

**Algorithms for Big Data**  
**Prof. John Ebenezer Augustine**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Madras**

**Lecture - 31**  
**The Count-Min Filter for Heavy Hitters in Data Streams**

In this segment, we are going to look at The Count-Min Filter and this count-min filter is used to detect heavy hitters and so typical example application would be in a network, where you want to monitor traffic and you have a packets avoiding from various sources, and reaching towards various destinations and these heavy hitters are those sort of what people call elephant flows. They are talking about those source destination pairs that are taking up a lot of bandwidth. So, just by quickly observing the traffic flowing through the network, we need to be able to identify those source destination pairs, for example, that are most frequently using a particular router that particular router.

So, in this context we can use the count-min filter. So, let us be more little bit precise about the data model the data stream model.

(Refer Slide Time: 01:29)

Data Stream Model



$X_1, X_2, \dots, X_t, \dots$

Total count of item  $i$   
 $\text{Count}(i, T) = \sum_{\substack{t: i_t = i \\ 1 \leq t \leq T}} c_t$

$\rightarrow (i_t, c_t)$  "Cash Register Model"

item  $\rightarrow$  Positive increment value

Heavy Hitter for a threshold  $q$  is an item whose total count  $\geq q$ .

Our items in the data stream arrive one after another, call them  $x_1, x_2$  and so on up to some  $x_t$  and they follow what is called the cash register model. So, each  $x_t$  each element in the data stream, say  $x_t$  comprises two items, two components with there is the item  $i_t$ . So, this could be, for example, the source destination pair that we are interested that this


particular packet is going connecting between, and  $c_t$  is a positive count it is an increment value.

Now, perhaps this could represent the number of bytes that are been sent from this particular source. This destination or something like that, but keep in mind that the same item can get repeated. So,  $x_1, x_t, x_t$  plus 200 all of them could be the same item. So, if you were to look at it from the perspective of the network example, the same source destination pair can send some amount of data at one point in time little bit later some more amount of data and so on and so forth, and this cash register model is going to update the total count and the total count of a particular item  $i$  is denoted by count  $c_{i,t}$  and this count is given by the summation. So, we sum over all time  $t$  and we look for those items  $i_t$ , that match the item  $i$  that we are interested in counting and when they do match we have it a count.

So, that is the count of particular item  $i$  from time period, time one all the way to time capital  $t$  and we define a heavy hitter with respect to a threshold  $q$  and simply any item whose total count is at least  $q$ . So, that is a very natural definition for heavy hitter and that will help us to formally understand an approach this problem. So, here is the output that is required.

(Refer Slide Time: 04:10)

## Required Output $(q, \epsilon, \delta)$





A set of items such that

- Every heavy hitter for threshold  $q$  is included
- an item with count  $< q - \epsilon Q$  is included with probability at most  $\delta$

↳ Total count over all items

Note: items with total count  $\in [q - \epsilon Q, q)$  are don't

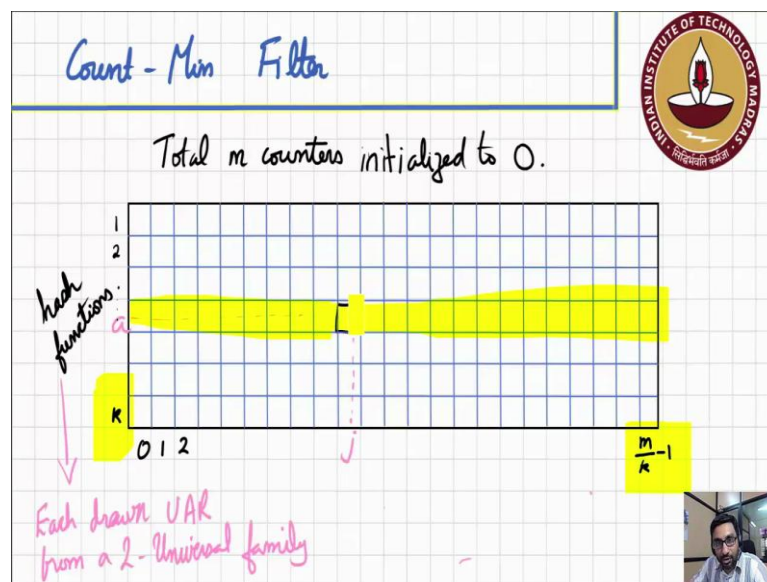
Now, keep in mind that this problem, the heavy hitter problem is going to come with three parameters. Apart from the stream it is going to come with the threshold  $q$  that

divides it is heavy hitters from non-heavy hitters then, there is the error parameter epsilon and the confidence parameter delta.

Now, the output will be a set of items such that every heavy hitter that has a count more than the threshold  $q$  must be included. This is the deterministic requirement and, but you know some items that are not heavy hitters may also be included. So, in another words there can be some false positives, but an item with count that is significantly far from the threshold. So, that is whose count is less than  $q$  minus epsilon times uppercase  $q$  is included the probability at most deltas.

So, it does not get, it should be hard for an item with a small count of  $q$  minus epsilon  $q$  to be included in the set of items that we output, here what is this uppercase  $q$ ? It is the total count over all items not just that particular item, but over all items and here is something that you need to notice, if there is an item with total count between  $q$  minus epsilon  $Q$  and  $q$  then they do not care, if they get included that is fine, if they do not get included that is also fine. We do not particularly care about whether they can be included or not.

(Refer Slide Time: 06:10)



The solution to this problem is, why is the count-min filter and this as you can imagine uses hashing. But in a very interesting way would going to maintain some number of counters, we are going to maintain particular, we are going to maintain  $m$  counters, we are going to layout the counters in this sort of two dimensional array for. So, they are

going to be  $k$  rows and  $m$  over  $k$  columns. Now, keep in mind that it is starting at 0 here and each row corresponds to a hash function. So, this row  $a$ , corresponds to hash function  $h_a$ , each of these hash functions is drawn uniformly at random from a large 2-universal family of hash functions.

(Refer Slide Time: 07:08)

Count-Min Filter: Processing  $(i_t, c_t)$

$(i_t, c_t)$

$h_a(i_t) = j$

For  $1 \leq a \leq k$  increment  $(a, h_a(i_t))$  by  $c_t$

Now, given an item given the arrival of some  $i_t$  comma  $c_t$ ,  $i_t$  is the item,  $c_t$  will be the count of that particular arrival of item that item  $i_t$ , here is what we do we hash the item into  $e$ . We use each of the hash functions and hash it and let say now, we use  $h_a$  and it is get hashed into location  $j$ , that is this location over here. So, then we go down row  $a$  because we are currently considering hash function  $h_a$ , we go down row  $a$  up to column  $j$ , that is this location over here and remember that is a counter and we simply increment that counter with  $c_t$  and this is of course, done for all hash functions.

Remember, we have  $k$  hash function; we do this for all  $k$  hash functions. So, we for each of these hash functions, we go to the we hash the item and then walked down the row corresponding to that item and find the hash location and increment the counter and of course, we have to start the all the counters at 0.

(Refer Slide Time: 08:40)

Now, let us look at an example just to see how this works. So, here is the arrival of sequence of the items and their counts. Now, what do we do? We look at the first item  $i_1$  and find out all the locations that it hashes into. So, we are using  $k$  different hash functions and wherever it hashes into each according to that particular, we increment the counter. So, for example, we since we started we initialized all the counters to 0. Let say, we will make this  $c_1$  and then we process item  $i_2$  and let say  $i_2$  hashes into these locations.

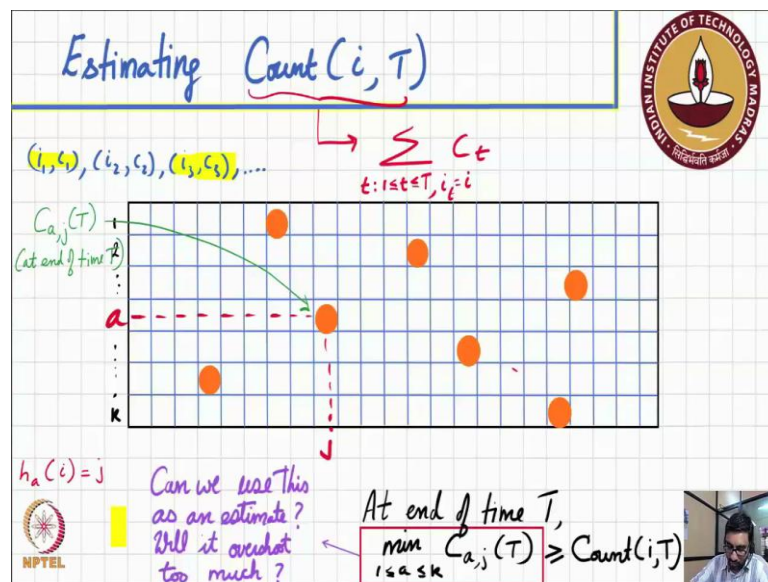
Let me use different color. So, let say a  $c_2$ ,  $c_2$ , but remember it can hash in to a location that is already being used, for example, this could be  $c_1$ ,  $c_2$  over here. So, you that is the case you go and  $c_2$  to the previously existing quantity and then we process the third item, now let say, for example, just for the argument sake let us take  $i_1$  is actually equal to  $i_2$ , sorry  $i_3$ . So, these 2 are the exact actually the same item being repeated. So, then we will have to go in and add,  $i_3$  will be hashing into the exact same locations that  $i_1$  hashed into and therefore, we have to go in and add these quantities.

So, here it is going to be all in this location, this is how this count-min filter works and as you can imagine, how would you find out whether an item is a heavy hitter or not. Let say, let us look at some arbitrary item  $I$ , we will now look at the hash locations where  $i$  maps to and look for the smallest counter in those locations, and here is the reason why, for example, let us say that  $i$  is actually equal to  $i_1$  then we will be looking at these

locations, notice that because of some overlap in the way the hash function works some of the counters are going to be large, but if you look at the minimum and in this case we processed only few.

There is more than one minimum, they are all  $c-1$  plus  $c-3$  and notice therefore, that we this minimum that we are looking at is going to be at least the count of the item  $i$ . It could be a little bit more, especially if once the count-min table starts to get filled up, there could be more collision and therefore, there can be more, I mean the minimum could actually be corrected with other because they also has into that location, but nevertheless the minimum is going to be a lower bound on the actual count of the item.

(Refer Slide Time: 12:26)



So, just to formalize, let us denote a location on or rather the count value of location, a comma  $j$  c a comma  $j$  t and this is the value of the counter at the end of time  $t$ . So, what we have done is we have processed several items. We have counter values in this table and remember we are trying to estimate the count the total count of each item  $i$  comma  $t$  to figure, whether it is a heavy fitter or not and this count is nothing, but the summation over the appropriate items in the stream of their count value  $c$   $t$ .

So, as I just mention just to make it formal, suppose we want to estimate the count  $i$  comma  $t$  at the end time  $t$ , we hash the we look at all the hash locations for item  $i$  and those are denoted by  $c$  a comma  $g$   $j$  and we look at their counter value that is  $c$  a comma  $j$  of  $t$  and the minimum of all of them is guaranteed to be at least the count, the actual



count value of the count value of that item  $i$  at the end of time  $t$ . So, that is exactly what we explained a little while ago with their example.

Now, this is just the formal description of the algorithm to estimate  $p$  that the total count, the key question here is we clearly have intuition that it is an upper bound on the total count, but will it over shoot the total count by too much that is the question because if it over shoots then it is no longer a good estimate it is to over shoot just a little bit, but not too much. So, that is the key question and analysis going to be focused on answering that question.

(Refer Slide Time: 14:42)

How much will  $C_{a,j}(T)$  overshoot  $\text{Count}(i,T)$ ?

$$Z_a = C_{a,j}(T) - \text{Count}(i,T)$$

Let  $X_t = 1 \iff h_a(i_t) = j \wedge i_t \neq i$

Then,  $Z_a = \sum_{t=1}^T X_t C_t$   $\Pr(h_a(i) = h_a(i_t)) \leq k/m$

$E[Z_a] = \sum_{t=1}^T C_t E[X_t] = \frac{k}{m} Q$   $E[X_t] = \frac{k}{m} \forall i_t \neq i$

Now, we have discussed; how the count-min filters works? How each item is processed? You go to the appropriate location within each row and increment the counters and then when you view to process all the items, if you want to identify the heavy hitters we just have to look at the hash location of each item that you are interested in and see whether and see all the hash locations and find and spot the minimum and that minimum is an upper bound or the total count value for that item and so, all items whose minimums are more than the threshold value should be reported as heavy hitters.

So, that is how the count-min filter works. Now, we have to analyze and show that the excess, we did not over shoot too much and for that purpose we are going to define the random variable  $Z_a$  and that is equal to the actual counter value at a comma  $j$  and so, keep in mind that we are focusing on a particular hash function  $h_a$  and this member this

will be therefore, an upper bound on this count  $i$  comma  $T$ . So, this quantity is going to be a non negative quantity and this difference tells you how much we have overshoot and so, naturally  $Z_a$  is what we want to. Now, to get handle on  $Z_a$ , we define an indicator random variable  $x_t$  and that  $x_t$  is going to be one for all those items whose when that hashed with  $h_a$  lead to the same location in  $j$  as item  $i$  remember.

So, we are just right now interested in how much  $c_a$  comma  $j$  of  $t$  is overshooting the total count of  $i$  comma  $t$ . So, this  $x_t$  equal to 1 whenever the hash location matches that of item  $i$  and number we are not interested we are interested in excess. So, this excess should be created by items other than the item  $i$  that is of interest we need this extra condition to ensure that we are only counting those items that are not  $i$  itself under this indicator variable.

It is clear to see that  $Z_a$  will be the summation over all time  $X_t$  which gives you the indication as to whether that particular item should be counted in  $Z_a$  or not times  $c_t$  and remember that we are using hash function and we are in particular using hash functions drawn uniformly at random from 2-universal family of hash function. So, the probability that particular item  $i$  that is not  $i$  will have the same location, the these two items will have the same hash location with probability at most one over the total number of hash locations, but remember that in this in this table number of columns is  $m$  over  $k$ .

So, this probability is therefore,  $1$  over  $m$  over  $k$ , which is nothing, but  $k$  over  $m$  and this means that the probability with which a particular location  $i$  mean particular other item will actually hashed in the same location we know the that is at most  $k$  over  $m$ . So, this indicator variable will have the expected value  $k$  over  $m$ , which means that now we can get a bound on the expectation of  $Z_a$  and if we plug-in the values we are going to get.

We can use the linearity of expectation to say that expectation  $Z_a$  is equal to  $t$  equal to one to  $t$  count of  $t$  times the expectation of  $x_t$ , but keep in mind that  $e$  of  $x_t$  is same for all  $i$  as long as it is not equal to  $i$ ,  $t$  is not equal to  $i$ . So, you can take that out and moreover notice that summation over all  $c_t$  is nothing, but the total summation  $q$  over all items and over all time. So, we can therefore, get this bound on the expected expectation on  $Z_a$  and with expectation, now we are ready to apply Markov's inequality.



(Refer Slide Time: 20:07)

How much will  $C_{a,j}(T)$  overshoot  $\text{Count}(c,T)$ ?

$$\Pr(Z_a \geq \epsilon Q) = \Pr(Z_a \geq \frac{\epsilon m}{k} \frac{k}{m} Q) \leq \frac{k}{\epsilon m}$$

Since  $Z_a$ 's ( $1 \leq a \leq k$ ) are independent

$$\Pr(\min_{1 \leq a \leq k} Z_a \geq \epsilon Q) \leq \left(\frac{k}{\epsilon m}\right)^k \leq \delta$$

Where  $k = \lceil \ln \frac{1}{\delta} \rceil$  and  $m = \lceil \frac{L}{\delta} \rceil$

Now, we ask, remember the whole goal is to ensure that the probability be that we will be reporting an item whose reporting account value that is more than epsilon q over the correct count that is we are trying to that is a bad event. So, we capture that bad event in this expression over here. So, probability of this bad event, where  $Z_a$  exceeds epsilon q now can be written as  $Z_a$  is greater than or equal to and I want write this in terms of the expectation.

So, I therefore, appropriately modify this constant over here and using Markov's inequality, I can bound this probability to k over epsilon m, but keep in mind each of the hash functions is chosen independent of each other. So, what we have is the probability that one of the hash functions will exceed this epsilon Q. Now, we want care if one of them exceeded that epsilon Q, we just wanted to ensure that at least one of them does not exceed this epsilon Q.

So, what instead of that event, we instead bound the probability of all of them the bad. The bad situation where all of them exceed epsilon q that is given by this event over here, the minimum over all  $Z_a$ 's is still larger than epsilon Q. So, all of them had to exceed epsilon q and because of the independence of the  $Z_a$ 's, we can upper bound this by k over epsilon m that is for one on other  $Z_a$ 's the whole rise to the k for all  $Z_a$ 's and remember, this quantity this is the probability of the bad event and we want this bad, the probability of this bad event to be at most delta. We can achieve that by setting our k to

be  $\ln 1/\delta$  over  $\epsilon$  and to mention that  $\delta$  is an integer, we take the (Refer Time: 22:55) and  $m$  equal to roughly  $\ln 1/\delta$  times  $e/\epsilon$  and if you plug these values, we will get the appropriate  $\delta$  that we want.

(Refer Slide Time: 23:12)

**Conclusion**

Given  $q$ ,  $\epsilon$ , and  $\delta$ , we can design a count-min sketch with  $m = \lceil \log 1/\delta \rceil \lfloor \frac{e}{\epsilon} \rfloor$  counters and  $k = \lceil \log 1/\delta \rceil$  hash functions such that the identified set of heavy hitters at any time  $T$  is guaranteed to

- Contain all heavy hitters w.r.t.  $q$  and
- $\Pr(\text{item } i \text{ with } \text{Count}(i, T) < q - \epsilon \text{ is identified})$

So, here is the conclusion, we have what we have looked at is given a threshold  $q$  error parameter  $\epsilon$  and confidence probability  $1 - \delta$ . We can design a count-min sketch with  $m$  equal to  $\log 1/\delta$   $e/\epsilon$ , total number of counters of course. We have to arrange this in two dimensional array, but this is total number of counters and we will also need to use  $k$  hash functions in  $k$  here is roughly  $\log 1/\delta$  and these are hash function choosing uniformly at random from 2-universal family of hash functions and these  $k$  hash function should be chosen independent of each other such that the identified set of heavy hitters at any time  $T$ .

So, we can create a count-min sketch that outputs this identified set of heavy hitters at any time  $T$ , which is guaranteed to contain all the heavy hitters with respect to  $q$ . So, it is guaranteed it is with probability  $1 - \delta$  to contain all the heavy hitters and then for any item  $i$  the probability that are item  $i$  whose counter value is actually less than  $q - \epsilon$ ,  $q - \epsilon$  is also identified and included into the set.

Essentially, if false, positive is at most  $\delta$  and of course, this is only true as long as this particular item is far from being a heavy hitter. So, this is count-min sketch and we keep in mind that we only needed use hash functions that are drawn from 2-universal family

of hash functions. As long as they are in eventually independently chosen of each other and this is a very, very nice data structures very compact as you can see the size of the data structures only a constant that depends only on delta and epsilon. So, it is become quite a famous sketch so this is essentially the sketch of the of the data stream that allows us to identify heavy hitters.

(Refer Slide Time: 25:35)

### A FINAL REMARK : Conservative Updates

Normally, we update all  $k$  counters for each item.  
*But do we need to increment all  $k$ ?*

(i,3)

3	8	5	1
4	8	3	2
3	0	5	9

3	8	5	1
7	8	3	2
3	0	7	9

**Conservative Update:**

- Update current minimum to the new minimum
- Ensure all hash locations are no smaller than the new minimum

Let me end with one final remark this one small improvement that we can make and this improvement is the following instead of updating all the hash locations you can be a bit more conservative about our updates.

Normally, we will update all the; for a particular item. We will compute the  $k$  hash functions going those hash locations and increment all the counters, but the question do we need to really increment all the  $k$  counters, remember the code here is to ensure that the minimum among all of them is always no more than or greater than or equal to the total count of that item. So, that is all we required, but if there is some other hash locations which is already got a very high value possibly because of collisions with other hash functions we do not really need to increment them. So, it is best to see this with an example.

Let say that, here is our example of a very small count-min filter. So, what here is what we do So, we are currently in this state over here at time say  $t$  minus 1 and we now have to process the arrival of this item  $i$  comma 3 and the item  $i$  hashes into these locations

that are shown shaded now clearly, now 4 has to be updated to 7, remember the count is 3, fine normally we would also update 5 to 8, but what is the point of updating it to 8 where incrementing the count of item  $i$  and the minimum is going to be 7. So, clearly the total count of item  $i$  is 7 or lesser. So, making this 8 is of no value at all instead, but we should not make it anything less than 7 because that might violate the bound.

So, we all we nearly need to do is to ensure that all hash locations are no smaller than the new minimum the new minimum we know is 7. So, we just have to ensure that all the hash functions has locations are 7 or more. So, the 5 hashes into locate you take going to the location that this sorry you look at this item, this counter with value 5 in it just take it up to 7. Do not have to take it to 8 and  $i$  this hash location have the counter value of 8, it retained at 8, we do not need to increment it at all.

So, this gives us slightly better results, but we are going to skip the analysis because that is outside the scope of what we do not want to talk about, but it is a neat little strict to keep the count min sketch a little bit more sharp. So, that concludes our segment on the count min sketch.