Vectors are the new JSON

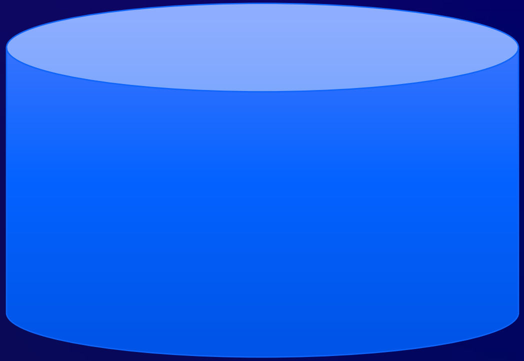
Jonathan Katz

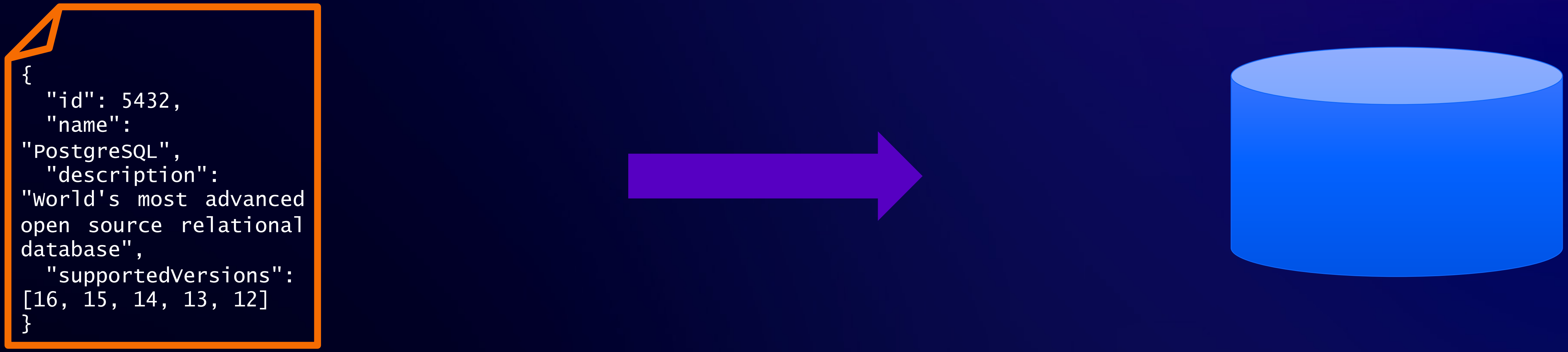
(he/him/his)
Principal Product Manager – Technical
AWS

```json
{
  "id": 5432,
  "name": "PostgreSQL",
  "description": "World's most advanced open source relational database",
  "supportedVersions": [16, 15, 14, 13, 12]
}
```

```
{
  "id": 5432,
  "name":
"PostgreSQL",
  "description":
"World's most advanced
open source relational
database",
  "supportedVersions":
[16, 15, 14, 13, 12]
}
```

id	5432
name	PostgreSQL
description	world's most...
supportedVersions	[16,15,14,13,12]

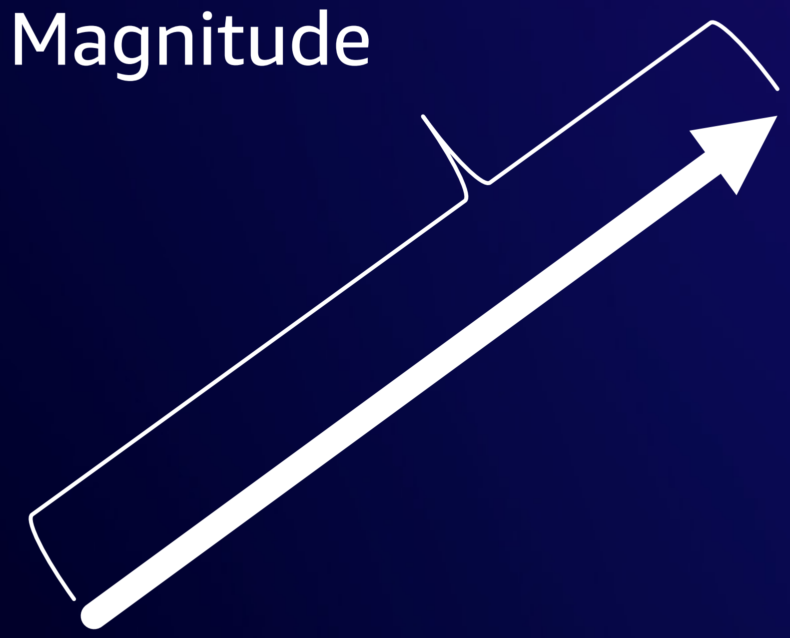



{
"id": 5432,
"name":
"PostgreSQL",
"description":
"World's most advanced
open source relational
database",
"supportedVersions":
[16, 15, 14, 13, 12]
}

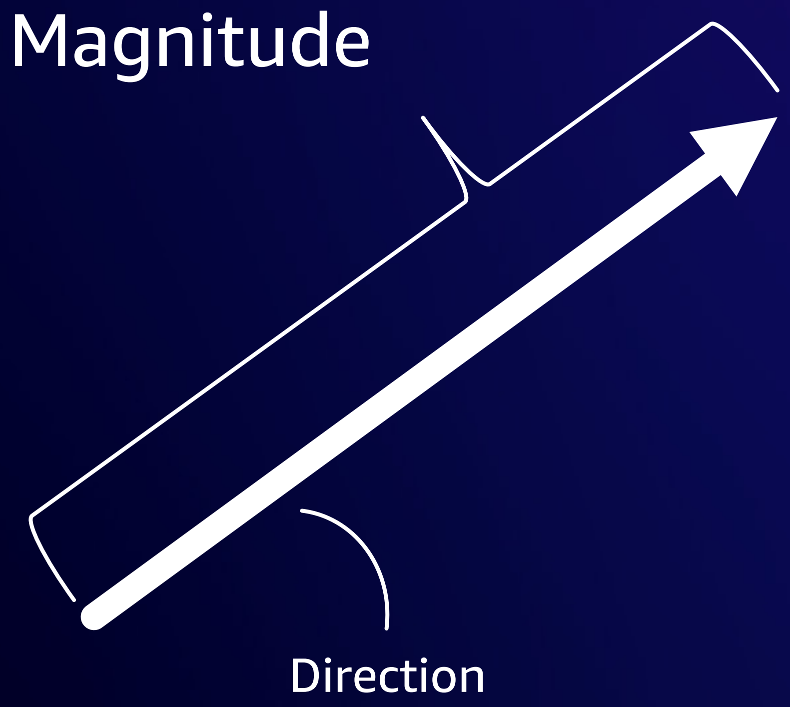
Timeline of JSON storage

- 2000-2001: JSON invented
- 2004: AJAX model emerges in wider deployments
- 2006: RFC 4627 publishes JSON format
- 2006-2009: JSON-specific data stores emerge
- 2012: PostgreSQL adds support for JSON (text)
- 2013: ECMA-404 standardizes JSON
- 2014: PostgreSQL adds support for JSONB (binary)
- 2017: SQL/JSON standard published
- 2019: PostgreSQL adds SQL/JSON path language
- 2023: PostgreSQL adds SQL/JSON constructors and predicates

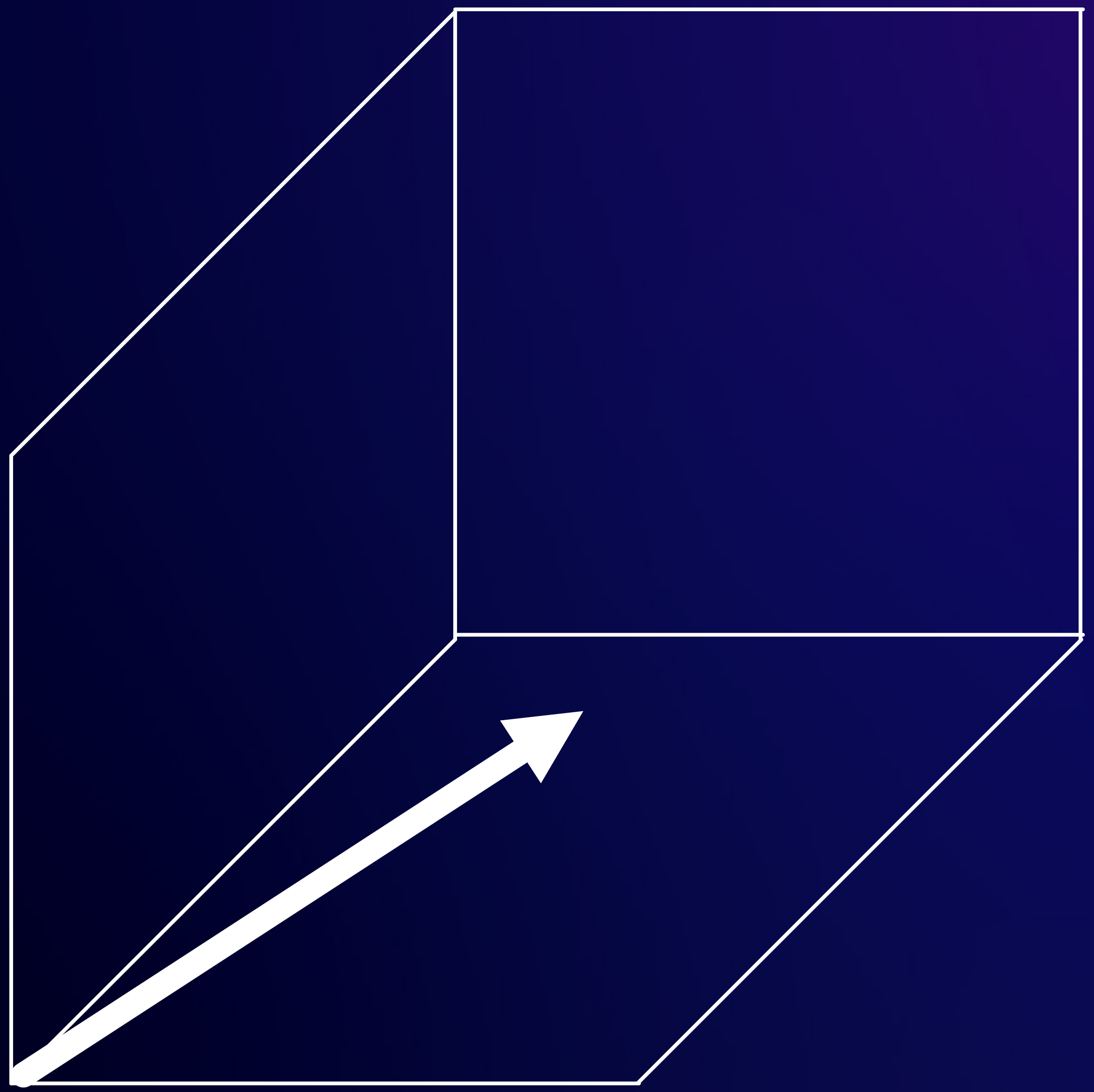
[0.5, 0.5]



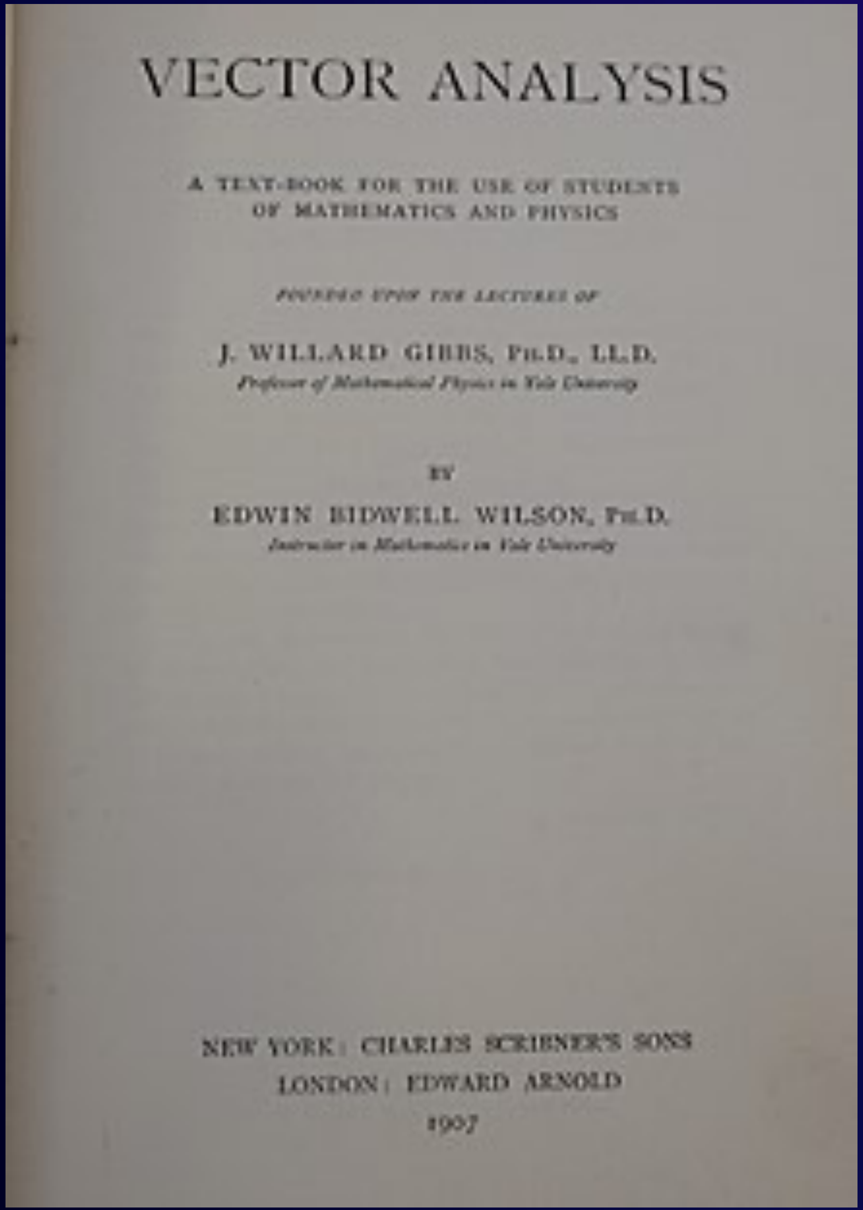
$$\| [0.5, 0.5] \| = \sqrt{(0.5^2 + 0.5^2)} = \mathbf{0.70710}$$



[0.5, 0.5]



[0.5, 0.5, 0.5]


VECTOR ANALYSIS

A TEXT-BOOK FOR THE USE OF STUDENTS
OF MATHEMATICS AND PHYSICS

FOUNDED UPON THE LECTURES OF

J. WILLARD GIBBS, PH.D., LL.D.
Professor of Mathematical Physics in Yale University

BY

EDWIN BIDWELL WILSON, PH.D.
Instructor in Mathematics in Yale University

NEW YORK: CHARLES SCRIBNER'S SONS
LONDON: EDWARD ARNOLD
1907

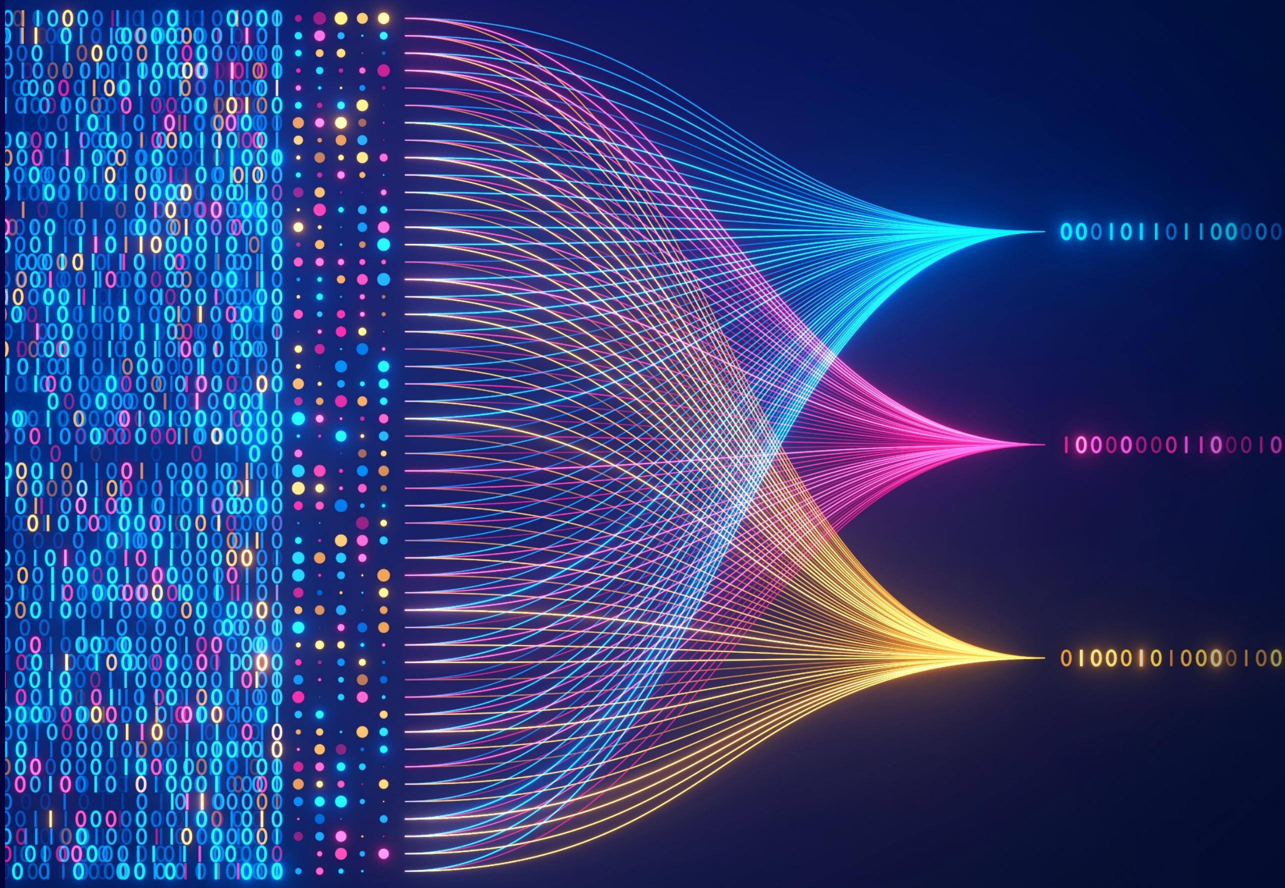
Generative AI is powered by foundation models

Pre-trained on vast amounts of unstructured data

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

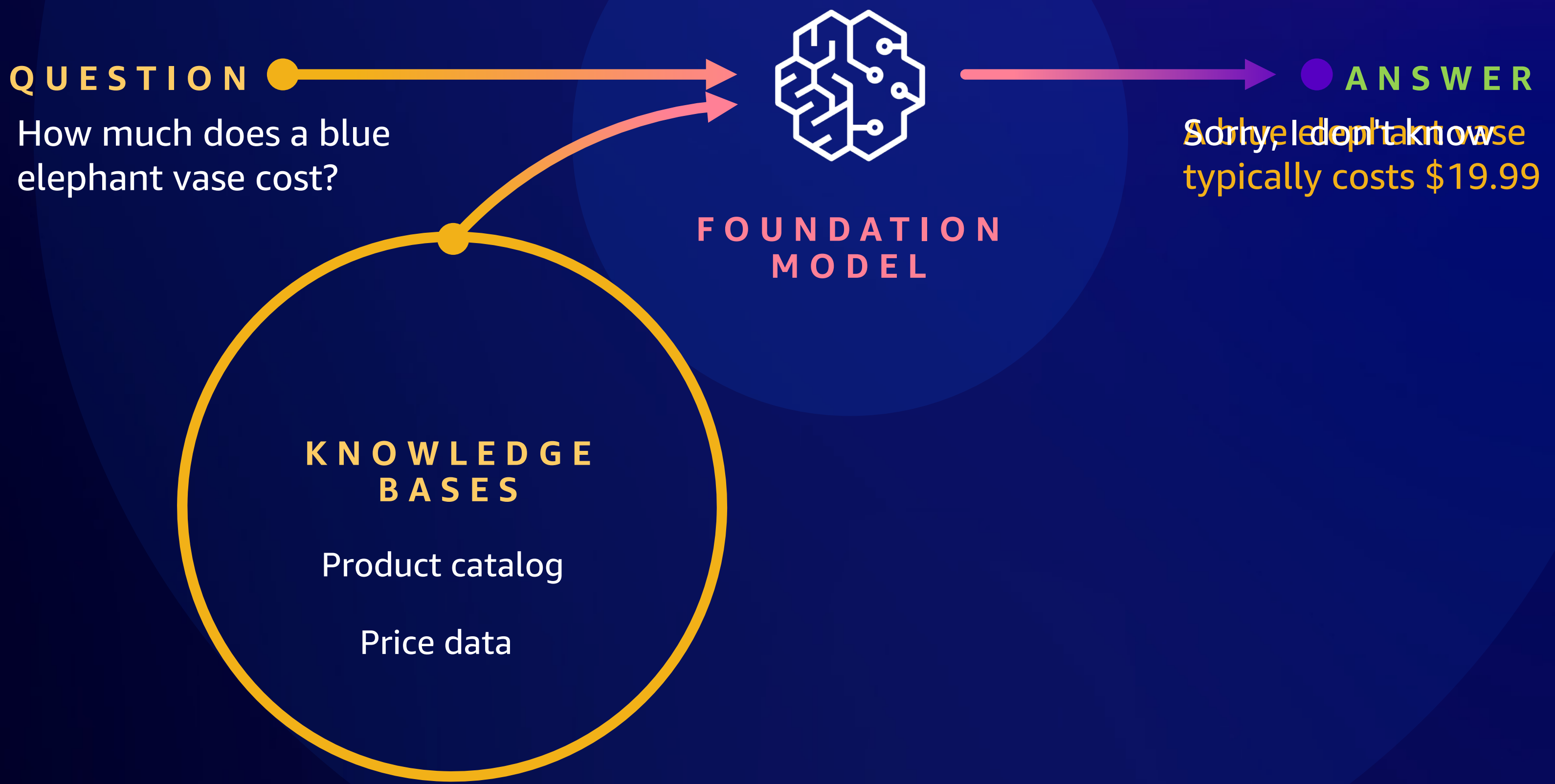
Can be applied in a wide range of contexts

Customize FMs using your data for domain specific tasks

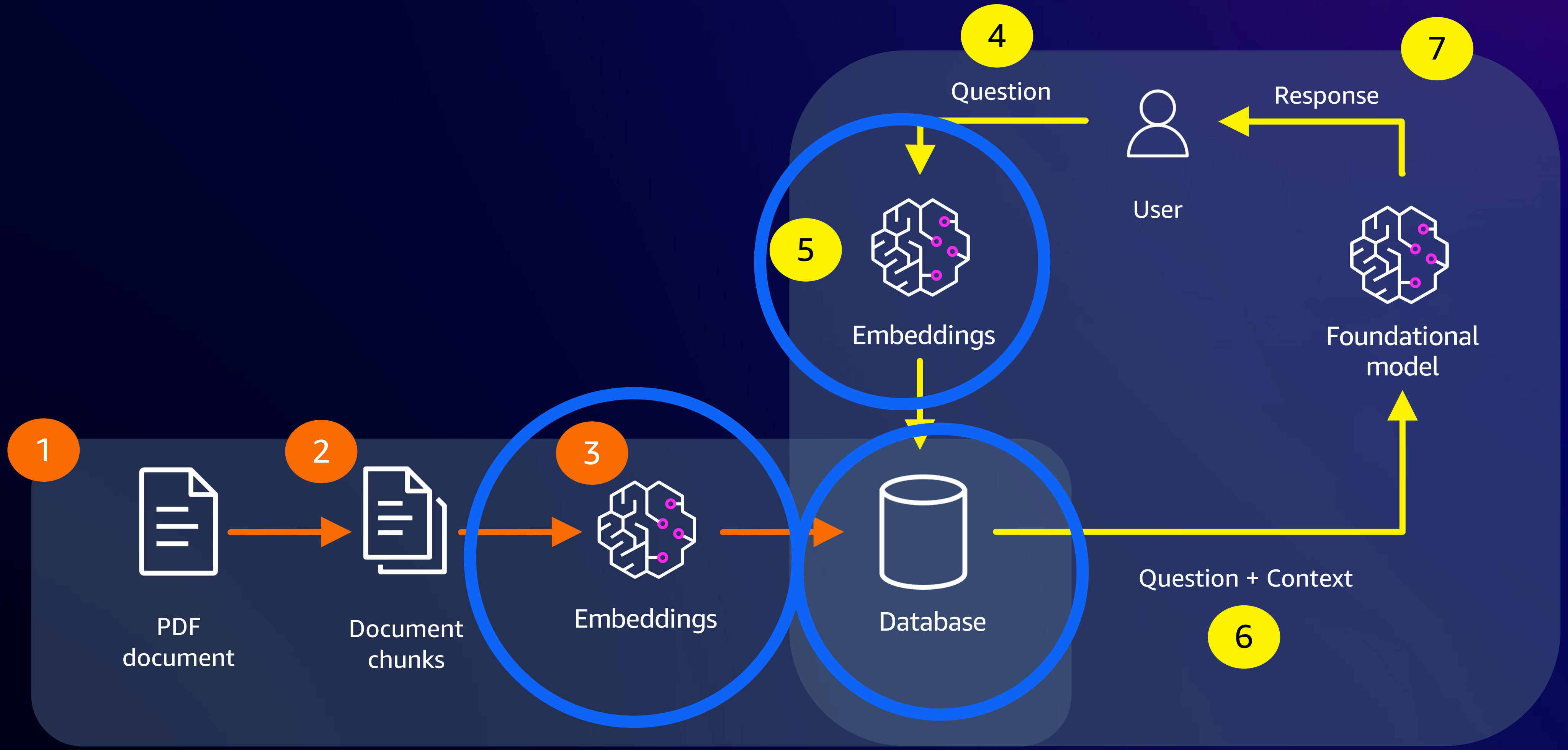


Retrieval Augmented Generation (RAG)

Configure FM to interact with your data

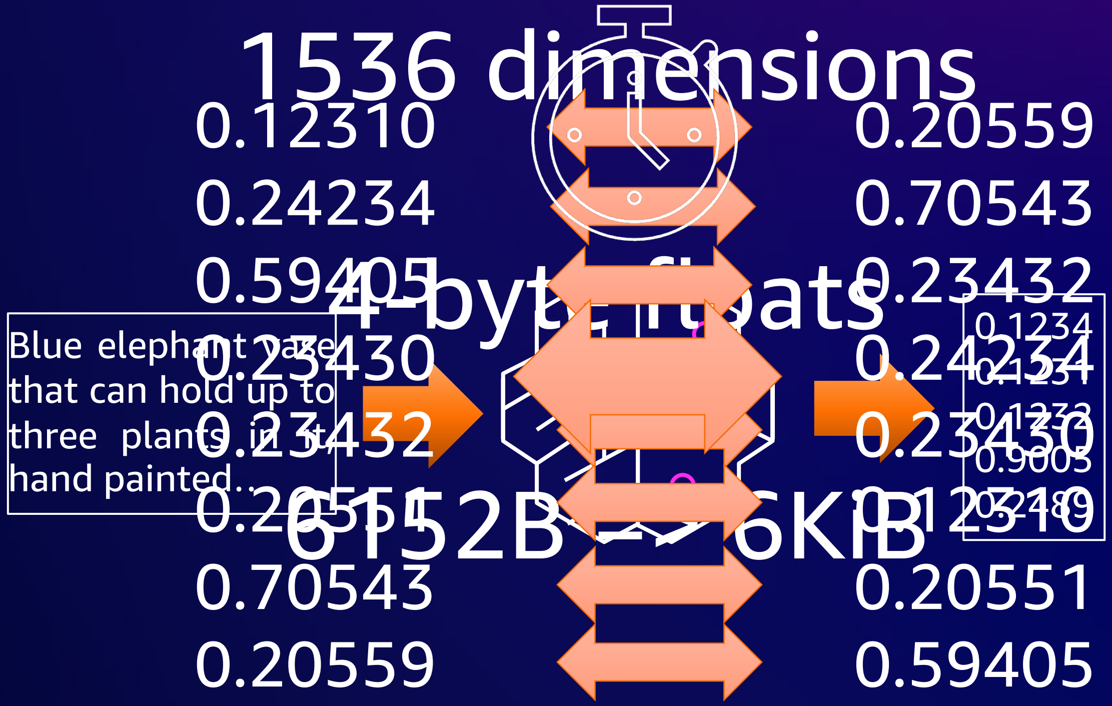


The role of vectors in RAG



Challenges with vectors

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



1,000,000 => 5.7GB

Approximate nearest neighbor (ANN)

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

Recall: 80%

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 tradeoffs: indexing, query time, schema design?**

PostgreSQL as a vector store

Why use PostgreSQL for vector searches?

- Existing client libraries work without modification
- Convenient to co-locate app + AI/ML data in same database
- PostgreSQL acts as persistent transactional store while working with other vector search systems

Native vector support in PostgreSQL

- ARRAY data type
  - Multiple data types (int4, int8, float4, float8)
  - “Unlimited” dimensions
  - No native distance operations
    - Can add using Trusted Language Extensions + PL/Rust
  - No native indexing

- Cube data type
  - float8 values
  - Euclidean, Manhattan, Chebyshev distances
  - K-NN GiST index – exact nearest neighbor search
  - Limited to 100 dimensions

What is pgvector?

An open source extension that:

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

`vector` data type

Co-locate with embeddings

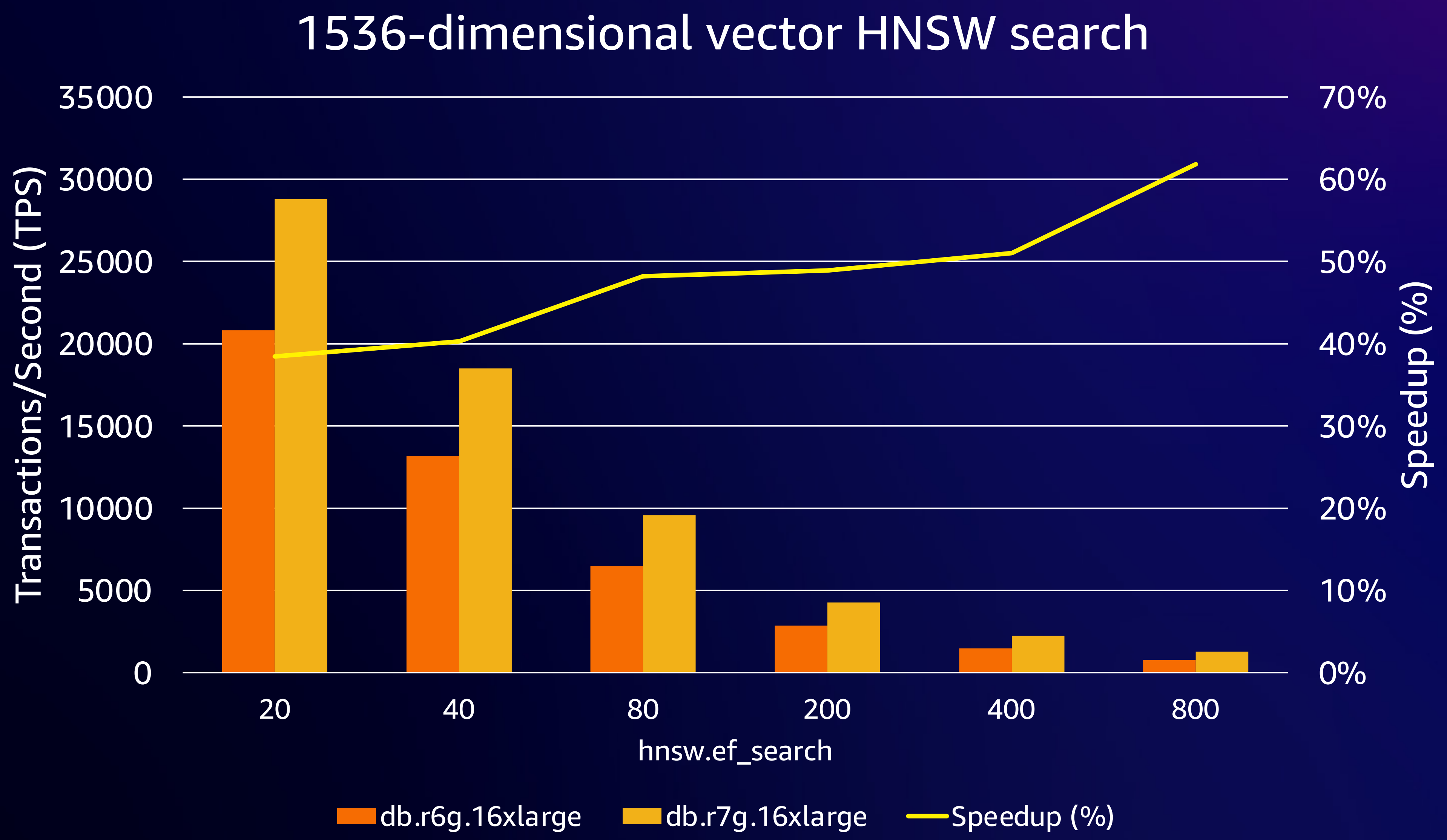
Exact nearest neighbor (K-NN)
Approximate nearest neighbor (ANN)

Supports IVFFlat/HNSW indexing

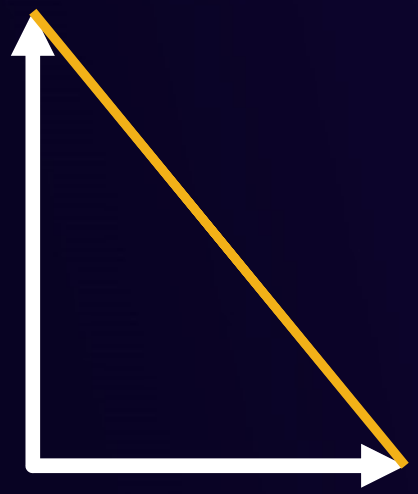
Distance operators (<->, <=>, <#>)

github.com/pgvector/pgvector

Understanding pgvector performance



pgvector distance operations



<->
Euclidean/L2

<=>
Cosine distance

<#>
Inner product

How does pgvector index a vector?

0.0234
0.093
-0.9123
0.1055

Valid?

✅ Same dimensions?
✅ Magnitude > 0?

Normalized?

🛠️ If not, normalize

0.0253
0.1007
-0.9880
0.1142

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

pgvector strategies and best practices

Best practices for pgvector

Storage strategies

HNSW strategies

IVFFlat strategies

Filtering

pgvector storage strategies

Understanding TOAST in PostgreSQL

- TOAST (The Oversized-Attribute Storage Technique) is a mechanism for storing data larger than 8KB
- By default, PostgreSQL “TOASTs” values over 2KB
- 510-dim 4-byte float vector

PostgreSQL column storage types

- `PLAIN`: Data stored inline with table
- `EXTENDED`: Data stored/compressed in TOAST table when threshold exceeded (pgvector default)
- `EXTERNAL`: Data stored in TOAST table when threshold exceeded
- `MAIN`: Data stored compressed inline with table

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

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

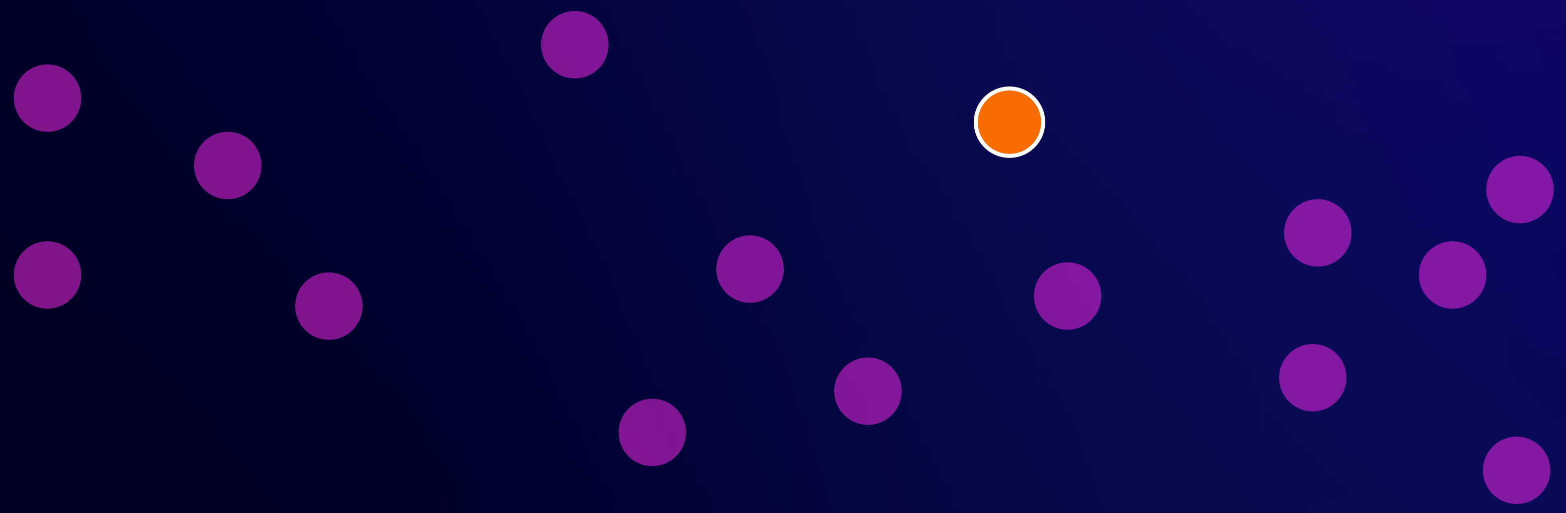
```
SET min_parallel_table_scan_size TO 1
```

HNSW strategies

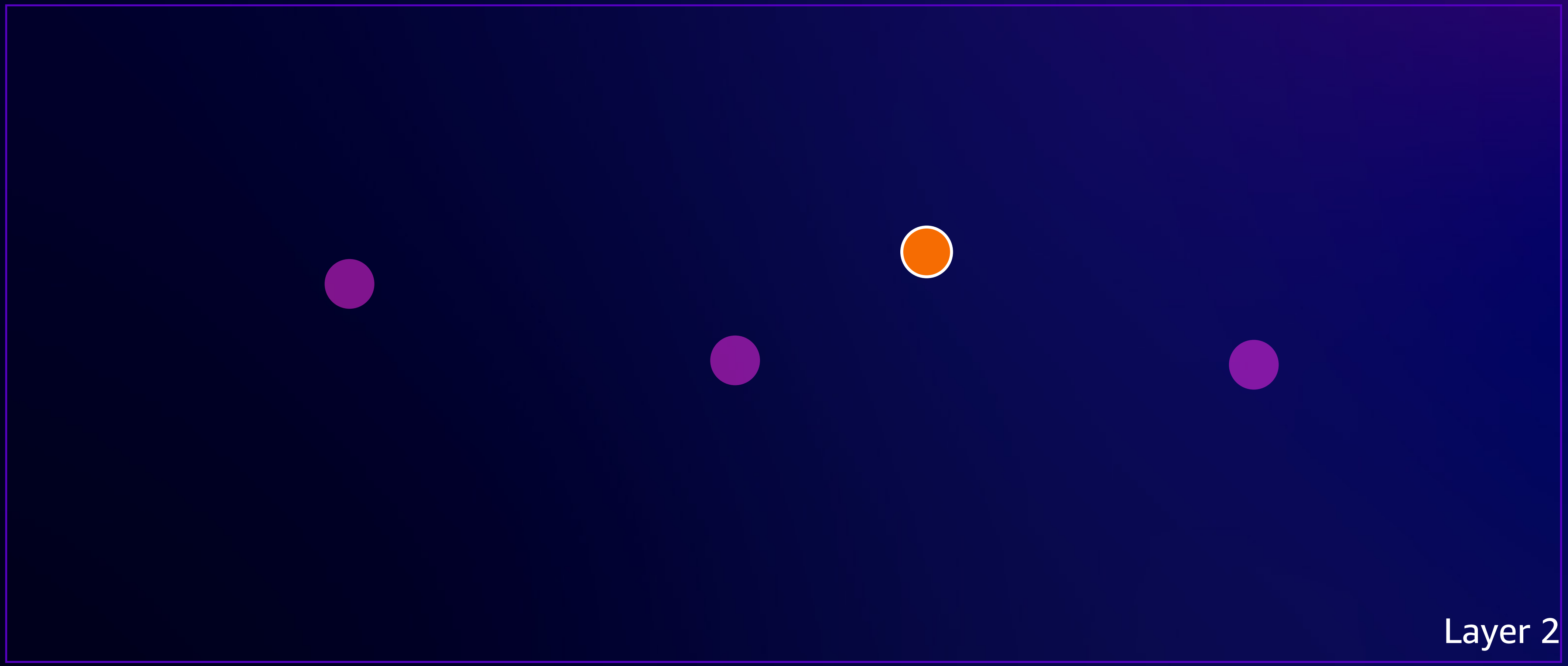
HNSW index building parameters

- `m`
  - Maximum number of bidirectional links between indexed vectors
  - Default: 16
- `ef_construction`
  - Number of vectors to maintain in “nearest neighbor” list
  - Default: 64

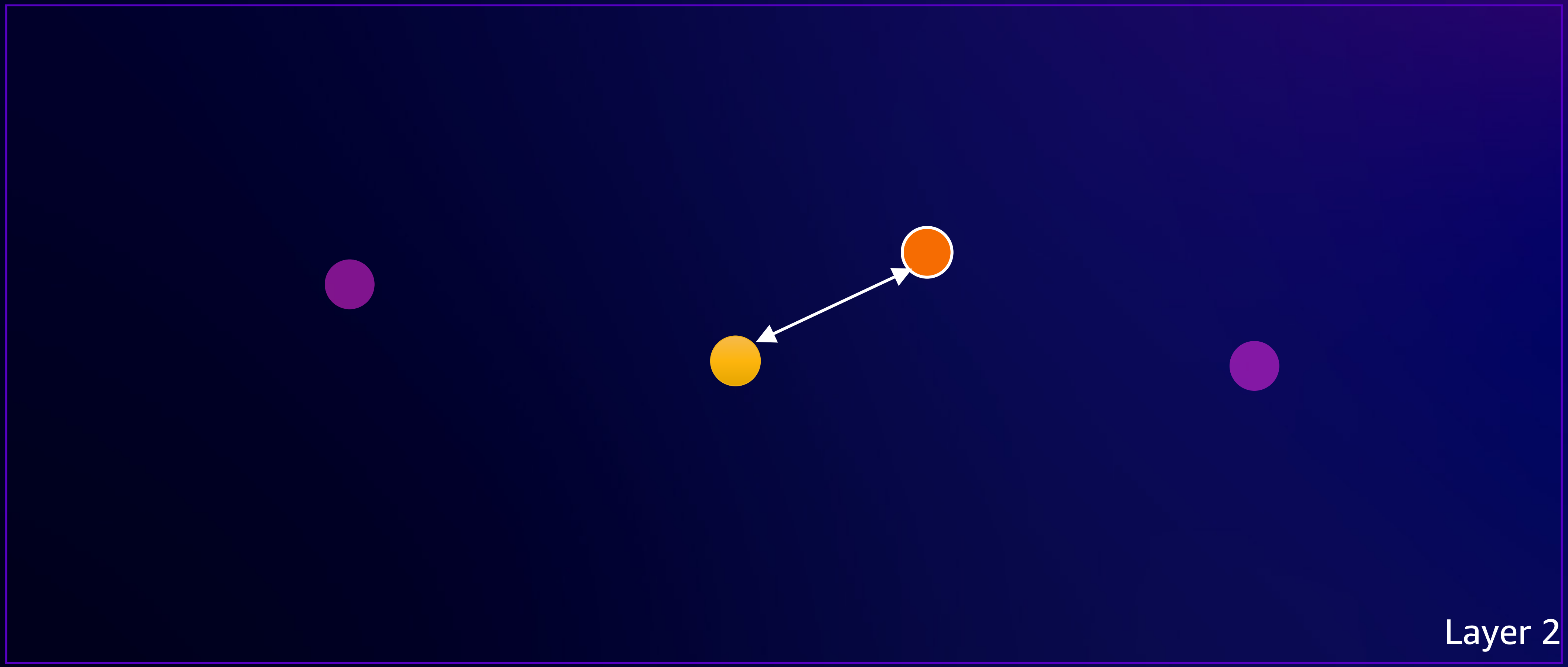
Building an HNSW index



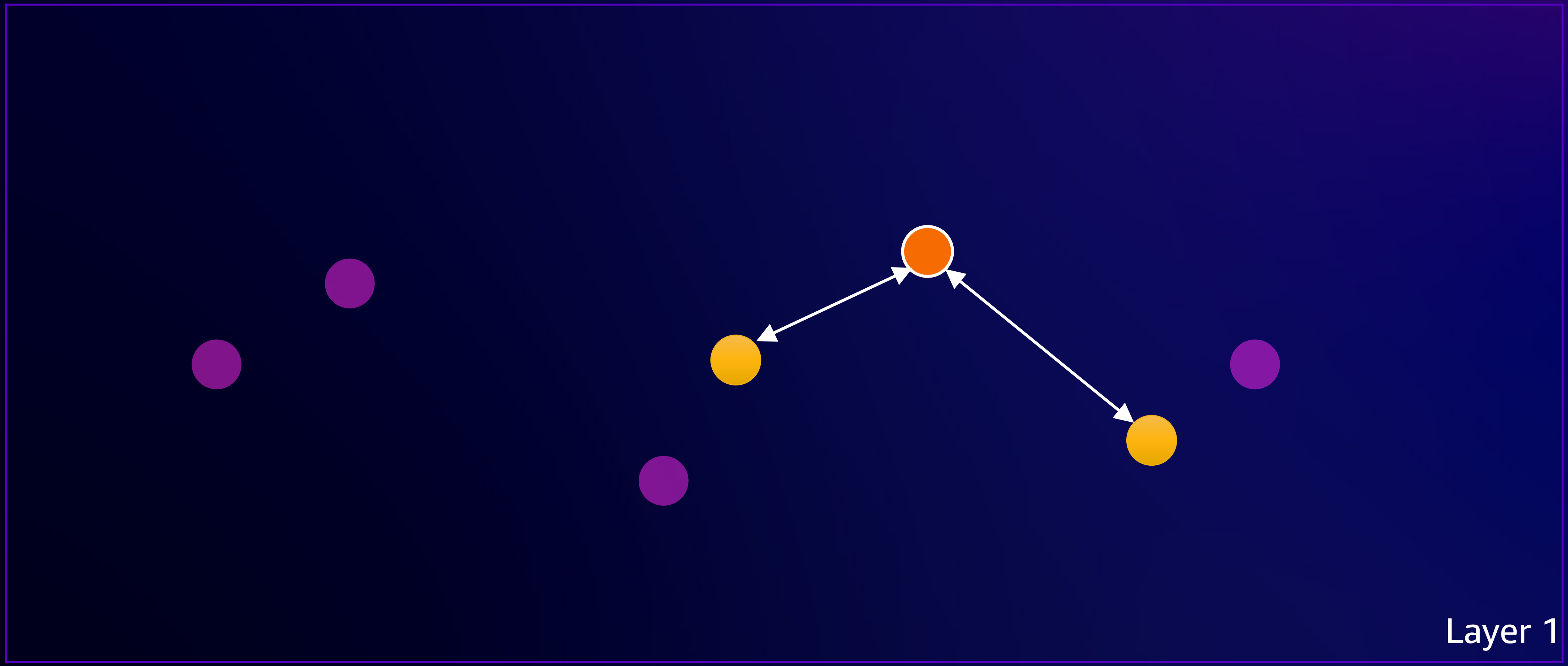
Building an HNSW index



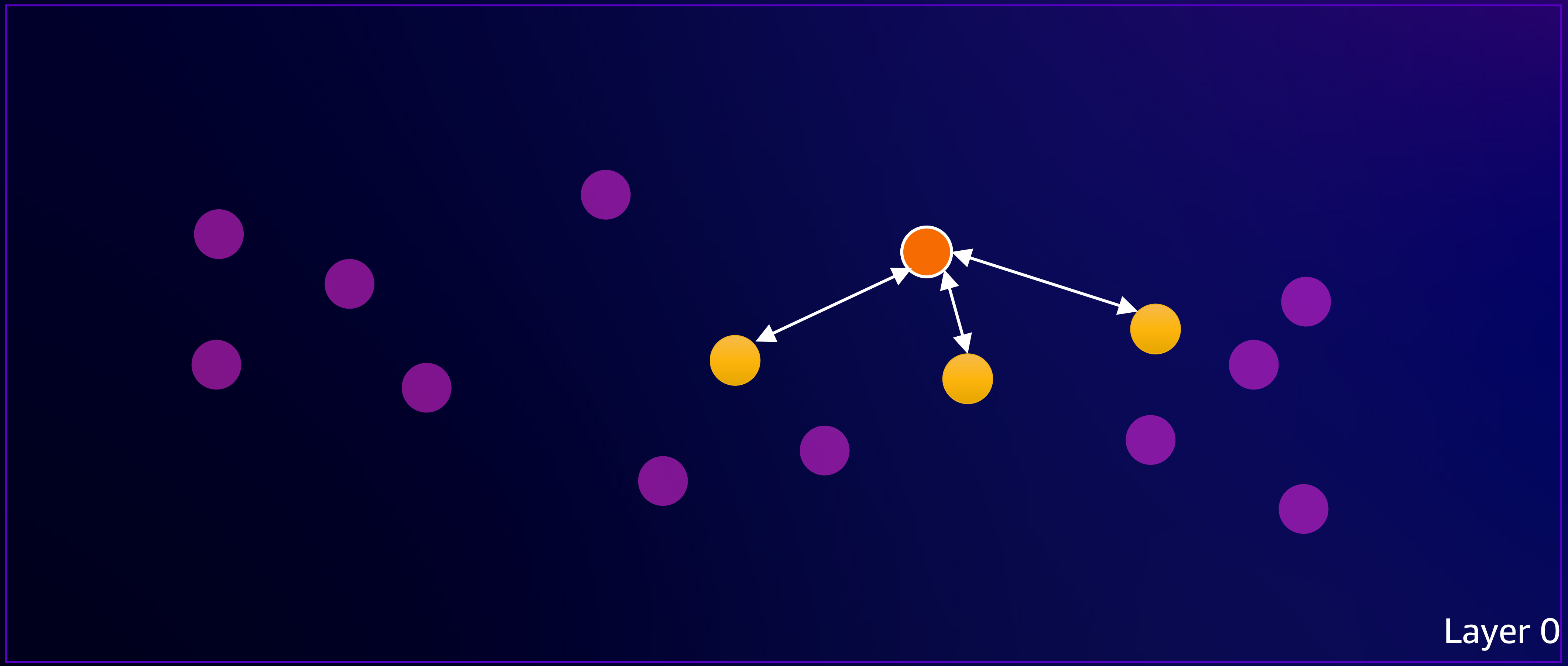
Building an HNSW index



Building an HNSW index



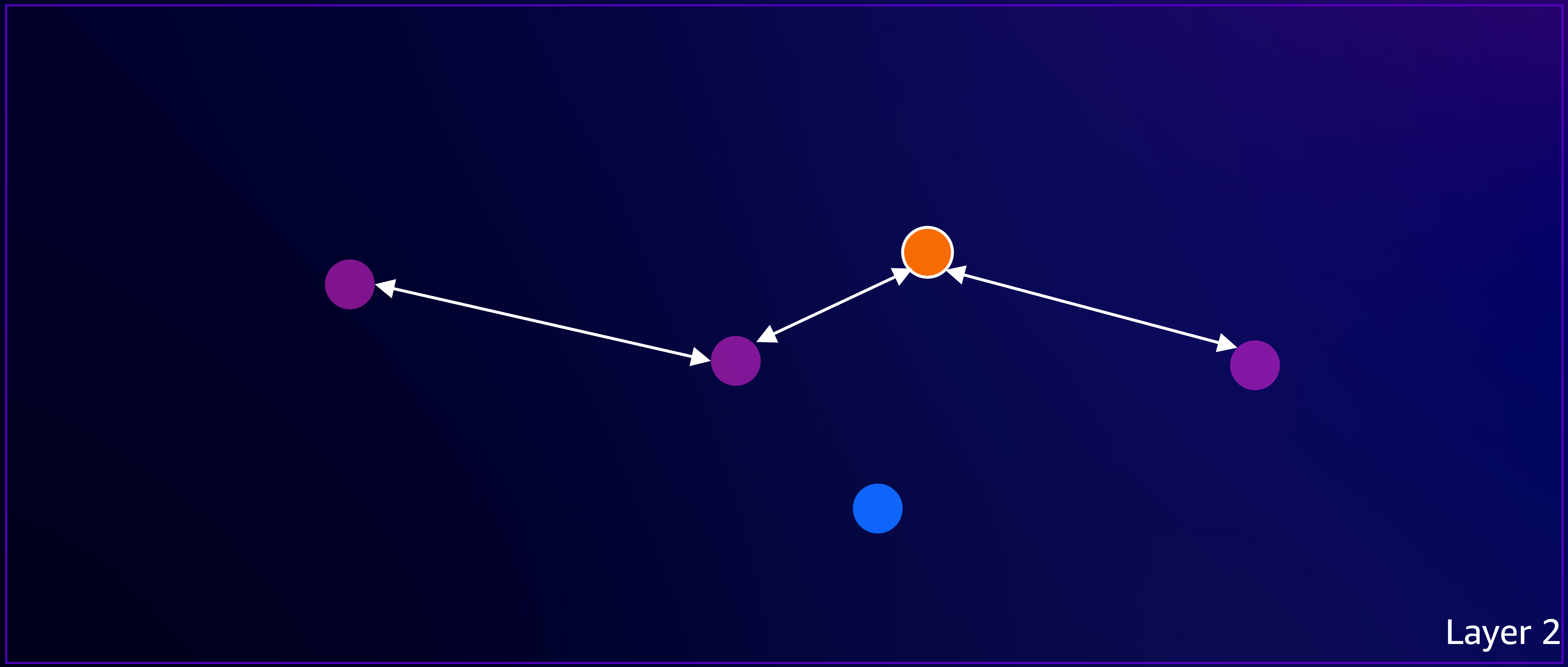
Building an HNSW index



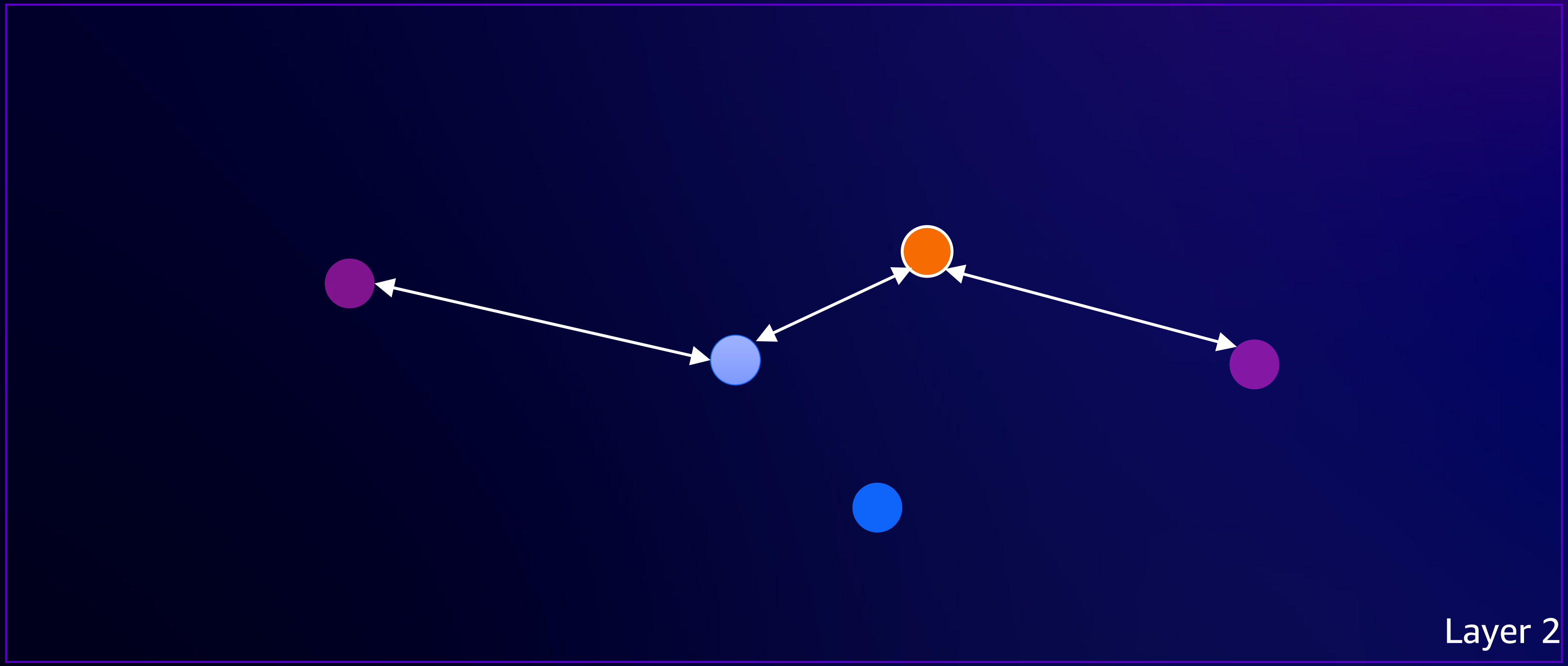
HNSW query parameters

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

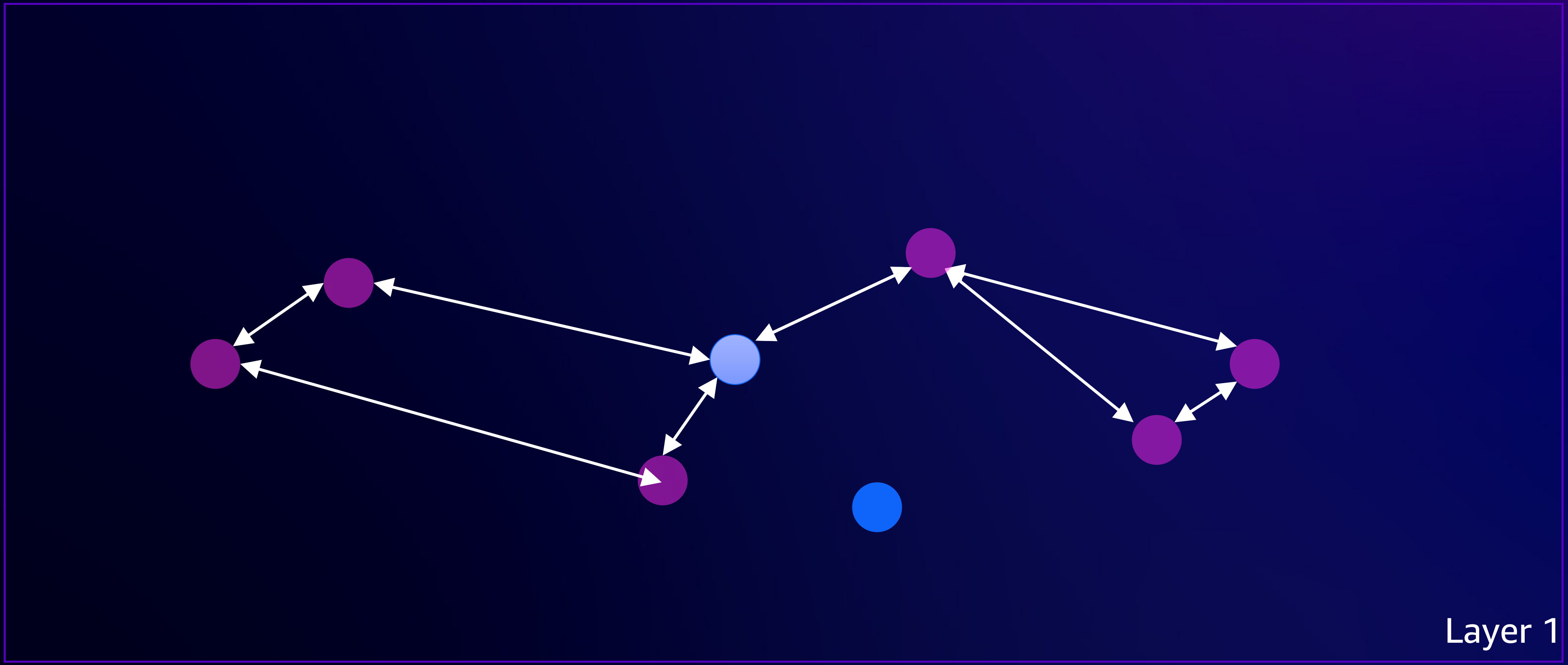
Querying an HNSW index



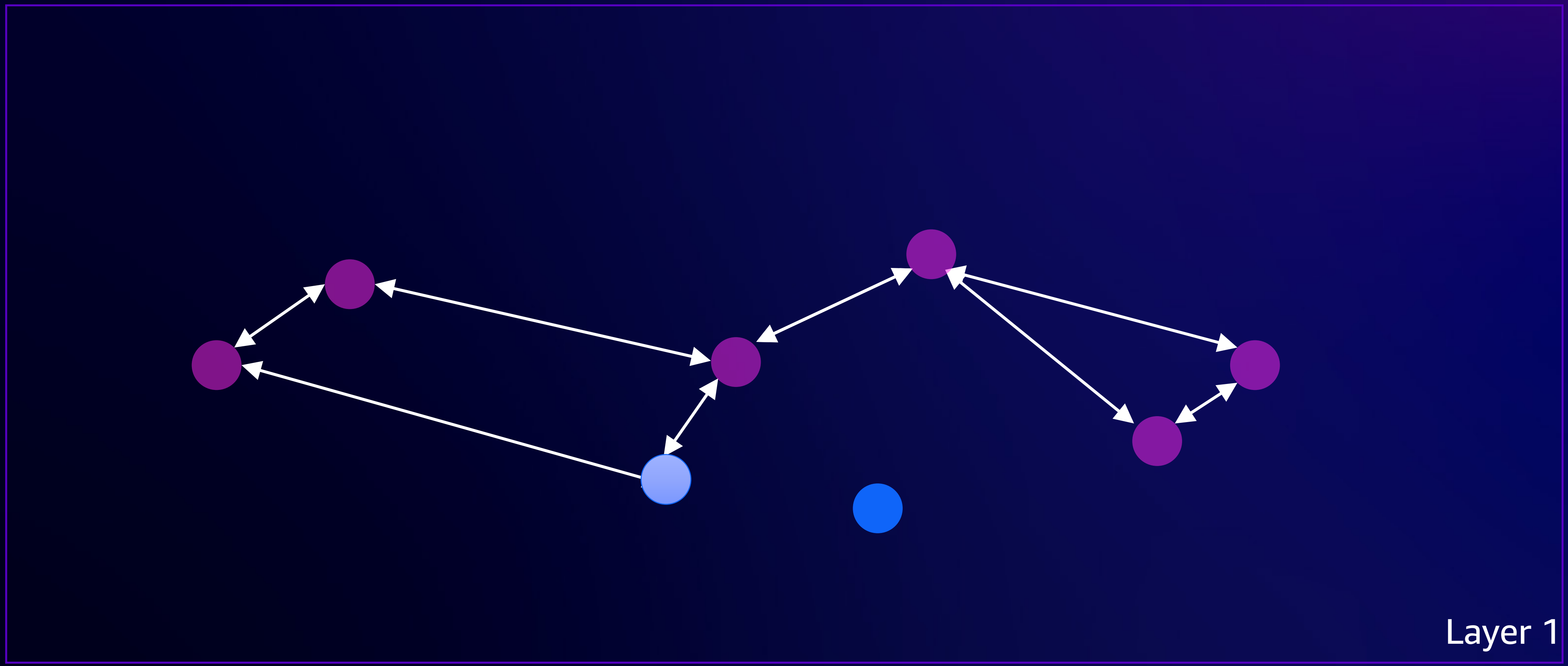
Querying an HNSW index



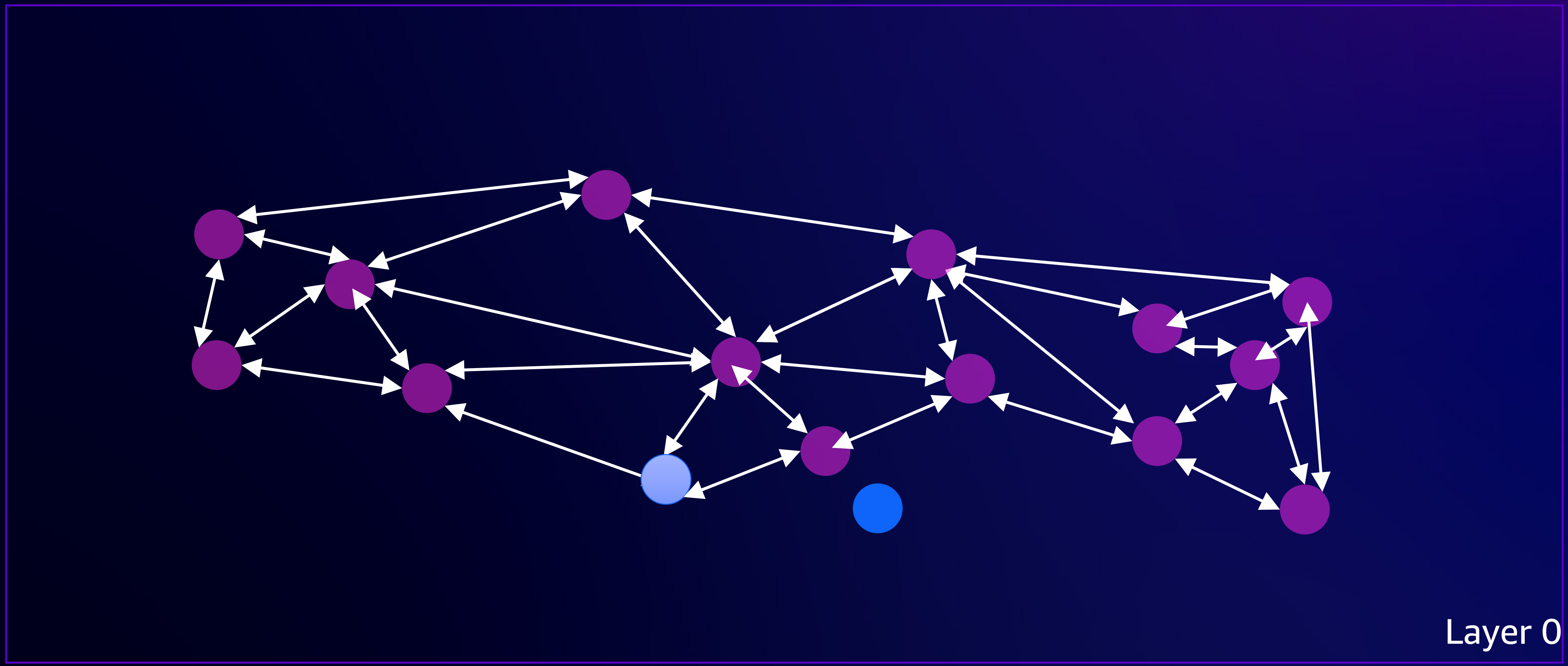
Querying an HNSW index



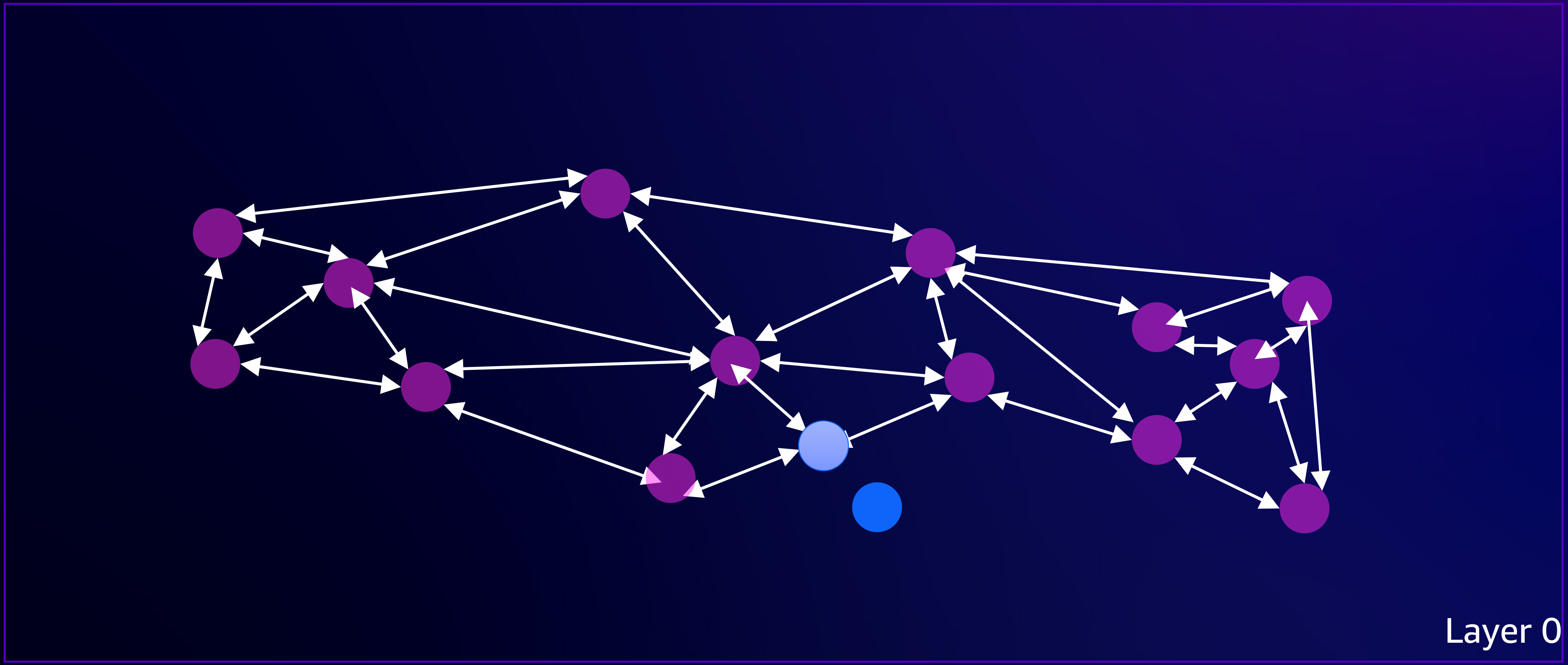
Querying an HNSW index



Querying an HNSW index



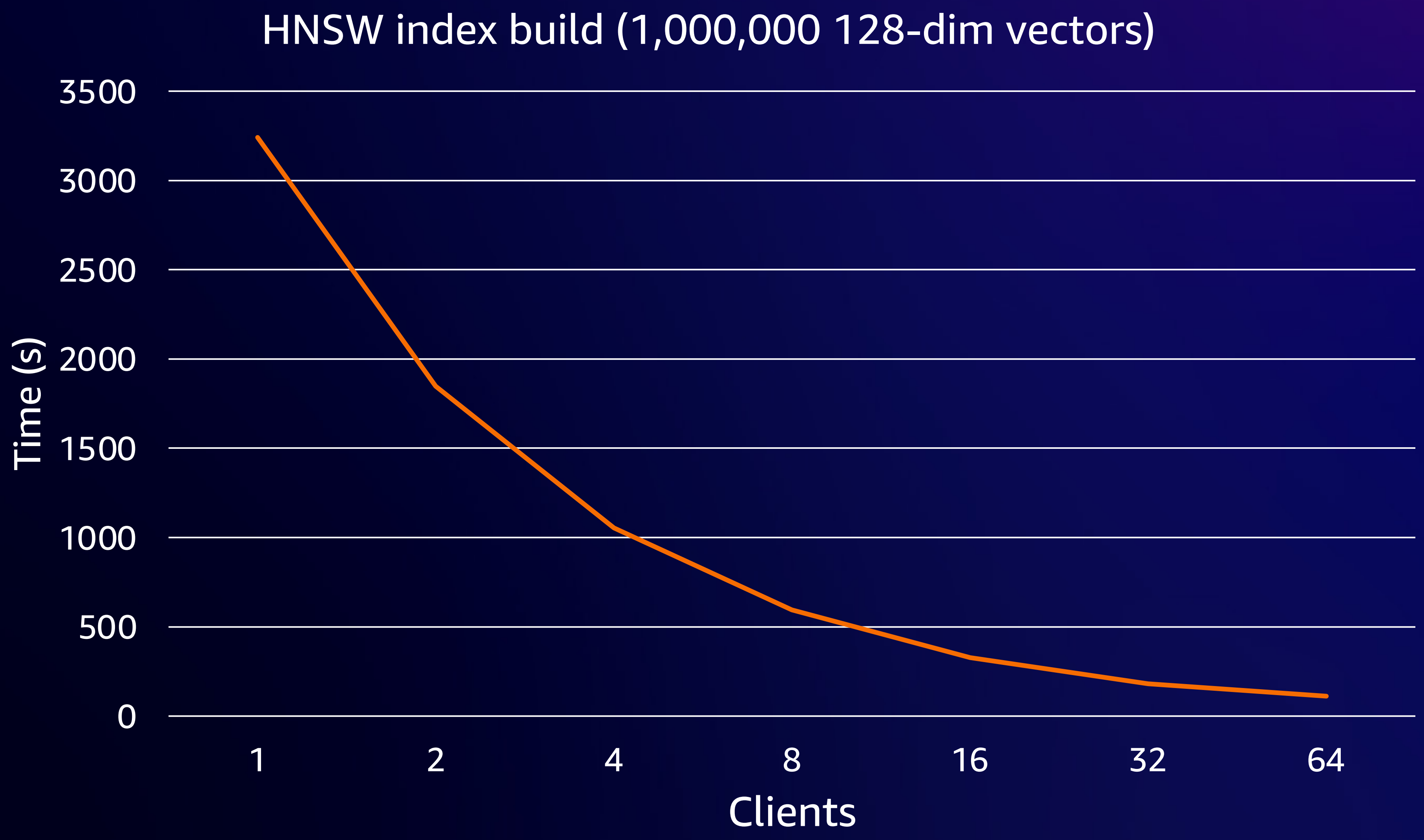
Querying an HNSW index



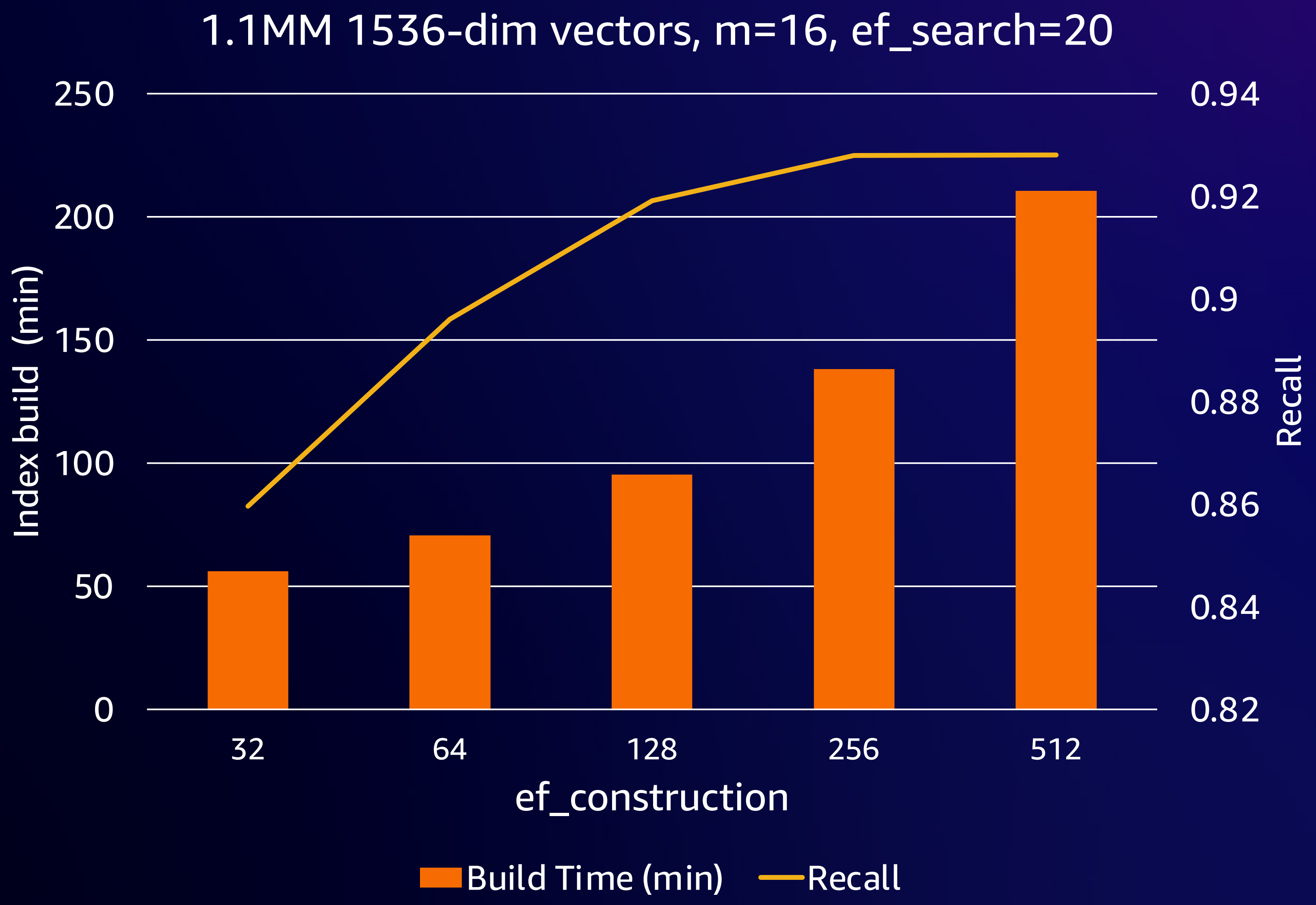
Best practices for building HNSW indexes

- Default values (`M=16,ef_construction=64`) usually work
- (pgvector 0.5.1) Start with empty index and use concurrent writes to accelerate builds
  - `INSERT` or `COPY`

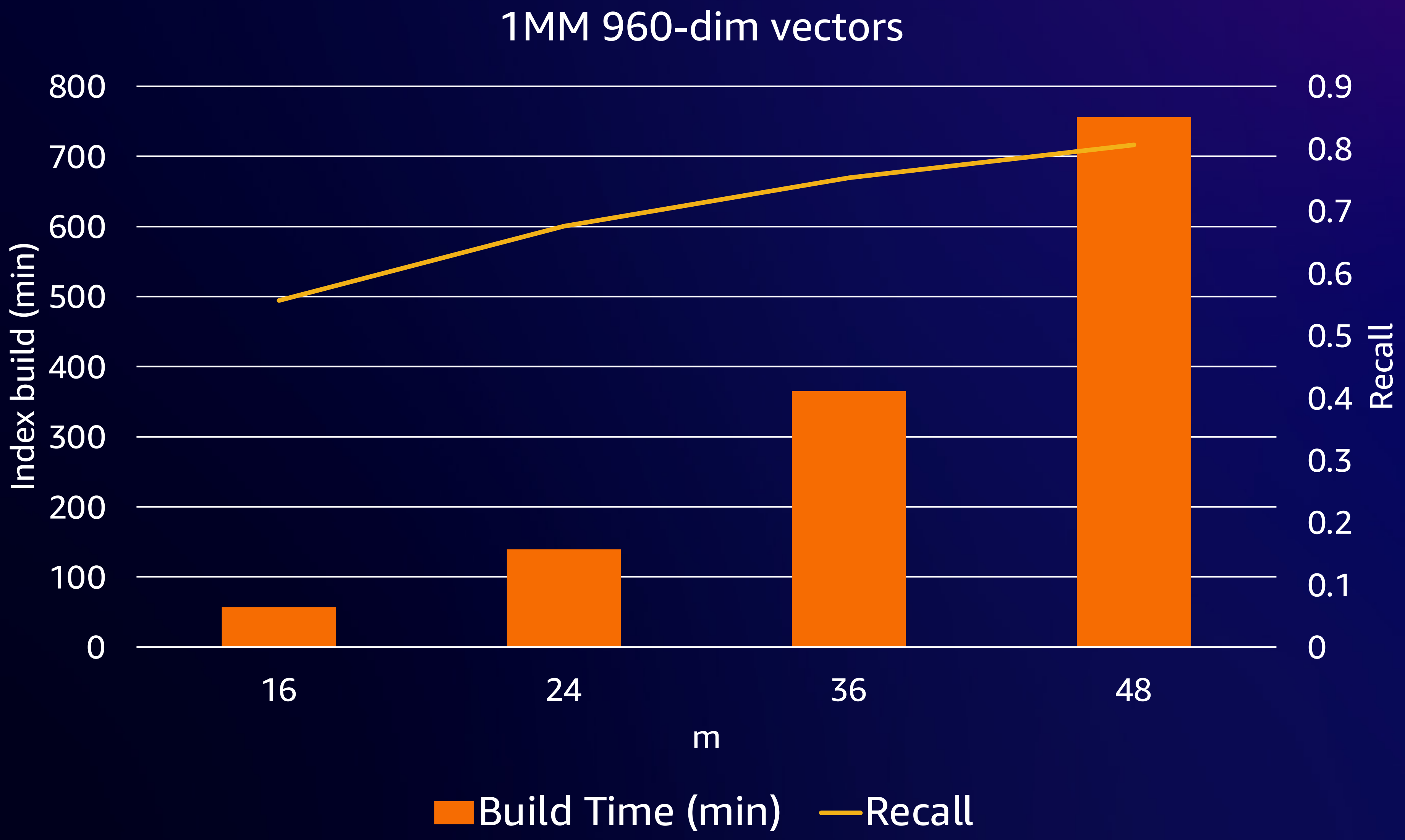
Impact of concurrent inserts on HNSW build time



Choosing m and ef_construction



Choosing m and ef_construction



Performance strategies for HNSW queries

- Index building has biggest impact on performance/recall
  - More time spent building increases likelihood of finding best candidates in a neighborhood

- Increasing `hnsw.ef_search` increases recall, decreases performance

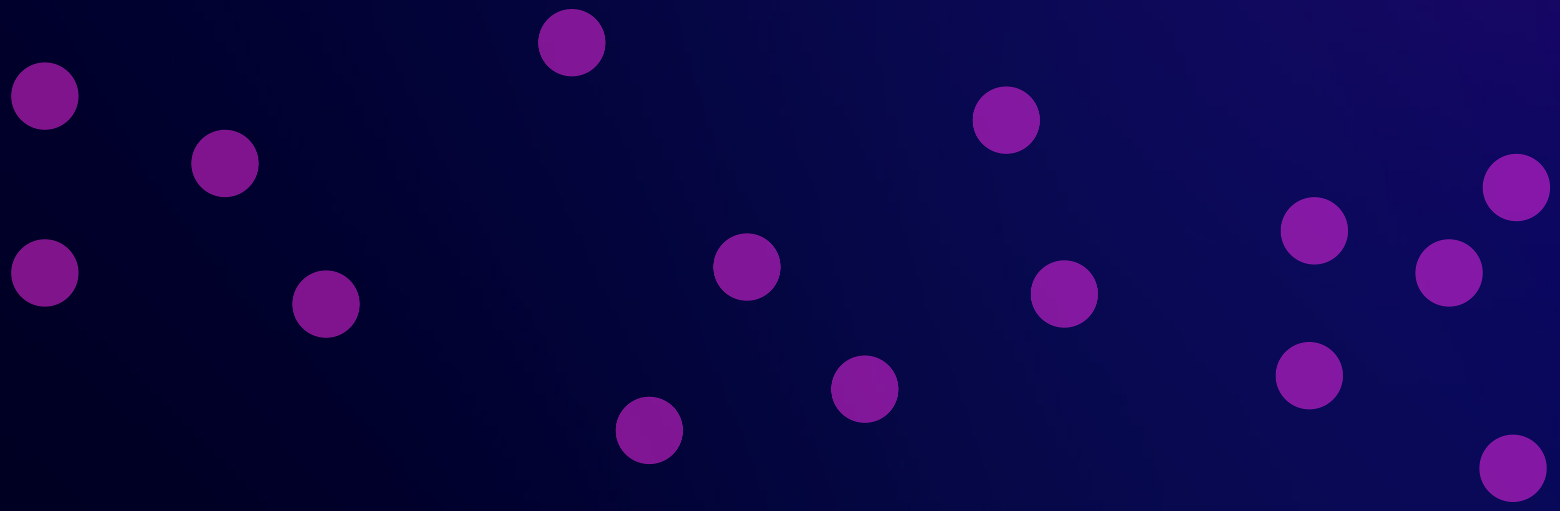
IVFFlat strategies

IVFFlat index building parameters

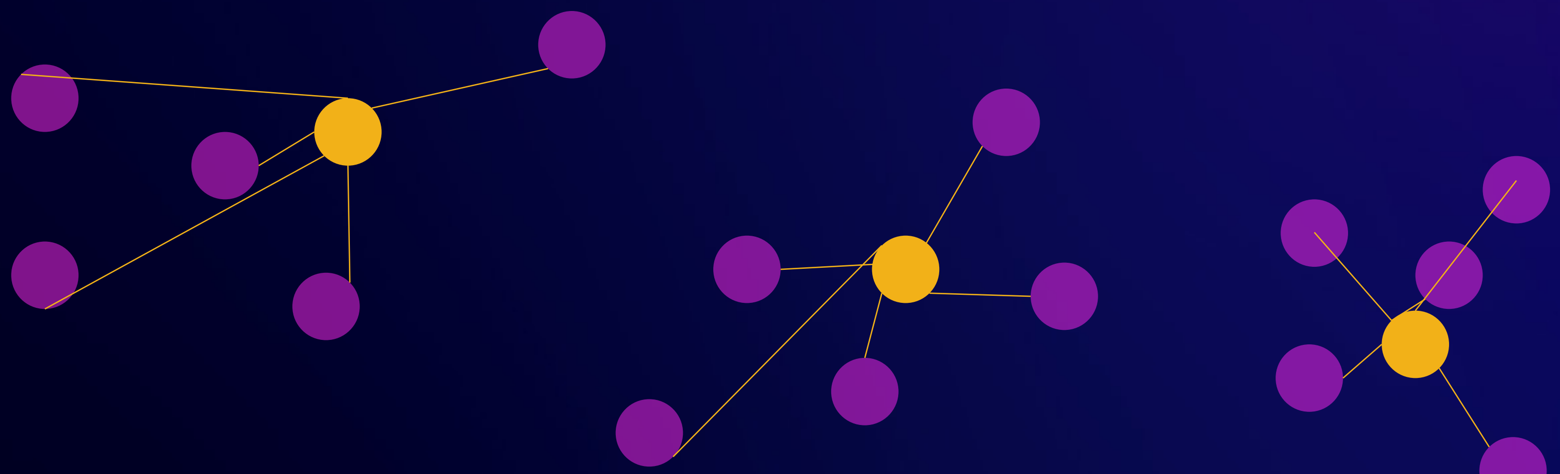
- `lists`
  - Number of “buckets” for organizing vectors
  - Tradeoff between number of vectors in bucket and relevancy

```
CREATE INDEX ON products
USING ivfflat(embedding) WITH (lists=3);
```

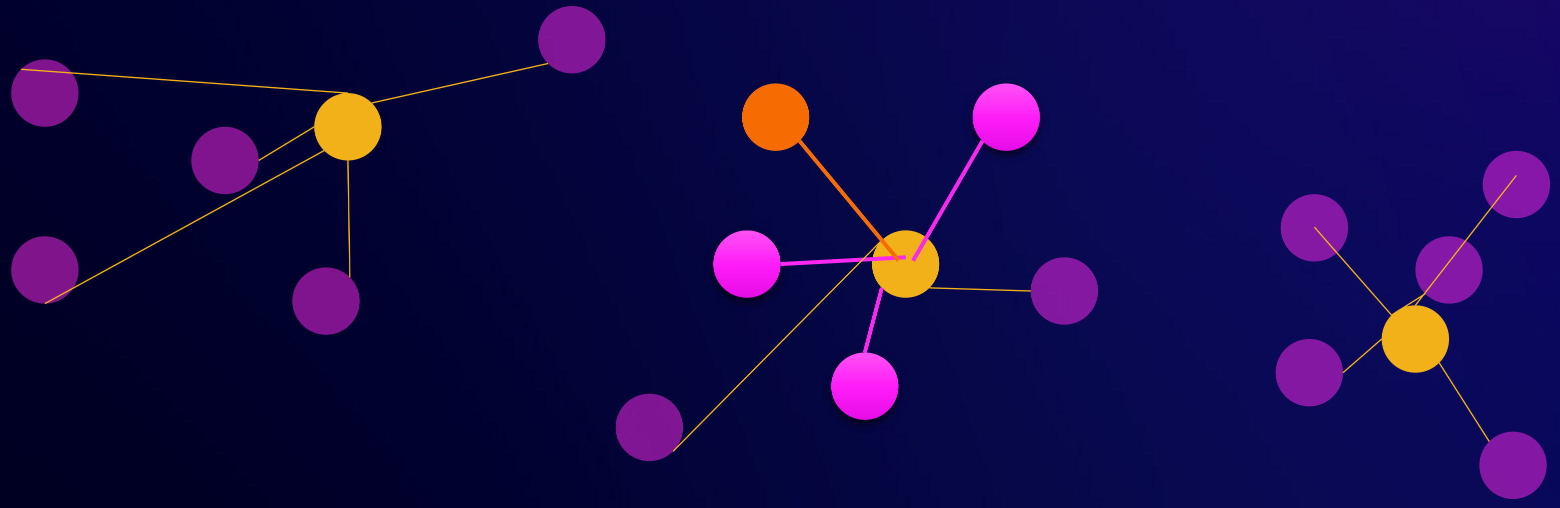
Building an IVFFlat index



Building an IVFFlat index: Assign lists



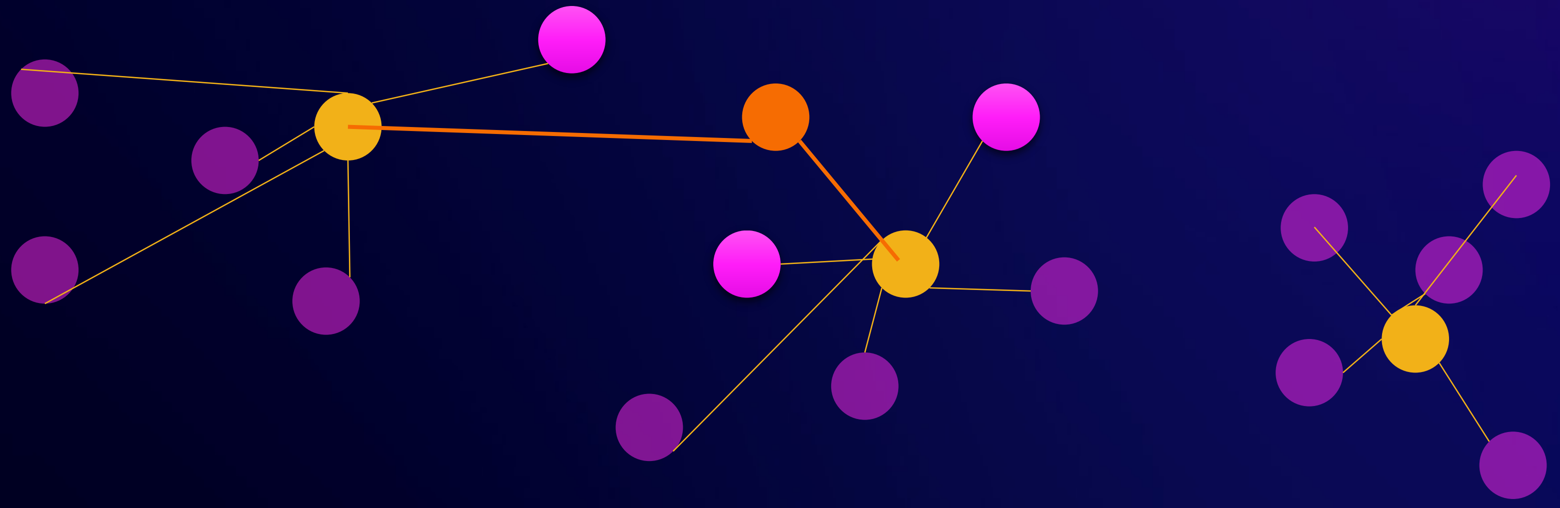
Querying an IVFFlat index



```
SET ivfflat.probes TO 1

SELECT id FROM products ORDER BY $1 <-> embedding LIMIT 3
```

Querying an IVFFlat index



```
SET ivfflat.probes TO 2

SELECT id FROM products ORDER BY $1 <-> embedding LIMIT 3
```

Performance strategies for IVFFlat queries

- Increasing `ivfflat.probes` increases recall, decreases performance
- Lowering `random_page_cost` on a per-query basis can induce index usage
- Set `shared_buffers` to a value that keeps data (table) in memory
- Increase `work_mem` on a per-query basis

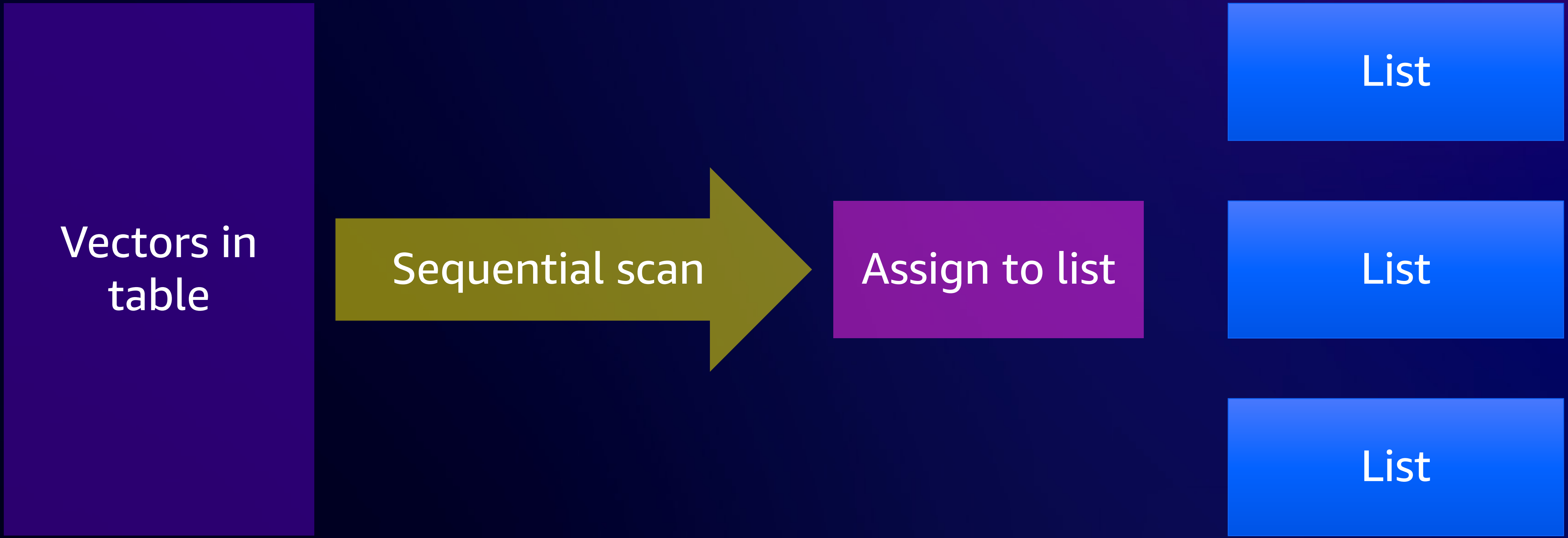
Best practices for building IVFFlat indexes

- Choose value of lists to maximize recall but minimize effort of search
  - < 1MM vectors: # vectors / 1000
  - > 1MM vectors: √(# vectors)

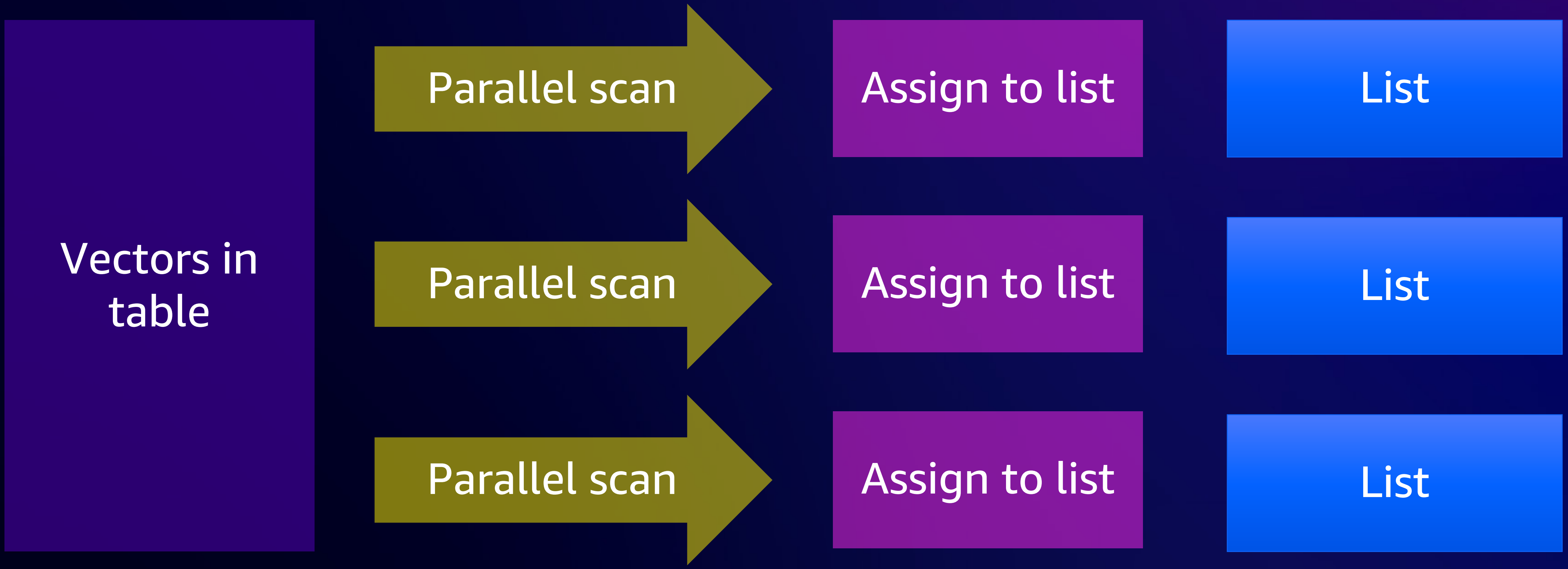
- May be necessary to rebuild when adding/modifying vectors in index

- Use parallelism to accelerate build times

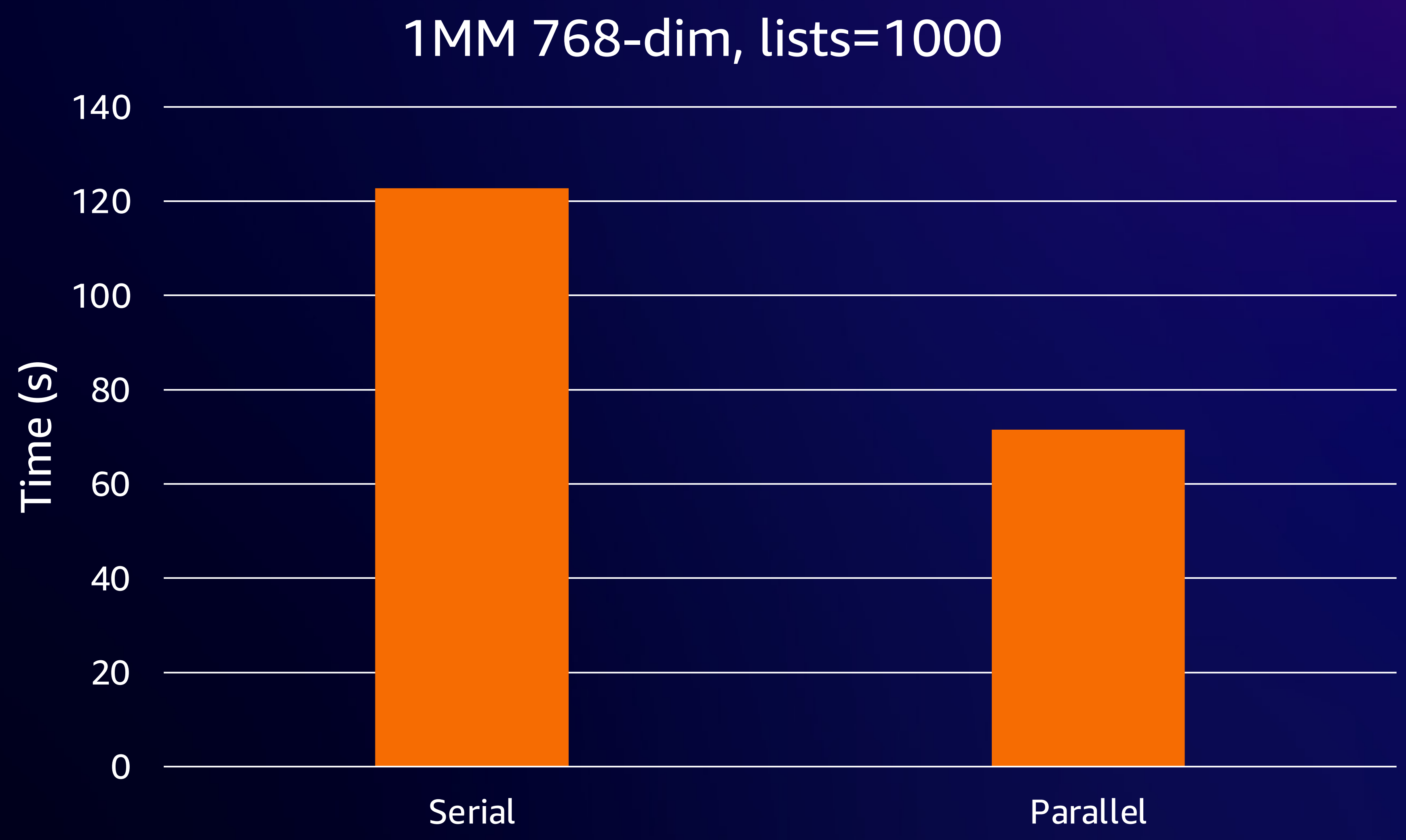
How ~~parallelism~~ works with pgvector IVFFlat



How parallelism works with pgvector IVFFlat



Using parallelism to accelerate IVFFlat builds



pgvector filtering strategies

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/IVFFlat index for a filter?

- Does the filter use a B-Tree (or other index) to reduce the data set?
- How many rows does the filter remove?
- Do I want exact results or approximate results?

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);
```

Filtering with existing embeddings

```sql
SELECT *
FROM (
  (SELECT id,
     embedding <=> (SELECT embedding FROM documents WHERE id = 1 LIMIT 1) AS dist
    FROM documents
    ORDER BY dist LIMIT 5)
  UNION
  (SELECT id,
     embedding <=> (SELECT embedding FROM documents WHERE id = 2 LIMIT 1) AS dist
    FROM documents
    ORDER BY dist LIMIT 5)
) x
WHERE x.id NOT IN (1, 2)
ORDER BY x.dist LIMIT 5;
```

Looking ahead

pgvector roadmap

- Parallel builds for HNSW (committed; targeted for pgvector 0.6.0)
- Enhanced index-based filtering/HQANN (in progress)
- More data types per dimension (float2, uint8) (in progress)
- Product quantization/scalar quantization
- Parallel query

Conclusion

- Like JSON, a **vector is just a data type**.
- Primary design decision: **query performance** and **recall**
- Determine where to invest: **storage**, **compute**, **indexing strategy**
- Plan for today and tomorrow: pgvector is rapidly innovating

Thank you!

Please complete the session survey in the mobile app

Jonathan Katz
jkatz@amazon.com