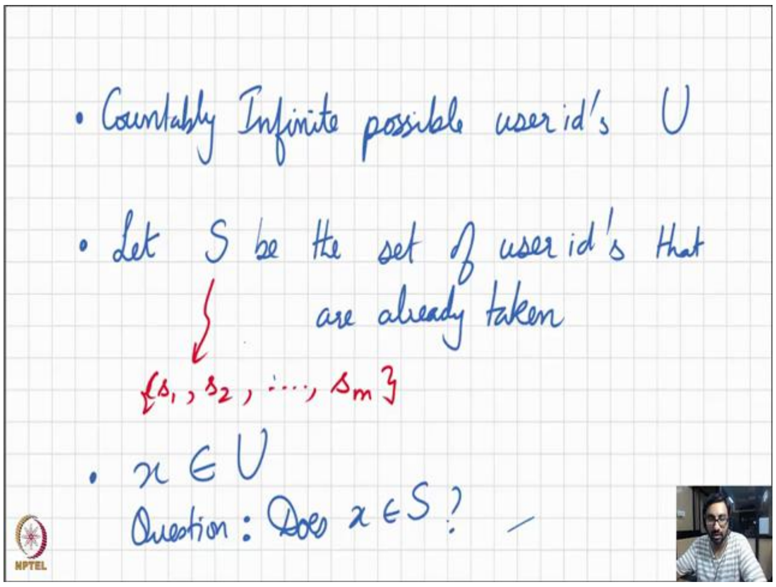
Algorithms for Big Data

Prof. John Ebenezer Augustine
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Lecture – 26
Bloom Filter

(Refer Slide Time: 00:33)



- Countably Infinite possible user id's U
- Let S be the set of user id's that are already taken

$\{s_1, s_2, \dots, s_m\}$

- $x \in U$

Question: Does $x \in S$?

Let us motivate this last segment with the following example, consider the case where there is a countably infinite number of user id’s that possible that people can use to access a certain web service. And as a business starts to pick up there will be a lot more lot of users joining the web service and this will mean that out of the possible set a size able number of user ids’ will be taken up.

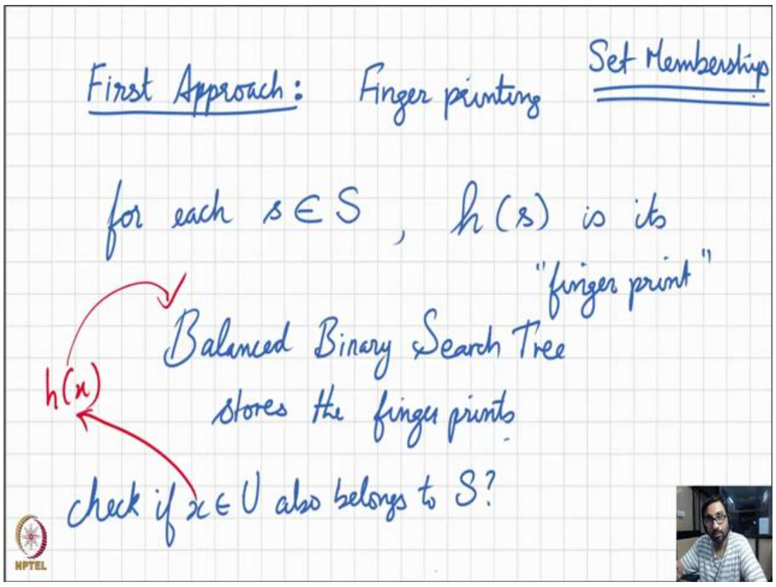
So, let us suppose that S is the set of user id’s that are already taken and we denote them by s 1, s 2 and so on up to s m. Now, given some element from the possible set of user id’s, and let us call this set U for the universe of user ids. Let us say we have some x and that belongs to the universe of id’s the question is does the user id x belong to the set of user id’s that are already taken. This is an important question, because as new users come into the system they going to try out some user id’s; and if the user id is taken then they will have to modify it and we have all done this before times I am sure.

The question is how does the system retrieve user id’s retrieve the fact that a particular user id is being used so quickly. I mean remember this these queries have to go into database which is probably distributed then in found in some particular server somewhere in their data center and you know the user id that you are trying to get might you might have to be searched against a lot of user id’s. And so, you may wonder how this works.

And we are going to look at some ways in which this can be implemented, but here something that I would like you to keep in mind or try to remember there are times when you might have tried some user id’s. And then this you would try some slightly complicated user id’s, add a few numbers and so on and so forth, and I am pretty sure you might have been surprised that even these complicated user id’s variations were rejected until you found one. And that is potentially because the system they are using behind the scenes is one that has a certain fraction of false positives, even though some user id’s are not actually taken the system may say that that user id is taken.

But one important aspect is that the system will never allow you to use a user id that has already been used. So, in other words it does not allow a false negative; it only allows a false positive, so that something to keep in mind and this might that might show up as we continue in this discussion.

(Refer Slide Time: 03:34)



First Approach: Finger printing

Set Membership

for each $s \in S$, $h(s)$ is its "finger print"

Balanced Binary Search Tree stores the finger prints

check if $x \in U$ also belongs to S?

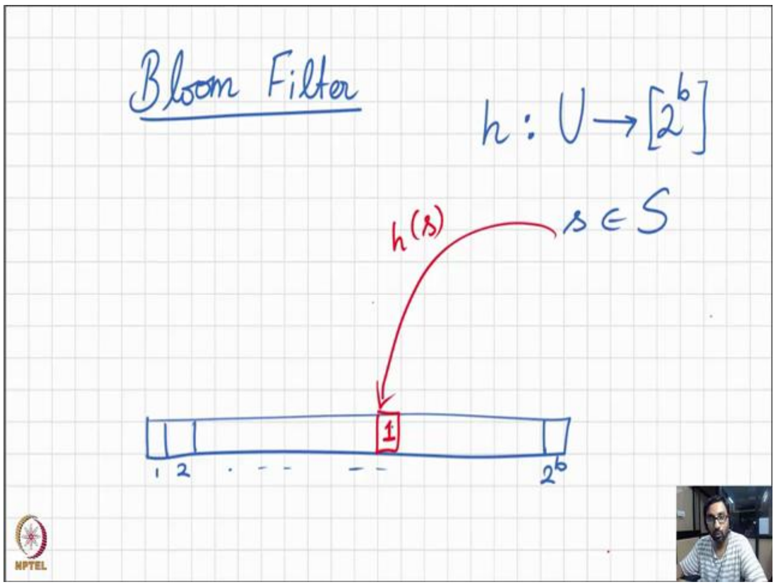
Let us first look at a first cut approach. We are going to look at the finger printing approach, where in we are going to use a hash function. So, every time a user id is included into the set S, the id is hashed, and a smaller finger print basically the hash value is going to be stored in a manner that it can be easily searched for each user id S. From the set of user ids that we have included so far, h of s is the hash value of S, and it is also called the finger print.

Basically, it is a way to identify the user I d s and these finger prints have to be stored in some dictionary. So, for example, the let us consider the case, where these finger prints these hash values are stored in a balanced binary search tree. So, when we have to check whether a particular user Id is already used then we have to compute the hash value of the user Id for that we need to check. And using the hash value we search in the balanced binary search tree and check to see if that hash value is present in the balanced binary search tree.

Now if the hash value is not in the balanced binary search tree, then for sure we know that user Id is not taken so far. So, no false negatives are allowed. However, if the hash value is in the balanced binary search tree that is only evidence that the user Id is possibly taken, it is not guaranteed because multiple items from the universe could potentially hash into the same fingerprint. So, unlike a human finger prints these are not unique, because typically when we think of these fingerprints, these hash values, they are going to use fewer bits than the number of bits needed to represent each element in the universe. So, there is going to be some loss of information and therefore, there is going to be collision in the hashed values

So, there is no guarantee that the user id is not taken, but nevertheless it is good evidence and this is why you sometimes encounter false positives, where potentially good user id's still rejected by the system. This you may wonder how why not you something like a chain hashing that is a little bit of over kill because for chain hashing, we actually have to store the items, it is a dictionary. So, what we are talking about here is not a dictionary, but really just set membership which is an easier problem, but a very commonly encountered problem in big data applications. And so we want to saw the set membership problem more efficiently than we saw the dictionary problem.

(Refer Slide Time: 07:08)



So, with that being the case let us look at a well known technique to implement the set membership data structure. This data structure is called the bloom filter named after the person who invented this data structure. To motivate the idea behind bloom filter is let us start with a preliminary approach.

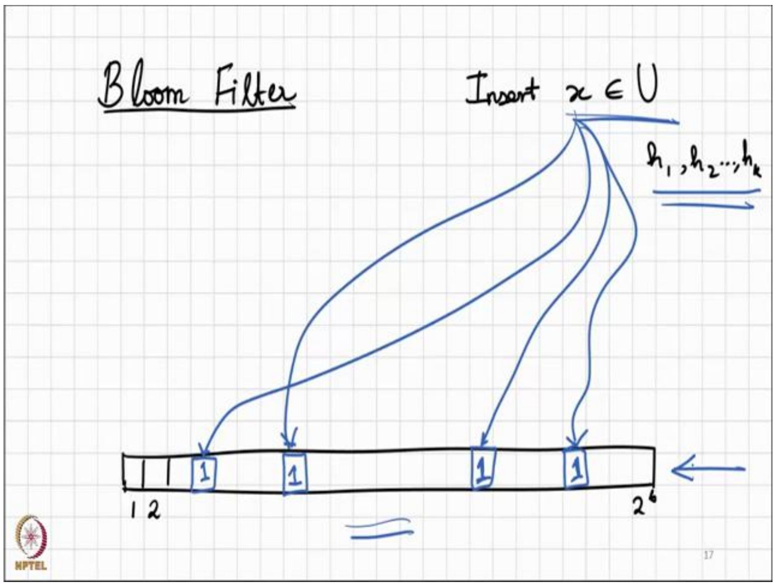
Now let us say that we have a binary array of 2 raise to the b cells. And let us say we have a hash function h that takes elements from the universe, and maps them to the indices ranging from 1 to 2 raise to the b. So as each element s or each user ids enters a system and you can potentially apply this to a streaming setting, where each element arise one by one. And as the element S appears, we hash the element and find the location where it will the index to which it hashes and set the value in that index position to 1.

I forgot to mention that this binary array is initializing such that all entries are 0 in the beginning. So, any time a new item, new user Id appears, its hash value points to an index location and that index location; if it is not already set to a one, it will be set to a one. And so this is one way to implicitly store all the fingerprints.

And this certainly is an improvement over using a balanced binary search tree to store all the fingerprints, because well the number, you do not need to store all the bits of the fingerprint, you just have to you can be we are implicitly storing the finger prints as indices in this binary array. And moreover the query time is just o of one when we get a new user id, all we have to do is compute the hash value and go to the particular index location that it points to see whether that particular index location set to is a one or not.

The one down side we may need to keep in mind is that we will have to pre compute the size of the array while the balanced binary search tree can grow as your business grows and as more user id’s as show up. In this technique, we have to pre calculate the number of user id’s that we are potentially have in the company, and therefore, we need to provision for the appropriate thumper even before the company takes off and this could lead to miscalculations of course, so we need to be careful about that. So, this is great, but this is there is something more we can do; right now we are using a single hash function.

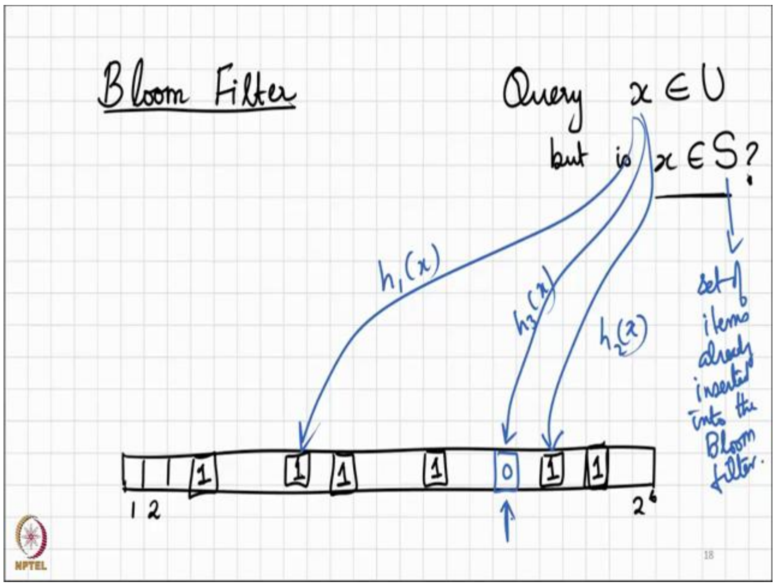
(Refer Slide Time: 10:15)



Let us now try to understand how the bloom filter will work when we have access to k different hash functions h 1 through h k. So, we have 2 raise to the b binary cells and that is available here, they are all initialized to 0. Now every time a new item shows up and here we have item x drawn from the universe, when the new item shows up, here is what we do we take the item x, and hash it using each one of the k hash functions. So, let us say h 1 of x leads us to this location, so then we will place a 1 in that location. And let us say h 2 of x leads us to this location, we would now place a 1 in that location as well.

And let us for example, let us say h 3 of x hashes into that location, we will place a 1 there as well. So, in those k locations, so we will continue to do that and let us say the kth hash leads us here, we will place a 1 there. So, as you can see for each item x that needs to be inserted, we will have to apply the k hash functions and they all hash into the same table. So, in the bloom filter we only have one array one binary array and each one of the hash functions is used to hash the value x and this leads each of these hash functions to a particular index where that index element is then set to a 1. Now this is how the insertion takes place.

(Refer Slide Time: 12:17)

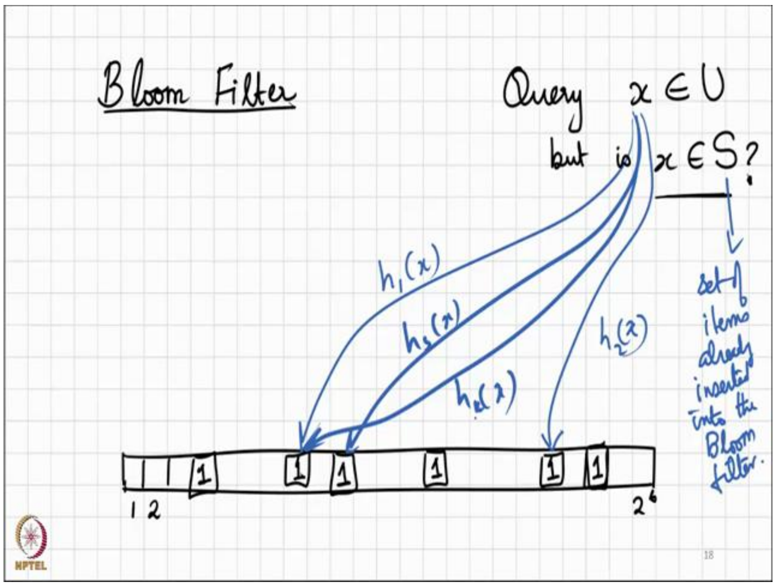


Now, let us see we would do a query. So, when we want to query an item we are given an item x which is drawn from the universe, and we want to find out if that item is actually in the set S that is currently stored in the bloom filter. And recall since several items have probably already been inserted into the bloom filter, several bit locations are already going to be holding 1s, but then there is going to be several 0s as well.

Hopefully, the bloom filter is not filled with all ones and you will see why that could not be a good situation. So, let us consider this situation where we have several ones in the bloom filter array. Now here is what we will do in order to test if x is actually in the set S or not. So, we will take each hash functions the same k hash functions, and we will see what is h 1 of x. Now if h 1 of x is leading to an index, where we have a 1, we will continue. Now let us see where say h 2 of x is also leading to a bit position where we have a 1. And let us say h 3 of x however, leads us to a bit position where there is a 0.

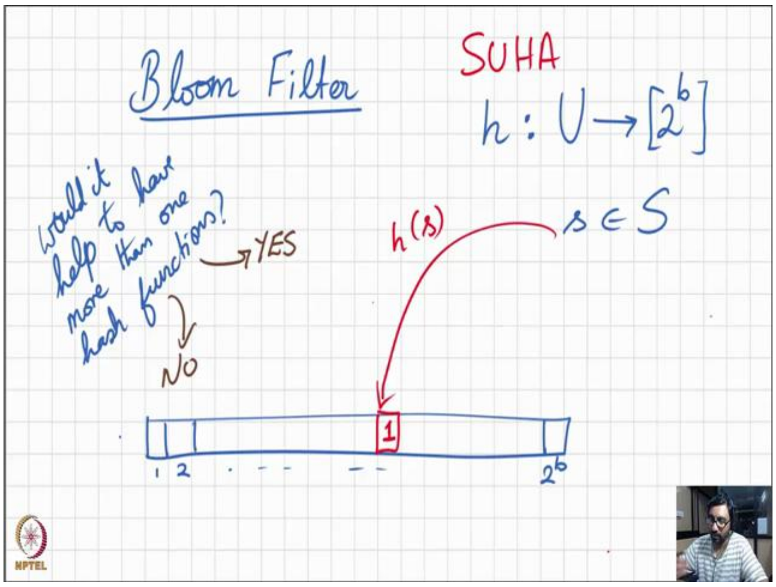
In this situation we know that x does not belong to the set S, and why is that because had x already been inserted and remember S is the set of items already inserted into the bloom filter. So, now when h 3 of x leads to a 0 bit, then we know for sure that this particular x value is not in S, because had it been inserted this location would have been set to a 1. So, we know so then we can immediately say for sure and we are 100 percent sure now that x is not in the bloom filter.

(Refer Slide Time: 14:47)



On the other hand, suppose all of the hash functions say h 3 of x leads to a 1 and so on, and including say h k of x if all of them lead to indices within the bloom filter, where we see a one bit stored then we will respond saying x is in the set S. And it is easy for you to see that there is still a possibility of having made a mistake. So, for example, it could have happened that several different other items even though let us say x was never inserted, but other items were hash into the same location. So, they all just happen to be once and these bit positions which we are query now were not set by x, but were set by other values so that is very much of possibility, so the that is why bloom filters sometimes have false positives, but they never will have false negatives.

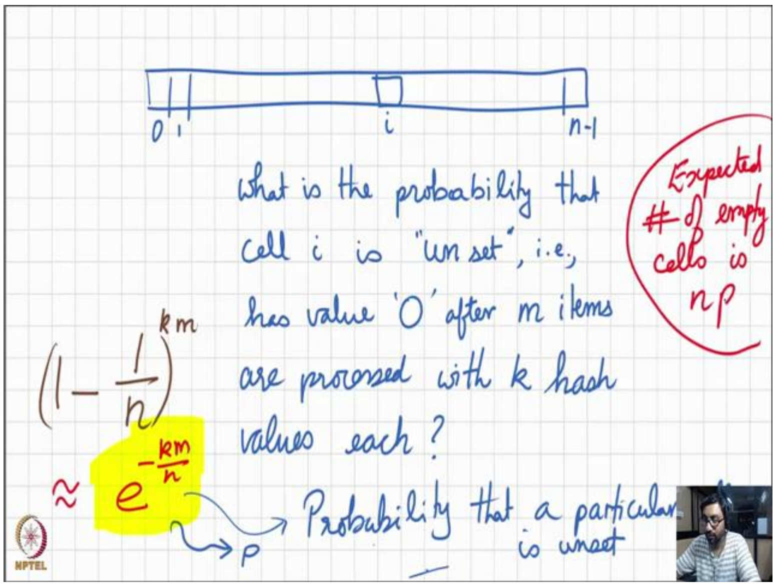
(Refer Slide Time: 15:58)



Would it help to use more than one hash function? On the one hand, more number of hash functions, we will be able to find a cell that is not set to one in case the item we are looking for is not already in their set S. But on the other hand, if we use too many hash functions, for each of the items that we have seen so far, if the number of hash functions we use is too large then each one of them fills up this binary vector by setting a large number of the cells to ones, and soon all the cells or most of the cells will be set to a 1.

So, find so when you have an encounter user Id that is not already in the set S that user Id when you hash it to the range 1 to 2 raise to the b, it is since the cells are all already set to a 1, they will all the hash values where fall into cells that are set to a 1, which means that we are going to conclude that that particular user Id is already taken and therefore, that is wrong, so that is a problem. So, there is a correct balance between two few number of hash functions and two many hash functions. So, let us try to understand this balance a little bit carefully now.

(Refer Slide Time: 17:31)



What is the probability that cell i is "un set", i.e., has value '0' after m items are processed with k hash values each?

$$\left(1-\frac{1}{n}\right)^{km} \approx e^{-\frac{km}{n}} \to p$$

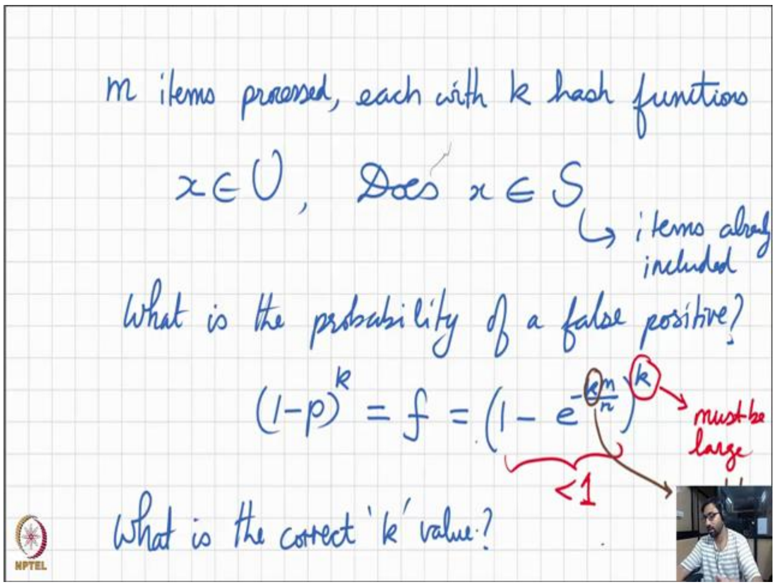
Probability that a particular is unset

Expected # of empty cells is np

So, let us assume that we have a binary vector of length n, and let us focus on one particular fixed cell. The question is what is the probability that particular cell is unset none of the hash values so far have mapped any of the items to that particular cell so far, what is the probability of that after m items of process and each item has been processed with k hash function. Well, each of the hash functions will take us to that cell with probability one over n, so each time an item is hashed that particular location will escape being hashed into the probability 1 minus 1 over n.

And how will this cell i escape being hashed into when m items have been processed, and each one of them with k hash functions, well this whole probability raise to the power k times m. And this we all know is approximately equal to e to the minus k m over n. Now this value is essentially the probability that a particular cell is going to be unset. So for e s of notation let us just denote this by lower case p. So, from this, we can conclude that the expected number of empty cells after m items have been processed it each with k hash functions is going to be n times p or n times e to the minus k m over n.

(Refer Slide Time: 19:37)



m items processed, each with k hash functions

$x \in U$, Does $x \in S$

$\hookrightarrow$ items already included

What is the probability of a false positive?

$$(1-p)^k = f = \Big(1 - \underbrace{e^{-\frac{\boxed{k}m}{n}}}_{<1}\Big)^{\boxed{k}}$$

must be large

What is the correct 'k' value?

Now, let us consider this scenario, where m items have been processed, and each of them have been processed with k hash functions. And now we get a particular item a user id from the universe, let us call that x. The question is does this item x already belong in the set S. So, what we are going to do now is we are going to take x, we are going to hash it using each of the k hash functions and check to see whether each of the indices pointed to by the hashed values is set to 1.

If all of them are set to 1, we are going to say yes, this set this item x is already in the set S. Of course, this could be wrong with some probability which we have to analyze a little bit carefully, but on the other hand, with good probability we will have to be able to say find out that this item if it not already in S, we should be able to report that it is not already in s.

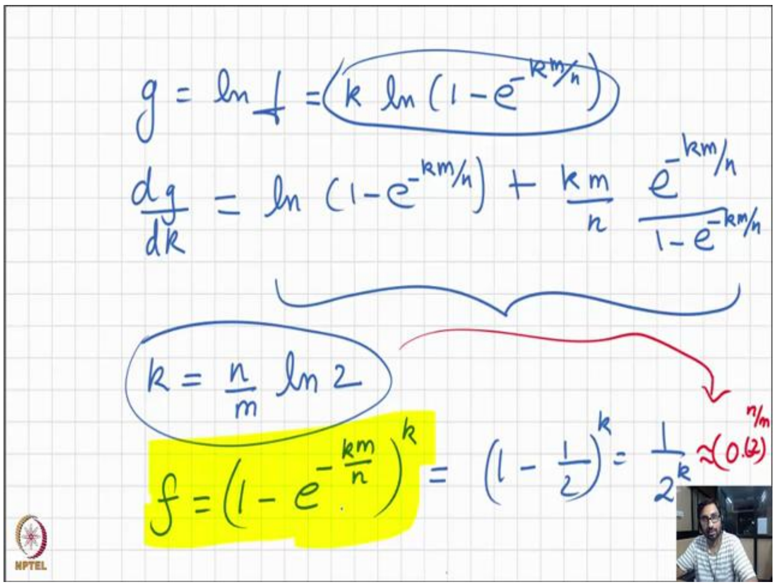
So, now, what is the probability of a false positive? Each time we hash x with probability one minus p will going to have to land in a cell that is already set to 1. And keep in mind that we are for our current purposes, assuming that x is already in S, but we are using hash functions that a way we simply uniform hashing assumption. And so when x is hashed given that a fraction n times p fraction I mean set of items are already filled with ones on expectation, the probability 1 minus p our hash value is going to fall into a cell that is already set to a 1, but that is with 1 hash value.

To get a full false positive, all k hashed values should land in cells with cells that already set to a 1, so that event is going to happen with probability 1 minus p raise to the power k. And let us denote this by f, and let us expand it out to see what it looks like. And say f is going to be 1 minus e raise to the minus k m over n the whole raise to the power k.

Now, let us make sure our intuition is fitting this formula that we have. And remember f is the probability of a false positive. Now you notice that k is appearing in this expression twice. Now let us focus on this appearance. As the value of k increases here, since this quantity is less than 1, the overall value is going to decrease as k increases. So, this k must be large. On the other hand, let us focus on this other k here; as this k increases e to the minus k is going to decrease. And if e to the minus k m over n decreases, 1 minus e raise to the minus k m over n will increase, and that is bad because that increase is the false positive. So, this k is giving us the opposing force if you will and to get a good false positive probability, here we want looking at this k we wanted to be small.

So, as you can see clearly, there is a tension here on the one hand, we want larger value of k; and on the other hand, we want smaller value of k. And this fits neatly with our intuition that we already built where in as the number of hash function is increase on the one hand we see false positives increasing, and on the other hand the false positives are decreasing. And so we have mathematical evidence that intuition is in fact correct. So, of course, the question that remains is what is the correct k value.

(Refer Slide Time: 24:52)



$$g = \ln f = \boxed{k \ln\left(1 - e^{-km/n}\right)}$$

$$\frac{dg}{dk} = \ln\left(1 - e^{-km/n}\right) + \underbrace{\frac{km}{n}\,\frac{e^{-km/n}}{1 - e^{-km/n}}}$$

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

$$f = \left(1 - e^{-\frac{km}{n}}\right)^k = \left(1 - \frac{1}{2}\right)^k = \frac{1}{2^k} \approx (0.62)^{n/m}$$

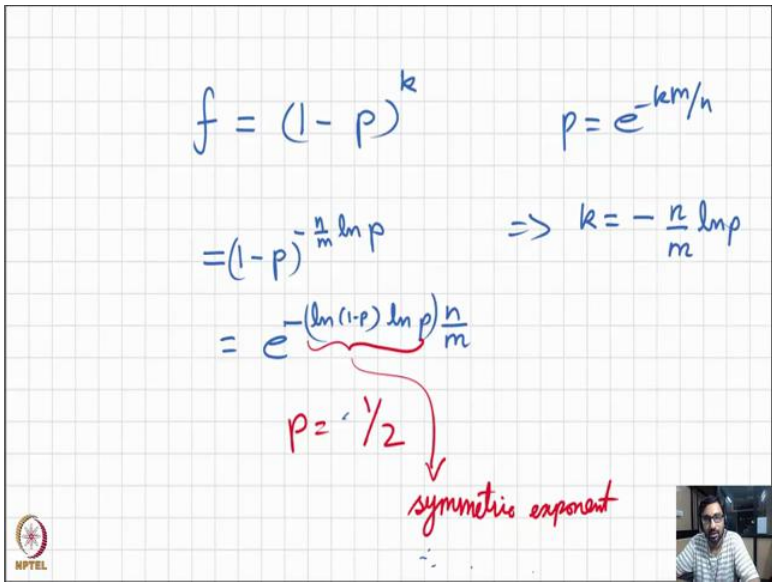
And to answer this, let us look at g equal ln of f and see what value of k minimizes g, because if we can minimize g then we are also going to be minimizing f. So, in order to find the optimal value of k, well this is classic calculus. We just have to look at d g over d k and that is going to equal this expression when we differentiate the right hand side here. And as a quick exercise, you can see that this expression gets minimized when we set k to equal n over m ln 2. And substituting this value of k into the expression for f, we get 1 minus half raise to the power k that is equal to 1 over two raise to the k.

And substituting the value of k, where ultimately going to get approximately 0.62 raise to the power n over m. So, let us look at the intuition here and recalls the number of bits in the bit vector that we used to store the bloom filter 80. And m is the total number of user id’s that are in the set s, so the probability of the false positives drop exponentially as this fraction n over m increases, so that is a very good sign here.

From a designer's point of view, this expression is very nice, because we can now use this expression to optimize the parameters that we care about. So, for example, if let us say f is fixed the probability of false positives, let us say that gets fixed down to something like 1 or 2 percent. And let us also say that the number of hash functions that we have or disposal some number say 10, then we can compute n, because well we assume we know m, there is an optimal size n for the length of the bit vector.

So, we can use this expression to optimize for n or k or whatever the parameter of interest might be. So the tradeoffs are coming up nicely in this expression. And we can just play with these tradeoffs until we get the right system for the application that had. Let us also look at this expression a little bit more carefully and try to get some more insight out of it.

(Refer Slide Time: 27:59)



$$f = (1-p)^{k} \qquad p = e^{-km/n}$$

$$= (1-p)^{-\frac{n}{m}\ln p} \qquad \Rightarrow \quad k = -\frac{n}{m}\ln p$$

$$= e^{-\underbrace{(\ln(1-p)\ln p)}_{p = 1/2 \;\to\; \text{symmetric exponent}}\frac{n}{m}}$$

Now consider this expression f equal to 1 minus p raise to the power k. What is k in terms of p? Now recall that p equals e to the minus k m over n and this implies that the value of k equals minus n over m ln p. So, let us apply that here. And this is equal to e to the minus ln 1 minus p ln p, the whole times n over m. And if you look at this expression, it is clear that this is going to be minimized when p equal half. And this can be deduced by just looking at the symmetry of the exponent here.

This actually has a very nice intuition. This tells us is that the bloom filter works best when about half the bits in the bloom filter are set to a 1; and the remaining half is set to a 0s. Such a bloom filter will essentially look like a random bit string. So, when you intuitively think of searching for an or asking a query in a particular user id or an item that is not already in the user id’s that are taken, with probability half each hash function is going to reveal to you that this particular user id is not yet taken. And you can see the exponential drop in the probability with which we will be able to detect that this particular user id is not taken as we try out each hash function.

So, this again gives us some very good intuition and with that we have come to the end of our lecture on bloom filters. And just to recap we have looked at a classic big data application, the set membership application which seeks to answer the question is a particular item in a given set. And in the spirit of big data applications, we are willing to live with some amount of error; in particular, we are willing to live with some small fraction of false positives. And as in return, we would like the bloom filter to be to be compact in size and it should be capable of handling streaming data distributed data so on and so forth is basically a sketch of the entire set S.

And we saw how you we can analyze the bloom filters which are the data structure we studied for set membership problem. We saw how the bloom filters can be analyzed when we make use of the simple uniform hashing assumption. And just to recall what we discussed early on we looked at the birthday paradox, we looked at how balls and bins can be used to model birthday paradox.

And then we looked at chain hashing which can be used to implement dictionary data's ADT. And we used our knowledge of balls and bins; and in particular, we analyzed balls and bins for their maximum load, and we applied that intuition into chain hashing to understand how to understand what the query time of chain hashing is and so on and so forth.

So, this whole lecture is been about hashing the simple uniform hashing assumption, the balls and bins model, and how these interoperate with each other to help us build some very nice data structures that are suitable for big data applications, so that is it.

Thank you.