

# UNIT-3

## Mining Data Streams

data arrives in a stream or streams, and if it is not processed immediately or stored, then it is lost forever. Moreover, we shall assume that the data arrives so rapidly that it is not feasible to store it all in active storage (i.e., in a conventional database), and then interact with it at the time of our choosing.

- Process of extracting knowledge structures from continuous, rapid data record, called **mining data stream**.
- Focused three dimensions:
  - Accuracy
  - Amount of memory space necessary
  - Time required to learn and to predict

### 3.1

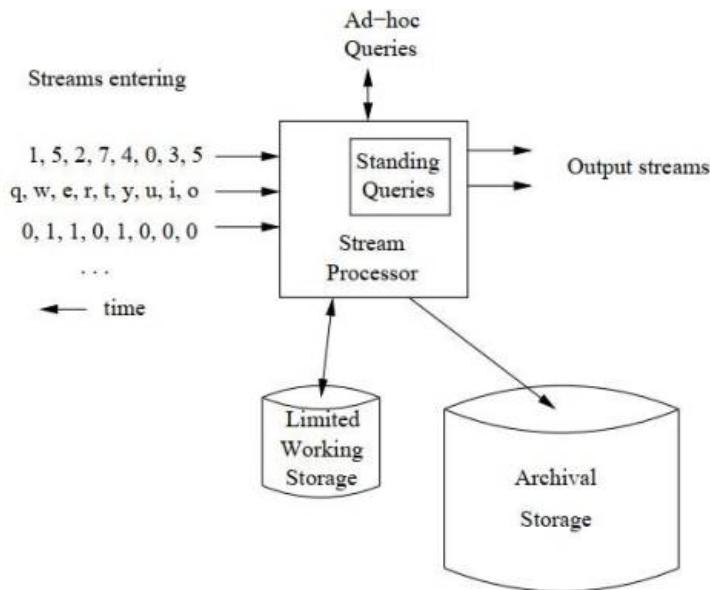


Figure3.1: A data-stream-management system

### **3.1.1 A Data-Stream-Management System**

In analogy to a database-management system, we can view a stream processor as a kind of data-management system.

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.

### **3.1.2 Examples of Stream Sources**

Before proceeding, let us consider some of the ways in which stream data arises naturally.

#### 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.

### 3.1.3 Stream Queries

There are two ways that queries get asked about streams. We show in Fig. 3.1 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 3.1 : The stream produced by the ocean-surface-temperature sensor mentioned at the beginning of Section 3.1.2 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 (unless there is some other standing query that requires it).

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. Q

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 can not, 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 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 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.  
Q

### 3.1.4 Issues in Stream Processing

Before proceeding to discuss algorithms, let us consider the constraints under which we work when dealing with streams. First, streams often deliver elements very rapidly. We must process elements in real time, or we lose the opportunity to process them at all, without accessing the archival storage. Thus, it often is important that the stream-processing algorithm is executed in main memory, without access to secondary storage or with only rare accesses to secondary storage. Moreover, even when streams are “slow,” as in the sensor-data example of Section 4.1.2, there may be many such streams. Even if each stream by itself can be processed using a small amount of main memory, the requirements of all the streams together can easily exceed the amount of available main memory.

Thus, many problems about streaming data would be easy to solve if we had enough memory, but become rather hard and require the invention of new techniques in order to execute them at a realistic rate on a machine of realistic size. Here are two generalizations about stream algorithms worth bearing in mind as you read through this chapter:

- Often, it is much more efficient to get an approximate answer to our problem than an exact solution.
- a variety of techniques related to hashing turn out to be useful. Generally, these techniques introduce useful randomness into the algorithm’s behavior, in order to produce an approximate answer that is very close to the true result.

## Sampling Data in a Stream

we shall look at extracting reliable samples from a stream. As with many stream algorithms, the “trick” involves using hashing in a somewhat unusual way.

### A Motivating Example

The general problem we shall address is selecting a subset of a stream so that we can ask queries about the selected subset and have the answers be statistically representative of the stream as a whole. If we know what queries are to be asked, then there are a number of methods that might work, but we are looking for a technique that will allow ad-hoc queries on the sample. We shall look at a particular problem, from which the general idea will emerge.

Our running example is the following. A search engine receives a stream of queries, and it would like to study the behavior of typical users.<sup>1</sup> We assume the stream consists of tuples (user, query, time). Suppose that we want to answer queries such as “What fraction of the typical user’s queries were repeated over the past month?” Assume also that we wish to store only 1/10th of the stream elements.

The obvious approach would be to generate a random number, say an integer from 0 to 9, in response to each search query. Store the tuple if and only if the random number is 0. If we do so, each user has, on average, 1/10th of their queries stored. Statistical fluctuations will introduce some noise into the data, but if users issue many queries, the law of large numbers will assure us that most users will have a fraction quite close to 1/10th of their queries stored.

However, this scheme gives us the wrong answer to the query asking for the average number of duplicate queries for a user. Suppose a user has issued  $s$  search queries one time in the past month,  $d$  search queries twice, and no search queries more than twice. If we have a 1/10th sample, of queries, we shall see in the sample for that user an expected  $s/10$  of the search queries issued once. Of the  $d$  search queries issued twice, only  $d/100$  will appear twice in the sample; that fraction is  $d$  times the probability that both occurrences of the query will be in the 1/10th sample. Of the queries that appear twice in the full stream,  $18d/100$  will appear exactly once. To see why, note that  $18/100$  is the probability that one of the two occurrences will be in the 1/10th of the stream that is selected, while the other is in the 9/10th that is not selected.

### Obtaining a Representative Sample

The query of Section 4.2.1, like many queries about the statistics of typical users, cannot be answered by taking a sample of each user’s search queries. Thus, we must strive to pick 1/10th of the users, and take all their searches for the sample, while taking none of the searches from other users. If we can store a list of all users, and whether or not they are in the sample, then we could do the following. Each time a search query arrives in the stream, we look up the user to see whether or not they are in the sample. If so, we add this search query to the sample, and if not, then not. However, if we have no record of ever having seen this user before, then we generate a random integer between 0 and 9. If the number is 0, we add this user to our list with value “in,” and if the number is other than 0, we add the user with the value “out.”

That method works as long as we can afford to keep the list of all users and their in/out decision in main memory, because there isn’t time to go to disk for every search that arrives. By using a hash function, one can avoid keeping the list of users. That is, we hash each user name to one of ten buckets, 0 through 9. If the user hashes to bucket 0, then accept this search query for the sample, and if not, then not.

Note we do not actually store the user in the bucket; in fact, there is no data in the buckets at all. Effectively, we use the hash function as a random-number generator, with the important property that, when applied to the same user several times, we always get the same “random” number. That is, without storing the in/out decision for any user, we can reconstruct that decision any time a search query by that user arrives.

### The General Sampling Problem

The running example is typical of the following general problem. Our stream consists of tuples with  $n$  components. A subset of the components are the *key* components, on which the selection of the sample will be based. In our running example, there are three components – user, query, and time – of which only *user* is in the key. However, we could also take a sample of queries by making *query* be the key, or even take a sample of user-query pairs by making both those components form the key.

To take a sample of size  $a/b$ , we hash the key value for each tuple to  $b$  buckets, and accept the tuple for the sample if the hash value is less than  $a$ . If the key consists of more than one component, the hash function needs to combine the values for those components to make a single hash-value. The

result will be a sample consisting of all tuples with certain key values. The selected key values will be approximately  $a/b$  of all the key values appearing in the stream.

## Varying the Sample Size

Often, the sample will grow as more of the stream enters the system. In our running example, we retain all the search queries of the selected 1/10th of the users, forever. As time goes on, more searches for the same users will be accumulated, and new users that are selected for the sample will appear in the stream.

If we have a budget for how many tuples from the stream can be stored as the sample, then the fraction of key values must vary, lowering as time goes on. In order to assure that at all times, the sample consists of all tuples from a subset of the key values, we choose a hash function  $h$  from key values to a very large number of values  $0, 1, \dots, B-1$ . We maintain a *threshold*  $t$ , which initially can be the largest bucket number,  $B-1$ . At all times, the sample consists of those tuples whose key  $K$  satisfies  $h(K) \leq t$ . New tuples from the stream are added to the sample if and only if they satisfy the same condition.

If the number of stored tuples of the sample exceeds the allotted space, we lower  $t$  to  $t-1$  and remove from the sample all those tuples whose key  $K$  hashes to  $t$ . For efficiency, we can lower  $t$  by more than 1, and remove the tuples with several of the highest hash values, whenever we need to throw some key values out of the sample. Further efficiency is obtained by maintaining an index on the hash value, so we can find all those tuples whose keys hash to a particular value quickly.

## Filtering Streams

Another common process on streams is selection, or filtering. We want to accept those tuples in the stream that meet a criterion. Accepted tuples are passed to another process as a stream, while other tuples are dropped. If the selection criterion is a property of the tuple that can be calculated (e.g., the first component is less than 10), then the selection is easy to do. The problem becomes harder when the criterion involves lookup for membership in a set. It is especially hard, when that set is too large to store in main memory. In this section, we shall discuss the technique known as “Bloom filtering” as a way to eliminate most of the tuples that do not meet the criterion.

## A Motivating Example

Again let us start with a running example that illustrates the problem and what we can do about it. Suppose we have a set  $S$  of one billion allowed email addresses – those that we will allow through because we believe them not to be spam. The stream consists of pairs: an email address and the email itself. Since the typical email address is 20 bytes or more, it is not reasonable to store  $S$  in main memory. Thus, we can either use disk accesses to determine whether or not to let through any given stream element, or we can devise a method that requires no more main memory than we have available, and yet will filter most of the undesired stream elements.

Suppose for argument's sake that we have one gigabyte of available main memory. In the technique known as *Bloom filtering*, we use that main memory as a bit array.

### **Bloom Filters –**

Suppose you are creating an account on Geekbook, you want to enter a cool username, you entered it and got a message, “Username is already taken”. You added your birth date along username, still no luck. Now you have added your university roll number also, still got “Username is already taken”. It’s really frustrating, isn’t it? But have you ever thought how quickly Geekbook check availability of username by searching millions of username registered with it. There are many ways to do this job –

- **Linear search** : Bad idea!
- **Binary Search** : Store all username alphabetically and compare entered username with middle one in list, If it matched, then username is taken otherwise figure out , whether entered username will come before or after middle one and if it will come after, neglect all the usernames before middle one(inclusive). Now search after middle one and repeat this process until you got a match or search end with no match.This technique is better and promising but still it requires multiple steps. But, There must be something better!!

**Bloom Filter** is a data structure that can do this job.

For understanding bloom filters, you must know what is [hashing](#). A hash function takes input and outputs a unique identifier of fixed length which is used for identification of input.

### **What is Bloom Filter?**

A Bloom filter is a **space-efficient probabilistic** data structure that is used to test whether an element is a member of a set. For example, checking availability of username is set membership problem, where the set is the list of all registered username. The price we pay for efficiency is that it is probabilistic in nature that means, there might be some False Positive results. **False positive means**, it might tell that given username is already taken but actually it's not.

### **Interesting Properties of Bloom Filters**

- Unlike a standard hash table, a Bloom filter of a fixed size can represent a set with an arbitrarily large number of elements.
- Adding an element never fails. However, the false positive rate increases steadily as elements are added until all bits in the filter are set to 1, at which point all queries yield a positive result.
- Bloom filters never generate **false negative** result, i.e., telling you that a username doesn't exist when it actually exists.
- Deleting elements from filter is not possible because, if we delete a single element by clearing bits at indices generated by k hash functions, it might cause deletion of few other elements. Example – if we delete “geeks” (in given example below) by clearing bit at 1, 4 and 7, we might end up deleting “nerd” also Because bit at index 4 becomes 0 and bloom filter claims that “nerd” is not present.

### **Working of Bloom Filter**

A empty bloom filter is a **bit array** of **m** bits, all set to zero, like this –

0	0	0	0	0	0	0	0	0	0
1	2	3	4	5	6	7	8	9	10

We need **k** number of **hash functions** to calculate the hashes for a given input. When we want to add an item in the filter, the bits at k indices  $h_1(x), h_2(x), \dots, h_k(x)$  are set, where indices are calculated using hash functions.

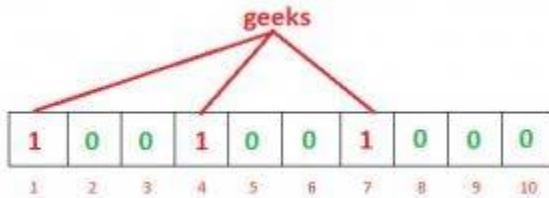
Example – Suppose we want to enter “geeks” in the filter, we are using 3 hash functions and a bit array of length 10, all set to 0 initially. First we’ll calculate the hashes as follows:  $h_1(\text{"geeks"}) \% 10 = 1$

$$h_2(\text{"geeks"}) \% 10 = 4$$

$$h_3(\text{"geeks"}) \% 10 = 7$$

**Note:** These outputs are random for explanation only.

Now we will set the bits at indices 1, 4 and 7 to 1



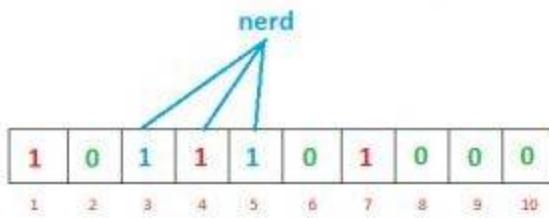
Again we want to enter “nerd”, similarly we’ll calculate hashes

$$h_1(\text{"nerd"}) \% 10 = 3$$

$$h_2(\text{"nerd"}) \% 10 = 5$$

$$h_3(\text{"nerd"}) \% 10 = 4$$

Set the bits at indices 3, 5 and 4 to 1



Now if we want to check “geeks” is present in filter or not. We’ll do the same process but this time in reverse order. We calculate respective hashes using  $h_1, h_2$  and  $h_3$  and check if all these indices are set to 1 in the bit array. If all the bits are set then we can say that “geeks” is **probably present**. If any of the bit at these indices are 0 then “geeks” is **definitely not present**.

## False Positive in Bloom Filters

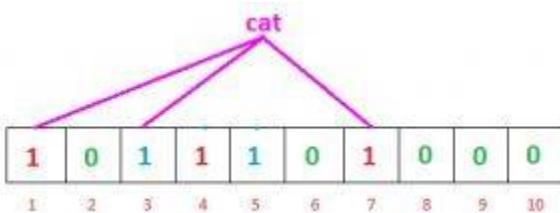
The question is why we said “**probably present**”, why this uncertainty. Let’s understand this with an example. Suppose we want to check whether “cat” is present or not. We’ll calculate hashes using h1, h2 and h3

$$h1(\text{"cat"}) \% 10 = 1$$

$$h2(\text{"cat"}) \% 10 = 3$$

$$h3(\text{"cat"}) \% 10 = 7$$

If we check the bit array, bits at these indices are set to 1 but we know that “cat” was never added to the filter. Bit at index 1 and 7 was set when we added “geeks” and bit 3 was set when we added “nerd”.



So, because bits at calculated indices are already set by some other item, bloom filter erroneously claim that “cat” is present and generating a false positive result.

We can control the probability of getting a false positive by controlling the size of the Bloom filter. More space means fewer false positives. If we want decrease probability of false positive result, we have to use more number of hash functions and larger bit array. This would add latency in addition of item and checking membership.

### **Operations that a Bloom Filter supports**

- `insert(x)` : To insert an element in the Bloom Filter.
- `lookup(x)` : to check whether an element is already present in Bloom Filter with a positive false probability.

**NOTE : We cannot delete an element in Bloom Filter.**

### **Space Efficiency**

If we want to store large list of items in a set for purpose of set membership, we can store it in hashmap, tries or simple array or linked list. All these methods require storing item itself, which is not very memory efficient. For example, if we want to store “geeks” in hashmap we have to store actual string “geeks” as a key value pair {some key : “geeks”}.

Bloom filters do not store the data item at all. As we have seen they use bit array which allow hash collision. Without hash collision, it would not be compact.

## **Counting distinct elements in a stream**

- Suppose stream elements are chosen from some universal set. We would like to know how many different elements have appeared in the stream.
- Example: a Web site gathering statistics on how many unique users it has seen in each given month.

## **Flajolet-Martin algorithm**

Sometimes, we need to know how many **UNIQUE** rows exist in a table. In a Relational Database World there is a simple but **high-costly** action for DB engine level (for example **DISTINCT** or **GROUP BY** with subquery). The simple business case for that action is to get amount of unique users who visited your web-site during period of time.

In the modern era of Big Data, Streaming Data, IoT we often face the speed of our request to the database and honestly we do not need to get the exact amount of unique users at the period of time. Just approximate number is enough for our needs. To help us with our task we can use a really awesome probabilistic structure.

Let me present an example of **Flajolet-Martin algorithm** (aka “**FM**” or “**LogLog** counting of large cardinalities”)

## Inputs

- A static or stream bag (~ multiset) of integers.
- A hash function.

### Example 1: Steps of FM-algorithm

#### Step #1

To calculate a hash for each element from a bag

$x_i$	$h(x_i)$
1	$(3 \cdot 1 + 1) \bmod 5 = 4$
3	$(3 \cdot 3 + 1) \bmod 5 = 0$
5	$(3 \cdot 5 + 1) \bmod 5 = 1$
7	$(3 \cdot 7 + 1) \bmod 5 = 2$
5	$(3 \cdot 5 + 1) \bmod 5 = 1$
2	$(3 \cdot 2 + 1) \bmod 5 = 2$
7	$(3 \cdot 7 + 1) \bmod 5 = 2$

## Step #2

To write a binary representation of a hash value

1	4	100
3	0	000
5	1	001
7	2	010
5	1	001
2	2	010
7	2	010

## Step #3

To calculate the count of trailing zeros in each binary representation of a hash value

1	100	2
3	000	0
5	001	0
7	010	1
5	001	0
2	010	1
7	010	1

#### Step #4

Let's estimate **unique** elements by formula

$$R = 2^{\max(r_i)} = 2^2 = 4$$

By fact, distinct values is 5. So, almost equal

#### Example 2: Give problem in Flajolet-Martin (FM) Algorithm to count distinct elements in a stream.

To estimate the number of different elements appearing in a stream, we can hash elements to integers interpreted as binary numbers. 2 raised to the power that is the longest sequence of 0's seen in the hash value of any stream element is an estimate of the number of different elements.

Eg. Stream: 4, 2, 5, 9, 1, 6, 3, 7

Hash function,

$$h(x) = x + 6 \bmod 32$$

$$h(x) = x + 6 \bmod 32$$

$$h(4) = (4) + 6 \bmod 32 = 10 \bmod 32 = 10 = (01010)$$

$$h(2) = (2) + 6 \bmod 32 = 8 \bmod 32 = 8 = (01000)$$

$$h(5) = (5) + 6 \bmod 32 = 11 \bmod 32 = 11 = (01011)$$

$$h(9) = (9) + 6 \bmod 32 = 15 \bmod 32 = 15 = (01111)$$

$$h(1) = (1) + 6 \bmod 32 = 7 \bmod 32 = 7 = (00111)$$

$$h(6) = (6) + 6 \bmod 32 = 12 \bmod 32 = 12 = (01110)$$

$$h(3) = (3) + 6 \bmod 32 = 9 \bmod 32 = 9 = (01001)$$

$$h(7) = (7) + 6 \bmod 32 = 13 \bmod 32 = 13 = (01101)$$

Trailing zero's {1, 3, 0, 0, 0, 1, 0, 0}

r = max [Trailing Zero] = 3

Output R=  $2^r = 2^3 = 8$

## Estimating moments

- ▶ Goal: Computing moment

Computing distribution of frequencies of different elements in stream

- ▶ Example:

1<sup>st</sup> moment=Sum of all  $m_i$ =length of the stream

2<sup>nd</sup> moment=Sum of all  $m_i^2$ =Surprise Number

- ▶ Formula:

$F = \text{Sum of } (m_i)^k$

where,

K=moment number

$m_i$ =number of times value I occurs

Example:

- ▶ Consider data stream:

$$5,5,5,5,5 = 5^2 + 5^2 + 5^2 + 5^2 + 5^2 = 5 * 5^2 = 125$$

$$9,9,1,1,5 = 2 * 9^2 + 2 * 1^2 + 5^2 = 189$$

$$10,9,9,9,9,9,9,9,9 = 10^2 + 10 * 9^2 = 910$$

## The Alon-Matias-Szegedy Algorithm for Second Moments

- ▶ N observations in stream
- ▶ Choose k random positions  $p_j \in [1, 2, \dots, N]$
- ▶ When reaching position  $p_j$ :
  - Store object at position
  - Store counting occurrences of object
- ▶ Estimate:  $M_2 = N/k(\text{sum of } (2m_i - 1))$

### Solved problem

- ▶ Consider the problem:  
 $a, b, c, b, d, a, c, d, a, b, d, c, a, a, b$
- ▶ By using second moment:  
$$a^2 + b^2 + c^2 + d^2 = 5^2 + 4^2 + 3^2 + 3^2 = 59$$

### Same problem solved by AMS algorithm

- ▶ Given : {a,b,c,b,d,a,c,d,a,b,d,a,b,d,c,a,a,b}

- ▶ Solution:

Length of stream n=15

Choose 3 random positions with different values say c,d,a

{a,b,c,b,d,a,c,d,a,b,d,c,a,a,b}

Count the number of times of occurrences from that position

X1=(c,3)

X2=(d,2)

X3=(a,2)

- ▶ Calculate the estimate:

$$\text{Estimate} = (n \times (2^{\text{X value}} - 1))$$

$$X1 \text{ Estimate} = 15 \times (2^3 - 1) = 75$$

$$X2 \text{ Estimate} = 15 \times (2^2 - 1) = 45$$

$$X3 \text{ Estimate} = 15 \times (2^2 - 1) = 45$$

- ▶ Calculate the average of X1,X2,X3:

$$\text{Avg} = (\text{Sum of the estimates}) / 3$$

$$= (75 + 45 + 45) / 3$$

$$= 55$$

## Advantage and disadvantages

► Advantages:

Simple

Needs to store only k counters

Larger the value of k->Accuracy increases

► Disadvantages:

N value not known

### Counting oneness in the window:

### THE NUMBER OF 1's IN THE DATA STREAM

Now let us suppose we have a window of length N (say N=24) on a binary system, We want at all times to be able to answer a query of the form “ How many 1's are there in the last K bits?” for  $K \leq N$ .

Here comes the **DGIM Algorithm** into picture:

### DGIM algorithm (Datar-Gionis-Indyk-Motwani Algorithm)

Designed to find the number 1's in a data set. This algorithm uses  $O(\log^2 N)$  bits to represent a window of N bit, allows to estimate the number of 1's in the window with an error of no more than 50%.

So this algorithm gives a 50% precise answer.

In DGIM algorithm, each bit that arrives has a timestamp, for the position at which it arrives. if the first bit has a timestamp 1, the second bit has a timestamp 2 and so on.. the positions are recognized with the window size N (the window sizes are usually taken as a multiple of 2).The windows are divided into buckets consisting of 1's and 0's.

## **RULES FOR FORMING THE BUCKETS:**

1. The right side of the bucket should always start with 1. (if it starts with a 0,it is to be neglected) E.g. .  
1001011 → a bucket of size 4 , having four 1's and starting with 1 on it's right end.
2. Every bucket should have at least one 1, else no bucket can be formed.
3. All buckets should be in powers of 2.
4. The buckets cannot decrease in size as we move to the left. (move in increasing order towards left)

Let us take an example to understand the algorithm.

Estimating the number of 1's and counting the buckets in the given data stream.

NO. \_\_\_\_\_

Date \_\_\_\_\_

..... 10101100010110110010110 .....

$N = 24$  (window size)

.... 101011 000 10111 0 11 00 101 1 0 ....

2 2 1 1 0  
2 2 2 2 2 ← size of the buckets  
(4) (4) (2) (2) (2)

This picture shows how we can form the buckets based on the number of ones by following the rules.

In the given data stream let us assume the new bit arrives from the right. When the new bit = 0

Timestamps → 87 ..... 92 ..... 95 ..... 98...100 101 102 103 104  
101011 000 10111 0 11 00 101 1 0 ← 0 | 1 | 1 |

When new bit = 0 enters :

87 ... 92 ..... 95 ..... 98...100...101 102 103 104  
101011 000 10111 0 11 00 101 1 0 0 ← 1 | 1 |

After the new bit ( 0 ) arrives with a time stamp 101, there is no change in the buckets.

But what if the new bit that arrives is 1, then we need to make changes..

Date . . .										
When the new bit = 1 enters:									New Bit =	
87	92	95	98	100	101	102	103	104		
101011	000	10111	011	00	101	1	00	1	← 1	1
2 <sup>2</sup>	2 <sup>2</sup>	2 <sup>1</sup>	2 <sup>1</sup>	2 <sup>0</sup>	2 <sup>0</sup>	2 <sup>0</sup>				
( CREATING A NEW BUCKET OF SIZE 1 )										
when next bit = 1 enters:										
87	92	95	98	100	101	102	103	104		
101011	000	10111	011	00	101	1	00	1	← 1	
2 <sup>2</sup>	2 <sup>2</sup>	2 <sup>1</sup>	2 <sup>1</sup>	2 <sup>0</sup>	2 <sup>0</sup>	2 <sup>0</sup>	2 <sup>0</sup>			
87	92	95	98	102	103	104				
101011	000	10111	011	00	101	1001	1	← 1		
2 <sup>2</sup>	2 <sup>2</sup>	2 <sup>1</sup>	2 <sup>1</sup>			2 <sup>0</sup>	2 <sup>0</sup>			
87	92	98	102	103	104					
101011	000	10111	100101	1001	1			← 1		
2 <sup>2</sup>	2 <sup>2</sup>	2 <sup>2</sup>	2 <sup>1</sup>	2 <sup>1</sup>	2 <sup>0</sup>					

- Create a new bucket with the current timestamp and size 1.

□ If there was only one bucket of size 1, then nothing more needs to be done. However, if there are now three buckets of size 1( buckets with timestamp 100,102, 103 in the second step in the picture) We fix the problem by combining the leftmost(earliest) two buckets of size 1. (purple box)

To combine any two adjacent buckets of the same size, replace them by one bucket of twice the size. The timestamp of the new bucket is the timestamp of the rightmost of the two buckets.

Now, sometimes combining two buckets of size 1 may create a third bucket of size 2. If so, we combine the leftmost two buckets of size 2 into a bucket of size 4. This process may ripple through the bucket sizes.

How long can you continue doing this...

You can continue if current timestamp- leftmost bucket timestamp of window  $< N$  ( $=24$  here) E.g.  $103 - 87 = 16 < 24$  so I continue, if it greater or equal to then I stop.

Finally the answer to the query.

How many 1's are there in the last 20 bits?

Counting the sizes of the buckets in the last 20 bits, we say, there are 11 ones.

## **Decaying Window Algorithm**

This algorithm allows you to identify the most popular elements (trending, in other words) in an incoming data stream.

The decaying window algorithm not only tracks the most recurring elements in an incoming data stream, but also discounts any random spikes or spam requests that might have boosted an element's frequency. In a decaying window, you assign a score or weight to every element of the incoming data stream. Further, you need to calculate the aggregate sum for each distinct element by adding all the weights assigned to that element. The element with the highest total score is listed as trending or the most popular.

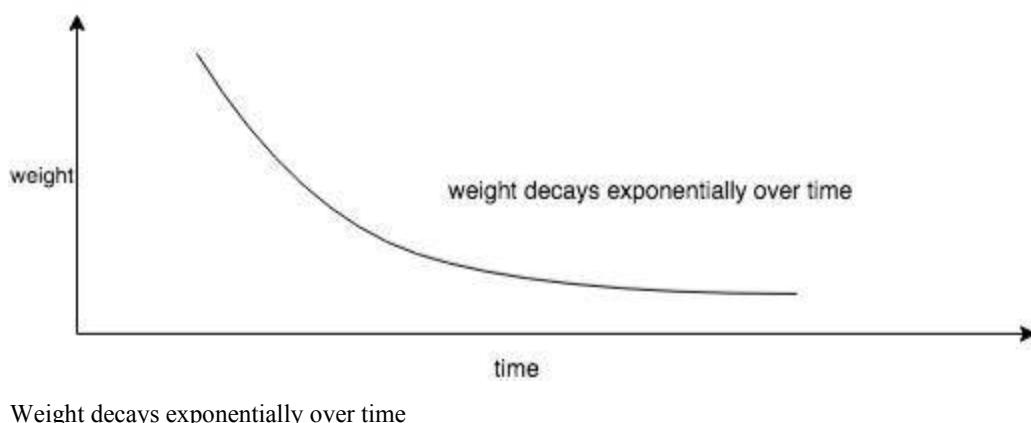
1. Assign each element with a weight/score.
2. Calculate aggregate sum for each distinct element by adding all the weights assigned to that element.

In a decaying window algorithm, ***you assign more weight to newer elements***. For a new element, you first reduce the weight of all the existing elements by a constant factor  $k$  and then assign the new element with a specific weight. The aggregate sum of the decaying exponential weights can be calculated using the following formula:

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

Here,  $c$  is usually a small constant of the order  $10^{-6}$  or  $10^{-9}$ . Whenever a new element, say  $a_{t+1}$ , arrives in the data stream you perform the following steps to achieve an updated sum:

1. Multiply the current sum/score by the value  $(1-c)$ .
2. Add the weight corresponding to the new element.



In a data stream consisting of various elements, you maintain a separate sum for each distinct element. For every incoming element, you multiply the sum of all the existing elements by a value of  $(1-c)$ . Further, you add the weight of the incoming element to its corresponding aggregate sum.

A threshold can be kept to, ignore elements of weight lesser than that.

Finally, the element with the highest aggregate score is listed as the most popular element.

## Example

For example, consider a sequence of twitter tags below:

fifa, ipl, fifa, ipl, ipl, ipl, fifa

Also, let's say each element in sequence has weight of 1.

Let's  $c$  be 0.1

The aggregate sum of each tag in the end of above stream will be calculated as below:

*fifa*

$$\text{fifa} - 1 * (1-0.1) = 0.9$$

$$\text{ipl} - 0.9 * (1-0.1) + 0 = 0.81 \text{ (adding 0 because current tag is different than fifa)}$$

$$\text{fifa} - 0.81 * (1-0.1) + 1 = 1.729 \text{ (adding 1 because current tag is fifa only)}$$

$$\text{ipl} - 1.729 * (1-0.1) + 0 = 1.5561$$

$$\text{ipl} - 1.5561 * (1-0.1) + 0 = 1.4005$$

$$\text{ipl} - 1.4005 * (1-0.1) + 0 = 1.2605$$

$$\text{fifa} - 1.2605 * (1-0.1) + 1 = 2.135$$

*ipl*

$$\text{fifa} - 0 * (1-0.1) = 0$$

$$\text{ipl} - 0 * (1-0.1) + 1 = 1$$

$$\text{fifa} - 1 * (1-0.1) + 0 = 0.9 \text{ (adding 0 because current tag is different than ipl)}$$

$$\text{ipl} - 0.9 * (1-0.01) + 1 = 1.81$$

$ipl - 1.81 * (1-0.01) + 1 = 2.7919$

$ipl - 2.7919 * (1-0.01) + 1 = 3.764$

$fifa - 3.764 * (1-0.01) + 0 = 3.7264$

In the end of the sequence, we can see the score of **fifa is 2.135** but **ipl is 3.7264**

So, **ipl** is more trending than **fifa**

Even though both of them occurred same number of times in input there score is still different.

### Advantages of Decaying Window Algorithm:

1. Sudden spikes or spam data is taken care.
2. New element is given more weight by this mechanism, to achieve right trending output.

## Real Time Analytics Definition

Real time analytics lets users see, analyze and understand data as it arrives in a system. Logic and mathematics are applied to the data so it can give users insights for making real-time decisions.

Understanding live analytics is best done by breaking down the terms:

- **Real-time:** operations are performed milliseconds before it becomes available to the user
- **Analytics:** a software capability to pull data from various sources and interpret, analyse and transform it into a format that is comprehensible by humans

Without real-time analytics, a business may absorb a ton of data that gets lost in the shuffle. Leading a finance team means leveraging data for both financial statement procurement, as well as to understand insights about the business and its customers' needs. The ability to work in real-time and respond to a customers' needs or prevent issues before they arise ends up benefitting the bottom line by reducing risk and enhancing accuracy.

In order for real-time data analytics to work, the software generally includes the following components:

- **Aggregator:** Pulls real-time data analytics from various sources
- **Analytics Engine:** Compares the values of data and streams it together while performing analysis
- **Broker:** Create the availability of data
- **Stream Processor:** Executes logic and performs analytics in real-time by receiving and sending data

## RTAP

- Manages and processes data and helps timely decision-making
- Helps to develop dynamic analysis applications
- Leads to evolution of business intelligence

### Widely used RTAPs

**Apache SparkStreaming**—a Big Data platform for data stream analytics in real time.

**Cisco Connected Streaming Analytics (CSA)**—a platform that delivers insights from high-velocity streams of live data from multiple sources and enables immediate action.

**Oracle Stream Analytics (OSA)**—a platform that provides a graphical interface to “Fast Data”.

**SAP HANA**— a streaming analytics tool which also does real-time analytics.

**SQL streamBlaze**—an analytics platform, offering a real-time, easy-to-use and powerful visual development environment for developers and analysts.

**TIBCO StreamBase**—streaming analytics, which accelerates action in order to quickly build applications.

**Informatica** — a real-time data streaming tool which transforms a torrent of small messages and events into unprecedented business agility.

**IBM Stream Computing**—a data streaming tool that analyzes a broad range of streaming data—unstructured text, video, audio, geospatial, sensor— helping organizations spot the opportunities and risks and make decisions in real time

### **Benefits of Real-Time Data Analytics**

You can minimise risks, reduce costs, and understand more about your employees, customers, and overall financial health of the business with the aid of real-time data.

Here are some of the key benefits:

- **Data visualisation:** With historical data, you can get a snapshot of information displayed in a chart. However, with real-time data, you can use data visualisations that reflect changes within the business as they occur in real-time. This means that dashboards are interactive and accurate at any given moment. With custom dashboards, you can also share data easily with relevant stakeholders so that decision-making never gets put on hold.

- **Competitive advantages:** You can easily understand benchmarks and view trends to make the wisest choices to boost your business.
- **Precise information:** Since real-time data analytics is focused on creating outcomes, there is no wasted effort. Rather than spending resources, time and money collecting data that's unnecessary, the software is set up to capture only the data you need.
- **Testing:** With the ability to test how changes will affect your business' processes in real-time, you can take calculated risks. As you make changes, you can gauge if there's any issues or negative effects and be able to revert and try again without undergoing too much damage.
- **Monitor customer behaviour:** With knowledge and insights about customer behavior, you can dive deep into customer behaviors and be able to monitor what is and isn't working to your business' advantage.
- **Lower costs:** you can lower the costs of hiring coding experts to take advantage of business data, reduce bottlenecks within processes and ensure team members have what to pull insights from the data.
- **Apply machine learning:** Machine learning improves as more data enters the system. Rather than requiring a human to update algorithms and spend time on tedious tasks, the machine manages to become more efficient as time goes on.
- **Drive better decision making:** Ultimately, one of the biggest benefits of real-time data analytics is the ability to move forward on both small and big decisions in a timely and productive manner. Through accurate insights, you can strip, update and introduce new business ideas and processes to your organization with little risk as the analytics provides you with all the necessary information to make sound business decisions.

## **RTAP Applications**

1. Fraud detection systems for online transactions
2. Log analysis for understanding usage pattern
3. Click analysis for online recommendations
4. Social Media Analytics

## **Sentiment Analysis: a case study**

**Opinion Mining** or also **Sentiment Analysis** is the computational study of opinions, sentiments and emotions expressed in texts

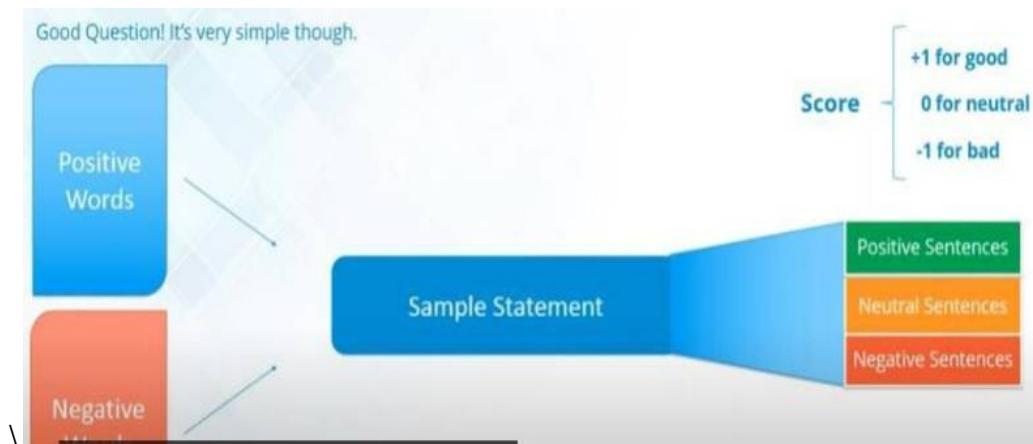
It is the detection of **attitudes**

Why opinion mining now?

Mainly because of the Web

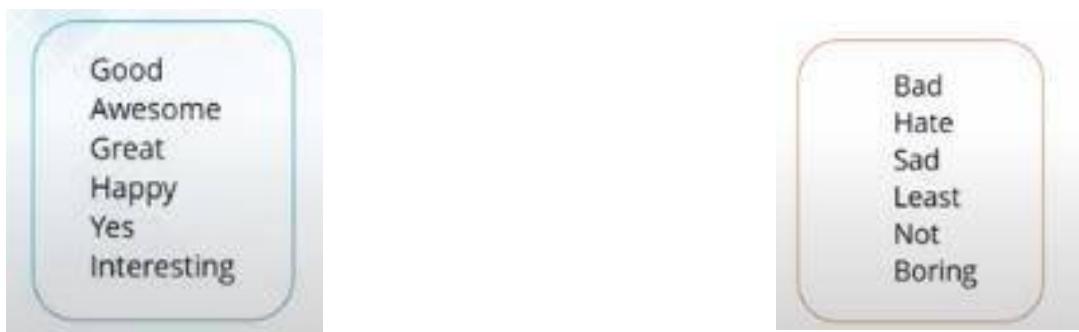
Huge volumes of opinionated text

## How does it work



## Example

- The iphone7 is **awesome!** = +1



- This movie is **not** that **great**, after the interval it was **boring!** = -1+1-1=-1

Let's take a complex statement now,

The service was terrible, but the food was great!

Such a case is called **Constructive Conjunction**.

The way we solve it is like this, whenever there is a BUT in the sentence, it reforms the sentence like this,

1.

The service was terrible

2.

AND

But the food was great!

Hence, calculate their scores separately, this method is called Binary Sentiment Analysis

## Sentiment Analysis – applications

**Businesses and organizations:** Market intelligence

Business spends a huge amount of money to find consumer sentiments and opinions

Consultants, surveys and focused groups

**Individuals:** interested in other's opinions when

Purchasing a product or using a service

Finding opinions on political topics

**Opinion retrieval/search:** providing general search for opinions

Search engines do not search for opinions

Opinions are hard to express with a few keywords

### **SA in Twitter**

Capture the *sentiment* expressed in short messages

No limit to the range of information conveyed by tweets

Often used to share opinions and sentiments

Twitter is seen today as an instrument to spread

messages VIPs use it to communicate with fans

They want to know if they are appreciated or  
not Politicians use it to make their claims

They want to know the reaction of people to them

Companies use it to create a direct channel with customers

They want to have an indicator of user's happiness

### **Two task for SA in Twitter**

Task A

#### **Contextual Polarity Disambiguation**

Given a tweet and a span on it classify the instance with respect to *positive*,  
*negative* o *neutral*

Task B

#### **Message Polarity Classification**

Given a tweet classify it with respect to *positive*, *negative* o *neutral*

### **Task A examples**

@\_Ms\_R have you watched Four Lions? Saw it again last night, love it its  
not that I'm a GSP fan, i just hate Nick Diaz. can't wait for february.

Going to Singapore tonight :) Excited for Skyfall + penny boarding  
tomorrow! Cowboys will beat the falcons sunday #iStamp

Contraband was prolly the worst below average movie I ever sat and finished watching in my life

### **Task B examples**

**negative** I officialy hate Windows 7, it just sucks on my laptop. Back to XP tomorrow! (and also a huge delay for the movie again :/, was so close!)

**negative** Desperation Day (February 13th) the most well known day in all mens life. **positive** #NBA I'm excited :)

**positive** Eli manning best 4th quarter QB in the game!

**neutral** Harry Redknapp is being heavily linked with the position as Blackburn manager today. However, QPR may have begun courting him already...

**neutral** 20 June 2012, out European Tour party drove the fabulous road from Davos to Stelvio. This is the last little bit... <http://fb.me/1ZkWMVYsV>

## **Stock market prediction**

It is the act of trying to determine the future value of a company stock or other financial instrument traded on an exchange. The successful prediction of a stock's future price could yield significant profit.

Different machine learning algorithms are used to predict the stock market trading.

Use text from different sources and use Text and Data Mining (TDM) to extract pattern or information or any hidden data of interest to predict the Ups and downs of the targeted stocks.

**Then**

Data Mining Isn't a Good Bet For Stock-Market Predictions [2]  
Aug. 8, 2009 - JASON ZWEIG , Wall Street Journal

**Now**

How Traders Are Using Text and Data Mining to Beat the Market [3]  
Feb 12 2015 - Market Roy Kaufman , The Street

**Applying Machine Learning to Stock Market Trading** - Bryce Taylor [1]

Machine learning algorithm to read headlines from financial news magazines and make predictions on the directional change of stock prices after a moderate-length time interval  
[Stanford Student project 2013, CS 229]

Machine learning algorithms like Bayesian classification, SVM, Regression etc. can be used for stock market prediction.