Module 3: Hashing

Lecture 11: Locality Sensitive Hashing and Grid File

The Lecture Contains:

- Locality sensitive hashing (LSH)
  - Example: Hamming space
  - kNN query
- Grid file
  - Grid directory
  - Insertion
  - Deletion
  - Searching

Locality sensitive hashing (LSH)

- To support range and kNN queries
- Idea of randomized algorithms

Locality sensitive hashing (LSH)

- To support range and kNN queries
- Idea of randomized algorithms
  - Algorithms that make random choices
  - Monte Carlo: Probabilistic but bounded error results in bounded time accuracy improves with each run
  - Las Vegas: Correct and deterministic result but varying time

Locality sensitive hashing (LSH)

- To support range and kNN queries
- Idea of randomized algorithms
  - Algorithms that make random choices
  - Monte Carlo: Probabilistic but bounded error results in bounded time accuracy improves with each run
  - Las Vegas: Correct and deterministic result but varying time
- Goal in LSH: To find a hashing function that is approximately distance-preserving (within some tolerance)
- A hash function h is (r_1, r_2, p_1, p_2)-sensitive if for any x, y,
  - If $d(x,y) \leq r_1$, then $P[h(x) = h(y)] \geq p_1$
  - If $d(x,y) \geq r_2$, then $P[h(x) = h(y)] \leq p_2$

Locality sensitive hashing (LSH)

- To support range and kNN queries
- Idea of randomized algorithms
  - Algorithms that make random choices
  - Monte Carlo: Probabilistic but bounded error results in bounded time accuracy improves with each run
  - Las Vegas: Correct and deterministic result but varying time
- Goal in LSH: To find a hashing function that is approximately distance-preserving (within some tolerance)
- A hash function h is (r_1, r_2, p_1, p_2)-sensitive if for any x, y,
  - If $d(x,y) \leq r_1$, then $P[h(x) = h(y)] \geq p_1$
  - If $d(x,y) \geq r_2$, then $P[h(x) = h(y)] \leq p_2$
- Can also be viewed as a dimensionality reduction technique

Example: Hamming space

ID O_i	Data x, y	f x', y'	Hamming space $h_1(x, y)$	Projection bits h_2 2,9,13	7,10,14	1,5,11	8,12,14
O_1	35,42	2,3	11000001110000	110	010	100	100
O_2	52,10	4,0	11110000000000	100	000	100	000
O_3	62,77	4,6	11110001111110	111	010	101	110
O_4	82,65	6,5	11111101111100	110	010	111	110
O_5	5,45	0,3	00000001110000	010	010	000	100
O_6	27,35	2,2	11000001100000	110	000	100	100
O_7	85,15	6,1	11111101000000	100	000	110	100
O_8	90, 5	7,0	11111110000000	100	100	110	000
Q	5,55	0,4	00000001111000	010	010	001	100

$$x' = x/12.5 \qquad y' = y/12.5 \qquad h_1(x, y) = unary(x').unary(y')$$

kNN query

- 1-NN query of Q has true answer O_5
- Applying 1-NN on l projections gives l collision sets:
  - 2,9,13: O_5
  - 7,10,14: O_1, O_3, O_4, O_5
  - 1,5,11: Φ
  - 8,12,14: O_1, O_5, O_6, O_7
- Now majority counting gives O_5

kNN query

- 1-NN query of Q has true answer O_5
- Applying 1-NN on *l* projections gives *l* collision sets:
  - 2,9,13: O_5
  - 7,10,14: O_1, O_3, O_4, O_5
  - 1,5,11: Φ
  - 8,12,14: O_1, O_5, O_6, O_7
- Now majority counting gives O_5
- No guarantee of correct answer
- Accuracy increases with more runs
- It can be proved that $h = h_2 \circ h_1$ is approximately distance preserving provided h_2 is so (h_1 already is so!)
- Therefore, this is a Monte Carlo algorithm
- Time: $O(dl)$ where d is the dimensionality (here, 3)

kNN query

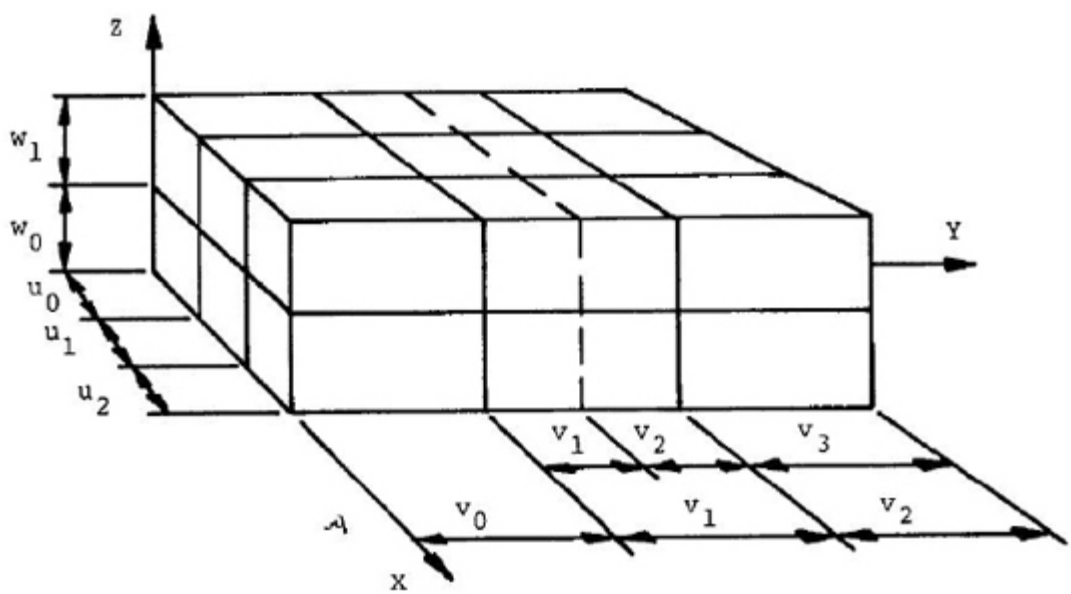
- 1-NN query of Q has true answer O_5
- Applying 1-NN on l projections gives l collision sets:
  - 2,9,13: O_5
  - 7,10,14: O_1, O_3, O_4, O_5
  - 1,5,11: Φ
  - 8,12,14: O_1, O_5, O_6, O_7
- Now majority counting gives O_5
- No guarantee of correct answer
- Accuracy increases with more runs
- It can be proved that $h = h_2 \circ h_1$ is approximately distance preserving provided h_2 is so (h_1 already is so!)
- Therefore, this is a Monte Carlo algorithm
- Time: $O(dl)$ where d is the dimensionality (here, 3)
- Extensions for Euclidean space: E2LSH

Grid file

- Multi-dimensional hashing
- Goal: To retrieve records with at most two disk accesses
- Two parts:
  - Grid directory
  - Buckets (or pages)
- Buckets hold data records
- Grid directory maintains grid cells that points to buckets
- Two parts:
  - d-dimensional array: One entry for each grid cell
  - d 1-dimensional linear scales: Defines partitions for each dimension
- Arrangement of records inside a bucket can be anything

Grid directory

- Capacity of buckets is large
- Hence, size of directory is much less
- Still, array may be disk-resident for high dimensions
- Linear scales reside in memory
- For p partitions in d dimensions each, size of array is p^d and size of scales is $p.d$
- More than one grid cell may point to same bucket



Insertion

- (35,42), (52,10), (62,77), (82,65), (5,45), (27,35), (85,15), (90,5)
- Bucket splitting heuristics
    - Middle of range - buckets of equal size
    - Round-robin through dimensions

Insertion

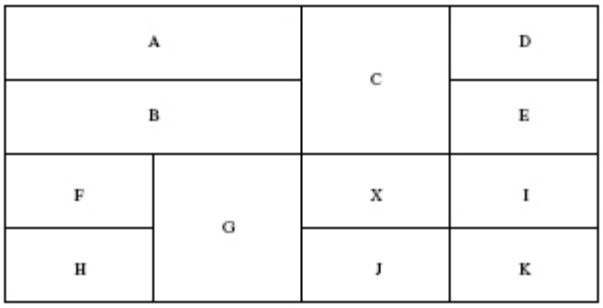
- (35,42), (52,10), (62,77), (82,65), (5,45), (27,35), (85,15), (90,5)
- Bucket splitting heuristics
  - Middle of range - buckets of equal size
  - Round-robin through dimensions
- Bulk loading: Loading all points together instead of one by one may help in choosing optimal partitions

Deletion

- Deletion of records may require bucket merging

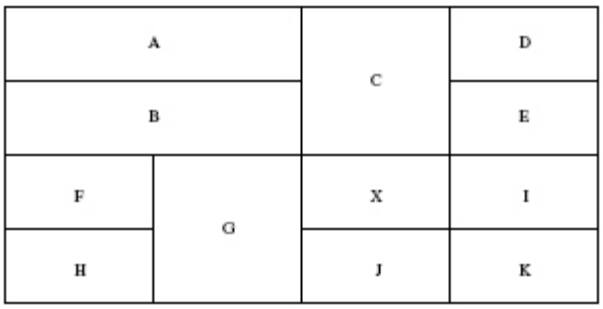
Deletion

- Deletion of records may require bucket merging
- Buddy system
  - Choice of neighbour in each dimension
  - Merges with "buddy": one that was created simultaneously by partitioning some old bigger cell
  - X chooses I (J is a potential buddy)



Deletion

- Deletion of records may require bucket merging
- Buddy system
  - Choice of neighbour in each dimension
  - Merges with "buddy": one that was created simultaneously by partitioning some old bigger cell
  - X chooses I (J is a potential buddy)
- Neighbour system
  - Choice of two adjacent neighbours in each dimension
  - Chooses any one maintaining the "convexity" property
  - X may choose either C, I or J (but not G)



Searching

- Point queries take at most 2 disk accesses
- Range queries are much more expensive
    - Lower dimensional slice cuts through all buckets in remaining dimensions
    - In each dimension, "fringe buckets" are also accessed