UNIT IV

CHAPTER 4

Mining Big Data Streams

University Prescribed Syllabus

The Stream Data Model : A DataStream-Management System, Examples of Stream Sources, Stream Queries, Issues in Stream Processing. Sampling Data in a Stream : Sampling Techniques, Filtering Streams: The Bloom Filter Counting Distinct Elements in a Stream : The Count-Distinct Problem, The Flajolet-Martin Algorithm, Combining Estimates, Space Requirements. Counting Ones in a Window : The Cost of Exact Counts, The Datar-Gionis-Indyk, Motwani Algorithm, Query Answering in the DGIM Algorithm.

Self-learning Topics : Streaming services like Apache Kafka/Amazon Kinesis/Google Cloud DataFlow. Standard spark streaming library. Integration with IOT devices to capture real time stream data.

4.1 THE STREAM MODEL

- As data arrives in various streams, it's important to stream the data as the data which arrives cannot be stored immediately and the probability of losing the data is more.
- To mine the data, we need to create the sample of stream data and to filter the stream to eliminate most of the "undesirable" data elements.

- To summarize the large stream of data, consider the fixed length window consisting of last n elements.
- The data stream model is considered for the management of data, along with the stream queries and issues in stream processing
- In recent years, advances in hardware technology have facilitated the ability to collect data continuously. Simple transactions of everyday life such as using a credit card, a phone or browsing the web lead to automated data storage. Similarly, advances in information technology have lead to large flows of data across IP networks.
- In many cases, these large volumes of data can be mined for interesting and relevant information in a wide variety of applications. When the volume of the underlying data is very large, it leads to a number of computational and mining challenges :
- With increasing volume of the data, it is no longer possible to process the data efficiently by using multiple passes. Rather, one can process a data item at most once.
- This leads to constraints on the implementation of the underlying algorithms. Therefore, stream mining algorithms typically need to be designed so that the algorithms work with one pass of the data.
- In most cases, there is an inherent temporal component to the stream mining process. This is because the data may evolve over time.
- This behavior of data streams is referred to as temporal locality. Therefore, a straightforward adaptation of one-pass mining algorithms may not be an effective solution to the task. Stream mining algorithms need to be carefully designed with a clear focus on the evolution of the underlying data.

- Another important characteristic of data streams is that they are often mined in a distributed fashion. Furthermore, the individual processors may have limited processing and memory.
- Common examples of streaming data sources include :
  - IoT sensors
  - Server and security logs
  - Real-time advertising platforms
  - Click-stream data from apps and websites

4.1.1 Data Stream Mining Characteristics

- **Continuous Stream of Data :** High amount of data in an infinite stream. we do not know the entire dataset
- **Concept Drifting :** The data change or evolves over time
- **Volatility of data :** The system does not store the data received (Limited resources). When data is analysed it's discarded or summarised.

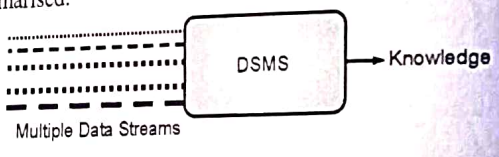


Fig. 4.1.1 : Data stream management system

Next, is discussed how to analyse Data Streams. Data-Based Techniques rely on analysing a representative subset of data. This technique also is used as pre-processing for Data Stream algorithms. On the Other Hand, Mining Techniques are enhanced versions of traditional Data Mining Algorithms.

4.2 DATA BASED TECHNIQUES

- **Sampling** is based on selecting subset of data uniformly distributed.
- **Reservoir Sampling** (Fixed Size Sample) This algorithm sample a subset m of the data from the stream. After the i^{th} item is chosen with probability 1 / i and if the i element is already sampled its randomly replace a sampled item.
- **Min-Wise Sampling** is based on assigning a random γ in range 0 to 1 to a subset of samples m. When the system retrieves m elements, we select the sample with the minimum γ.
- **Sketching** is based on reducing the dimensionality of the dataset. Sketch is based on performing a linear transformation of the data delivering a summarization of the Stream. See also Count-min Sketch.
- **Approximation Techniques** are based on maintaining only a subset of data and discarding previous data (sliding windows).
- **Sequence based :** The size of the window depends on the number of observations. The window store γ elements and when a new element arrives, the last element is removed if the window is full.
- **Timestamp based :** The size of the window depends on time. The window is bounded in instant T_n and T_{n+1} and holds the elements received in this period.
- At a high level, MapReduce breaks input data into fragments and distributes them across different machines. The input fragments consist of key-value pairs. Parallel map tasks process the chunked data on machines in a cluster. The mapping output then serves as input for the reduce stage. The reduce task combines the result into a particular key-value pair output and writes the data to HDFS.

- The Hadoop Distributed File System usually runs on the same set of machines as the MapReduce software. When the framework executes a job on the nodes that also store the data, the time to complete the tasks is reduced significantly.

4.2.1 Data Stream Management System (DSMS)

UQ. Explain with block diagram architecture of Data stream Management System. **(MU - Dec. 19, 10 Marks)**

UQ. Explain abstract architecture of Data Stream Management System (DSMS). **(MU - Dec. 16, 10 Marks)**

UQ. What is Data Stream Management System? Explain with block diagram. **(MU - May 17, 10 Marks)**

- We can view a stream processor as a kind of data-management system, the high-level organization which is shown in Fig. 4.2.1.
- Any number of streams can enter the system. Each stream can provide elements at its own schedule; they need not have the same data rates or data types, and the time between elements of one stream need not be uniform.
- The fact that the rate of arrival of stream elements is not under the control of the system distinguishes stream processing from the processing of data that goes on within a database-management system. The latter system controls the rate at which data is read from the disk, and therefore never has to worry about data getting lost as it attempts to execute queries.
- Streams may be archived in a large archival store, but we assume it is not possible to answer queries from the archival store.
- It could be examined only under special circumstances using time-consuming retrieval processes.

- There is also a working store, into which summaries or parts of streams may be placed, and which can be used for answering queries.
- The working store might be disk, or it might be main memory, depending on how fast we need to process queries. But either way, it is of sufficiently limited capacity that it cannot store all the data from all the streams.

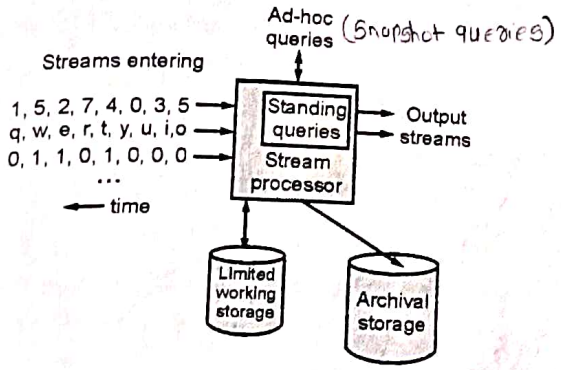


Fig. 4.2.1 : A data-stream-management system

- A Data Stream Management System (DSMS) is a computer software system to manage continuous **Data Streams**. It is similar to a **Database Management System** (DBMS), which is, however, designed for static data in conventional **Databases**.
- A DBMS also offers a flexible query processing so that the information needed can be expressed using queries. However, in contrast to a DBMS, a DSMS executes a continuous query that is not only performed once, but is permanently installed.
- Therefore, the query is continuously executed until it is explicitly uninstalled. Since most DSMS are data-driven, a continuous query produces new results as long as new data arrive at the system. This basic concept is similar to **Complex Event Processing** so that both technologies are partially coalescing.

4.3 STREAMSQL

- StreamSQL is a query language that extends SQL with the ability to process real-time data streams. SQL is primarily intended for manipulating relations (also known as tables), which are finite bags of tuples (rows).
- StreamSQL adds the ability to manipulate streams, which are infinite sequences of tuples that are not all available at the same time.
- Because streams are infinite, operations over streams must be monotonic.
- Queries over streams are generally "continuous", executing for long periods of time and returning incremental results.
- The StreamSQL language is typically used in the context of a Data Stream Management System (DSMS), for applications including market data analytics, network monitoring, surveillance, e-fraud detection and prevention, clickstream analytics and real-time compliance (anti-money laundering, RegNMS, MiFID).
- Other streaming and continuous variants of SQL include StreamSQL.io, Kafka KSQL, SQLStreamBuilder, WSO2 Stream Processor, SQLStreams, SamzaSQL, and Storm SQL.

4.3.1 StreamSQL Operations

- It extends the type system of SQL to support streams in addition to tables. Several new operations are introduced to manipulate streams.
- **Selecting from a stream :** A standard SELECT statement can be issued against a stream to calculate functions (using the target list) or filter out unwanted tuples (using a WHERE clause). The result will be a new stream.

- **Stream-Relation Join :** A stream can be joined with a relation to produce a new stream. Each tuple on the stream is joined with the current value of the relation based on a predicate to produce 0 or more tuples.
- **Union and Merge :** Two or more streams can be combined by unioning or merging them. Unioning combines tuples in strict FIFO order. Merging is more deterministic, combining streams according to a sort key.
- **Windowing and Aggregation :** A stream can be windowed to create finite sets of tuples. For example, a window of size 5 minutes would contain all the tuples in a given 5-minute period. Window definitions can allow complex selections of messages, based on tuple field values. Once a finite batch of tuples is created, analytics such as count, average, max, etc., can be applied.
- **Windowing and Joining :** A pair of streams can also be windowed and then joined together. Tuples within the join windows will combine to create resulting tuples if they fulfill the predicate.

4.3.2 Examples of Stream Sources

GQ. Explain different types of Stream sources.

Sensor Data

- Imagine a temperature sensor bobbing about in the ocean, sending back to a base station a reading of the surface temperature each hour. The data produced by this sensor is a stream of real numbers. It is not a very interesting stream, since the data rate is so low. It would not stress modern technology, and the entire stream could be kept in main memory, essentially forever.

- Now, give the sensor a GPS unit, and let it report surface height instead of temperature. The surface height varies quite rapidly compared with temperature, so we might have the sensor send back a reading every tenth of a second. If it sends a 4-byte real number each time, then it produces 3.5 megabytes per day. It will still take some time to fill up main memory, let alone a single disk.
- But one sensor might not be that interesting. To learn something about ocean behavior, we might want to deploy a million sensors, each sending back a stream, at the rate of ten per second. A million sensors isn't very many; there would be one for every 150 square miles of ocean. Now we have 3.5 terabytes arriving every day, and we definitely need to think about what can be kept in working storage and what can only be archived.

☞ **Image Data**

- Satellites often send down to earth streams consisting of many terabytes of images per day. Surveillance cameras produce images with lower resolution than satellites, but there can be many of them, each producing a stream of images at intervals like one second.
- London is said to have six million such cameras, each producing a stream.

☞ **Internet and Web Traffic**

- A switching node in the middle of the Internet receives streams of IP packets from many inputs and routes them to its outputs. Normally, the job of the switch is to transmit data and not to retain it or query it. But there is a tendency to put more capability into the switch, e.g., the ability to detect denial-of-service attacks or the ability to reroute packets based on information about congestion in the network.

- Web sites receive streams of various types. For example, Google receives several hundred million search queries per day. Yahoo! accepts billions of "clicks" per day on its various sites. Many interesting things can be learned from these streams. For example, an increase in queries like "sore throat" enables us to track the spread of viruses. A sudden increase in the click rate for a link could indicate some news connected to that page, or it could mean that the link is broken and needs to be repaired.

4.3.3 Stream Queries

UQ. With respect to data stream querying, give example of Ad-hoc queries and standing queries. (MU - May 17, 5 Marks)

- There are two ways that queries get asked about streams. We show in Fig. 4.2.1 a place within the processor where standing queries are stored. These queries are, in a sense, permanently executing, and produce outputs at appropriate times.

Example 4.3.1

The stream produced by the ocean-surface-temperature sensor. It might have a standing query to output an alert whenever the temperature exceeds 25 degrees centigrade. This query is easily answered, since it depends only on the most recent stream element.

- Alternatively, we might have a standing query that, each time a new reading arrives, produces the average of the 24 most recent readings. That query also can be answered easily, if we store the 24 most recent stream elements. When a new stream element arrives, we can drop from the working store the 25th most recent element, since it will never again be needed.

- Another query we might ask is the maximum temperature ever recorded by that sensor. We can answer this query by retaining a simple summary: the maximum of all stream elements ever seen. It is not necessary to record the entire stream. When a new stream element arrives, we compare it with the stored maximum, and set the maximum to whichever is larger.
- We can then answer the query by producing the current value of the maximum. Similarly, if we want the average temperature over all time, we have only to record two values: the number of readings ever sent in the stream and the sum of those readings. We can adjust these values easily each time a new reading arrives, and we can produce their quotient as the answer to the query.
- The other form of query is ad-hoc, a question asked once about the current state of a stream or streams. If we do not store all streams in their entirety, as normally we cannot, then we cannot expect to answer arbitrary queries about streams. If we have some idea what kind of queries will be asked through the ad-hoc query interface, then we can prepare for them by storing appropriate parts or summaries of streams as in Example 4.3.1.
- If we want the facility to ask a wide variety of ad-hoc queries, a common approach is to store a sliding window of each stream in the working store. A sliding window can be the most recent n elements of a stream, for some n, or it can be all the elements that arrived within the last t time units, e.g., one day. If we regard each stream element as a tuple, we can treat the window as a relation and query it with any SQL query. Of course the stream-management system must keep the window fresh, deleting the oldest elements as new ones come in.

Example 4.3.2

Web sites often like to report the number of unique users over the past month. If we think of each login as a stream element, we can maintain a window that is all logins in the most recent month. We

must associate the arrival time with each login, so we know when it no longer belongs to the window. If we think of the window as a relation Logins(name, time), then it is simple to get the number of unique users over the past month. The SQL query is: SELECT COUNT(DISTINCT(name)) FROM Logins WHERE time >= t; Here, t is a constant that represents the time one month before the current time. Note that we must be able to maintain the entire stream of logins for the past month in working storage. However, for even the largest sites, that data is not more than a few terabytes, and so surely can be stored on disk.

4.4 KEY ISSUES IN BIG DATA STREAM ANALYSIS

UQ. What are the challenges of querying on large data stream? (MU - May 18, 5 Marks)

Scalability

- One of the main challenges in big data streaming analysis is the issue of scalability. The big data stream Is experiencing exponential growth in a way much faster than computer resources.
- The processors follow Moore's law, but the size of data is exploding. Therefore, research efforts should be geared towards developing scalable frameworks and algorithms that will accommodate data stream computing mode, effective resource allocation strategy and parallelization issues to cope with the ever-growing size and complexity of data.

Integration

- Building a distributed system where each node has a view of the data flow, that is, every node performing analysis with a small number of sources, then aggregating these views to build a global view is non-trivial.

- An integration technique should be designed to enable efficient operations across different datasets.

☞ **Fault-tolerance**

- High fault-tolerance is required in life-critical systems.
- As data is real-time and infinite in big data stream computing environments, a good scalable high fault-tolerance strategy is required that allows an application to continue working despite component failure without interruption.

☞ **Timeliness**

- Time is of the essence for time-sensitive processes such as mitigating security threats, thwarting fraud, or responding to a natural disaster.
- There is a need for scalable architectures or platforms that will enable continuous processing of data streams which can be used to maximize the timeliness of data.
- The main challenge is implementing a distributed architecture that will aggregate local views of data into global view with minimal latency between communicating nodes.

☞ **Consistency**

- Achieving high consistency (i.e. stability) in big data stream computing environments is non-trivial as it is difficult to determine which data are needed and which nodes should be consistent.
- Hence a good system structure is required.

☞ **Heterogeneity and incompleteness**

- Big data streams are heterogeneous in structure, organisations, semantics, accessibility and granularity. The challenge here is how to handle an always ever-increasing data, extract meaningful content out of it, aggregate and correlate streaming data from multiple sources in real-time.

- A competent data presentation should be designed to reflect the structure, diversity and hierarchy of the streaming data.

☞ **Load balancing**

- A big data stream computing system is expected to be self-adaptive to data streams changes and avoid load shedding.
- This is challenging as dedicating resources to cover peak loads 24/7 is impossible and load shedding is not feasible when the variance between the average load and the peak load is high.
- As a result, a distributing environment that automatically streams partial data streams to a global centre when local resources become insufficient is required.

☞ **High throughput**

- Decision with respect to identifying the sub-graph that needs replication, how many replicas are needed and the portion of the data stream to assign to each replica is an issue in big data stream computing environment.
- There is a need for good multiple instances replication if high throughput is to be achieved.

☞ **Privacy**

- Big data stream analytics created opportunities for analyzing a huge amount of data in real-time but also created a big threat to individual privacy.
- According to the International Data Cooperation (IDC), not more than half of the entire information that needs protection is effectively protected.
- The main challenge is proposing techniques for protecting a big data stream dataset before its analysis.

☞ **Accuracy**

- One of the main objectives of big data stream analysis is to develop effective techniques that can accurately predict future observations.
- However, as a result of inherent characteristics of big data such as volume, velocity, variety, variability, veracity, volatility, and value, big data analysis strongly constrain processing algorithms spatio-temporally and hence stream-specific requirements must be taken into consideration to ensure high accuracy.

4.5 SAMPLING TECHNIQUES FOR EFFICIENT STREAM PROCESSING

UQ. Describe any two sampling techniques for big data with the help of examples. **(MU - May 16, 10 Marks)**

4.5.1 Sliding Window

- This is the simplest and most straightforward method. A first-in, first-out (FIFO) queue with size n and a skip / sub-sampling factor $k \geq 1$ is maintained.

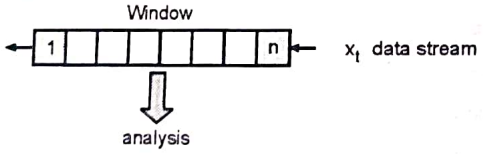


Fig. 4.5.1 : Sliding Window

- In addition to that, a stride factor $s \geq 1$ describes by how many time-steps the window is shifted before analyzing it.

☞ **Advantage**

(1) Simple to implement.

(2) Deterministic reservoir can be filled very fast from the beginning.

☞ **Drawbacks**

The time history represented by the reservoir R is short; long-term concept drifts cannot be detected easily outliers can create noisy analyses.

4.5.2 Unbiased Reservoir Sampling

- A reservoir R is maintained such that at time $t > n$ the probability of accepting point x(t) in the reservoir is equal to n/t.
- The algorithm [1] is as follows :
  - Fill the reservoir R with the first n points of the stream.
  - At time $t > n$ replace a randomly chosen (equal probability) entry in the reservoir R with acceptance probability n/t.
- This leads to a reservoir R(t) such that each point x(1)...x(t) is contained in R(t) with equal property n/t.

☞ **Advantages**

(1) The reservoir contains data points from all history of the stream with equal probability.

(2) Very simple implementation; adding a point requires only O(1)

☞ **Drawbacks**

A concept drift cannot be compensated; the oldest data point x(1) is equal important in this sampling technique as the latest data point x(t).

4.5.3 Biased Reservoir Sampling

The bias function associated with the r-th data point at the time of arrival of the t-th point ($r \leq t$) is given by f(r, t) and is related to the probability p(r, t) of the r-th point belonging to the reservoir at the time of arrival of the t-th point. Specifically, p(r, t) is proportional to f(r, t). The function f(r, t) is monotonically decreasing with t

(for fixed r) and monotonically increasing with r (for fixed t). Therefore, the use of a bias function ensures that recent points have higher probability of being represented in the sample reservoir. Next, we define the concept of a bias sensitive sample S(t), which is defined by the bias function f(r, t).

Definition : Let f(r, t) be the bias function for the r-th point at the arrival of the t-th point. A biased sample S(t) at the time of arrival of the t-th point in the stream is defined as a sample such that the relative probability p(r, t) of the r-th point belonging to the sample S(t) (of size n) is proportional to f(r, t).

Algorithm 4.5.1

We start off with an empty reservoir with capacity $n = pin/\lambda$, and use the following replacement policy to gradually fill up the reservoir. Let us assume that at the time of (just before) the arrival of the t-th point, the fraction of the reservoir filled is $F(t) \in [0, 1]$. When the (t + 1)-th point arrives. We add it to the reservoir with insertion probability pin. However, we do not necessarily delete one of the old points in the reservoir. We flip a coin with success probability F(t). In the event of a success, we randomly pick one of the points in the reservoir, and replace its position in the sample array by the incoming (t + 1)-th point. In the event of a failure, we do not delete any of old points and simply add the (t + 1)-th point to the reservoir. In the latter case, the number of points in the reservoir (current sample size) increases by 1.

The algorithm has a lower insertion probability, and a corresponding reduced reservoir requirement. In this case, the value of λ and n are decided by application specific constraints, and the value

of pin is set to $n \cdot \lambda$. We show that the sample from the modified algorithm shows the same bias distribution, but with a reduced reservoir size.

- In biased reservoir sampling Alg. 4.5.1, [2] the probability of a data point x(t) being in the reservoir is a decreasing function of its lingering time within R.
- So the probability of finding points of the sooner history in R is high. Very old data points will be in R with very low probability.

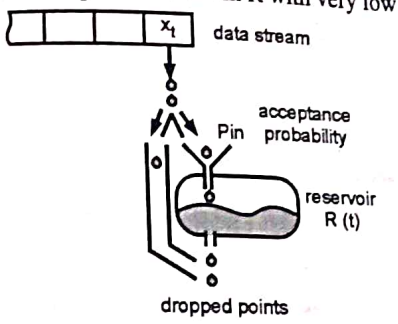


Fig. 4.5.2

Illustration of the Biased Reservoir Sampling

The probability that point x(r) is contained in R(t) equals to

$$P(r, t) = e^{-\lambda (t - r)}$$

- So this is an exponential forgetting. For details of the algorithm, see [2] and the goreservoir package.
- The example from the github.com\andremueller\goservoir package shows the lingering time of a stack of unbiased reservoir samplers.

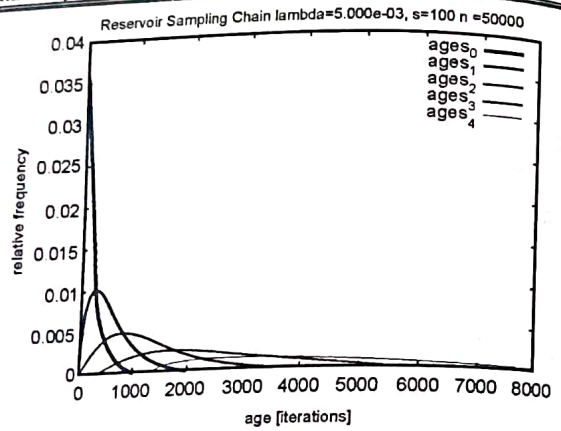


Fig. 4.5.3 : Reservoir Sampling Chain output

☞ Advantages

(1) O(1) algorithm for adding a new data point.

(2) Slowly moving concept drifts can be compensated.

(3) An adjustable forgetting factor can be tuned for the application of interest.

☞ Drawbacks

It is a randomized technique. So the algorithm is non-deterministic. However, the variance might be estimated by running an ensemble of independent reservoirs.

$$R_1 \ldots R_B$$

4.5.4 Histograms

A histogram is maintained while observing the data stream. Hereto, data points are sorted into intervals/buckets

$$[l_i, u_i]$$

- If the useful range of the observed values is known in advance, a simple vector with counts and breakpoints could do the job.

- V-optimal histograms tries to minimize the variance within each histogram bucket.
- Proposes an algorithm for efficiently maintaining an approximate V-optimum histogram from a data stream. This is of relevance for interval data, such as a time-series of temperature values; i.e., absolute value and distance between values have a meaning.

4.6 FILTERING STREAMS : BLOOM FILTER WITH ANALYSIS

GQ. Explain filtering streams.

☞ Filtering Streams

It identifies the sequence patterns in a stream. Stream filtering is the process of selection or matching instances of a desired pattern in a continuous stream of data.

Example

Assume that a data stream consists of tuples

Filtering steps :

(i) Accept the tuples that meet a criterion in the stream,

(ii) Pass the accepted tuples to another process as a stream and

(iii) Discard remaining tuples.

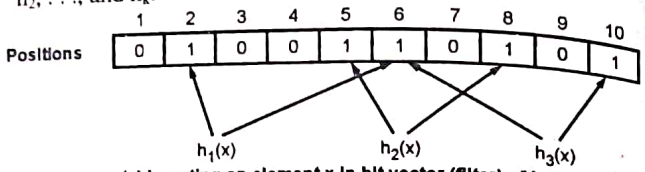
UQ. How bloom filter is useful for big data analytics? Explain with one example. **(MU - Dec. 18, 10 Marks)**

UQ. Explain the concept of Blooms Filter using an example. **(MU - Dec. 17, 10 Marks)**

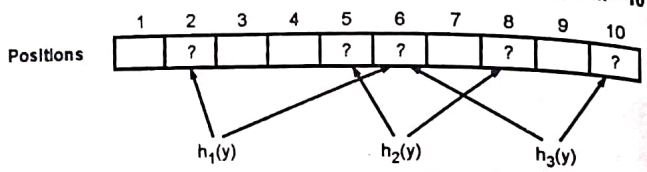
☞ Bloom Filter with Analysis

- A simple space-efficient data structure introduced by Burton Howard Bloom in 1970. The filter matches the membership of an element in a dataset.
- The filter is basically a bit vector of length m that represent a set $S = \{x_1, x_2, \ldots, x_m\}$ of m elements,

- Initially all bits 0. Then, define k independent hash functions, h_1, h_2, . . ., and h_k.



(a) Inserting an element x in bit vector (filter) of length m = 10



(b) Finding element y is the Filter

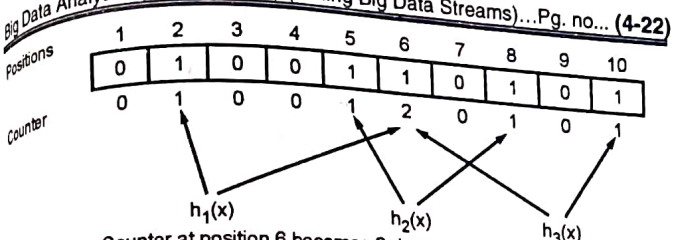
(a) Inserting an element x in bit vector (filter) of length m = 10, (b) finding an element y in an example of Bloom filter

Fig. 4.6.1

- Each of which maps (hashes) some element x in set S to one of the m array positions with a uniform random distribution.
- Number k is constant, and much smaller than m. That is, for each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$. [$\in$ is symbol in set theory for 'contained in'.]

☞ Counting Bloom Filter: A Variant of Bloom Filter

- It mmaintains a counter for each bit in the Bloom filter. The counters corresponding to the k hash values increment or decrement, whenever an element in the filter is added or deleted, respectively.
- As soon as a counter changes from 0 to 1, the corresponding bit in the bit vector is set to 1. When a counter changes from 1 to 0, the corresponding bit in the bit vector is set to 0. The counter basically maintains the number of elements that hashed.



Counter at position 6 becomes 2 due to two hash functions h_1 and h_3, when an element x is inserted in bit vector (filter) of lenght m = 10

Fig. 4.6.2 : Example of counting bloom filter

4.7 COUNTING DISTINCT ELEMENTS IN A STREAM

It relates to finding the number of dissimilar elements in a data stream. The stream of data contains repeated elements. This is a well-known problem in networking and databases.

4.7.1 The Count-Distinct Problem

UQ. What do you mean by Counting Distinct Element in a stream? Illustrate with an example working of an Flajolet-martin algorithm used to count number of distinct elements. **(MU - Dec. 19, 10 Marks)**

- The count-distinct problem (also known in applied mathematics as the cardinality estimation problem) is the problem of finding the number of distinct elements in a data stream with repeated elements.
- This is a well-known problem with numerous applications.
- The elements might represent IP addresses of packets passing through a router, unique visitors to a web site, elements in a large database, motifs in a DNA sequence, or elements of RFID/sensor networks.

☞ **Formal definition**

Instance

A stream of elements x_1, x_2,...x_3 with repetitions, and an integer m. Let n be the number of distinct elements, namely $n = \{x_1, x_2,...x_3\}$ and let these elements by $\{e_1, e_2, ...e_n\}$.

☞ **Objective**

Find an estimate n of n using only m storage units, where $m << n$

An example of an instance for the cardinality estimation problem is the steam a,b,a,c,d,b,d. For this instance.

$$n = |\{a, b, c, d\}| = 4$$

☞ **Naive solution [edit]**

The naive solution to the problem is as follows :

- Inititalize a counter, c, to zero, $c \leftarrow 0$,
- Initialize an efficient dictionary data structure, D, such as hash table or search tree in which insertion and membership can be performet quickly.

For each element x_i, a membership query is issued.

If x_i is not a member of D ($x_i \notin D$)

Add x_i to D

Increase c by one, $c \leftarrow c + 1$

Otherwise ($x_i \in D$) do nothing.

Output n = c.

4.7.2 Flajolet Martin Algorithm

UQ. What do you mean by Counting Distinct Elements in a stream? Illustrate with an example working of a Flajolet – Martin Algorithm used to count number of distinct elements. **(MU - Dec. 19, 10 Marks)**

UQ. Give problem in Flajolet-Martin (FM) algorithm to count distinct element in a stream. **(MU - Dec. 16, 5 Marks)**

- Flajolet Martin Algorithm, also known as FM algorithm, is used to approximate the number of unique elements in a data stream or database in one pass.
- The highlight of this algorithm is that it uses less memory space while executing.
- Pseudo Code-Stepwise Solution :

1. Selecting a hash function h so each element in the set is mapped to a string to at least $\log_2 n$ bits.
2. For each element x, r(x) = length of trailing zeroes in h(x)
3. R = max(r(x)) => Distinct elements = 2_R

☞ **Reasons for using Flajolet Martin algorithm**

- Let us compare this algorithm with our conventional algorithm using python code. Assume we have an array (stream in code) of data of length 20 with 8 unique elements.
- Using the brute force approach to find the number of unique elements in the array, each element is taken into consideration. Another array (st_unique in code) is formed for unique elements.
- Initially, the new array is empty (st_unique length equals zero), so naturally, the first element is not present in it.
- The first element is considered to be unique as it does not exist in the new array and thus a copy of the first element is inserted into the new array (1 is appended to st_unique)
- Similarly, all the elements are checked, if they are already present in the new array, they are not considered to be unique, else a copy of the element is inserted into the new array.

- Running the brute force algorithm for our array, we will get 8 elements in the new array.
- If each element takes 20 bytes of data, the new array will take 8*20 = 160 bytes memory to run the algorithm.

```
stream=[1,2,3,4,5,6,4,2,5,9,1,6,3,7,1,2,2,4,2,1]
print('Using conventional Algorithm:')
start_time = time.time()
st_unique=[]
for i in stream:
if i in st_unique:
continue
else:
st_unique.append(i)
print('distinct elements',len(st_unique))
print("--- %s seconds ---" % (time.time() - start_time))
```

- For the same array, if use the FM algorithm for the same array, we define a variable (maxnum in code) that stores the maximum number of zeroes at the end.
- For each value in the array(stream in code), we run a loop to covert its hash function in the form ax + b mod c,(a = 1, b = 6 and c = 32 in this case) into binary (we place [2:] at the end because in python when converted to binary, the number starts with '0b').
- We run another loop to find if the number of zeroes at the end exceeds the maximum number of zeroes.
- In this case, if each variable occupies 2 bytes of data, the whole program takes 4*20= 80 bytes of data, i.e. half of the memory used in the above case.
- Here, we have considered the variables maxnum, sum, val, and j. We have not considered i, time, and stream as they are common in both of the codes.


```
stream=[1,2,3,4,5,6,4,2,5,9,1,6,3,7,1,2,2,4,2,1]
print('Using Flajolet Martin Algorithm:')
import time
start_time = time.time()
maxnum=0
for i in range(0,len(stream)):
val= bin((1*stream[i] + 6) % 32)[2:]

sum=0
for j in range(len(val)-1,0,-1):

if val[j]=='0':
sum+=1
else:
break
if sum>maxnum:
maxnum=sum
print('distict elements', 2**maxnum)
print("--- %s seconds ---" % (time.time() - start_time))
```

- For small values of m(where m is the number of unique elements), the brute force approach can work, but for large data sets or data streams, where m is very large, a lot of space is required. The compiler may not let us run the algorithm in some cases.
- This is where the Flajolet Martin Algorithm can be used.
- Not only does it occupy less memory, but it also shows better results in terms of time in seconds when the python code is run which can be shown in our output as we calculated seconds taken by both algorithms by using time.time() in python.

- As shown in the output, it can clearly be said that the FM algorithm takes very little time as compared to the conventional algorithm.

```
Using Flajolet Martin Algorithm :
distict elements 8
- - - 0.0001180171966552734 4 seconds - -
Using conventional Algorithm
distinct elements 8
---
7.295608520507812e-05 seconds - - -
```

4.8 COMBINING ESTIMATES

GQ. Explain the combining estimates method.

- The strategy for combining the estimates of m, the number of distinct elements, that we obtain by using many different hash functions.
- Our first assumption would be that if we take the average of the values 2^R that we get from each hash function, we shall get a value that approaches the true m, the more hash functions we use. Consider a value of r such that 2^r is much larger than m. There is some probability p that we shall discover r to be the largest number of 0's at the end of the hash value for any of the m stream elements. Then the probability of finding r + 1 to be the largest number of 0's instead is at least p/2. However, if we do increase by 1 the number of 0's at the end of a hash value, the value of 2^R doubles.
- Consequently, the contribution from each possible large R to the expected value of 2^R grows as R grows, and the expected value of 2^R is actually infinite.

- Another way to combine estimates is to take the median of all estimates. The median is not affected by the occasional outsized value of 2^R, so the worry described above for the average should not carry over to the median. Unfortunately, the median suffers from another defect: it is always a power of 2. Thus, no matter how many hash functions we use, should the correct value of m be between two powers of 2, say 400, then it will be impossible to obtain a close estimate.
- There is a solution to the problem, however. We can combine the two methods. First, group the hash functions into small groups, and take their average. Then, take the median of the averages.
- It is true that an occasional outsized 2^R will bias some of the groups and make them too large. However, taking the median of group averages will reduce the influence of this effect almost to nothing.
- Moreover, if the groups themselves are large enough, then the averages can be essentially any number, which enables us to approach the true value m as long as we use enough hash functions. In order to guarantee that any possible average can be obtained, groups should be of size at least a small multiple of $\log_2 m$.

4.8.1 Space Requirements

- As we read the stream it is not necessary to store the elements seen. The only thing we need to keep in main memory is one integer per hash function; this integer records the largest tail length seen so far for that hash function and any stream element.
- If we are processing only one stream, we could use millions of hash functions, which is far more than we need to get a close estimate. Only if we are trying to process many streams at the

- As shown in the output, it can clearly be said that the FM algorithm takes very little time as compared to the conventional algorithm.

```
Using  Flajolet Martin Algorithm :
distict elements 8
- - - 0.0001180171966552734 4 seconds – –
Using conventional Algorithm
distinct elements 8
---
7.295608520507812e-05 seconds - - -
```

4.8 COMBINING ESTIMATES

GQ. Explain the combining estimates method.

- The strategy for combining the estimates of m, the number of distinct elements, that we obtain by using many different hash functions.
- Our first assumption would be that if we take the average of the values 2^R that we get from each hash function, we shall get a value that approaches the true m, the more hash functions we use. Consider a value of r such that 2^r is much larger than m. There is some probability p that we shall discover r to be the largest number of 0's at the end of the hash value for any of the m stream elements. Then the probability of finding r + 1 to be the largest number of 0's instead is at least p/2. However, if we do increase by 1 the number of 0's at the end of a hash value, the value of 2^R doubles.
- Consequently, the contribution from each possible large R to the expected value of 2^R grows as R grows, and the expected value of 2^R is actually infinite.

- Another way to combine estimates is to take the median of all estimates. The median is not affected by the occasional outsized value of 2^R, so the worry described above for the average should not carry over to the median. Unfortunately, the median suffers from another defect: it is always a power of 2. Thus, no matter how many hash functions we use, should the correct value of m be between two powers of 2, say 400, then it will be impossible to obtain a close estimate.
- There is a solution to the problem, however. We can combine the two methods. First, group the hash functions into small groups, and take their average. Then, take the median of the averages.
- It is true that an occasional outsized 2^R will bias some of the groups and make them too large. However, taking the median of group averages will reduce the influence of this effect almost to nothing.
- Moreover, if the groups themselves are large enough, then the averages can be essentially any number, which enables us to approach the true value m as long as we use enough hash functions. In order to guarantee that any possible average can be obtained, groups should be of size at least a small multiple of $\log_2 m$.

4.8.1 Space Requirements

- As we read the stream it is not necessary to store the elements seen. The only thing we need to keep in main memory is one integer per hash function; this integer records the largest tail length seen so far for that hash function and any stream element.
- If we are processing only one stream, we could use millions of hash functions, which is far more than we need to get a close estimate. Only if we are trying to process many streams at the

same time would main memory constrain the number of hash functions we could associate with any one stream.

- In practice, the time it takes to compute hash values for each stream element would be the more significant limitation on the number of hash functions we use.

4.9 COUNTING ONES IN A WINDOW

UQ. Give two applications for counting the number of 1's in a long stream of binary values. Using a stream of binary digits, illustrate how the DGIM algorithm will find the number of 1's ? **(MU - May 18, 5 Marks)**

- The Cost of Exact Counts Sliding window model - data elements arrive at every instant; each data element expires after exactly N time steps; and the portion of data that is relevant to gathering statistics or answering queries is the set of the last N elements to arrive.
- Able to count exactly the number of 1s in the last k bits for any k<=N, it is necessary to store all N bits of the window. Problem :
- Most counting applications N is still so large that it cannot be stored on disk or there are so many streams that windows for all cannot be stored.
- Another method: Random Sampling – this will fail when the 1s may not arrive in a uniform manner.

4.9.1 Datar-Gionis-Indyk-Motwani

UQ. Explain DGIM algorithm for counting ones in stream with example. **(MU - May 19, Dec. 18, 10 Marks)**

UQ. Using an example bit stream explain the working of the DGIM algorithm to count number of 1's in a data stream. **(MU - May 16, 10 Marks)**

The simplest case of an algorithm called DGIM. This version of the algorithm uses $O(\log^2 N)$ bits to represent a window of N bits and enables the estimation of the number of 1s in the window with an error of no more than 50%.

Divide the window into buckets, consisting of – The timestamp of its right 1 – The number of 1s in the bucket. This number must be power of 2, and we refer to the number of 1s as the size of the bucket.

The right end of a bucket always starts with a position with a 1.

- Number of 1s must be power of 2.
- Either one or two buckets with the same power of 2 number of 1s exists.
- Buckets do not overlap in timestamps
- Buckets are stored by size.
- Buckets disappear when their end-time is N time units in the past.

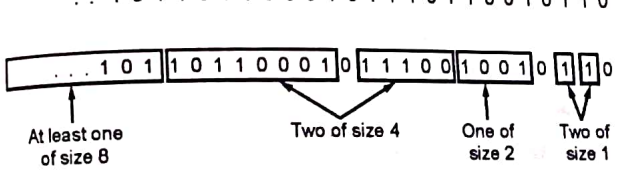


Fig. 4.9.1 : A bit-stream divided into buckets following the DGIM rules

4.9.2 Query Answering in the DGIM Algorithm

- Suppose we are asked how many 1's there are in the last k bits of the window, for some $1 \leq k \leq N$. Find the bucket b with the earliest timestamp that includes at least some of the k most recent bits. Estimate the number of 1's to be the sum of the sizes of all the buckets to the right (more recent) than bucket b, plus half the size of b itself

- **Example :** Suppose the stream is that of Fig. 4.9.1, and k = 10. Then the query asks for the number of 1's in the ten rightmost bits, which happen to be 0110010110. Let the current timestamp (time of the rightmost bit) be t. Then the two buckets with one 1, having timestamps t – 1 and t – 2 are completely included in the answer.
- The bucket of size 2, with timestamp t – 4, is also completely included. However, the rightmost bucket of size 4, with timestamp t–8 is only partly included. We know it is the last bucket to contribute to the answer, because the next bucket to its left has timestamp less than t – 9 and thus is completely out of the window.
- On the other hand, we know the buckets to its right are completely inside the range of the query because of the existence of a bucket to their left with timestamp t – 9 or greater.
- Our estimate of the number of 1's in the last ten positions is thus 6. This number is the two buckets of size 1, the bucket of size 2, and half the bucket of size 4 that is partially within range. Of course the correct answer is 5. 2 Suppose the above estimate of the answer to a query involves a bucket b of size 2j that is partially within the range of the query.
- Let us consider how far from the correct answer c our estimate could be. There are two cases: the estimate could be larger or smaller than c.
- **Case 1 :** The estimate is less than c. In the worst case, all the 1's of b are actually within the range of the query, so the estimate misses half bucket b, or 2j–1 1's. But in this case, c is at least 2j; in fact it is at least 2j+1 – 1, since there is at least one bucket of each of the sizes 2j – 1, 2j – 2, . . . , 1. We conclude that our estimate is at least 50% of c.

Case 2 : The estimate is greater than c. In the worst case, only the rightmost bit of bucket b is within range, and there is only one bucket of each of the sizes smaller than b. Then c = 1 + 2j –1 + 2j – 2 + · · · + 1 = 2j and the estimate we give is 2j–1 + 2j–1 + 2j–2 + · · · + 1 = 2j + 2j –1 – 1. We see that the estimate is no more than 50% greater than c.

4.9.3 Decaying Windows

GQ. Explain Decaying windows method.

It useful in applications which need identification of most common elements. Decaying window concept assigns more weight to recent elements. The technique computes a smooth aggregation of all the 1's ever seen in the stream, with decaying weights. When element further appears in the stream, less weight is given. The effect of exponentially decaying weights is to spread out the weights of the stream elements as far back in time as the stream flows.

The Problem of Most-Common Elements

- Suppose we have a stream whose elements are the movie tickets purchased all over the world, with the name of the movie as part of the element. We want to keep a summary of the stream that is the most popular movies "currently." While the notion of "currently" is imprecise, intuitively, we want to discount the popularity of a movie like Star Wars–Episode 4, which sold many tickets, but most of these were sold decades ago.
- On the other hand, a movie that sold n tickets in each of the last 10 weeks is probably more popular than a movie that sold 2n tickets last week but nothing in previous weeks. One solution would be to imagine a bit stream for each movie.
- The i^{th} bit has value 1 if the i^{th} ticket is for that movie, and 0 otherwise. Pick a window size N, which is the number of most

recent tickets that would be considered in evaluating popularity. Then, use the method of Section 4.6 to estimate the number of tickets for each movie, and rank movies by their estimated counts.

- This technique might work for movies, because there are only thousands of movies, but it would fail if we were instead recording the popularity of items sold at Amazon, or the rate at which different Twitter-users tweet, because there are too many Amazon products and too many tweeters. Further, it only offers approximate answers

☞ Definition of the Decaying Window

- An alternative approach is to redefine the question so that we are not asking for a count of 1's in a window. Rather, let us compute a smooth aggregation of all the 1's ever seen in the stream, with decaying weights, so the further back in the stream, the less weight is given.
- Formally, let a stream currently consist of the elements a1, a2, . . ., at, where a1 is the first element to arrive and at is the current element. Let c be a small constant, such as 10–6 or 10–9. Define the exponentially decaying window for this stream to be the sum

$$\sum_{i=0}^{t=1} a_{t-i}(1-c)^i$$

- The effect of this definition is to spread out the weights of the stream elements as far back in time as the stream goes.
- In contrast, a fixed window with the same sum of the weights, 1/c, would put equal weight 1 on each of the most recent 1/c elements to arrive and weight 0 on all previous elements. The distinction is suggested by Fig. 4.9.2.

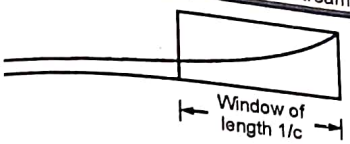


Fig. 4.9.2 : A decaying window and a fixed-length window of equal weight

It is much easier to adjust the sum in an exponentially decaying window than in a sliding window of fixed length. In the sliding window, we have to worry about the element that falls out of the window each time a new element arrives. That forces us to keep the exact elements along with the sum, or to use an approximation scheme such as DGIM. However, when a new element at + 1 arrives at the stream input, all we need to do is :

1. Multiply the current sum by 1 – c. 2. Add a_{t+1}.

The reason this method works is that each of the previous elements has now moved one position further from the current element, so its weight is multiplied by 1 – c. Further, the weight on the current element is $(1 - c)^0 = 1$, so adding a_{t+1} is the correct way to include the new element's contribution.

☞ Finding the Most Popular Elements

- The problem of finding the most popular movies in a stream of ticket sales. We shall use an exponentially decaying window with a constant c, which you might think of as 10–9 . That is, we approximate a sliding window holding the last one billion ticket sales.
- For each movie, we imagine a separate stream with a 1 each time a ticket for that movie appears in the stream, and a 0 each time a ticket for some other movie arrives. The decaying sum of the 1's measures the current popularity of the movie.

- We imagine that the number of possible movies in the stream is huge, so we do not want to record values for the unpopular movies. Therefore, we establish a threshold, say 1/2, so that if the popularity score for a movie goes below this number, its score is dropped from the counting. For reasons that will become obvious, the threshold must be less than 1, although it can be any number less than 1.
- When a new ticket arrives on the stream, do the following :
  1. For each movie whose score we are currently maintaining, multiply its score by (1 – c).
  2. Suppose the new ticket is for movie M. If there is currently a score for M, add 1 to that score. If there is no score for M, create one and initialize it to 1.
  3. If any score is below the threshold 1/2, drop that score.

Chapter Ends...

❑❑❑