UNIT-II

Stream Processing: Mining data streams: Introduction to Streams Concepts, Stream Data Model and Architecture, Stream Computing, Sampling Data in a Stream, Filtering Streams, Counting Distinct Elements in a Stream, Estimating Moments, Counting Oneness in a Window, Decaying Window, Real time Analytics Platform (RTAP) Applications, Case Studies - Real Time Sentiment Analysis - Stock Market Predictions

ASSIGNMENT QUESTIONS

Stream Processing: Mining data streams: Introduction to Streams Concepts:

1. What is meant by sampling of stream data? What factors need to consider in sampling? **[7M-R20-SET-2-July 2023] [Remembering]**
2. What are the Characteristics of Data streaming models? **[Remembering]**

Stream Data Model and Architecture, Stream Computing:

3. Explain the Stream Data Model and Architecture. **[7M-R20-SET-1,4-July 2023][Understanding]**
4. Describe the Streaming Computing? **[Create]**

Sampling Data in a Stream, Filtering Streams:

5. Explain in detailed about the Sampling data in a stream. **[Understanding]**
6. How sampling and filtering can be performed in streams? Illustrate. **[7M-R20-SET-1-July 2023] [Remembering]**

Counting Distinct Elements in a Stream, Estimating Moments

7. Describe the following with respect to streams: **[7M-R20-SET-2-July 2023] [Create]**
   i) Estimating moments
   ii) Counting distinct elements
8. Explain the concepts of filtering stream and counting distinct elements of a stream. **[7M-R20-SET-3-July 2023] [Understanding]**

Counting Oneness in a Window, Decaying Window, Real time Analytics Platform (RTAP) Applications:

9. Explain Counting Oneness in a Window, Decaying Window. **[7M-R20-SET-4-July 2023] [Understanding]**
10. Write about Real Time Analytical Platform and its applications in detail? **[7M-R20-SET-3-July 2023] [Create]**

Case Studies - Real Time Sentiment Analysis - Stock Market Predictions:

11. Describe briefly in case studies of Real time Analysis. **[Create]**
12. Describe briefly in case studies of Stock market Predictions? **[Create]**

Stream Processing

Golab &Oszu (2003): "A data stream is a real-time, continuous, ordered (implicitly by arrival time or explicitly by timestamp) sequence of items. It is not impossible to control the order in which items arrive, nor is it feasible to locally store a stream in its entirety."

Massive volumes of data, items arrive at a high rate.

A **data stream** is a (potentially unbounded) sequence of tuples. Each tuple consist of a set of attributes, similar to a row in database table.

Transactional data streams: log interactions between entities

- Credit card: purchases by consumers from merchants
- Telecommunications: phone calls by callers to dialed parties
- Web: accesses by clients of resources at servers

Measurement data streams: monitor evolution of entity states

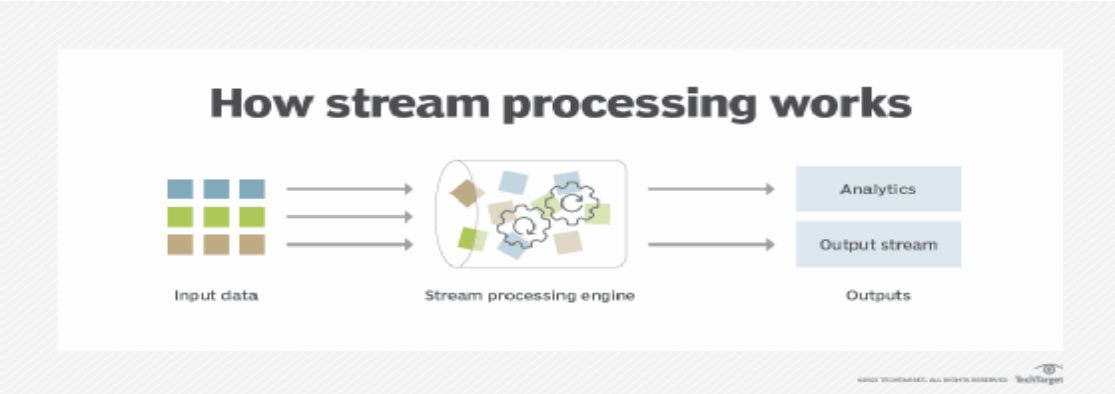
- Sensor networks: physical phenomena, road traffic
- IP network: traffic at router interfaces
- Earth climate: temperature, moisture at weather stations

Stream processing is a data management technique that involves ingesting a continuous data stream to quickly analyze, filter, transform or enhance the data in real time.

Once processed, the data is passed off to an application, data store or another stream processing engine.

Stream processing architectures help simplify the data management tasks required to consume, process and publish the data securely and reliably.

These actions can include processes such as analyzing, filtering, transforming, combining or cleaning data



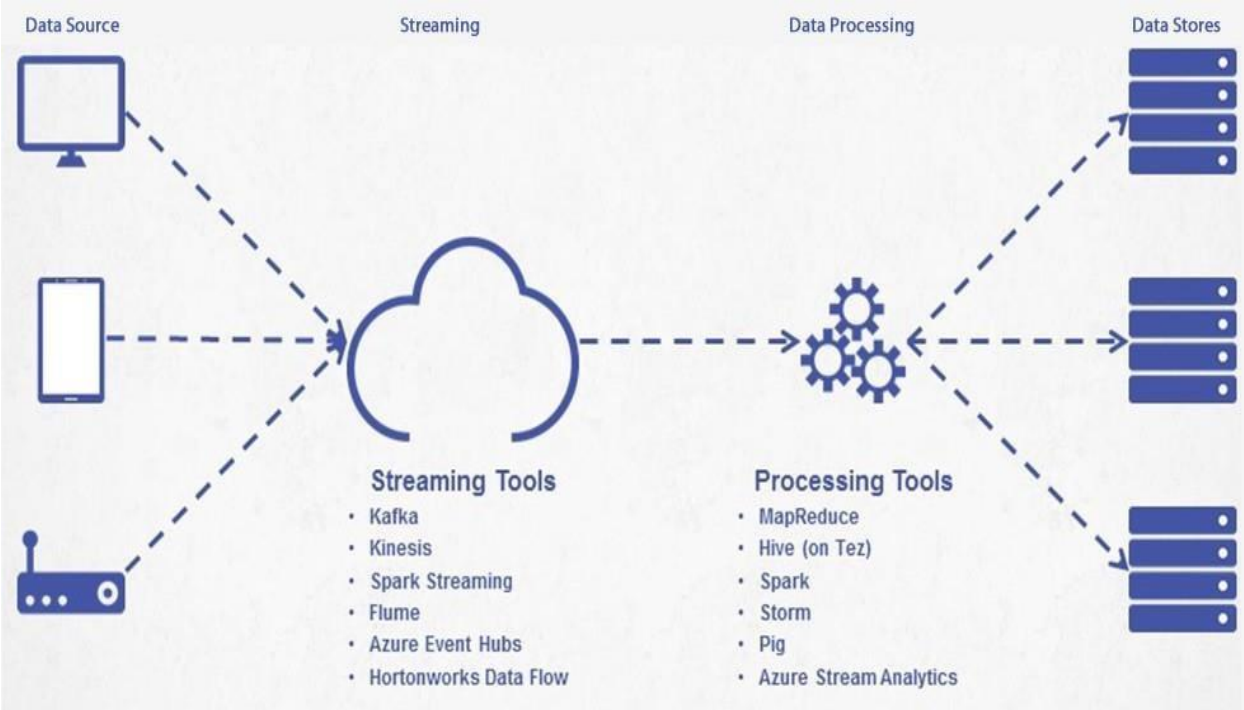
Why is stream processing needed?

- Develop adaptive and responsive applications
- Help enterprises improve real-time business analytics
- Facilitate faster decisions
- Accelerate decision-making
- Improve decision-making with increased context

- Improve the user experience
- Create new applications that use a wider variety of data sources

Characteristics of Data Streams:

- Huge volumes of continuous data, possibly infinite
- Fast changing and requires fast, real-time response
- Data stream captures nicely our data processing needs of today
- Random access is expensive—single scan algorithm (*can only have one look*)



Data stream management system (DSMS)

- Is a computer software system to manage continuous data streams.
- 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.

DBMS versus DSMS (Data Stream Management System)

• Persistent relations	• Transient streams
• One-time queries	• Continuous queries
• Random access	• Sequential access
• "Unbounded" disk store	• Bounded main memory
• Only current state matters	• Historical data is important
• No real-time services	• Real-time requirements
• Relatively low update rate	• Possibly multi-GB arrival rate
• Data at any granularity	• Data at fine granularity
• Assume precise data	• Data stale/imprecise
• Access plan determined by query processor, physical DB design	• Unpredictable/variable data arrival and characteristics

```
SELECT AVG(price) FROM examplestream [SIZE 10 ADVANCE 1 TUPLES] WHERE value > 100.0
```

This stream continuously calculates the average value of "price" of the last 10 tuples, but only considers those tuples whose prices are greater than 100.0.

How is stream processing used?

Stream processing can reduce data transmission and storage costs by distributing processing across edge computing infrastructure.

Streaming data architectures can also make it easier to integrate data from multiple business applications or operational systems.

For example, telecom service providers are using stream processing tools to combine data from numerous operations support systems. Healthcare providers use them to integrate applications that span multiple medical devices, sensors and electronic medical records systems.

Spark, Flink and Kafka Streams are the most common open source stream processing frameworks

For example, Apache Kafka is a popular open source publishes-subscribe framework that simplifies integrating data across multiple applications.

Apache Kafka Streams is a stream processing library for creating applications that ingest data from Kafka, process it and then publish the results back to Kafka as a new data source for other applications to consume.

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.

We then show how to estimate the number of different elements in a stream using much less storage than would be required if we listed all the elements we have seen.

Anotherapproachtosummarizingastreamistolookatonlyafixed-length–window‖consisting of the last n elements for some (typicallylarge) n.

2.2 TheStreamDataModel

Let us begin by discussing the elements of streams and stream processing. Weexplain thedifference between streams and databases and the special problems that arise when dealing withstreams.Some typical applications wherethestream model applieswill beexamined.

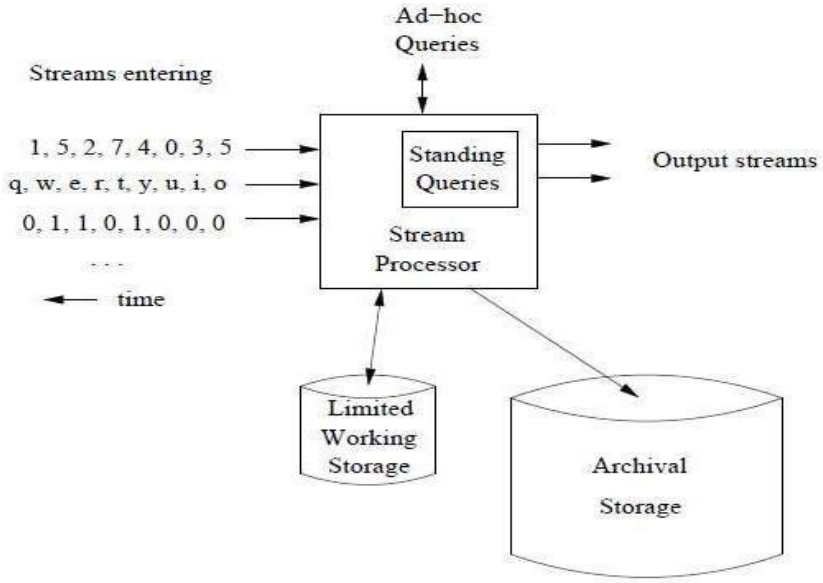


Figure 4.1: A data-stream-management system

2.2.1 :AData-Stream-ManagementSystem

In analogy to a database-management system, we can view a stream processor as a kind of data-management system,the high-level organization of which is suggested inFig.4.1.Anynumber of streams can enter the system. Each stream can provide elements at its own schedule;

They neednot 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 th processing of data that goes on with in a database-management system.
The latter system controls the rate at which data is read from the disk,and there forenever 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.

The working store might be disk, or it might be main memory, depending on how fast we need to process queries.

SensorData

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.

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.

A million sensors is n't very many; there would be one for every 150 square miles of ocean. Now we have 3.5 terabytes arriving every day.

ImageData

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.

InternetandWebTraffic

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 retainit 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.

For example, an increase in queries like ―sore throat‖ enables us to track the spread of viruses

4.1.3 Stream Queries

There are two ways that queries get asked about streams. We show in Fig. 4.1 a place within the processor where standing queries are stored. These queries are, in a sense, permanently executing, and produce outputs at appropriate times

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

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

Example 4.2 : 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.

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.

2.1.4 Issues in Stream Processing

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.

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.

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.

4.2 Sampling Data in a Stream

Extracting reliable samples from a stream.Stream sampling is the process of collecting a representative sample of the elements of a data stream.

4.2.1 A Motivating Example

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.

A search engine receives a stream of queries, and it would like to study the behavior of typical users. We assume the stream consists of tuples (user, query,time).

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.

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 $1/10^{th}$ 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 $1/10^{th}$ of the stream that is selected, while the other is in the 9/10th that is not selected.

The correct answer to the query about the fraction of repeated searches is d/(s+d). However, the answer we shall obtain from the sample is d/(10s+19d).

To derive the latter formula, note that d/100 appear twice, while s/10+18d/100 appear once. Thus, the fraction appearing twice in the sample is d/100 divided by d/100+ s/10 + 18d/100. This ratio is d/(10s+ 19d). For no positive values of s and d is d/(s + d) = d/(10s + 19d).

4.2.2 Obtaining a Representative Sample

Many queries about the statistics of typical users cannot be answered by taking a sample of each user's search queries.

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.

if we have no record of ever having seen this user before, then we generate a random integer between 0 and 9.

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.

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.

4.2.3 The General Sampling Problem

Our stream consists of tuples with n components. A subset of the components is 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.

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.

4.2.4 Varying the Sample Size

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, . . . ,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) ≤ 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.

4.3 Filtering Streams

Generated reports and query results from database tools often result in large and complex data sets. Redundant or impartial pieces of data can confuse or disorient a user.

Filtering data can also make results more efficient

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.

4.3.1 A Motivating Example

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.

Since there are one billion members of S, approximately 1/8th of the bits will be 1. The exact fraction of bits set to 1 will be slightly less than 1/8th, because it is possible that two members of S hash to the same bit.

When a stream element arrives, we hash its email address. If the bit to which that email address hashes is 1, then we let the email through.

4.3.2 The Bloom Filter

A Bloom filter consists of:
1. An array of n bits, initially all 0's.
2. A collection of hash functions h1, h2, . . . ,hk. Each hash function maps ―key‖ values to n buckets, corresponding to the n bits of the bit-array.
3. A set S of m key values.

The purpose of the Bloom filter is to allow through all stream elements whose keys are in S,while rejecting most of the stream elements whose keys are not in S.

To initialize the bit array, begin with all bits 0. Take each key value in S and hash it using each of the k hash functions. Set to 1 each bit that is hi(K) for some hash function hi and some key value K in S.

To test a key K that arrives in the stream, check that all of h1(K), h2(K), . . . ,hk(K) are 1's in the bit-array. If all are 1's, then let the stream element through. If one or more of these bits are 0,then K could not be in S, so reject the stream element.

4.3.3 Analysis of Bloom Filtering

If a key value is in S, then the element will surely pass through the Bloom filter. However, if the key value is not in S, it might still pass.

We need to understand how to calculate the probability of a false positive, as a function of n, the bit-array length, m the number of members of S, and k, the number of hash functions.

The analysis is similar to the analysis in Section 3.4.2, and goes as follows:

- The probability that a given dart will not hit a given target is $(x-1)/x$.
- The probability that none of the y darts will hit a given target is $\left(\frac{x-1}{x}\right)^y$. We can write this expression as $(1-\frac{1}{x})^{x(\frac{y}{x})}$.
- Using the approximation $(1-\epsilon)^{1/\epsilon} = 1/e$ for small ϵ (recall Section 1.3.5), we conclude that the probability that none of the y darts hit a given target is $e^{-y/x}$.

Suppose we used the same S and the same array, but used two different hash functions. This situation corresponds to throwing two billion darts at eight billion targets, and the probability that a bit remains 0 is e−1/4.

In order to be a false positive, a nonmember of S must hash twice to bits that are 1, and this probability is $(1 - e^{-1/4})^2$, or approximately 0.0493.

Thus, adding a second hash function for our running example is an improvement, reducing the false-positive rate from 0.1175 to 0.0493.

4.3.4 Exercises for Section 4.3

Exercise 4.3.1 : For the situation of our running example (8 billion bits, 1 billion members of the set S), calculate the false-positive rate if we use three hash functions? What if we use four hash functions?

Exercise 4.3.3 : As a function of n, the number of bits and m the number of members in the set S, what number of hash functions minimizes the false positive rate?

4.4 Counting Distinct Elements in a Stream

sampling and filtering – it is some what tricky to do what we want in a reasonable amount of main memory, so we use a variety of hashing and a randomized algorithm to get approximately what we want with little space needed per stream.

4.4.1 The Count-Distinct Problem

How many different elements have appeared in the stream, counting either from the beginning of the stream or from some known time in the past.

Example 4.5 : As a useful example of this problem, consider a Web site gathering statistics on how many unique users it has seen in each given month. The universal set is the set of logins for that site, and a stream element is generated each time someone logs in.

Appropriate for a site like Amazon, where the typical user logs in with their unique login name.

Web site like Google that does not require login to issue a search query, and may be able to identify users only by the IP address from which they send the query.

There are about 4 billion IP addresses,2 sequences of four 8-bit bytes will serve as the universal set in this case.

The obvious way to solve the problem is to keep in main memory a list of all the elements seen so far in the stream.

Keep them in an efficient search structure such as a hash table or search tree, so one can quickly add new elements and check whether or not the element that just arrived on the stream was already seen.

if the number of distinct elements is too great, or if there are too many streams that need to be processed at once (e.g., Yahoo! wants to count the number of unique users viewing each of its pages in a month), then we cannot store the needed data in main memory

4.5.2 The Alon-Matias-Szegedy Algorithm for Second Moments

that a stream has a particular length n. We shall show how to deal with growing streams in the next section. Suppose we do not have enough space to count all the mi's for all the elements of the stream.

We can still estimate the second moment of the stream using a limited amount of space; the more space we use, the more accurate the estimate will be. We compute some number of variables.
For each variable X, we store:

1. A particular element of the universal set, which we refer to as X.element ,and
2. An integer X.value, which is the value of the variable. To determine the value of a variable X,we choose a position in the stream between 1 and n, uniformly and at random.

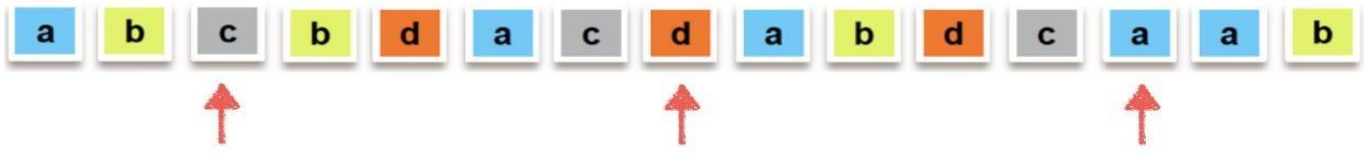
Set X.element to be the element found there, and initialize X.value to 1. As we read the stream, add 1 to X.value each time we encounter another occurrence of X.element.

Example 4.7 : Suppose the stream is a, b, c, b, d, a, c, d, a, b, d, c, a, a, b. The length of the stream is n = 15. Since a appears 5 times, b appears 4 times, and c and d appear three times each,the second moment for the stream is 52+42+32+32 = 59. Suppose we keep three variables, X1,X2, and X3. Also, assume that at ―random‖ we pick the 3rd, 8th, and 13th positions to define these three variables.

AMS algorithm (Alon-Matias-Szegedy) for 2. order moments

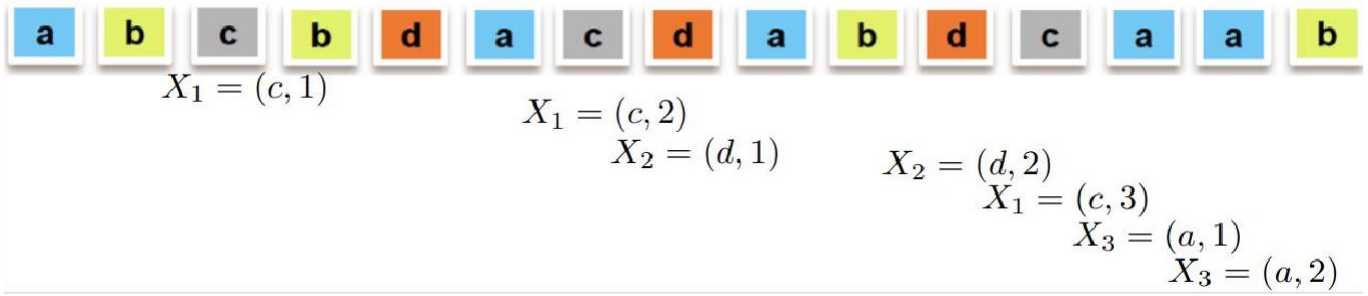
$X = (element, value)$
$X.element$: element of the universal set
$X.value$: counter of $X.element$ in the stream starting at a randomly chosen position

$$n = 15$$
$$m_a^2 + m_b^2 + m_c^2 + m_d^2 = 5^2 + 4^2 + 3^2 + 3^2 = 59$$



(1) **Randomly** pick x positions (here: 3; i.e. 3 variables to compute the 2nd order) from stream with **known length**

the more space we use, the more accurate the estimate



(2) Process the stream, **one element at a time**

- **Estimate of the 2nd order moment** from any $X = (element, value)$:

argument will follow

$$n \times (2 \times X.value - 1)$$

- Applied to our example:

estimate from X_1: $15 \times (2 \times 3 - 1) = 75$
estimate from X_2: $15 \times (2 \times 2 - 1) = 45$
estimate from X_3: $15 \times (2 \times 2 - 1) = 45$

average

$AVG(X_1, X_2, X_3) = 55$

approximate solution

4.5.3 Why the Alon-Matias-Szegedy Algorithm Works

Some notation will make the argument easier to follow. Let e(i) be the stream element that appears at position i in the stream, and let c(i) be the number of times element e(i) appears in the stream among positions i, i + 1, . . . , n.

Example 4.9 : Consider the stream of Example 4.7. e(6) = a, since the 6th position holds a. Also, c(6) = 4, since a appears at positions 9, 13, and 14, as well as at position 6. Note that a also appears at position 1, but that fact does not contribute to c(6).

The expected value of n(2X.value − 1) is the average over all positions i between 1 and n of n(2c(i) − 1), that is

Given: $n \times (2 \times X.value - 1)$

$X.value$: counter of $X.element$ in the stream starting at a randomly chosen position

Expectation: $E(n(2X.value - 1)) = \frac{1}{n} \sum_{i=1}^{n} n \times (2 \times c(i) - 1)$

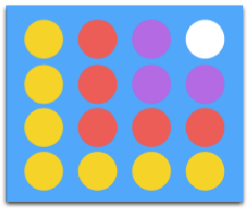
Simplifies to $= \sum_{i=1}^{n} (2 \times c(i) - 1)$

To show: this expression is the 2. order moment

We know: $(m_a)^2 = \sum_a 1 + 3 + 5 + .. + (2m_a - 1)$

$16 = 4^2 = 1 + 3 + 5 + 7$

$E(n(2X.value - 1)) = \sum_a (m_a)^2$



$e(i)$: stream element at position i in the stream
$c(i)$: number of times $e(i)$ appears starting at position i

4.5.4 Higher-Order Moments

We estimate kth moments, for k> 2, in essentially the same way as we estimate second moments. The only thing that changes is the way we derive an estimate from a variable.

we used the formula n(2v − 1) to turn a value v, the count of the number of occurrences of some particular stream element a, into an estimate of the second moment.

this formula works: the terms 2v − 1, for v = 1, 2, . . . ,m sum to m2, where m is the number of times a appears in the stream.

Notice that 2v − 1 is the difference between v2 and (v − 1)2. Suppose we wanted the third moment rather than the second.

Then all we have to do is replace 2v−1 by v3−(v−1)3 =3v2−3v+1. Then Pm v=1 3v2−3v+1 = m3, so we can use as our estimate of the third moment the formula n(3v2−3v+1), where v = X.value is the value associated with some variable X.

More generally, we can estimate kth moments for any k ≥ 2 by turning value v = X

4.5.5 Dealing With Infinite Streams

The estimate we used for second and higher moments assumes that n, the stream length, is a constant. In practice, n grows with time

If we count the number of stream elements seen and store this value, which only requires log n bits, then we have n available whenever we need it.

On the other hand, if we wait too long to pick positions, then early in the stream we do not have many variables and so will get an unreliable estimate.

The discarded variables are replaced by new ones, in such a way that at all times, the probability of picking any one position for a variable is the same as that of picking any other position.

Suppose we have space to store s variables. Then the first s positions of the stream are each picked as the position of one of the s variables.

we have seen n stream elements, and the probability of any particular position being the position of a variable is uniform, that is s/n. When the (n+1)st element arrives, pick that position with probability s/(n+1). If not picked, then the s variables keep their same positions.

However, if the (n+1)st position is picked, then throw out one of the current s variables, with equal probability. Replace the one discarded by a new variable whose element is the one at position n + 1 and whose value is 1.

the probability that position n + 1 is selected for a variable is what it should be: s/(n + 1). By the inductive hypothesis, before the arrival of the (n + 1)st stream element, this probability was s/n.

With probability 1 − s/(n + 1) the (n + 1)st position will not be selected, and the probability of each of the first n positions remains s/n.

However, with probability s/(n + 1), the (n + 1)st position is picked, and the probability for each of the first n positions is reduced by factor (s−1)/s. Considering the two cases, the probability of selecting each of the first n positions is

$$(1 - \frac{s}{n+1})(\frac{s}{n}) + (\frac{s}{n+1})(\frac{s-1}{s})(\frac{s}{n})$$

This expression simplifies to

$$(1 - \frac{s}{n+1})(\frac{s}{n}) + (\frac{s-1}{n+1})(\frac{s}{n})$$

and then to

$$\left((1 - \frac{s}{n+1}) + (\frac{s-1}{n+1})\right)(\frac{s}{n})$$

which in turn simplifies to

$$(\frac{n}{n+1})(\frac{s}{n}) = \frac{s}{n+1}$$

4.5 ESTIMATING MOMENTS

Estimating moments is a generalization of the problem of counting distinct elements in a stream. The problem, called computing "moments," involves the distribution of frequencies of different elements in the stream.

Suppose a stream consists of elements chosen from a universal set. Assume the universal set is ordered so we can speak of the ith element for any i. Let mi be the number of occurrences of the ith element for any i. Then the kth order moment (or just kth moment) of the stream is the sum over all i of (mi)k.

Example 6 : The 0th moment is the sum of 1 for each mi that is greater than 0.4 That is, the 0th moment is a count of the number of distinct elements in the stream

We can use the method to estimate the 0th moment of a stream. 4Technically, since mi could be 0 for some elements in the universal set, we need to make explicit in the definition of "moment" that 00 is taken to be 0.

For moments 1 and above, the contribution of mi's that are 0 is surely 0.

The 1st moment is the sum of the mi's, which must be the length of the stream.

The second moment is the sum of the squares of the mi's.

The Alon-Matias-Szegedy Algorithm for Second Moments:

For now, let us assume that a stream has a particular length n. We shall show how to deal with growing streams in the next section. Suppose we do not have enough space to count all the mi's for all the elements of the stream. We can still estimate the second moment of the stream using a limited amount of space; the more space we use, the more accurate the estimate will be. We compute some number of variables. For each variable X, we store:

1. A particular element of the universal set, which we refer to as X.element , and
2. An integer X.value, which is the value of the variable.

Example 7 : Suppose the stream is a, b, c, b, d, a, c, d, a, b, d, c, a, a, b. The length of the stream is n = 15. Since a appears 5 times, b appears 4 times, and c and d appear three times each, the second moment for the stream is 52+42+32+32 = 59. Suppose we keep three variables, X1, X2, and X3. Also, assume that at "random" we pick the 3rd, 8th, and 13th positions to define these three variables. When we reach position 3, we find element c, so we set X1.element = c and X1.value = 1. Position 4 holds b, so we do not change X1.

Higher-Order Moments:

We estimate kth moments, for k > 2, in essentially the same way as we estimate second moments. The only thing that changes is the way we derive an estimate from a variable. In Section 4.5.2 we used the formula n(2v − 1) to turn a value v, the count of the number of occurrences of some particular stream element a, into an estimate of the second moment.

we saw why this formula works: the terms 2v − 1, for v = 1, 2, . . . ,m sum to m2, where m is the number of times a appears in the stream. Notice that 2v − 1 is the difference between v2 and (v − 1)2. Suppose we wanted the third moment rather than the second. Then all we have to do is replace 2v−1 by v3−(v−1)3 = 3v2−3v+1. Then Pmv=1 3v2−3v+1 = m3, so we can use as our estimate of the third moment the formula n(3v2−3v+1), where v = X.value is the value associated with some variable X.

4.6 Counting Ones in a Window

Suppose we have a window of length N on a binary stream. We want at all times to be able to answer queries of the form ―how many 1's are there in the last k bits?‖ for any $k \leq N$.

4.6.1 The Cost of Exact Counts

Suppose we want to be able to count exactly the number of 1's in the last k bits for any $k \leq N$. Then we claim it is necessary to store all N bits of the window, as any representation that used fewer than N bits could not work.

suppose we have a representation that uses fewer than N bits to represent the N bits in the window. Since there are 2N sequences of N bits, but fewer than 2N representations, there must be two different bit strings w and x that have the same representation.

Since $w \neq x$, they must differ in at least one bit. Let the last $k-1$ bits of w and x agree, but let them differ on the kth bit from the right end.

Example 4.10: If w = 0101 and x = 1010, then k = 1, since scanning from the right, they first disagree at position 1. If w = 1001 and x = 0101, then k = 3, because they first disagree at the third position from the right.

Suppose the data representing the contents of the window is whatever sequence of bits represents both w and x.

Ask the query ―how many 1's are in the last k bits?‖ The query-answering algorithm will produce the same answer, whether the window contains w or x, because the algorithm can only see their representation.

we need N bits, even if the only query we can ask is ―how many 1's are in the entire window of length N?‖ The argument is similar to that used above.

Suppose we use fewer than N bits to represent the window, and therefore we can find w, x, and k as above

if we follow the current window by any $N - k$ bits, we will have a situation where the true window contents resulting from w and x are identical except for the leftmost bit, and therefore, their counts of 1's are unequal.

4.6.2 The Datar-Gionis-Indyk-Motwani Algorithm:

This version of the algorithm uses $O(\log_2 N)$ bits to represent a window of N bits, and allows us to estimate the number of 1's in the window with an error of no more than 50%.

To begin, each bit of the stream has a timestamp, the position in which it arrives. The first bit has timestamp 1, the second has timestamp 2, and so on.

Since we only need to distinguish positions within the window of length N, we shall represent timestamps modulo N, so they can be represented by $\log_2 N$ bits.

If we also store the total number of bits ever seen in the stream (i.e., the most recent timestamp) modulo

N, then we can determine from a timestamp modulo N where in the current window the bit with that timestamp is.

We divide the window into buckets, consisting of:

1. The timestamp of its right (most recent) end
2. The number of 1's in the bucket. This number must be a power of 2, and we refer to the number of 1's as the size of the bucket

To represent a bucket, we need log2 N bits to represent the timestamp (modulo N) of its right end. To represent the number of 1's we only need log2 log2 N bits.

The reason is that we know this number i is a power of 2, say 2j , so we can represent i by coding j in binary. Since j is at most log2 N, it requires log2 log2 N bits. Thus, O(logN) bits suffice to represent a bucket.

There are six rules that must be followed when representing a stream by buckets:

- The right end of a bucket is always a position with a 1.
- Every position with a 1 is in some bucket.
- No position is in more than one bucket.
- There are one or two buckets of any given size, up to some maximum size.
- All sizes must be a power of 2.
- Buckets cannot decrease in size as we move to the left (back in time).

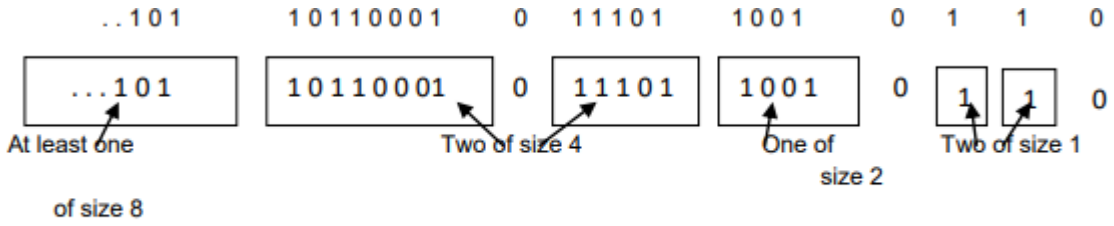


Figure 2: A bit-stream divided into buckets following the DGIM rules

1.4.1 Advantages

Stores only O(log2 N) bits

- O(log N)counts of log2N bits each Easy update as more bits enter

- Error in count no greater than the number of 1's in the unknown area.

4.6.3 Storage Requirements for the DGIM Algorithm

Each bucket can be represented by O(logN) bits. If the window has length N, then there are no more than N 1's, surely.

Suppose the largest bucket is of size 2j. Then j cannot exceed log2 N, or else there are more 1's in this bucket than there are 1's in the entire window.

Thus, there are at most two buckets of all sizes from log2 N down to 1, and no buckets of larger sizes.

4.6.4 Query Answering in the DGIM Algorithm

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 4.12: Suppose the stream is that of Fig. 4.2, 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.

4.6.5 Maintaining the DGIM Conditions

Suppose we have a window of length N properly represented by buckets that satisfy the DGIM conditions.

When a new bit comes in, we may need to modify the buckets, so they continue to represent the window and continue to satisfy the DGIM conditions.
First, whenever a new bit enters:

Check the leftmost (earliest) bucket. If its timestamp has now reached the current timestamp minus N, then this bucket no longer has any of its 1's in the window.

Therefore, drop it from the list of buckets. Now, we must consider whether the new bit is 0 or 1. If it is 0, then no further change to the buckets is needed. If the new bit is a 1, however, we may need to make several changes. First:

- 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, that is one too many.

• 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 (later in time) of the two buckets.

Suppose we start with the buckets of **Fig. 2 and a 1 enters**. First, the leftmost bucket evidently has not fallen out of the window, so we do not drop any buckets. We create a new bucket of size 1 with the current timestamp, say t.

There are now three buckets of size 1, so we combine the leftmost two. They are replaced with a single bucket of size 2. Its timestamp is t − 2, the timestamp of the bucket on the right (i.e., the rightmost bucket that actually appears in Fig. 3

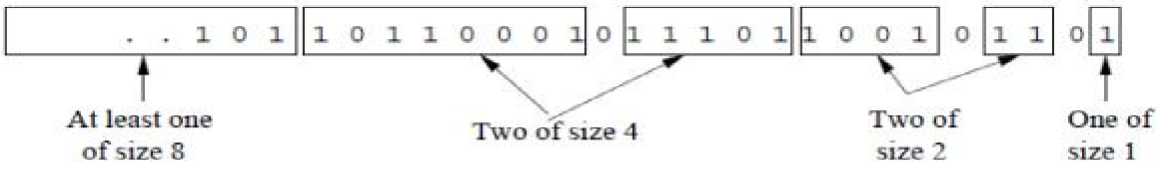


Figure 3: Modified buckets after a new 1 arrives in the stream

4.7 Decaying Windows

That a sliding window held a certain tail of the stream, either the most recent N elements for fixed N, or all the elements that arrived after some time in the past.

4.7.1 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.‖

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.

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.

4.7.2 Definition of the Decaying Window

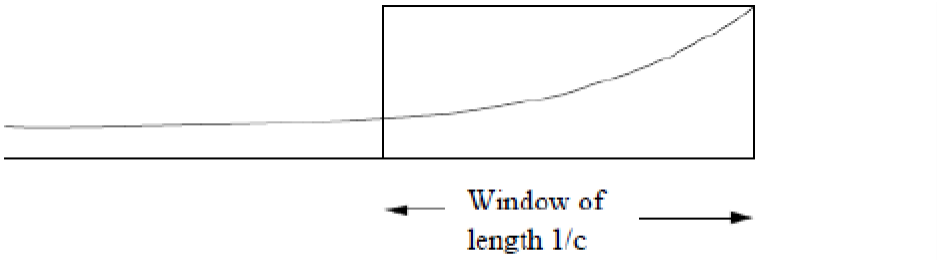
An alternative approach is to redefine the question so that we are not asking for a count of 1's in a window. Rather, let us compute a smooth aggregation of all the 1's ever seen in the stream, with decaying weights,

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.

The distinction is suggested by Fig. 4.4.



igure 4.4: 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.
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 at+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.

the weight on the current element is (1 − c)0 = 1, so adding at+1 is the correct way to include the new element's contribution.

4.7.3 Finding the Most Popular Elements

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.

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. It may not be obvious that the number of movies whose scores are maintained at any time is limited. However, note that the sum of all scores is 1/c.

4.8 Real Time Analytical Platform

Real time analytics makes use of all available data and resources when they are needed.
It consists of dynamic analysis and reporting based on the entered on to a system less than one minute before the actual time of use.

Real time denotes the ability to process as it arrives, rather than storing the data and retrieving it at some point in the future.

Real time analytics is thus delivering meaningful patterns in the data for something urgent.
Types of real time analytics

On Demand Real Time Analytics – It is reactive because it waits for users to request a query and then delivers the analytics. This is used when someone within a company needs to take a pulse on what is happening right this minute.

Continuous Real Time Analytics – It is more proactive and alerts users with continuous updates in real time. Example Monitoring stock market trends provide analytics to help users make a decision to buy or sell all in real time.

Real Time Analytics Applications

Financial Services – Analyze tickets, tweets, satellite integrity, weather trends, and any other type of data to inform trading algorithm in real-time.

Government – Identify social program fraud within seconds based on program history, citizen profile, and geographical data.

E-Commerce sites – Real time analytics will help to tap into user preferences as people are on the site or using product. By knowing what user likes at a run time can help the site to decide relevant content to be made available to that user.

This can result in better customer experience overall leading to increase in sales. Insurance Industry – Digital channel of customers interaction as well as conversations online have created new stream of real time event data.

Generic Design of an RTAP

Companies like Facebook and twitter generates petabytes of real time data. This data must be harnessed to provide real time analytics to make better business decisions. Today Billions of devices are already connected to the internet with more connecting everyday.

Real time analytics will leverage information from all these devices to apply analytics algorithms and generate automated actions within milliseconds of a trigger.

Real time analytics needed the following aspects of data flow, Input - An event happens (New sale, new customer, someone enters a high security zone etc.)

Process and Store Input – Capture the data of the event, and analyze the data without leveraging resources that are dedicated to operations.

Output – Consume this data without distributing operations

The following key capabilities must be provided by any analytical platform

- Delivering in Memory Transaction Speed
- Quickly Moving Unneeded data to disk for long term storage
- Distributing data and processing for speed
- Supporting continuous queries for real time events
- Embedding data into apps or apps into database
- Additional requirements

Many technologies support real time analytics, they are,

- Processing in memory
- In database analytics
- Data warehouse applications
- In memory analytics
- Massive parallel programming

Case Studies - Real Time Sentiment Analysis

Sentiment analysis also known as opinion mining refers to the use of natural language processing, text analysis and computational linguistics to identify and extract subjective information in source materials.

Sentiment analysis is widely applied to reviews and social media for a variety of applications ranging from marketing to customer service.

A basic task in sentiment analysis is classifying the polarity of a given text at the document, sentence, or feature/aspect level where the expressed opinion in a document, a sentence or an entity feature aspect is positive negative or neural.

Applications

News media website interested in getting edge over its competitors by featuring site content that is immediately relevant to its readers. They use social media analysis topics relevant to their readers by doing real time sentiment analysis on twitter data. Specifically to identify what topics are trending in real time on twitter.

Twitter has become a central site where people express their opinions and views on political parties and candidates.

Emerging events or news or often followed almost instantly by a burst in twitter volume which if analyzed in real time can help explore how these events affect public opinion While traditional content analytics takes days or weeks to complete, RSTA can look into entire content about election and delivering results instantly and continuously.

Ad agencies can track the crowd sentiment during commercial viewing on TV and decide which commercials are resulting in positive sentiment and which are not.

Analyzing sentiments of messages posted to social media or online forums can generate countless business values for the organizations, which aims to extract timely business intelligence about how their products or services are perceived by their customers.

As a result proactive marketing or product design strategies can be developed to efficiently increase the customer base.

Tools

Apache Strom is a distributed real time computation system for processing large volume of data. It is part of Hadoop. Strom is extremely fast with the ability to process over a million of records per second per node on a cluster of modestsize.

Apache Solr is another tool from Hadoop which provides a highly reliable scalable search engine facility at real time.

RADAR is a software solution for retailers built using a Natural language processing based sentiment analysis engine and utilizing Hadoop technologies including HDFS, YARN, Apache Storm, Apache Solr, Oozie and Zookeeper to help them maximize sales through databased continuous repricing.

Online retailers can track the following for number of products in their portfolio,

a. Social sentiment for each product
b. Competitive pricing / promotions being offered in social media and in theweb.

REAL TIME STOCK PREDICTION

Traditional stock market prediction algorithm check historical stock price and try to predict the future using different models.

But in real time scenario stock market trends continually change economic forces, new products, competition, world events, regulations and even tweets are all factors to affect stock prices. Thus real time analytics to predict stock prices is the need of the hour.

A general real time stock prediction and machine learning architecture comprises three basic components,

a. Incoming real time trading data must be captured and stored becoming historical data.
b. The system must be able to learn from historical trends in the data and recognize patterns and probabilities to inform decisions.
c. The system needs to do a real time comparison of new incoming trading data with the learned patterns and probabilities based in historical data. Then it predicts an outcome and determines a action to take.

For Example consider the following,

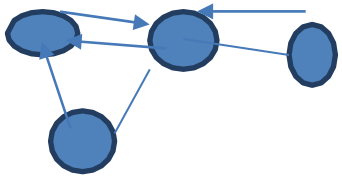
Live data from Yahoo Finance or any other finance news RSS feed is real and processed. The data is then stored in memory with a fast consistent resilient and linearly scalable system.

Using the live hot data from Apache Geode, a Spark MLib application creates and trains a model computing new data to historical patterns. The models could also be supported by other toolsets such as Apache MAD lib or R.

Results of the machine-learning model are pushed to other interested applications and also updated within Apache Geode for real time predication and decisioning.

As data ages and starts to become cool it is moved from Apache Geode to Apache HAWQ and eventually lands in Apache Hadoop. Apache HAWQ allows for SQL based analysis on petabyte scale data sets and allows data scientists to iterate on and improve models.

Another process is triggered to periodically retain and update the machine learning model based on the whole historical data set. This closes the loop and creates ongoing updates and improvements when historical patterns change or as new models emerge.



24