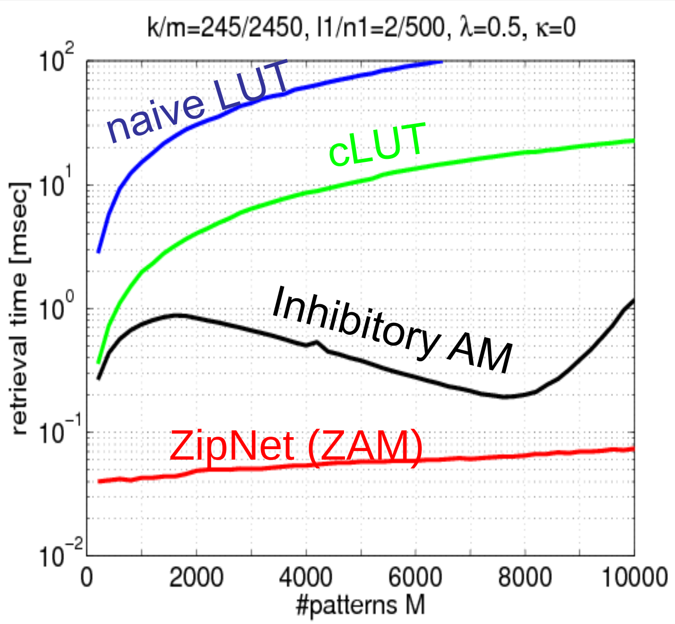
Efficient nearest neighbor search with KD trees and LSH

Andreas Knoblauch

HRI-EU Scientific Seminar, 21/03/2012

Applications:
- Similarity search in large databases
- prototype-based classification
- matching of local image features or image patches

Algorithms:
- look-up tables (naive)
- kd-trees (Bentley '90)
- multi-index hashing (Greene/Parnas/Yao '94)
- neural associative networks (Willshaw'69, Knoblauch '10)
- locality-sensitive-hashing (Andoni/Indyk '06)
- …
- ideal: search time O(1) independent of DB size N (as in brain)



Today: Look at kd-trees & LSH

(locality sensitive hashing
→ ist alternative läuft über Hashing ⇒ sehr schnell

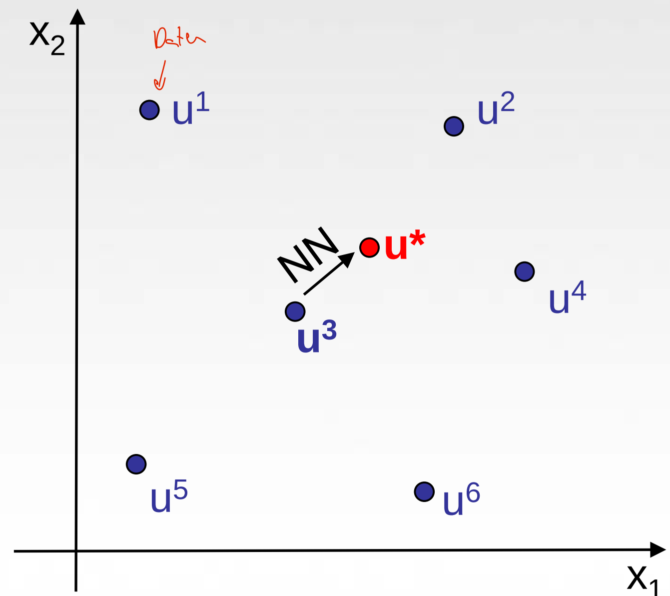
The nearest neighbor problem

Given a database of key vectors $\{u^\mu\}$ and a query u^* ,
return the nearest neighbor (NN) key vector $u^{\mu*}$
(e.g., w.r.t. Euklidian distance $d=d_2$)

2D Datenraum

Relevant problem parameters:

N : size of database (# stored key vectors) (Anzahl Datenvektoren)
D : dimensionality of key vectors
(K : # non-zero entries per key vector)



N=6, D=2, K=2 (non-sparse)

→ nearest neighbor zu u^* Datensatz finden → ist u^3
→ wie am schnellsten machen?

am wichtigsten!

Relevant complexity measures:

- t_q : query time (Anzahl Rechenschritte um NN zu finden)
- t_c : time to create index structure (wie lang geht es um KD-Tree aufzubauen)
- t_u : update time (insert / delete keys) zu updaten
- S : size of index structure und wieviel Speicher nötig

Datenbank linear durchgehen.
Suche den Datenvektor, der den kleinsten Abstand zu μ^* hat

Naive algorithm (linear scan of database):

→ linear Schritt DB durchlaufen und Abstand für jeden Datenvektor berechnen

```
μ* = 1 ;
For μ=2 TO N DO
    IF d(u*,uμ) < d(u*,uμ*) μ*= μ
RETURN μ*
```

← Distanz zw. u^* und u^μ

neuer NN definieren der besser ist

Index des finalen NN

$t_q = O(ND)$	(or O(NK) for sparse data vectors)
$t_c = O(ND)$	(or O(NK))
$t_u = O(D)$	(or O(K))
$S = O(ND)$	(or O(NK))

→ prohibitive for large N,D

Rückgabeparameter

KD-trees (1): Creation

Binärer Baum jeder Knoten hat max. 2 Zweige!

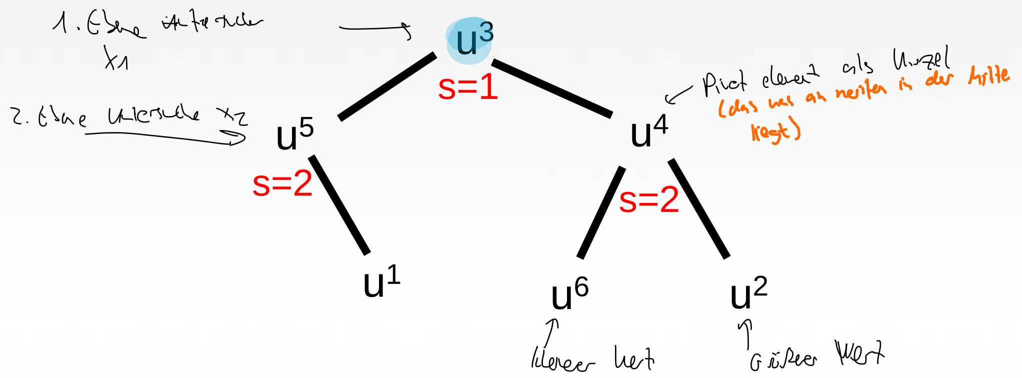
Wurzel Knoten — Pointer auf linken und rechten Baum — Split-Direction — baut KD-Tree auf — Menge von Daten (Matrix)

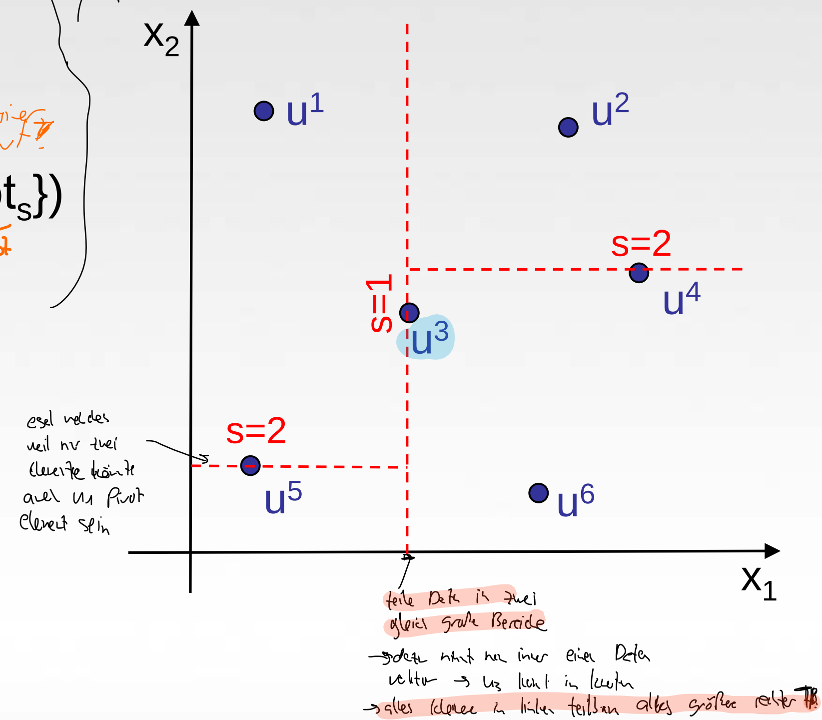
An jeder Stelle des Baumes wird 1 D. angesehen (1 Attribut wie z.B. x_1) → dann sucht man schnellstmöglich, sodass Daten gleichmäßig aufgeteilt werden

function (root,left,right,s) = Create-KD-Tree(DB) // returns kd-tree for database DB:=$\{u^\mu\}$

1) IF isEmpty(DB) THEN RETURN 0 (Rekursions-Ende)
2) Select pivot vector root ∈ DB
   and split dimension s ∈ {1,…,D} (suche split Direction und wähle Start vektor)
3) left := Create-KD-Tree($\{u \in DB : u_s < root_s\}$) — kleiner als root — rekursiver Aufruf?
4) right := Create-KD-Tree($\{u \in DB\backslash\{root\}: u_s \geq root_s\}$) — größer als Root
5) RETURN (root,left,right,s)

rekursiv





- KD-tree defines partitioning of key space into **rectangular regions**
- **various variants** to implement step 2 (choose root, s)

Ziel:
  - try to get balanced trees → short paths → dann halbiert sich in jedem Schritt übrige Daten → beantwortet ist dann Log(x)
  - try to get square regions (and avoid elongated regions)
    → maximize „cut-off opportunities“ (see retrieval…)

ähnlich wie bei Dichteschätzung! Raum wird in Rechtecke eingeteilt!

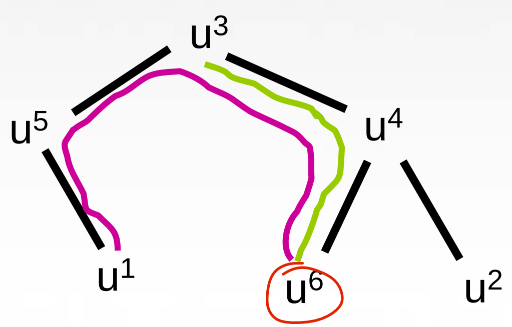
KD-trees (2): Retrieval

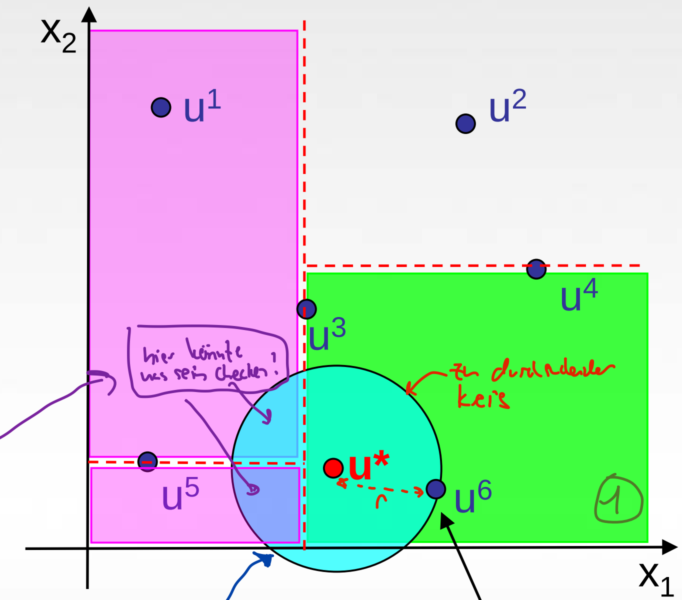
⇒ Ziel: Finde zu u* den NN aus dem KD-Tree

KD-Tree ↓ Anfragevektor ↓

Function u^{NN} = Query-KD-Tree(KDT,u*) // Find NN of query u* in KD-Tree

1) Find first approximation u^{NN1} in leaf region of u* (suche Approx. für NN)
   ↳ suche von der Partielle in der u* liegt
   ↳ ist vektor im linken oder rechten teilbaum? (kleiner oder größer als pivot?)
2) Do backtracking to investigate keys in regions that overlap with sphere $\{x: d(u^*,x) \le d(u^*,u^{NN'})\}$
3) Return true nearest neighbor u^{NN}





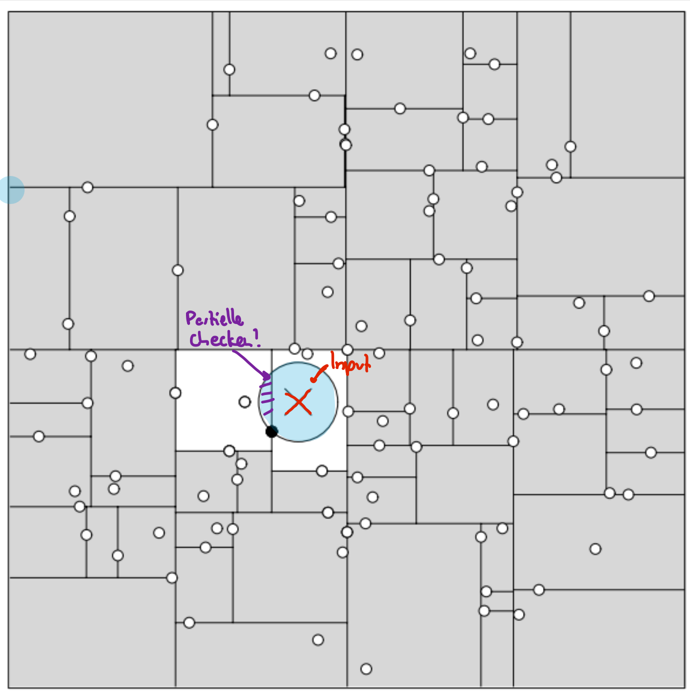
1. Kandidat für NN → $=u^{NN1}$ $=u^{NN}$

Problem:
→ evtl. noch ein NN in nebenliegender Partielle
Alls noch einer existiert muss der im Kreis sich befinden
⇒ Prüfe diese Nachbarregionen

→ Je mehr Nachbarregionen desto schlechter!

→ Funktioniert falls 1. NN sehr nahe am inputvektor ist

KD-trees (3): Performance & Limitations



KD-tree algorithm works fine if the first approximative NN is close enough to the query to cut-off rest of the tree

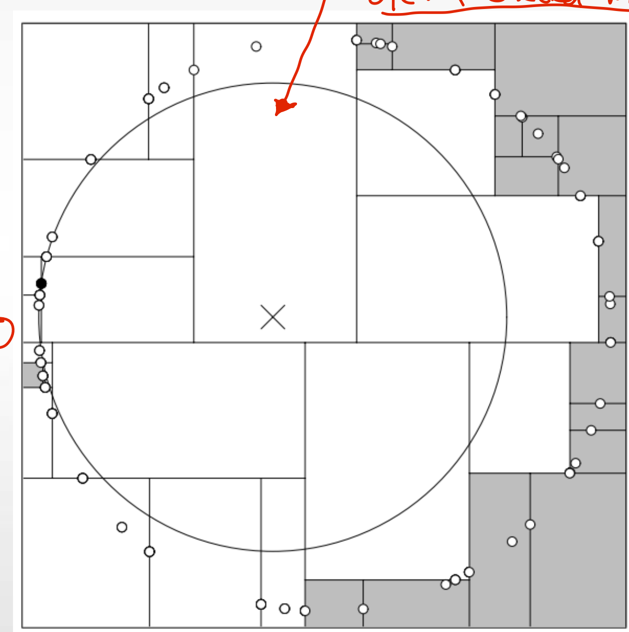
→ je mehr Regionen der Kreis schneidet desto schlechter!

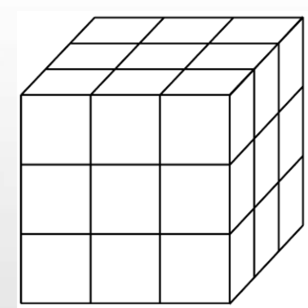
→ path to leaf ~ log N steps (for balanced trees)

→ constant steps for backtracking (for hypercubic regions)

→ query time t_q ~ log N (for fixed D)

schneidet viele Partiellen die man checken muss! → hohe retrival-time





Potential problems:

- unbalanced trees → dann muss man alles durchlaufen
- queries not drawn from data distribution — alle liegen außen, ich muss in der Mitte... schlecht!
- non-hyercubic regions (länglich statt quadrat → Kreis schneidet oft!)
- sparse data vectors
- curse of dimensionality

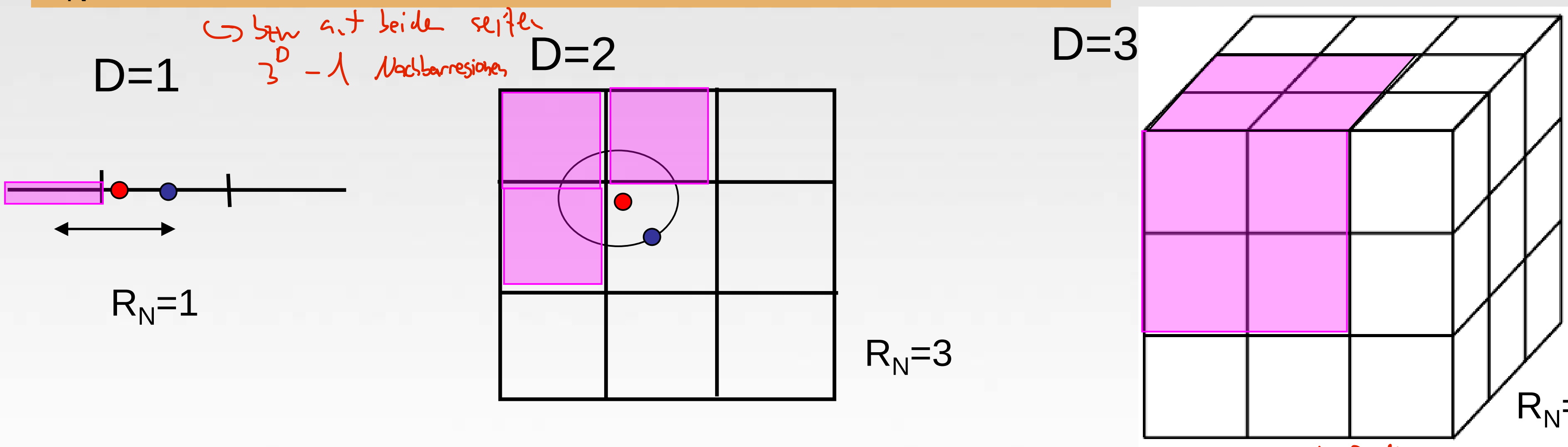
↳ wird bei viel D sehr kompliziert

→ extrem viele Nachbarregionen

KD-trees (4): Curse of dimensionality

- # backtracking steps depends on R_N := # neighboring regions
- $R_N = 2^D - 1$ increases exponentially with dimension D

↪ bzw. gilt beide seiten $3^D - 1$ Nachbarregionen



Scheiße → schlechte Laufzeit

Curse of dimensionality: query time $\mathbf{t_q \sim \log N + 2^D}$ exponential in D

→ KD-trees are useful only for low-dimensional data, e.g., 1<D<20 bzw. bis 50 noch okay

→ other tree-based partitioning algorithms (KDB, R, SR, SS trees) suffer from similar problems

→ there are no known efficient NN algorithms for large D

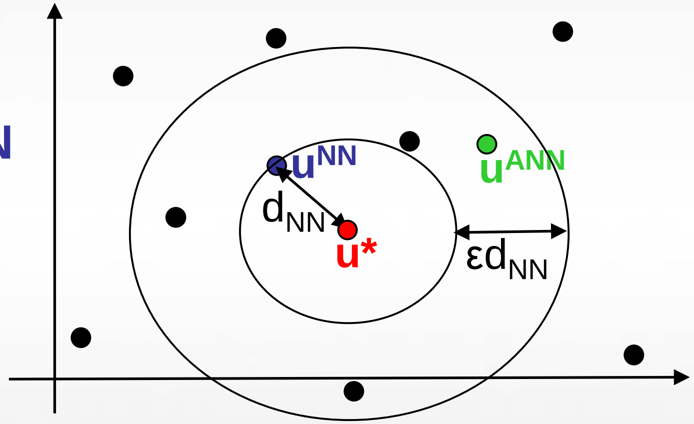
→ next best solution: Approximate NN algorithms… (sind Ähnliche Algorithmen…) (liefern nur Näherung) nicht mehr relevant...

Approximate Nearest Neighbor (ANN)

Reasoning: Choosing a particular metrics is a heuristics anyway
→Therefore, for many applications, **solving ANN may be sufficient**

Many different methods for ANN problem:
- KD (or other) trees without backtracking
- K-means NN (for clustered data)
- Neural associative networks
- Locality sensitive hashing (LSH) (ähnlich effizient)

Related problem with bounded error: ε-NN
An algorithm solving ε-NN returns an
ANN that is at most factor 1+ε worse than **true NN**



Locality-sensitive-hashing

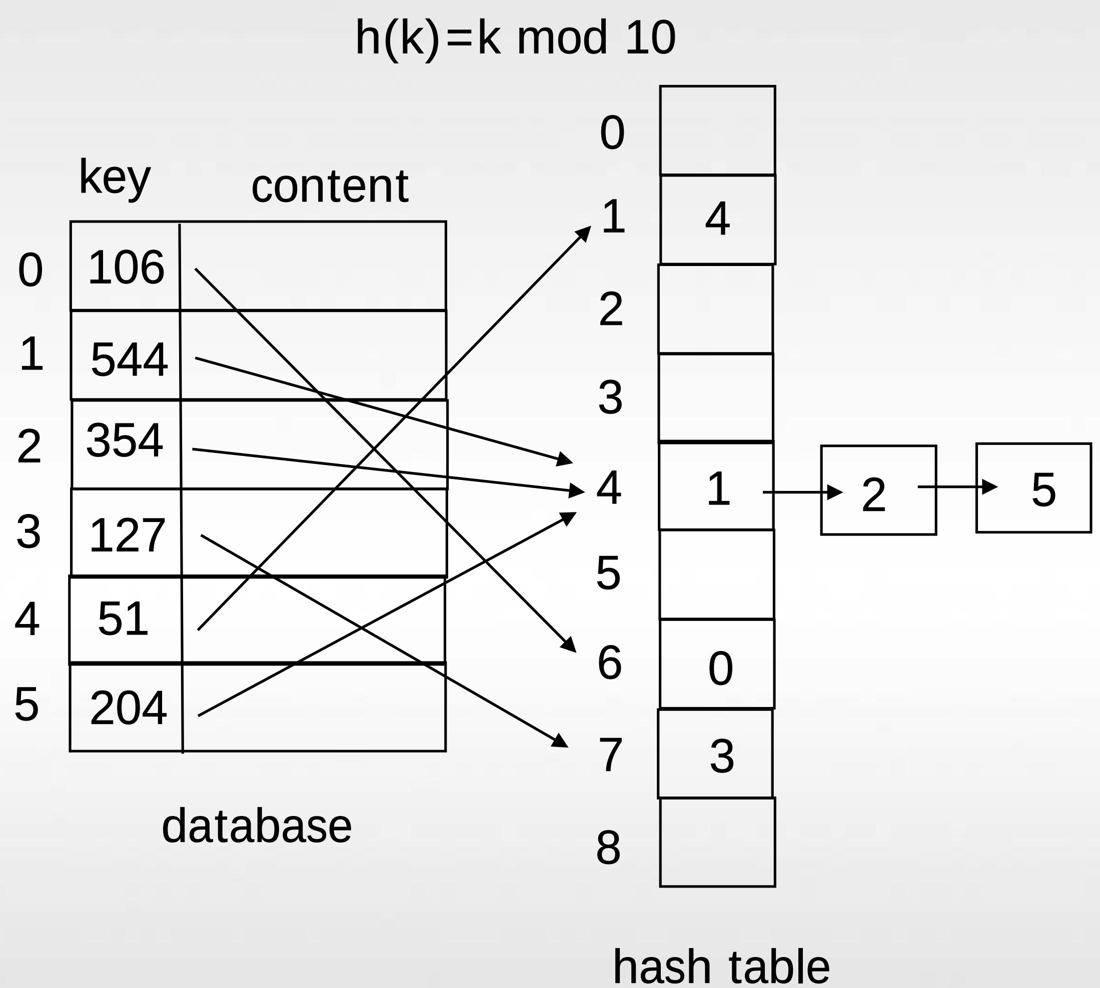
What is actually hashing?

Given: large database, key k
Goal : find content associated with key k

Linear search: O(n)
Hashing with h(k): O(1)

Desired poperties of h(k):
- compute h(k) in O(1)
- distribute keys uniformly

→ Not locality-sensitive!



Locality-Sensitive Hashing (LSH) Algorithm (1)

(i) <u>Build-up of index structure</u>

Input: Key vectors $\{u_1,u_2,\ldots,u_N)$ and l:=#hash tables
Output: Hash tables $T_1,\ldots,T_l$

Algorithm:
1) for each i=1..l
   Generate random hash function g_i:$u_j \rightarrow$bucket index
2) for each i=1..l
   for each j=1..N
   store key vector u_j in bucket $g_i(u)$ of hash table T_i

For example: g_i are chosen randomly from set
$H=\{h_i:u \rightarrow u_i \mid i=1..D\}$ (projections onto i-th component)

Locality-Sensitive Hashing (LSH) Algorithm (2)

(ii) Approximate Nearest-Neighbor Query

Input: Query vector u* and K:=# nearest neighbors
Output: K (or less) NN vectors

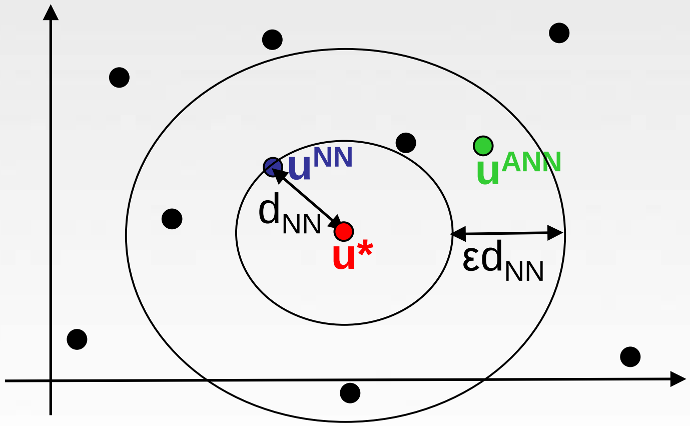
Algorithm:
1) S = { }
2) for each i=1..l
   S = S + {vectors in bucket $g_i(u^*)$ of table T_i
3) Return K nearest neighbors of u* found in set S
   (e.g., by using linear search)

Note: S is typically much(!) smaller than DB size

Locality-Sensitive Hashing - Evaluation

One can show (Andoni/Indyk '06) for appropriate choices of H (family of hash functions) and l (# of hash tables):

- LSH solves ε-NN problem
- LSH is efficient:
  query time $t_q \sim DN^{1/(1+\varepsilon)2}$
  space $\sim n^{1+(1+\varepsilon)2}$



For example, for ε=1:
$t_q \sim DN^{0.25}$

→ Efficient solution for ANN problem in high-dim spaces
→ recent variants
- Coherency Sensitive Hashing (Korman/Avidan, 2011)
- Complementary Hashing (Xu et al, 2011)
- …

Thank you for your attention!