Best practices for using pgvector

Jonathan Katz
(he/him)
Principal Product Manager – Technical
AWS

Agenda

Overview of generative AI and the role of databases

PostgreSQL as a vector store

pgvector best practices

Ongoing work

CUSTOMER

Is it possible to exchange the shoes I bought for brown ones?

DEVELOPER CREATED AGENT

Of course, do you have your order number?

Tags
1
Human: You are an agent who manages orders and returns on
executing the set of APIs in order to fulfill user input.

Emphasis (capitalized)
2
Valid "api" values are GetOrderHistory::GetProductCatalogue,
- DO NOT return an api if all required parameter values are not
- DO NOT replace the placeholders in the api_name with api_input
- Return available parameters in api_inputs ONLY.

Valid "verb" is HTTP verb used in "APIs" e.g. GET, PUT etc

Valid "api_input" as json from "User Input", "Observation" or
- NEVER assume value for any parameter, mark the value as

Convergence criteria
3
DO NOT go into a loop and return exact same apis with exact

Format (JSON)
4
Provide only ONE action per $JSON_BLOB, as shown:

{ "api": $API_NAME, "verb": $HTTP_VERB, "api_input": { $PARAME

History format
5
Conversation History: Below is the history of the conversation

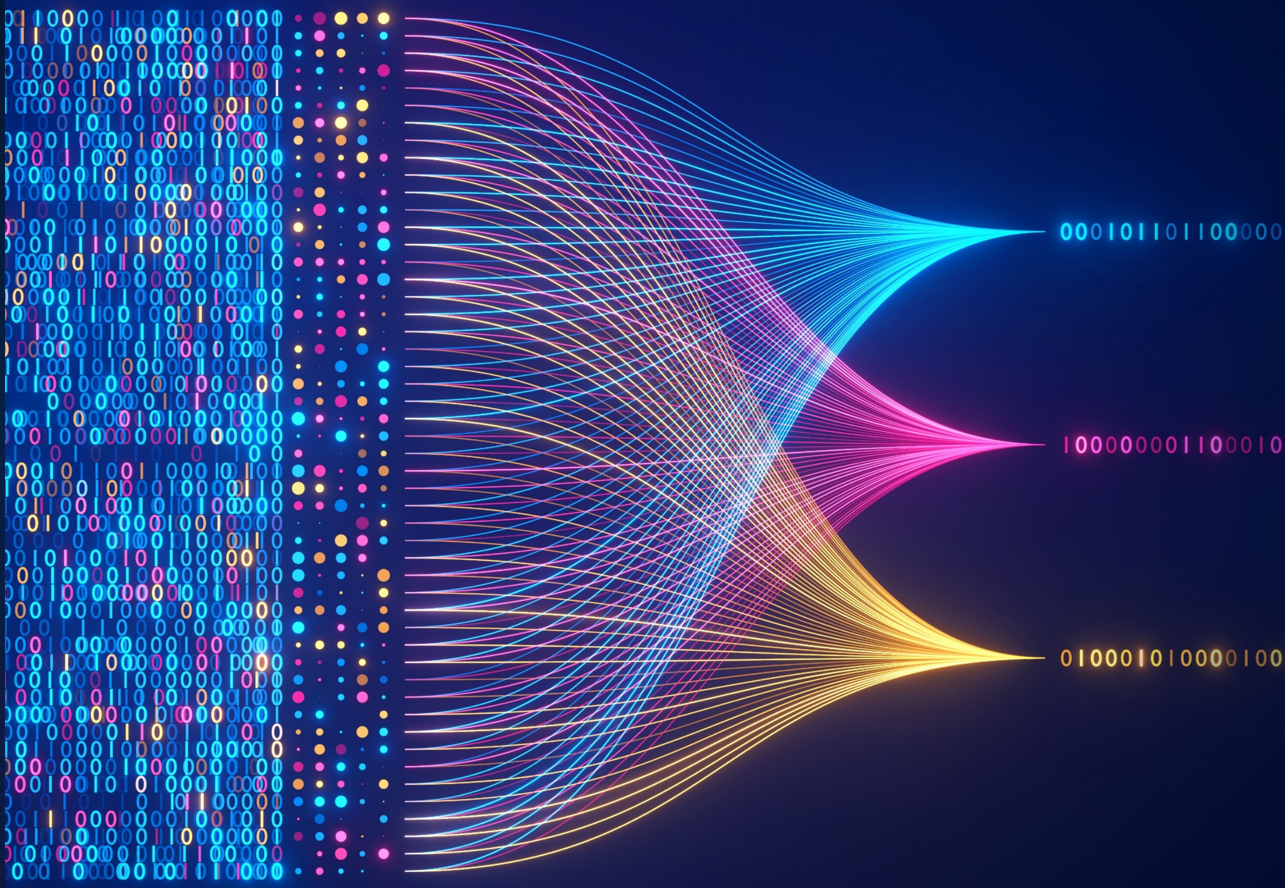
Generative AI is powered by foundation models

Pretrained on vast amounts of unstructured data

Contain a large number of parameters that make them capable of learning complex concepts

Can be applied in a wide range of contexts

Customize FMs using your data for domain-specific tasks



How to provide your data to generative AI applications?

Training your own purpose-built LLM foundation models

Train a foundation model using your curated, specialized data

Fine-tuning a foundation model

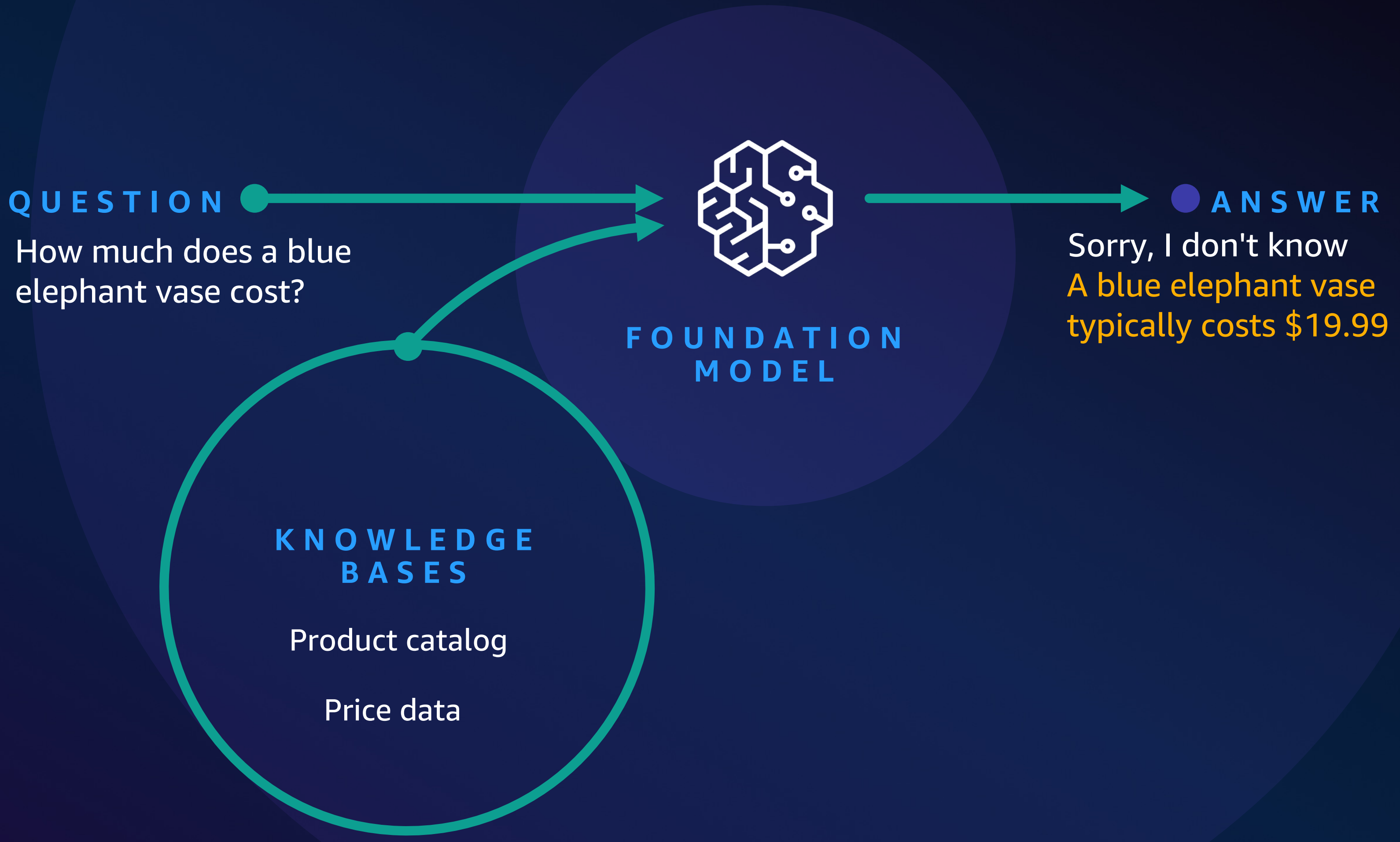
Fine-tune a foundation model using your curated, labeled data

Context engineering using RAG

Guide foundation models by prompting with contextually relevant data (RAG)

Retrieval Augmented Generation (RAG)

Configure FM to interact with your company data



What are vector embeddings?

Source domain-specific data → Tokenization → Vectorization → Store in vector data store → Perform semantic similarity search → Include semantically similar context in prompt

Documents

Audio/video

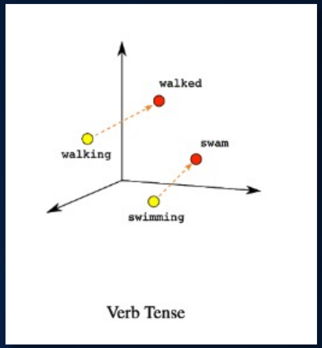
Images

Semantic elements:
- Words, phrases
- Paragraphs, documents
- Scenes, song sections
- Faces, detected picture elements
- And more

0.35 0.1 0 0.9 001.0 00 0001.0 0 0…

0.35 0.1 0 0.8 001.0 00 0001.0 0 0…

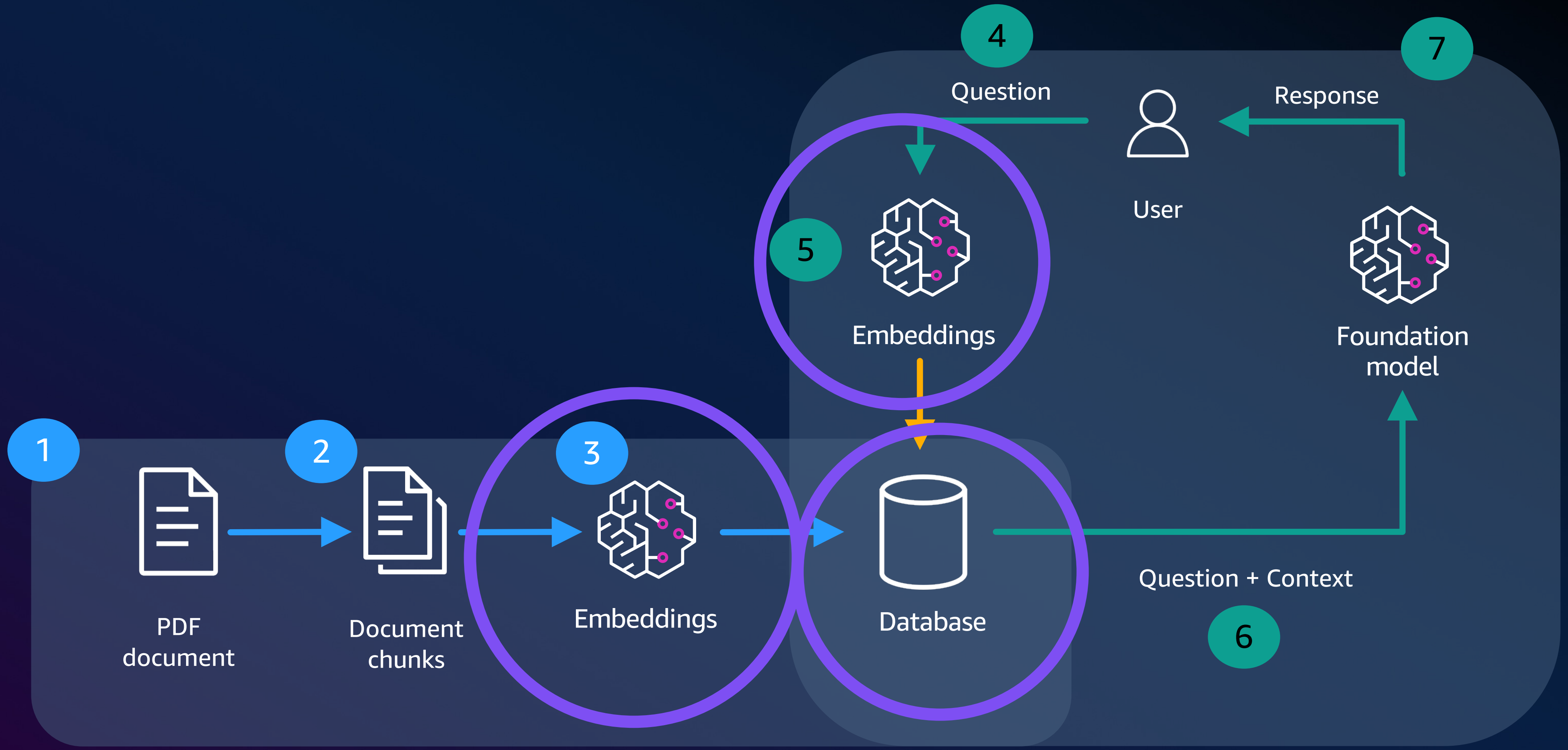
0.15 0.1 0 0.7 001.0 00 0001.0 0 0…



3D simplified representation. Embeddings can have thousands of dimensions. Source: https://daleonai.com/embeddings-explained

Embeddings: When vector elements are semantic, used in generative AI

The role of vectors in RAG



Challenges with vectors

- Time to generate embeddings
- Embedding size
- ~~Compression~~
- Query time

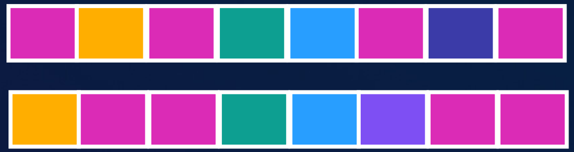
1,000,000 => 5.7 GB

Approximate nearest neighbor (ANN)

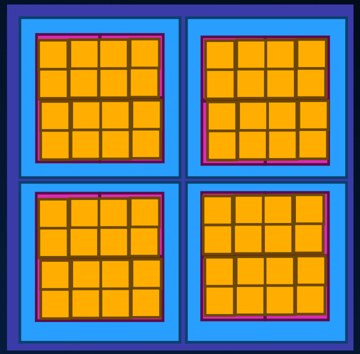
- Find similar vectors without searching all of them
- Faster than exact nearest neighbor
- “Recall” – % of expected results

Recall: 80%

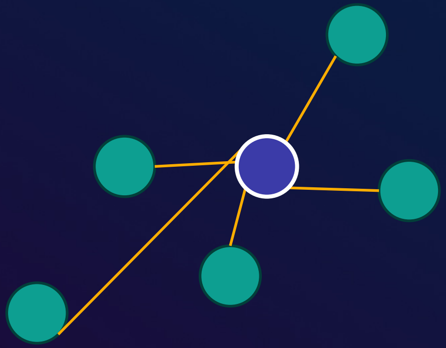
ANN indexing algorithm types and tradeoffs



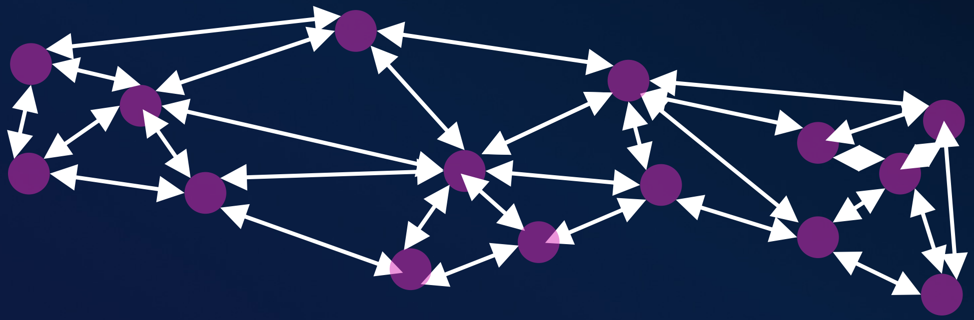
Hash



Tree

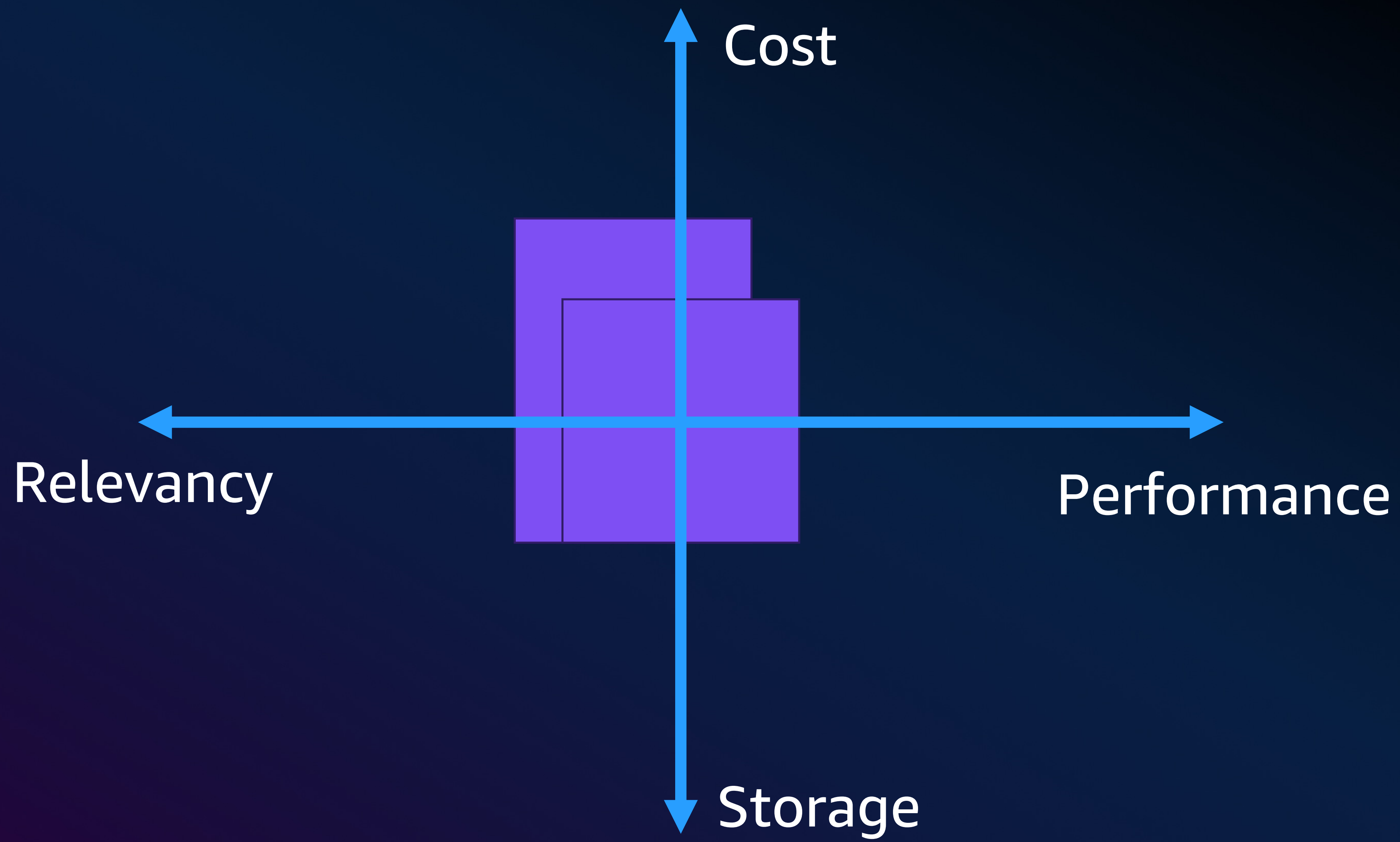


Cluster



Graph

Considerations for vector storage



Questions for choosing a vector storage system

- Where does vector storage fit into my workflow?
- How much data am I storing?
- What matters to me: **Storage**, **performance**, **relevancy**, **cost**?
- **What are my trade-offs: Indexing, query time, schema design?**

PostgreSQL as a vector store

Why use PostgreSQL for vector searches?

Existing client libraries work without modification

- May require an upgrade

Convenient to co-locate app + AI/ML data in same database

Interfacing with PostgreSQL storage gives ACID transactional storage

Why care about ACID for vectors?

- <u>A</u>tomicity: "All or nothing" stored in transaction (bulk loads)
- <u>C</u>onsistency: Follows rules for other data stored in database
- <u>I</u>solation: Correctness in returned results; committed transactions "immediately available"
- <u>D</u>urability: One committed, vectors are safely stored.

What is pgvector?

Adds support for storage, indexing, searching, metadata with choice of distance

`vector` data type

Co-locate with embeddings

Exact nearest neighbor (K-NN)
Approximate nearest neighbor (ANN)

Supports HNSW & IVFFlat indexing, with options for scalar and binary quantization

Distance operations include
Cosine, Euclidean/L2, Manhattan/L1, Dot product, Hamming, Jaccard

github.com/pgvector/pgvector

Why pgvector?

2023

- Vector searches in PostgreSQL
  - "It was there"
- Can use existing PostgreSQL drivers
- Open source
- C-based

2024

- High performance vector searches
- Support for larger vectors
- Sustained, rapid improvements
- Better support in developer tools

pgvector: Year-in-review timeline

- **v0.4.x** (1/2023 – 6/2023)
  - Improved IVFFlat plan costs
  - Increasing dimension of vectors stored in table + index
- **v0.5.x** (8/2023 – 10/2023)
  - Add HNSW index + distance function performance improvements
  - Parallel IVFFlat builds
- **v0.6.x** (1/2024 – 3/2024)
  - Parallel HNSW index builds + in-memory build optimizations
- **v0.7.x** (4/2024)
  - halfvec (2-byte float), bit(n) index support, sparsevec (up to 1B dim)
  - Quantization (scalar/binary), Jaccard/hamming distance, explicit SIMD

Indexing methods: IVFFlat and HNSW

- IVFFlat
  - K-means based
  - Organize vectors into lists
  - Requires prepopulated data
  - Insert time bounded by # lists
- HNSW
  - Graph based
  - Organize vectors into "neighborhoods"
  - Iterative insertions
  - Insertion time increases as data in graph increases

Which search method do I choose?

Exact nearest neighbors: No index

Fast indexing: IVFFlat

Easy to manage: HNSW

High performance/recall: HNSW

Best practices for pgvector

Storage strategies

HNSW strategies

Quantization

Filtering

Best practices: Vector storage

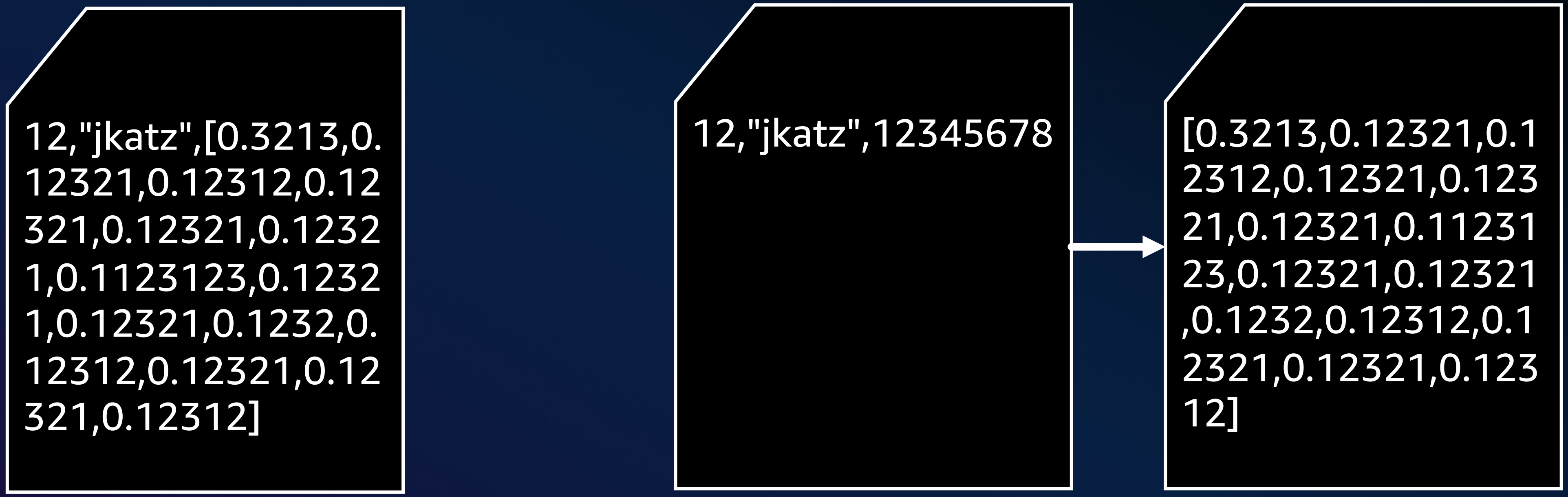
How does PostgreSQL store vectors?

- Page: PostgreSQL atomic storage unit
  - 8192 bytes = 8K = 8KiB
- Heap (table) pages are resizable as a compile time flag
- Index pages are not resizable
- This is a real (😉) problem for vectors
  - 1536-dim 4-byte vector = 6KiB
  - 3072-dim 4-byte vector = 12KiB

🍞 TOAST – handling larger data

- TOAST (The Oversized-Attribute Storage Technique) is a mechanism for storing data larger than 8KB
  - By default, PostgreSQL “TOASTs” values over 2KB (510d 4-byte float)
- Storage types:
  - `PLAIN`: Data stored inline with table
  - `EXTENDED`: Data stored/compressed in TOAST table when threshold exceeded
    - pgvector default before 0.6.0
  - `EXTERNAL`: Data stored in TOAST table when threshold exceeded
    - pgvector default 0.6.0+
  - `MAIN`: Data stored compressed inline with table

Visualizing TOAST for pgvector



PLAIN

EXTENDED / EXTERNAL

Impact of TOAST on vector data

- Traditionally, TOAST data is not on the "hot path"
  - Impacts query plan and maintenance operations

- Compression is ineffective

- Unable to use for index pages

Impact of TOAST on pgvector queries

```
Limit (cost=772135.51..772136.73 rows=10 width=12)
-> Gather Merge (cost=772135.51..1991670.17 rows=10000002 width=12)
   Workers Planned: 6
   -> Sort (cost=771135.42..775302.08 rows=1666667 width=12)
      Sort Key: ((<-> embedding))
      -> Parallel Seq Scan on vecs128 (cost=0.00..735119.34 rows=1666667
width=12)
```

128 dimensions

Impact of TOAST on pgvector queries

```
Limit (cost=149970.15..149971.34 rows=10 width=12)
-> Gather Merge (cost=149970.15..1347330.44 rows=10000116 width=12)
   Workers Planned: 4
   -> Sort (cost=148970.09..155220.16 rows=2500029 width=12)
        Sort Key: (($1 <-> embedding))
        -> Parallel Seq Scan on vecs1536 (cost=0.00..94945.36 rows=2500029
width=12)
```

1,536 dimensions

Strategies for pgvector and TOAST

- Use `PLAIN` storage
  - `ALTER TABLE … ALTER COLUMN ... SET STORAGE PLAIN`
  - Requires table rewrite (`VACUUM FULL`) if data already exists
  - Limits vector sizes to 2,000 dimensions

- Use `min_parallel_table_scan_size` to induce more parallel workers

- TOAST is currently not available for indexes

Impact of TOAST on pgvector queries

```
Limit (cost=95704.33..95705.58 rows=10 width=12)
-> Gather Merge (cost=95704.33..1352239.13 rows=10000111 width=12)
   Workers Planned: 11
   -> Sort (cost=94704.11..96976.86 rows=909101 width=12)
      Sort Key: (($1 <-> embedding))
      -> Parallel Seq Scan on vecs1536 (cost=0.00..75058.77 rows=909101 width=12)
```

1,536 dimensions

```
SET min_parallel_table_scan_size TO 1
```

Best practices: HNSW best practices

HNSW index building parameters

m

Maximum number of bidirectional links between indexed vectors

Default: 16

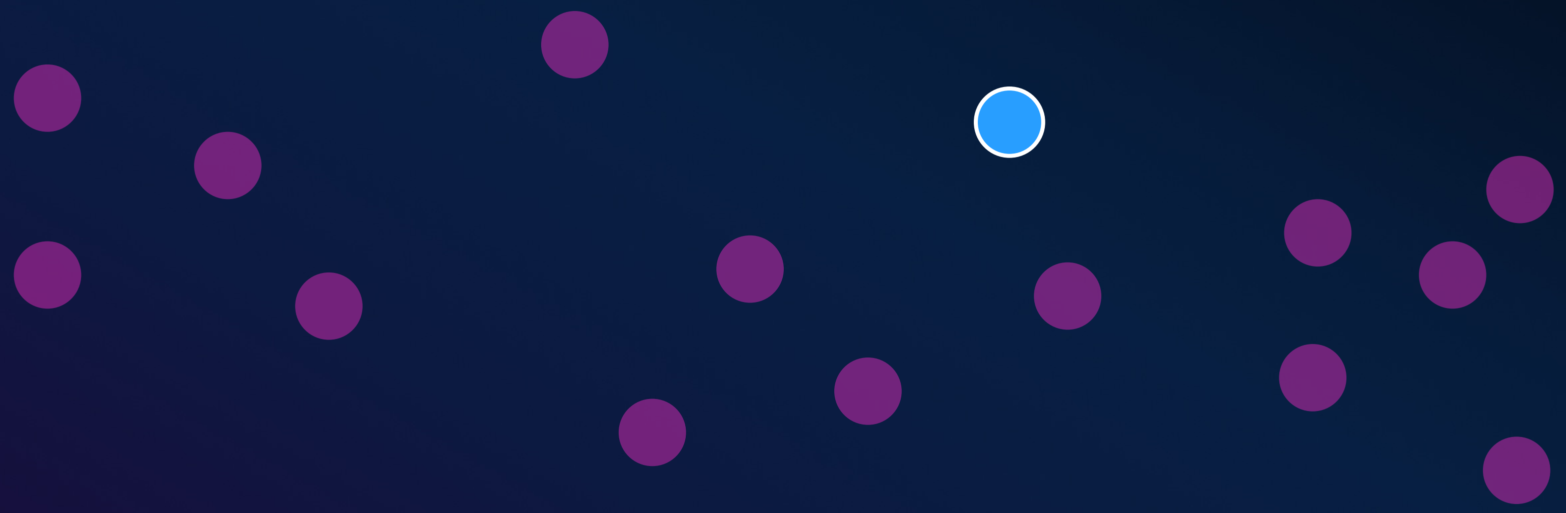
ef_construction

Number of vectors to maintain in “nearest neighbor” list

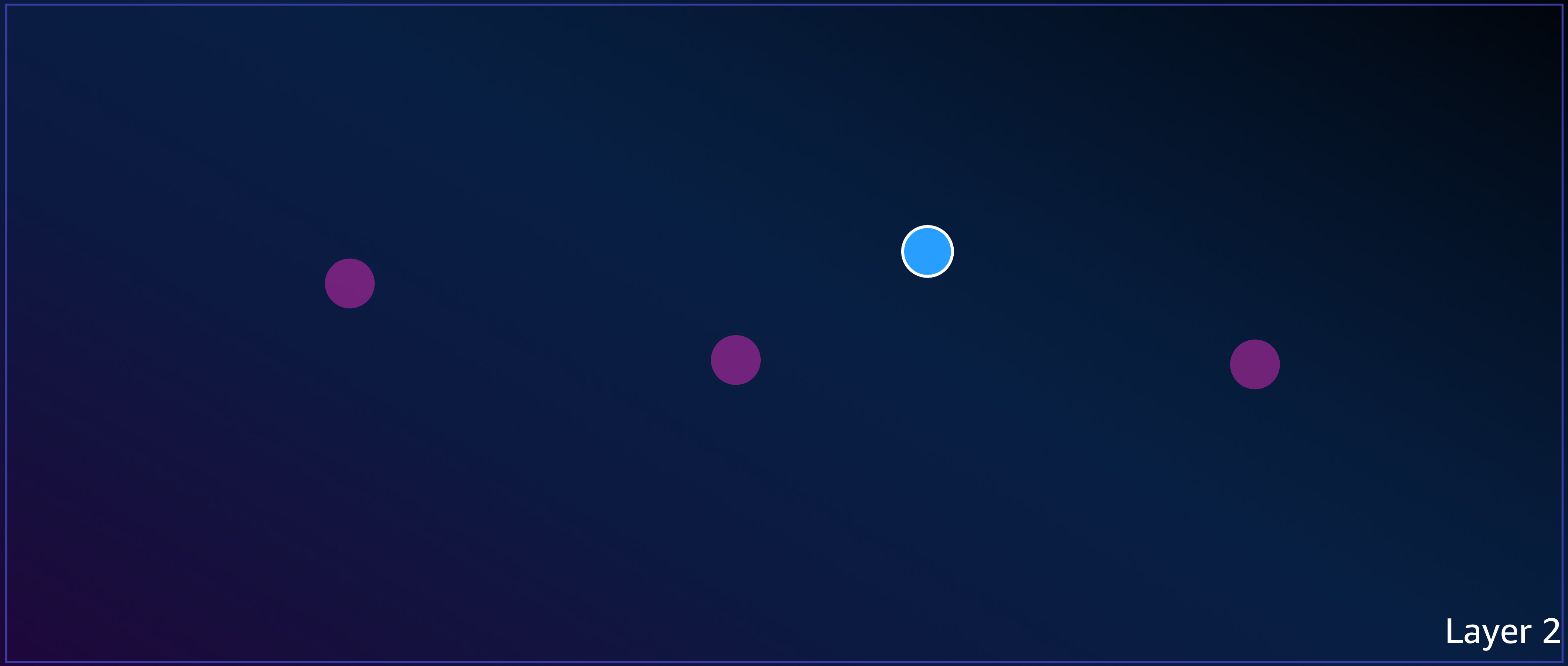
Default: 64

Recommendation: 256

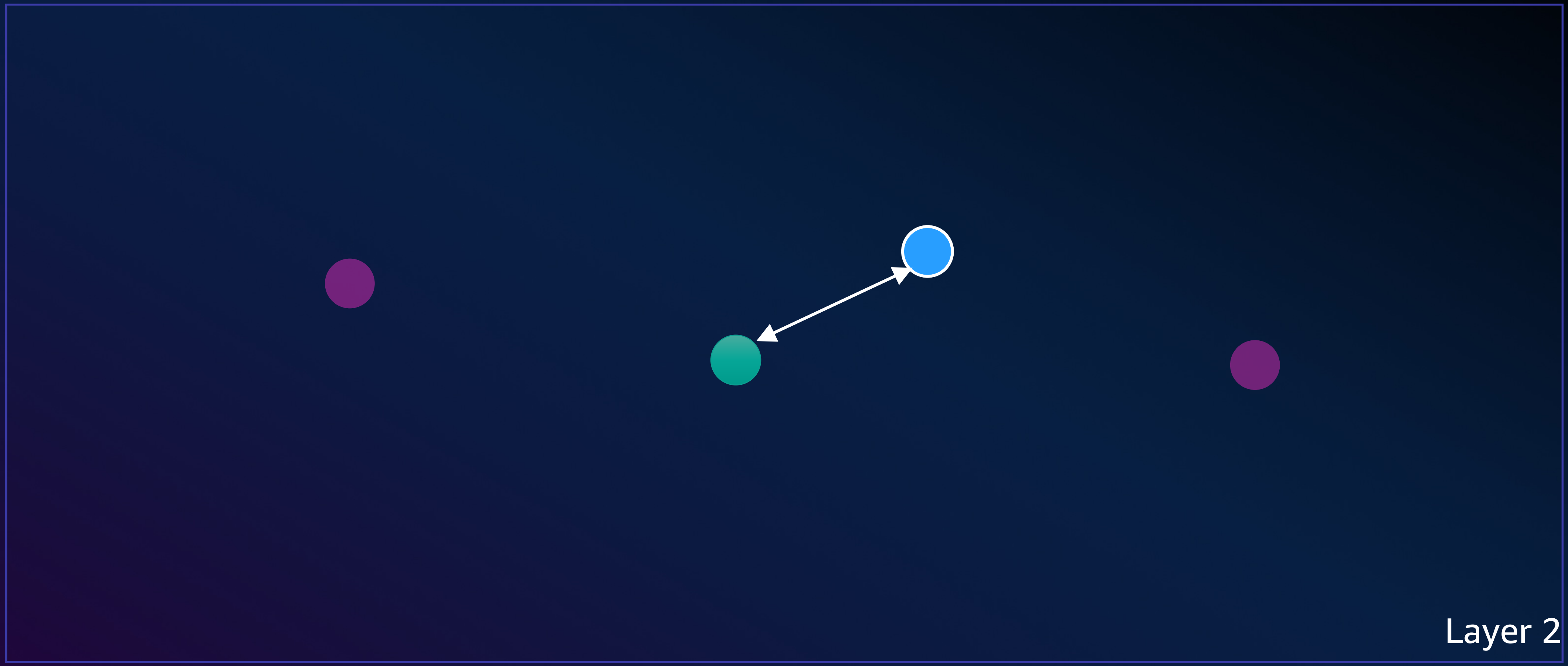
Building an HNSW index



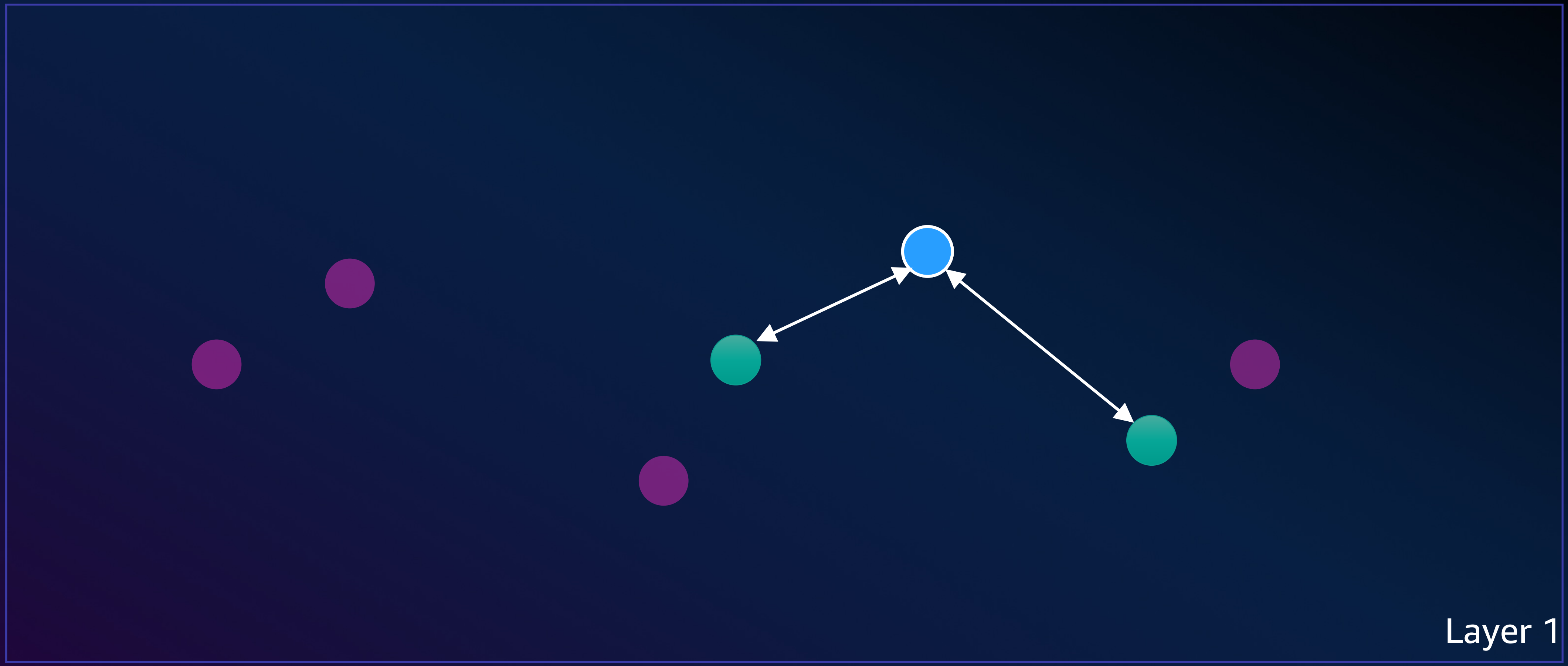
Building an HNSW index



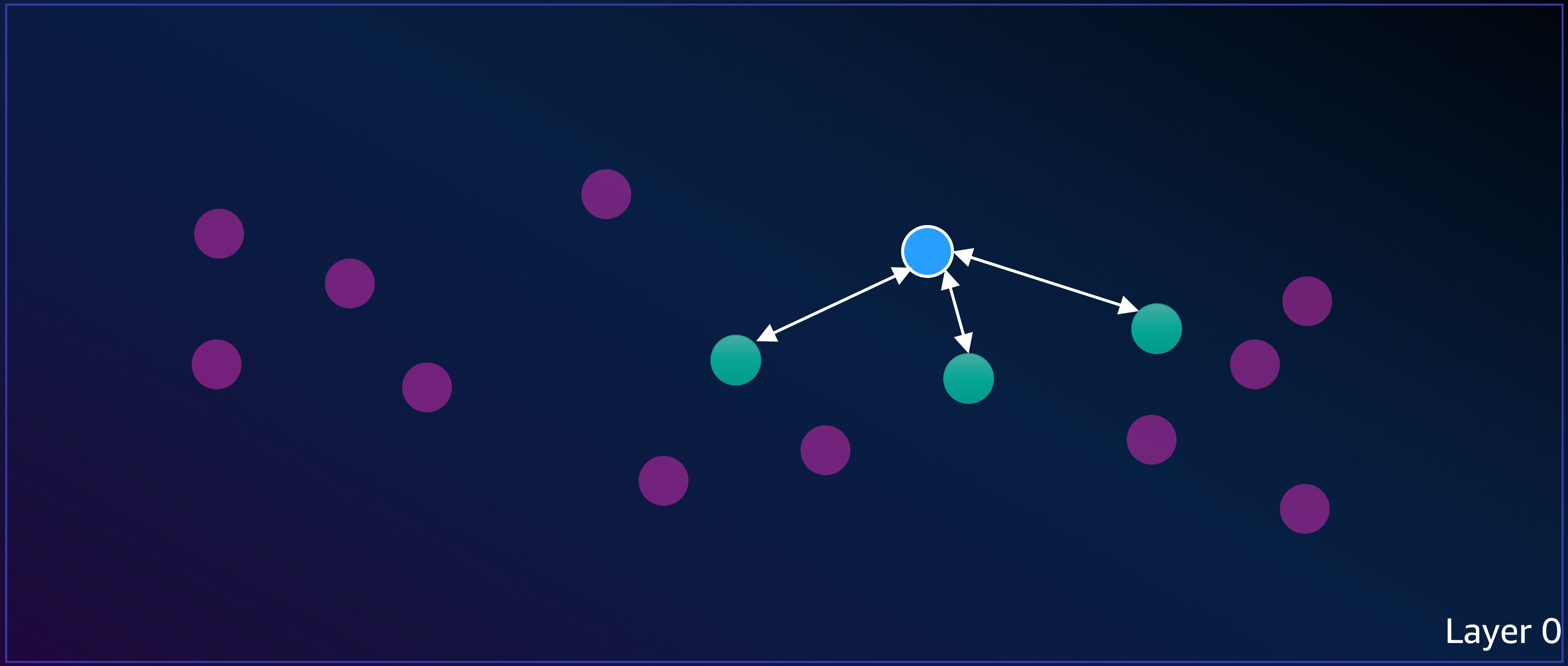
Building an HNSW index



Building an HNSW index



Building an HNSW index

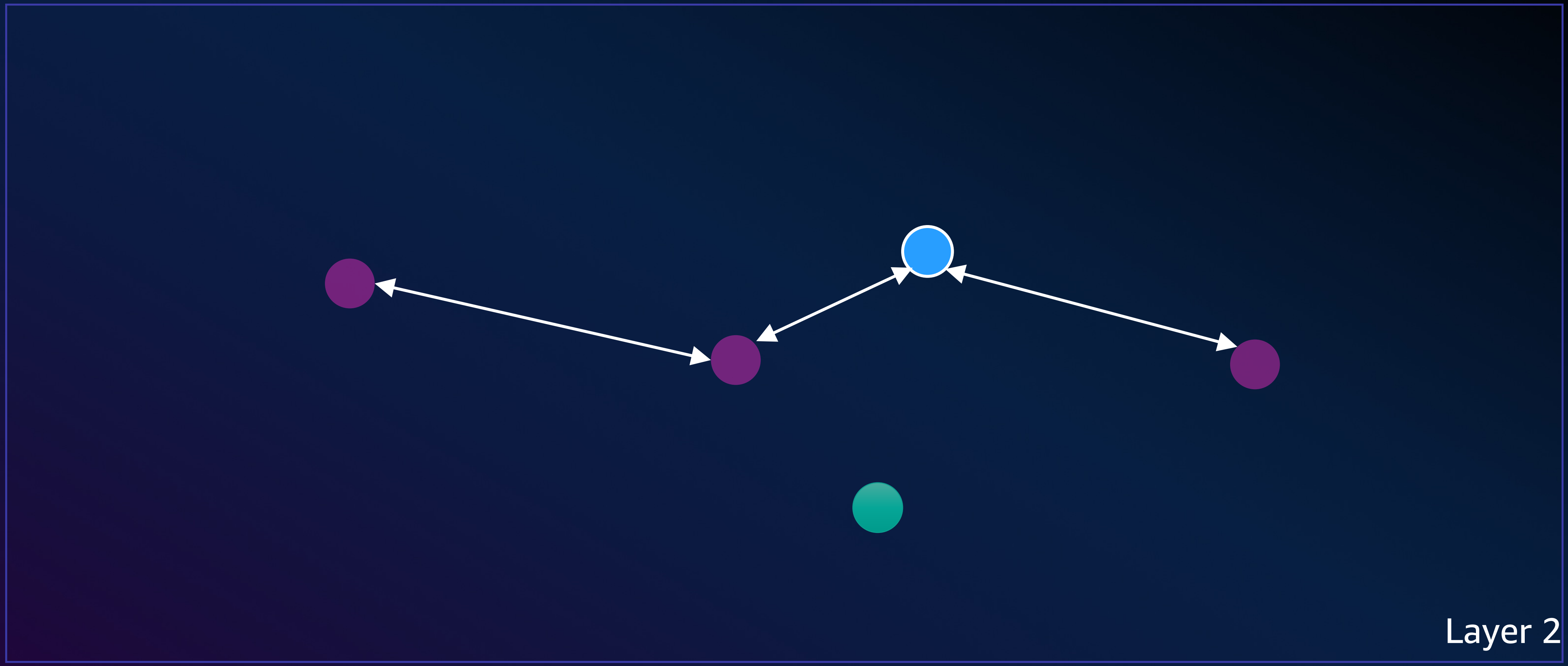


HNSW query parameters

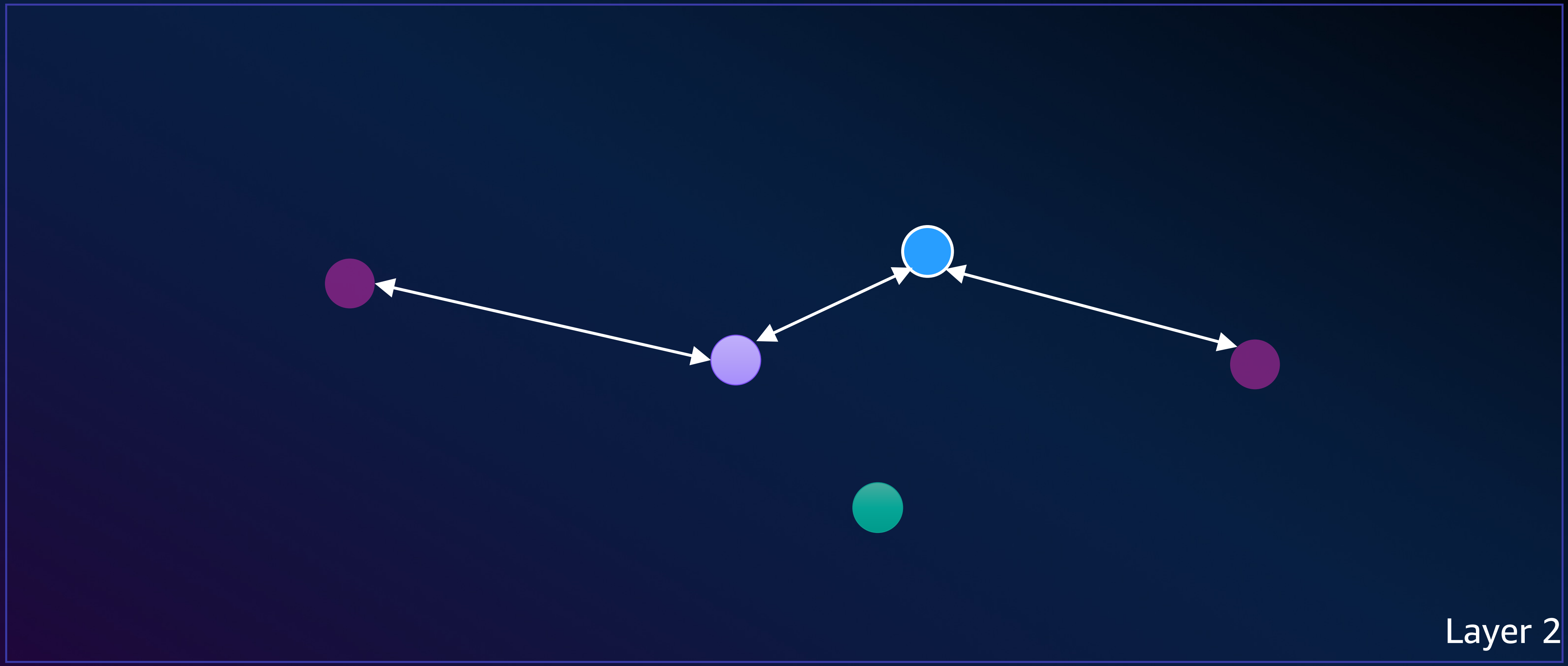
hnsw.ef_search

- Number of vectors to maintain in “nearest neighbor” list
- Must be greater than or equal to LIMIT

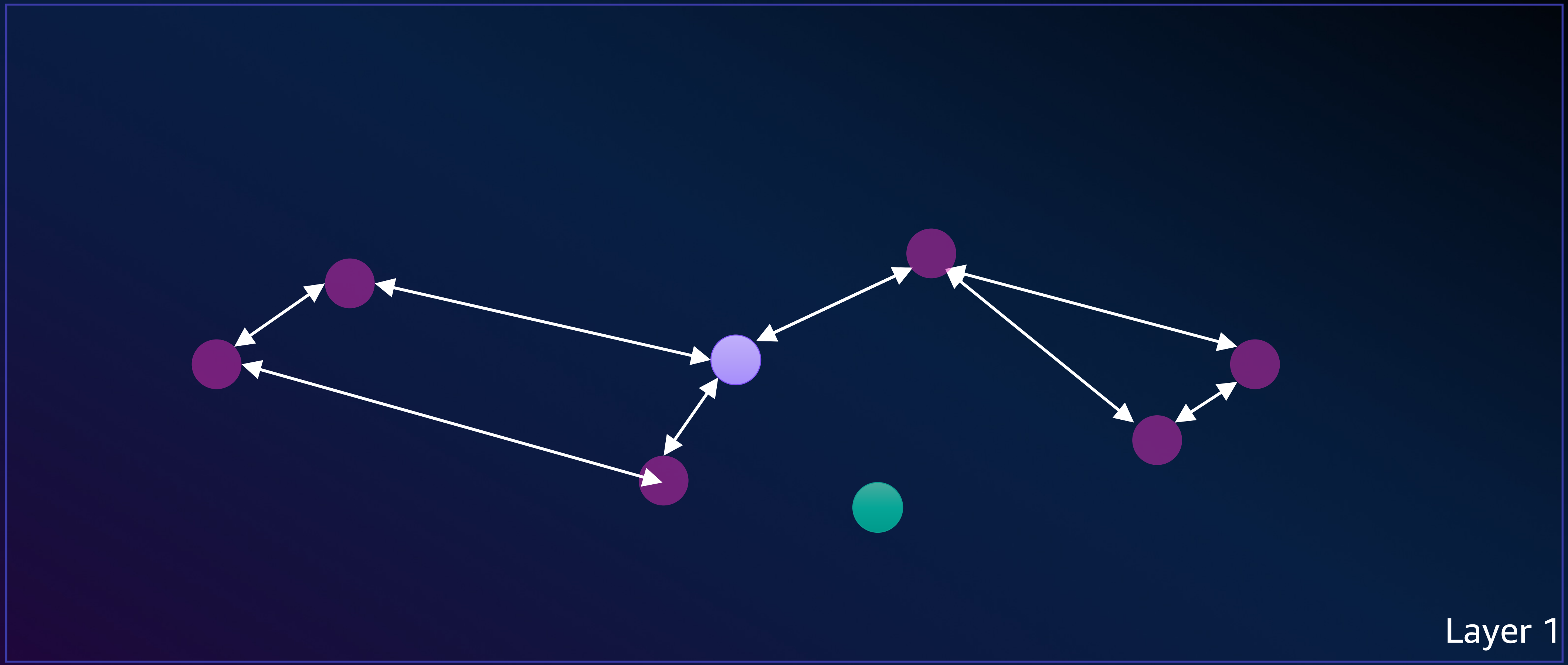
Querying an HNSW index



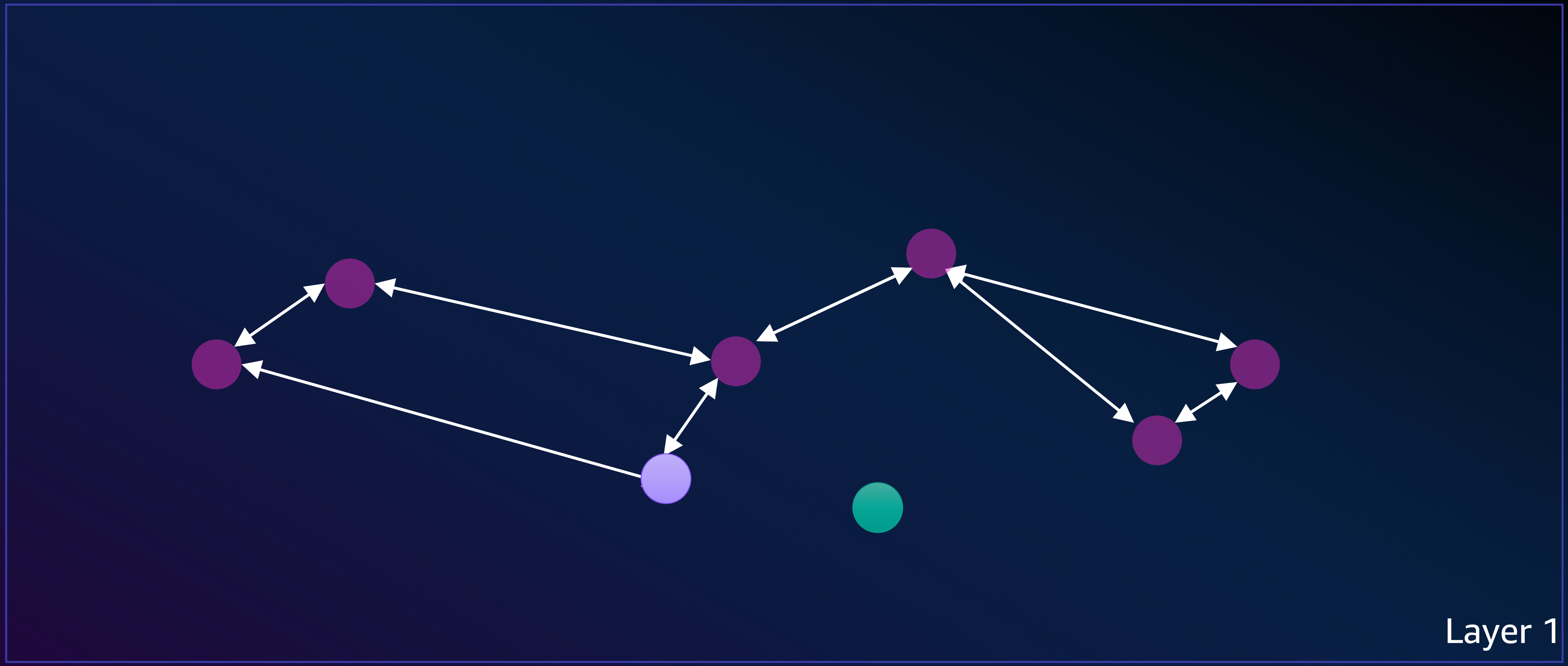
Querying an HNSW index



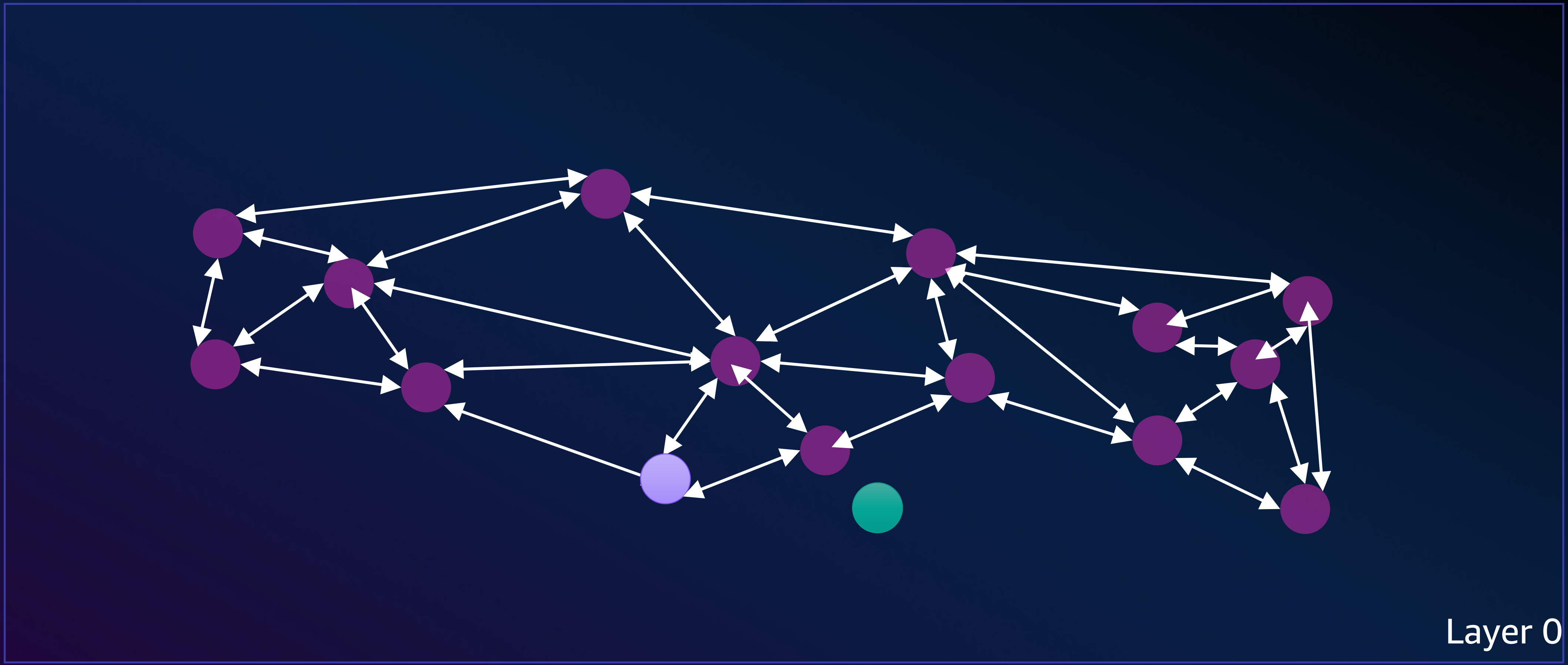
Querying an HNSW index



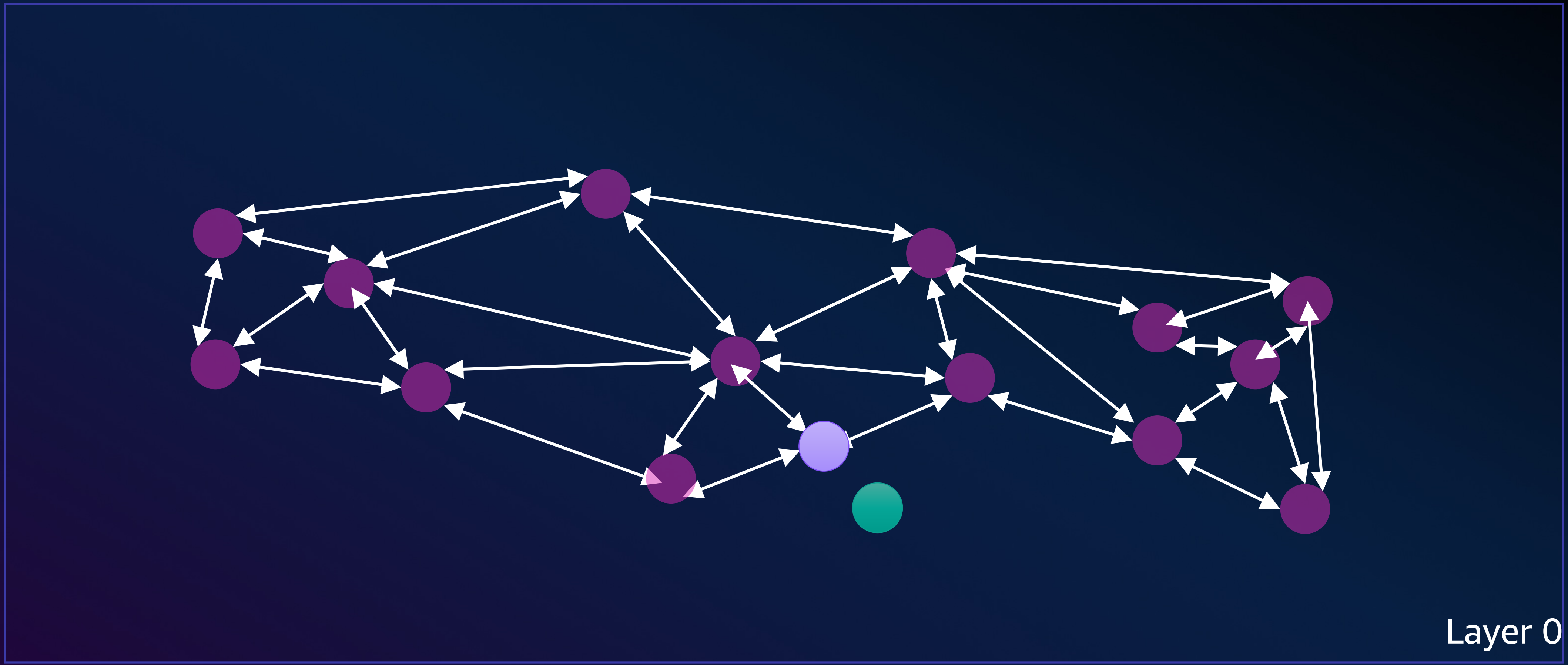
Querying an HNSW index



Querying an HNSW index

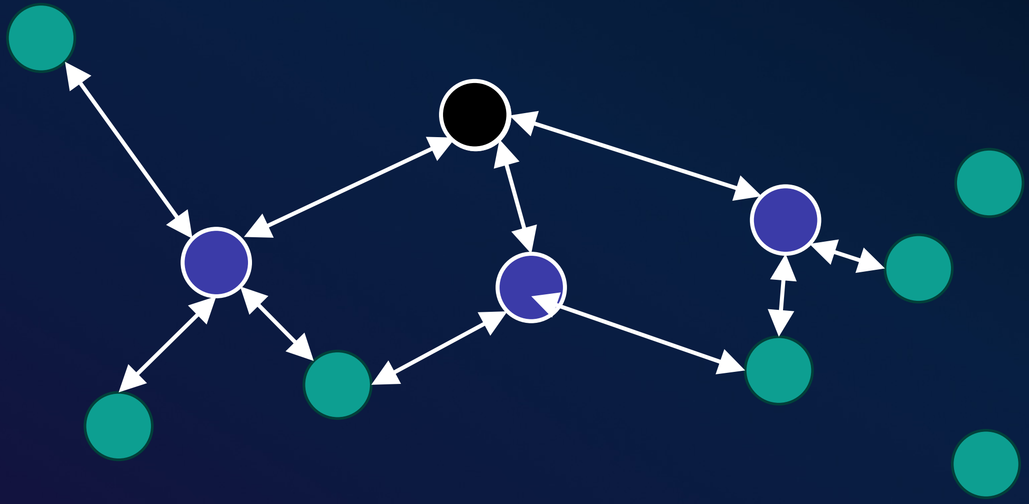


Querying an HNSW index



pgvector and HNSW index maintenance

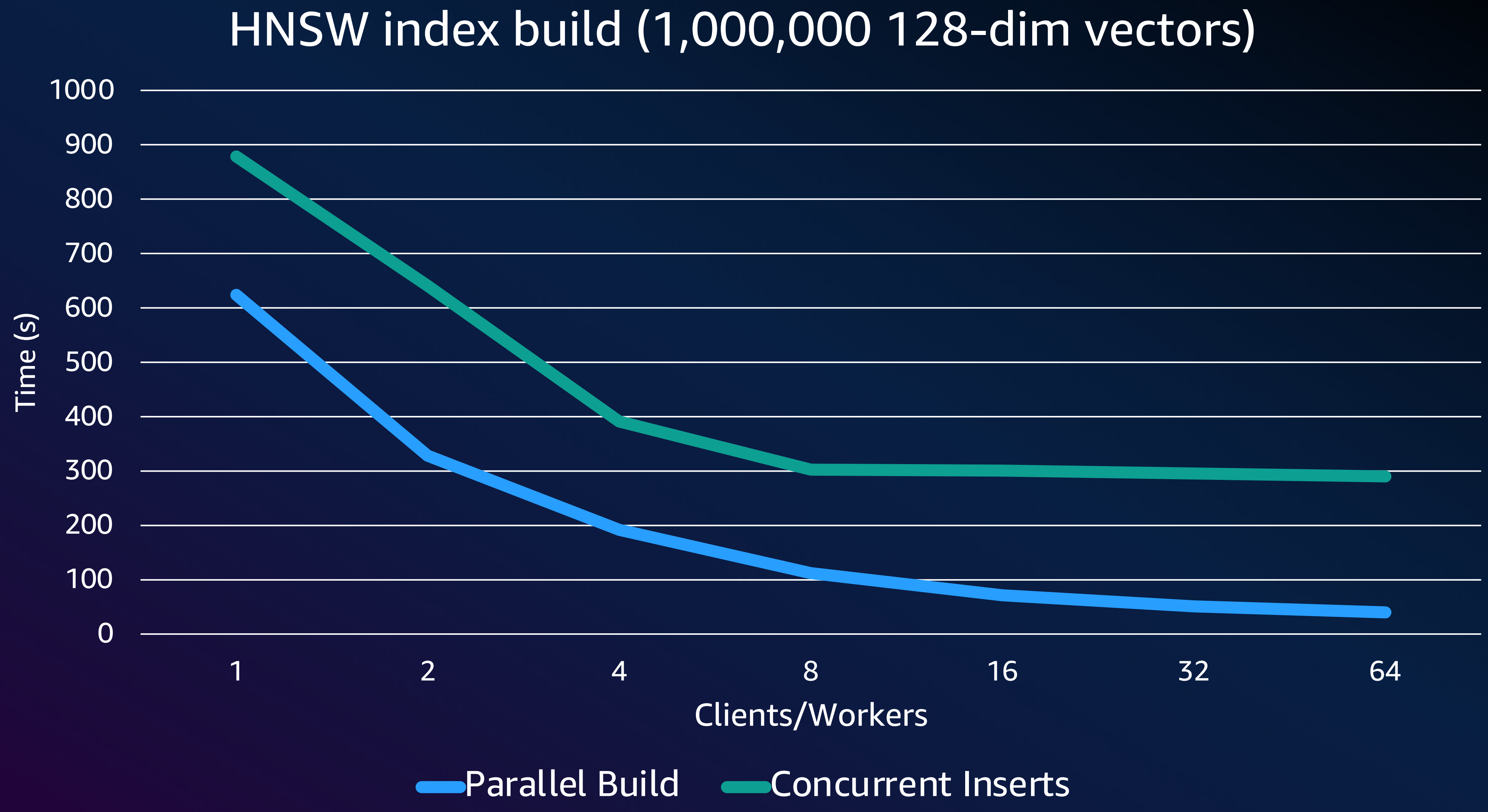
- Innovation: pgvector HNSW implementation supports updates and deletes!



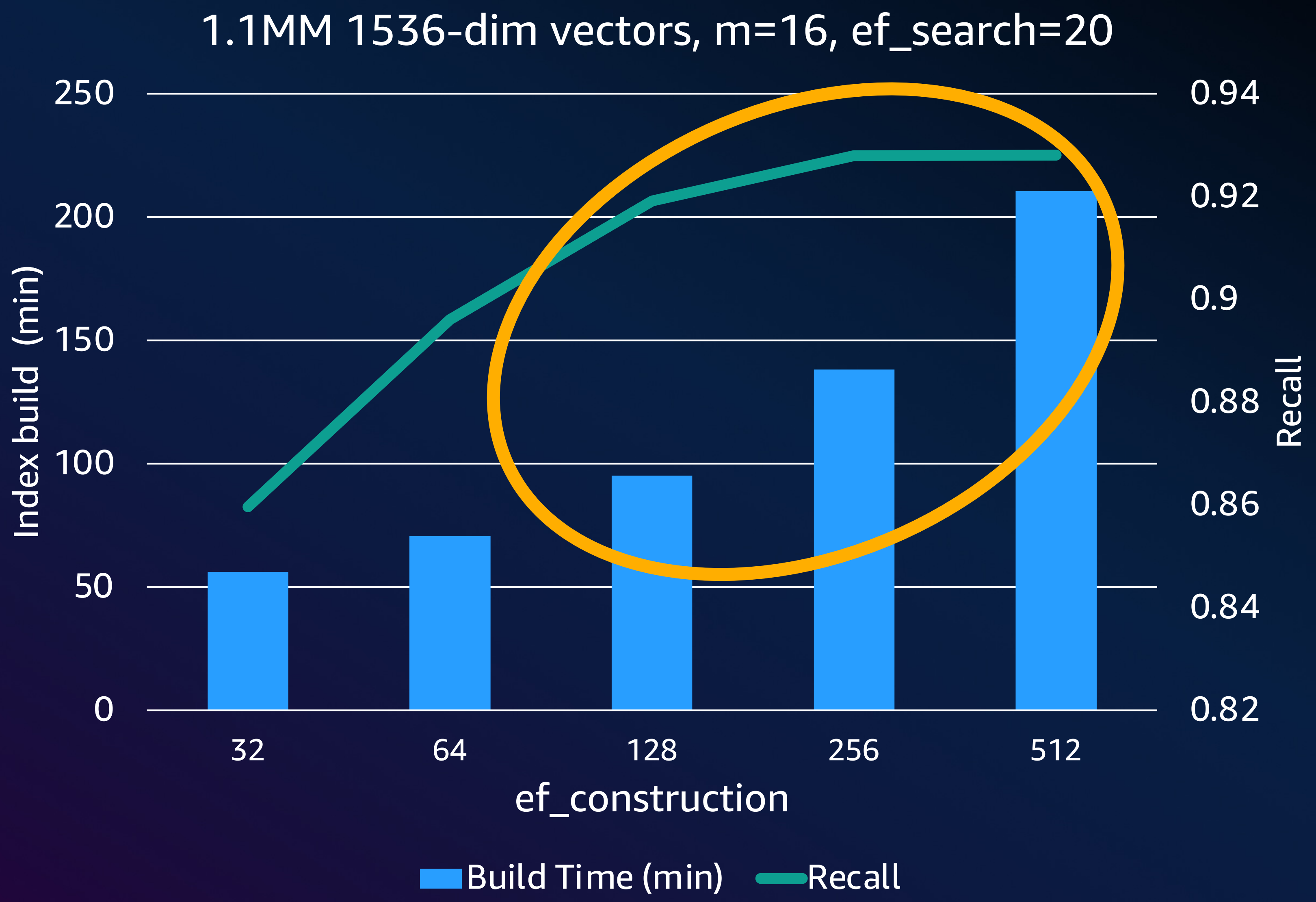
Phase 1: Hide

Phase 2: Repair

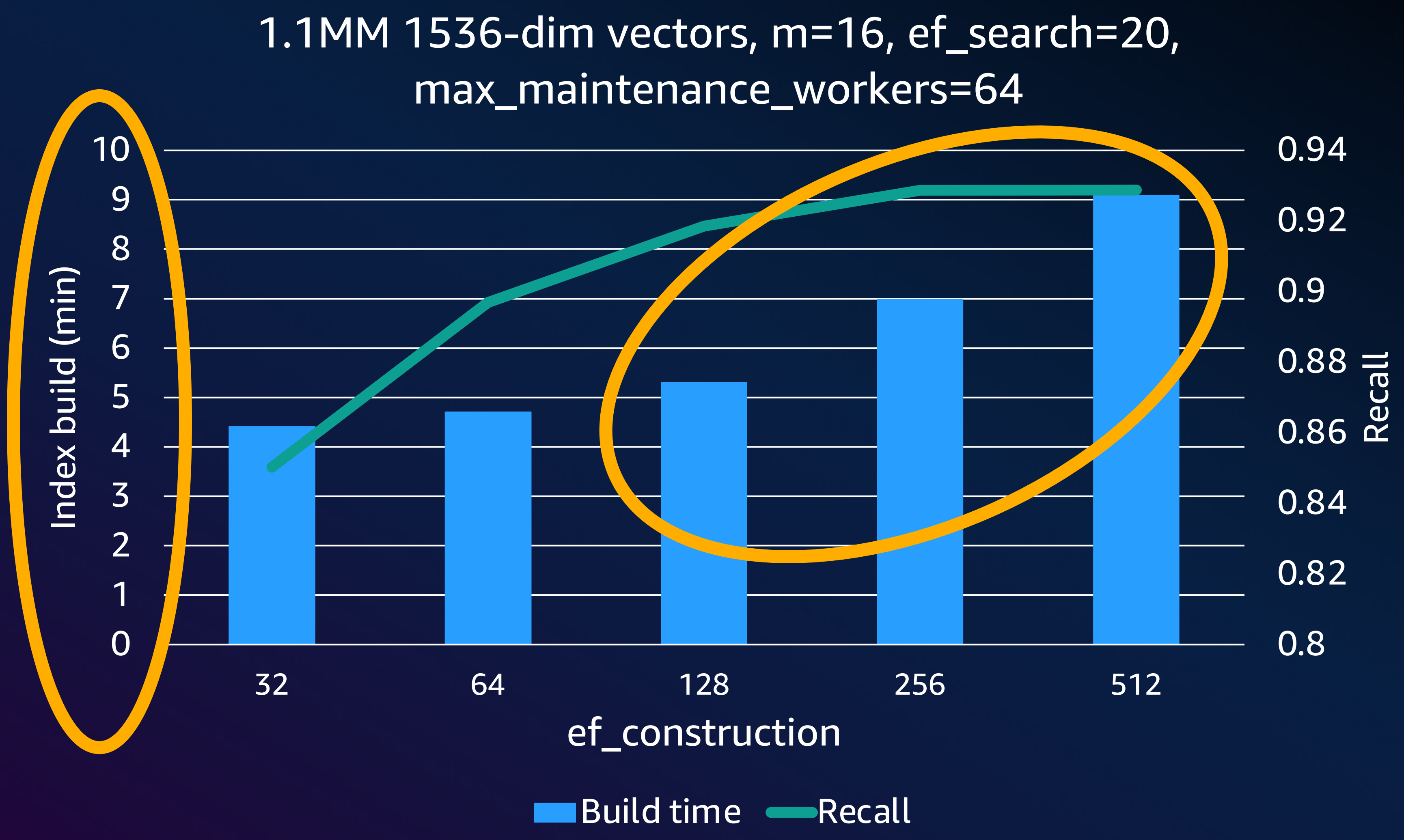
Impact of parallelism on HNSW build time



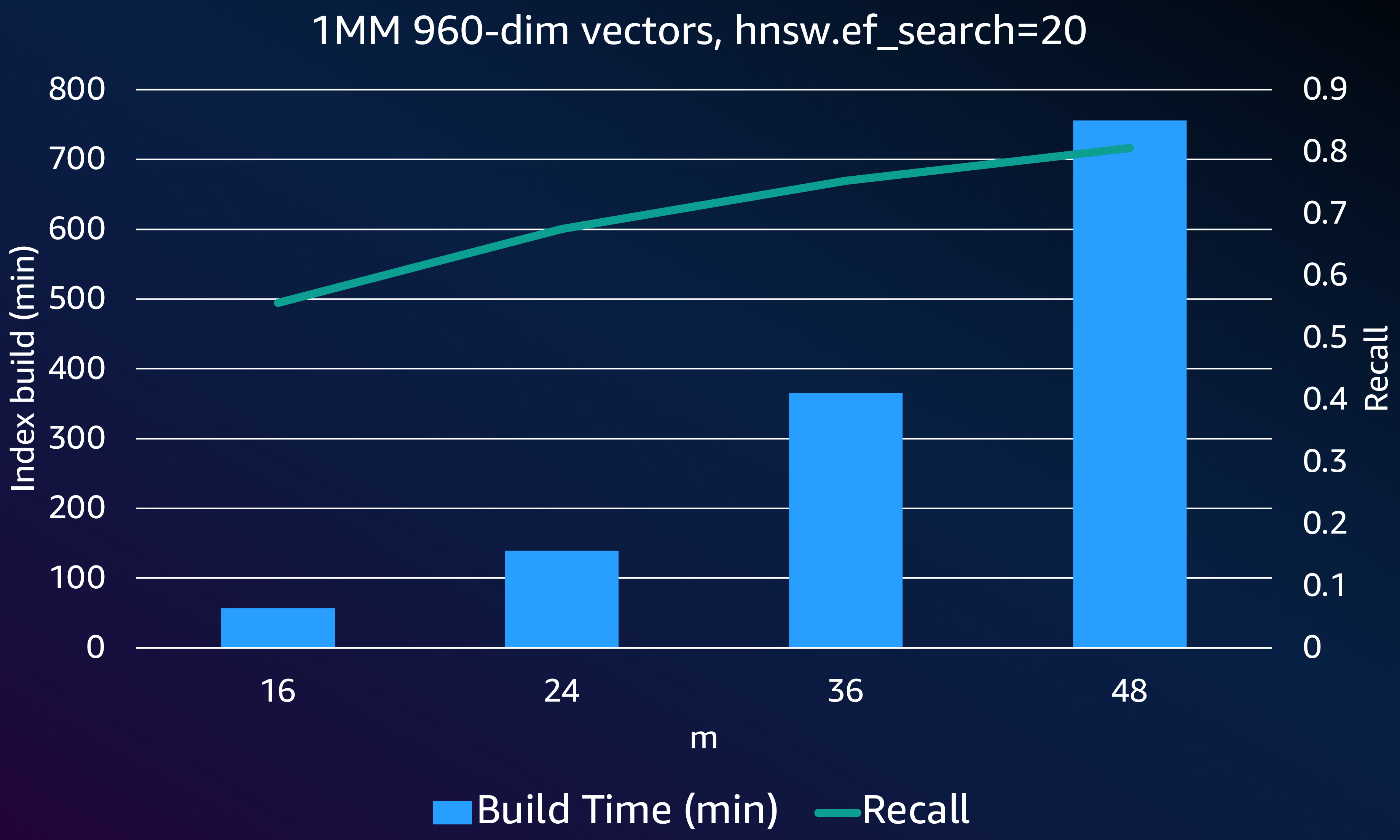
Why index build speed matters (serial build)



Why index build speed matters (parallel build)



How “m” impacts index build time & search quality



Best practices for building HNSW indexes

Start with m=16, ef_construction=256

pgvector (0.5.1) Start with empty table and use concurrent writes to accelerate builds

- INSERT or COPY

pgvector (0.6.0+) use parallel builds on a full table

- max_parallel_maintenance_workers

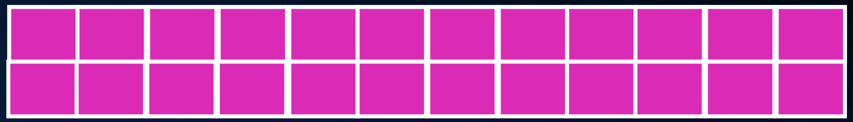
pgvector (0.7.0+) evaluate using quantization to decrease index size

Deep dive: Quantization

What is quantization?

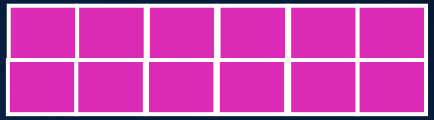
Flat

[0.0435122, -0.2304432, -0.4521324,
0.98652234, -0.1123234, 0.75401234]



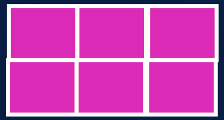
Scalar quantization (2-byte float)

[0.0432, -0.234, -0.452,0.986,
-0.112, 0.751]



Scalar quantization (1-byte uint)

[129, 99, 67, 244, 126, 230]



Binary quantization

[1, 0, 0, 1, 0, 1]

pgvector and scalar quantization (2 byte)

```
CREATE INDEX ON documents USING
      hnsw((embedding::halfvec(3072)) halfvec_cosine_ops);


SELECT id
FROM documents
ORDER BY embedding::halfvec(3072) <=> $1::halfvec(3072)
LIMIT 10;
```

Impact of scalar quantization

dbpedia-openai-1m-angular (1MM 1,536-dim); m=16; ef_construction=256

	No quantization	2-byte float quantization
Index size (MB)	7734	3867
Index build time (s)	250	146
Recall @ ef_search=10	0.851	0.854
QPS @ ef_search=10	1154	1164
Recall @ ef_search=40	0.967	0.968
QPS @ ef_search=40	567	583
Recall @ ef_search=200	0.996	0.996
QPS @ ef_search=200	158	163

pgvector and binary quantization

```sql
CREATE INDEX ON documents USING
     hnsw ((binary_quantize(embedding)::bit(3072)) bit_hamming_ops);


SELECT i.id FROM (
     SELECT id, embedding <=> $1 AS distance
     FROM items
     ORDER BY
       binary_quantize(embedding)::bit(3072) <~> binary_quantize($1)
     LIMIT 800 -- bound by hnsw.ef_search
) i
ORDER BY i.distance
LIMIT 10;
```

Impact of binary quantization

dbpedia-openai-1m-angular (1MM 1,536-dim); m=16; ef_construction=256

	No quantization	Binary quantization/rerank
Index size (MB)	7734	473
Index build time (s)	250	49
Recall @ ef_search=10	0.851	0.604
QPS @ ef_search=10	1154	1687
Recall @ ef_search=40	0.967	0.916
QPS @ ef_search=40	567	883
Recall @ ef_search=200	0.996	0.990
QPS @ ef_search=200	158	236

Quantization takeaways

- Quantizing a vector may result in losing information
- Binary quantization works best for vectors with “bit diversity”
- Possible to add custom quantization functions

Best practices: Filtering

What is filtering?

```
SELECT id
FROM products
WHERE products.category_id = 7
ORDER BY :'q' <-> products.embedding
LIMIT 10;
```

How filtering impacts ANN queries

PostgreSQL may choose to not use the index

Uses an index, but does not return enough results

Filtering occurs after using the index

Do I need an HNSW index for a filter?

Does the filter use a B-Tree (or other index) to reduce the dataset?

How many rows does the filter remove?

Do I want exact results or approximate results?

Pre-v0.8.0 filtering strategies

- Partial index
- Partition

```
CREATE INDEX ON docs
  USING hnsw(embedding vector_l2_ops)
  WHERE category_id = 7;
---
CREATE TABLE docs_cat7
  PARTITION OF docs
  FOR VALUES IN (7);

CREATE INDEX ON docs_cat7
 USING hnsw(embedding vector_l2_ops);
```

Ongoing work

Performance and filtering improvements

Reduced memory usage for HNSW lookups

Performance improvements to insert / on-disk HNSW index builds

Better planner cost estimates for HNSW lookups

Iterative / streaming scans => better performance / avoids overfiltering

Iterative scans and streaming

	Recall		QPS (peak concurrency)		
ef_search	**0.7.4**	**0.8.0 (planned)**	**0.7.4**	**0.8.0 (planned)**	**%**
20	0.874	0.870	27,608	32,810	19%
40	0.934	0.928	19,538	22,235	14%
60	0.956	0.953	14,554	16,839	16%
80	0.968	0.965	10,961	13,410	22%
220	0.989	0.990	4,880	5,506	13%

r7gd.16xlarge (64 vCPU, 512 GiB RAM)
OpenAI 5MM (1536d)
k=10
HNSW – m=16, ef_construction=256
No quantization

https://github.com/zilliztech/VectorDBBench

Iterative scans and streaming

	Recall		QPS (peak concurrency)		
ef_search	0.7.4	0.8.0 (planned)	0.7.4	0.8.0 (planned)	%
80	0.783	0.951	10,626	6,840	-36%
100	0.920	0.921	9,023	10,378	15%
120	0.934	0.934	8,273	8,668	5%
155	0.950	0.950	6,668	6,983	5%
585	0.990	0.990	2,323	2,791	20%

r7gd.16xlarge (64 vCPU, 512 GiB RAM)
OpenAI 5MM (1536d)
k=100
HNSW – m=16, ef_construction=256
No quantization

https://github.com/zilliztech/VectorDBBench

Post-v0.8.0 filtering strategies

- Low selectivity – use alternative index (B-tree, GIN)
  - "Too many filters" => JSOB + GIN

- HNSW/IVFFlat + iterative scans
  - `hnsw.streaming` / `ivfflat.streaming`

- Streaming can improve query performance with quantization

pgvector roadmap

- Enhanced index-based filtering (in progress)
- Parallelized vacuum
- Parallel query
- Improved async pushdown for postgres_fdw
- TOAST/storage updates

Conclusion

Conclusion

Primary design decision: **Query performance** and **recall**

Determine where to invest: **Storage**, **compute**, **indexing strategy**

Plan for today and tomorrow: vector search capabilities are rapidly evolving

Thank you!

Jonathan Katz
jkatz@amazon.com
@jkatz05