

Multi-Probe LSH: Efficient Indexing for High-Dimensional Similarity Search

UNIVERSITY OF
WATERLOO



DAVID R. CHERITON SCHOOL
OF COMPUTER SCIENCE

Helia Yazdanyar - CS 848

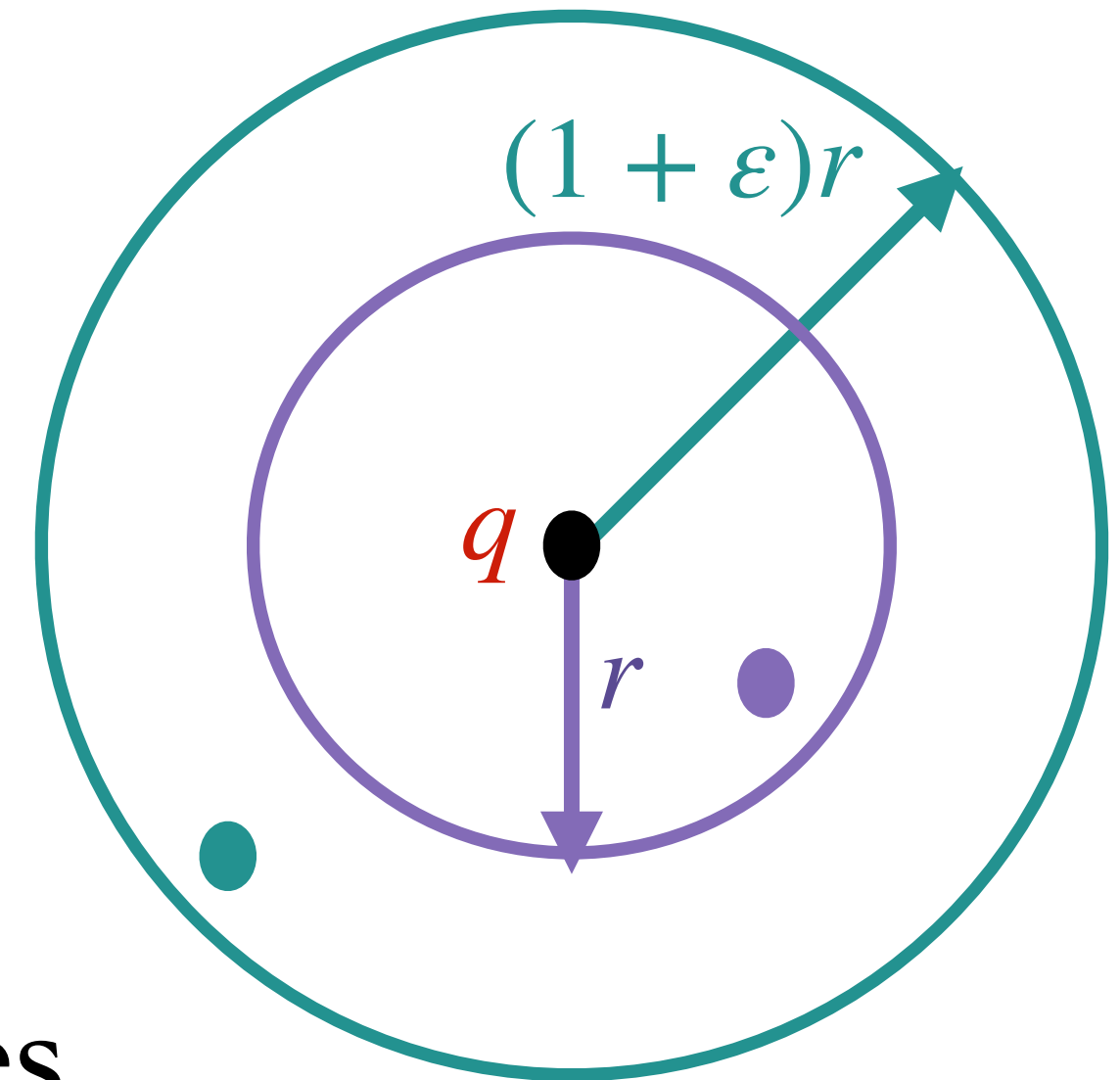
Goal: Similarity search in high-dimensional spaces

Ideal Indexing for Similarity Search

- **Accurate:** returns results that are very close to brute-force,
- **Time efficient:** takes $O(1)$ or $O(\log N)$ time where N is the number of data objects in the dataset.
- **Space efficient:** small space usage for index; may fit into main memory even for large datasets.
- **High-dimensional:** works well for datasets with very high dimensionalities

Locality-Sensitive Hash Functions

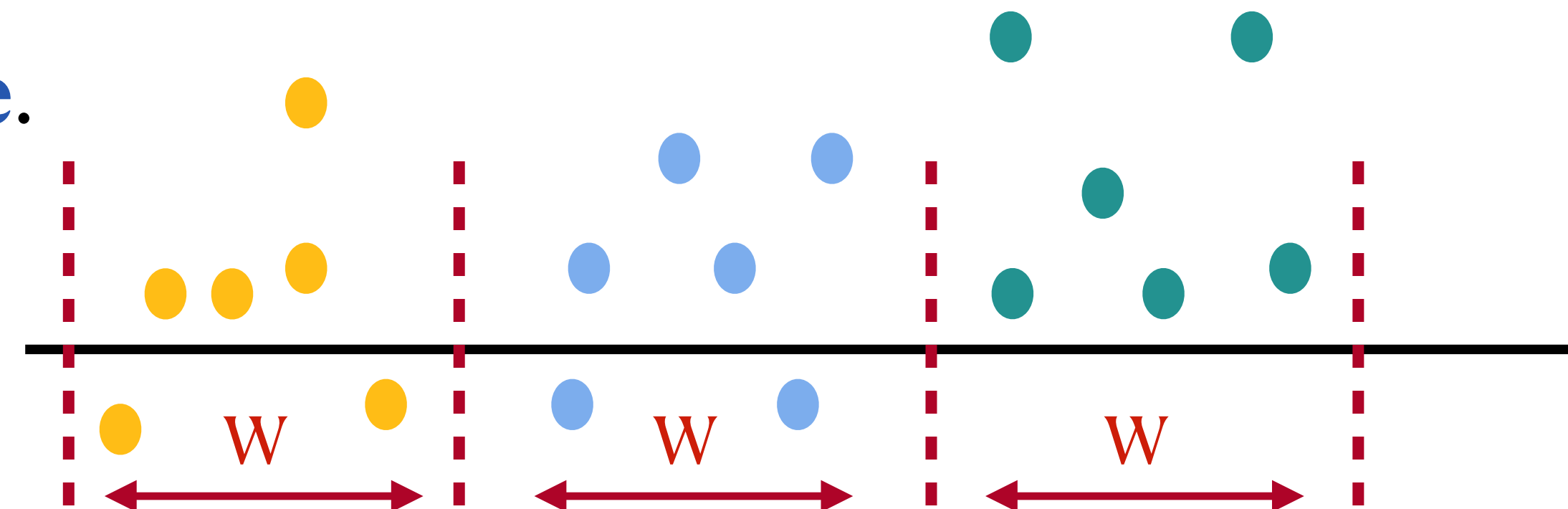
- **Idea:** Nearby objects should collide more often than distant objects.
- A hash family H is $(r, (1 + \epsilon)r, p_1, p_2)$ -sensitive if:
 - $D(q, p) \leq r$: collision probability is at least p_1 .
 - $D(q, p) > (1 + \epsilon)r$: collision probability is at most p_2 .
- r defines “near,” and $(1 + \epsilon)r$ defines “far”.
- We require $p_1 > p_2$ to distinguish nearby objects from distant ones.



Hash Function Used in This Paper

$$h_{a,b}(v) = \left\lfloor \frac{a \cdot v + b}{W} \right\rfloor$$

- $(v \in \mathbb{R}^d)$: the object's feature vector.
- (a) : a random vector with independent Gaussian entries.
- (b) : a random offset sampled uniformly from $[0, W]$.
- (W) : the width of each interval on the projected line.
- $h_{a,b}(v)$ projects the vector onto a line, shift it, and assign it an integer interval number.
- Gaussian projections support **Euclidean distance**.



Basic LSH

- **Each table uses a composite hash key:**

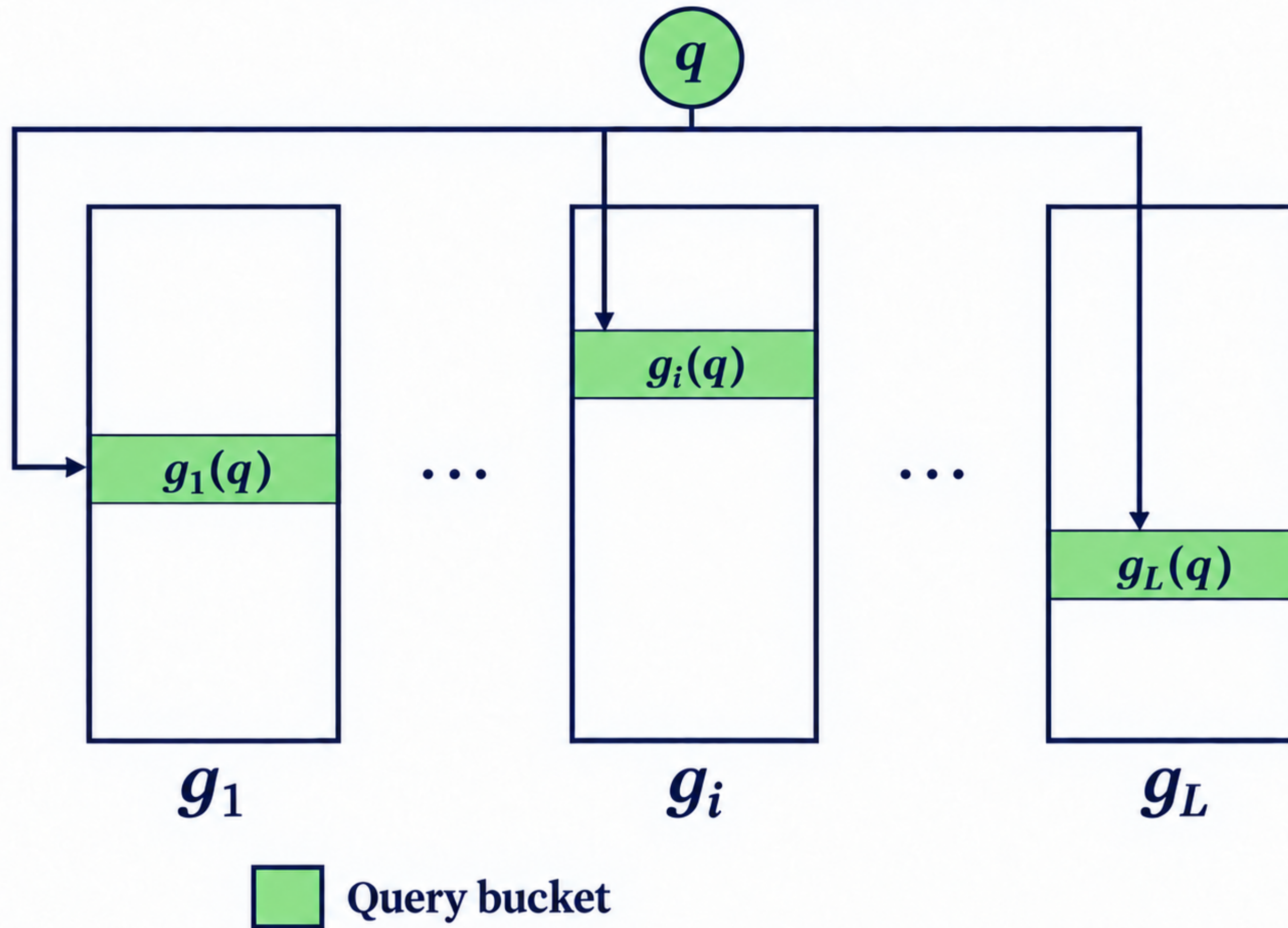
$$g(v) = (h_1(v), h_2(v), \dots, h_M(v))$$

- **(M) : hash functions per table.** An object must match all **M** values to share a bucket.
- **(L) : number of hash tables.** More tables increase the chance of finding a true neighbor.
- **(K) : neighbors returned.** Candidates are ranked to select the closest **K**.

Main drawback

- requires large number of tables.
- Index entries grow proportionally to $N \times L$, where **N** is the dataset size.
- An index that exceeds main memory may incur expensive disk accesses.

Basic LSH



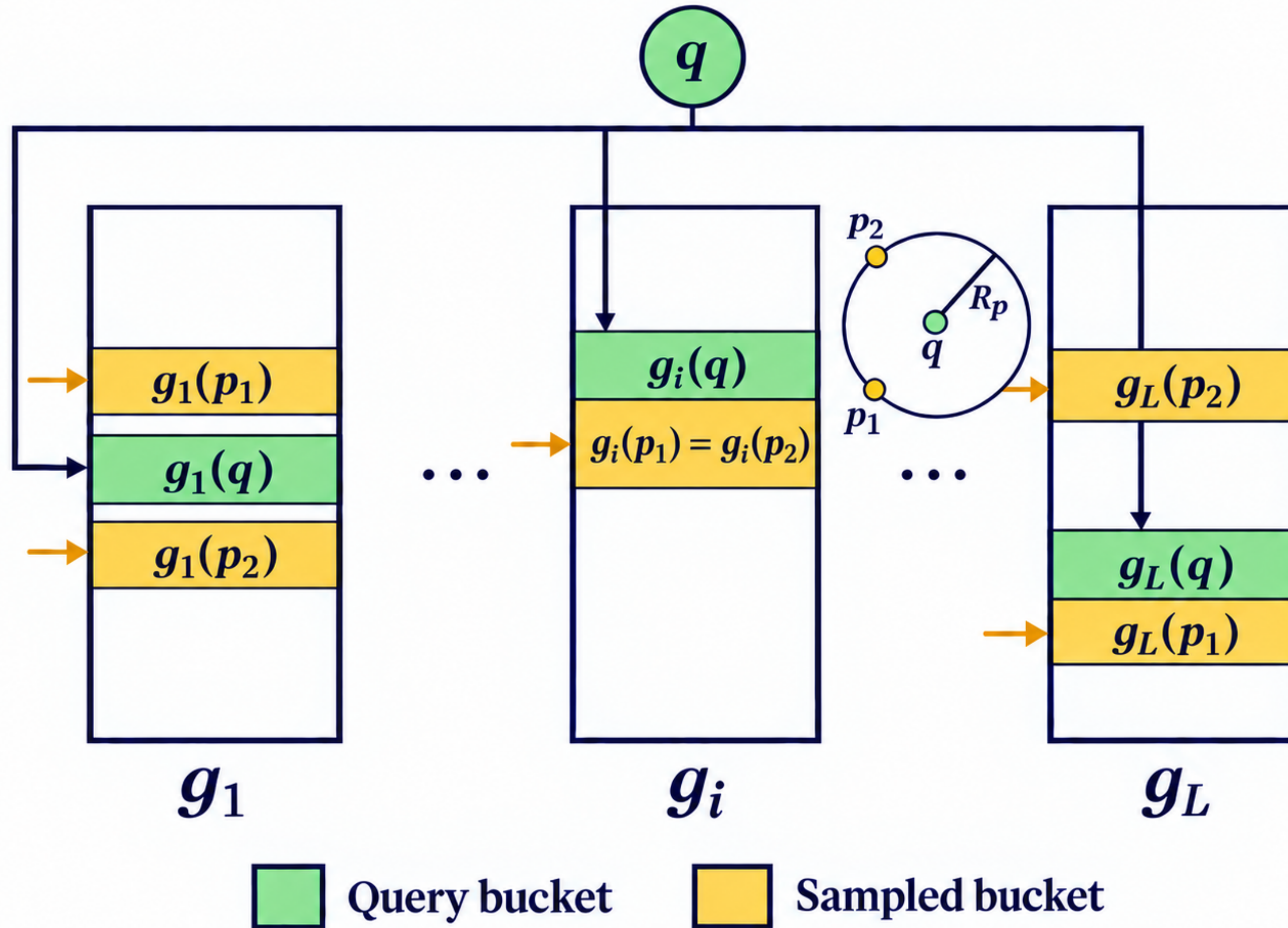
Entropy-based LSH

- **Goal:** Reduce the number of tables by checking more buckets within each table.
- Use the same index structure as basic LSH, but change the query procedure.
- Assume the nearest neighbor of query q is at distance R_p , and this value **is known**.
- Repeatedly generate a random point p' at distance R_p from q , then probe its hash bucket.
- These buckets are likely to contain a nearby object.

Limitations

- Generating and hashing perturbed points adds computation.
- Sampling can revisit the same buckets and waste probes.
- Choosing R_p is difficult

Entropy-Based LSH



Why Multi-Probe LSH?

- **Basic LSH:** More tables improve recall, but increase memory use.
- **Entropy-based LSH:** Extra probes save space, but random sampling repeats buckets.
- **Idea:** Probe promising buckets systematically, so fewer tables can provide good coverage.

Multi-Probe LSH

- A neighbor may miss the query's exact bucket but land in a **nearby** hash bucket.
- Check the original bucket, then selected alternative “close by” buckets.
- Keep the same index structure; change the query procedure.

Key question: Which buckets should we probe first?

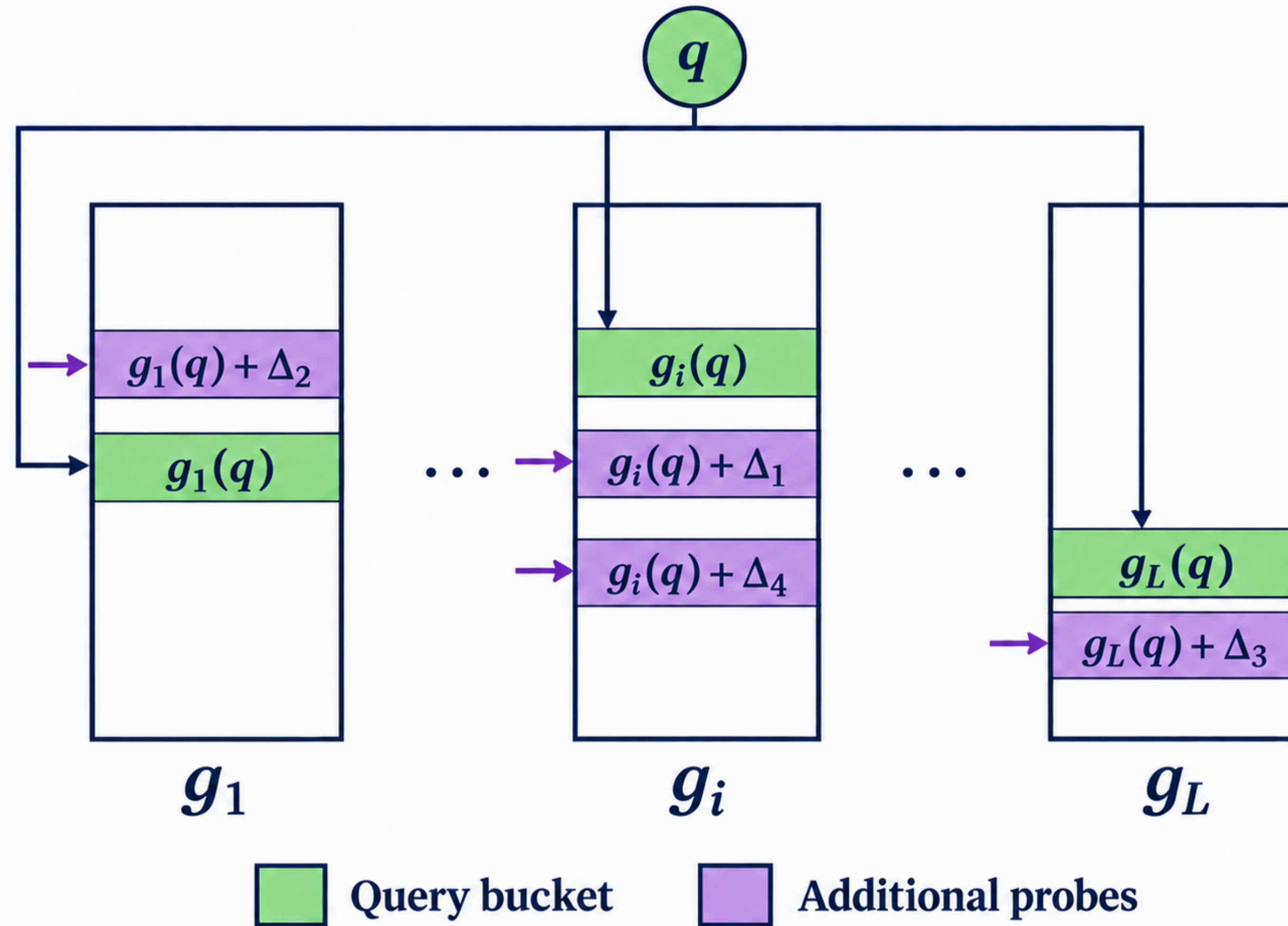
Multi-Probe LSH

- Define a hash perturbation vector:

$$\Delta = (\delta_1, \dots, \delta_M)$$

- For a query q with bucket key $g(q) = (h_1(q), \dots, h_M(q))$, we will probe the bucket $g(q) + \Delta$
- With a sufficiently large W , neighbors usually differ by **at most one** in each hash coordinate.
- So we only need to considers $\delta_i \in \{-1, 0, +1\}$.
- **Example:** $g(q) = (4, 2, 7)$, $\Delta = (-1, 0, +1)$, probed bucket = $(3, 2, 8)$.

Multi-Probe LSH



Probing sequence $(\Delta_1, \Delta_2, \Delta_3, \dots)$

How to generate the sequence of
perturbation vectors?

Strategy 1: Step-Wise Probing

Strategy 1: Step-Wise Probing

- First probe buckets that differ in **one coordinate**.
- Then probe those differing in **two coordinates**, and **so on**.
- Treat all directions and coordinates equally within each step.
- For an LSH index with **L** hash tables and **M** hash functions, the total number of

n-step buckets is:

$$L \times \binom{M}{n} \times 2^n$$

Step-Wise Probing

Original bucket:

$$g(q) = (3, 2, 5, 4, 1)$$

1-step: 10 buckets:

$(\text{orange } 2, 2, 5, 4, 1)$... $(\text{orange } 4, 2, 5, 4, 1)$... $(3, 2, 5, 4, \text{orange } 2)$

2-step: 40 buckets:

$(\text{orange } 2, \text{green } 1, 5, 4, 1)$... $(\text{orange } 4, 2, \text{green } 6, 4, 1)$... $(3, 2, 5, \text{green } 3, \text{orange } 2)$

3-step: 80 buckets:

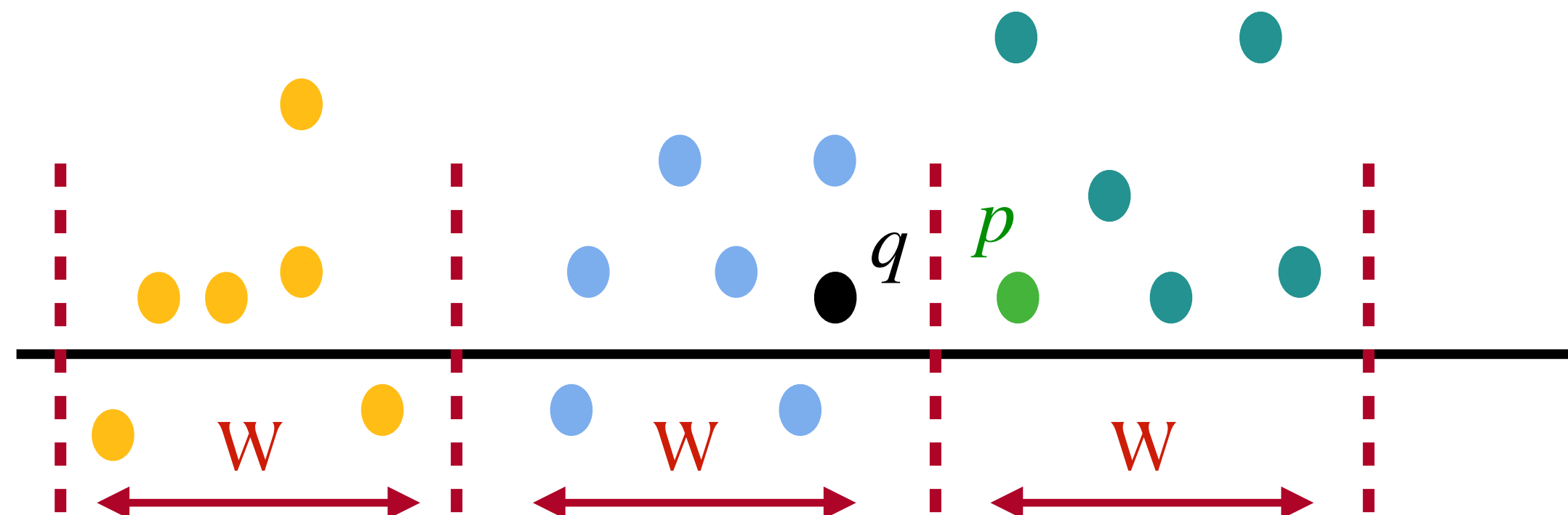
$(\text{orange } 2, \text{green } 1, \text{purple } 6, 4, 1)$... $(\text{orange } 4, 2, \text{green } 6, \text{purple } 3, 1)$... $(3, \text{purple } 3, 5, \text{green } 3, \text{orange } 2)$

Orange: first change **Green:** second change **Purple:** third change

Probe all 1-step buckets before 2-step buckets, then 3-step buckets.

Distance to a Bucket Boundary

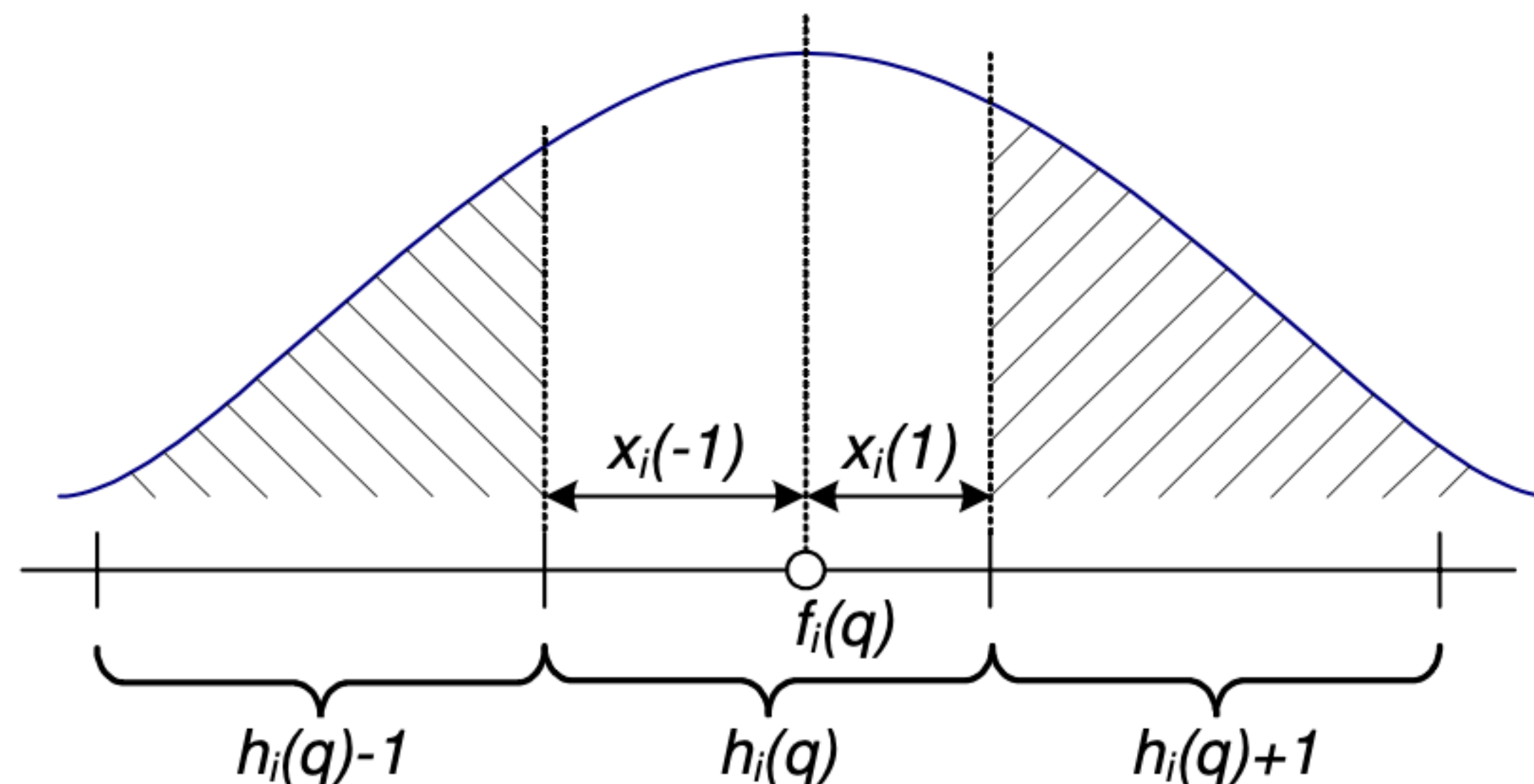
- In the **step-wise** probing strategy, all coordinates in the hash values of the query **q** are treated the **same**
- A point **p** close to **q** is likely to fall in either the same slot as **q** or an adjacent slot
- The probability that **p** falls into the slot to the right of **q** depends on how close **q** is to the right boundary



Distance to a Bucket Boundary

- Let $f_i(q) = a_i \cdot q + b_i$ be the projection of query q on the line for the i -th hash function
- And the slot q is hashed to is $h_i(q) = \left\lfloor \frac{f_i(q)}{W} \right\rfloor$.
- Distances to the left and right boundaries are

$$x_i(-1) = f_i(q) - h_i(q)W, \quad x_i(+1) = W - x_i(-1)$$



Scoring a Perturbation

- For a nearby point $\mathbf{p}$, $f_i(\mathbf{p}) - f_i(\mathbf{q})$ is a Gaussian random variable, with variance proportional to $\|\mathbf{p} - \mathbf{q}\|_2^2$
- Thus the probability that point $\mathbf{p}$ falls into slot $h_i(\mathbf{q}) + \delta$ can be estimated by:

$$\Pr[h_i(\mathbf{p}) = h_i(\mathbf{q}) + \delta_i] \approx e^{-Cx_i(\delta_i)^2}$$

- **Closer boundary:** crossing is more likely.
- **Farther boundary:** crossing is less likely.
- $C > 0$ depends on the distance between $\mathbf{p}$ and $\mathbf{q}$.

Scoring a Perturbation

- A bucket perturbation $\Delta = (\delta_1, \dots, \delta_M)$ requires all hash coordinates to match.
- The success probability of permutation vector is

$$\begin{aligned}\Pr[g(p) = g(q) + \Delta] &= \prod_{i=1}^M \Pr[h_i(p) = h_i(q) + \delta_i] \\ &\approx \prod_{i=1}^M e^{-Cx_i(\delta_i)^2} \\ &= e^{-C \sum_{i=1}^M x_i(\delta_i)^2}.\end{aligned}$$

Scoring a Perturbation

- This suggests that the likelihood that perturbation vector $\Delta = (\delta_1, \dots, \delta_M)$ will find a point close to $\mathbf{q}$ is related to

$$\text{score}(\Delta) = \sum_{i=1}^M x_i (\delta_i)^2$$

- **Smaller** scores indicate **more promising** buckets.
- The score depends on the query's position within each interval.
- We can **rank** probes without knowing $\mathbf{C}$, since it does not change their order.

Strategy 2: Query-Directed Probing

Strategy 2: Query-Directed Probing

- Suppose the query hashes to: $g(q) = (2,5,1)$
- **Basic** LSH checks this bucket.
- **Step-wise** multi-probe also checks other buckets, such as:
 $(2,5,0), (3,5,1), (3,5,0)$
- **Query-directed probing** decides which additional buckets to check first, using where the query falls inside each hash interval.
- **Which individual changes look promising?**

Strategy 2: Query-Directed Probing

- Probe buckets in increasing order of their **query-dependent scores**.
- Generating and sorting **all** possible perturbation would be wasteful.
- Instead, generate promising perturbations incrementally using a **min-heap**.

Goal: Produce the next lowest-score probe without enumerating all buckets.

Strategy 2: Query-Directed Probing

- For each coordinate i , consider both boundary moves:

$$(i, -1), \quad (i, +1)$$

- **Sort** their $2M$ boundary distances:

$$z_1 \leq z_2 \leq \dots \leq z_{2M}, \quad \pi_j = (i, \delta)$$

- z_j : the j -th smallest boundary distance. That is for some i with $\pi_j(i, \delta)$ we have $z_j = x_i(\delta)$
- π_j : the coordinate and direction of that move.

Strategy 2: Query-Directed Probing

- Given the sorted order of π
- We can represent a perturbation vector as a subset of $\{1, 2, \dots, 2M\}$
- For a **perturbation set** A its corresponding permutation vector is $\Delta_A = \{\pi_j \mid j \in A\}$
- Every set is associated with a score: $\text{score}(A) = \sum_{j \in A} z_j^2$
- The problem of **generating** perturbation vectors now reduces to the problem of **generating** perturbation sets in increasing order of their scores

Distances to hash boundaries

$W = 1 \cdot g(q) = (2, 5, 1)$

Hash function	Left boundary distance	Right boundary distance
h_1	0.7	0.3
h_2	0.4	0.6
h_3	0.2	0.8

Rank the individual moves.

Rank	Move	Distance	Resulting bucket
1	Decrease h_3 by 1	0.2	(2, 5, 0)
2	Increase h_1 by 1	0.3	(3, 5, 1)
3	Decrease h_2 by 1	0.4	(2, 4, 1)
4	Increase h_2 by 1	0.6	(2, 6, 1)
5	Decrease h_1 by 1	0.7	(1, 5, 1)
6	Increase h_3 by 1	0.8	(2, 5, 2)

A probe can combine several moves

Set A	Perturbation Δ_A	Bucket	Score
{1}	(0, 0, -1)	(2, 5, 0)	$0.2^2 = 0.04$
{2}	(+1, 0, 0)	(3 , 5, 1)	$0.3^2 = 0.09$
{1, 2}	(+1, 0, -1)	(3 , 5, 0)	$0.2^2 + 0.3^2 = 0.13$
{3}	(0, -1, 0)	(2, 4 , 1)	$0.4^2 = 0.16$

- Notice that combining two small changes costs less than making the third individual change: $0.13 < 0.16$.

Generating New Perturbation Sets

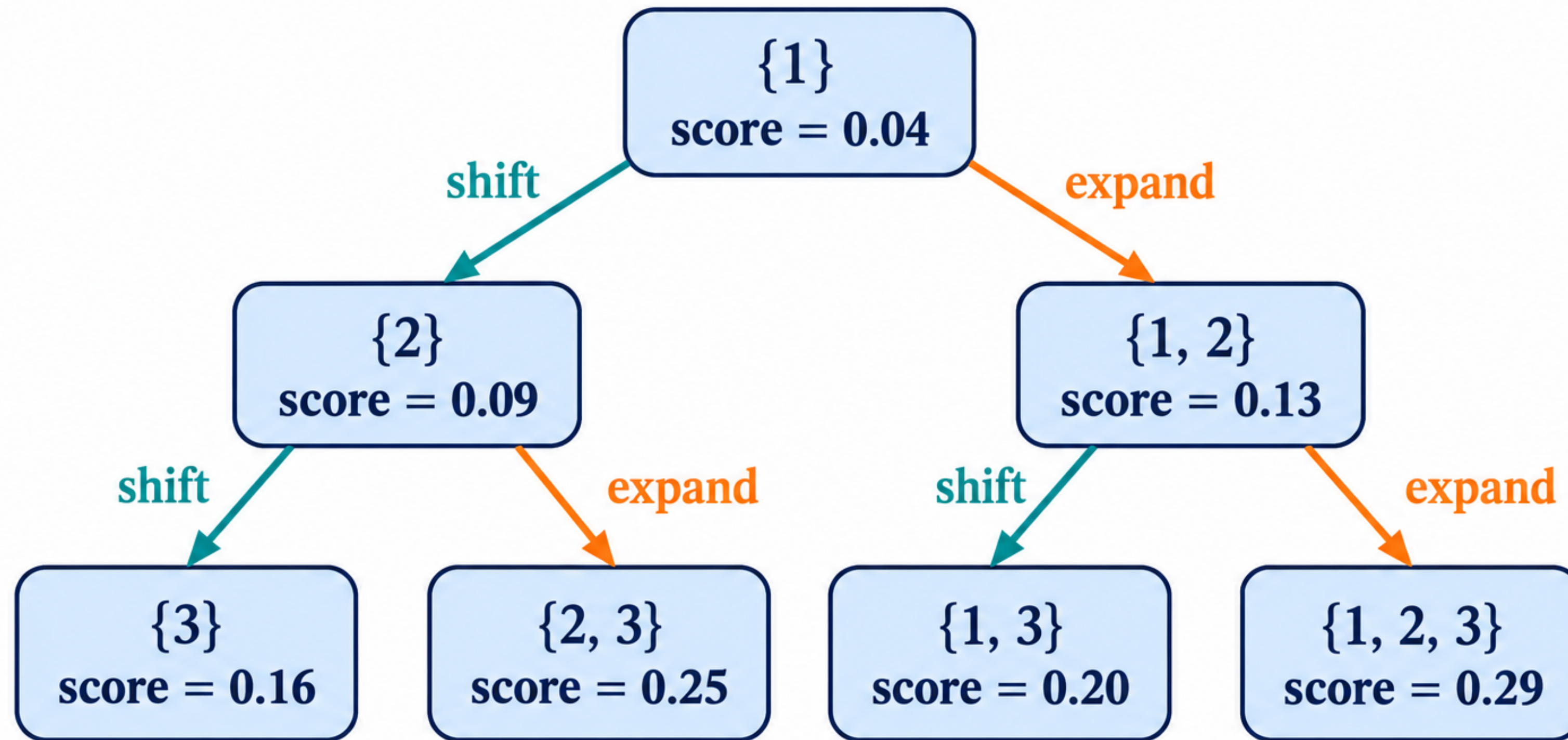
- We define two operations on a perturbation set:
 - **shift(A)**: This operation replaces $\max(A)$ by $1 + \max(A)$.
 - E.g. $\text{SHIFT}(\{1, 3, 4\}) = \{1, 3, 5\}$.
 - **expand(A)**: This operation adds the element $1 + \max(A)$ to the set A.
 - E.g. $\text{EXPAND}(\{1, 3, 4\}) = \{1, 3, 4, 5\}$.
- **Why do we need these operations?**
 - So that we can construct the first T perturbations sets using a min-heap.

Generating New Perturbation Sets

- Starting from min-heap containing only $\{1\}$
- Then we find the next candidate using operations **SHIFT** and **EXPAND**
- Neither operation decreases the score.
- Every set has a unique generation path.
- The min-heap determines which set to explore next.
- A child cannot have a lower score than its parent.

Generating probing candidates

$$z_1 = 0.2, z_2 = 0.3, z_3 = 0.4$$



$$\text{score}(A) = \sum_{j \in A} z_j^2$$

The heap selects the lowest score among all waiting candidates.

Query-Directed Probing Algorithm

```
Insert {1} into a min-heap, keyed by score
emitted = 0

while emitted < T and heap is not empty:
    A = extract the minimum-score set

    if max(A) < 2M:
        insert shift(A), keyed by its score
        insert expand(A), keyed by its score

    if A is valid:
        probe bucket  $g(q) + \Delta A$ 
        emitted = emitted + 1
```

- This is for one table only. We have to run this for all hash tables

Optimized Probing Sequence Construction

- **Idea:** Replace query-specific scores with **expected** scores

$$\text{score } q(A) = \sum_{j \in A} z_j(q)^2 \quad \longrightarrow \quad \widehat{\text{score}}(A) = \sum_{j \in A} \mathbb{E}[z_j^2]$$

- **Precompute once:**

- Expected costs depend on **M** and **W**, not the query. For $1 \leq j \leq M$:

$$\mathbb{E}[z_j^2] = \frac{j(j+1)}{4(M+1)(M+2)} \cdot W^2$$

- Run the heap procedure once and **store** the ordered perturbation sets $(A_1, A_2, \dots)$.

Optimized Probing Sequence Construction

- **For each query:**

- Sort its boundary moves to obtain the mapping $\pi_j = (i, \delta)$

- Look up the stored sets and map their ranks to actual perturbations:

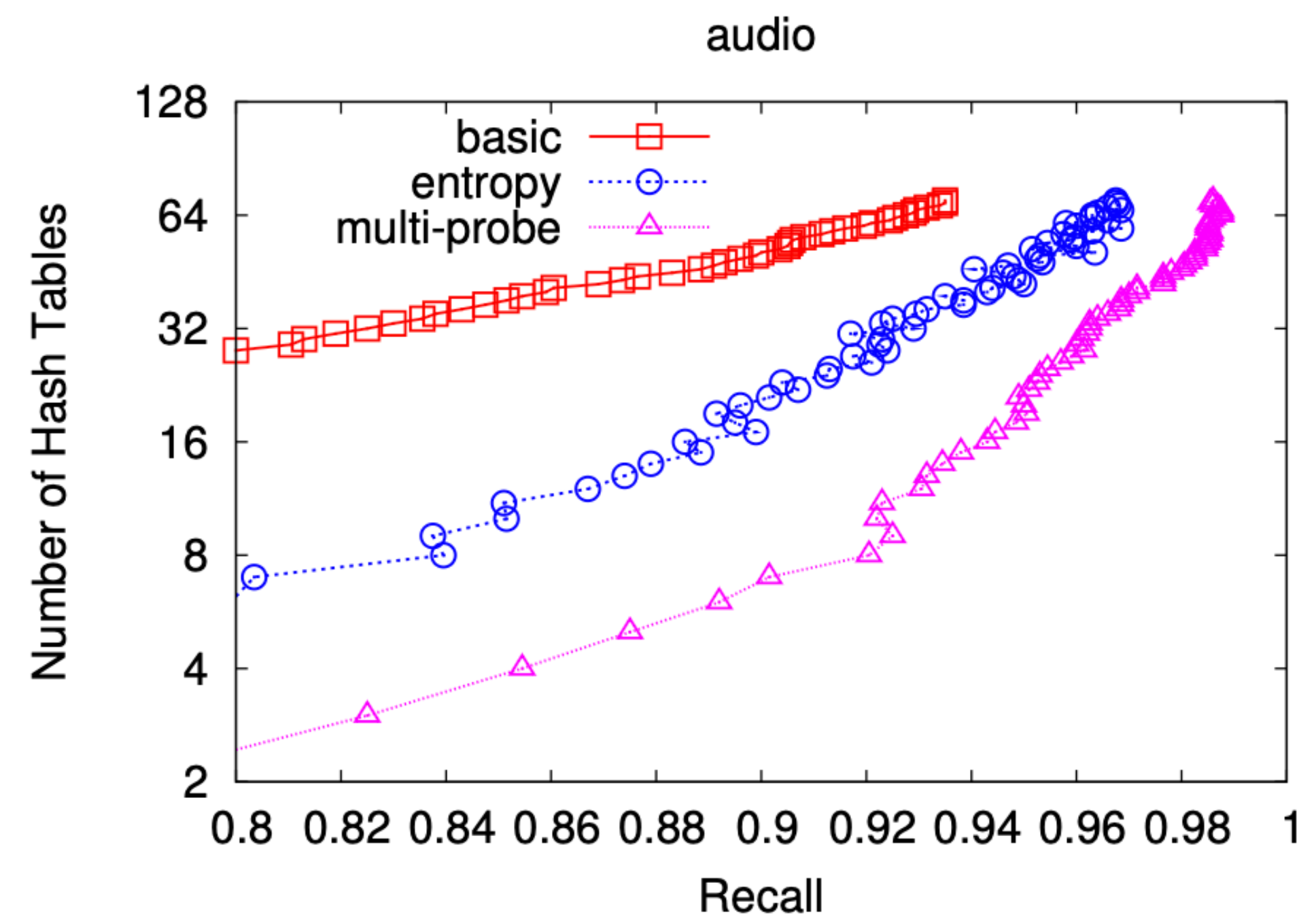
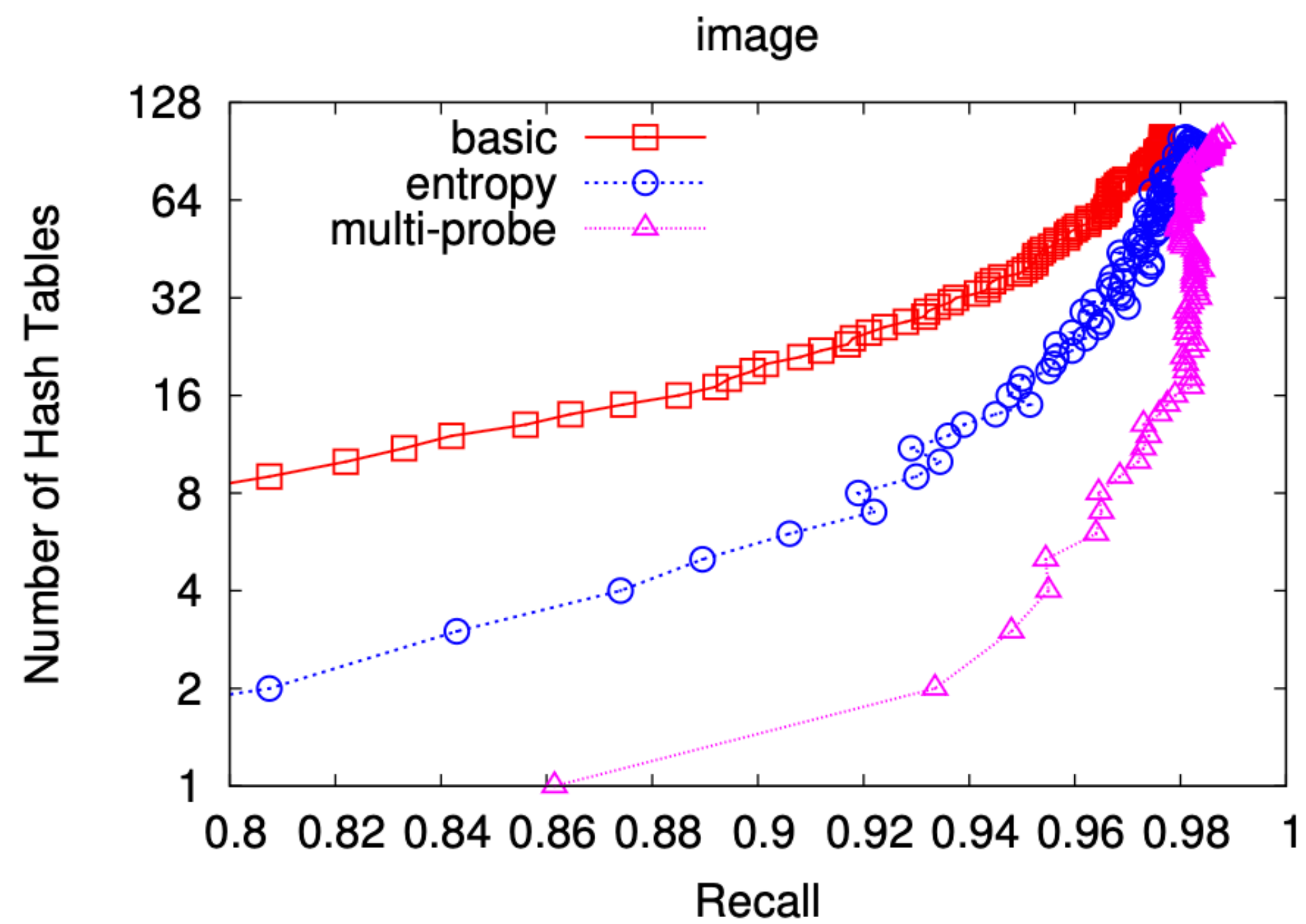
$$A = \{1, 2\} \xrightarrow{\pi_1=(3,-1), \pi_2=(1,+1)} \Delta_A = (+1, 0, -1)$$

- **Benefit:** No heap updates or candidate scoring at query time.
- **Trade-off:** The probing order is approximate; the rank-to-move mapping remains query-specific

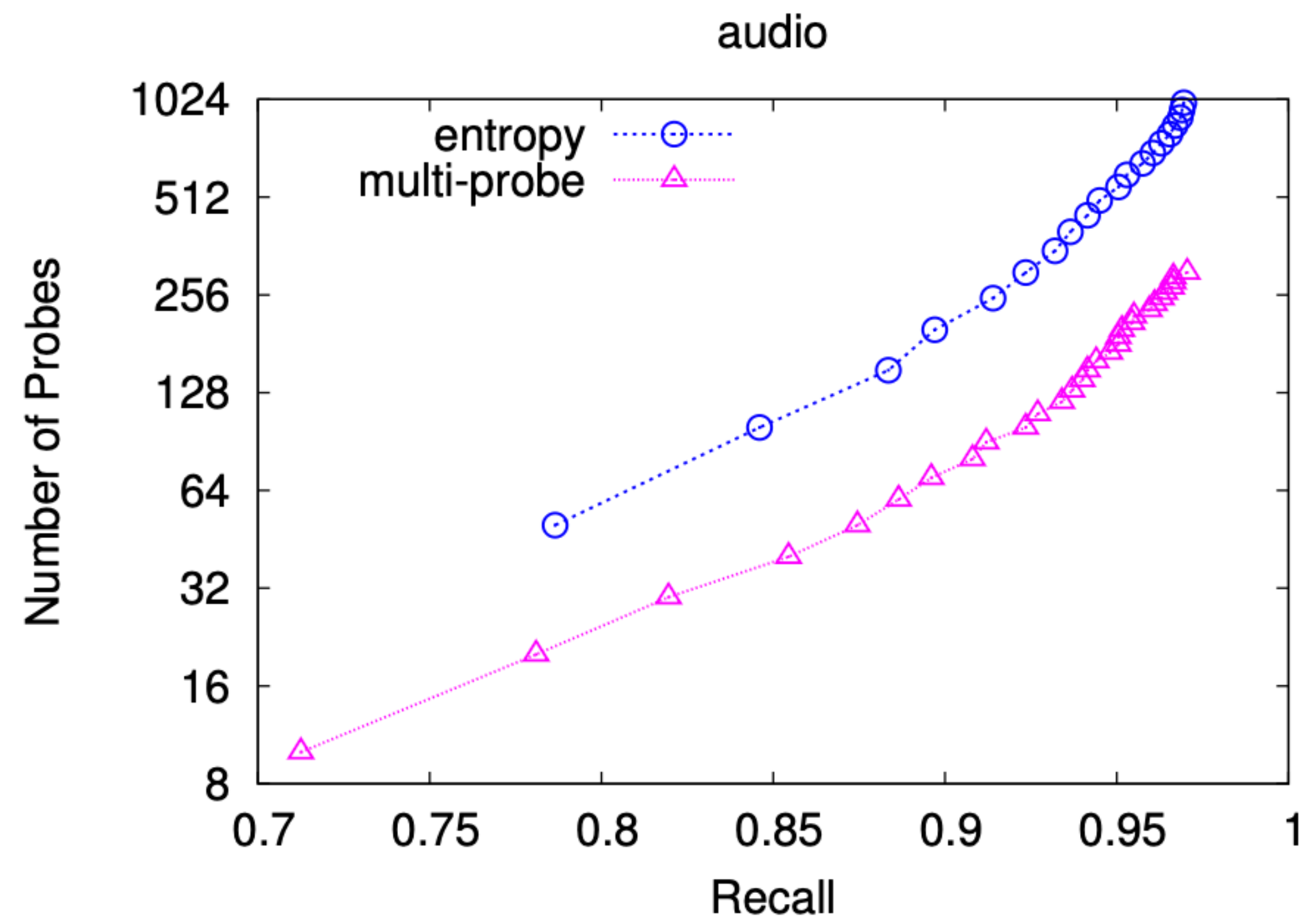
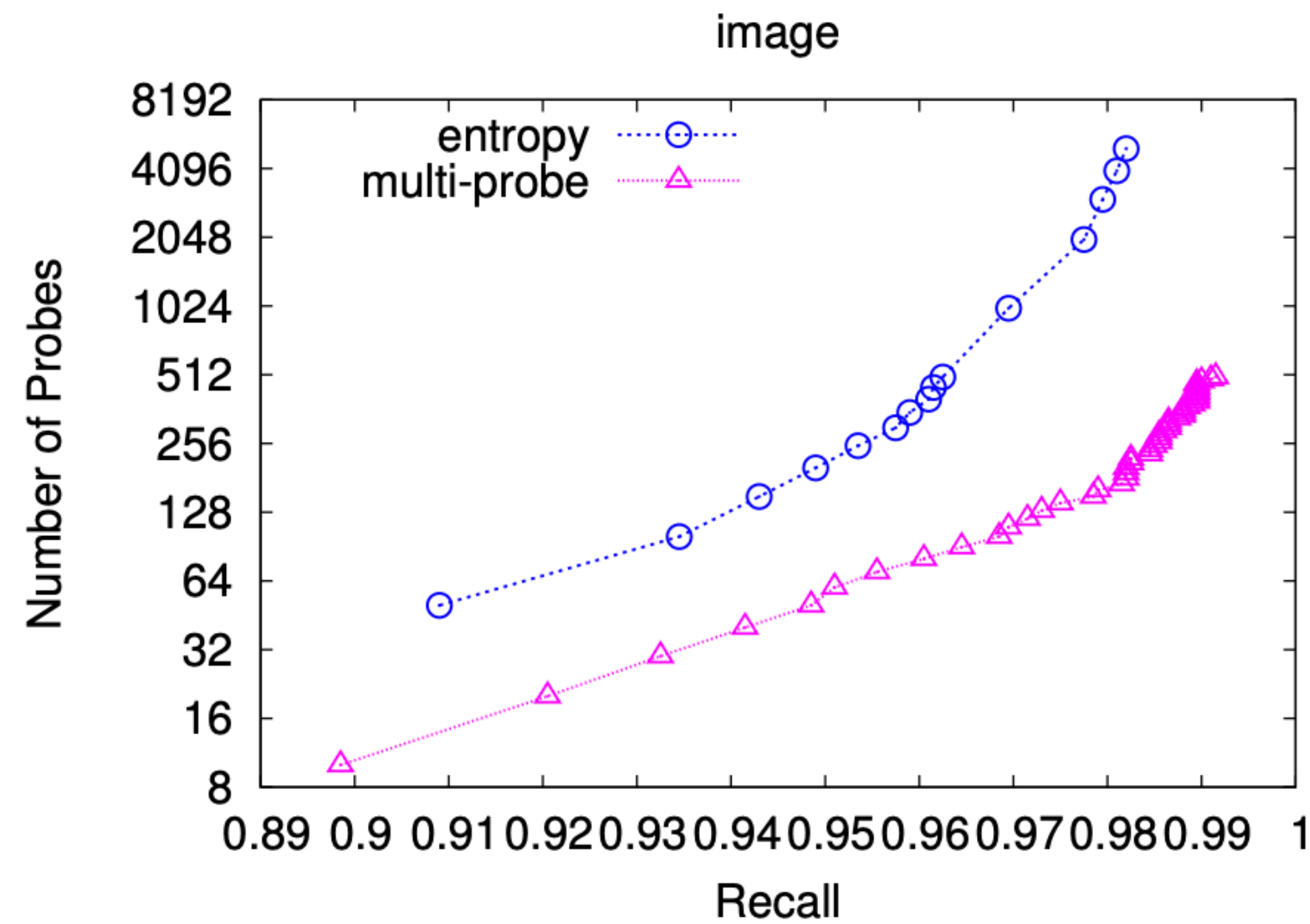
Experiments

Dataset	#Objects	#Dimension	Total Size
Image	1,312,581	64	336 MB
Audio	2,663,040	192	2.0 GB

Different LSH methods



Multi-probe vs. Entropy based



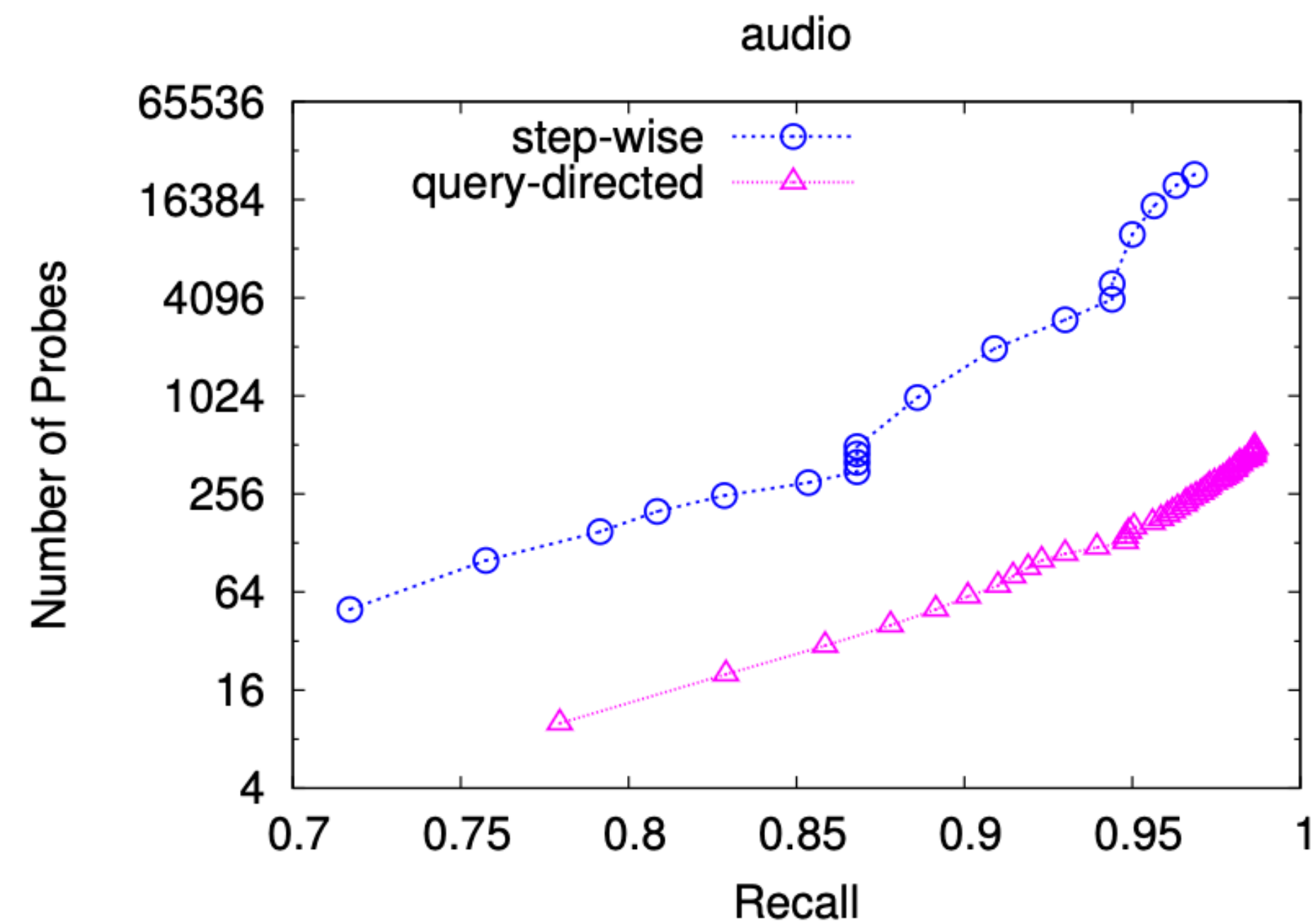
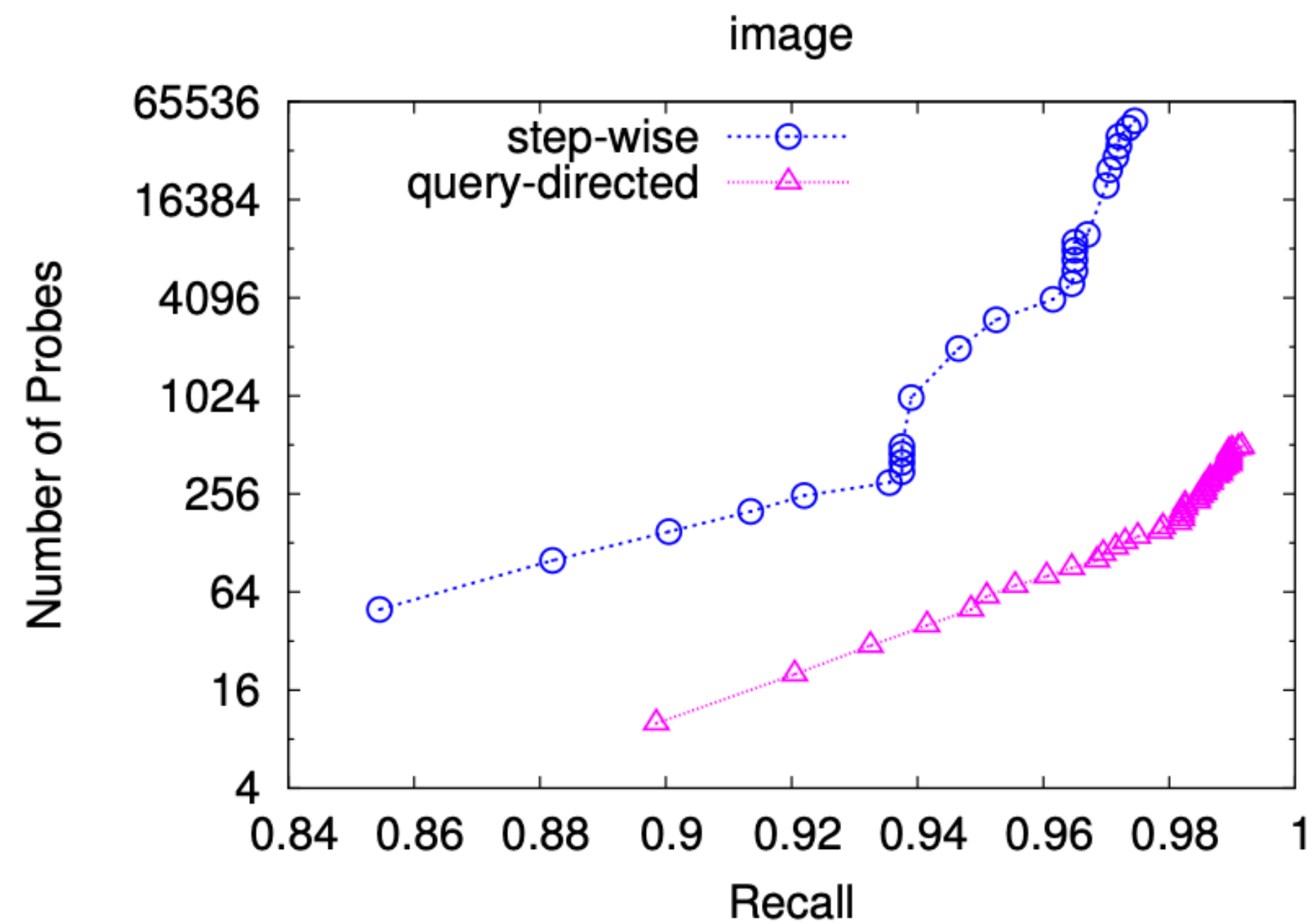
recall	method	error ratio	query time (s)	#hash tables	space ratio
0.96	basic	1.027	0.049	44	14.7
	entropy	1.023	0.094	21	7.0
	multi-probe	1.015	0.050	3	1.0
0.93	basic	1.036	0.044	30	15.0
	entropy	1.044	0.092	11	5.5
	multi-probe	1.053	0.039	2	1.0
0.90	basic	1.049	0.029	18	18.0
	entropy	1.036	0.078	6	6.0
	multi-probe	1.029	0.031	1	1.0

(a) image dataset

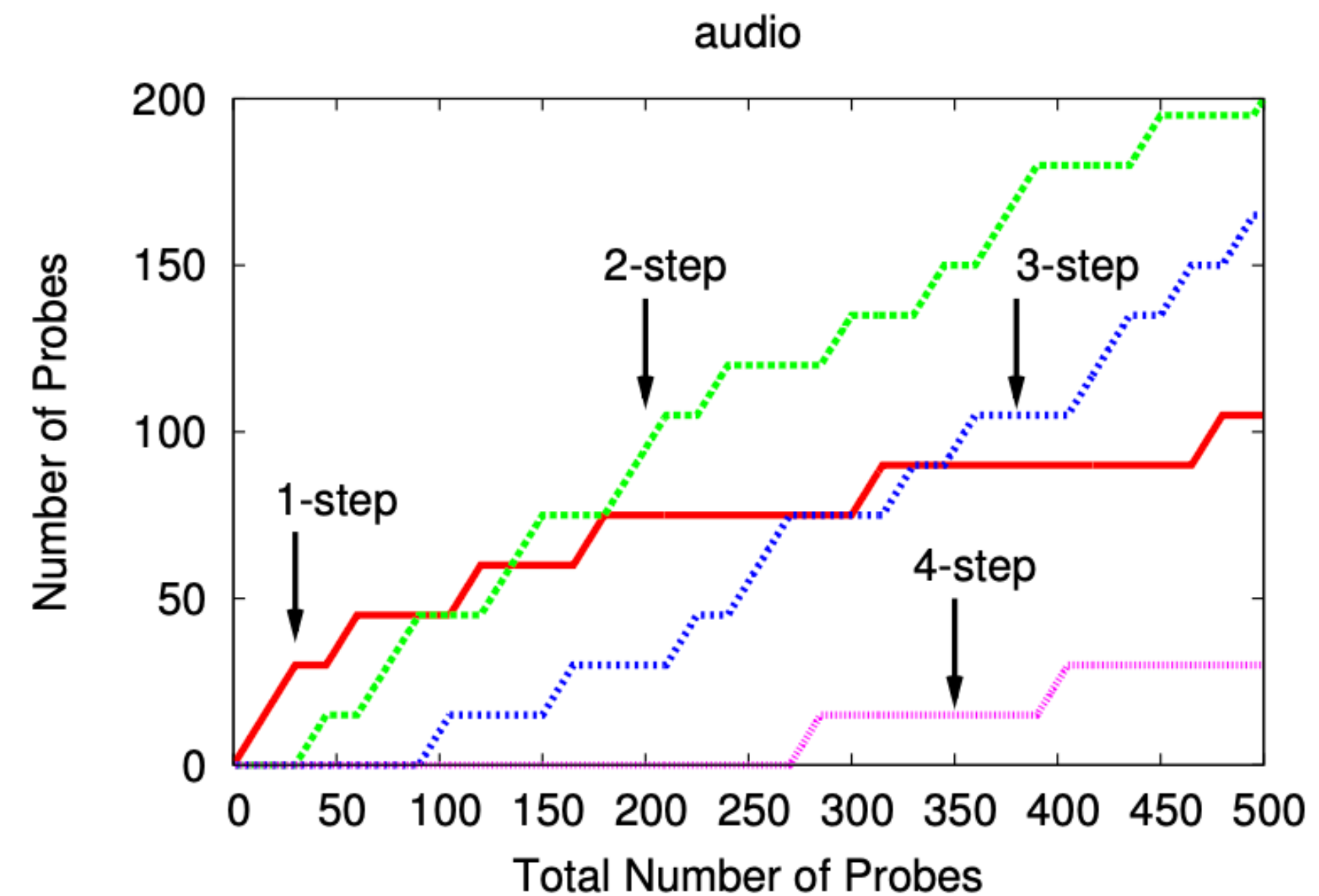
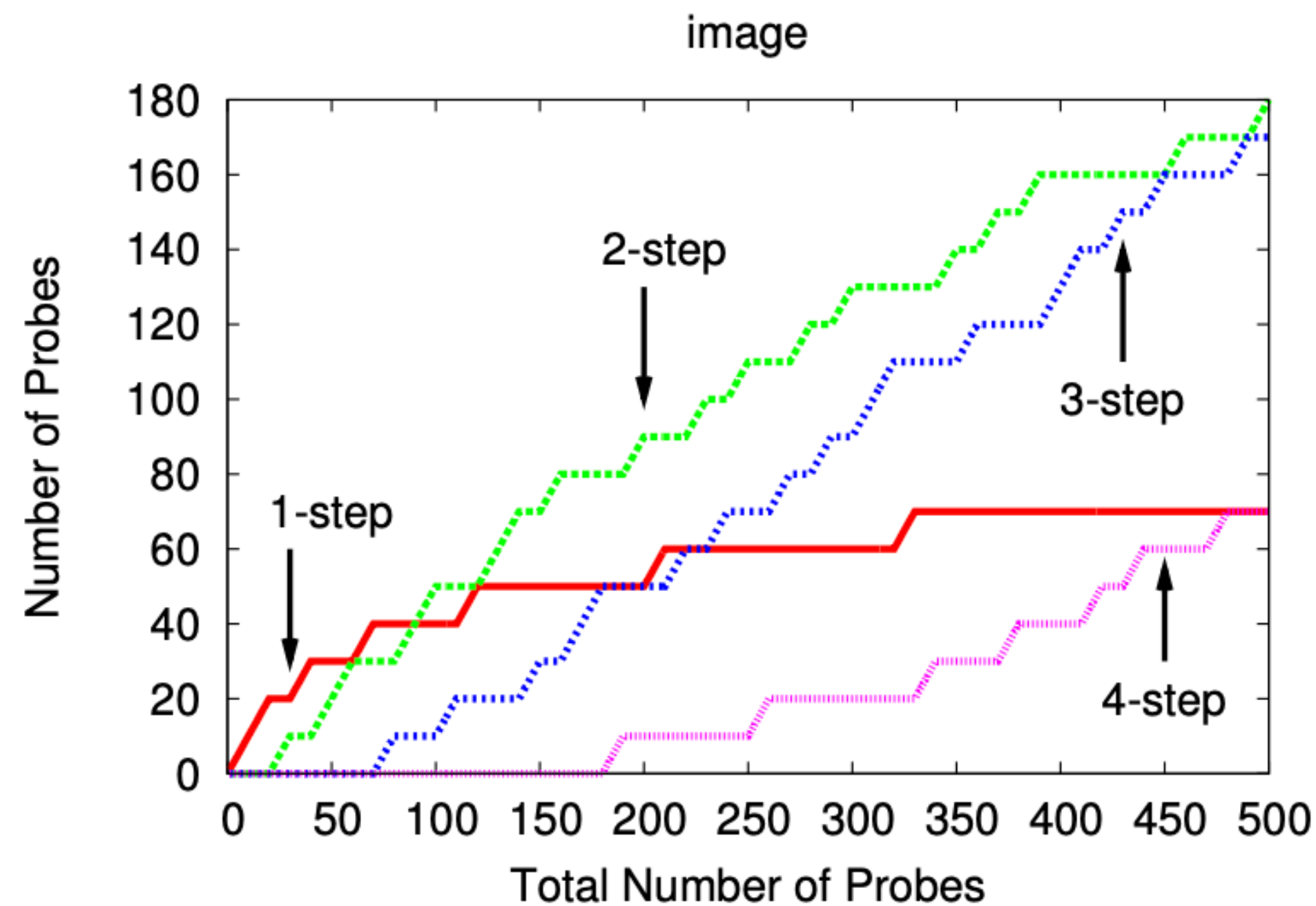
recall	method	error ratio	query time (s)	#hash tables	space ratio
0.94	basic	1.002	0.191	69	13.8
	entropy	1.002	0.242	44	8.8
	multi-probe	1.002	0.199	5	1.0
0.92	basic	1.003	0.174	61	15.3
	entropy	1.003	0.203	25	6.3
	multi-probe	1.002	0.163	4	1.0
0.90	basic	1.004	0.133	49	16.3
	entropy	1.003	0.181	19	6.3
	multi-probe	1.003	0.143	3	1.0

(b) audio dataset

Step-wise vs. Query directed probing



Step-wise vs. Query directed probing



Number of n-step perturbation sequences picked by query-directed probing:
many 2,3,4-step sequences are picked before all 1-step sequences are picked.

Conclusion

- Multi-probe LSH indexing:
 - **Systematically** probes multiple buckets per hash table
 - **Space-efficient** compared to basic LSH and entropy-based LSH
 - More **Time-efficient** than entropy-based LSH
 - Query-directed probing is far superior to step-wise probing.