries**: The continuously monitor the stream for certain conditions.
2. **Ad-hoc Queries**: These are made on-demand for specific information about the stream.

However, some restrictions are forced:

- **Storage Space**: It's impossible to store all items in an infinite stream.
- **Processing time**: The stream must be processed in real-time; once data is passed, it's lost forever unless processed immediately.

The difference between sampling items (Reservoir Sampling) and filtering items (Bloom Filter) from a stream lies in purpose, functionality, and the nature of how they handle elements in the stream:

- **Reservoir Sampling**: The goal of reservoir sampling is to select a random sample of size s s from an infinite or unknown-sized stream of elements. It ensures that each element in the stream has an equal probability of being included in the final sample. It's used when you want to maintain a representative sample from a large stream with memory constraints.
- **Bloom Filter**: A Bloom Filter is used for efficient membership testing. It answers the question: "Have I seen this item before?" by checking if an element is possibly in a set (but with false positives) or definitely not. It helps in filtering out elements that aren't in a predefined set with very low memory requirements. It is primarily used to prevent reprocessing or duplicate detection

2 Sampling Items from a Stream

Objective: The goal is to keep a proportion **p** of the items in a stream. For instance, we might want to keep 1 in every 10 elements from a stream. Let's consider that we are working with queries on search engines.

- **First Solution**: One straightforward approach is to randomly decide for each element whether to keep it. For instance, we generate a random number between 0 and 9 for each item, and we only keep the item if we generate a 0. However, this clearly leads to an inaccurate estimation, as it doesn't correctly capture the true proportion of singletons or duplicates (of queries) in the stream. (Proof on Slides)
- **Better Solution**: To improve the accuracy, it's better to sample based on users rather than individual queries. That way, for a given sampled user, we would retain all of their queries, ensuring a more accurate representation of their behavior.

  This can be done by hashing user identifiers to integers and using the hash value to decide which users to sample. Now, assume we need to keep a sample of exactly s items (due to memory limitations). The goal is that each item in the stream should be in the sample with equal probability.

Let's see an example to understand it.

2.1 Real-World Scenario

Let's consider a real-world scenario where we are analyzing search queries submitted by users to a search engine. The stream consists of tuples in the form (user, query, timestamp), representing each user's query and the time it was made. We want to sample a proportion of users to retain all their queries, instead of sampling individual queries. Imagine the following sequence of tuples arriving in the stream:

$$
\begin{aligned}
(UserA, "howtobakeacake", 09:00)\\
(UserB, "bestpizzarecipe", 09:01)\\
(UserA, "chocolatecakeingredients", 09:02)\\
(UserC, "weatherinNewYork", 09:03)\\
(UserB, "howtomakepizzadough", 09:04)\\
(UserA, "cakefrostingideas", 09:05)\\
(UserC, "NYweatherforecast", 09:06)
\end{aligned}
$$

Instead of sampling queries, we aim to sample users, so that all queries from a sampled user are kept. This can be done by hashing user identifiers (like usernames or user IDs) into integers and using the hash value to decide whether to keep the user.

Steps:

1. Hash user identifiers: We will apply a hash function to each user (e.g., hash(user) mod 100).
2. Set sampling threshold: Let's say we want to sample 10% of the users. We retain a user if the hash of their identifier is less than 10 (out of 100).
   - For example, hash(UserA) mod 100 = 5, hash(UserB) mod 100 = 35, and hash(UserC) mod 100 = 15.
3. Sampling logic: Based on the hash values:
   - UserA is sampled because $5 < 10$. Therefore, all of UserA's queries will be kept in the sample.
   - UserB is not sampled because $35 \geq 10$. Hence, none of UserB's queries will be included.
   - UserC is not sampled because $15 \geq 10$. None of UserC's queries will be included either.

4. Resulting Sample: After processing the stream, the only sampled user is UserA, so the sample would look like this:

$$(UserA, "howtobakeacake", 09:00)$$
$$(UserA, "chocolatecakeingredients", 09:02)$$
$$(UserA, "cakefrostingideas", 09:05)$$

The advantages of such procedure include:

1. **Complete user behavior**: By sampling users instead of individual queries, we capture all the queries from a sampled user, allowing for more accurate analysis of user behavior patterns.
2. **Equal probability**: Each user is sampled independently with equal probability (based on the hash function), ensuring fairness.
3. **Efficient memory usage**: We can control the memory usage by adjusting the sample size based on user sampling, rather than tracking individual queries.

2.2 Reservoir Sampling Algorithm

The Reservoir Sampling Algorithm is a popular approach to maintain a random sample of s elements from a stream, ensuring each element is included with equal probability.

2.2.1 Algorithm

1. **Initialization**: Store the first s elements from the stream.
2. **Selection**: For each new element n (where $n > s$):
   - With probability $\frac{s}{n}$, keep the new element, otherwise discard it.
   - If the element is kept, replace a randomly chosen element in the current sample.

2.2.2 Proof

Let's prove it by induction by claiming as follows: The algorithm ensures that after processing n items, each element has been included in the sample with probability $\frac{s}{n}$.

- Base case: After seeing the first s elements, they are in the sample with probability $s/s = 1$. All original elements are included in the reservoir sample.
- Inductive hypothesis: After processing n elements, each element is in the sample with probability $\frac{s}{n}$.

  For the inductive step, when element $n+1$ arrives, the probability that the new element is kept is $\frac{s}{n+1}$. We know that the probability of being in the reservoir sample, under the inductive hypothesis, is $\frac{s}{n}$ and the probability of leaving due to the new arrival of the element $n+1$ is $\frac{n}{n+1}$, therefore the probability that any previously sampled element is retained in the sample at time $n+1$ is:

$$\frac{s}{n} \times \frac{n}{n+1} = \frac{s}{n+1}.$$

  This ensures that at each step, every element has an equal chance of being in the sample, which guarantees the desired property of the algorithm.

2.2.3 Python Implementation

```python
import random

def selectKItems(stream, n, k):
    # Initialize reservoir with the first k elements
    reservoir = stream[:k]

    # Replace elements with a decreasing probability
    for i in range(k, n):
        j = random.randrange(i + 1)
        if j < k:
            reservoir[j] = stream[i]

    return reservoir

# Driver Code
if __name__ == "__main__":
    stream = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
    k = 5
    result = selectKItems(stream, len(stream), k)
    print("Randomly selected items:", result)
```

3 Filtering Items from a Stream

A Bloom Filter is a probabilistic data structure used for efficiently checking whether an element might belong to a set or is definitely not in the set. It uses hashing and a fixed-size bit array to achieve memory-efficient membership tests.

However, a Bloom Filter can have false positives (incorrectly indicating whether an element is in the set) but guarantees no false negatives (if it says an element is not in the set, it is definitely not).

3.1 Algorithm

1. **Bit Array Initialization**

   - **Array** B: The Bloom Filter starts with a bit array B of size n bits, all initialized to 0.

   $$B = [0, 0, 0, 0, ..., 0] \quad (n \text{ bits})$$

   - **Hash Functions** $h_1, h_2, \ldots, h_k$: A collection of k independent hash functions is used, each mapping elements to an index in the bit array B. Each hash function takes an input element and maps it to one of the n bit positions.

   $$h_i(x) : \text{element} \rightarrow \{0, 1, 2, ..., n-1\}$$

2. **Adding Elements to the Bloom Filter**

   When adding an element (key) x to the Bloom Filter:

   - Each of the k hash functions is applied to x, producing k positions in the bit array B.
   - The bits at these positions are set to 1. If a bit is already set to 1 by a previous element, it remains 1.

   $$B[h_1(x)] = 1, \quad B[h_2(x)] = 1, \quad ..., \quad B[h_k(x)] = 1$$

Let's say we are inserting an item x into a Bloom Filter with $n = 10$ bits and $k = 3$ hash functions. Assume the hash functions give the following indices for x:

$$h_1(x) = 2$$
$$h_2(x) = 4$$
$$h_3(x) = 7$$

After inserting x, the bit array looks like this:

$$B = [0, 0, 1, 0, 1, 0, 0, 1, 0, 0]$$

3. **Checking Membership (Querying the Bloom Filter)**

   To check if an element y is in the Bloom Filter:

   - Apply the same k hash functions to y, producing k bit positions.
   - Check the bits at those positions in the bit array:
     - If all the bits at the hashed positions are 1, the Bloom Filter returns "might be in the set" (there is a possibility the item is in the set, but there could be a false positive).
     - If any bit is 0, the Bloom Filter returns "definitely not in the set" (the item is not in the set).

   Suppose we want to check if y is in the filter:

$$h_1(y) = 3$$
$$h_2(y) = 7$$
$$h_3(y) = 9$$

   We check bits 3, 7, and 9 in the bit array. If all are 1, we say "maybe in the set"; if any bit is 0, we say "definitely not."

Comment: Bloom Filters do not support deletion directly, once a bit is set to 1, it cannot be reset to 0, since this would affect other elements that also hash to that bit. There are variations like counting Bloom filters that can handle deletions by maintaining a count of how many elements have set each bit, but the basic Bloom Filter does not support this.

3.2 Probability of False Positives

A false positive occurs when the Bloom Filter incorrectly reports that an element might be in the set, even though it was never added. This happens because multiple elements can set the same bits in the bit array, causing hash collisions. The probability of a false positive P_{fp} is defined as follows:

$$\boxed{P_{fp} = \left(1 - e^{-\frac{km}{n}}\right)^k}$$

Where:

P_fp : Probability of a false positive.
m : Number of elements inserted into the Bloom Filter.
n : Size of the bit array (number of bits).
k : Number of hash functions.
e : Euler's number (approximately 2.71828).

3.3 Optimal Number of Hash Functions

The optimal number of hash functions k that minimizes the probability of false positives is given by:

$$\boxed{k = \frac{n}{m} \cdot ln(2)}$$

Where:

k : Optimal number of hash functions.
n : Size of the bit array (number of bits).
m : Number of elements inserted into the Bloom Filter.

4 Counting Distinct Items from a Stream

In various real-world applications, the need arises to count how many distinct elements are observed in a massive data stream. Traditional methods may be inefficient due to the vast size of the data. Here are a few applications:

- Web analytics Counting how many distinct words are on webpages, useful for spam detection or keyword analysis.
- Retail analytics Tracking how many distinct products were sold in the last week.
- Astronomy Counting how many new stars are discovered in space over time.

4.1 The Flajolet-Martin (FM) Approach

The Flajolet-Martin (FM) algorithm is a probabilistic algorithm used for estimating the number of distinct elements (cardinality) in a large data stream, especially when storing all the data is impractical due to space limitations. It provides a memory-efficient way to estimate the count of distinct items in streaming data, which makes it ideal for applications like web analytics, network traffic monitoring, and database systems.

- Why not count the distinct items directly?
  For large datasets, counting distinct elements exactly requires either storing the entire dataset or using a large amount of memory. This is inefficient when the dataset is huge, or when you're dealing with continuous streams of data that you cannot store entirely. Instead, Flajolet-Martin leverages hash functions and bit patterns to estimate the count using much less memory.
- Main Drawback:
  The expectation $E[2^R]$ can sometimes become very high, leading to overestimation of the distinct count.
- Solution:
  To mitigate this overestimation problem, the Flajolet-Martin approach is extended with multiple estimators using m different hash functions.

4.1.1 FM Algorithm

The three main steps of the Flajolet-Martin algorithm are:

1. **Pick a hash function** : Map each item in the stream to a binary value with at least $\log_2 N$ bits using a hash function.

2. **Calculate trailing zeros** : For each stream item s, compute $r(s)$, the number of trailing zeros in its hash value.

3. **Track maximum R** : Keep $R = \max r(s)$, the largest number of trailing zeros observed across the stream.

4. **Estimate the number of distinct elements** : The number of distinct elements is 2^R.

The benefits of keeping only R for each hash function are:

- **Memory efficiency**: Storing just R requires very little memory.
- **Scalability**: You can use as many hash functions as memory permits to improve accuracy.
- **Time trade-off**: While using more hash functions improves accuracy, it increases computation time for hashing and maintaining estimates (e.g., averages/medians).

Now, let's say we want to estimate the total number of distinct elements across the M streams, after obtaining the maximum R value for each stream:

- Estimate per stream: For each stream, you have an estimate of 2^R distinct elements.
- Combine estimates: To estimate the total number of distinct elements across all 10 streams, you can't simply sum the individual 2^R values. This is because some elements may appear in multiple streams, leading to overcounting.

  A common approach is to use averaging or more sophisticated statistical methods like taking the median of the R values across all streams, then using that median to estimate the total number of distinct elements.

  The final distinct count is typically estimated as:

$$\text{Total distinct elements} \approx 2^{\text{median}(R_1, R_2, \ldots, R_{10})}$$

  where $R_1, R_2, \ldots, R_{10}$ are the maximum R values for each stream.

This approach reduces the bias caused by overlapping elements in the streams.

4.1.2 Basic Example: Single Hash Function

Here's a simple example to understand the main idea of the Flajolet-Martin approach. Let's say we have a small data stream of 4 distinct items:

Items: ['apple', 'banana', 'orange', 'apple', 'grape', 'banana']

Our goal is to estimate the number of distinct items, which in this case are *apple*, *banana*, *orange*, and *grape*, so the true distinct count is 4.

1. Step 1: Hash the Items We apply a hash function to each item. For simplicity, let's pretend we hash the items to these numbers (in binary):

$$\begin{aligned}
&\text{hash('apple')} = 101000 \text{ (40 in decimal)} \\
&\text{hash('banana')} = 1100000 \text{ (96 in decimal)} \\
&\text{hash('orange')} = 10000 \text{ (16 in decimal)} \\
&\text{hash('grape')} = 101000 \text{ (40 in decimal)}
\end{aligned}$$

2. Step 2: Count Trailing Zeros Next, we count the number of trailing zeros (the trailing zeros are a sequence of 0 in the decimal representation of a number, after which no other digits follow) in each binary hash:

$$\text{hash('apple')} = 101000 \text{ has 3 trailing zeros.}$$
$$\text{hash('banana')} = 1100000 \text{ has 5 trailing zeros.}$$
$$\text{hash('orange')} = 10000 \text{ has 4 trailing zeros.}$$
$$\text{hash('grape')} = 101000 \text{ has 3 trailing zeros.}$$

3. Step 3: Track the Maximum Trailing Zeros We keep track of the maximum number of trailing zeros, which comes from the word 'banana', across all hashed values:

$$\boxed{\text{Maximum number of trailing zeros} = 5}$$

4. Step 4: Estimate the Distinct Count The Flajolet-Martin algorithm estimates the number of distinct items as:

$$\text{Estimate} = 2^{\text{Max Trailing Zeros}} = 2^5 = 32 \text{ distinct elements}$$

In this example, the algorithm overestimates the number of distinct items because we only had 4 distinct items. The estimate is 32, which shows that with a small number of hash functions or items, the algorithm can be imprecise. But as the stream size grows and with multiple hash functions, the estimate becomes much more accurate.

4.1.3 Refined Example: Multiple Random Hash Functions

Let's say we have 5 hash functions, and for each one, the maximum number of trailing zeros (out of 1000 hashed values from the stream) is as follows:

- $h1 = 10$ trailing zeros - $h2 = 9$ trailing zeros - $h3 = 11$ trailing zeros - $h4 = 12$ trailing zeros - $h5 = 9$ trailing zeros

1. Average of the Estimators

As mentioned, the first estimation using the average is:

$$\text{Average estimation} = \frac{2^{10} + 2^9 + 2^{11} + 2^{12} + 2^9}{5} = \frac{1024 + 512 + 2048 + 4096 + 512}{5} = \frac{8192}{5} = 1638.4$$

2. Median of the Estimators

For the median method, we take the powers of 2 for each trailing zero count and then find the median of those values. The estimations based on trailing zeros:

$$h1 = 2^{10} = 1024$$
$$h2 = 2^9 = 512$$
$$h3 = 2^{11} = 2048$$
$$h4 = 2^{12} = 4096$$
$$h5 = 2^9 = 512$$

Now, sort these estimations:

$$[512, 512, 1024, 2048, 4096]$$

The median value is the middle one, which is 1024.

3. Group median method

   For the group median method, let's divide the 5 estimators into groups. Since 5 is a small number, we can group them into two groups:

$$\textbf{Group 1} : 2^{10} = 1024, 2^9 = 512, 2^{11} = 2048$$
$$\textbf{Group 2} : 2^{12} = 4096, 2^9 = 512$$

   Now, compute the median of each group:

$$\textbf{Median of Group 1} : [512, 1024, 2048] \Longrightarrow 1024.$$
$$\textbf{Median of Group 2} : [512, 4096] \Longrightarrow \frac{512 + 4096}{2} = \frac{4608}{2} = 2048.$$

   Finally, we take the average of these two medians:

$$\text{Group median average} = \frac{1024 + 2048}{2} = 1536$$

4. Summary of Estimates

$$\textbf{Average estimation} : 1638.4$$
$$\textbf{Median estimation} : 1024$$
$$\textbf{Group median average} : 1536$$

5 Estimating the Surprise Number of a Stream

Both the Flajolet-Martin (FM) algorithm and the Alon-Matias-Szegedy (AMS) algorithm tackle a common challenge in stream processing: estimating certain properties of a data stream (like distinct elements or moments) while limiting memory usage. In large data streams, we often need to compute statistical properties like:

1. **Distinct items in the stream** (0th moment) $\Longrightarrow$ FM Algorithm.
2. **Length of the stream** (1st moment) $\Longrightarrow$ AMS Algorithm.
3. **The surprise number** (2nd moment) $\Longrightarrow$ AMS Algorithm.

The AMS algorithm is used to estimate the **2nd moment**, which measures how skewed or uneven the distribution of distinct elements is. This is important when we want to understand how the counts of elements vary.

5.1 AMS Algorithm: Surprise Number

1. **Randomly pick a position i in the stream**
2. **Store the element at position i**

   Set $X.val = s_i$, the value of the element at position i.

   Set $X.c = 1$, which tracks the count of how many times this element appears in the stream.

3. **Update the count** As the stream progresses, every time the value $X.val$ is encountered again in the stream, increment $X.c$ (the count of that element).
4. **Estimating the Second Moment** After processing the stream, the estimate of the second moment (surprise number) based on the sampled variable X is:

$$\text{Estimate} = n \times (2X.c - 1) \text{ where: } n \text{ is the total length of the stream}$$

5.2 General Formula for the k-th Moment

The AMS algorithm can estimate any moment k of a stream. The general estimator for the k-th moment is:

$$\text{Estimate} = n \times \left(c^k - (c-1)^k\right)$$

Where:

n is the length of the stream

c is the count of the sampled element

5.3 Refining the Estimate with Multiple Samples

Instead of using a single variable X, the accuracy of the estimate can be improved by using **k** different variables $X_1, X_2, \ldots, X_k$, each independently sampled using the same method as above. The final estimate of the second moment is then the **average of the estimates** from these k variables:

$$\text{Final Estimate} = \frac{n}{k} \sum_{i=1}^{k} (2X_i.c - 1)$$

This way, the AMS algorithm gives a probabilistic estimate of the second moment without needing to store all distinct elements and their counts in memory. By averaging across multiple samples, the accuracy of the estimate improves.

5.4 Interpretation

To determine whether the estimated second moment is "big" or "small," we need to compare it against the first moment.

- If Second Moment $\gg$ First Moment Certain elements dominate the stream, appearing many more times than others.
- If Second Moment $\approx$ First Moment This suggests a more even or uniform distribution.
- Perfect Uniform Distribution Every element appears the same number of times, the second moment would be equal to the first moment.
- Extreme Skew One element dominates the stream (appears much more frequently than others), the second moment becomes very large compared to the first moment.

5.5 Refined Example: Multiple Independent Variables

Let's understand the AMS Algorithm refined version with a short example, where we use multiple samples in order to estimate the second moment. Let's consider a stream S with the following elements:

$$S = \{a, b, a, a, c, b, b, d, a, b, c\}$$

1. Step 1: Random Sampling with $k = 3$ We choose 3 random positions in the stream and track the element and its count at those positions.

Position 2: $X_1.val = b$, initialize $X_1.c = 1$.
Position 5: $X_2.val = c$, initialize $X_2.c = 1$.
Position 7: $X_3.val = b$, initialize $X_3.c = 1$.

2. Step 2: Count Occurrences of Sampled Elements As we continue processing the stream, we update the counts for each sampled element whenever it reappears.

$$X_1.val = b \text{ appears 4 times in total, so } X_1.c = 4.$$
$$X_2.val = c \text{ appears 2 times in total, so } X_2.c = 2.$$
$$X_3.val = b \text{ appears 4 times in total, so } X_3.c = 4.$$

3. Step 3: Estimate the Second Moment For each sample, compute the second moment estimate ($n = 11$):

$$X_1 : n \times (2 \times 4 - 1) = 11 \times 7 = 77$$
$$X_2 : n \times (2 \times 2 - 1) = 11 \times 3 = 33$$
$$X_3 : n \times (2 \times 4 - 1) = 11 \times 7 = 77$$

4. Step 4: Refined Estimate by Averaging Average the estimates from the 3 samples:

$$\text{Final Estimate} = \frac{77 + 33 + 77}{3} = 62.33$$

Thus, the refined estimate of the second moment for this stream is approximately 62.33, giving us an indication of how skewed or uneven the distribution of element frequencies is. The first moment is simply the length of the stream, in our case:

$$\text{First Moment} = \text{n} = 11 \quad \& \quad \frac{\text{Second Moment}}{\text{First Moment}} = \frac{62.33}{11} \approx 5.67$$

Since the second moment is about 5.67 times larger than the first moment, it indicates that the distribution is somewhat skewed - certain elements (like a and b) appear significantly more often than others (like c and d).

6 Sliding Windows in Streams

As we explore the challenges of processing data streams, an important aspect arises: the need to query only the last N elements of a stream. This scenario is common in applications where recent data is more relevant than older data, such as in transaction logs (e.g., sales transactions or ad clicks). However, managing the storage of entire streams can be impractical, especially when dealing with multiple streams or very large datasets.

The discussion around the Flajolet-Martin (FM) algorithm and the Alon-Matias-Szegedy (AMS) algorithm provided insights into estimating distinct elements and statistical moments in streaming data with limited memory. Similarly, when dealing with sliding windows over streams, we need efficient methods to summarize the most recent data without storing everything in memory.

This leads us to the Datar-Gionis-Indyk-Motwani (DGIM) method, which is specifically designed for efficiently maintaining counts over sliding windows. The DGIM method focuses on summarizing the stream using exponential windows to capture recent data in a memory-efficient manner.

6.1 Datar-Gionis-Indyk-Motwani (DGIM) Method

The main idea of the DGIM method is to use buckets that summarize regions of the stream with exponentially increasing sizes. This allows the algorithm to efficiently keep track of the number of **1s** (or **significant events**) in the stream while requiring very little memory.

Key Features

- **Exponential Windows**: The buckets grow exponentially in size, following a sequence like $1, 1, 2, 4, 8, 16, \ldots$. Each bucket contains the **timestamp** and the **count of 1s**.

- **Storage**: The DGIM method requires only $O(\log^2 N)$ bits for storage (specifically, $O(\log N)$ counts of $\log_2 N$ bits).
- **Easy Updates**: The method allows for straightforward updates as new bits arrive in the stream.
- **Error**: The error in the count is constrained; if 1s are relatively evenly distributed, the error is no more than 50%. However, if all 1s are located in the unknown area, the error can be unbounded.

6.2 DGIM Algorithm

6.2.1 Updating Buckets

When a new bit arrives:

1. If it's a **0**, no changes are made.
2. If it's a **1**, a new bucket of size 1 is created.
   - If there are 3 buckets of size 1, the oldest two are merged into a new bucket of size 2.
   - This merging continues recursively for larger bucket sizes.

6.2.2 Queries

To perform a query using the DGIM method:

1. Sum the sizes of all buckets except the last one.
2. Add half the size of the last bucket to account for the unknown proportion of the last window in the total N.

6.3 Example

Let S be a binary stream with sliding window size $N = 8$, representing significant events **1s** and non-events **0s**.

$$\text{Stream S: } [1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0]$$

1. Step 1: Process the Stream Initially, there are no buckets.
   (a) **Value = 1**: Create a new bucket of size 1 $\Longrightarrow$ Buckets: [(1, timestamp)]
   (b) **Value = 0**: No change $\Longrightarrow$ Buckets: [(1, timestamp)]
   (c) **Value = 1**: Create a new bucket of size 1 $\Longrightarrow$ Buckets: [(1, timestamp), (1, timestamp)]
   (d) **Value = 1**: Create a new bucket of size 1 $\Longrightarrow$ Buckets: [(1, timestamp), (1, timestamp), (1, timestamp)]. As we have 3 buckets, we combine the oldest two buckets blue and red (both size 1) into a new bucket of size 2 which inherits the timestamp for the red bucket $\Longrightarrow$ Buckets: [(2, timestamp), (1, timestamp)]
   (e) **Value = 0**: No change $\Longrightarrow$ Buckets: [(2, timestamp), (1, timestamp)]
   (f) **Value = 0**: No change $\Longrightarrow$ Buckets: [(2, timestamp), (1, timestamp)]
   (g) **Value = 1**: Create a new bucket of size 1 $\Longrightarrow$ Buckets: [(2, timestamp), (1, timestamp), (1, timestamp)]. As we have 3 buckets, we combine the oldest two buckets red and orange (size 2 and 1) into a new bucket of size 3 which inherits the timestamp for the orange bucket $\Longrightarrow$ Buckets: [(3, timestamp), (1, timestamp)]
   (h) **Value = 0**: No change $\Longrightarrow$ Buckets: [(3, timestamp), (1, timestamp)]

(i) **Value = 1**: Create a new bucket of size 1 $\Longrightarrow$ Buckets: [(3, timestamp), (1, timestamp), (1, timestamp)]. As we have 3 buckets, we combine the oldest two buckets orange and purple (size 3 and 1) into a new bucket of size 4 $\Longrightarrow$ Buckets: [(4, timestamp), (1, timestamp)]

(j) **Value = 1**: Create a new bucket of size 1 $\Longrightarrow$ Buckets: [(4, timestamp), (1, timestamp), (1, timestamp)]. As we have 3 buckets, we combine the oldest two buckets purple and cyan $\Longrightarrow$ Buckets: [(5, timestamp), (1, timestamp)]

(k) **Value = 0**: No change $\Longrightarrow$ Buckets: [(5, timestamp), (1, timestamp)]

2. Step 2: Remove Old Buckets If any bucket's timestamp exceeds $N = 8$, remove it. Let's consider 2 different possible timestamp values to understand what would happen. Let's say that timestamp $= 7$ and timestamp $= 10$. In such case, as $7 < 8$, the bucket is valid and stays in the list of buckets, however, as $10 > 8$ we remove this bucket, leading to a new final bucket list: [(5, timestamp)]

3. Step 3: Query the Count To query the number of 1s in the last N elements, we sum the sizes of all buckets except the last one and add half the size of the last bucket.

   - Sum of Sizes (without last bucket): 0 (we just have one bucket and is the last one)
   - Last Bucket: Size = 5 (we assume we don't know the proportion) and we add half of the last bucket size $= \frac{1}{2}$.
   - Total Count Estimate: Estimated count $= 0 + \frac{5}{2} = 2.5$.

In this case, the estimated count of 1s is approximately 2.5, with an error margin that can be controlled by managing bucket sizes effectively. The error can not be determined in this scenario because we only have 1 bucket, however generally speaking, the error in the count estimate is at most 50% of the size of the last bucket, meaning that the maximum error is $\frac{1}{2} \cdot 5 = 2.5$, leading to a True Count $\in [2.5 - 2.5, 2.5 + 2.5] = [0, 5]$.