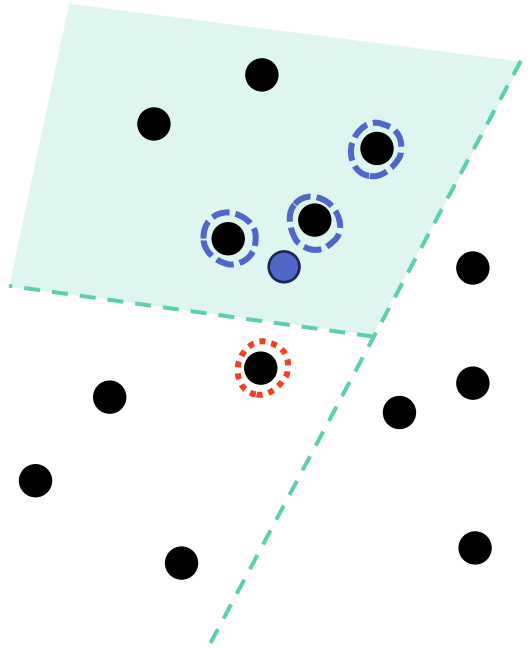
III. INDEXING

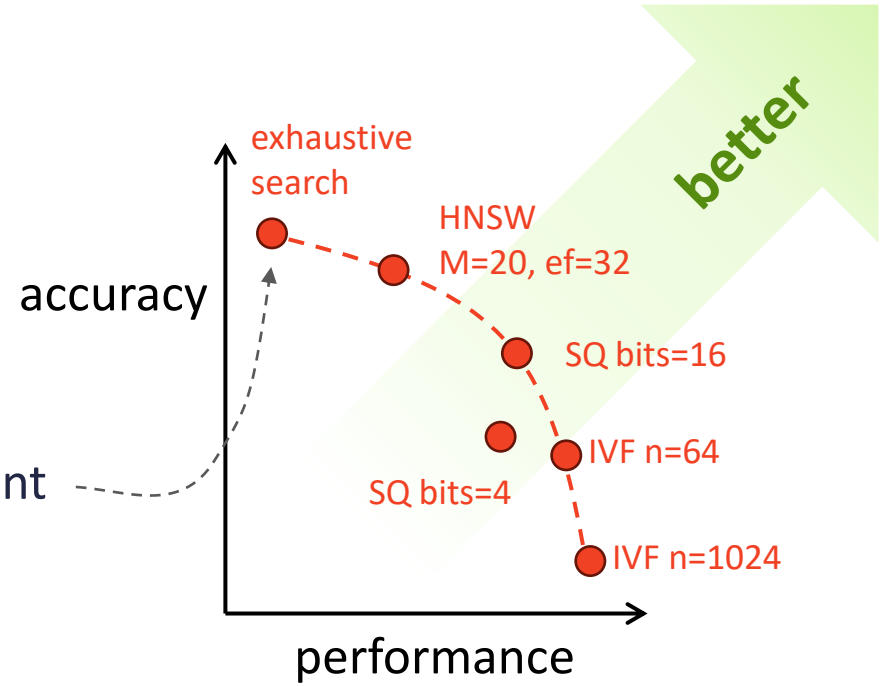
WHY INDEX?

- Flat (brute force) kNN search
  - Exact results
  - But N comparisons → Too slow!
- If we partition dataset
  - Fewer comparisons → faster search
  - Can introduce errors
- ANNS index is key to VDBMS performance!
- **But has costs!**
  - Can increase errors, memory, update cost



PERFORMANCE TRADEOFFS

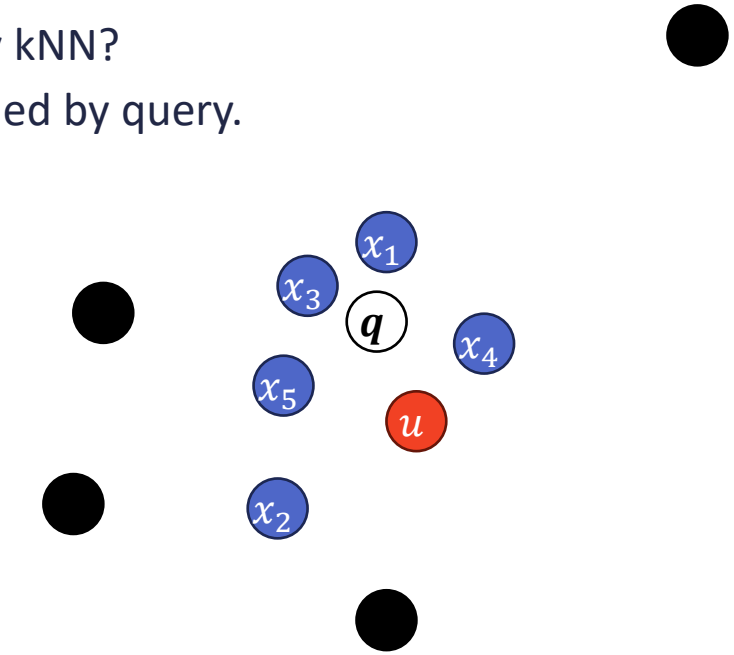
- ANNS indexes trade between:
  - Search speed
  - Accuracy
  - Memory
  - Build/update cost
- Crucial: search speed-accuracy tradeoff
  - **Index type + configuration** determine specific point
- How to choose index and configuration?
  - **Tune manually** on your data
  - **Automatically** using VDBMS optimizer (if exists)



(illustration only, not real data)

METRICS

- **Recall-K@K (aka "recall@K" aka "recall")**
  - Fraction of true *k* nearest neighbours returned by query
    - Alternative: out of *k* vectors returned, how many are truly kNN?
    - Alternative: overlap between true kNN and vectors returned by query.
  - Example:
    query $\boldsymbol{q}$ with $k = 5$ returns $x_1 \ldots x_5$
    x_1, x_3, x_4, x_5 are true nearest neigbhbours
    but x_2 is not ($\boldsymbol{u}$ is closer to $\boldsymbol{q}$ but is missed)
    → recall5@5 = $4/k$ = $4/5$ = 0.8 or 80%
- **Latency**: milliseconds per query
- **Throughput**: queries per second



ANNS INDEXES

- Vectors ≠ attributes
  ANNS index ≠ RDBMS index
- Can’t use B-trees.
- **Need new tricks:**
  - Learned (data-based) partitioning.
  - Lossy compression (quantization).
  - Randomization.
- New problems:
  - Slow index updates (due to complex structure)
  - Frequent rebuilds (since updates degrade index performance)

	Attribute	Vector
Granularity	attribute	whole vector
Accuracy	exact	approximate
Natural structure	**ordinal or sortable**	**no natural structure**
Fast updates, delete	**yes**	**sometimes**
Self-balancing	**yes**	**sometimes**
Data-dependent	no	often

INDEX TYPES

- **Cluster-based**: partition space to buckets of similar vectors
  - IVF, PQ
- **Graph-based**: connect similar vectors to make traversal graph
  - HNSW
  - DiskANN
- **LSH**: locality-sensitive hashing
  - Many variants
- **Tree**: partition space hierarchically
  - RP tree
  - ANNOY

ANNS index characteristics:

- Memory or disk resident:
  - Most are memory-resident
- Periodic rebuilding:
  - Some indexes require occasional rebuilds
  - E.g., classic HNSW, IVF
- Support incremental updates:
  - Yes, no, or partially supported
  - Real delete or tombstone?
- Error bounds:
  - Only LSH, RPTree

IT'S TIME!

- Let's look at a real index.
- **Flat Index**

IT'S TIME!

- Let's look at a real index.
- **Flat Index** … actually just brute force search.

FLAT INDEX

- Store vectors in table.
- To query: scan table, compute distances, return top-k
  - With priority queue or just sorting
- Really?

FLAT INDEX

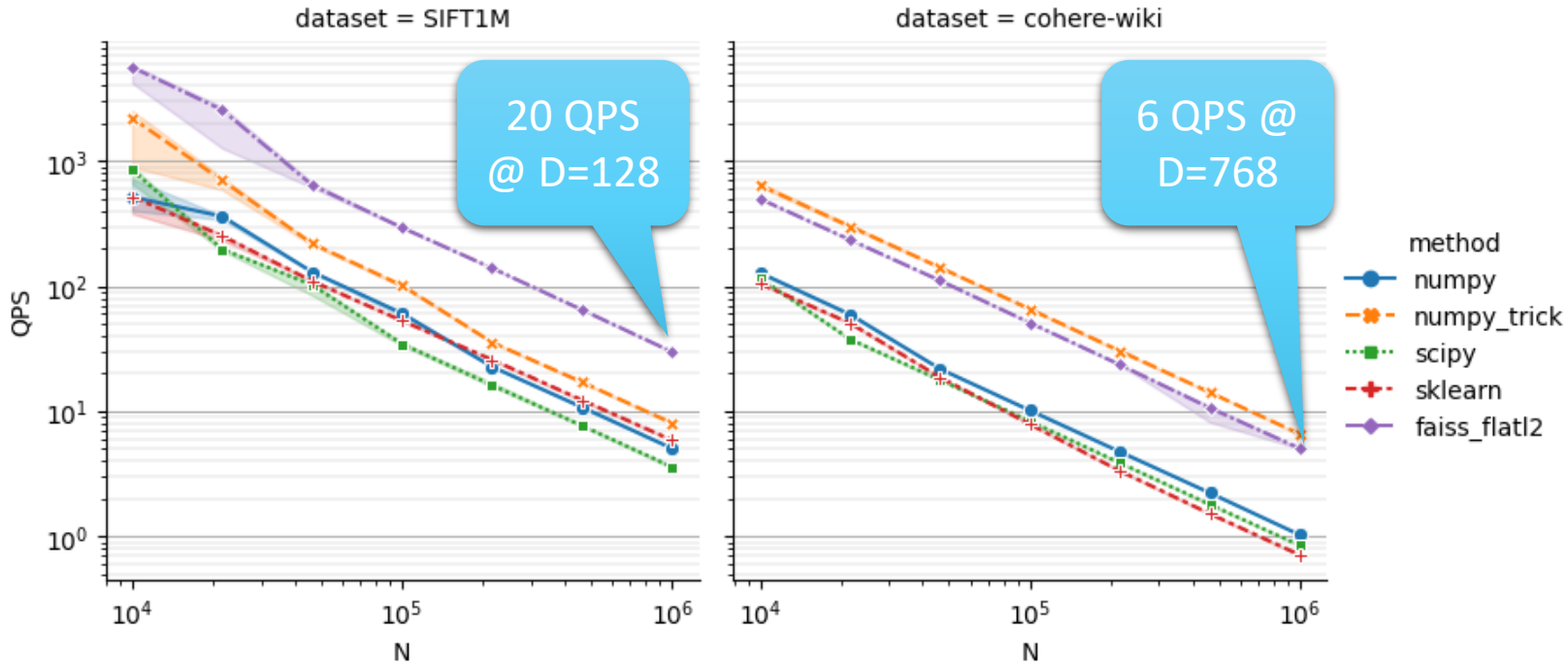
- Store vectors in table.
- To query: scan table, compute distances, return top-k
  - With priority queue or just sorting
- Really? Really!
  - Exact search (fully accurate)
  - Very fast updates, no rebuilds
  - Filtered search is easy
  - Fast GPU implementations
  - No extra memory

FLAT INDEX

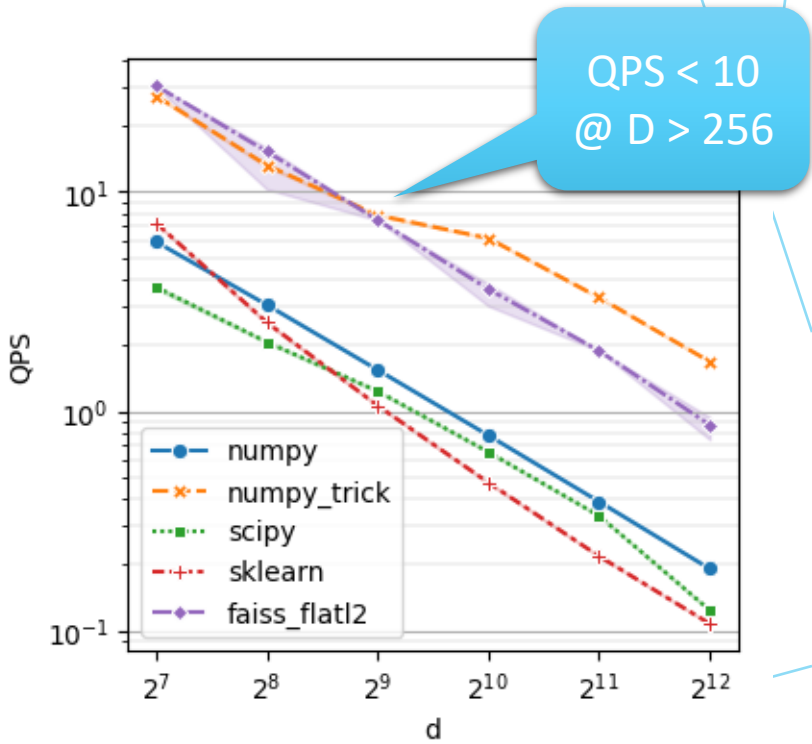
- Store vectors in table.
- To query: scan table, compute distances, return top-k
  - With priority queue or just sorting
- Really? Really!
  - Exact search (fully accurate)
  - Very fast updates, no rebuilds
  - Filtered search is easy
  - Fast GPU implementations
  - No extra memory
- Good for **small** collections (< 100K vectors)
  - Or more!
    Neos [Huang, ICDE'24]: 1M vectors @ 1500 QPS with 4 GPUs off SSD

SINGLE CORE PERFORMANCE @ 1M

- AMD EPYC 7302 at 3GHz, AVX 2
- FAISS using inner-prod trick

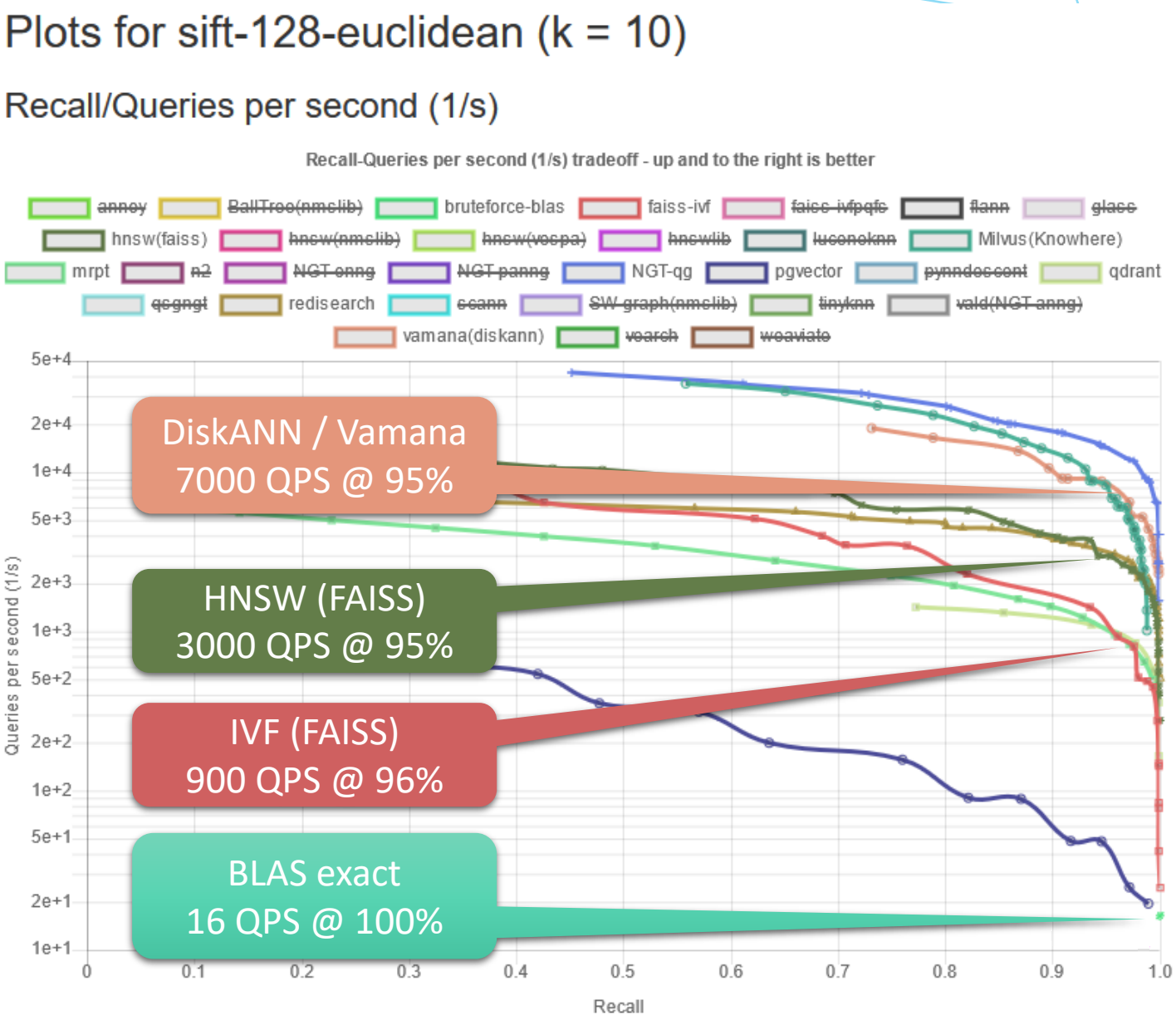


- Synthetic Gaussian data



ANN-BENCHMARKS

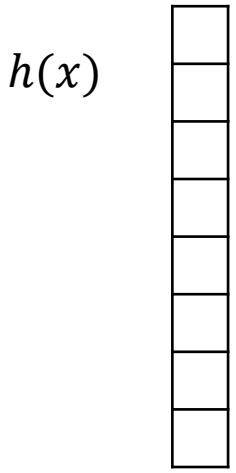
- AWS r6i.16xlarge (3rd gen Xeon)
- SIFT 1M
- Benchmark by ann-benchmarks.com
- Exact search with BLAS: **16 QPS**
- IVF (clustering):
  → **x50 time faster**
- Graph index:
  → **×100 – ×500 times faster**



LSH: LOCALITY SENSITIVE HASHING

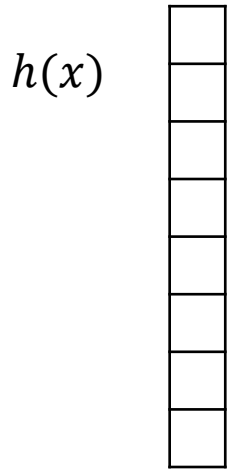
NORMAL HASH

- Different items → different hash
  - Similar but not identical? Different hash
- Minimize collisions.



LOCALITY SENSITIVE HASH

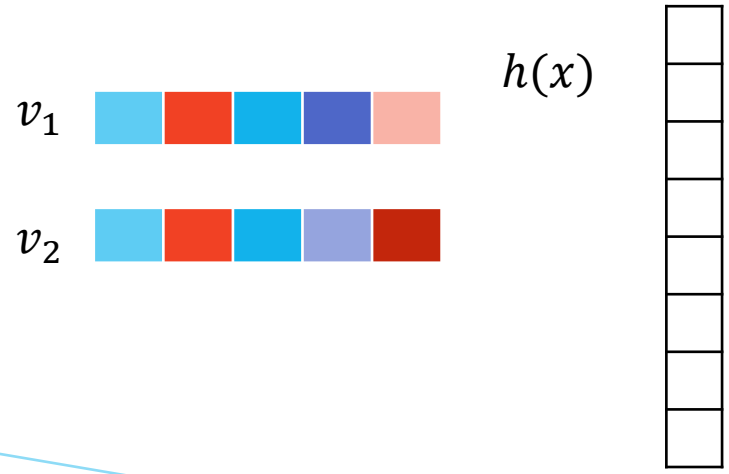
- Hash depends on location
- Near items → same hash
- **Maximize** collisions



LSH: LOCALITY SENSITIVE HASHING

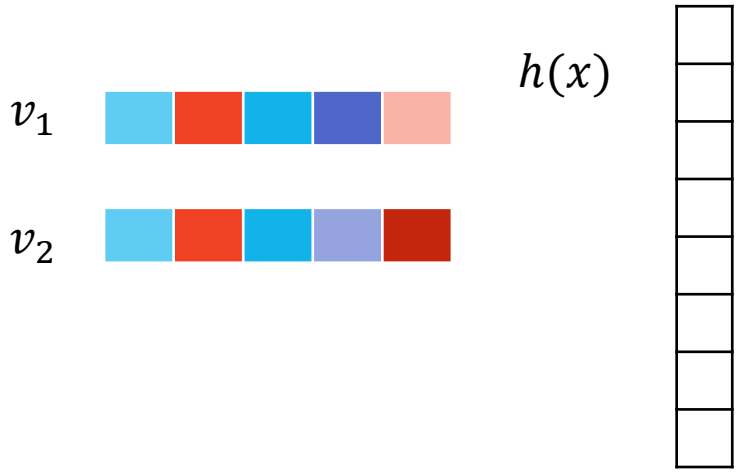
NORMAL HASH

- Different items → different hash
  - Similar but not identical? Different hash
- Minimize collisions.



LOCALITY SENSITIVE HASH

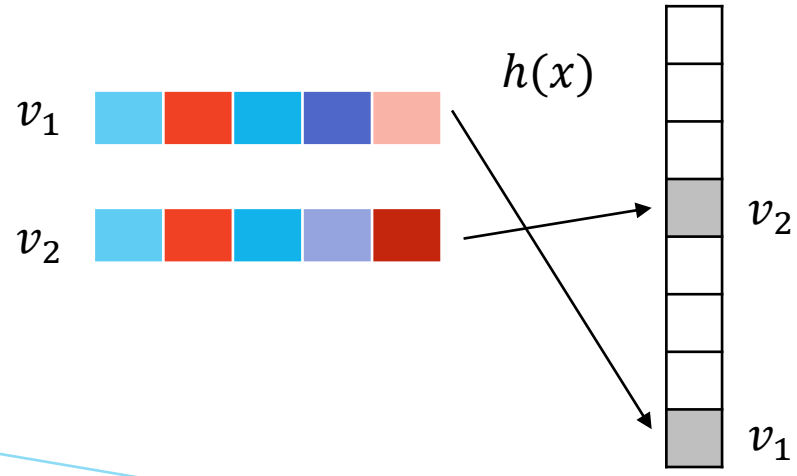
- Hash depends on location
- Near items → same hash
- **Maximize** collisions



LSH: LOCALITY SENSITIVE HASHING

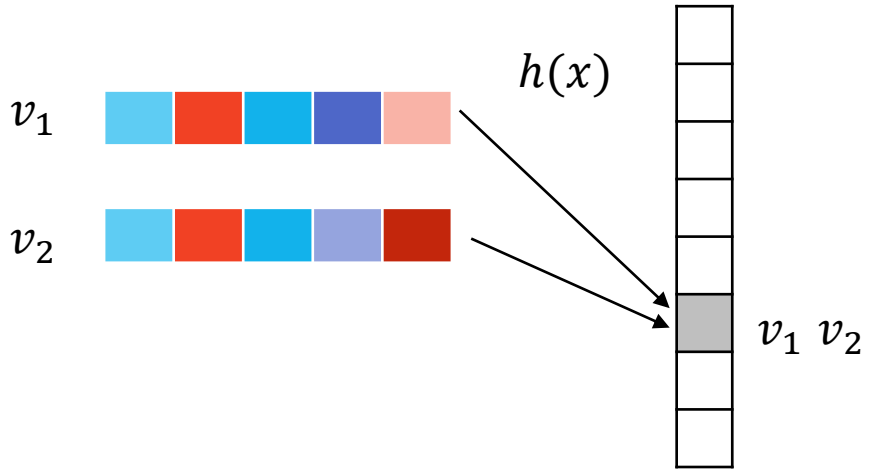
NORMAL HASH

- Different items → different hash
  - Similar but not identical? Different hash
- Minimize collisions.



LOCALITY SENSITIVE HASH

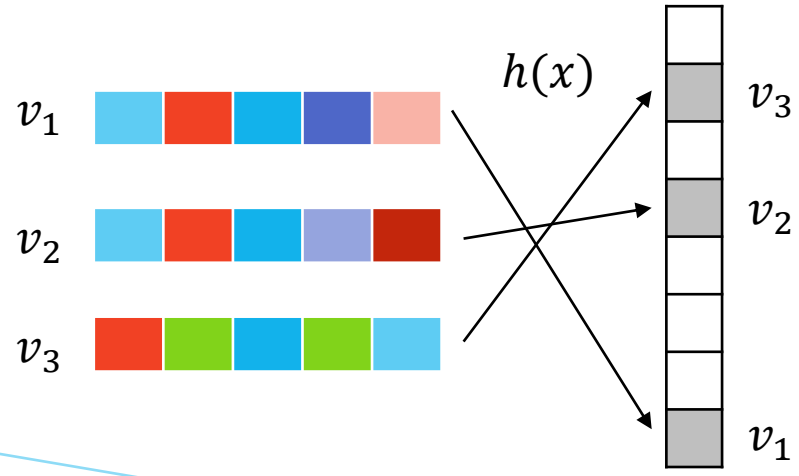
- Hash depends on location
- Near items → same hash
- **Maximize** collisions



LSH: LOCALITY SENSITIVE HASHING

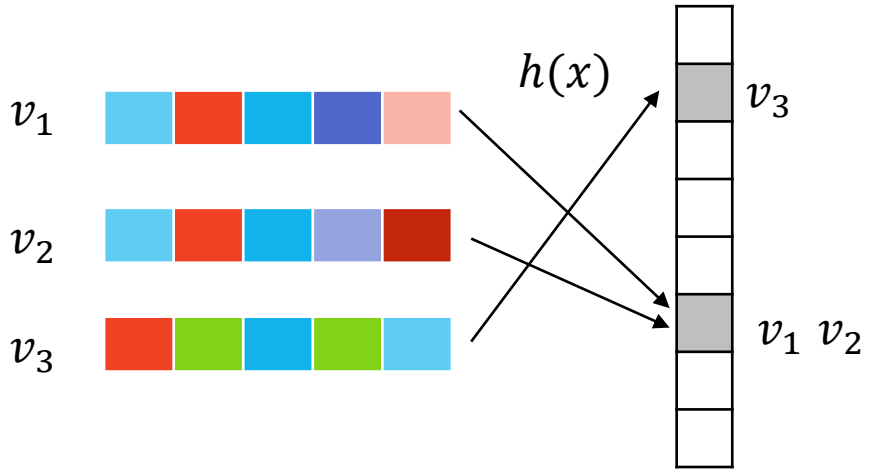
NORMAL HASH

- Different items → different hash
  - Similar but not identical? Different hash
- Minimize collisions.



LOCALITY SENSITIVE HASH

- Hash depends on location
- Near items → same hash
- **Maximize** collisions



ANNS WITH LSH

Suppose we have $h(\cdot)$ where

- u near v → $h(u) = h(v)$
- u far from v → $h(u) \neq h(v)$

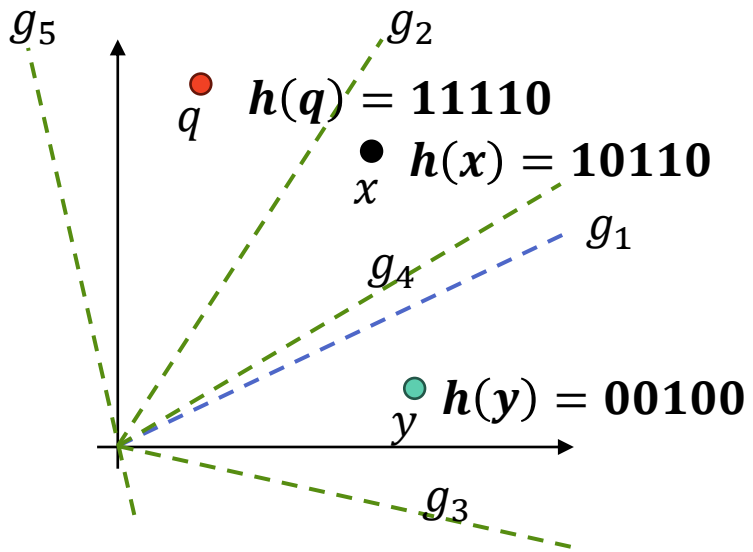
} with high probability

To insert x:

- Add x to bucket $h(x)$

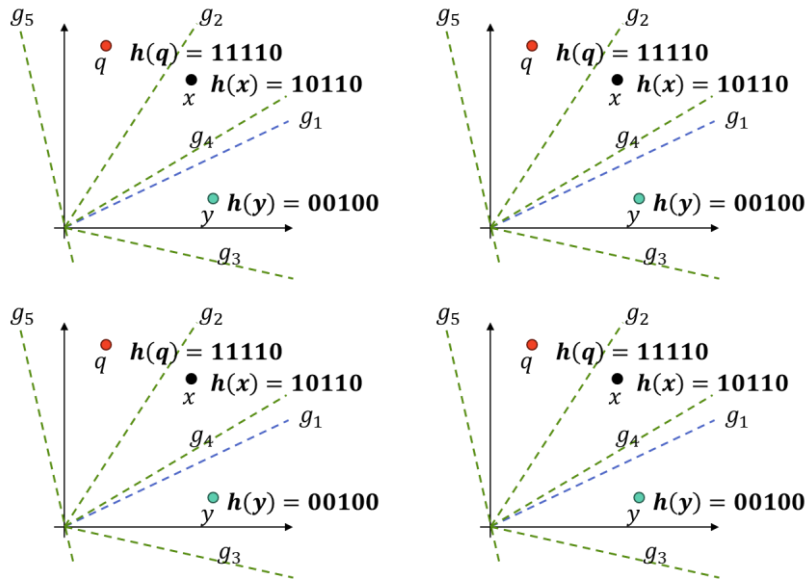
To query for q:

- Compare q to vectors in $h(q)$ bucket
  - Ideally: nearby vectors → same hash bucket
- Return k nearest (rerank or use prio-queue)



NEAREST NEIGHBOUR WITH LSH

- One $h(x)$ not enough
- **Repeat** to reduce error probability
  - L hash tables, each with different random $h(x)$
  - Insert vector into L tables
  - Query in L tables
  - Rerank
- Downside: larger query time
- Set L to achieve desired error bounds
  - Or based on memory, and estimate bounds
  - Both depend on *N*, $h(x)$



Vast literature:

- Finding good $h(\cdot)$ for different d, data
- Accuracy, memory guarantees
- Searching buckets

LSH PROPERTIES

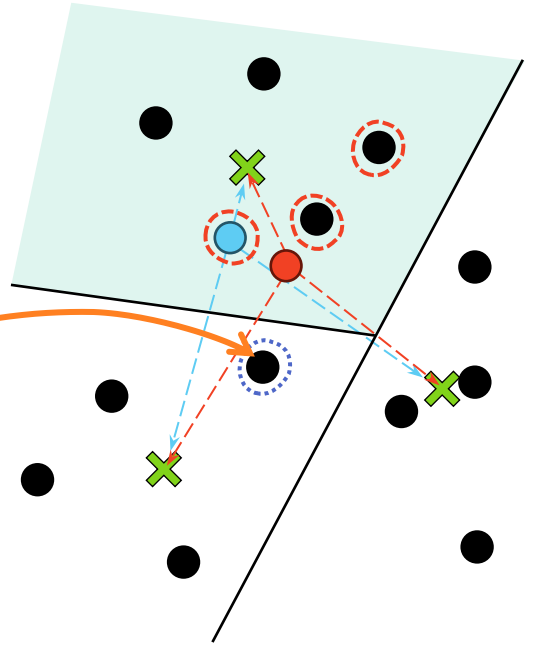
- Many variants:
  - E^2LSH – classic random projection LSH, works for Euclidean distances
  - IndexLSH – project to binary, search with Hamming distance [FAISS, 2023]
  - FALCONN – cosine similarity, needs rebuilds [Andoni, NeurIPS'15]
  - SPANN – learn $h(x)$, disk resident, needs rebuilds [Chen, NeurIPS'21]
- LSH is theoretically elegant…
  - ✓ Fast incremental updates
  - ✓ **Bounded error**, memory (one of very few)
  - ✓ No rebuilds (data-independent)
- **… but seldom actually used in VDBMS**
  - Inferior performance in all aspects [Briggs, 2024]
  - Too slow when D > 128

Research continues…

IVF: INVERTED FILE INDEX

(also called clustering)

- Choose number of buckets (clusters) k
- Cluster vectors (e.g., k-means)
- To index v ●
  - Find nearest centroid
  - Add v to cell (cluster)
- To query q ●
  - Find nearest centroid
  - Compare q to vectors inside cell

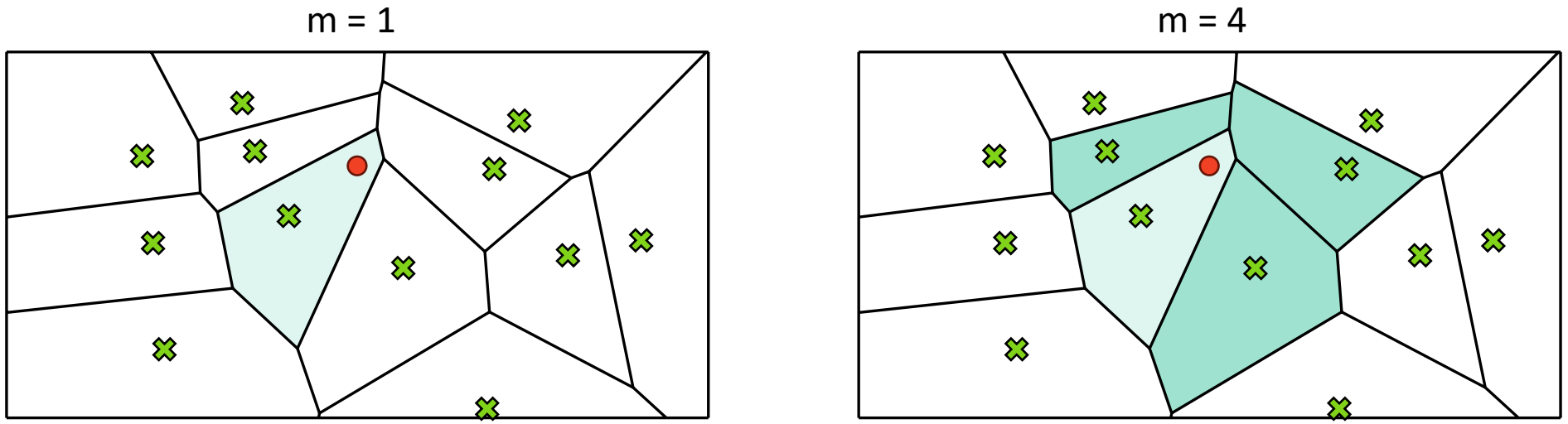


Problem:
missed neighbour
in other cluster

NEIGHBOUR IN OTHER CLUSTER?

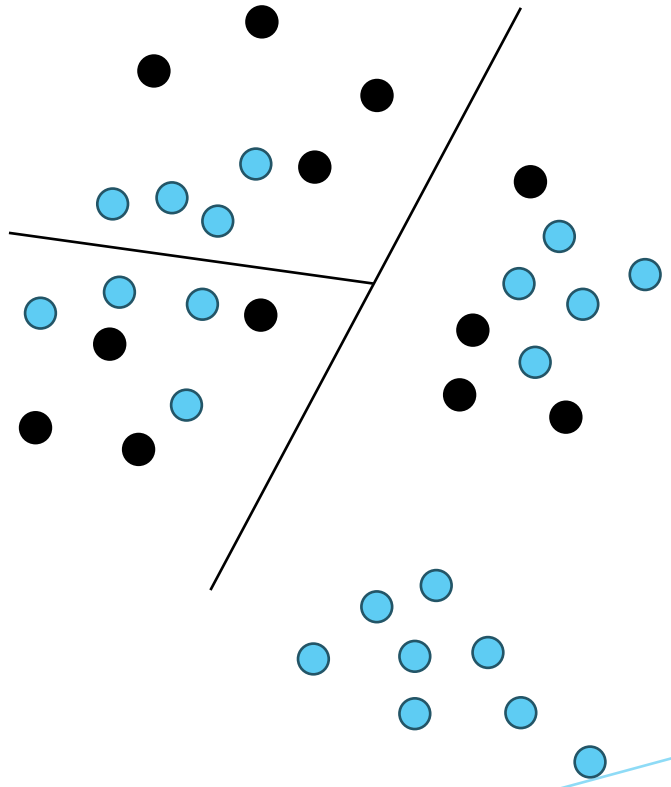
→ List vectors near edges in two cells

→ Increase search scope: probe *m* > 1 nearest cells



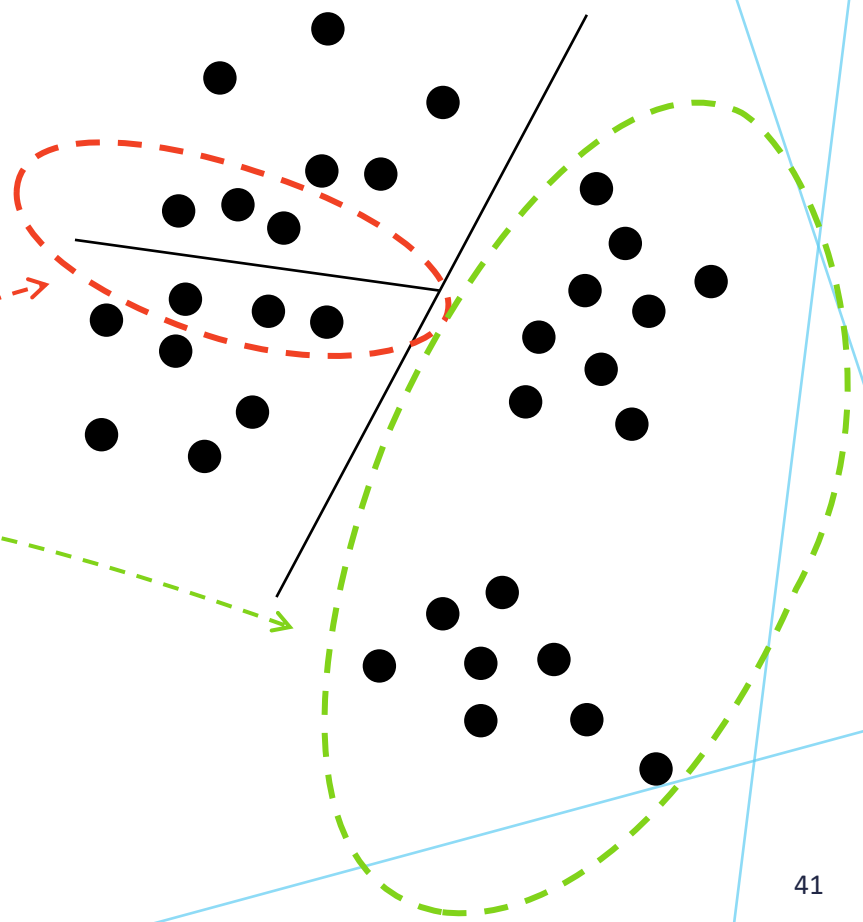
IVF OVER TIME

- Add some points.
  - Cluster selected by nearest centroid
- Add more points…
- Even more!



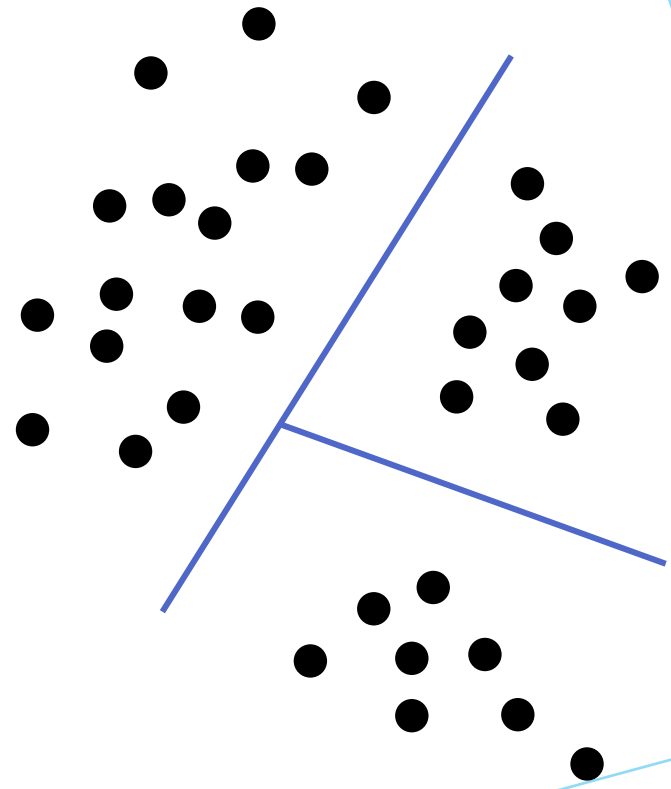
IVF OVER TIME

- Add some points.
  - Cluster selected by nearest centroid
- Add more points…
- Even more!
- IVF cluster do not match data
  - More error near edges
  - Unbalanced cluster lists



IVF OVER TIME

- Add some points.
  - Cluster selected by nearest centroid
- Add more points…
- Even more!
- IVF cluster do not match data
  - More error near edges
  - Unbalanced cluster lists
- Solution: **rebuild clusters periodically**
  - Or update clusters incrementally [Xu, SOSP'23]



IVF PROPERTIES

PARAMETERS

- k number of cells
  - Higher k → sparser cell → faster search
- m number of probes
  - High m → more accuracy, slower search

PROPERTIES

✓ Low memory overhead

✓ Fast updates
  - New vectors immediately visible

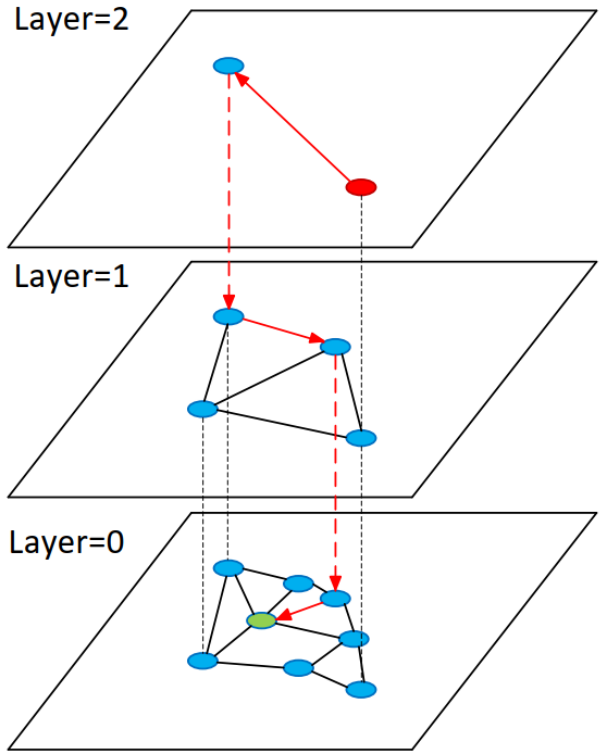
✓ Easy to scale (higher k)

• Reasonable performance

× Needs rebuilding
  - Adding vectors degrades accuracy
  - Recompute clusters periodically

HNSW [MALKOV, TPAMI'20]

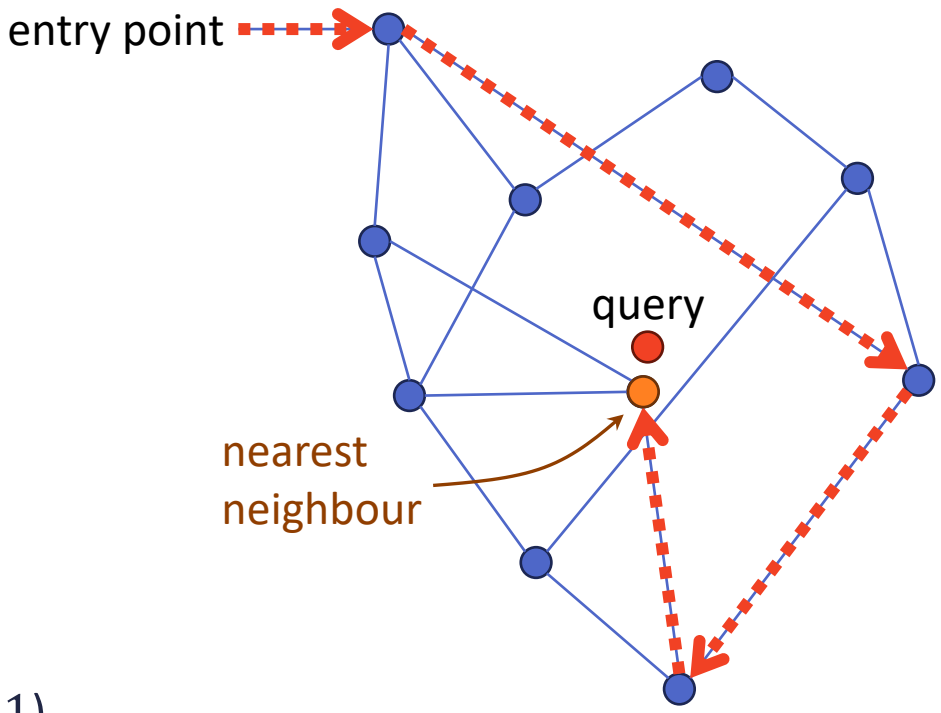
- **H**ierarchically **N**avigable **S**mall **W**orlds
- Crown jewel of ANN indexes
  - Near SotA speed-accuracy tradeoff
  - Available quality implementations
  - Used in Qdrant, Weaviate (custom variants)
  - … not quite SotA in academia
- Combines two ideas:
  1. Navigable Small World:
     graph for traversal with greedy search
  2. Hierarchical skips:
     higher layers allow fast skips



NAVIGABLE SMALL WORLD (NSW)

- Class of graphs:
  - Add long- and short-range edges
  - Characteristic path length $O(\log N)$
- Greedy search (DFS):
  - Start at entry point
  - Add its neighbours to candidate list
  - Go to candidate nearest to query
  - Repeat until no such candidate
- Search path is $O(\log N)$
- **Problem**: average out-degree $O(\log N)$

  → Polylogarithmic $O(\log^C N)$ search time ($C > 1$)



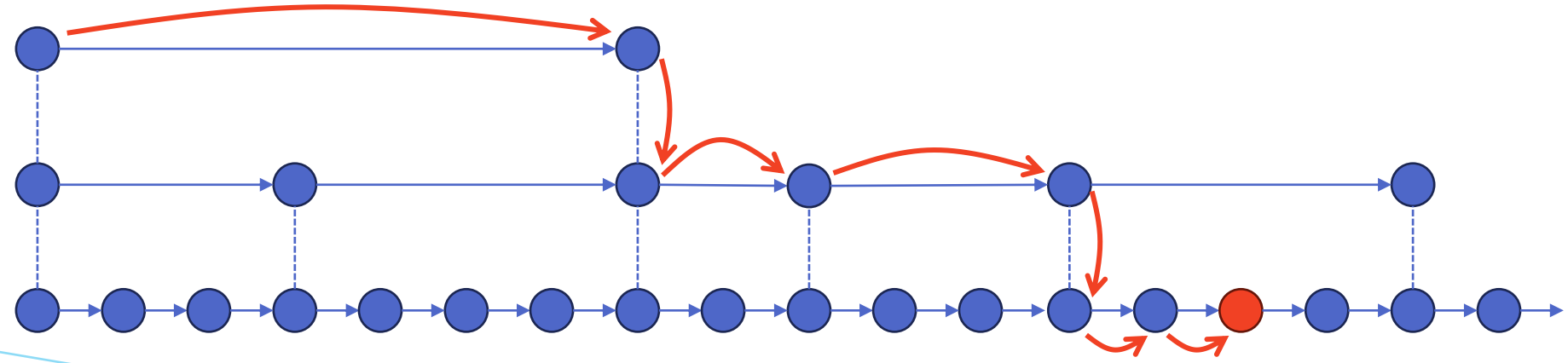
SKIP LIST

SEARCH

- Start at top layer
- Search in current layer.
- When cannot continue, move to lower layer.

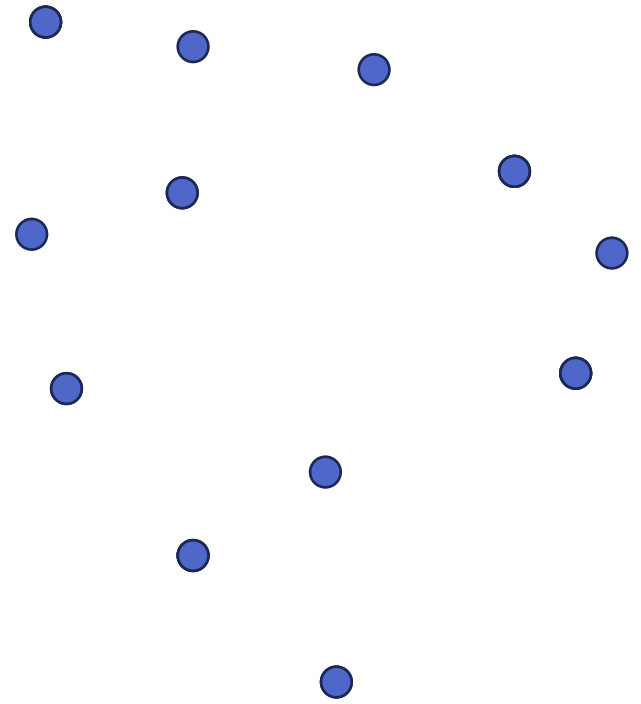
INSERT

- Insert to bottom list
- Flip coin, if heads stop.
- Otherwise, move to higher layer and insert

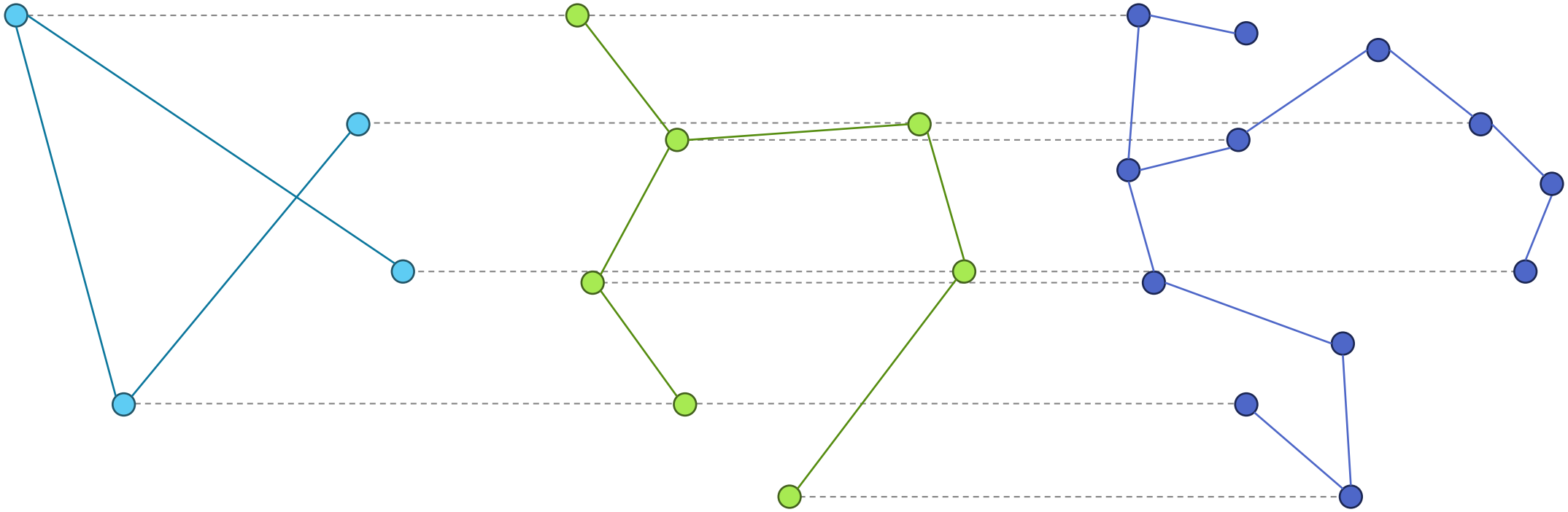


HNSW = NSW + SKIP LIST

- Hierarchical graphs
  - Replicate some points across layers
  - Make long edges at higher layers
  - Make short edges at lower layers
- Out-degree bounded by construction
  - Parameter M_{max}

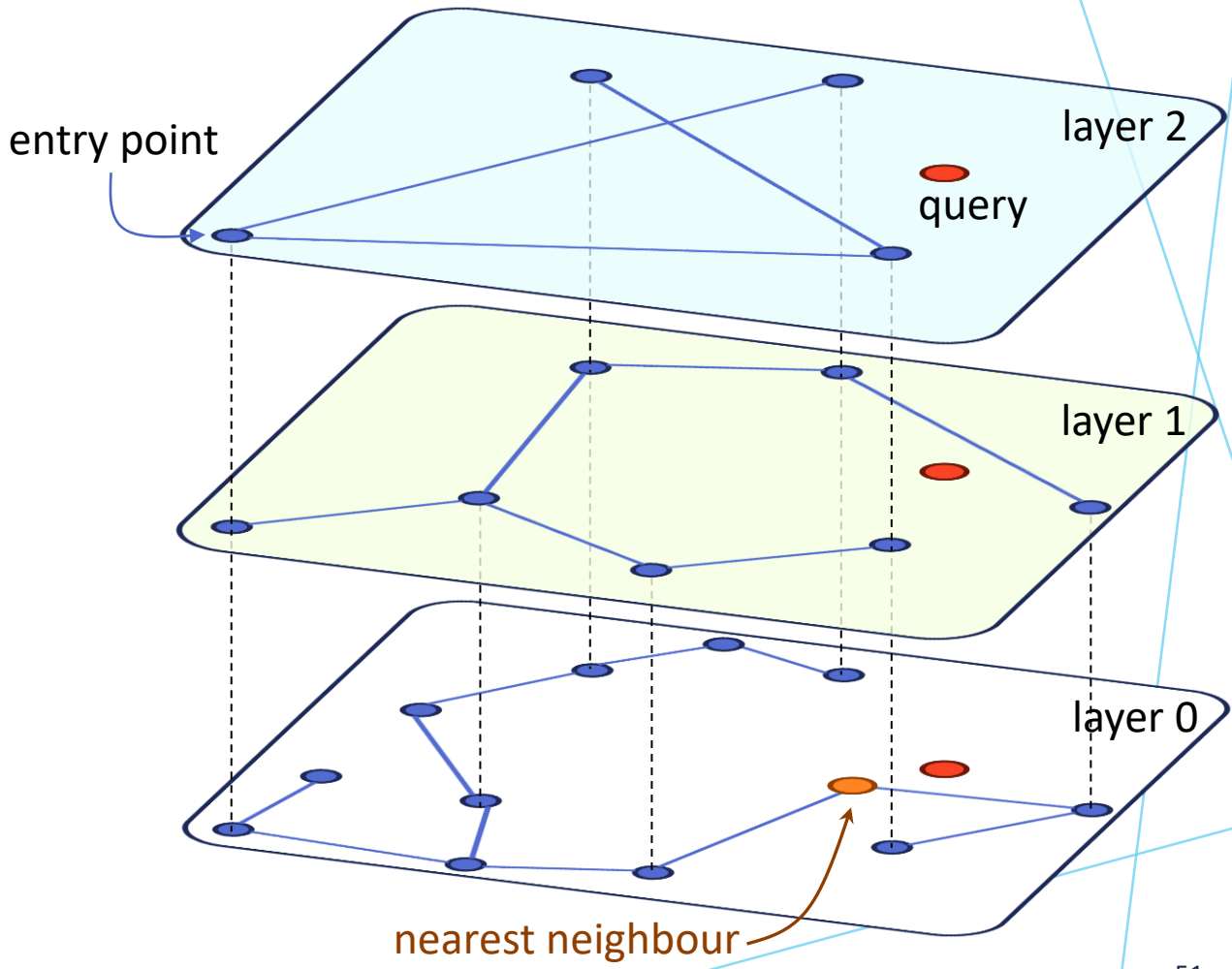


HNSW = NSW + SKIP LIST



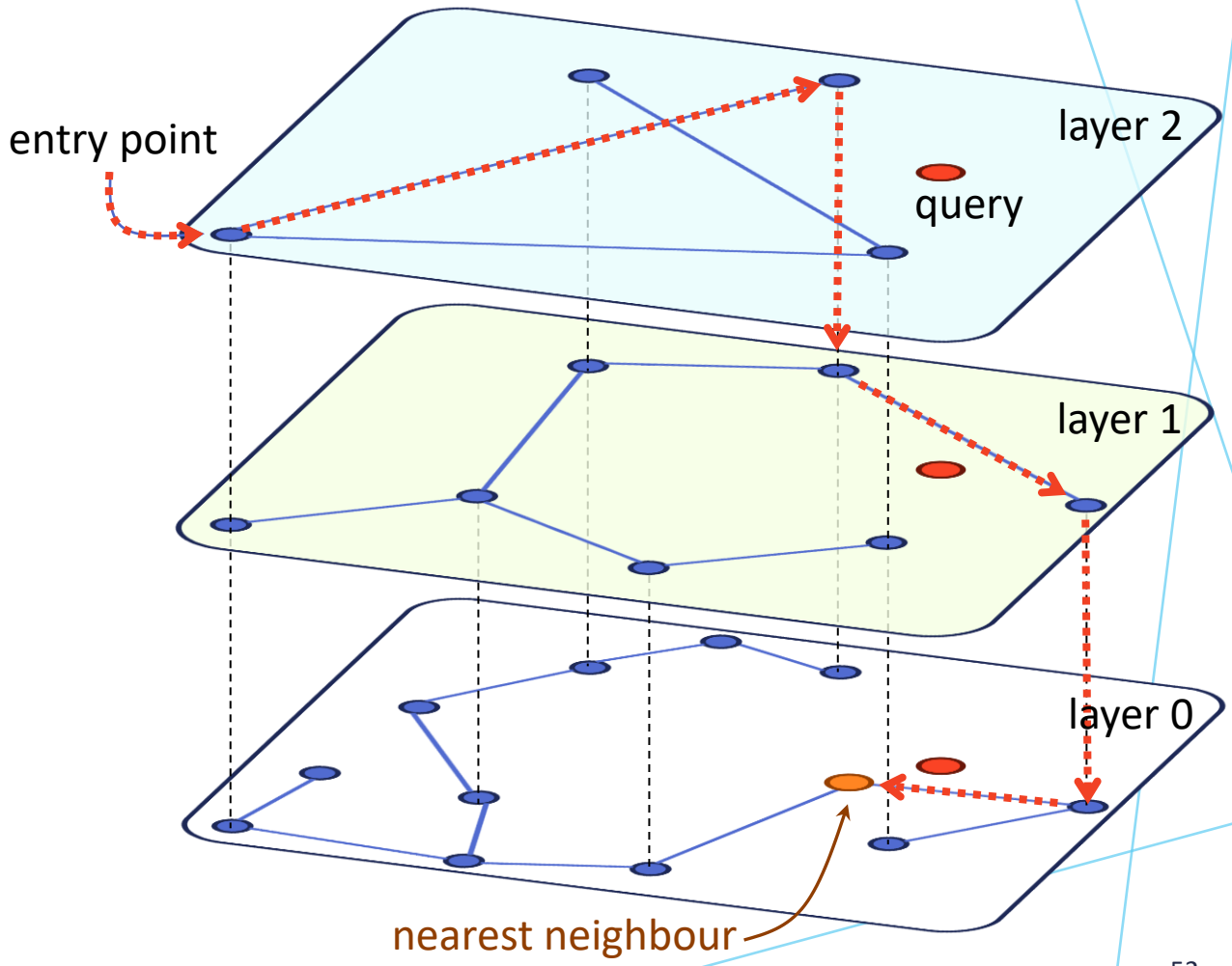
HNSW = NSW + SKIP LIST

- To query:



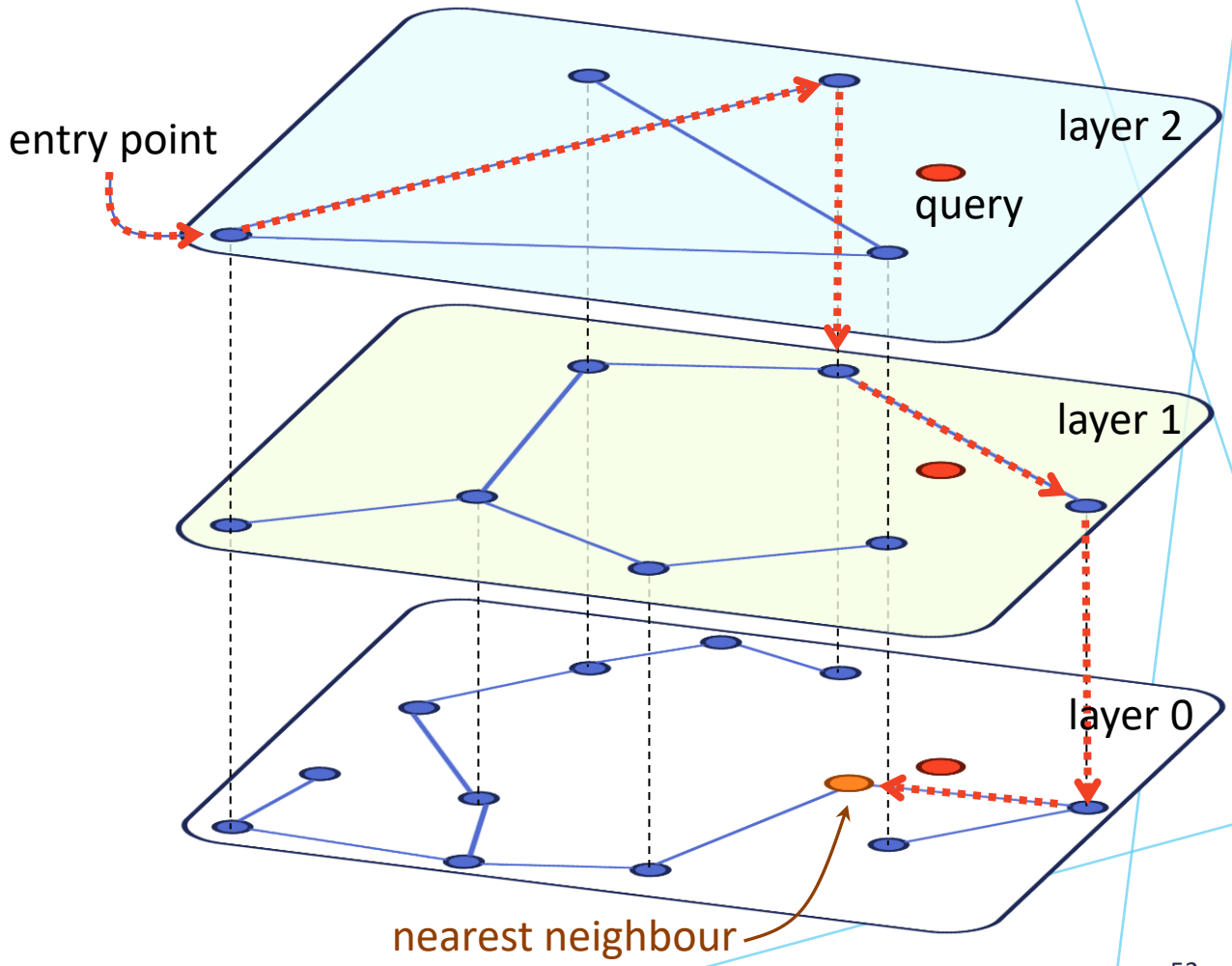
HNSW = NSW + SKIP LIST

- To query:
  - Enter at top layer
  - Greedy search
    - Move to nearest connected neighbour
  - Done? Move to lower layer



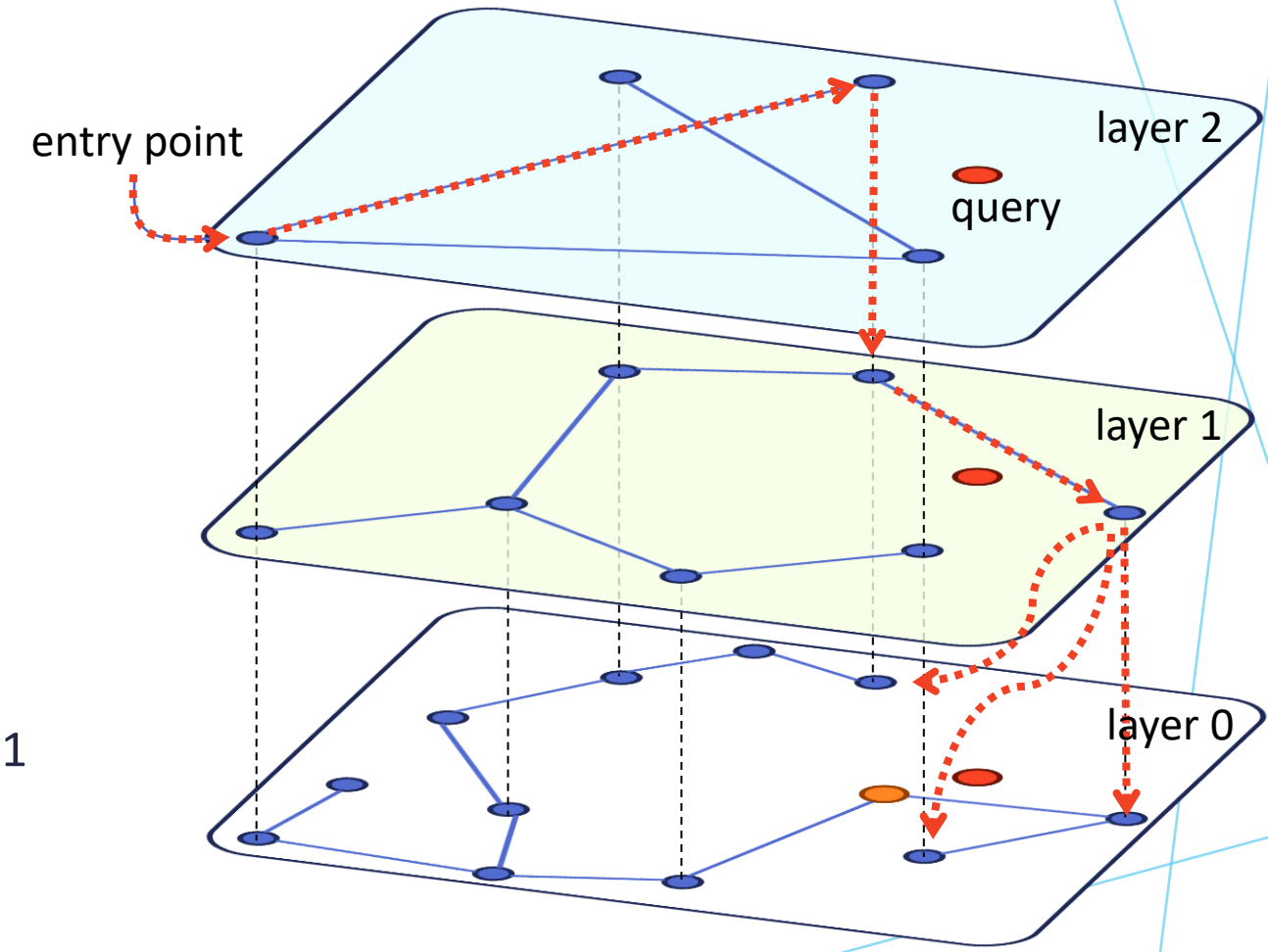
HNSW = NSW + SKIP LIST

- To query:
  - Enter at top layer
  - Greedy search
    - Move to nearest connected neighbour
  - Done? Move to lower layer
- Result: $O(\log N)$ search
  - Because out-degree bounded



HNSW = NSW + SKIP LIST

- To query:
  - Enter at top layer
  - Greedy search
    - Move to nearest connected neighbour
  - Done? Move to lower layer
- Result: $O(\log N)$ search
  - Because out-degree bounded
- Improve recall (incurs overhead):
  - At layer 0, expand search to *efSearch* > 1 neigbours



INSERTION (OVERVIEW)

To insert vector x:

1. Choose highest layer for x **(randomly)**
2. Add x to each layer, one at a time:
   - Do greedy search in the layer
   - Create node and connect to neighbours in layer (based on greedy search)

Some details:

- Search output used as entry point of next layer (same with query)
- Expand: multiple entry points → better recall (same with query)
- Trim trim edge lists → avoid inflating out-degree

INSERTION (MORE DETAILS)

To insert x:

- Select highest layer L for x
  - Sample L ~ LogUniform
- Find candidate entry point p
  - Greedy search for x stops, stop at layer L+1
- Find W = *efConstruction* closest neigbours of p to x
- Phase 2:
  - Insert x at this layer
  - Connect x to M best neighbours in W
  - Trim edges of W to M_{max}
  - Move to lower layer, W as set of candidate points

More candidates → better accuracy

M parameter

prevent out-degree from growing due to multiple inserts

PROPERTIES OF HNSW

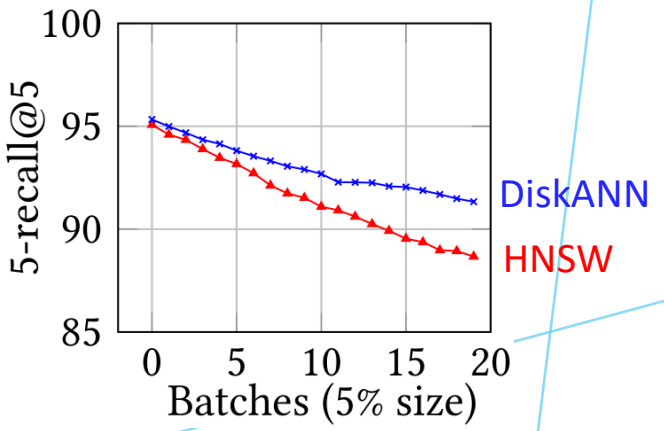
YAY! ☺

- ✓ Excellent accuracy
- ✓ Very fast
- ✓ Near SotA for speed + accuracy
- ✓ Good for relatively static data
- ✓ Less memory than LSH
  - Reasonable for many datasets
- ✓ Incremental updates possible…
  - In theory

OH NOES! ☹

- × Higher memory than IVF, PQ
  - Addressed
- × Insert slower than query
- × Delete could disconnect graph
  - Uses tombstones

→ Updates **ballon memory** or **degrade accuracy**

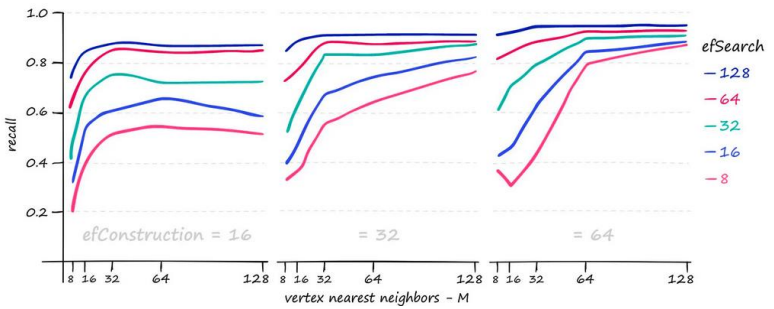


[Singh, arXiv '21]

HNSW CONFIGURATION

		Affects…			
Parameter	**Description**	**query time**	**insert time**	**memory**	**recall**
M M_{max}	nearest neighbours added per vertex max out-degree (often set M_{max} = M)	✓	✓	✓	✓
efSearch	how many searches in query last phase	✓			✓
efConstruction	how many entry points when building index	(if *M* high)	✓		(if *M* low)

- Complex tradeoffs!
  - Many parameters
  - Interplay between them
- How to select?
  - Tune manually
  - Optimizer (if available)
  - Rules of thumb and guides



From Pinecone documentation [Briggs, 2024]

efConstruction does not affect recall beyond reasonable value … except if *M* is very low, then *efConstruction* can compensate

RECAP

Index	Good	Bad
Flat	Exact Low memory Fast updates No rebuilds	Not feasible for large datasets (> 10K)

RECAP

Index	Good	Bad
Flat	Exact Low memory Fast updates No rebuilds	Not feasible for large datasets (> 10K)
IVF	**Decent speed** **Good accuracy** **One parameter**	**Not fastest** **Needs rebuilds**

RECAP

Index	Good	Bad
Flat	Exact Low memory Fast updates No rebuilds	Not feasible for large datasets (> 10K)
IVF	**Decent speed** **Good accuracy** **One parameter**	**Not fastest** **Needs rebuilds**
HNSW	**Excellent speed** **Excellent accuracy**	**Needs rebuilds** **Memory ballooning** **More parameters**

RECAP

Index	Good	Bad
Flat	Exact Low memory Fast updates No rebuilds	Not feasible for large datasets (> 10K)
IVF	**Decent speed** **Good accuracy** **One parameter**	**Not fastest** **Needs rebuilds**
HNSW	**Excellent speed** **Excellent accuracy**	**Needs rebuilds** **Memory ballooning** **More parameters**
LSH	Bounded error Fast updates No rebuilds	Not feasible for large vectors (>128) High memory Inferior performance (very rarely used)

To choose, consider requirements:

- Dataset size
- Vectors length
- Update frequency
  - Dynamicity
- Update visibility
  - Freshness
- Speed
  Recall
  Memory

WE JUST TOUCHED THE SURFACE

Structure	Index[a]	Partitioning	Residence	Complexity[b]			Update[c]	Error Bound
				Constr	Space	Query		
Table	E^{2}LSH [47]	Space	Mem	Med.	High	Med.	Y	✓
	FALCONN [30]	Space	Mem	Med.	High	Med.	R	✓
	*SQ	Discrete	Mem	Med.	Low	Med.	Y	✗
	*PQ	Clustering	Mem	Med.	Low	Med.	R	✗
	*IVFSQ	Clustering	Mem	Med.	Low	Med.	R	✗
	*IVFADC [66]	Clustering	Mem	Med.	Low	Med.	R	✗
	SPANN [43]	Clustering	Disk	Med.	Med.	Med.	R	✗
Tree	FLANN [92]	Space	Mem	High	High	Low	R	✗
	RPTree [45, 46]	Space	Mem	Low	High	Low	R	✓
	*ANNOY	Space	Mem	Low	High	Low	R	✗
Graph	NN-Descent (KGraph) [51]	Proximity	Mem	Med.	Med.	Med.	N	✗
	EFANNA	Proximity	Mem	High	Med.	Low	N	✗
	FANNG [64]	Proximity	Mem	High	Med.	Med.	N	✗
	NSG [56]	Proximity	Mem	High	Med.	Low	N	✗
	Vamana (DiskANN) [111]	Proximity	Disk	Med.	Med.	Low	N	✗
	*HNSW [85]	Proximity	Mem	Low	Med.	Low	Y	✗

Not today

from [Pan, VLDBJ '24]