

M2

Hashing: From Data to Algorithms

A hash function maps a key to a fixed-size bucket id, so you can find or insert in expected $O(1)$ by reading the page where the bucket lives.

Integer hashing: $h(x) = x \% 10$

Same modulo, different inputs, same bucket = a collision

Input $\rightarrow h(x)$

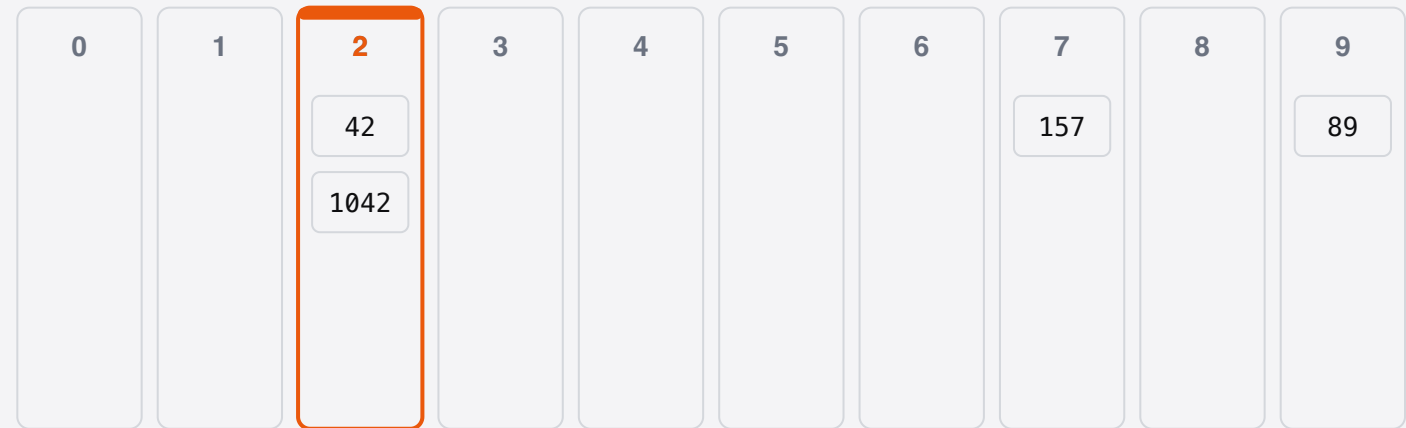
$$42 \% 10 = 2$$

$$157 \% 10 = 7$$

$$1042 \% 10 = 2$$

$$89 \% 10 = 9$$

Buckets [0–9]

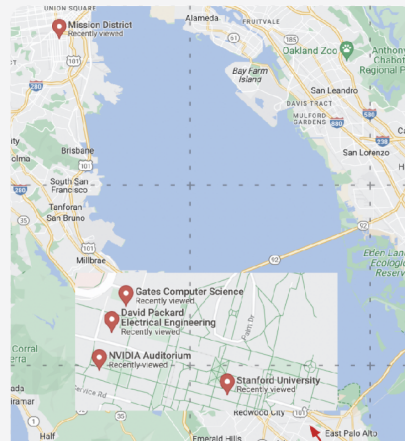


$h(x) = x \% 10$ maps four integers into ten buckets numbered 0 through 9. 42 and 1042 both land in bucket 2 (orange) because both end in 2, while 157 and 89 fall in distinct grey buckets. The collision is the point: an index must handle two or more keys landing in the same bucket. Chaining stores a list per bucket.

Downloaded from <http://ajph.org/> on November 10, 2015

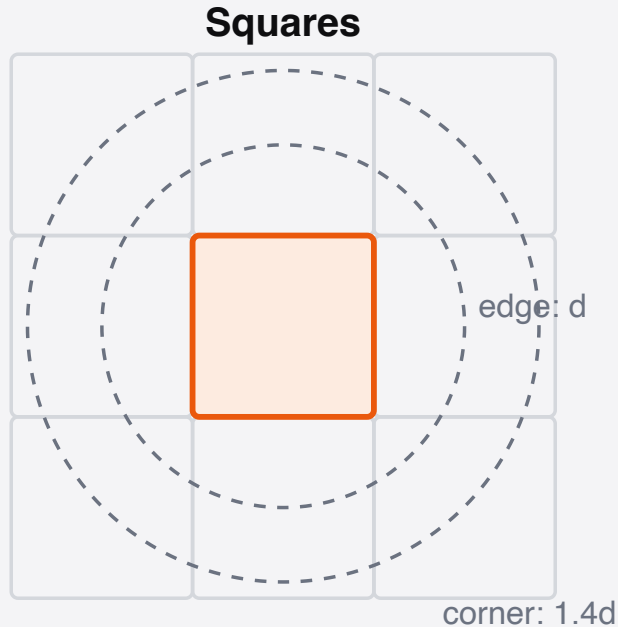
CLOSED NANO QUIZ · REVIEW

A map of the San Francisco Peninsula. A large white rectangle highlights the area around San Francisco. Within this rectangle, four red location pins are marked with the following text: 'Google Computer Science recently viewed', 'David Packard Electrical Engineering Recently viewed', 'NVIDIA Auditorium recently viewed', and 'Stanford University Recently viewed'. Other labels on the map include 'Mission District recently viewed', 'Alameda', 'Hay Farm Island', 'Oakland Zoo', 'Anthony Chabot Regional Park', 'San Leandro', 'WU FORD JARDINES', 'San Lorenzo', 'Castroville', 'Hayward', 'Elgin Landing Ecological Reserve', 'Redwood City', 'Emeryville', 'Half Moon Bay', 'San Bruno', 'San Francisco', 'Millbrae', 'Corral Point', and 'East Palo Alto'. Major highways like I-580, I-805, I-80, and SR-92 are shown. The map also shows the San Francisco Bay and surrounding areas like Marin and Contra Costa counties.

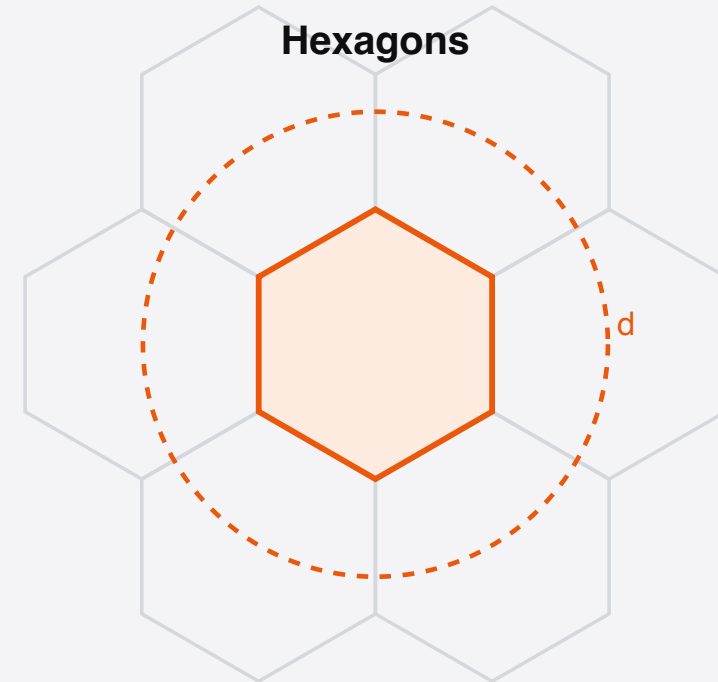


Why hexagons, not squares?

A radius query reads a cell plus its neighbour ring. The ring is only even if the neighbours are.



8 neighbours, TWO distances



6 neighbours, ONE distance

Cell shape determines the neighbor ring. A square has eight neighbors at two distances: four edge-neighbors one cell away, four corner-neighbors farther by a factor of root two. A hexagon has six neighbors, each at the same distance from the center, which yields a symmetric ring for a radius query. Hexagons approximate a circle better than squares.

Nearby points share a cell, far points do not

Three Stanford buildings vs SF Mission, computed with H3. Campus shares a cell at res 8 and splits at res 12.

Resolution 8 · zoomed out

NVidia	8828347417fffff	}	campus: one cell, one code
Packard	8828347417fffff		
Gates	8828347417fffff		
SF Mission	88283082a1fffff		29 mi away: a different cell

Resolution 12 · zoomed in

NVidia	8c28347416 257 ff	}	campus: three cells now
Packard	8c28347416 02b ff		
Gates	8c28347416 021 ff		
SF Mission	8c283082a06bbff		still a different cell

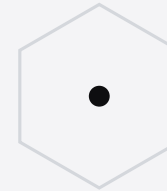
H3 codes for three Stanford campus buildings and SF Mission. At resolution 8 the three campus buildings map to 8828347417fffff and SF Mission, 29 miles away, maps to a different cell. At resolution 12 the three campus buildings split into three cells, differing in the trailing digits (orange), and SF Mission remains in a different cell.

Step 1, the hash: a location becomes a cell code

Every order takes the code of the hexagon it falls in. Nearby orders land in the same hexagon.



88283082a1ffff

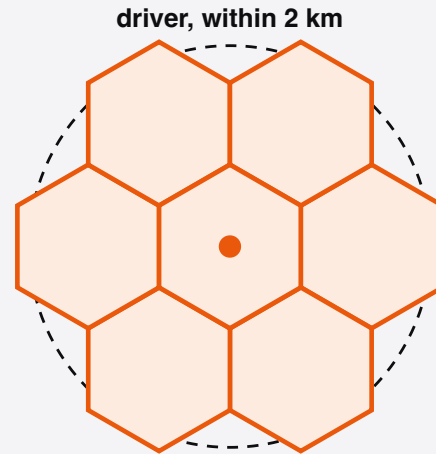


SF order, far away

H3 maps each food-delivery order's latitude and longitude to a cell ID. The system stores 10 million orders by that ID, so nearby orders share one and distant orders fall under different ones.

Step 2, the lookup: a radius query is a few cell reads

At this resolution, one ring means seven reads before the exact distance check.



the radius crosses borders: + 6 neighbours



7 cell IDs, maybe 7 different pages.

7 cell IDs, then check candidate distances.

At a resolution where one neighboring ring covers 2 km, the query reads the driver's cell and its six adjacent cells. Those seven cell IDs can map to seven different pages, yielding up to seven reads. The system then checks exact distances for the candidate food-delivery orders. The number of cell lookups stays constant as the table grows from ten thousand orders to ten million; a larger search radius just reads a few more rings.

M2

Compression Basics: Making Data Smaller

Columnar compression (run-length, dictionary, delta)
exploits repetition inside one column, typically shrinking on-disk size 10× thereby cutting IO by the same factor.

Four ways to shrink a column

Same idea: a column repeats, so stop repeating it. Pick by the data.

RLE

RAW

Mickey,Mickey,Mickey,
Daffy,Daffy,Daffy,Daffy

ENCODED

(Mickey, 3) (Daffy, 4)

Ratio **10–100x**

Speed very fast

BEST FOR

Sorted, repeated
values

Dictionary

RAW

rock, pop, rock, jazz

ENCODED

0, 1, 0, 2

+ dict {rock:0, pop:1, jazz:2}

Ratio **5–20x**

Speed fast

BEST FOR

Low-cardinality
strings

Delta

RAW

1700000000, 1700000001,
1700000003, 1700000004

ENCODED

1700000000, +1, +2, +1

Ratio **5–10x**

Speed fast

BEST FOR

Sorted numbers,
timestamps

Bit packing

RAW

5 3 4 1 (32-bit ints)

ENCODED

101 011 100 001

3 bits each

Ratio **4–8x**

Speed very fast

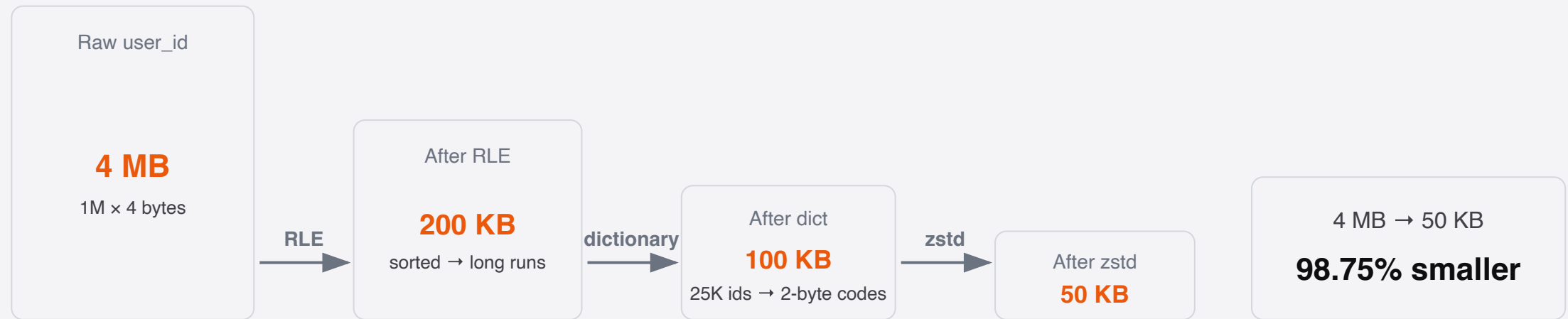
BEST FOR

Small integers,
flags

We can compress a column by removing repeated structure. RLE stores each run as a value plus a count (i.e., length of a run of Mickeys). Dictionary encoding maps each distinct value to a small integer code. Delta encoding stores differences between adjacent sorted numbers. Bit packing stores small integers in the minimum number of bits. The raw input is grey, the encoded result orange. Ratios run from about 4x (bit packing) to 100x (RLE on sorted data); choose based on the column's value distribution.

Stack the techniques and the column collapses

1M user_id values, sorted by user_id, ~25K users. Box height tracks size, not to scale.



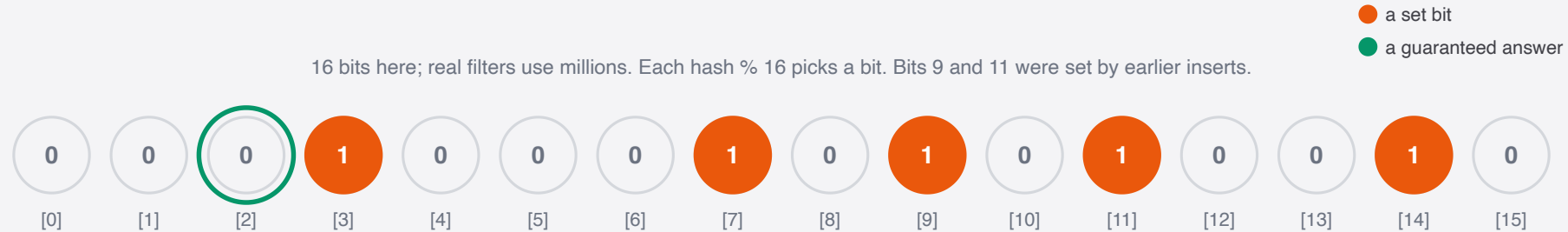
A 1M-row user_id column sorted by user_id with ~25,000 active users compresses by chaining encodings. RLE collapses the sorted runs (4 MB to 200 KB). Dictionary encoding then replaces ids with codes; 25,000 ids fit in a 2-byte code rather than a 4-byte int (to 100 KB). zstd entropy-codes the remaining byte patterns (to 50 KB), a 98.75% reduction. Unsorted data or high cardinality is less compressible.

M2

Bloom Filters

A Bloom filter stores a set in a bit array and uses k hash functions to map each item to k bit positions.

Bloom filter: a bit array plus $k = 3$ hash functions



1 INSERT "alice"

$h1("alice") \% 16 = 3 \rightarrow 1$
 $h2("alice") \% 16 = 7 \rightarrow 1$
 $h3("alice") \% 16 = 14 \rightarrow 1$

Sets bits 3, 7, 14.

The row above turns them on.

2 LOOKUP "alice"

$h1("alice") \% 16 = 3 \rightarrow 1$
 $h2("alice") \% 16 = 7 \rightarrow 1$
 $h3("alice") \% 16 = 14 \rightarrow 1$

All 1 $\Rightarrow$ probably in the set

*Could be a false positive:
other items set bits too.*

3 LOOKUP "bob"

$h1("bob") \% 16 = 3 \rightarrow 1$
 $h2("bob") \% 16 = 9 \rightarrow 1$
 $h3("bob") \% 16 = 2 \rightarrow 0$

A 0 $\Rightarrow$ definitely NOT in the set

*One 0 is proof, so there
are no false negatives.*

A Bloom filter stores a 16-bit array and $k = 3$ hash functions; earlier inserts set bits 9 and 11. (1) INSERT "alice" hashes to bits 3, 7, and 14 and sets them. (2) LOOKUP "alice" checks the same three bits: all are 1, so "alice" is probably in the set. All 1s can also be a false positive, because other items set bits too. (3) LOOKUP "bob" checks bits 3, 9, and 2: bit[2] = 0 guarantees "bob" is not in the set (green). Any 0 means "definitely not in the set", so false negatives never occur. A bigger bit array lowers the false-positive rate at the cost of space; k is tuned to the array size and item count. In general, a Bloom filter checks membership in RAM before paying an expensive IO.

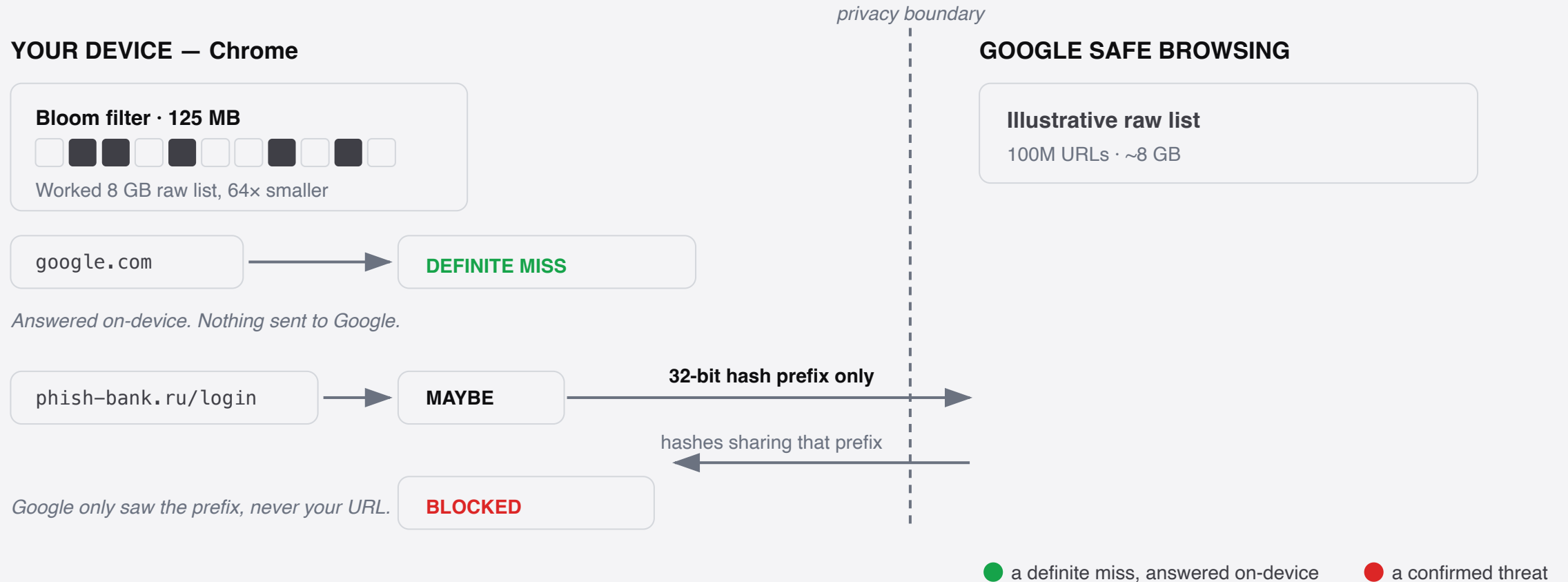
M2

Case Study 2.2: How early Chrome Safe Browsing used a Bloom filter

Chrome Safe Browsing checks if a user is visiting a phishing
site.

Early Safe Browsing: check on-device, confirm possible matches

Worked estimate: a 125 MB Bloom filter derived from an 8 GB raw list.



This example starts with 100 million URLs averaging 80 bytes, or 8 GB. At 10 bits per item, a Bloom filter occupies about 125 MB, 64 times less space. A definite miss ends locally. A possible match sends a 32-bit hash prefix to Google, which returns the matching full hashes so the device can finish the check. The server receives a prefix instead of the full URL.

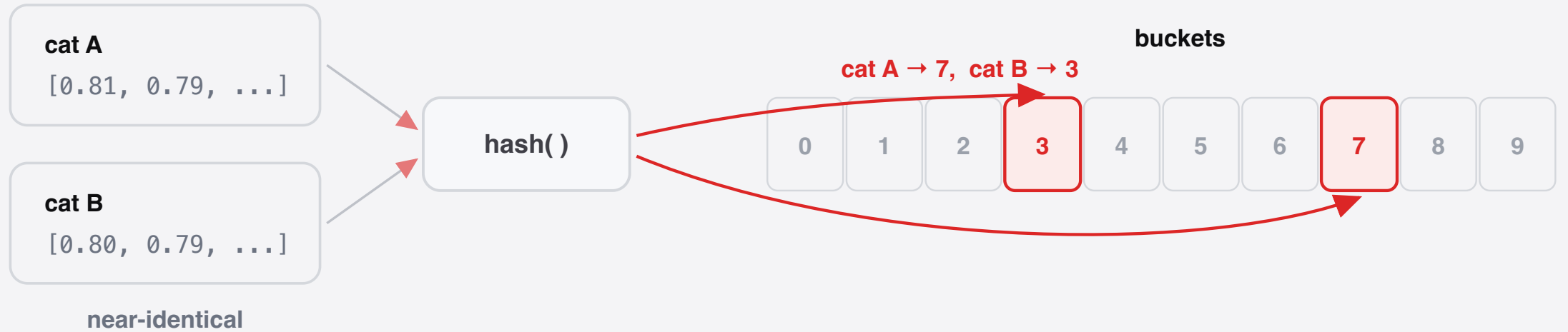
M2

Locality-Sensitive Hashing and Vector Embeddings

An ordinary hash spreads similar inputs across different buckets.

An ordinary hash scatters similar items

A normal hash is built to spread inputs out, so two near-identical vectors land in unrelated buckets.



Similar items, unrelated buckets: "find similar" would have to scan every bucket.

● the two similar items ● where they scatter

An ordinary hash spreads inputs across buckets. Two near-identical vectors (cat A and cat B) land in unrelated buckets, 7 and 3. The bucket index carries no information about distance, so a similarity query still scans every bucket. Locality-sensitive hashing groups near vectors instead.

LSH: similar items land in the same bucket

k random hyperplanes turn each vector into a sign-bit code, then a bucket

Item	Feature vector	k-bit LSH code	Bucket
Cat 1	→ [0.8, 0.7, 0.9, ...]	→ 1011	→ Bucket 11
Cat 2	→ [0.7, 0.8, 0.8, ...]	→ 1011	→ Bucket 11
Dog	→ [0.3, 0.9, 0.2, ...]	→ 1001	→ Bucket 9
Car	→ [0.1, 0.0, 0.5, ...]	→ 0100	→ Bucket 4

Query “find items similar to Cat”: scan only bucket 11 → find Cat 2; never compare against Dog or Car

Each item becomes a feature vector, an embedding, then a short LSH code, then a bucket. The two similar cats receive code 1011 and share a bucket, while the dog and car map to different buckets. A query for items like the cat reads that one bucket instead of comparing against every stored vector. Code bits come from random cuts of the space.

Computing the code: one cut, one bit

Above a cut implies 0, below implies 1. Three cuts give every item three bits.

Which side of each cut?

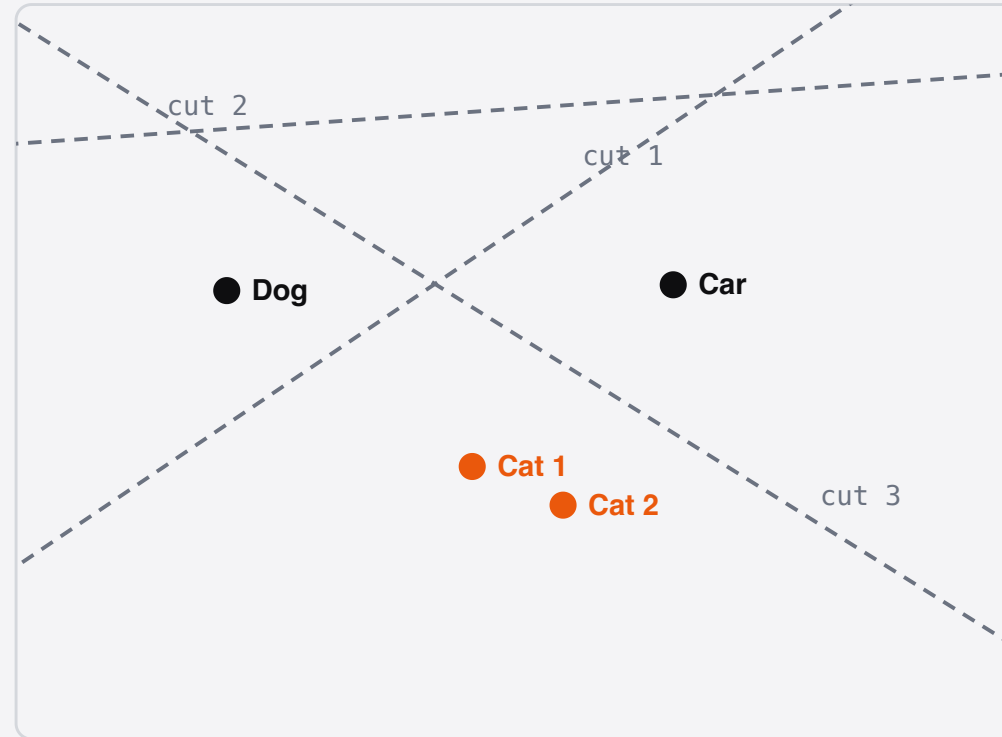
	cut 1	cut 2	cut 3	
Cat 1	below	below	below	111
Cat 2	below	below	below	111
Dog	above	below	below	011
Car	below	below	above	110

above = 0, below = 1

The two cats match on all three cuts.

The dog differs on cut 1, the car on cut 3.

Equal codes go in the same bucket. No distance is ever computed.



Each cut asks one yes-or-no question: above the line is 0, below is 1, and the code assembles one bit at a time. The same four items as Figure 2. The two cats end on 111 because no cut fell between them, so they share a bucket; the dog differs on cut 1 and the car on cut 3, so each lands elsewhere. Nothing measures a distance, so a nearness lookup becomes an ordinary hash-table lookup: cost $O(N)$ drops to about $O(N / 2^k)$. Misses happen when a cut does fall between a close pair.

A cut is always one dimension below the space it splits

Drawn here at d = 3, where the cut is an ordinary plane.

200 vectors indexed

3 cuts · k = 3 of 3 · d = 3

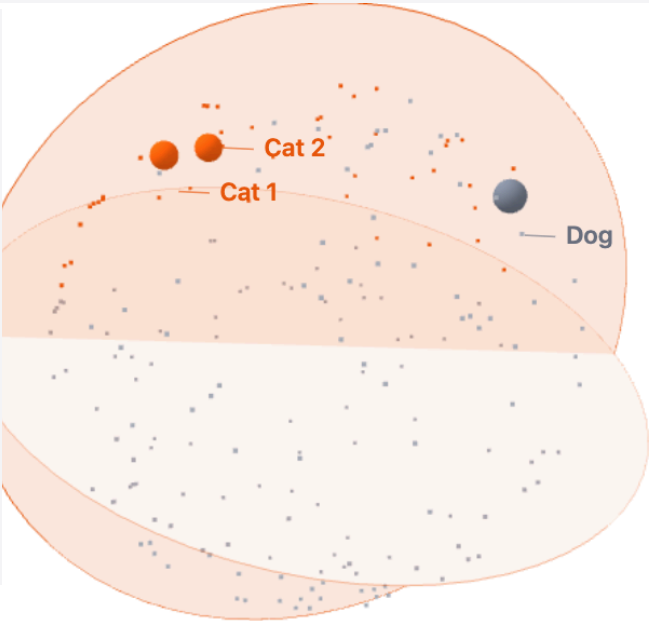
Cat 1 000

Cat 2 000 same bucket

Dog different bucket

19

of 200 still in Cat 1's bucket



three planes: 19 of 200 still share Cat 1's bucket, Cat 2 among them

d = 2 the cut is a line
1-dimensional

d = 3 the cut is a plane
2-dimensional ← you are here

d = 256 the cut is a hyperplane
255-dimensional

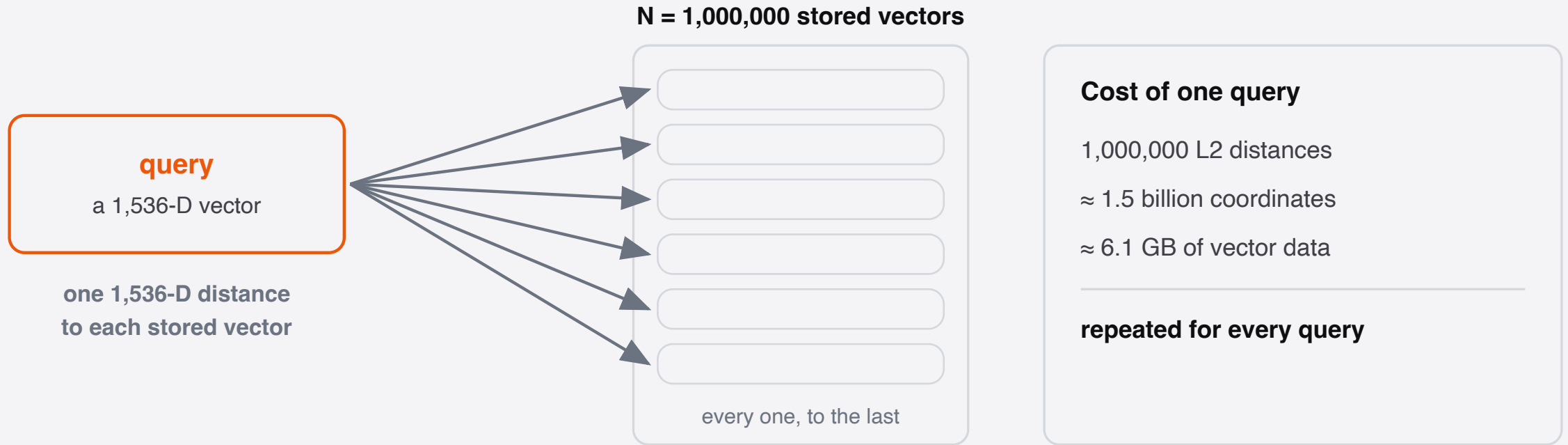
In 2D that is a line, which is what Figure 3 drew; in 3D it is an ordinary plane, drawn here; in an embedding's few hundred dimensions it is a hyperplane (hard to picture). Nothing else changes: above or below is still the only question. One cut is insufficient. After the first plane the dog is still in the cat's bucket, because one cut halves the space and a coin flip put it on that side. The second ejects it, and the near-duplicate survives all three. Each plane roughly halves the candidates: 200, then 94, then 43, then 19.

M2

Case Study 2.3: How OpenAI and Meta's FAISS Speed Up Vector Search

OpenAI's embeddings are vectors.

The exact scan: one query against every vector



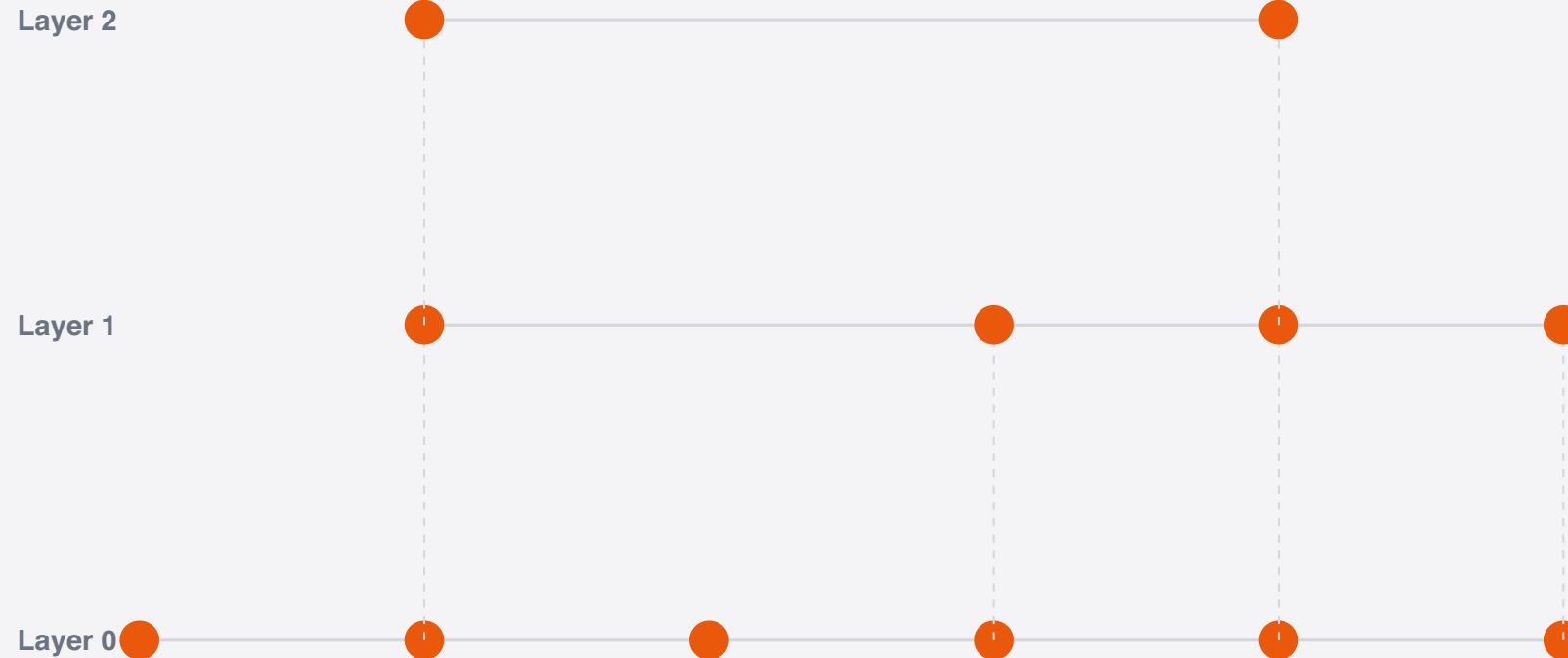
Exact search performs $O(N)$ distance evaluations per query.

Exact scan against all N vectors. At one million vectors, a query computes one million 1,536-dimensional L2 distances, covering about 1.5 billion coordinate values and 6.1 GB of vector data. In FAISS, IndexFlatL2 performs this exact search.

HNSW is built in layers

Every point sits on the ground floor; a random few are promoted higher.

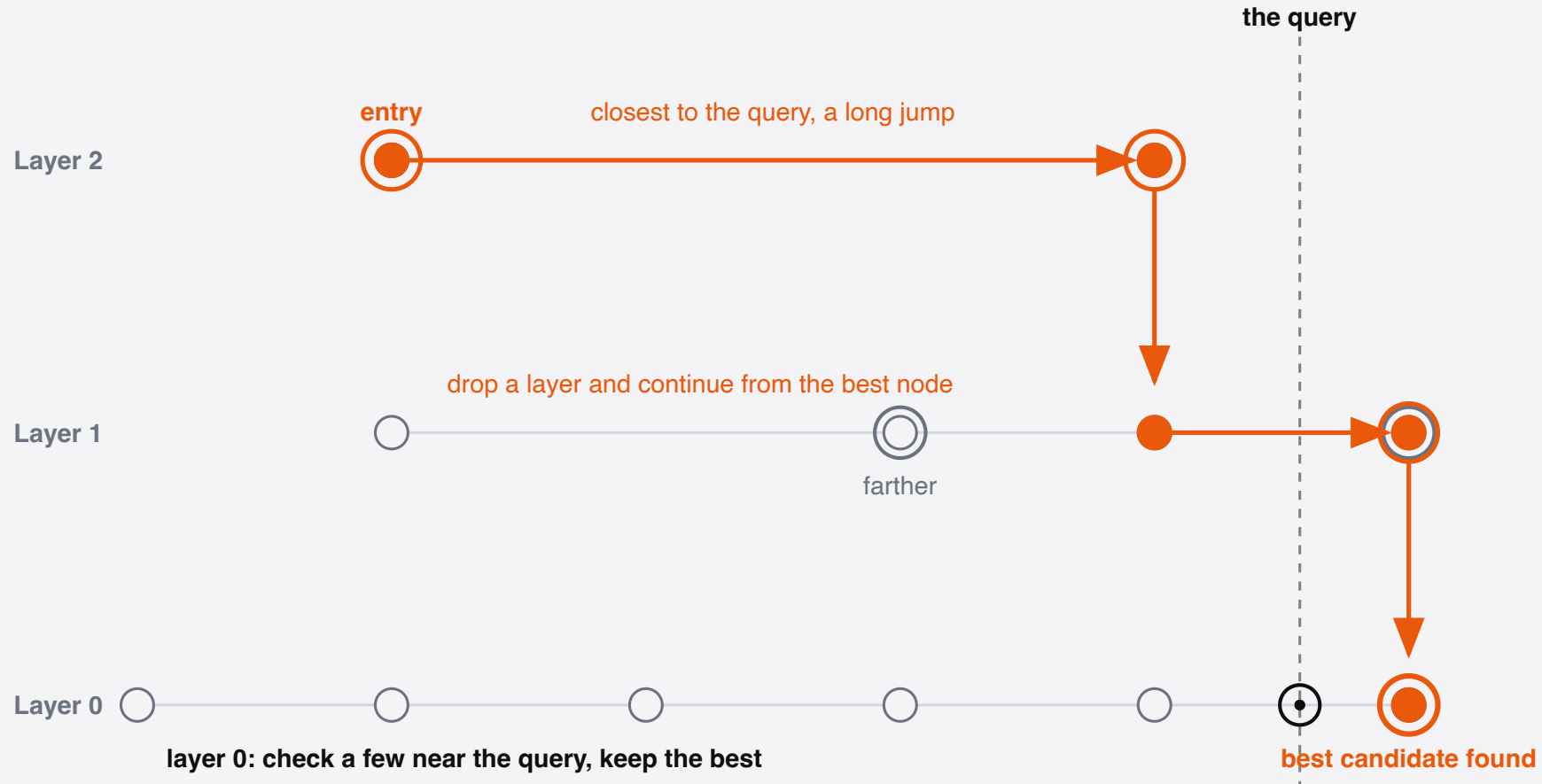
The promoted few form the long links up top.



Layer 0: every point, linked to its nearest neighbors.

Each vector is assigned a random height, so only a few rise to the upper layers. At the bottom, nearly all of them connect to their neighbors like local streets. The few promoted higher become the sparse freeway stops the search starts from.

HNSW search narrows candidates by layer



Top layers are freeways: long links jump you near the query. The bottom layer is local streets: short hops close in. It checks only a handful of candidates, so it is fast but can miss the true nearest.

One query over a million 1,536-D embeddings

ANN examines fewer candidates and may miss an exact neighbour.

Exact search (IndexFlatL2)

Compare the query to every vector

- $1\text{M} \times 1,536\text{-D}$ L2 distances (~6.1 GB read)
- Exact answer
- $O(N)$ work per query

index

ANN index (IVF or HNSW)

Look at a small fraction

- IVF: probe a few clusters
- HNSW: walk a graph to the neighbourhood
- Fewer candidates; may miss a neighbour

Exact search compares the query with all one million vectors. An ANN index checks only a slice of them, so it answers far faster but can skip the true closest. Check more candidates and you trade speed for accuracy.

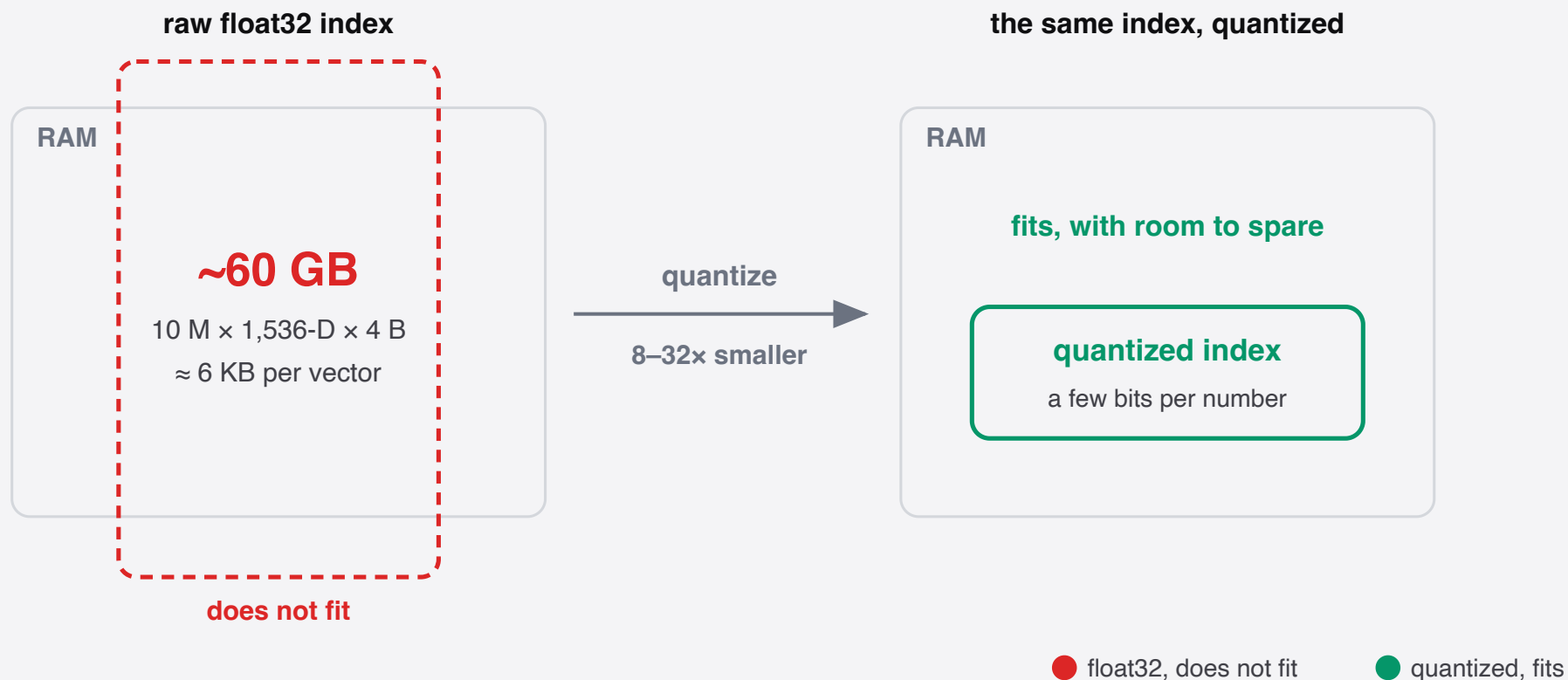
M2

Case Study 2.4: Compressing a Billion Embeddings

A float32 vector index stores more precision than search needs.

The demo fit in RAM; production does not

The index has to sit in RAM to search fast, and raw float32 embeddings are too big to fit in RAM.



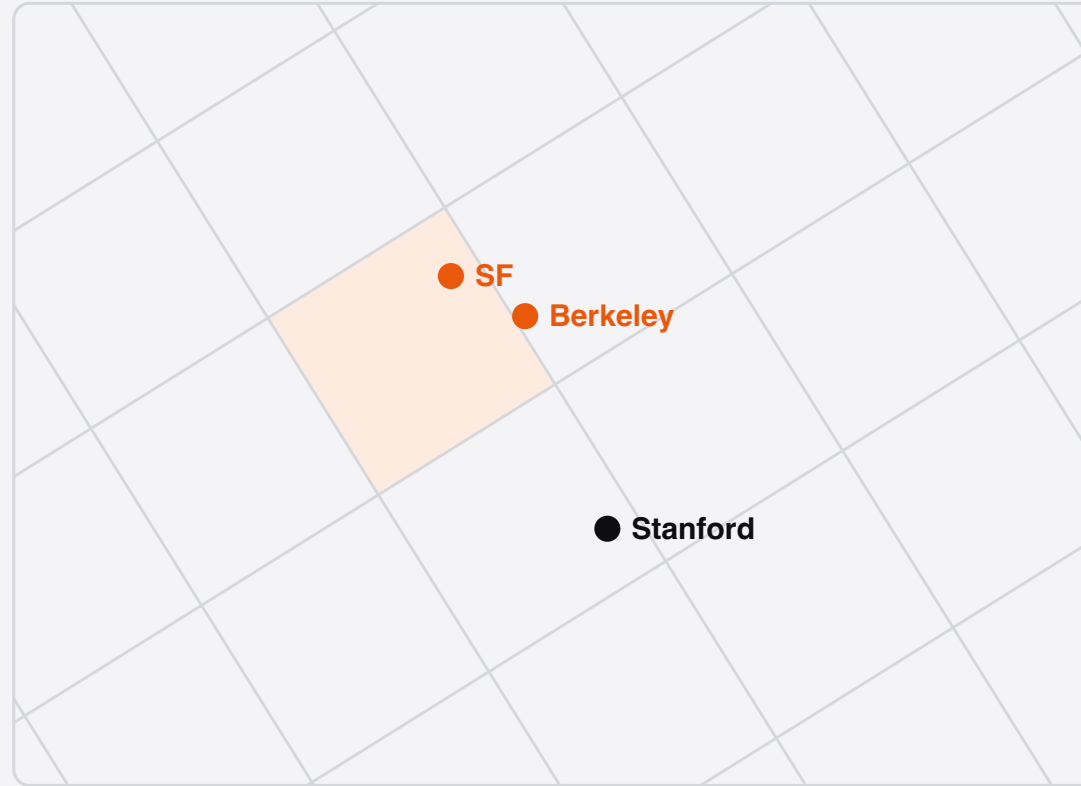
A full-precision embedding is big. Most have about a thousand dimensions or more. OpenAI's smaller embedding model is 1,536-D and the large one is 3,072-D. At 1,536-D float32 that is about 6 KB per vector, so 10 million vectors take ~60 GB in RAM for fast search. The large model uses double. Quantizing each coordinate to a few bits shrinks the index 8 to 32 times and brings it under the RAM limit.

SF $(-1, 0)$
Berkeley $(-1, -1)$
Stanford $(0, 0)$

SF and Berkeley are now in different cells

rotation 58°

The points never move. Only the grid turns.



Eight frames from 0° to 58° . The three cities sit at identical coordinates in every frame, so you can check for yourself that only the grid moved. San Francisco and Berkeley begin in one cell and end in different ones. Stanford, much further off, is untouched.

256 COORDINATES

THE 16 LEVELS THEY ROUND INTO

levels holding any data

15 of 16

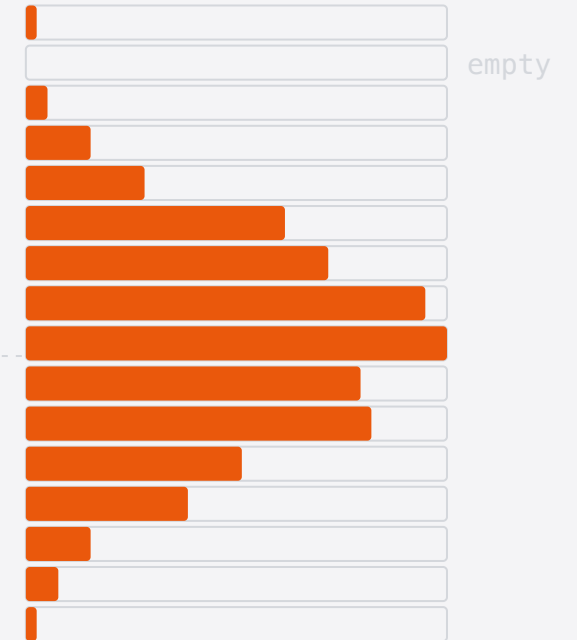
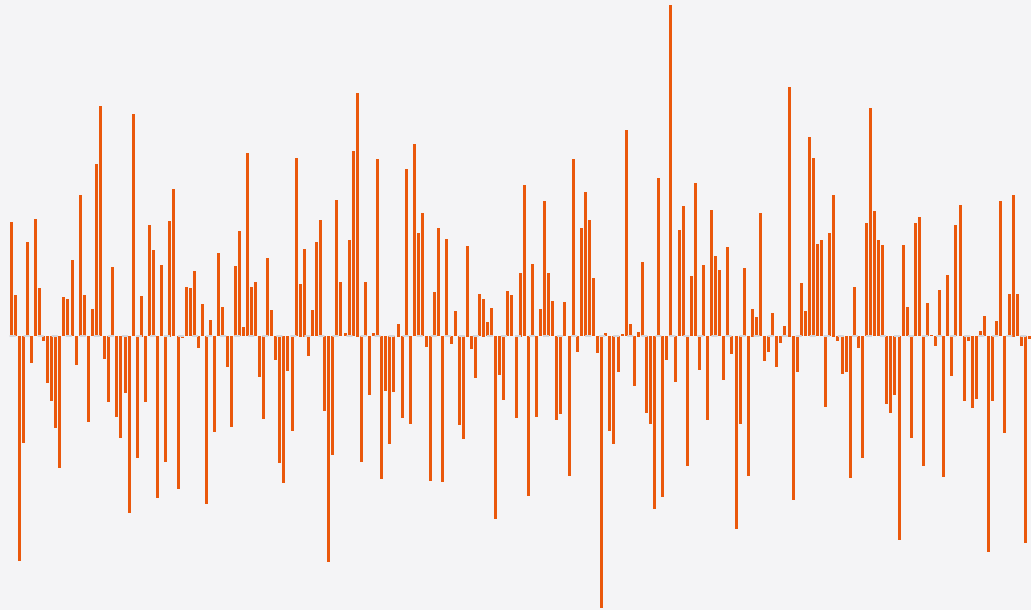
levels sitting empty

1

error at the same 4 bits

11%

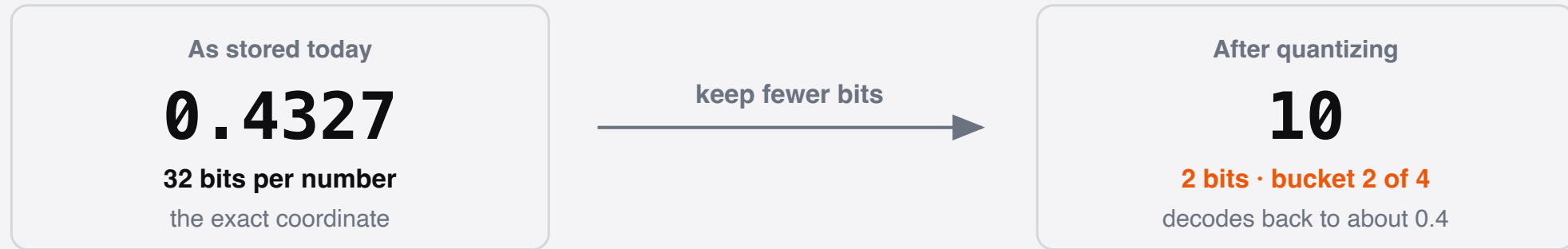
one shared axis: read straight across from a bar to its slot



Every coordinate costs 4 bits to store either way. What changes is how many of the 16 levels do any work.

The same idea at 256 dimensions. A 4-bit budget is 16 levels. Before the rotation the coordinates are so uneven that only 5 levels are ever used, which is 2.3 bits of information in the 4 bits you are paying to store. After it, 15 levels are in play at 3.9 bits. Storage is 4 bits per coordinate either way; only the useful fraction changes.

Drop the precision, keep the distance



The distance between two vectors barely moves, because after the spin no single coordinate carries the answer.

Sixteen times smaller, and no codebook to train.

Scalar quantization of one coordinate. 0.4327 stored in 32 bits maps to a 2-bit bucket 10 and decodes to about 0.4. One coordinate is now rough, but a distance adds up hundreds of them, and the rounding errors mostly cancel out. The rotation first spreads the values evenly, so one simple quantizer fits every coordinate and every dataset, with no training. Two bits gives 16× compression, and one bit gives 32×.

M2

Practice Problems for Basic Systems Design

Every structure here turns a big problem into a small one

Too slow to scan all N, too costly to store all N. Two levers, used together.

1 · Don't scan all N

Partition the lookup, so a query touches one slice

EXACT

Hash to one bucket

Partition by key; read one bucket, not N.

Hashing

APPROXIMATE

Bucket by similarity, or prune early

Bucket similar items, or prune what is definitely absent; check only the candidates.

Bloom · LSH · IVF · HNSW

2 · Don't store all N bytes

Shrink each item, so the index fits

EXACT · LOSSLESS

Drop the redundancy

Exploit the structure the data already has.

RLE · dictionary · delta · bit-pack

APPROXIMATE · LOSSY

Drop the precision, keep the order

Reframe with a random rotation so one coarse quantizer fits any data.

PQ · TurboQuant

Stack both, and you get, for example, IVF-PQ: the vector database.

Two levers for a too-big problem. Lever 1 partitions the lookup so a query examines one slice instead of all N: exactly, by hashing to a bucket; approximately, by pruning what is definitely absent (Bloom) or bucketing by similarity (LSH, IVF, HNSW) and checking only the candidates near the query. Lever 2 shrinks each item: losslessly by dropping the redundancy the data already has (only if it has structure), or lossily by dropping precision while keeping the rank order (a random rotation reframes the data so one coarse quantizer fits any of it). Each lever has an exact form and an approximate one. Real systems pull both at once: a vector database like IVF-PQ partitions the data and then compresses what is left.

PRACTICE

[all 11 problems, with solutions ↗](#)