UNIT IV

- Stream memory

Stream data model(contd..)

3.Stream Queries:

Analyze and perform actions on real time data through use of continuous queries

2 types:

1.standing queries

2.Adhoc queries

1.Standing queries:

- A place within the processor where standing queries are stored. These queries are, in a sense, permanently executing, and produce outputs at appropriate times.
- Eg: 1.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

- 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.

2.Adhoc queries:The other form of query is **ad-hoc**, a question asked once about the **current state** of a stream or streams.

If we do not store all streams in their entirety, as normally we cannot, then we cannot expect to answer arbitrary queries about streams.

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.
- **Example**: 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.

4.Issues in Stream Processing

- 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.
- Here are **two generalizations** about stream algorithms

1. It is much more efficient to get an **approximate answer** to our problem than an exact solution.
2. 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.

4.2 sampling data in a stream

- Stream sampling is the process of collecting a representative sample of the elements of a data stream.
- The sample is usually much smalller than the entire stream,but used to retain many important characteristics of the stream
- From selected set of data streams we can pose queries and have answers.

4.2.1 Motivating example

- 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).
- 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.
- 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.
- In such situations take all their searches for the sample. 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.
- More generally, we can obtain a sample consisting of any rational fraction a/b of the users by hashing user names to b buckets, 0 through b − 1. Add the

  search query to the sample if the hash value is less than a.

4.2.3 The general sampling problem

- 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 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.

4.2.4 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, . . . ,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.
- 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.

4.4Filtering 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**

1. 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.
- In this case, we have room for eight billion bits, since one byte equals eight bits.

- Devise a hash function h from email addresses to eight billion buckets. Hash each member of S to a bit, and set that bit to 1. All other bits of the array remain 0.

2.Bloom Filter

- It is a space –efficient probabilistic data structure used to test whether an element is a member of a set.
- **False positive are possible, but false negatives are not** .
- In other words , a query returns either “possibly in set” or definitely not in set”.
- Elements can be added to the set,but not removed

- 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.

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 model to use is throwing darts at targets. Suppose we have x targets and y darts.

- Any dart is equally likely to hit any target. After throwing the darts, how many targets can we expect to be hit at least once?
- The analysis is similar to the analysis 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 $\left(1-\frac{1}{x}\right)^{x\left(\frac{y}{x}\right)}$.
- 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 $e^{-y/x}$.

Modern applications

1. Peer to peer (P2P) communication
2. Resource location
3. Routing
4. Measurement infrastructure

4.4 Counting Distinct Elements in a Stream

- we use variety of hashing and a randomized algorithm in order to consume a resonable amount of memory.
- The required scenario is "**Too much input too little space**- how to deal with big data.

1. The Count-Distinct Problem

- Suppose stream elements are chosen from some universal set.
- We would like to know 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.

- 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.
- This measure is appropriate for a site like Amazon, where the typical user logs in with their unique login name.

- 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.

- As long as the number of distinct elements is not too great, this structure can fit in main memory and there is little problem obtaining an exact answer to the question how many distinct elements appear in the stream.
- However, 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

This can be handled by the following ways:

1. more machines, each machine handling only one or several of the streams.
2. secondary memory and batch stream elements so whenever we brought a disk block to main memory there would be many tests and updates to be performed on the data in that block.

2. The Flajolet-Martin Algorithm

- It is possible to estimate the number of distinct elements by hashing the elements of the universal set to a bit-string that is sufficiently long.
- The length of the bit-string must be sufficient that there are more possible results of the hash function than there are elements of the universal set.
- For example, 64 bits is sufficient to hash URL's. The important property of a hash function is that when applied to the same element, it always produces the same result and lead to collision

- The particular unusual property we shall exploit is that the value ends in many 0's, although many other options exist.
- Apply a hash function h to a stream element a, the bit string h(a) will end in some number of 0's, possibly none.
- Call this number the tail length for a and h. Let R be the maximum tail length of any a seen so far in the stream.

  Then we shall use estimate 2R for the number of distinct elements seen in the stream.

Example

1. Pick a hash function h that maps each of the n elements to atleast $\log_2 n$ bits
2. For each stream element a,let r(a) be the number of trailing 0's in h(a).

   (a)called the tail length

   (b)Example:000101 has tail length 0;101000 has tail length 3.
3. Record R=the maximum r(a) seen for any a in the stream.
4. Estimate(based on this hash function)=2^R

This estimate makes intuitive sense. The probability that a given stream element a has $h(a)$ ending in at least r 0's is 2^{-r}. Suppose there are m distinct elements in the stream. Then the probability that none of them has tail length at least r is $\left(1-2^{-r}\right)^m$. This sort of expression should be familiar by now. We can rewrite it as $\left(\left(1-2^{-r}\right)^{2^r}\right)^{m2^{-r}}$. Assuming r is reasonably large, the inner expression is of the form $\left(1-\epsilon\right)^{1/\epsilon}$, which is approximately $1/e$. Thus, the probability of not finding a stream element with as many as r 0's at the end of its hash value is $e^{-m2^{-r}}$. We can conclude:

- 1. If m is much larger than 2r, then the probability that we shall find a tail of length at least r approaches 1.
- 2. If m is much less than 2r, then the probability of finding a tail length atleast r approaches 0.
- We conclude from these two points that the proposed estimate of m, which is 2R (recall R is the largest tail length for any stream element) is unlikely to be either much too high or much too low.

3. Combining Estimates

- There is a trap regarding the strategy for combining the estimates of m, the number of distinct elements, that we obtain by using many different hash functions.

Method 1:

1. we take the average of the values 2R that we get from each hash function.
2. value of r such that 2r is much larger than m.
3. probability p that we shall discover r to be the largest number of 0's at the end of the hash value for any of the m stream elements

4.Then the probability of finding r +1 to be the largest number of 0's instead is at least p/2.

- However, if we do increase by 1 the number of 0's at the end of a hash value, the value of 2R doubles.
- Consequently, the contribution from each possible large R to the expected value of 2R grows as R grows, and the expected value of 2R is actually infinite.

Method II:

1.Another way to combine estimates is to take the median of all estimates

- Unfortunately, the median suffers from another defect: it is always a power of 2.
- Thus, no matter how many hash functions we use, should the correct value of m be between two powers of 2, say 400, then it will be impossible to obtain a close

solution:

- We can combine the two methods.
- First, group the hash functions into small groups, and take their average.
- Then, take the median of the averages.
- However, taking the median of group averages will reduce the influence of this effect almost to nothing.

4.Groups should be of size at least a small multiple of log2m so that true value of m can be guaranteed.

4. Space Requirements

It is not necessary to store the elements seen.

- The only thing we need to keep in main memory is one integer per hash function; this integer records the largest tail length seen so far for that hash function and any stream element.
- If we are processing only one stream, we could use millions of hash functions, which is far more than we need to get a close estimate.
- Only if we are trying to process many streams at the same time would main memory constrain the number of hash functions we could associate with any one stream.

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

Time and space complexity

- If the stream contains n elements with m of them unique, this algorithm runs in $O(n)$
- It needs $O(log(m))$ memory.

Find the distinct element in the stream

- Example:
- S=1,3,2,1,2,3,4,3,1,2,3,1
- $h(x)=(6x+1) \ mod \ 5$ .consider |b|=5

Solution:

Steps:

1.calculate hash function

2.calculate binary bit

3. find Trailing Zeros

4.find the distinct element

1.Hash function

$h(x)=(6x+1) \ mod \ 5$

X=1

$h(x)=(6x+1) \ mod \ 5=(6(1)+1) \text{mod } 5$

$=7 \text{ mod } 5$

$=2$

h(1)=2

Similarly find for all terms

2.Binary bit calculation

- H(1)=2=00010

3.Trailing Zeros

- Count the Zeros h(1)=2=00010

  trailing zero is one

4.Distinct element

- Value of r=2(max no of zeros)
- $R=2^R$
- $2^2=4$

x	h(x)	Rem	Binary	r(a)
1	7	2	00010	1
3	19	4	00100	2
2	13	3	00011	0
1	7	2	00010	1
2	13	3	00011	0
3	19	4	00100	2
4	25	0	00000	0
3	19	4	00100	2
1	7	2	00010	1
2	13	3	00011	0
3	19	4	00100	2
1	7	2	00010	1

Reservoir sampling

- It is a family of randomized algorithm for randomly choosing k samples from a list of n items without replacement of k items ,where n is either a very large or unknown number.

4.5 Estimating Moments

- Computing “moments,” involves the distribution of frequencies of different elements in the stream.

4.5.1 Definition of moments

- 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 m_i be the number of occurrences of the i^{th} element for any i.
- Then the k^{th}-order moment (or just kth moment) of the stream is the sum over all i of $(m_i)k$.

example

- 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 of Section 4.4 to estimate the 0th moment of a stream

Solution:

- The **1st moment = the sum of the mi's**, which must be the length of the stream.
- Thus, first moments are especially easy to compute; just count the length of the stream seen so far

- The **second moment** is **the sum of the squares of the mi's.(surprise number)**
- It is sometimes called the **surprise number**, since it measures how uneven the distribution of elements in the stream is.
- To see the distinction, suppose we have a stream of **length 100**, in which **eleven** different elements appear.
- The most even distribution of these eleven elements would have **one** appearing **10 times** and the **other ten** appearing **9 times each..**

- In this case, the surprise number is 102 + 10 × 92 = 910. At the other extreme, one of the eleven elements could appear 90 times and the other ten appear 1 time each.
- Then, the surprise number would be 902 + 10 × 12 = 8110.
- Formula:f=sum of$(mi)^k$
- K=moment number
- Mi=number of times value i occurs

4.5.2 **The Alon-Matias-Szegedy Algorithm for Second Moments**

- 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.

  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

- 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 $5^2+4^2+3^2+3^2 = 59$. (second moment calculation)

  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.
- Likewise,nothing happens at positions 5 or 6. At position 7, we see c again, so we set X1.value = 2.
- At position 8 we find d, and so set X2.element = d and X2.value = 1.
- Positions 9 and 10 hold a and b, so they do not affect X1 or X2.
- Position 11 holds d so we set X2.value = 2, and position 12 holds c so we set X1.value = 3.
- At position 13, we find element a, and so set X3.element = a and X3.value = 1.
- Then, at position 14 we see another a and so set X3.value = 2.

- Position 15 with element b does not affect any of the variables, so we are done, with final values X1.value = 3 and X2.value = X3.value = 2.
- We can derive an estimate of the second moment from any variable X. **This estimate is n(2X.value − 1).**
- **1.Estimate:**

  X1 estimate= 15*(2*3-1)=75

  X2 estimate=15*(2*2-1)=45

  X3 estimate=15*(2*2-1)=45

- **2.calculate the average of estimates**

  avg=(x1+x2+x3)/3

  =(75+45+45)/3=55

Second moment calculation **directly** we get **59.**

By AMS algorithm we get **55**. Nearly accurate.

Pros and Cons

Pros:

- simple
- Needs store only k counters(memory efficient)
- Larger the value of k->accuracy increases

Cons:

- N value not known.(N –length of stream)

4.5.3 .Why the Alon-Matias-Szegedy Algorithm Works

- We can prove that the expected value of any variable constructed as like second moment of the stream from which it is constructed.
- 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 6^{th} 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

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

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

We can simplify the above by canceling factors $1/n$ and n, to get

$$E\big(n(2X.value - 1)\big) = \sum_{i=1}^{n} \big(2c(i) - 1\big)$$

- However, to make sense of the formula, we need to change the order of summation by grouping all those positions that have the same element.
- For instance, concentrate on some element a that appears ma times in the stream.
- The term for the **last position** in which a appears must be **2 × 1 – 1 = 1**.
- The term for the **next-to-last** position in which a appears is 2 × 2 − 1 = 3.
- The positions with a before that yield terms 5, 7, and so on, up to **2ma – 1**, which is the term for the first position in which a appears. That is, the formula for

the expected value of **2X.value – 1 can be written**:

$$E\left(n\left(2X.value-1\right)\right)=\sum_{a}1+3+5+\cdots+\left(2m_a-1\right)$$

Note that $1+3+5+\cdots+\left(2m_a-1\right)=\left(m_a\right)^2$. The proof is an easy induction on the number of terms in the sum. Thus, $E\left(n\left(2X.value-1\right)\right)=\sum_a\left(m_a\right)^2$, which is the definition of the second moment.

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. 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.
- Then, in Section 4.5.3 we saw why this formula works: the terms $2v - 1$, for $v = 1, 2, \ldots, m$ sum to m^2, where m is the number of times a appears in the stream.

Notice that $2v-1$ is the difference between v^2 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 $v^3-(v-1)^3=3v^2-3v+1$. Then $\sum_{v=1}^{m} 3v^2-3v+1=m^3$, so we can use as our estimate of the third moment the formula $n(3v^2-3v+1)$, where $v=X.value$ is the value associated with some variable X. More generally, we can estimate kth moments for any $k \geq 2$ by turning value $v=X.value$ into $n(v^k-(v-1)^k)$.

4.5.5 Dealing With Infinite Streams

- Technically, the estimate we used for second and higher moments assumes that n, the stream length, is a constant. In practice, **n grows with time**.
- That fact, by itself, doesn't cause problems, since we store only the values of variables and multiply some function of that value by n when it is time to estimate the moment.
- 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.
- A more serious problem is that we must be careful how **we select the positions for the variables**.

If we do this selection once and for all, then as the stream gets longer, we are biased in favor of early positions, and the estimate of the moment will be too large.

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

The proper technique is to maintain **as many variables as we can store at all times, and to throw some out as the stream** grows.

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.
- Inductively, suppose 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 $(n+1)^{s}$t 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.

- However, the probability of every other position also **c is s/(n + 1),** as we can prove by induction on n.
- By the inductive hypothesis,cbefore 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 of 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 2 cases, the probability of selecting each of the first n positions is

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

This expression simplifies to

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

and then to

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

which in turn simplifies to

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

Thus, we have shown by induction on the stream length n that all positions have equal probability s/n of being chosen as the position of a variable.

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 ≤ N. For this purpose we use DGIM algorithm**

4.6.1 The Cost of Exact Counts

- To begin, suppose we want to be able to count exactly the number of 1's in the last k bits for any k ≤ 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.**
- In proof, 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 6= 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. But the correct answers are surely different for these two bit-strings.
- Thus, we have proved that we must use at least N bits to answer queries about the last k bits for any k.

- we follow the current window by any $N - k$ bits.

4.6.2 The Datar-Gionis-Indyk-Motwani Algorithm

- This version of the algorithm uses O(log2 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 log2 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,5 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 $\log_2 N$** bits to represent the timestamp (modulo N) of its right end. To represent the number of 1's we only need **$\log_2 \log_2 N$ 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 $\log_2 N$, it requires $\log_2 \log_2 N$ 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).

. . 1 0 1 1 0 1 1 0 0 0 1 0 1 1 1 0 1 1 0 0 1 0 1 1 0

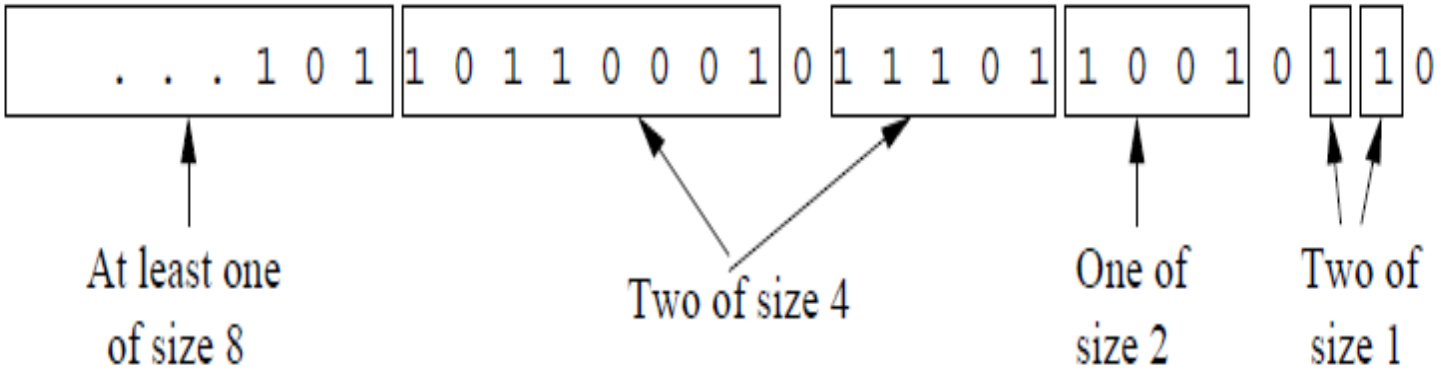


Figure 4.2: A bit-stream divided into buckets following the DGIM rules

4.6.3 Storage Requirements for the DGIM Algorithm

- We observed that 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 $\log_2 N$ down to 1, and no buckets of larger sizes.
- We conclude that there are O(logN) buckets. Since each bucket can be represented in O(logN) bits, the total space required for all the buckets representing a window of size N is $O(\log_2 N)$.

4.6.4 **Query Answering in the DGIM Algorithm**

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

- We know it is the last bucket to contribute to the answer, because the next bucket to its left has timestamp less than t − 9 and thus is completely out of the window.
- On the other hand, we know the buckets to its right are completely inside the range of the query because of the existence of a bucket to their left with timestamp t − 9 or greater.

- Our estimate of the number of 1’s in the last ten positions is thus 6.
- This number is the two buckets of size 1, the bucket of size 2, and half the bucket of size 4 that is partially within range. Of course the correct answer is 5.
- Suppose the above estimate of the answer to a query involves a bucket b
- of size 2j that is partially within the range of the query. Let us consider how far from the correct answer c our estimate could be.

- There are two cases: the estimate could be larger or smaller than c.
- **Case 1**: The estimate is less than c. In the worst case, all the 1's of b are actually within the range of the query, so the estimate misses half bucket b, or 2^{j-1} 1's.
- But in this case, c is at least 2j; in fact it is at least $2^{j+1} - 1$, since there is at least one bucket of each of the sizes 2^{j-1}, 2^{j-2}, . . . , 1.

- We conclude that our estimate is at least 50% of c.
- **Case 2**: The estimate is greater than c. In the worst case, only the rightmost bit of bucket b is within range, and there is only one bucket of each of the sizes smaller than b.
- Then $c = 1 + 2^{j-1} + 2^{j-2} + \cdots + 1 = 2^j$ and the estimate we give is $2^{j-1} + 2^{j-1} + 2^{j-2} + \cdots + 1 = 2^j + 2^{j-1} - 1$. We see that the estimate is no more than 50% greater than c.

4.6.5Maintaining 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. We fix this problem by combining the leftmost (earliest) two buckets of size 1.
- **Second**:**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

- Any new bit can be processed in O(logN) time

Example : Suppose we start with the buckets of Fig. 4.2 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 size2.
- Its timestamp is t − 2, the timestamp of the bucket on the right (i.e., the rightmost bucket that actually appears in Fig. 4.2)

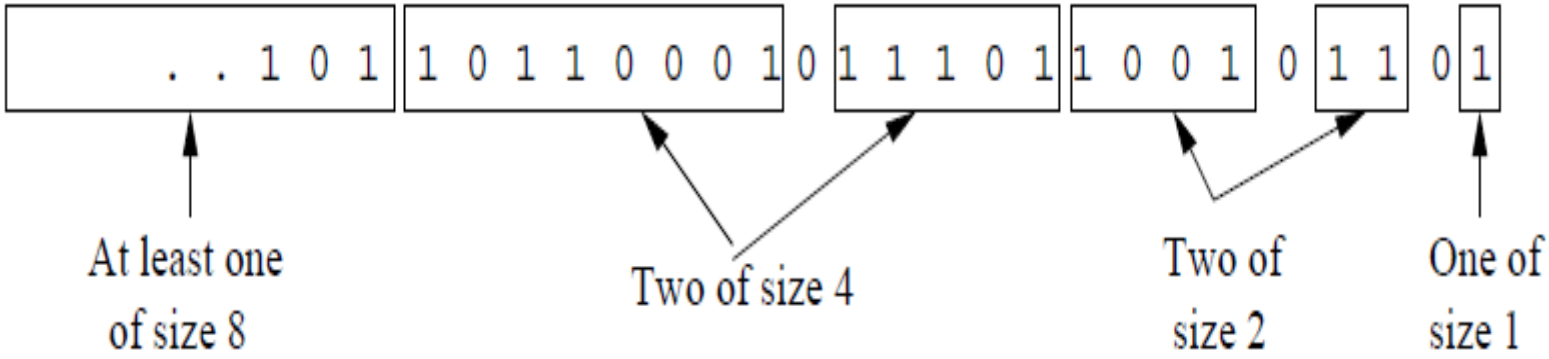


Figure 4.3: Modified buckets after a new 1 arrives in the stream

There are now two buckets of size 2, but that is allowed by the DGIM rules. Thus, the final sequence of buckets after the addition of the 1 is as shown in Fig. 4.3. □

4.6.6 **Reducing the Error**

- **Instead of allowing either one or two of each size bucket**, suppose we allow either r −1 or r of each of the exponentially growing sizes 1, 2, 4, . . ., for some integer r > 2.
- **In order to represent any possible number of 1's, we must relax this condition for the buckets of size 1** and buckets of the largest size present; there may be any number, from 1 to r, of buckets of these sizes.
- The rule for combining buckets is essentially the same as in Section 4.6.5

- If we get r + 1 buckets of size 2j, combine the leftmost two into a bucket of size 2j+1. That may, in turn, cause there to be r + 1 buckets of size 2j+1, and if so we continue combining buckets of larger sizes.
- The argument used in Section 4.6.4 can also be used here. However, because there are more buckets of smaller sizes, we can get a stronger bound on the error.
- We saw there that the largest relative error occurs when only **one 1 from the leftmost bucket b is within the query range, and we therefore overestimate the true count. Suppose bucket b is of size 2j. Then the true count** is at least

$1 + (r-1)(2^{j-1} + 2^{j-2} + \cdots + 1) = 1 + (r-1)(2^j - 1)$. The overestimate is $2^{j-1} - 1$. Thus, the fractional error is

$$\frac{2^{j-1} - 1}{1 + (r-1)(2^j - 1)}$$

No matter what j is, this fraction is upper bounded by $1/(r-1)$. Thus, by picking r sufficiently large, we can limit the error to any desired $\epsilon > 0$.

4.7 Decaying Windows

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, so the further back in the stream, the less weight is given. Formally, let a stream currently consist of the elements $a_1, a_2, \ldots, a_t$, where a_1 is the first element to arrive and a_t is the current element. Let c be a small constant, such as 10^{-6} or 10^{-9}. Define the *exponentially decaying window* for this stream to be the sum

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

The effect of this definition is to spread out the weights of the stream elements as far back in time as the stream goes. In contrast, a fixed window with the same sum of the weights, $1/c$, would put equal weight 1 on each of the most recent $1/c$ elements to arrive and weight 0 on all previous elements. The distinction is suggested by Fig. 4.4.

- 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.
- Further, the weight on the current element is $(1 - c)^0 = 1$, so adding
- a_{t+1} is the correct way to include the new element's contribution

4.7.3 Finding the Most Popular Elements

- Let us return to the problem of finding the most popular movies in a stream of ticket sales.6 We shall use an exponentially decaying window with a constant c, which you might think of as 10−9. That is, we approximate a sliding window holding the last one billion ticket sales.
- For each movie, we imagine a separate stream with a 1 each time a ticket for that movie appears in the stream, and a 0 each time a ticket for some other movie arrives. The decaying sum of the 1's

  measures the current popularity of the movie

- Therefore, we establish a threshold, say 1/2, so that if the popularity score for a movie goes below this number, its score is dropped from the counting.
- 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.
- There cannot be more than 2/c movies with score of 1/2 or more, or else the sum of the scores would exceed 1/c.
- **Thus, 2/c is a limit on the number of movies being counted at any time**. Of course in practice, the ticket sales would be concentrated on only a small number of movies at any time, so the number of actively counted movies would be much less than 2/c.

- **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
- fifa - 1 * (1-0.1) = 0.9
- ipl - 0.9 * (1-0.1) + 0 = 0.81 (adding 0 because current tag is different than fifa)
- fifa - 0.81 * (1-0.1) + 1 = 1.729 (adding 1 because current tag is fifa only)
- ipl - 1.729 * (1-0.1) + 0 = 1.5561
- ipl - 1.5561 * (1-0.1) + 0 = 1.4005
- ipl - 1.4005 * (1-0.1) + 0 = 1.2605
  fifa - 1.2605 * (1-0.1) + 1 = ***2.135***

ipl

- fifa - 0 * (1-0.1) = 0
- ipl - 0 * (1-0.1) + 1 = 1
- fifa - 1 * (1-0.1) + 0 = 0.9 (adding 0 because current tag is different than ipl)
- 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 then ***fifa***
  Even though both of them occurred same number of times in input there score is still different.

- **Advantages of Decaying Window Algorithm:**
- Sudden spikes or spam data is taken care.
- New element is given more weight by this mechanism, to achieve right trending output.

4.8. REAL TIME ANALYTICS PLATFORM (RTAP) APPLICATIONS

- **Real time Analytics Platform (RTAP) Applications**
- Real Time Analytics Platform (RTAP) analyzes **data, correlates and predicts outcomes** on a real time basis.
- The platform enables enterprises to track things in real time on a worldwide basis and helps in **timely decision making.** This platform provides us to build a range of powerful analytic applications.
- **The platform has two key functions**: they **manage** stored data and **execute** analytics program against it.

Why we need RTAP?

RTAP addresses the following issues in the traditional or existing RDBMS system

- **Server based licensing** is too expensive to use large DB servers
- **Slow processing speed**
- **Little support tools for data extraction** outside data warehouse
- **Copying large datasets into system is too slow**
- **Workload differences among jobs**
- **Data kept in files and folder, managing them are difficult**

- **Analytic platform combines tools** for creating analyses with an engine to execute them, a DBMS to keep and manage them for ongoing use and mechanism for acquiring and preparing data that are not already stored.

The **components of the platform** are depicted in the figure.

- The **data can be collected** from multiple data sources and feed through the data integration process.
- The **data can be captured and transformed and loaded** into analytic database management system (ADBMS).
- This **ADBMS has separate data store to mange data**. It also has the provision for creating functions and procedures to operate on the data.
- **Models can be created for analysis in the ADBMS itself**. Analytic applications can make use of the data in the ADBMS and apply the algorithms on it.

The application has the following facilities

- **Ad-hoc reporting**
- **Model building**
- **Statistical Analysis**
- **Predictive Analysis**
- **Data visualization**

ADBMS has the following desirable features for data analytics

- Use of proprietary hardware
- Hardware sharing model for processing and data through MPP (Massive Parallel Processing)
- Storage format (row and column manner) and smart data management
- SQL support and NoSQL support

- Programming extensibility and more cores, threads which yield more processing power
- Deployment model
- Infiniband -speed network

Applications

1.Social Media Analytics

- Social Media is the modern way of communication and networking.
- It is a growing and widely accepted way of interaction these days and connects billions of people on a real time basis.
- Fan page analysis – face book and twitter
- Tweeter analysis on followers, locations, time to tweet, interest to influence
- Measure customer service metrics on twitter
- Social media impacts on website performance

2.Business Analytics

- Business analytics focuses on developing new insights and understanding of business
- performance based on data and statistical methods.
- It gives critical information about supply and demand of business/product's viability in the marketplace.
- Goal tracking and returning customers trendlines
- Brand influence and reputation
- Combines online marketing and e-commerce data

3.Customer analytics

- Customer profiling
- Customer segmentation based on behaviour
- Customer retention by increasing lifetime value
- Applications: Google, IBM, SAS, WEKA analytic tools

4.Web Analytics

- It is the process of collecting, analyzing and reporting of web data for the purpose of understanding and optimizing web usage.
- On-site analytics (No. of users visited, no. of current users and actions, user locations etc…)
- Logfile analysis
- Click analytics
- Customer life cycle analytics
- Tracking web traffic
- ***Applications: clicky, shinystat, statcounter, site meters etc..***

4.7. CASE STUDIES - REAL TIME SENTIMENT ANALYSIS, STOCK MARKET PREDICTIONS

Introduction

- Historically, **stock market movements have been highly unpredictable.**
- With the advent of technological advances over the past two decades, nancial institutions and researchers have **developed computerized mathematical models to maximize their returns while minimizing their risk**.
- **One recent model by Johan Bollen involves analyzing the public's emotional states,represented by Twitter feeds, in order to predict the market**.
- The state of the art in **sentiment analysis** suggests there are **6** important mood states that enable the prediction of mood in the general public.

The prediction of mood uses the sentiment word lists obtained in various sources where general state of mood can be found using such **word list or emotion tokens.**

With the number of tweets posted on Twitter, it is believed that the general state of mood can be predicted with certain statistical significance.

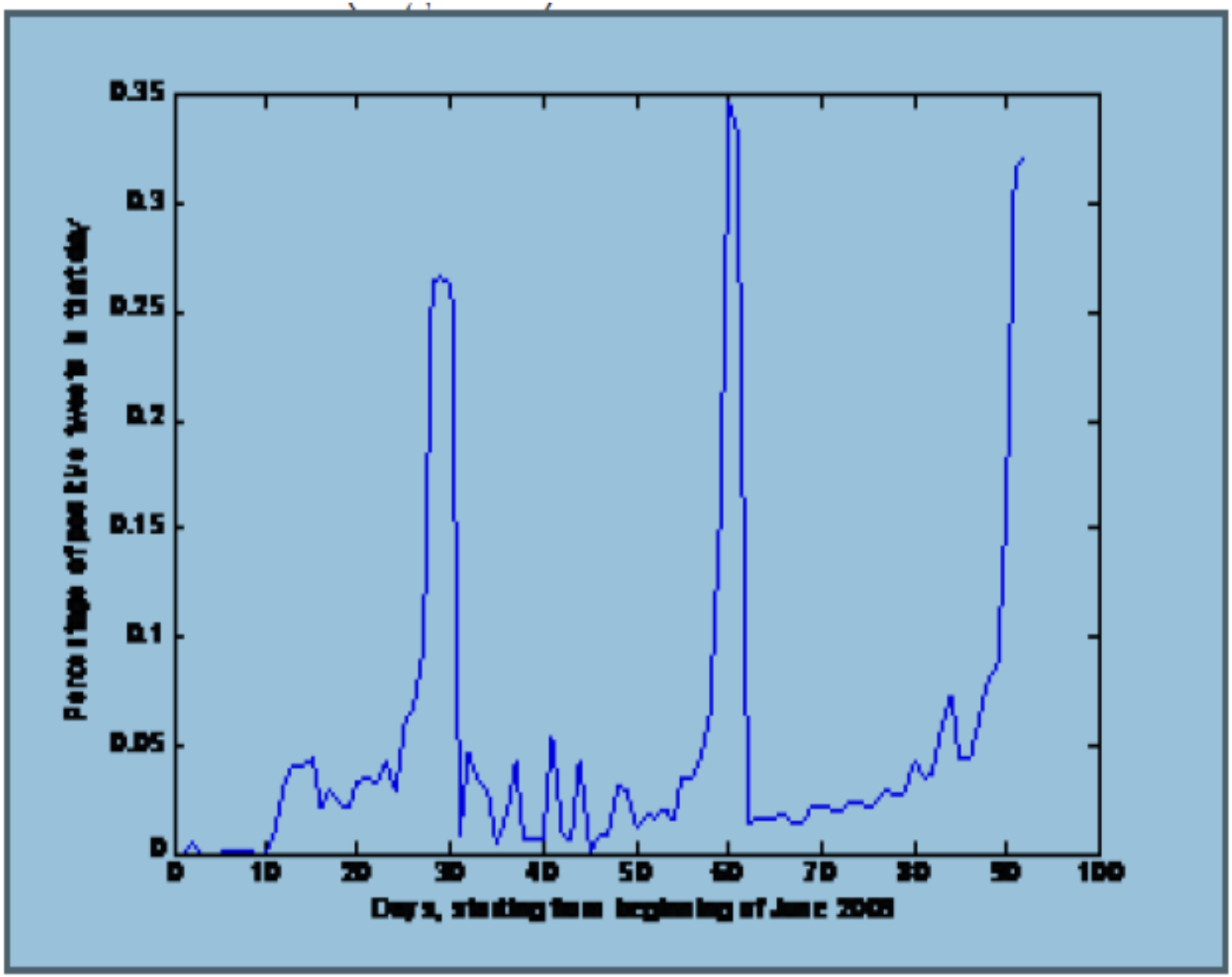
- According to Bollen's paper, Twitter sentiment is correlated with the market, preceding it by a few days.
- Specically, the Google Prole of Mood States' (GPOMS) `calm' state proved to be a reliable predictor of the market.
- Due to the proprietary nature of the GPOMS algorithm, we wish to see if a simpler method could provide similar results, while still being able to make accurate enough predictions to be protable.

- **Sentiment Analysis**
- We begin our sentiment analysis by applying **Alex Davies' word list** in order to see if a simple approach is suffcient enough to correlate to market movement.
- For this, we use a pregenerated word list of roughly ve thousand common words along with log probabilities of `happy‘ or `sad' associated with the respective words. The process works as follows.
- **First**, **each tweet is tokenized into a word list**. The parsing algorithm separates the tweets using whitespace and punctuation, while accounting for common syntax found in tweets, such as URLs and emoticons.
- **Next, we look up each token's log-probability in the word list**; as the word list is not comprehensive,
- we choose to ignore words that do not appear in the list. The log probabilities of each token was simply added to determine the probability of `happy' and `sad' for the entire tweet.
- **These were then averaged per day to obtain a daily sentiment value**

- As expected, this method resulted in **highly uncorrelated data** (with correlation coecients of almost zero).
- We tried to improve this by using a more comprehensive and accurate dictionary for positive and negative sentiments. Specially, we swapped our initial word list with a sentiment
- score list we generated using SentiWordNet, which consisted of over 400 thousand words. Since this list considers relationships between each word and includes multi-word expressions**, it provided better results**.
- We also tried representing the daily sentiment value in a different way - instead of averaging the probabilities of each tweet, we counted the frequency of `happy' tweets (such as using

- a threshold probability of above 0.5 for happy) and represented this as a percentage of all tweets for that day.
- While this did not improve the output's correlation with stock market data, it did provide us with more insight into our Twitter data.

For example, we see a spike in the percentage `happy‘ tweets toward the end of each month (Figure 1).


0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
0
10
20
30
40
50
60
70
80
90
100

- We did not news events which could have caused these spikes;
- however, upon investigating the source of Twitter data, we found that it had been pre-altered for a previous research project (i.e. there may be some bias in what we assumed to be raw Twitter data).
- Due to a lack of access to better Twitter data, we conclude that using the frequency of happy tweets is not a reliable indicator of sentiment for our application and revert to our averaging method.
- **Constructing the Model**
- In this section, we discuss the construction of our model, from choosing an appropriate algorithm to finding a suitable set of features, and provide justication for these decisions.

The Algorithm

We chose to model the data using a linear regression. This decision was motivated by several factors:

Speed - A fast, ecient algorithm was one of our original specications. This is a must when working with massive amounts of data in real time, as is the case in the stock market.

Regression - We sought to be able to make investment decisions not only on direction of market movement, but also to quantify this movement. A simple classier was insucient for this; we required a regressor.

- **Accurate** - Naturally, we needed an algorithm that would model the data as accurately as possible.
- Since our data is, by its nature, very noisy, we chose a simple model to avoid high variance

- **Testing the Model**
- We have built a model for predicting changes in the stock market price from day to day.
- We have identify the accuracy-maximizing set of features and trained our model on these features.
- Now we must put it to the test using real-world data to determine if it is protable.
- We develop 2 different investment strategies based on predictions from our model, apply them over some time period, report on the results, and compare them to 2 benchmark investment strategies.

Our Investment Strategies

- **Classication** - The simpler of our 2 strategies considers only the predicted direction of market movement. That is, we look only at the sign of the anticipated change in price.

  If it is positive, we buy as many shares as possible with our current funds. Otherwise, we buy no shares, and simply sit on the money until the following day when we reevaluate the situation.

- **Regression** - With this more complicated strategy, we seek to exploit knowledge of how much the market will change, rather than simply the direction it will shift.

  This allows us to base how much we invest on how certain we are of our prediction. There are countless such strategies that one could propose, we chose the following based on observations of good performance:

$$invest = \begin{cases} 100\% & \text{if } .05\% < (predicted\ \%\ change) \\ 25\% & \text{if } -.1\% \leqslant (predicted\ \%\ change) \leqslant .05\% \\ 0\% & \text{if } (predicted\ \%\ change) < -.1\% \end{cases}$$

- Here, invest is the percent of our funds we use to buy stock and predicted % change q is computed by dividing the predicted change in the market tomorrow by the price today.

The Benchmark Investment Strategies

- **Default** - This strategy uses no information about the market, and will simply buy as many shares as possible each day.
- **Maximal** - This strategy assumes perfect knowledge about future stock prices.

- We will invest all available funds when we know the market will go up the following day, and invest no funds when we know the market will go down.
- This strategy is, of course, impossible to execute in reality.

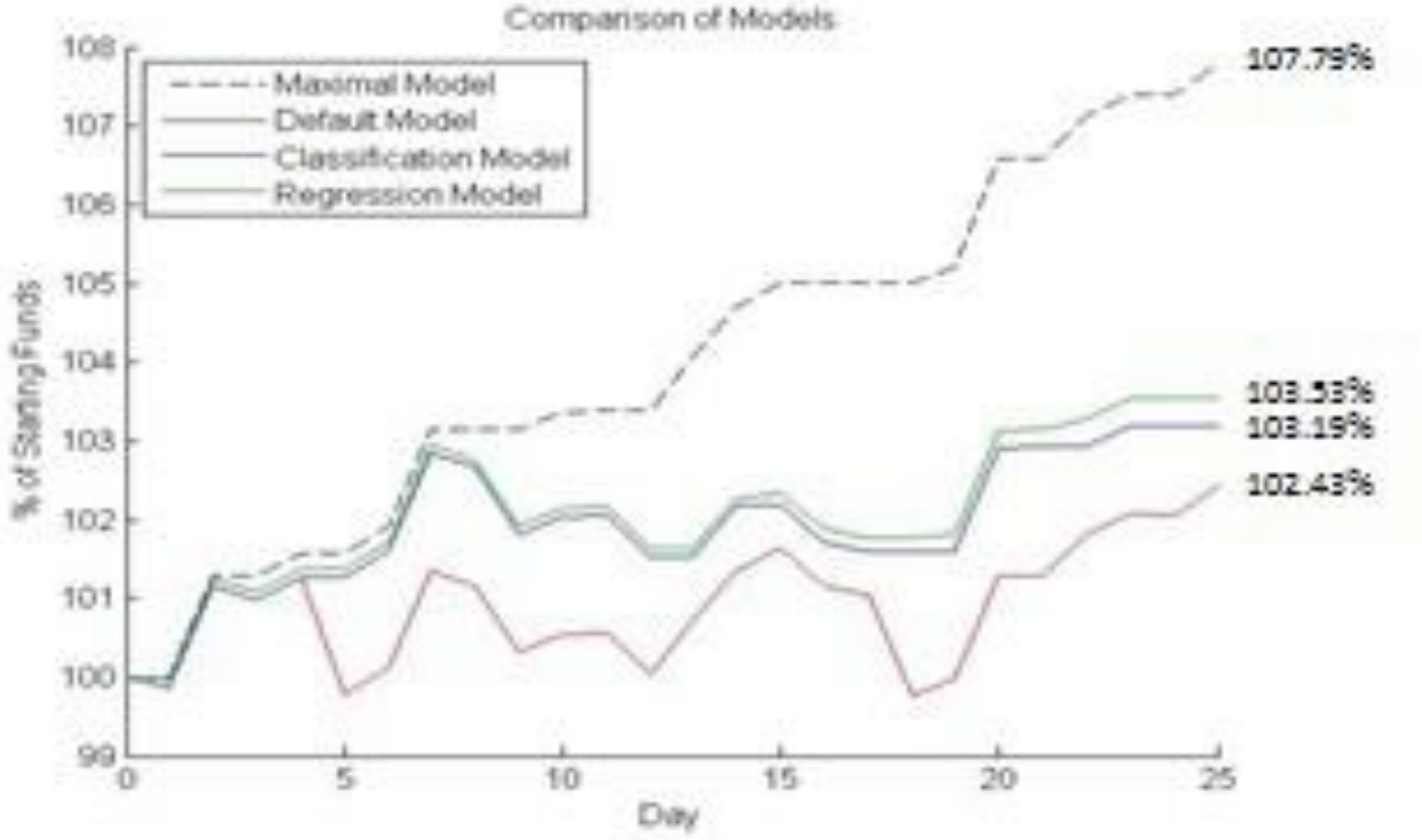
Simulation

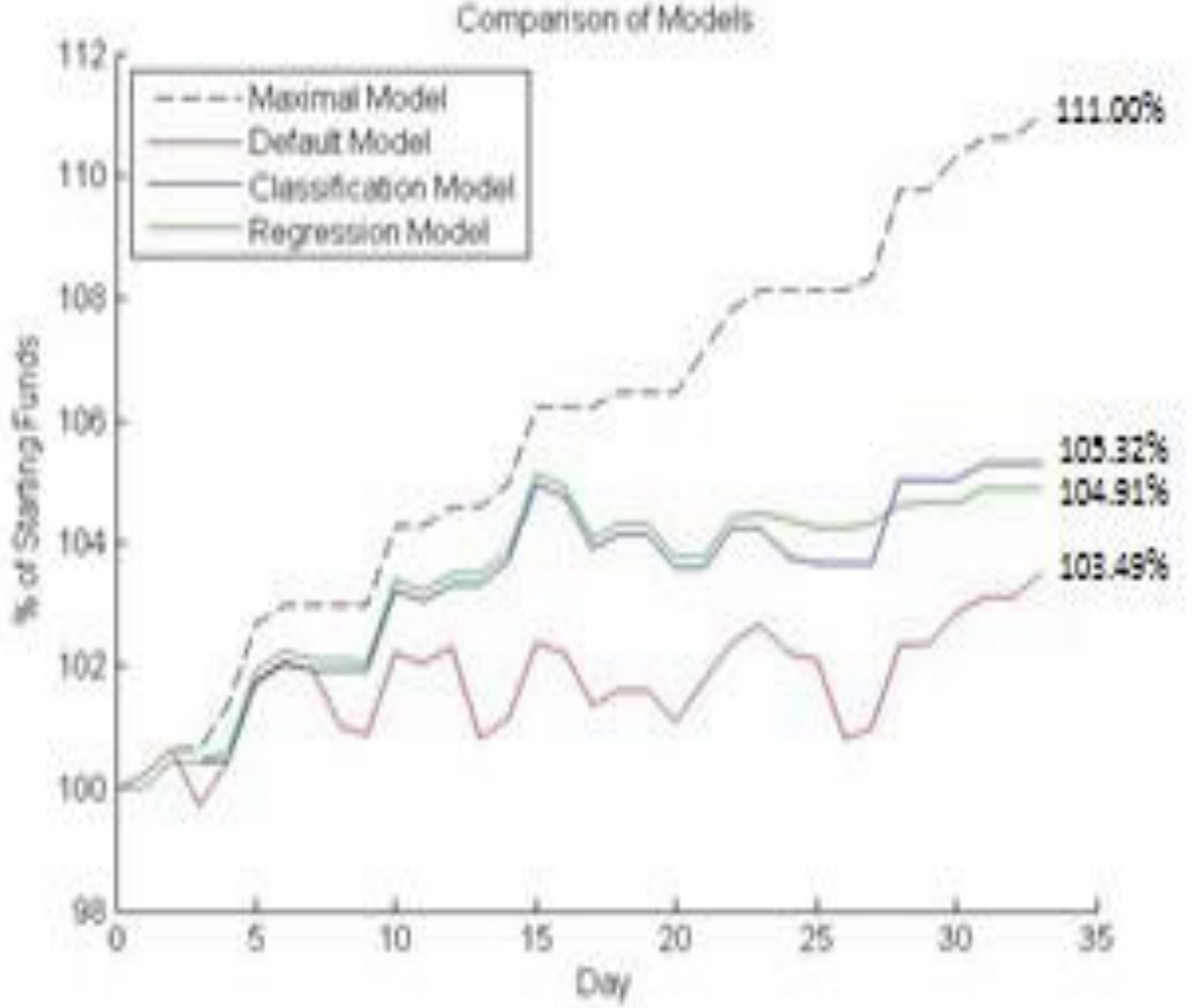
- We start with exactly enough money to buy 50 shares of stock on the first day. Note that since we output results as percentages of starting money, they do not depend on this value, and as such it is chosen arbitrarily.

- At the start of each day, we make a prediction and invest according to some strategy. At the end of the day, we sell all shares at closing price and put the money in this bank.
- This is done so that any gains or losses can future gains or losses by virtue of being able to purchase more or less stock at every time step.

Results

- We ran the simulation for each investment strategy, as described above, on 2 different time intervals. The results are shown below:


Comparison of Models
Maximal Model
Default Model
Classification Model
Regression Model
% of Starting Funds
Day
108
107
106
105
104
103
102
101
100
99
0
5
10
15
20
25
107.79%
103.53%
103.19%
102.43%


Comparison of Models
Maximal Model
Default Model
Classification Model
Regression Model
111.00%
105.32%
104.91%
103.49%
% of Starting Funds
Day
112
110
108
106
104
102
100
98
0
5
10
15
20
25
30
35

- In the Diagram on the left, we trained on about 3 4 of the data (72 days) and simulated on about 14 of the data (25 days). In the Diagram on the right, we trained on about 2 3 of the data (64 days) and simulated on about 1 3 of the data (33 days).
- We immediately see that both of our strategies fare better than the default strategy in both simulations.
- Note, however, that the regression strategy is more protable in the first simulation while the classification strategy is more protable in the second simulation.

- We observe that on the simulation in which the model was given less training data (figure on the right), on day 27, our regression strategy opted to invest only 25% of funds that day because it perceived gains as being uncertain.
- This did not happen on the corresponding day in the first simulation (with more training data).

- Indeed, with less data to train on, imprecision in our model resulted in a poor investment decision
- when using the more complex regression strategy.
- In general, the classification strategy tends to be more consistent, while the regression strategy, though theoretically more protable, is also more sensitive to noise in the model.

Example

Sentiment analysis is an automated process capable of understanding the feelings or opinions that underlie a text. It is one of the most interesting subfields of NLP, a branch of Artificial Intelligence (AI) that focuses on how machines process human language.

Sentiment analysis studies the subjective information in an expression, that is, the opinions, appraisals, emotions, or attitudes towards a topic, person or entity. Expressions can be classified as **positive**, **negative**, or **neutral**. For example:

"I really like the new design of your website!" → Positive

"I'm not sure if I like the new design" → Neutral

"The new design is awful!" → Negative

4.9Using Graph Analytics for Big Data

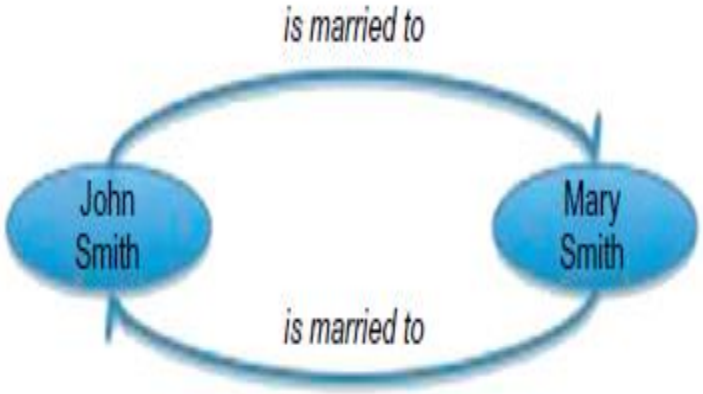
- Graph analytics are analytic tools used to determine strength and direction of relationships between objects in a graph.
- The focus of graph analytics is on pair wise relationship between two objects at a time and structural characteristics of the graph as a whole.
- Eg: In a graph representing relationship between individuals graph analytics can help answer questions like the following:
- How many other individuals does the average individual “friend” with?
- What is the maximum number of “friends” any one individual has?

- How interconnected are groups?

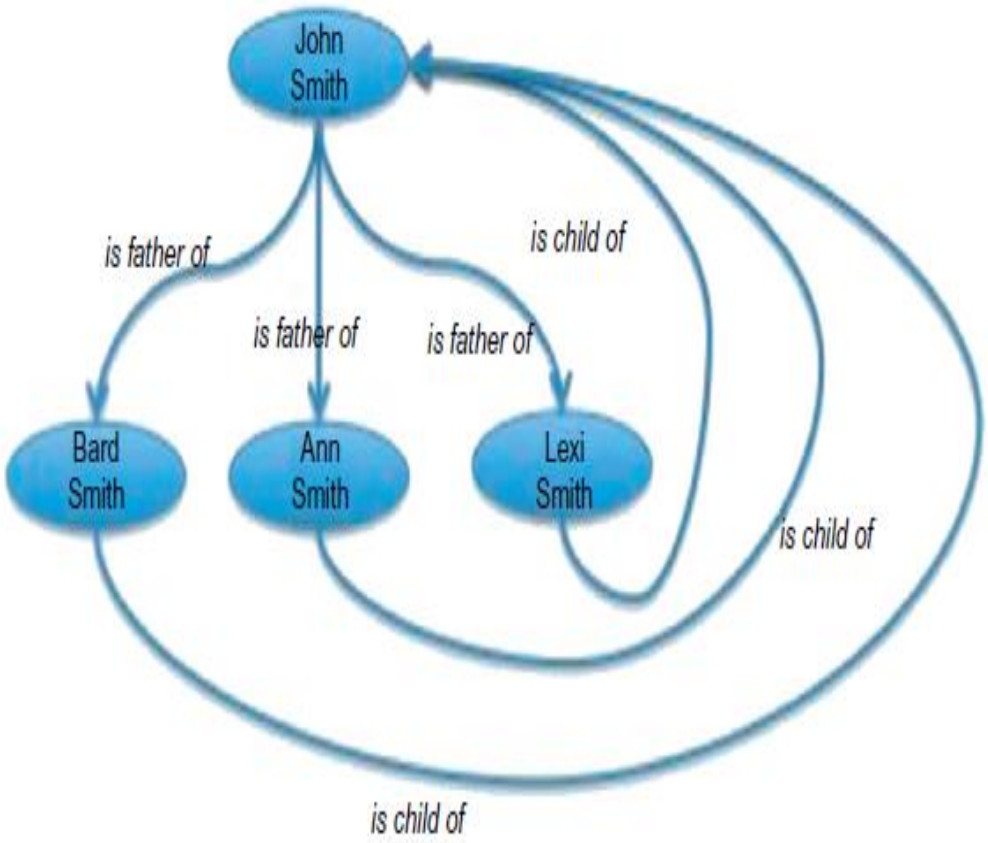
- Are there isolated groups of individuals who are connected to each other but not to individuals not in their group?

4.9.1. THE SIMPLICITY OF THE GRAPH MODEL

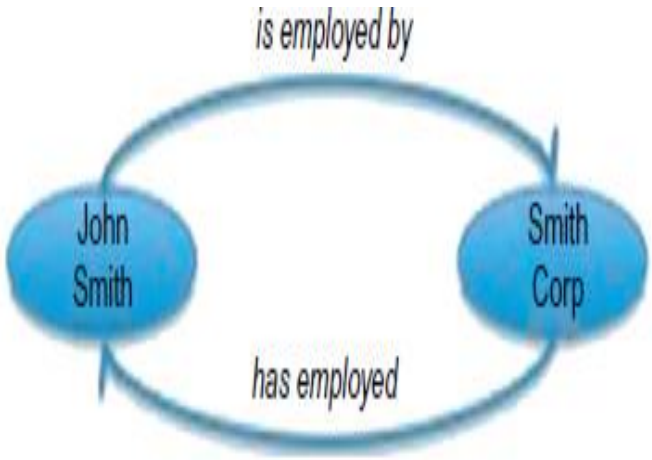
- **Graph analytics** is based on a **model of representing individual entities and numerous kinds of relationships that connect those entities.**
- More precisely, it employs the graph abstraction for representing connectivity, consisting of a **collection of vertices** (which are also referred to as nodes or points) that represent the modeled entities, **connected by edges** (which are also referred to as links, connections, or relationships) that capture the way that two entities are related.



John Smith is married to Mary Smith



John Smith has three children:
Bard, Ann, and Lexi Smith



John Smith works for Smith Corp

Figure 10.1 Examples of relationships represented as labeled directed graphs.

- **Vertices** can be labeled to indicate the **types of entities** that are related.
- **Edges** can be labeled with the **nature of the relationship**.
- Edges can be directed to indicate the “flow” of the relationship.
- **Weights can be added to the relationships** represented by the edges.
- Additional properties can be attributed to both edges and vertices.
- **Multiple edges can reflect multiple relationships** between pairs of vertices.

4.9.2 REPRESENTATION AS TRIPLES

- A directed graph that can be represented using a **triple format** consisting of a **subject** (the source point of the relationship), **an object** (the target),and a **predicate** (that models the type of the relationship). The same examples in Figure 10.1 could be modeled as a set of triples, as shown in Table 10.1.

Table 10.1 Triples Derived from the Example in Figure 10.1

Subject	Predicate	Object
John Smith	Is married to	Mary Smith
Mary Smith	Is married to	John Smith
John Smith	Is father of	Brad Smith
John Smith	Is father of	Ann Smith
John Smith	Is father of	Lexi Smith
Brad Smith	Is child of	John Smith
Ann Smith	Is child of	John Smith
Lexi Smith	Is child of	John Smith
John Smith	Is employed by	Smith Corp
Smith Corp	Has employed	John Smith

- A collection of these triples is called a **semantic database**, and this kind of database can capture additional properties of each triple relationship

  as attributes of the triple.

  Almost any type of entity and relationship can be represented in a graph model, which **means two key things**: **the process of adding new entities** and **relationships is not impeded** when new datasets are to be included, with new types of entities and connections to be incorporated,

  the model is particularly suited to types of discovery analytics that seek out new patterns embedded within the graph that are of critical business interest.

4.9.4 GRAPHS AND NETWORK ORGANIZATION

One of the benefits of the graph model is the ability to detect patterns or organization that are inherent within the represented network, such as:

- **Embedded micronetworks**: Looking for small collections of entitie that form embedded "microcommunities." Some examples include

  determining the originating sources for a hot new purchasing trend

identifying a terrorist cell based on patterns of communication across a broad swath of call detail records, or sets of individuals within a particular tiny geographic region with similar political views.

- **Communication models**: **Modeling communication across a community triggered by a specific event**, such as monitoring the “buzz” across social media channel associated with the rumored release of a new product, evaluating best methods for communicating news releases, or correlation between travel delays and increased mobile telephony activity.
- **Collaborative communities**: **Isolating groups of individuals that share similar interests,** such as groups of health care professionals working in the same area of specialty, purchasers with similar product tastes, or individuals with a precise set of employment skills

- **Influence modeling**: **Looking for entities holding influential positions within a network for intermittent periods of time**, such as computer nodes that have been hijacked and put to work as proxies for distributed denial of service attacks or for emerging cybersecurity threats, or individuals that are recognized as authorities within a particular area.

- **Distance modeling**: **Analyzing the distances between sets of entities, such as looking for strong correlations between occurrences of sets** of statistically improbable phrases among large sets of search engines queries, or the amount of effort necessary to propagate a message among a set of different communities.

Each of these example applications is a discovery analysis that looks for patterns that are not known in advance. As a result, these are less suited to pattern searches form relational database systems, such as a data warehouse or data mart, and are **better suited to a more dynamic representation like the graph model.**

10.5 CHOOSING GRAPH ANALYTICS

Deciding the appropriateness of an analytics application to a graph analytics solution instead of the other big data alternatives can be based on these characteristics and factors of business problems:

- **Connectivity**: The solution to the business problem requires the analysis of relationships and connectivity between a variety of different types of entities.
- **Undirected discovery:** Solving the business problem involves iterative undirected analysis to seek out as-of-yet unidentified patterns.

- **Absence of structure**: Multiple datasets to be subjected to the analysis are provided without any inherent imposed structure.
- **Flexible semantics**: The business problem exhibits dependence on contextual semantics that can be attributed to the connections and corresponding relationships.
- **Extensibility**: Because additional data can add to the knowledge embedded within the graph, there is a need for the ability to quickly add in new data sources or streaming data as needed for further interactive analysis.

- **Knowledge is embedded in the network**: Solving the business problem involves the ability to exploit critical features of the embedded relationships that can be inferred from the provided data.
- **Ad hoc nature of the analysis**: There is a need to run ad hoc queries to follow lines of reasoning.
- **Predictable interactive performance**: The ad hoc nature of the analysis creates a need for high performance because discovery in big data is a collaborative man/machine undertaking, and predictability is critical when the results are used for operational decision making.

10.6 GRAPH ANALYTICS USE CASES

Having reviewed the business problems that are suited to using a solution based on a graph analytics model, we can examine some use cases for graph analytics.

In these scenarios, we look at the business problem, why the traditional approach may not be optimal, and the discuss how the graph analytics approach is best suited to solving the business challenges.

Some example business applications that are aligned with these criteria include: 1.**Health care quality analytics**, in which patient health encounter histories, diagnoses, procedures, treatment approaches, and results of clinical trials.

2.Cyber security

10.7 GRAPH ANALYTICS ALGORITHMS AND SOLUTION APPROACHES

The graph model is inherently **suited to enable a broad range of analyses** that are generally unavailable to users of a standard data warehouse framework.

As suggested by these examples, instead of just providing reports or enabling online analytical processing (OLAP) systems, **graph analytics applications employ algorithms that traverse or analyze graphs to detect and potentially identify interesting patterns that are sentinels for business opportunities** for increasing revenue, identifying security risks, detecting fraud, waste, or abuse, financial trading signals, or even looking for optimal individualized health care treatments.

Some of the types of analytics algorithmic approaches include:

- **Community and network analysis**, in which the graph structures are traversed in search of groups of entities connected in particularly "close" ways.

  One example is a collection of entities that are completely connected (i.e., each member of the set is connected to all other members of the set)
- **Path analysis**, which analyze the shapes and distances of the different paths that connect entities within the graph.
- **Clustering,** which examines the properties of the vertices and edges to identify characteristics of entities that can be used to group them together.
- **Pattern detection and pattern analysis**, or methods for identifying anomalous or unexpected patterns requiring further investigation.
- **Probabilistic graphical models** such as Bayesian networks or Markov networks for various application such as medical diagnosis, protein structure prediction, speech recognition, or assessment of default risk for credit applications.

- **Graph metrics** that are applied to measurements associated with the network itself, including the **degree of the vertices** (i.e., the number of edges in and out of the vertex), or **centrality and distance** (including the degree to which particular vertices are "centrally located" in the graph, or how close vertices are to each other based on the length of the paths between them).

  These graph analytic algorithms can yield interesting patterns that might go undetected in a data warehouse model, and these patterns themselves can become the templates or models for new searches.

- In other words, the graph analytics approach can satisfy both the discovery and the use of patterns typically used for analysis and reporting.

10.8 TECHNICAL COMPLEXITY OF ANALYZING GRAPHS

There are some characteristics of graphs that inhibit the ability of typical computing platforms to provide the rapid responses or satisfy the need for scalability, especially as data volumes continue to explode.

Some of the factors addressed are:

Unpredictability of graph memory accesses: The types of discovery analyses in graph analytics often require the **simultaneous traversal of multiple paths** within a network to find the interesting patterns for further review or optimal solutions to a problem. The graph model is represented using data structures that incorporate the links between the represented entities, and this differs substantially from a traditional database model

. In a parallel environment, **many traversals can be triggered at the same time**, but each graph traversal is inherently dependent on the ability to follow the links from the subject to the target in the representation.

Unlike queries to structured databases, **the memory access patterns are not predictable,** limiting the ability to reduce data access latency by efficiently streaming prefetched data through the different levels of the memory hierarchy,

- **Graph growth models**:The larger the graph becomes, the more it exhibits what is called "**preferential connectivity"**—newly introduced entities are more likely to connect to already existing ones, and existing nodes with a high degree of connectivity are more likely to continue to be "popular" in that they will continue to attract new connections.

  This means that graphs continue to grow, but apparently that growth does not scale equally across the data structure

Dynamic interactions with graphs: As with any big data application, graphs to be analyzed are populated from a wide variety of data sources of **large or massive data volumes**, streamed at varying rates.

But while the **graph must constantly absorb many rapidly changing data streams and process the incorporation of connections and relationships** into the persistent representation of the graph, the environment must also satisfy the need to rapidly respond to many simultaneous discovery analyses.

Therefore, a high-performance graph analytics solution must accommodate the dynamic nature without allowing for any substantial degradation in analytics performance

Complexity of graph partitioning: Graphs tend to cluster among centers of high connectivity, as **many networks** naturally have **"hubs"** consisting of a small group of nodes with many connections.

The benefit of having hubs is that in general it shortens the distances among collections of nodes in the graph.

The hubs often showcase entities that may be of particular interest because of their perceived centrality.

But while one of **the benefits of big data platforms is the expectation of distribution of data and computation**, **the existence of hubs make it difficult to partition** the graph among different processing units because of the multiplicity of connections.

- Arbitrarily distributing the data structures in ways that span or cross multiple partitions will lead to increased cross-partition network traffic,which effectively eliminates the perceived performance benefit of data distribution.

10.9 FEATURES OF A GRAPH ANALYTICS PLATFORM

The ease of development features of the platform is enabled through the adoption of industry standards as well as supporting general requirements for big data applications, such as:

- **Seamless data intake:** absorb and fuse data from a variety of different sources.
- **Data integration**: A semantics-based approach is necessary for graph analytics to integrate different sets of data that do not have predetermined structure.

- Similar to other NoSQL approaches the schema-less approach of the semantic model must provide the flexibility not provided by relational database models.
- **Inferencing**: The application platform should provide methods for inferencing and deduction of new information and insights derived from the embedded relationships and connectivity.

- **Standards-based representation**: Any graph analytics platform must employ the resource description framework standard (the RDF, see http://www.w3.org/RDF/) to use triples for representing the graph.

  Representing the graph using RDF allows for the use of the SPARQL query language standard (see http://www.w3.org/TR/rdfsparql-query/) for executing queries against a triple-based data management environment.

Interoperability is critical, and here are some considerations for the operational aspects of the platform:

- **Workflow integration**: Providing a graph analytics platform that is segregated from the existing reporting and analytics environments will have limited value when there are gaps in incorporating results from across the different environments. Make sure that the graph analytics platform is aligned with the analysis and decision-making workflows for the associated business processes.
- **Visualization**: Presenting discoveries using visualization tools is critical to highlight their value. Look for platforms that have tight integration with visualization services.
- "**Complementariness**": A graph analytics platform augments an organization's analytics capability and is not intended to be a replacement. Any graph analytics capabilities must complement existing data warehouses, data marts, OLAP engines, and Hadoop analytic environments.

Finally, although all of the following are reasonable expectations for any big data platform, these criteria are particularly important for graph analytics due to the nature of the representation and the types of discovery analyses performed:

- **High-speed I/O**: Meeting this expectation for real-time integration requires a scalable infrastructure, particularly with respect to highspeed I/O channels, which will speed the intake of multiple data streams and thereby support graphs that are rapidly changing as new data is absorbed

High-bandwidth network: As a significant burden of data access may cross node barriers, employing a high-speed/high-bandwidth network interconnect will also help reduce data latency delays, especially for chasing pointers across the graph.

- **Multithreading**: Fine-grained multithreading allows the simultaneous exploration of different paths and efficiently creating, managing, and allocating threads to available nodes on a massively parallel processing architecture can alleviate the challenge of predictability.

Large memory: **A very large memory shared across multiple processors reduces the need to partition** the graph across independent environments.

This helps to reduce the performance impacts associated with graph partitioning.

Large memories are useful in keeping a large part, if not all of the graph resident in-memory.

By migrating allocated tasks to where the data is resident in memory reduces the impact of data access latency delays