

Similarity Search in High Dimensions via Hashing

10/20/26

Brandon Zhou
CS 848 – Fall 2026



Can We Find Similar Images?

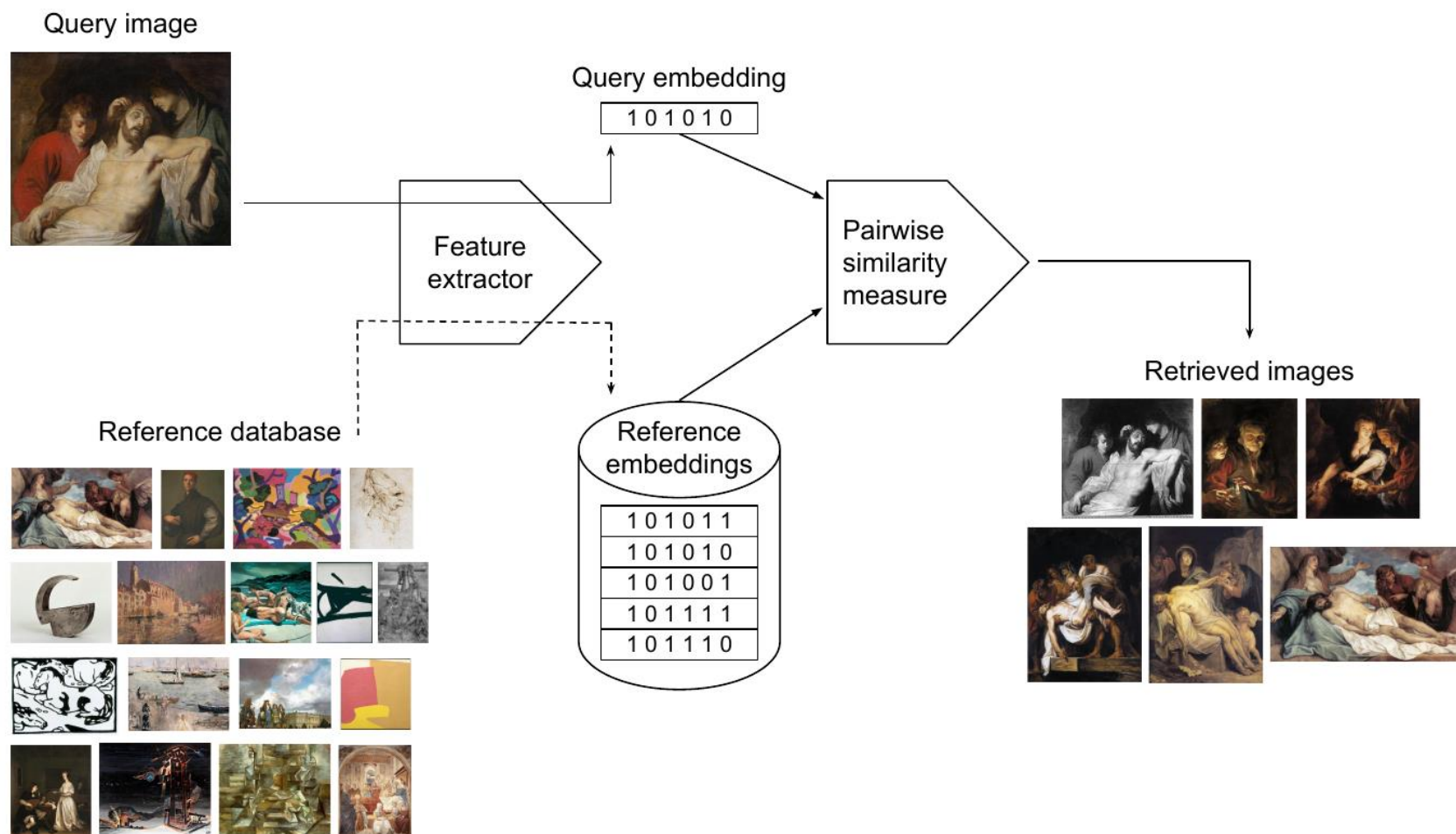
- Imagine a database containing millions of images.
- Given a query image, we want to retrieve the most similar images from the database.



Query



How Does a Computer Compare Images?



The Naive Solution: Exact Nearest-Neighbor Search

Given a dataset: $P = \{p_1, p_2, \dots, p_n\}$, and a query q , find $p^* = \operatorname{argmin}_{p \in P} \operatorname{dist}(p, q)$

Brute-force approach:

1. Calculate the distance from q to every point.
2. Identify the point with the minimum distance.
3. Return the nearest neighbor.

Time Complexity: $O(nd)$ where n is the number of points and d is the dimension.

The Cost of a Large Exact Scan

Suppose there are 10 million vectors, 128 dimensions each, then

$$10^7 \times 128 = 1.28 \times 10^9$$

coordinate comparisons for a single query. True nearest neighbour is guaranteed.

Challenges

1. Large computation cost
2. Potentially expensive disk accesses
3. Higher query latency as the database grows

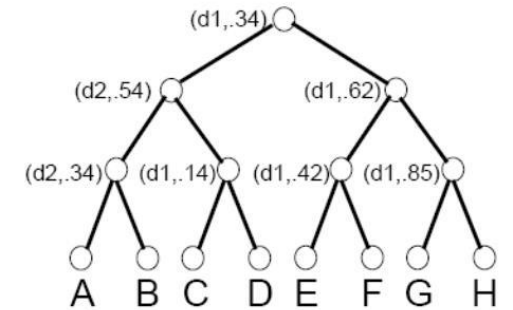
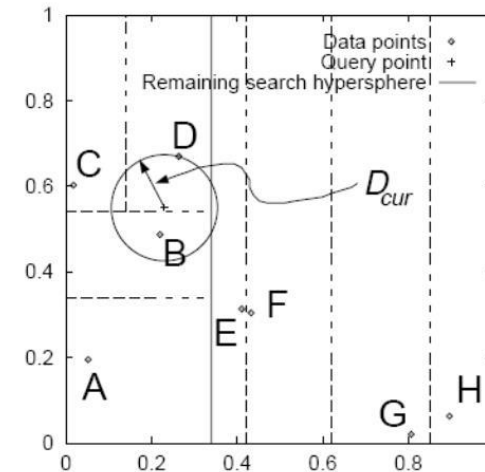
But do we always need the exact answer?

Can Tree-Based Indexes Help?

Spatial indexes organize points into regions. Instead of checking every point:

1. Partition the data space.
2. Build a hierarchical index.
3. Prune regions unlikely to contain the nearest neighbor.

KD Tree

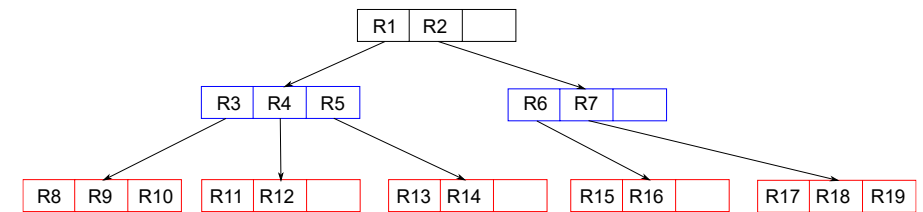
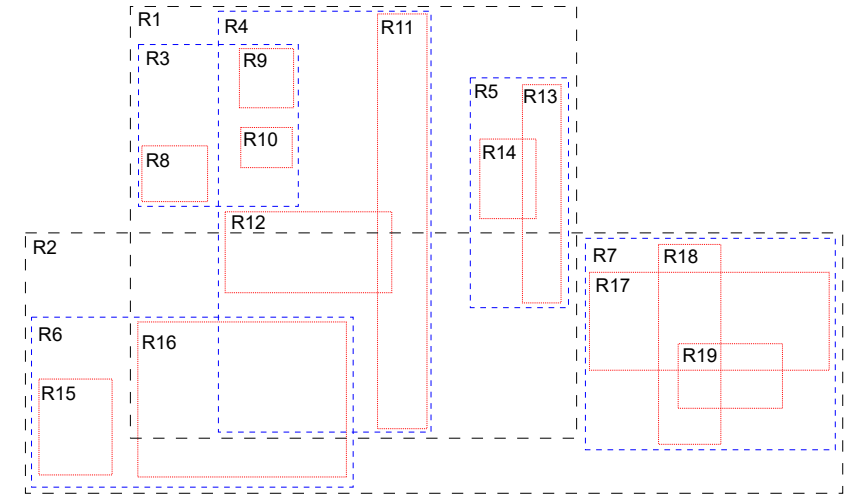


In low-dimensional spaces, this can significantly reduce search cost.

The Curse of Dimensionality

Tree-based indexes rely on pruning regions. However, as dimensionality increases:

- Bounding regions become less selective.
- Spatial regions may overlap extensively.
- More branches remain possible candidates.
- Search may approach linear-scan cost.



A tree is useful when it can eliminate large portions of the search space. High dimensionality can make that elimination much less effective.

A Nearly Identical Result May Be Enough



dist = 0



dist = 2



dist = 3

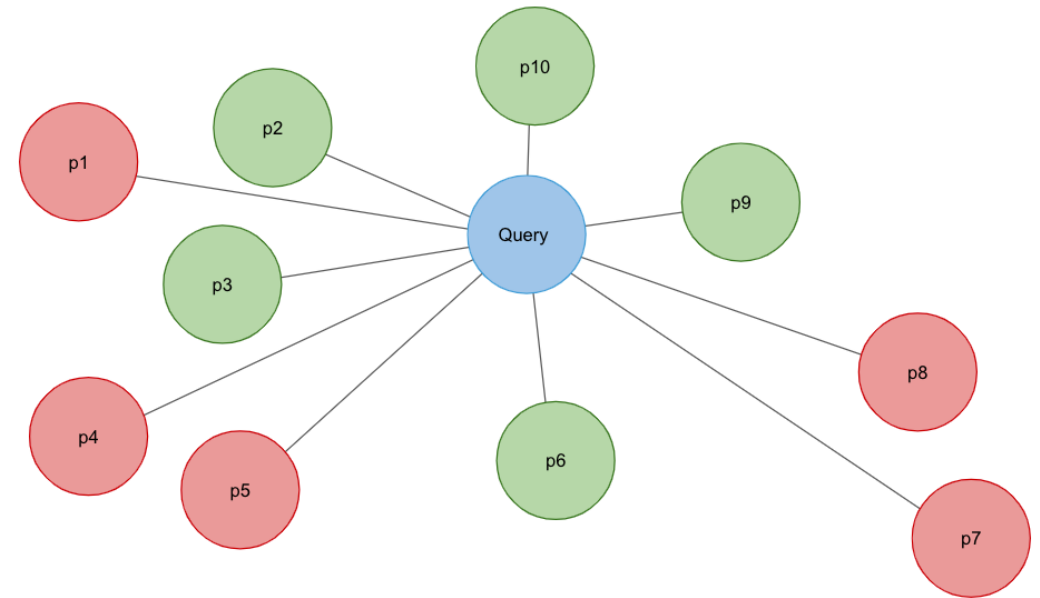
Would the second or third image be acceptable if the search were much faster?

Approximate Nearest-Neighbor Search

$$d(\hat{p}, q) \leq (1 + \varepsilon)d(p^*, q)$$

where

- q = Query point
- p^* = Exact nearest neighbor
- $\hat{p}$ = Approximate nearest neighbor
- $d(p, q)$ = Distance between p and q
- $\varepsilon \geq 0$, approximation tolerance
(larger ε allows greater error)



Locality-Sensitive Hashing

Traditional hashing: Distributes keys across hash buckets.

Locality-sensitive hashing: Makes nearby points more likely to map to the same hash value.

- **Nearby points:** High collision probability
- **Distant points:** Low collision probability



Hamming Distance

Hamming distance measures the number of different positions between two binary vectors.

	1	2	3	4	5	6
p	1	0	1	1	0	1
q	1	0	1	1	0	0

$$d_H(p, q) = \sum_{i=1}^D |p_i - q_i|$$

This is equivalent to L1 distance when all coordinates are binary.

Hamming Distance

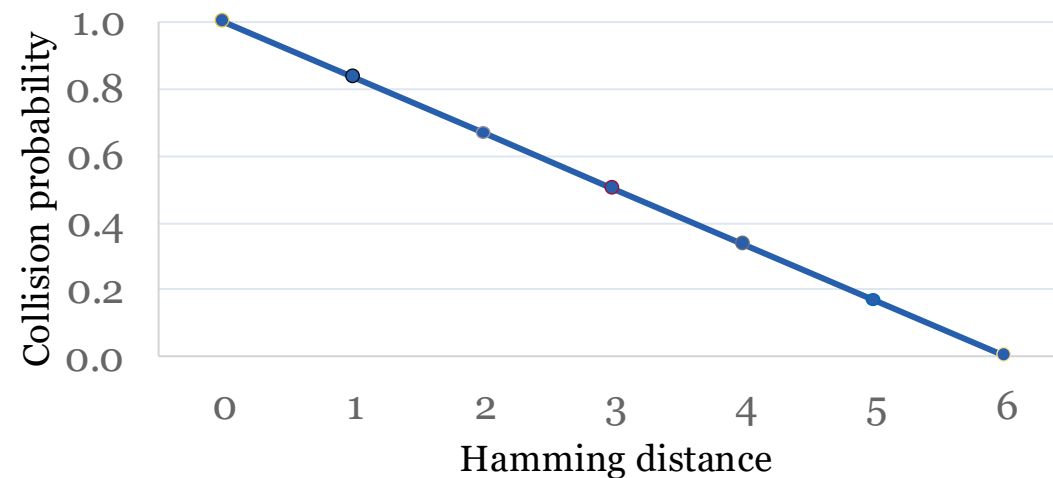
	1	2	3	4	5	6
p	1	0	1	1	0	1
q	1	0	1	1	0	0

Sampled position	$h(p)$	$h(q)$	Collision?
3	1	1	Yes
6	1	0	No

Similar vectors agree at more positions, so random bit sampling is more likely to produce matching values.

Collision Probability

$$\Pr[h(p) = h(q)] = 1 - \frac{d_H(p, q)}{D}$$



Collision probability decreases as Hamming distance increases.

Concatenating k Sampled Bits

Instead of selecting only one bit, select k random positions.

$$g(p) = (h_1(p), \dots, h_k(p))$$

	1	2	3	4	5	6
p	1	0	1	1	0	1

$$g(p) = \mathbf{110}$$

With $k = 3$, there are $2^3 = 8$ possible hash values.

The Trade-Off in k (Sample without Replacement)

$$\Pr[g(p) = g(q)] = \left(1 - \frac{d_H(p, q)}{D}\right)^k$$

Small k :

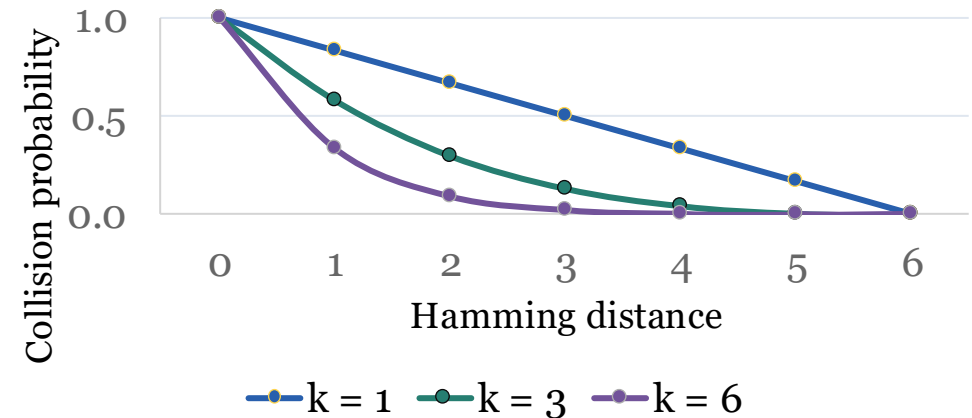
- Higher collision probability
- Larger candidate sets
- Higher chance of retrieving nearby points

Large k :

- Lower collision probability
- Smaller candidate sets
- Higher chance of missing nearby points

Trade-off

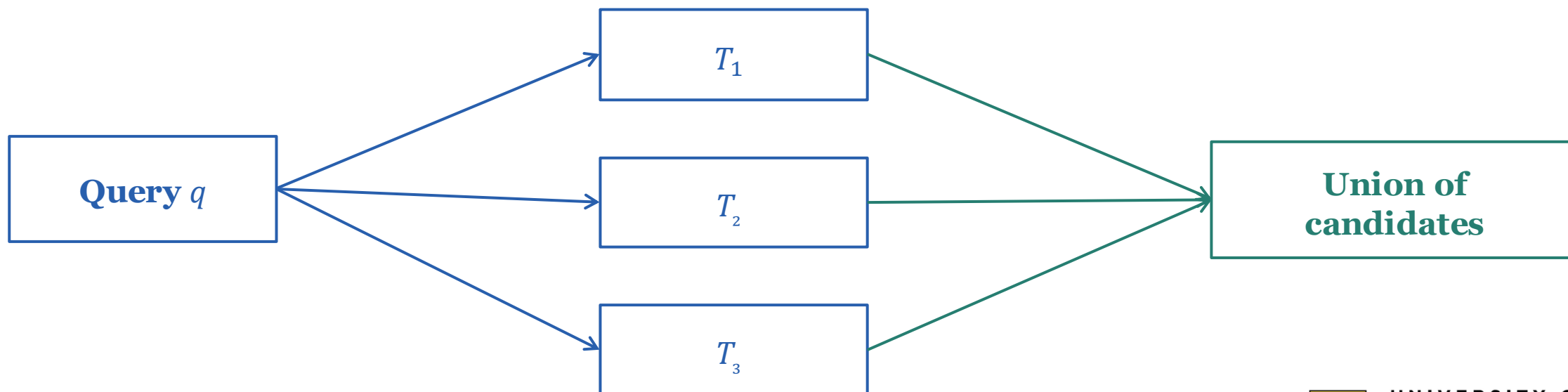
- More selective hashing reduces false matches,
- but can also miss true neighbors.



l Independent Hash Tables

A nearby point may fail to collide with the query in one hash table. To increase the chance, construct l independently sampled hash functions: $g_1, g_2, \dots, g_l$.

For a query q , we are interested in $T_1[g_1(q)], T_2[g_2(q)], \dots, T_l[g_l(q)]$, where T is the hash table. Retrieve candidates from the matching bucket in each table. Then combine the candidates.



Preprocessing Algorithm

Algorithm Preprocessing

Input A set of points P ,

l (number of hash tables),

Output Hash tables $\mathcal{T}_i, i = 1, \dots, l$

Foreach $i = 1, \dots, l$

Initialize hash table $\mathcal{T}_i$ by generating
a random hash function $g_i(\cdot)$

Foreach $i = 1, \dots, l$

Foreach $j = 1, \dots, n$

Store point p_j on bucket $g_i(p_j)$ of hash table $\mathcal{T}_i$

- **Initialize l Hash Tables**

- Create l independent hash tables.
- Each table uses a randomly generated hash function g_i .

- **Hash Every Data Point**

- For each point p_j , compute its hash key $g_i(p_j)$.
- Insert the point into the corresponding bucket in table $\mathcal{T}_i$.

- **Repeat for All Tables**

- Every data point is indexed in all l hash tables.
- Different hash functions create different groupings of the same dataset.

Query Answering Algorithm

Algorithm Approximate Nearest Neighbor Query

Input A query point q ,

K (number of appr. nearest neighbors)

Access To hash tables $\mathcal{T}_i, i = 1, \dots, l$

generated by the preprocessing algorithm

Output K (or less) appr. nearest neighbors

$S \leftarrow \emptyset$

Foreach $i = 1, \dots, l$

$S \leftarrow S \cup \{\text{points found in } g_i(q) \text{ bucket of table } \mathcal{T}_i\}$

Return the K nearest neighbors of q found in set S

/* Can be found by main memory linear search */

- **Hash the Query**

- Compute the hash key $g_i(q)$ for each of the l hash tables.
- Retrieve the corresponding bucket from each table.

- **Collect Candidate Neighbors**

- Combine the retrieved points into a candidate set S .
- Remove duplicates if the same point appears in multiple buckets.

- **Find the Nearest Neighbors**

- Compute the actual distances between the query and each candidate in S .
- Return the K closest candidates.

Hamming LSH

DATABASE

p_1	1	0	1	1	0	1
p_2	1	1	1	0	0	0
p_3	0	0	1	1	1	0
p_4	0	1	0	0	1	1
p_5	1	1	1	0	1	0

$$d = 6, k = 3, l = 2$$

QUERY

q	1	0	1	1	0	0
-----	---	---	---	---	---	---

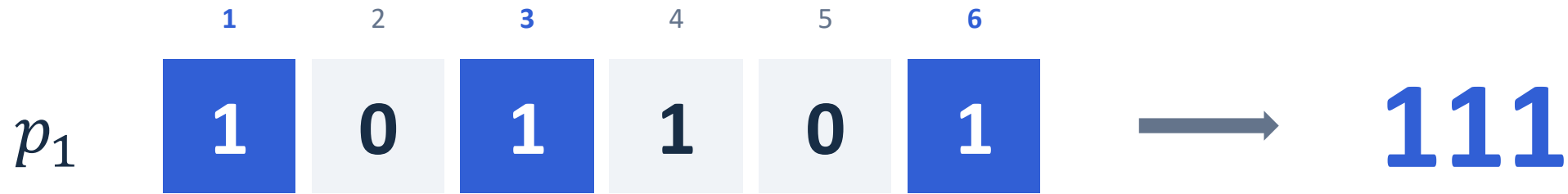
Find the closest point

Count the bits that differ.

Index Construction

$$I_1 = (1,3,6)$$

$$g_1(p) = (p[1], p[3], p[6])$$



$$g_1(p_1) = 111$$

The selected bits become a three-bit bucket key.

Table 1: Hashes and Buckets

	$I_1 = (1,3,6)$							BUCKET KEY
	1	2	3	4	5	6		
p_1	1	0	1	1	0	1	→	111
p_2	1	1	1	0	0	0	→	110
p_3	0	0	1	1	1	0	→	010
p_4	0	1	0	0	1	1	→	001
p_5	1	1	1	0	1	0	→	110

TABLE 1

111	p_1
110	p_2, p_5
010	p_3
001	p_4

Table 2: A Different Projection

$I_2 = (2,4,5)$

	1	2	3	4	5	6	
p_1	1	0	1	1	0	1	→ 010
p_2	1	1	1	0	0	0	→ 100
p_3	0	0	1	1	1	0	→ 011
p_4	0	1	0	0	1	1	→ 101
p_5	1	1	1	0	1	0	→ 101

TABLE 2

010	p_1
100	p_2
011	p_3
101	p_4, p_5

Query Processing

$$q = 101100$$

TABLE 1

1	2	3	4	5	6
1	0	1	1	0	0

→

$$g_1(q) = 110$$

TABLE 2

1	2	3	4	5	6
1	0	1	1	0	0

→

$$g_2(q) = 010$$

Query Processing

TABLE 1

110	p_2, p_5
-----	------------

TABLE 2

010	p_1
-----	-------



$$C = \{p_1, p_2, p_5\}$$

3 of 5 points

Compare only these candidates.

Exact Distances Decide the Ranking

$$d_H(q, p) = \sum_{i=1}^6 [q[i] \neq p[i]]$$

	1	2	3	4	5	6
q	1	0	1	1	0	0

p_1	1	0	1	1	0	1
-------	---	---	---	---	---	---

$$d_H(q, p_1) = 1$$

p_2	1	1	1	0	0	0
-------	---	---	---	---	---	---

$$d_H(q, p_2) = 2$$

p_5	1	1	1	0	1	0
-------	---	---	---	---	---	---

$$d_H(q, p_5) = 3$$

The Nearest Candidate is the Answer

$$p^* = p_1$$

$$p_1 \quad \begin{array}{|c|c|c|c|c|c|} \hline 1 & 0 & 1 & 1 & 0 & 1 \\ \hline \end{array}$$

$$d_H(q, p_1) = 1$$

The true nearest neighbor in this example, found from Table 2!

In general, LSH may miss the global nearest neighbor.

A Second Table Gives Another Chance

TABLE 1

$$g_1(q) = 110 \neq 111 = g_1(p_1)$$

Misses the nearest neighbor

TABLE 2

$$g_2(q) = 010 = g_2(p_1)$$

Retrieves the nearest neighbor

Nearby points get more chances to collide.

LET'S DO A LIVE LSH DEMO

Why is LSH Approximate?

Suppose the true nearest neighbor does not share a bucket with q in any hash table.

Then:

True nearest neighbor $\notin$ Candidate set

LSH cannot return that point. This is why the algorithm is *probabilistic*.

Increasing l : Improves the probability of retrieving nearby points.

Increasing k : Reduces unwanted collisions but can also reduce true-neighbor collisions.

What Makes a Hash Function Locality-Sensitive?

A family H of hash functions is (r_1, r_2, p_1, p_2) –sensitive if:

$$d(x, y) \leq r_1 \Rightarrow \Pr[h(x) = h(y)] \geq p_1$$

$$d(x, y) > r_2 \Rightarrow \Pr[h(x) = h(y)] \leq p_2$$

and $r_1 < r_2$, $0 < p_2 < p_1 < 1$.

Nearby points should collide more frequently than distant points.

What Is the Probability of Finding a Neighbor?

Suppose two points collide in one table with probability s .

Probability of not colliding:

$$1 - s$$

Probability of not colliding in any of l independent tables:

$$(1 - s)^l$$

Therefore:

$$\Pr[\text{Collision in at least one table}] = 1 - (1 - s)^l$$

More tables increase the chance of finding a nearby point, at the cost of more storage.

Improved Theoretical Query Time

Original LSH (Indyk & Motwani, 1998)

$$O(dn^{1/\varepsilon})$$

Improved LSH (Gionis et al., 1999)

$$O(dn^{1/(1+\varepsilon)})$$

where n = number of data points, d = dimensionality, ε = approximation tolerance.

- For any $\varepsilon > 0$, the new exponent is less than 1.
- Query time grows sublinearly with dataset size.

Key Idea: Choose k and l to balance the probability of finding nearby points against the cost of retrieving distant points.

Unary Encoding Preserves L_1 Distance

Let C be the maximum coordinate value.

$$v(x) = \text{Unary}_C(x_1) \cdots \text{Unary}_C(x_d)$$

$$d_H(v(x), v(y)) = \sum_{i=1}^d |x_i - y_i| = d_1(x, y)$$

New Dimension: $d' = d \times C$

For example, when $C = 5$,

$$\text{Unary}_5(2) = 11000, \quad \text{Unary}_5(4) = 11110$$

If $p_1 = (2,4), p_2 = (1,3)$, we have $v(p_1) = 11000 \ 11110, v(p_2) = 10000 \ 11100$.

So $d_H(v(p_1), v(p_2)) = 2 = d_1(p_1, p_2)$.

Optimizing Unary Encoding

Problem

- Unary encoding expands a d -dimensional vector to dC dimensions.
- Explicitly constructing these larger vectors can be expensive.

Optimization

- Only compute the bits needed by the hash function.
- For each original coordinate, determine its values at the selected unary positions.

Two Approaches

- Binary search: $O(d \log C)$ per hash evaluation.
- Precomputed lookup tables: $O(d)$ per hash evaluation.

Avoid explicitly constructing the expanded binary vectors, reducing hash computation cost.

From Logical Hashes to Physical Storage

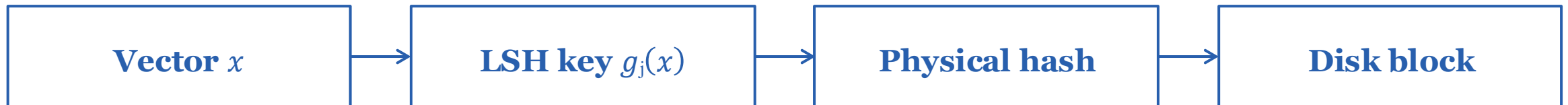
Since the logical LSH hash space may be much larger than the allocated physical table, the paper uses two levels of hashing.

First level: Locality-Sensitive Hashing

Vector $\rightarrow$ LSH bucket key

Second level: Conventional Hashing

LSH bucket key $\rightarrow$ Physical hash-table bucket



Handling Bucket Overflow

Suppose the bucket capacity $B = 3$, current bucket: $[p_1, p_2, p_3]$.

Another point p_4 hashes to the same bucket.

What happens?

- Theoretical construction: Overflow can be handled using chaining.
- Experimental implementation: If the physical bucket is full, the insertion into that table is skipped.

Trade-off

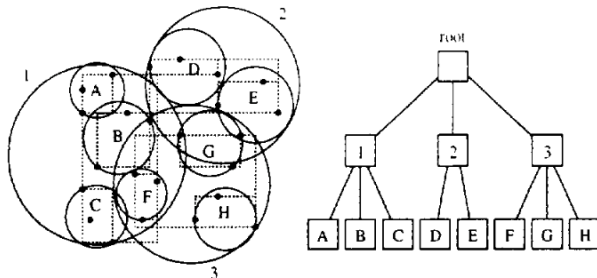
A point may be absent from every table if all relevant physical buckets are full.

SR-Trees: How the Index Organizes Points

- An SR-tree groups nearby feature vectors in a balanced tree. Leaves store points; internal entries describe the points in each child subtree.
- SR means Sphere/Rectangle. Each group has a bounding sphere and an axis-aligned bounding rectangle. All its points lie inside their intersection.

Why combine them?

- The intersection is no larger than either bound alone. The sphere bounds the maximum distance between points within the region, while the rectangle excludes empty space that the sphere may include.



SR-Trees: How Nearest-Neighbor Search Works

- **Visit promising branches first**

Compute distance lower bounds from the query to each child's sphere and rectangle; use the larger bound. Visit children with smaller lower bounds first and maintain the best candidates found.

- **Prune branches that cannot improve the answer**

For K neighbors, skip a subtree when its lower bound exceeds the current K^{th} best distance. Search ends when no unvisited branch can improve the candidate set.

- **Why compare SR-tree with LSH?**

SR-tree is shown to be comparable to or significantly more efficient than other tree-based spatial indexing methods.

The Experimental Setup

	Color histograms	Texture features
Representation	Color distribution of each image (CIE-Lab histogram)	Texture of each aerial-image tile (Gabor filter features)
Indexed points	19,000	270,000
Dimensions	8, 27, 64	60
Queries	500	1,000
Baseline	SR-tree	SR-tree / Sampled SR-tree

Main evaluation dimensions:

- *Search Cost*: Number of disk accesses (I/O operations) per query.
- *Approximation Quality*: How close the returned neighbors are to the true nearest neighbors (approximation error).

Effective Error

Approximate 1-NN Error

$$E = \frac{1}{|Q|} \sum_{q \in Q} \frac{d_{\text{LSH}}(q)}{d^*(q)}$$

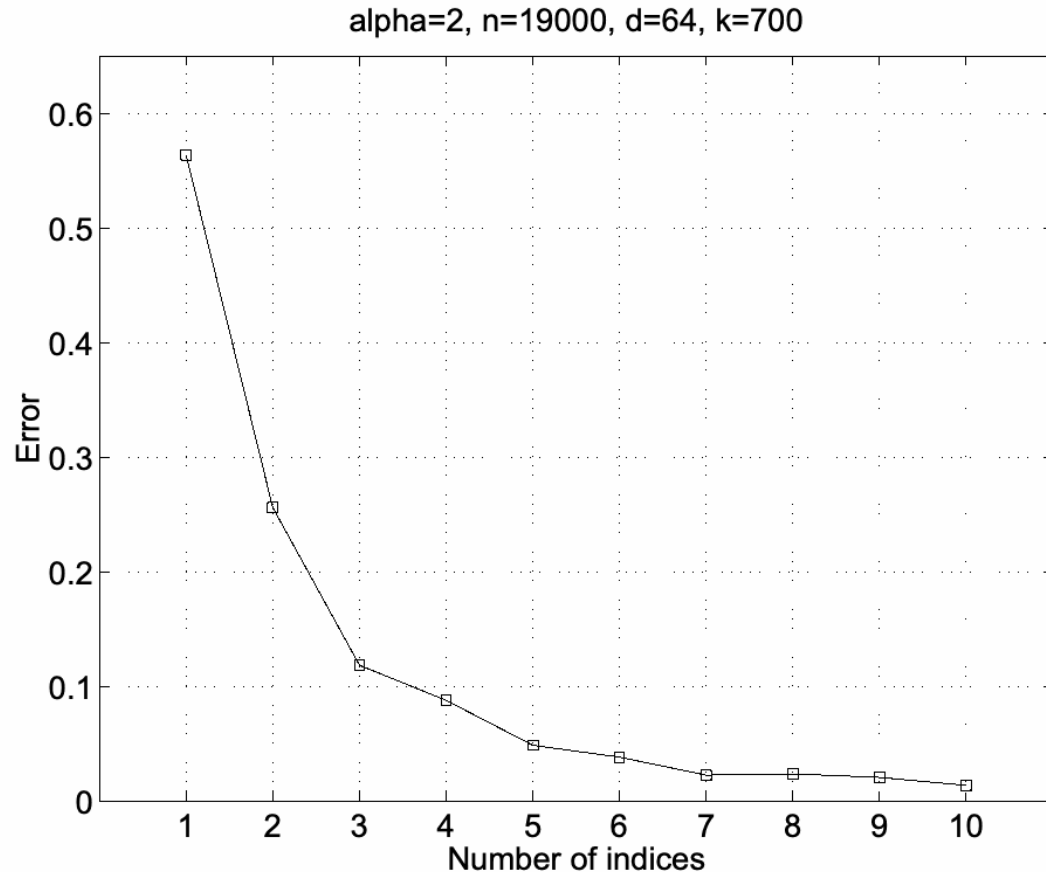
Approximate K -NN Error

$$E_K = \frac{1}{|Q|K} \sum_{q \in Q} \sum_{i=1}^K \frac{d_{\text{LSH},i}(q)}{d_i^*(q)}$$

where

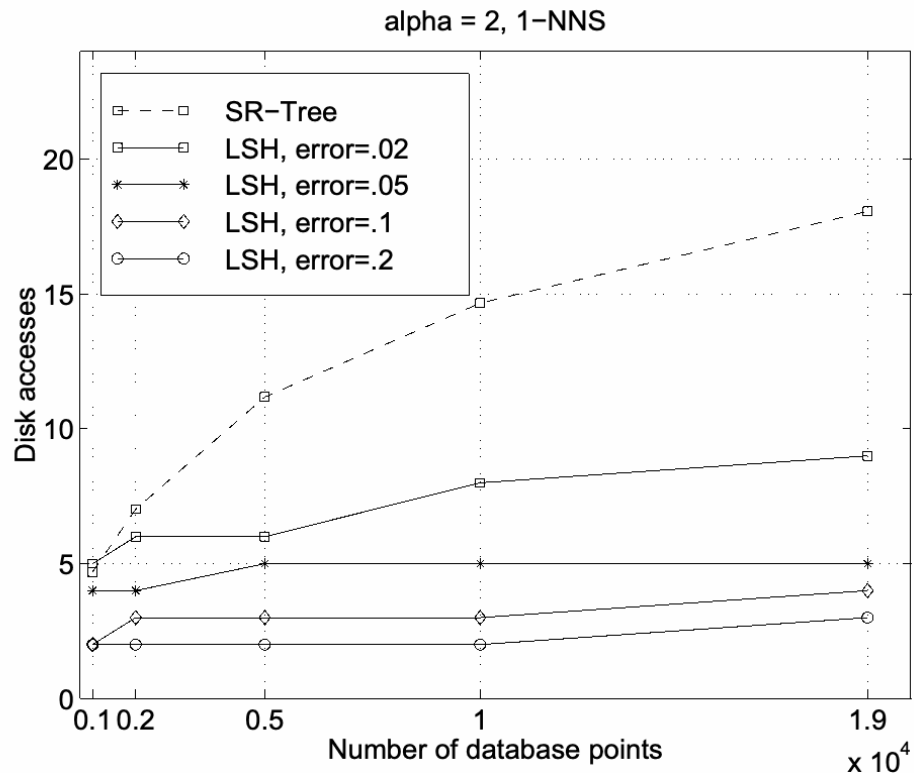
- Q : Queries for which LSH returns a result.
- $d_{\text{LSH}}(q)$: Distance from the query to the nearest point returned by LSH.
- $d^*(q)$: Distance from the query to the true nearest neighbor.

More Tables Reduce Average Error

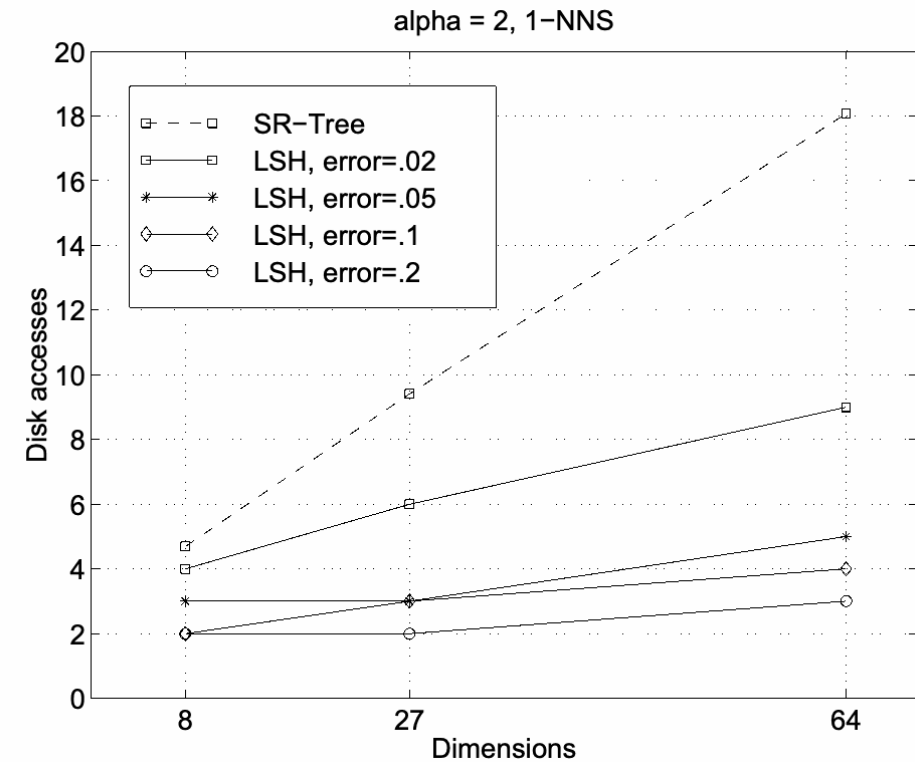


- Color histograms: 19,000 points, 64 dimensions.
- Average error falls as the number of indexes grows.
- Additional tables increase storage and probes.

Scaling with Database Size and Dimension

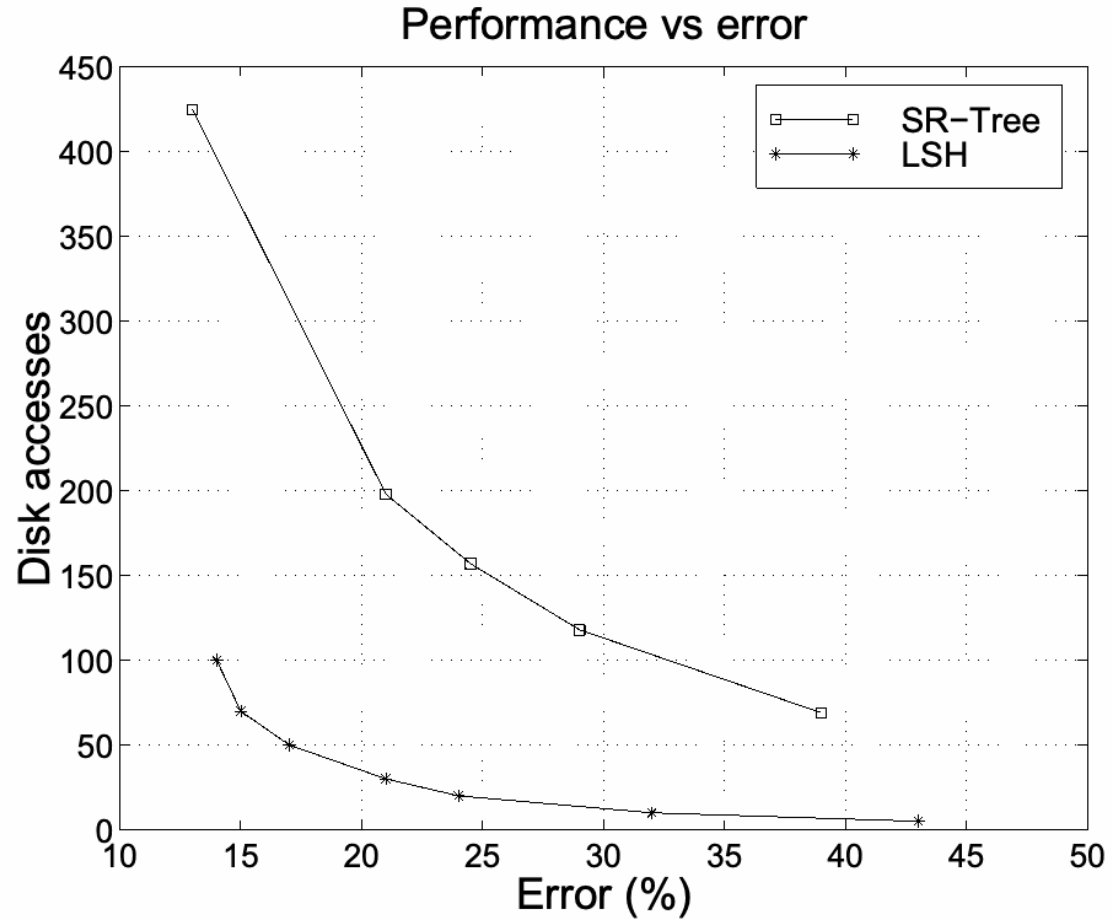


Database Size



Dimensions

Texture Results



- 270,000 texture vectors, 60 dimensions.
- The sampled SR-tree is used, which is built on a random subset of the entire dataset, allowing it to return approximate answers.
- LSH uses fewer block accesses at comparable average error.

Interpreting the Results

The results support three conclusions.

1. Approximation can improve efficiency. Exact search is not always necessary for useful similarity retrieval.
2. Randomized indexing can work well in high-dimensional spaces. LSH does not rely on conventional geometric space partitioning.
3. Parameter selection matters. Increasing l improves retrieval probability but also increases index size and I/O.

Strengths

- **Finds similar points efficiently:** Similar points are more likely to land in the same hash bucket, so LSH searches fewer candidates. It then compares their actual distances to select the closest ones.
- **Avoids unnecessary data expansion:** The implementation computes hashes without explicitly constructing the much larger unary-encoded vectors.
- **Offers control over accuracy and cost:** More hash tables improve the chances of finding nearby points, at the cost of more storage and disk reads.
- **Uses realistic experiments and a fairer comparison:** The paper tests real image features and compares LSH with both exact and approximate SR-tree searches, showing how the methods perform when both can trade accuracy for speed.

Limitations

- **Can miss the true nearest neighbors:** LSH may return points that are close to the query without finding the actual closest points.
- **The accuracy measures give an incomplete picture:** Low distance error does not show how many true nearest neighbors were found. A query can return enough results and still miss them.
- **Better accuracy requires more resources:** Using more hash tables increases storage needs and the number of disk reads.
- **Limited evaluation on modern high-dimensional data:** The authors evaluate vectors with dimensions ranging from 8 to 64, which is much lower than the hundreds or thousands of dimensions, which is commonly used in modern embedding-based applications.

Proposed Improvements

- Build the index more systematically to reduce reliance on random choices and potentially improve search efficiency.
- Use patterns in the data to create an index that better matches how points are distributed.
- Combine tree-based indexing with hashing to explore whether their strengths can improve search performance.
- Probe a few additional buckets that are likely to contain nearby points, trading more work per hash table for fewer tables and lower storage costs.
- Adapt the amount of searching to local data density, spending more effort where nearby points are spread across buckets and less where a small search already finds good candidates.

Real-World Applications of LSH

- **Image Similarity Search** (Wong et al., ICME 2007)
 - Find visually similar images in large image collections.
 - LSH retrieves candidate images without comparing every image.
- **Near-Duplicate Document Detection** (Manku et al., WWW 2007)
 - Identify web pages or documents with highly similar content.
 - Use SimHash-based fingerprints to efficiently detect near-duplicate documents.
- **Recommendation Systems** (Das et al., WWW 2007)
 - Group users with similar reading histories.
 - Google News used MinHash clustering to help generate personalized news recommendations.

UNIVERSITY OF WATERLOO



Questions?

References

- Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 1999. Similarity Search in High Dimensions via Hashing. In Proceedings of the 25th International Conference on Very Large Data Bases (VLDB '99). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 518–529.
- Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In Proceedings of the thirtieth annual ACM symposium on Theory of computing (STOC '98). Association for Computing Machinery, New York, NY, USA, 604–613. <https://doi.org/10.1145/276698.276876>
- Norio Katayama and Shin'ichi Satoh. 1997. The SR-tree: an index structure for high-dimensional nearest neighbor queries. SIGMOD Rec. 26, 2 (June 1997), 369–380. <https://doi.org/10.1145/253262.253347>
- Y. M. Wong, S. C. H. Hoi and M. R. Lyu, "An Empirical Study on Large-Scale Content-Based Image Retrieval," 2007 IEEE International Conference on Multimedia and Expo, Beijing, China, 2007, pp. 2206-2209, doi: 10.1109/ICME.2007.4285123
- Gurmeet Singh Manku, Arvind Jain, and Anish Das Sarma. 2007. Detecting near-duplicates for web crawling. In Proceedings of the 16th international conference on World Wide Web (WWW '07). Association for Computing Machinery, New York, NY, USA, 141–150. <https://doi.org/10.1145/1242572.1242592>
- Abhinandan S. Das, Mayur Datar, Ashutosh Garg, and Shyam Rajaram. 2007. Google news personalization: scalable online collaborative filtering. In Proceedings of the 16th international conference on World Wide Web (WWW '07). Association for Computing Machinery, New York, NY, USA, 271–280. <https://doi.org/10.1145/1242572.1242610>
- ChatGPT. LSH Demo