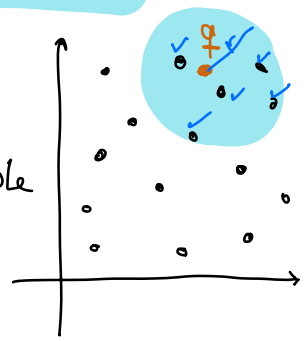
Similarity Search Queries

1. Rang queries

Given a query Point (vector) q

Find all the Points in the vicinity of it

Table T:
Each black dot represents a tuple (vector) in Table T



Formally

Find $\{t \in T \mid d(q,t) \leq r\}$

SQL Extension

```
SELECT *
FROM T
WHERE d(q) ≤ r
```

e.g., Pgvector Standard for l_2-norm

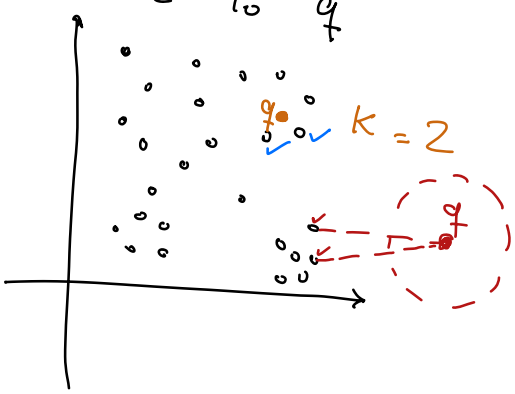
```
Select *
from T
where
Embedding <-> [3,5,8,-1] < 5
```

Column Name — d — q — r

2 - nearest Neigh bor queries

Given a query Point q, and a value k

Find the k nearest Points in the Table To q



Formally

Find $k\text{-argmin}_{t \in T} d(t,q)$

SQL Extension

```
SELECT *
From T
order by d(q)
Limit k
```

In, Pgvector, Euclidean (l_2-norm)

```
Select * from T
order by
    Embedding <-> [-1,3,-1,2]
Limit 10
```

↑ k

Efficient Processing of k-NN queries

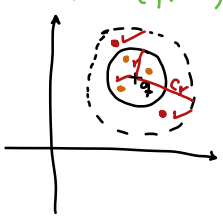
Query Answering

- **Exact:** Given a value k, Find the k-nearest neighbor of Q
  - $S = \underset{x \in T}{k\text{-argmin}}\, d(q, x)$
- **Approximate:** Returns a set S' which may not the exact KNN, but is "close" enough

measuring Approximation

1. by distance

$\forall x' \in S' \quad dist(q, x') \leq C\, dist(X, q)$



2. by recall

Recall@K: what Percentage of the returned Points belong to the KNN Set

$$\text{Recall@K} = \frac{|S \cap S'|}{K}$$

Question: How to Find KNN exactly?

Step 1:

Compute $d(q,x)$ for all points in the Table

$\Theta(n\,d)$ — d: Number of Columns; n: Number of rows

Step 2:

apply the extended version of the median Finding Alg. on the Scores to Find the top-K (K-NN)

$\Theta(n)$

$\Rightarrow$ Total time Complexity: $\Theta(nd)$ X

- This is very slow, particularly when n is large

$\Rightarrow$ Question: Can we Preprocess the database & Create Indices that enable Sublinear query answering at least approximately?

Indexing the Database for fast ANN query answering

- To Partition the Data
  - ↳ How?
- Build Some Data Structures on top of the Data

Data Structures
- Tables (Partition the Data)
- Trees
- Graphs
- Hybrid Data Structures

Local Sensitive Hashing (LSH)

A hash function H is in class of LSH, if

$\forall a, b$, $dist(a,b) \leq r$

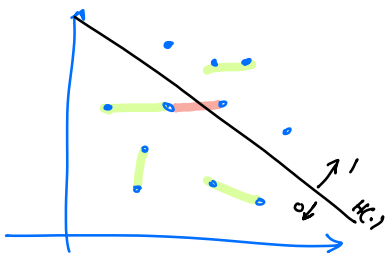
$P(H(a) = H(b)) \geq P_1$

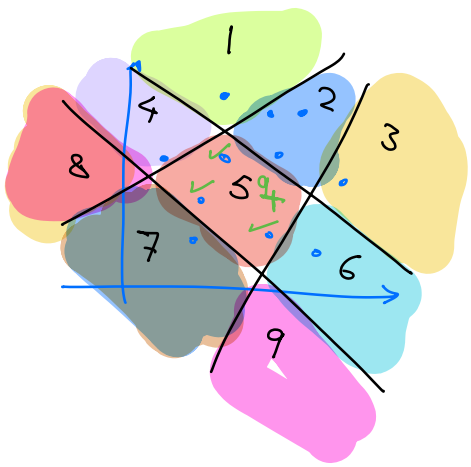
$\forall a, b$, $dist(a,b) \geq cr$

$P(H(a) = H(b)) < P_2$

$P_1 > P_2$

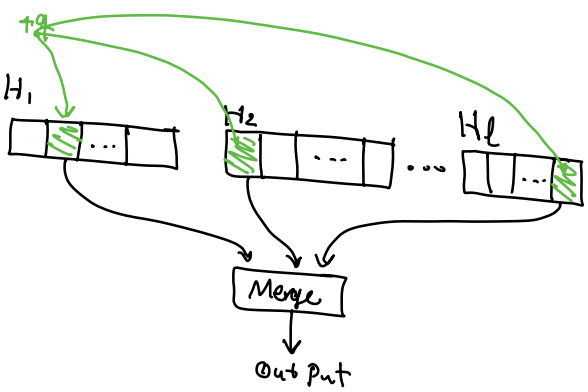
e.g.,





Using one Hash function, it is likely that we miss some of the NNs

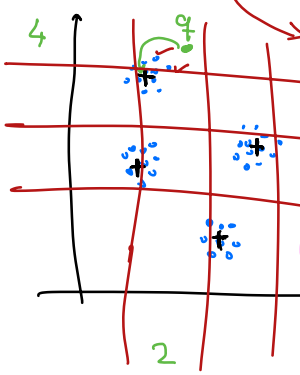
* Resolution: Use multiple Hash Functions



This is called LSH

* when Data Points are not evenly distributed in the Space (when many points are in close vicinities), LSH loses its power.
  → LSH does not Prune the Search Space in these Cases.

Resolutions
- Learned Hash Functions
- Quantization (K-means clustering)
  - Find the Centroid of Clusters
  - given a query Check wich Cluster Center is nearest to it
  - return the KNN from the Target Cluster

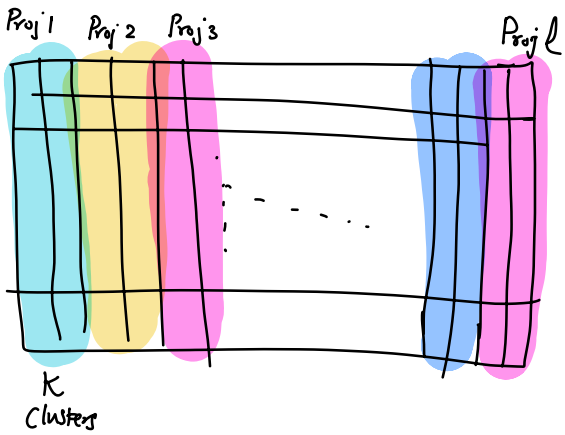


$q \in \langle 2, 4 \rangle$

4^2 clusters

* Issue 1: K Should in $O(n/\log n)$ for this approach to have $O(\log n)$ query time
* Issue 2: Curse of Dimensionality

Resolution:
Projection on lower Dimensions



$\Rightarrow$ No. clusters $= K^l$

$K^l = n/\log n$

$\Rightarrow l = \log_K n - \log_K \log n$

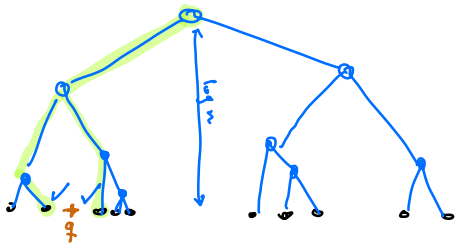
→ Given a query Point:

- Step 1: to find the Cluster Centers that q falls into

$q \in \langle c_1, c_2, \dots c_l \rangle$

c_i: is the closest cluster center to q, on projection i

- Step 2: Check the Points within the Same Cluster & return the top-k



To Construct the k-NN Tree

ConstructTree (S , i)

if $|S| \leq \tau$ ← Size Threshold

return S

Rule ← SelectRule (S, i)

↑ The rule for Partitioning S

Left ← $\{x \in S \mid Rule(x) = false\}$

Right ← $\{x \in S \mid \sim = True\}$

return (rule, ConstructTree(Left, $i-1$), ConstructTree(Right, $i-1$))

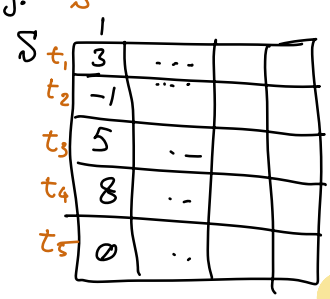
(A) Round Robin on Columns

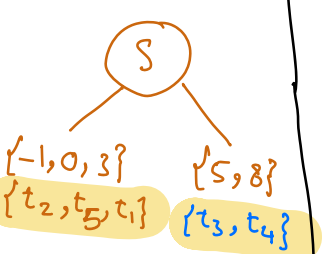
SelectRule$_{RR}$ (S, i)

j ← median of S based on Column (i mod d)

Rule ← ($x[i] \leq j$)

e.g. S





Observation: This is not a Binary Search Tree!

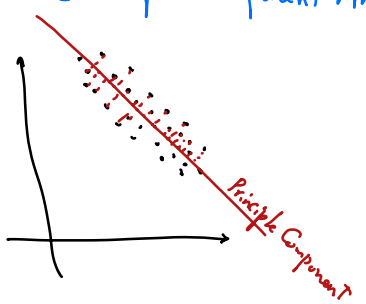
⇒ We Initially need to check both Left & Right Branches to an Leaves of

↳ This Can become Linear $\Theta(nD)$

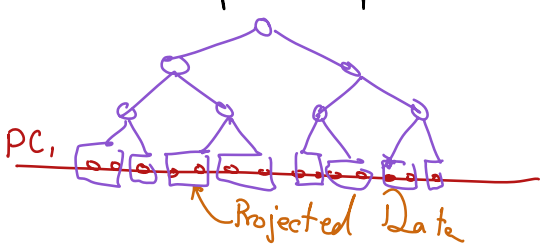
A Practical Resolution

- Use PCA (Principle Component Analysis)

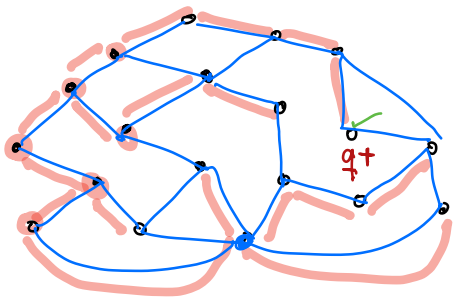
e.g.



- Instead of Creating the Tree based ind. Columns, Constructs the Tree on Projection (S) of Data on Principle Components



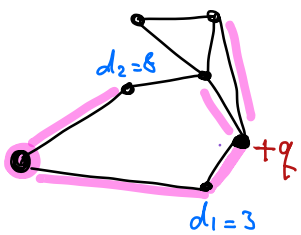
Graph Indices



Approach 1:

- Start from an arbitrary Point in the Graph
- follow a BFS Traversal & find the NN of q.

$\Theta(nd)$ X



Approach 2: Greedy Search (DFS)

- at each iteration, Next ← the neighbor with Smallest distance to q
- Continue the Search on Next
- Stop when all neighbors have Larger distances

Time Complexity of Greedy Search

$$\Theta(l \cdot deg(u))$$

l: length of the DFS path; $deg(u)$: degree of the nodes

Properties of the Graph

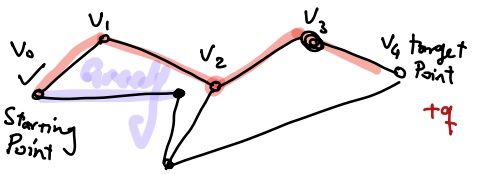
1- G Should be Sparse
deg(u) ≈ Small

2- l Should be Small.
after a Small number of hops, the alg. Stops

3- when the Greedy Stops a Close-to-NN Point is found

How about Connecting each Point to its K nearest neighbors ?

Monotonicity of the Search Path.



There should exist a monotically decreasing Path on $d(X, q)$ from the Starting Point to $NN(q)$

$\forall i \leq l$

$$d(v_i, q) > d(v_{i+1}, q)$$

* Monotonic Search Network

A Graph, where the Greedy Path Satisfies the monotonicity Property.

→ if d is Small ($d \leq 3$)

Such a graph with low degrees exist

↳ Use Voronoi Diagram

→ what if d is not Small?

Navigable Small-world Graph

↳ Should be Small

$\Theta(\log n)$

NSW

— In Practice

KNNG:

- Iteratively add the nodes to the Graph.
- for each node Connect it to its K-nearest neighbors

HNSW (Hierchical NSW)