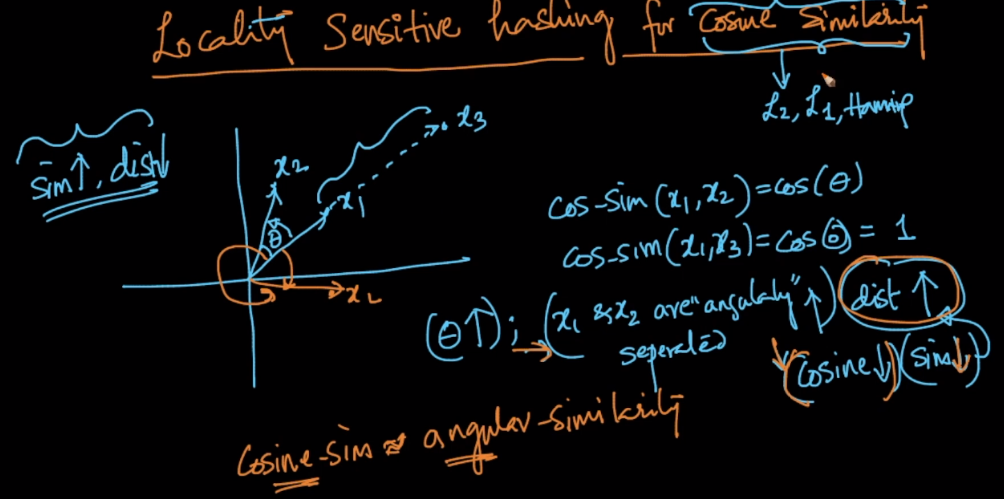
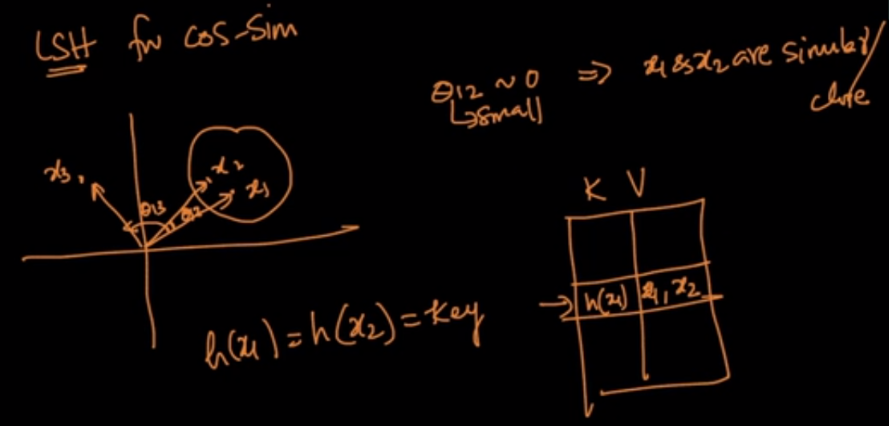
We’ll see LSH for Cosine similarity:

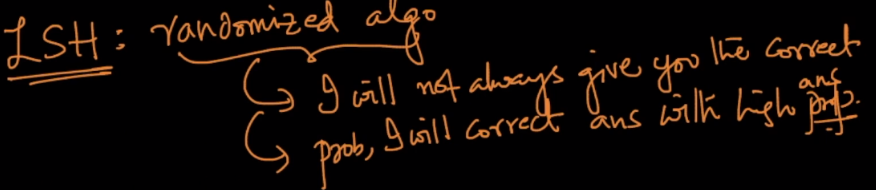
As in cosine similarity the distance is measured angularly, so we do LSH for it.



We’ll do some thing that generate the same key for points which have smaller angle between them.



LSH is a randomized algorithm, which do not gives guarantee that it will generate correct results each time, but it will correct ans with high probability most of the times.



For Any plane let say pie1, having unit vector w1.

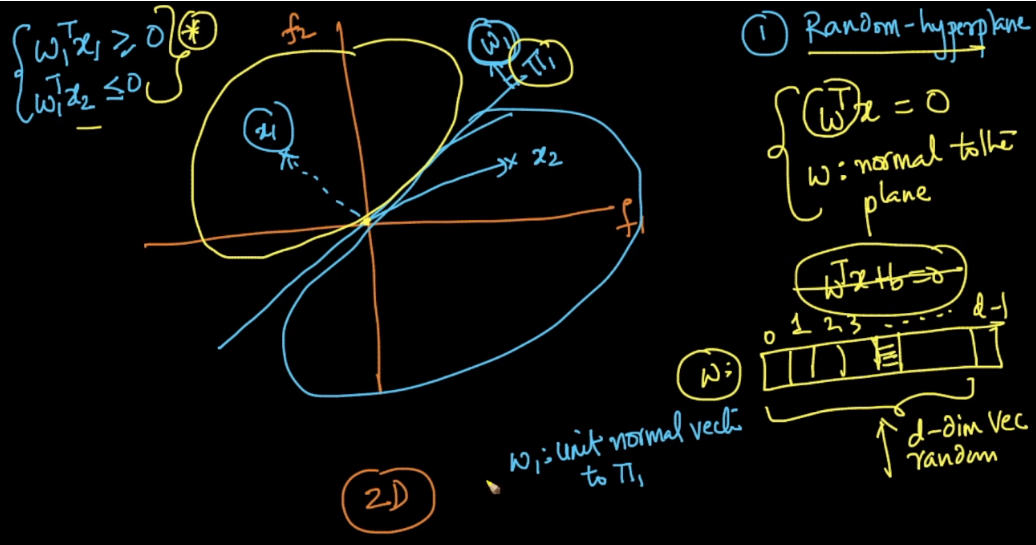
Now for any point x1 if it’s in direction of w1, then

W1T. x1 >= 0

And if x1 in opposite direction of w1, then

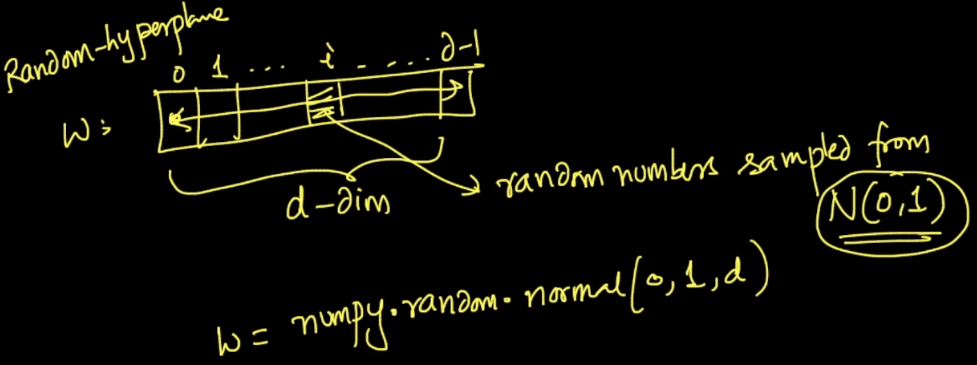
W1T. x1 <= 0

And we know that equation of a random hyperplane passing from origin is W1T. x1 = 0



equation of hyperplane passing through origin is WT  \* X = 0. where W is d-dimensional vector. if we can create d-dimension vector using random number generator we are indirectly creating different random hyperplanes only. each time you run this random number you get different random hyper-plane each time passing through origin.

So we are creating a hperplane passing through origin having unit vector w generate using random number generator.



Suppose we have created three hyperplanes, each having a unit vector.

All these hyperplanes are creating slices(like in pizza), and for each slice we will create a separate entry in hash table whose values will be all the points present in that silce.

**How do calculate key for each slice:**

For each slice we see direction of points in that slice with respect to the unit vector of each hyperplane.

Let’s take example of cluster formed on top (or sliced from plane pie1 and pie3).

Direction of slice points, wrt w1 are same ie W1T. x1 >= 0 therefore we take +1 for this.

Direction of slice points, wrt w2 are same ie W2T. x1 >= 0 therefore we take +1 for this.

Direction of slice points, wrt w3 are same ie W3T. x1 >= 0 therefore we take +1 for this.

So our final vector hash key will be

|  |  |  |
| --- | --- | --- |
| +1 | +1 | +1 |

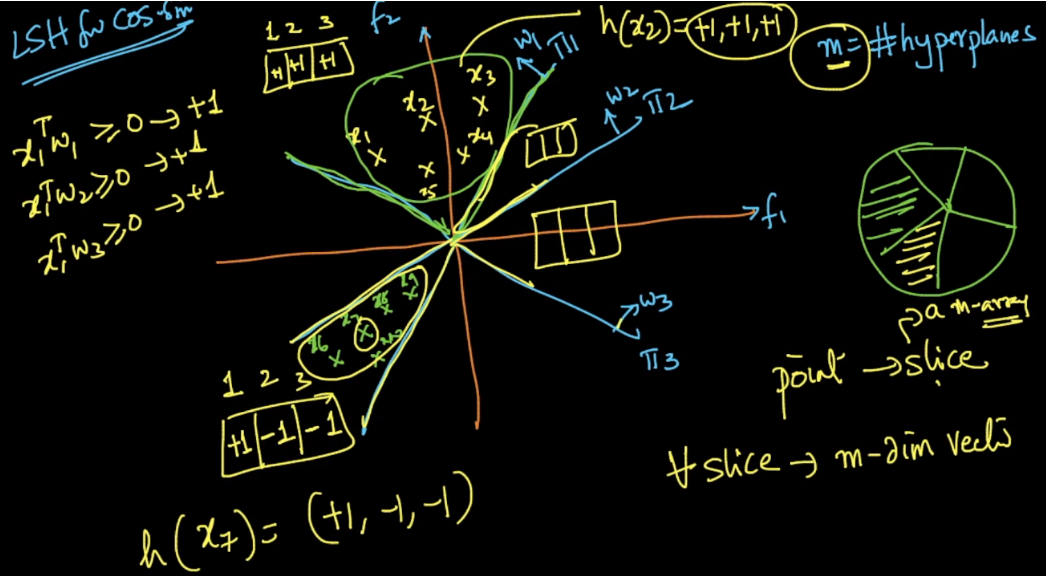
Similarly if we calculate hash key or vector will be

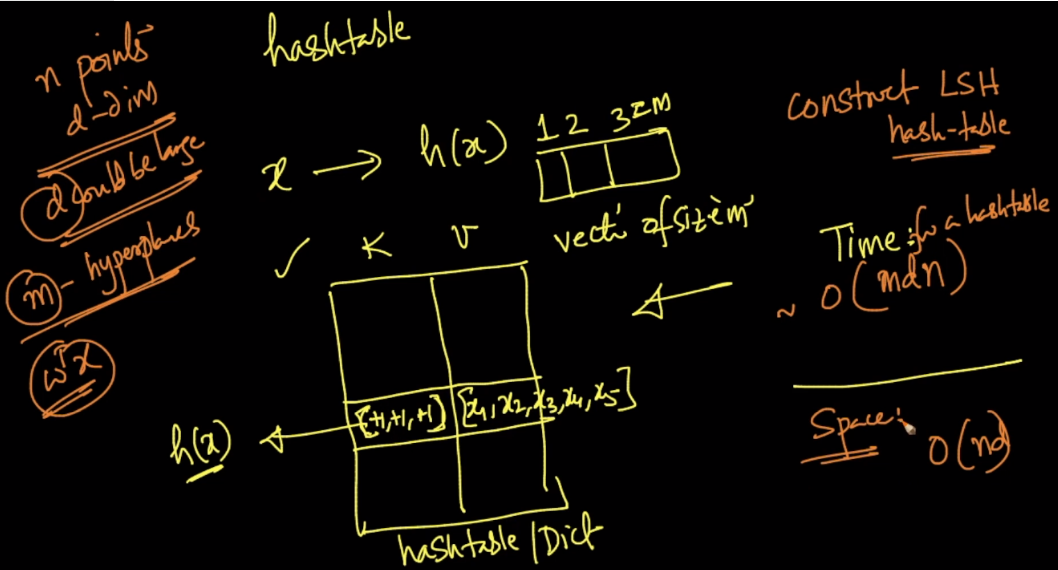
|  |  |  |
| --- | --- | --- |
| +1 | -1 | -1 |

Since they are in opposite direction to w2 and w3.

And similarly we will find the hash key vector for each slice,

And we’ll put that in table where key will be these hash vector and values will be the points present in that slices.





**Time Complexity for constructing hash table:**

Let’s say we create **m** hyperplanes, there are **n** # of points each with **d** dimension.

So calculating W3T. x1 will take ( m \* d ) for m hyperplanes and there d dimensions so d multiplication.

And since we have to this for all **n** points, therefore time complexity will O(m\*n\*d).

**Space Complexity for hash table:** O(n\*d)

**Note:** Hash table calculation is only performed in training phase, after training phase we only use created hash table, ex: for CV or test data.

**How to get NN for query Points:**

Now if any query point comes we’ll do WT. x1 with unit vector of each hyperplane which yield +1 or -1 for each plane, eventually we get a vector which tells us that we have to find nearest neighbors in only those points which are located corresponding to generated vector.

|  |  |  |
| --- | --- | --- |
| +1 | +1 | +1 |

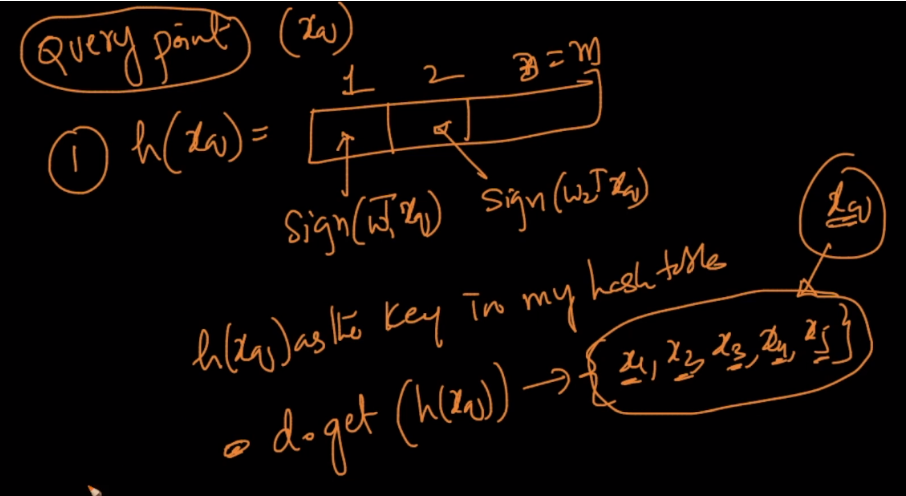
**Time Complexity for querying points:**

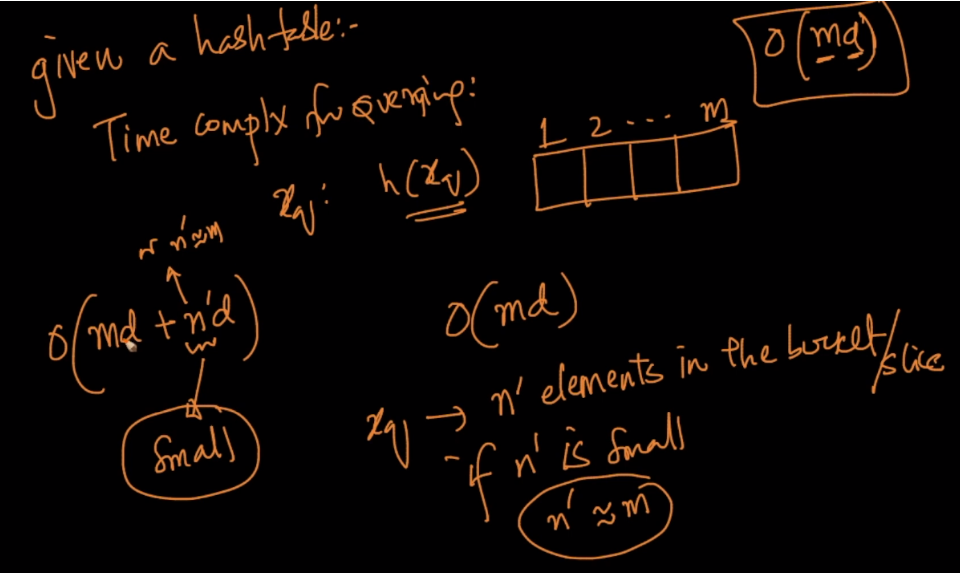
For generating hash key vector: O(m \* d)

For finding nearest neighbors in points get in hash value: O(n’ \* d). where n’ is no. of points in selected bucket.

So final time complexity will be O(m\*d + n’\*d)

Here n is small or approximately equal to m therefore querying time complexity will be O(m\*d)

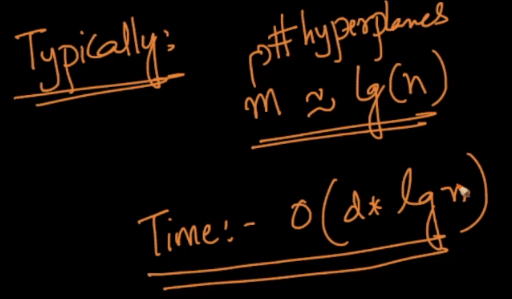




**What should the ideal no of hyperplanes should be generated:**

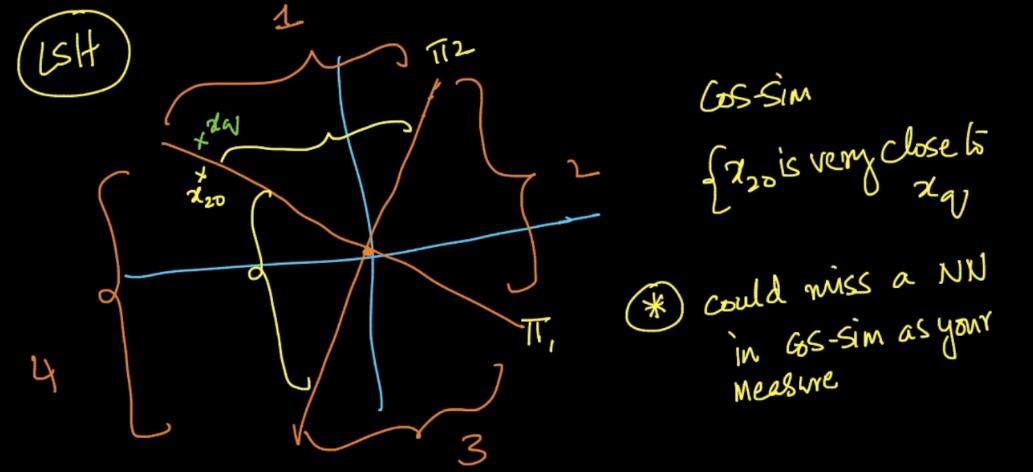
The # of m = log(n), where n is no of points in training data.

Therefore time complexity will be changed as O(d\*log n)



**Problem or drawback or LSH.**

Suppose there is a point xq which is one slice, but it’s nearest to the point that presents in other slice, so in such case we could miss a nearest neighbor in cos-sine as measure of distance.



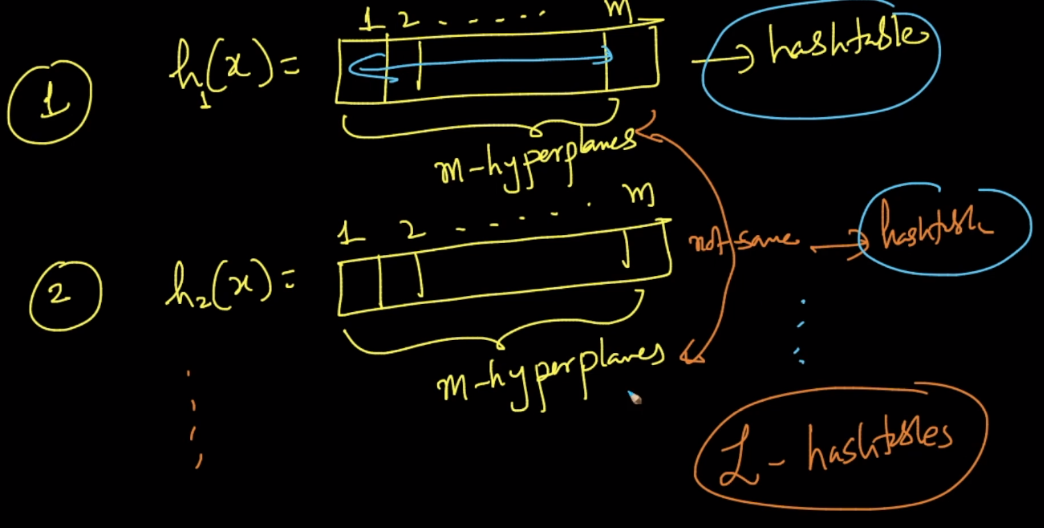
**So how to deal with such situation:**

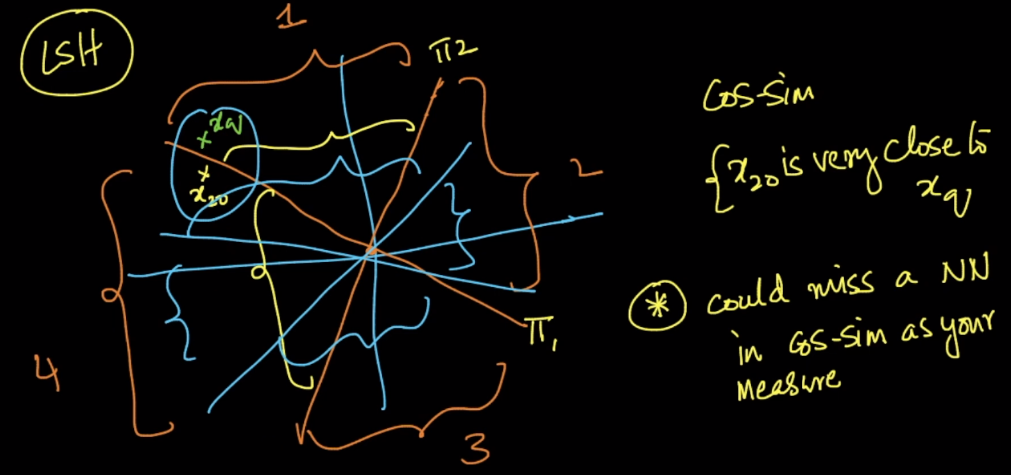
Suppose we are creating m hyperplanes then

First we generate m hyperplanes and create hash table for it.

Again we generate new m hyperplanes and create a new hash table for it.

And do it for let say **L** times, so finally we will have L hash tables.



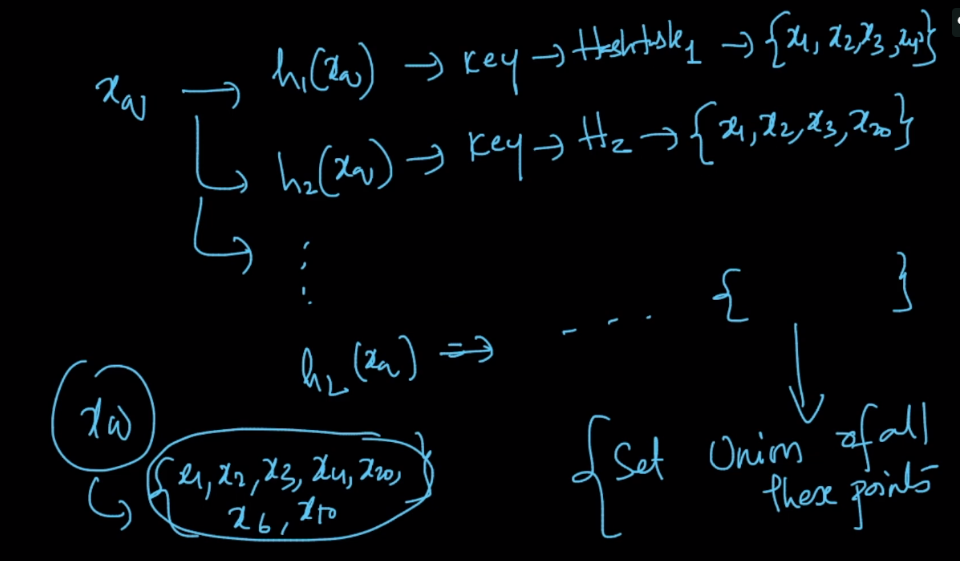


**How to querying a new point for L hash tables:**

If any new querying point arrives, we’ll fetch points from all the hash tables for according to the hash key vector generated for each hash table.

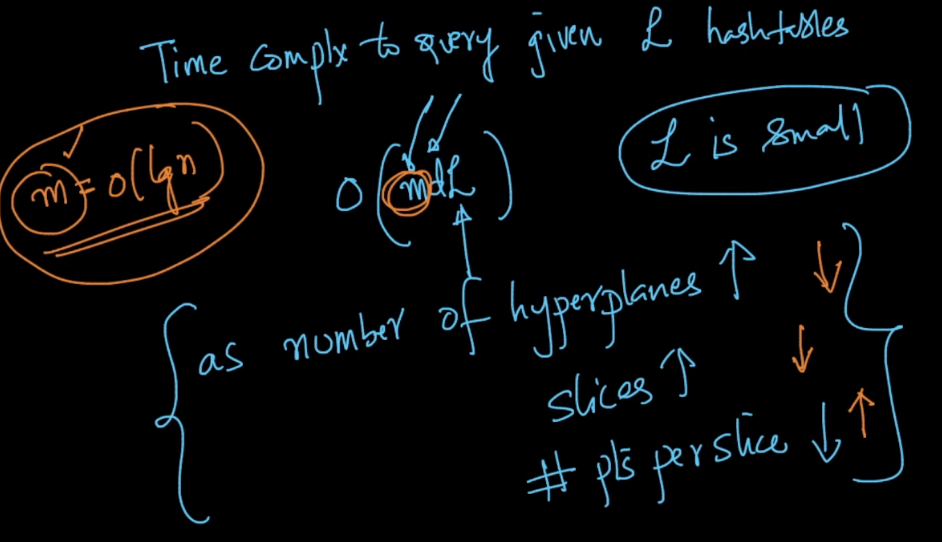
Then we take union of all the points retrieved from all the hash tables.

And find the neareset neighbors on this union of points.



**Time complexity for query in L hashtable:**

It will be O(m\*d\*l), but since L is usually small therefor it’s time complexity is also O(m\*d)



As no of hyperplanes increase, slices will be increases and # of pts per share decreases. And vice versa.