
AWS
re:Invent
NOV. 27 – DEC. 1, 2023 | LAS VEGAS, NV

DAT407

Best practices for querying vector data for gen AI apps in PostgreSQL

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

Agenda

Overview of generative AI and the role of databases

PostgreSQL as a vector store

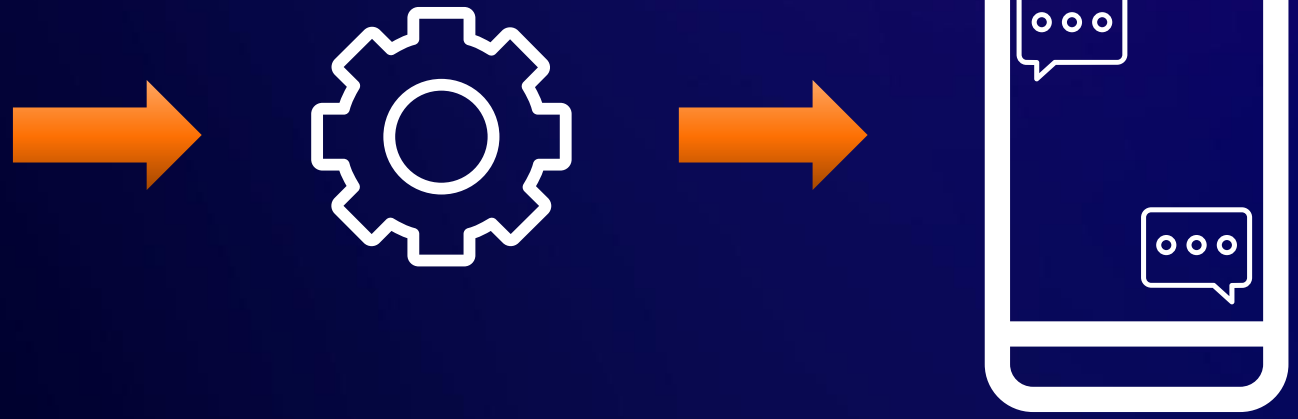
pgvector strategies and best practices

Amazon Aurora features for vector queries

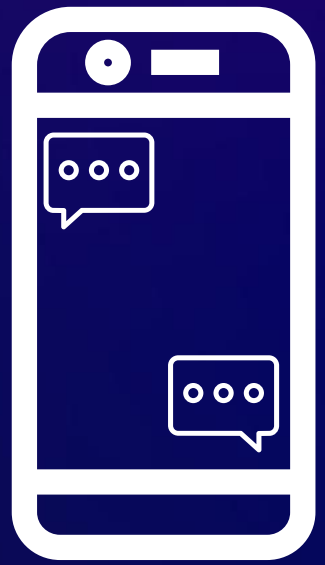
Looking ahead: pgvector roadmap

Blue elephant vase that can hold up to three plants in it, hand painted...
Parrot decorative figure stands 12 inches high, red, and orange, has...
Reddish vase six inches deep perfect for cactuses and desert plants...
Rabbit planter suitable for growing vegetables indoors, green and ...
Decorative ceramic turtle, blue and about eight inches wide, makes...
Bird feeder shaped like a bird, can hold birdseed for 28 days and is...
Sea shell themed vase that's two feet wide and can hold a variety...
Garden variety owl planter, great for keeping your favorite flowers...

Blue elephant vase that can hold up to three plants in it, hand painted...
Parrot decorative figure stands 12 inches high, red, and orange, has...
Reddish vase six inches deep perfect for cactuses and desert plants...
Rabbit planter suitable for growing vegetables indoors, green and ...
Decorative ceramic turtle, blue and about eight inches wide, makes...
Bird feeder shaped like a bird, can hold birdseed for 28 days and is...
Sea shell themed vase that's two feet wide and can hold a variety...
Garden variety owl planter, great for keeping your favorite flowers...



Blue elephant vase that can hold up to three plants in it, hand painted...
Parrot decorative figure stands 12 inches high, red, and orange, has...
Reddish vase six inches deep perfect for cactuses and desert plants...
Rabbit planter suitable for growing vegetables indoors, green and ...
Decorative ceramic turtle, blue and about eight inches wide, makes...
Bird feeder shaped like a bird, can hold birdseed for 28 days and is...
Sea shell themed vase that's two feet wide and can hold a variety...
Garden variety owl planter, great for keeping your favorite flowers...



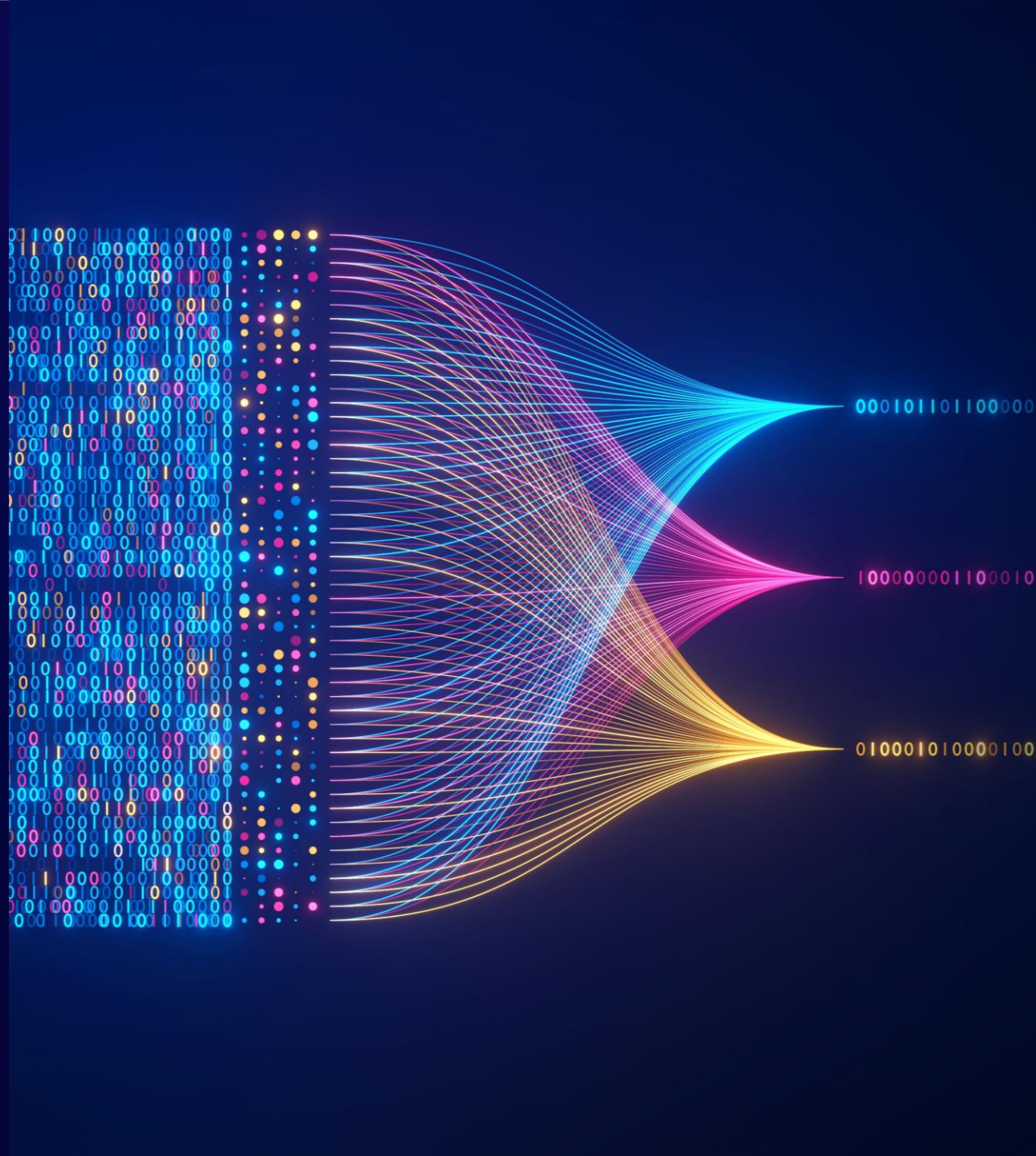
Generative AI is powered by foundation models

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



NOW GENERALLY AVAILABLE

Amazon Bedrock

The easiest way to build and scale generative AI applications with foundation models (FMs)

Accelerate development of generative AI applications using FMs through an API, without managing infrastructure

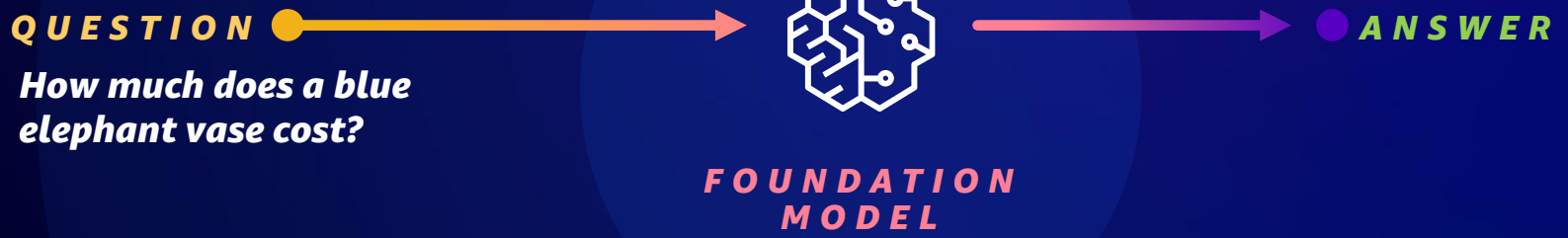
Choose FMs from Amazon, AI21 Labs, Anthropic, Cohere, Meta, and Stability AI to find the right FM for your use case

Privately customize FMs using your organization's data

aws

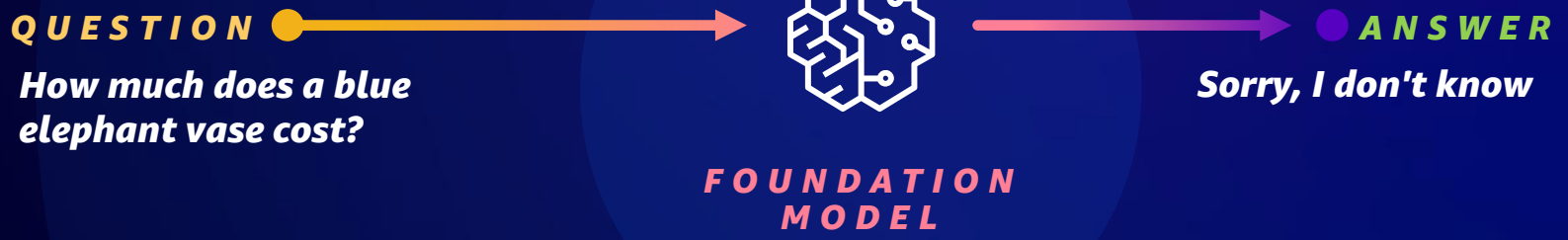
Retrieval Augmented Generation (RAG)

Configure FM to interact with your company data



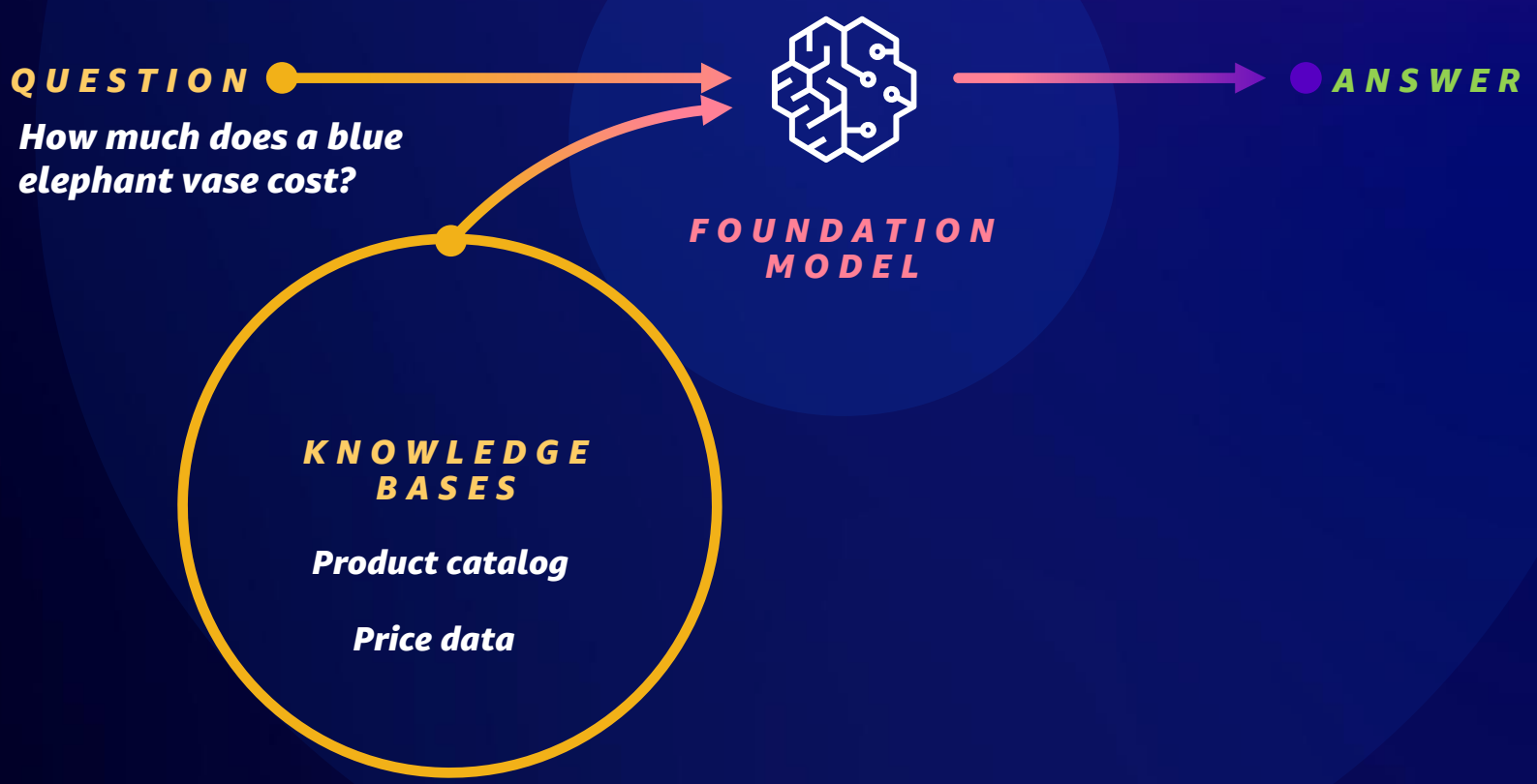
Retrieval Augmented Generation (RAG)

Configure FM to interact with your company data



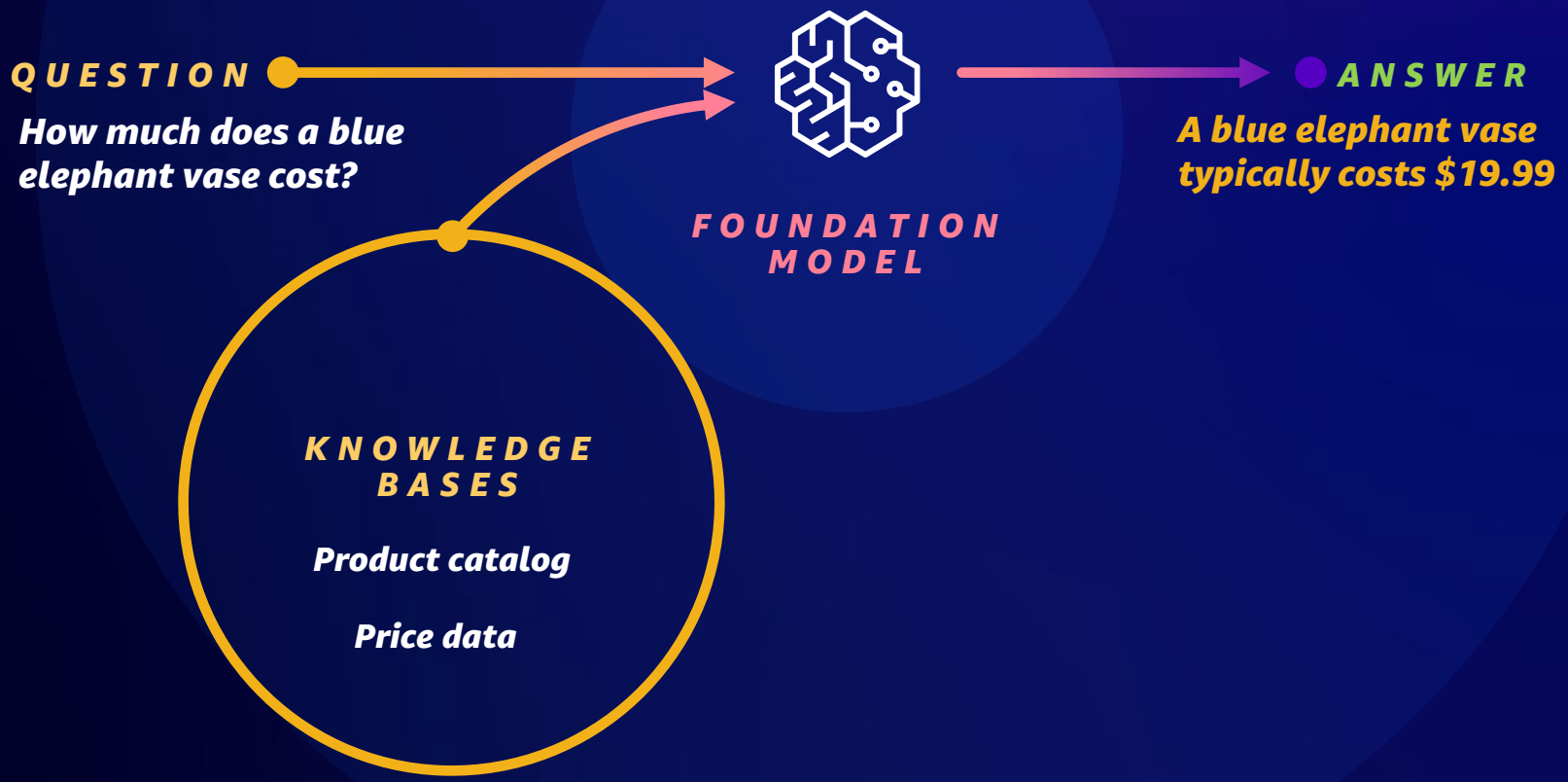
Retrieval Augmented Generation (RAG)

Configure FM to interact with your company data



Retrieval Augmented Generation (RAG)

Configure FM to interact with your company data



What are vector embeddings?

Unstructured data has to be vectorized into vectors to be used in generative AI applications

How vector embeddings are used

How vector embeddings are used

1

PDF document

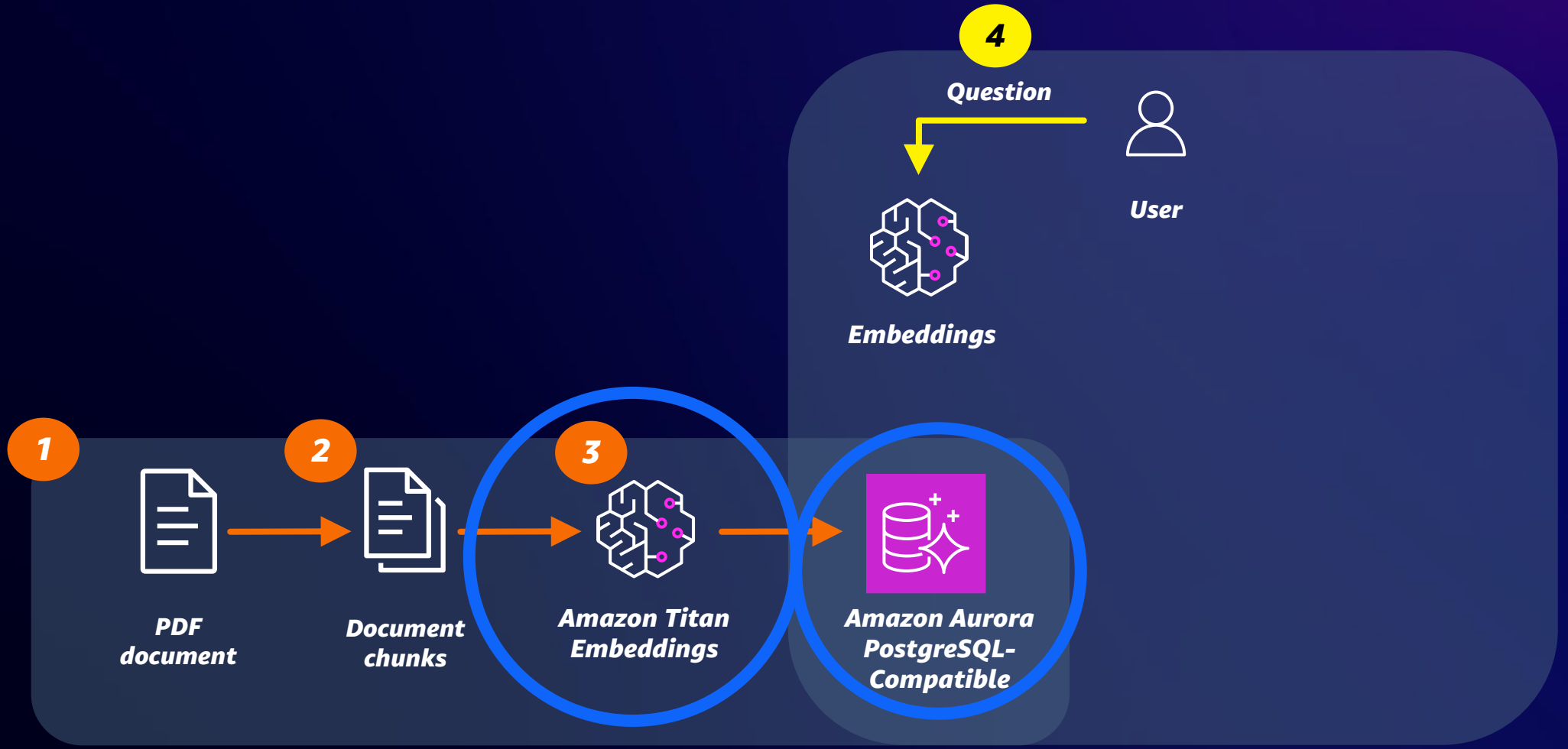
How vector embeddings are used

How vector embeddings are used

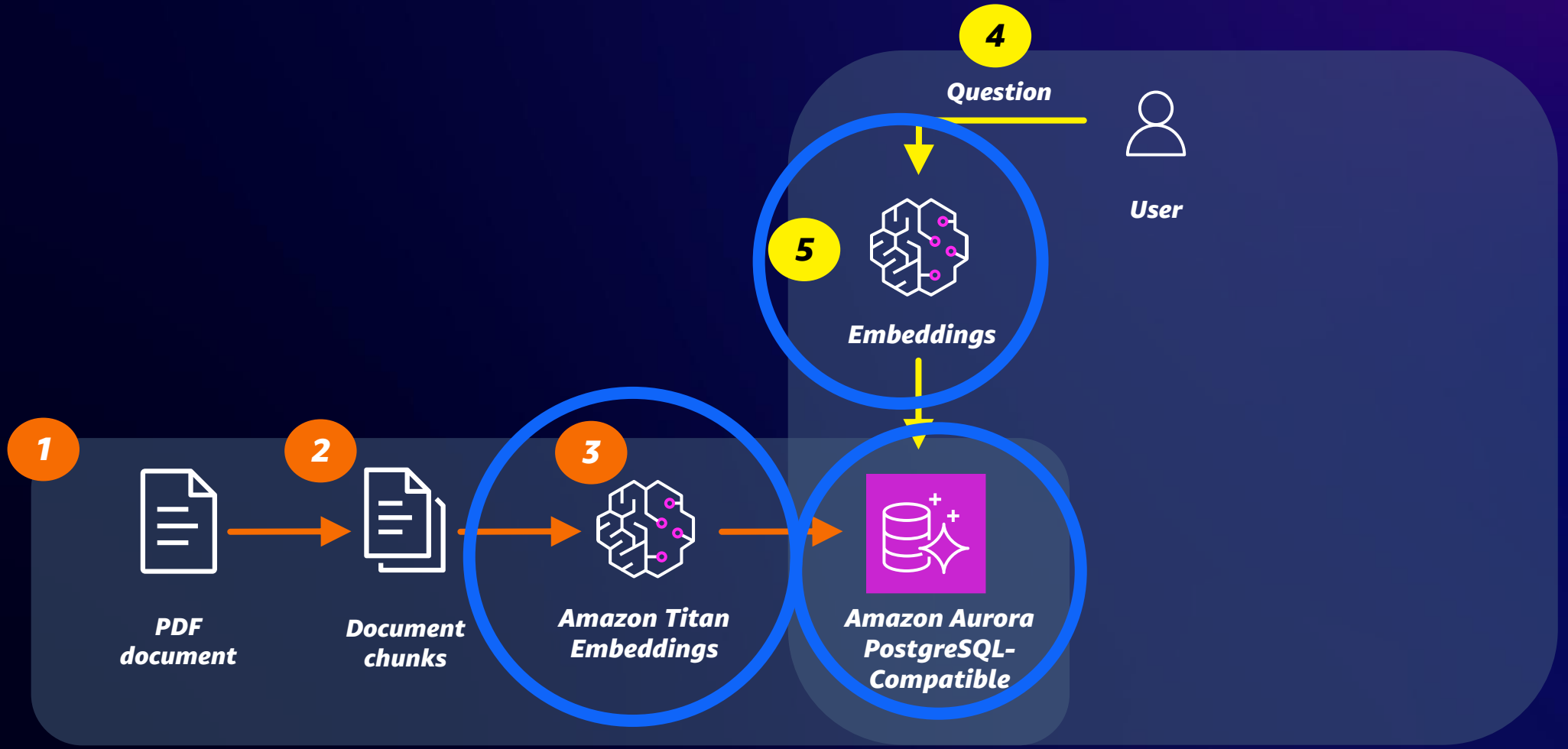
How vector embeddings are used

How vector embeddings are used

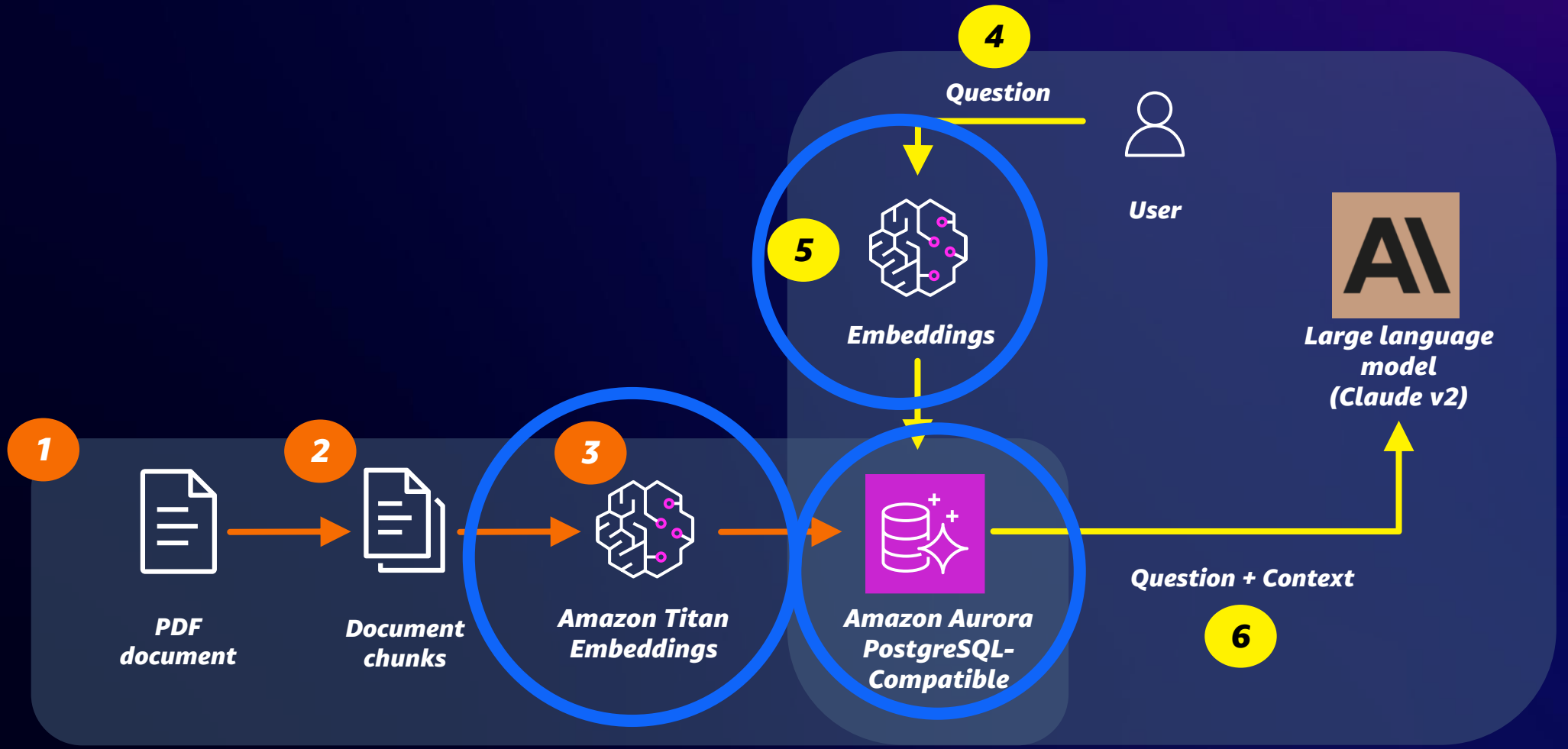
How vector embeddings are used



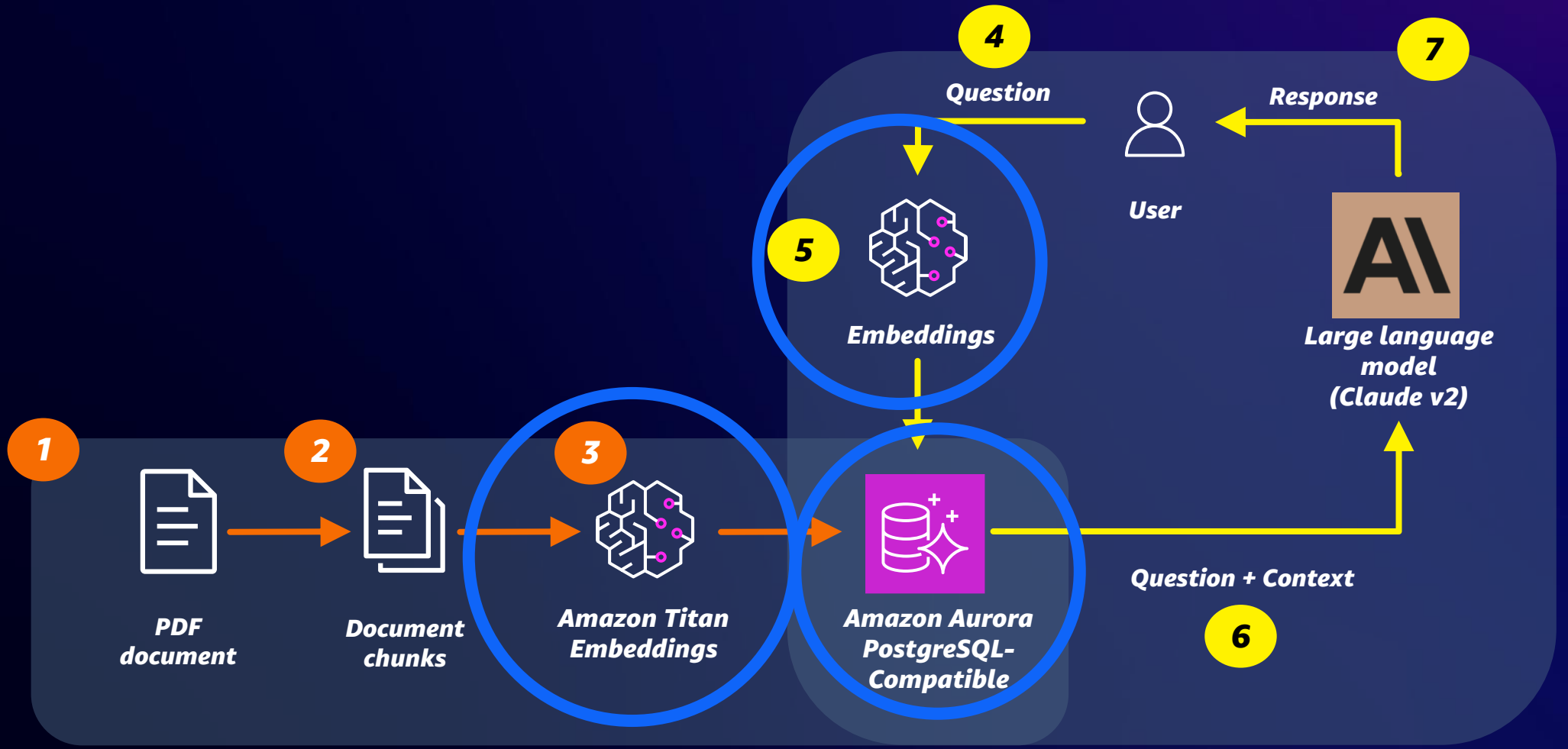
How vector embeddings are used



How vector embeddings are used



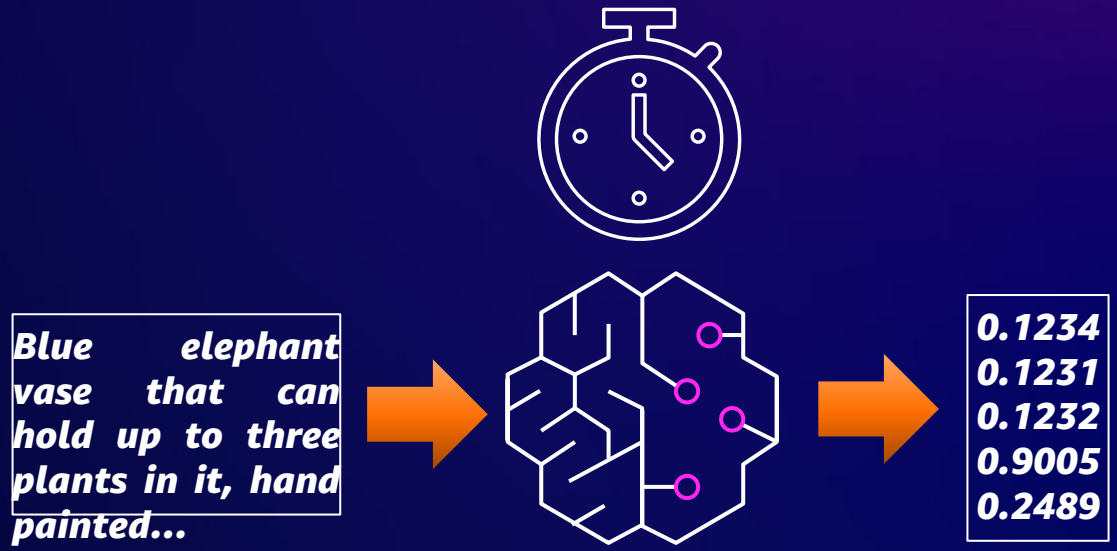
How vector embeddings are used



Challenges with vectors

Challenges with vectors

- Time to generate embeddings



Challenges with vectors

- Time to generate embeddings
- Embedding size

1536 dimensions

4-byte floats

6152B => 6KiB

Challenges with vectors

- Time to generate embeddings
- Embedding size

1536 dimensions

4-byte floats

6152B => 6KiB

1,000,000 => 5.7GB

Challenges with vectors

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

1536 dimensions

4-byte floats

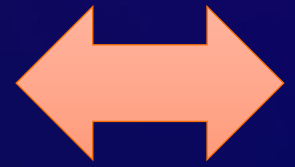
6152B => 6KiB

1,000,000 => 5.7GB

Challenges with vectors

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

0.12310	0.20559
0.24234	0.70543
0.59405	0.23432
0.23430	0.24234
0.23432	0.23430
0.20551	0.12310
0.70543	0.20551
0.20559	0.59405



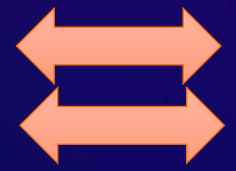
Challenges with vectors

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

0.12310	0.20559
0.24234	0.70543
0.59405	0.23432
0.23430	0.24234
0.23432	0.23430
0.20551	0.12310
0.70543	0.20551
0.20559	0.59405

Challenges with vectors

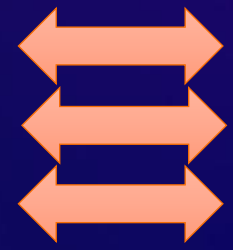
- Time to generate embeddings
- Embedding size
- ~~Compression~~
- Query time



0.12310	0.20559
0.24234	0.70543
0.59405	0.23432
0.23430	0.24234
0.23432	0.23430
0.20551	0.12310
0.70543	0.20551
0.20559	0.59405

Challenges with vectors

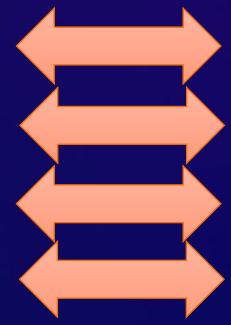
- Time to generate embeddings
- Embedding size
- ~~Compression~~
- Query time



0.12310	0.20559
0.24234	0.70543
0.59405	0.23432
0.23430	0.24234
0.23432	0.23430
0.20551	0.12310
0.70543	0.20551
0.20559	0.59405

Challenges with vectors

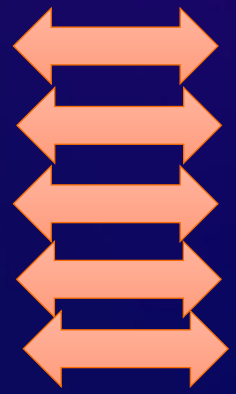
- Time to generate embeddings
- Embedding size
- ~~Compression~~
- Query time



0.12310	0.20559
0.24234	0.70543
0.59405	0.23432
0.23430	0.24234
0.23432	0.23430
0.20551	0.12310
0.70543	0.20551
0.20559	0.59405

Challenges with vectors

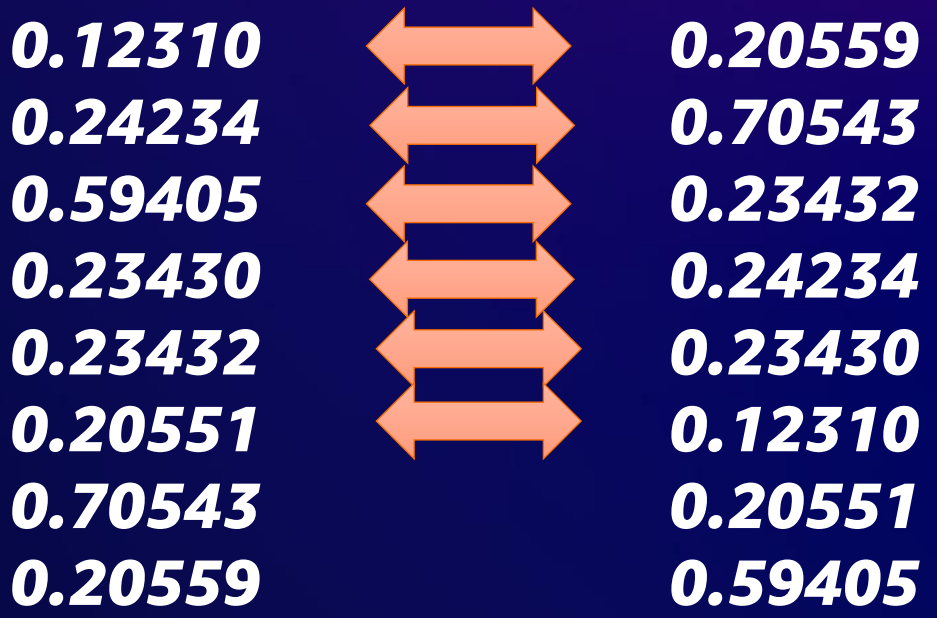
- Time to generate embeddings
- Embedding size
- ~~Compression~~
- Query time



0.12310	0.20559
0.24234	0.70543
0.59405	0.23432
0.23430	0.24234
0.23432	0.23430
0.20551	0.12310
0.70543	0.20551
0.20559	0.59405

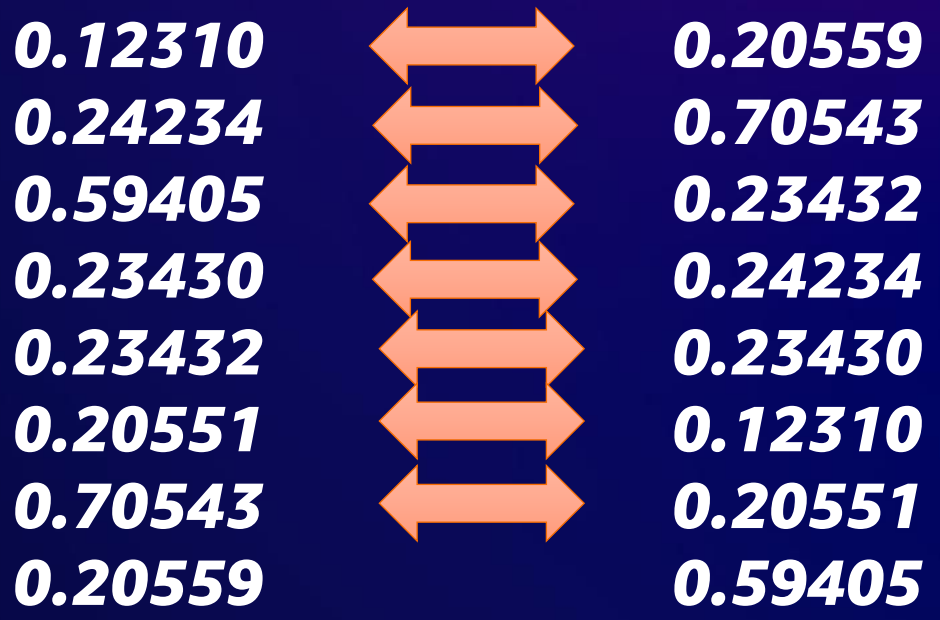
Challenges with vectors

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



Challenges with vectors

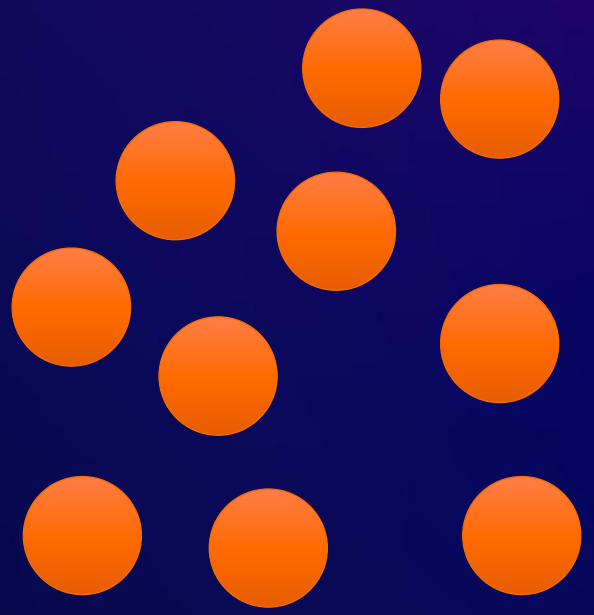
- Time to generate embeddings
- Embedding size
- ~~Compression~~
- Query time



Challenges with vectors

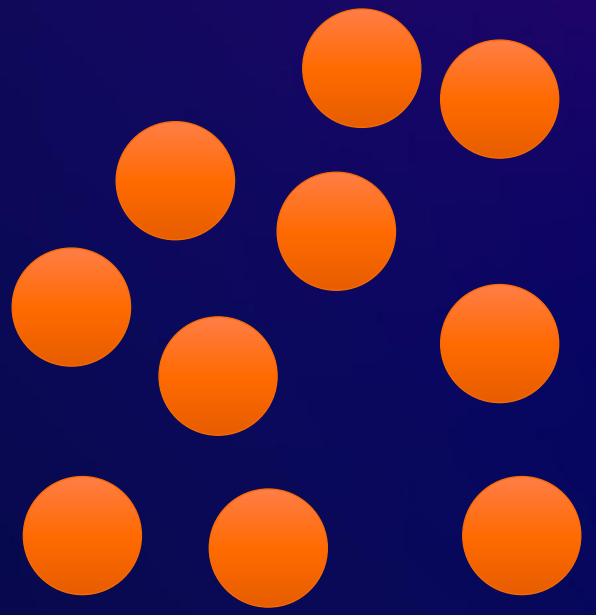
- Time to generate embeddings
- Embedding size
- ~~Compression~~
- Query time

Approximate nearest neighbor (ANN)



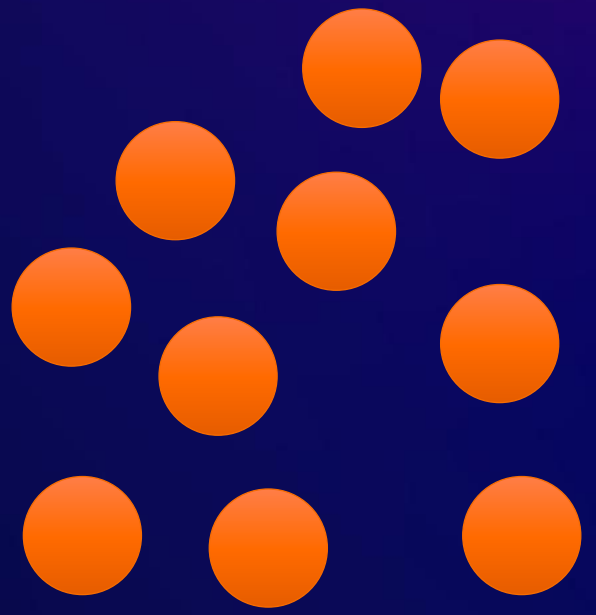
Approximate nearest neighbor (ANN)

- Find similar vectors without searching all of them



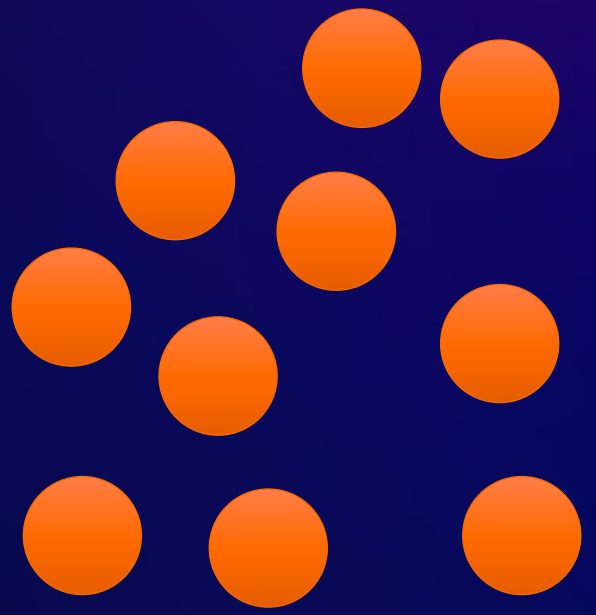
Approximate nearest neighbor (ANN)

- Find similar vectors without searching all of them
- Faster than exact nearest neighbor



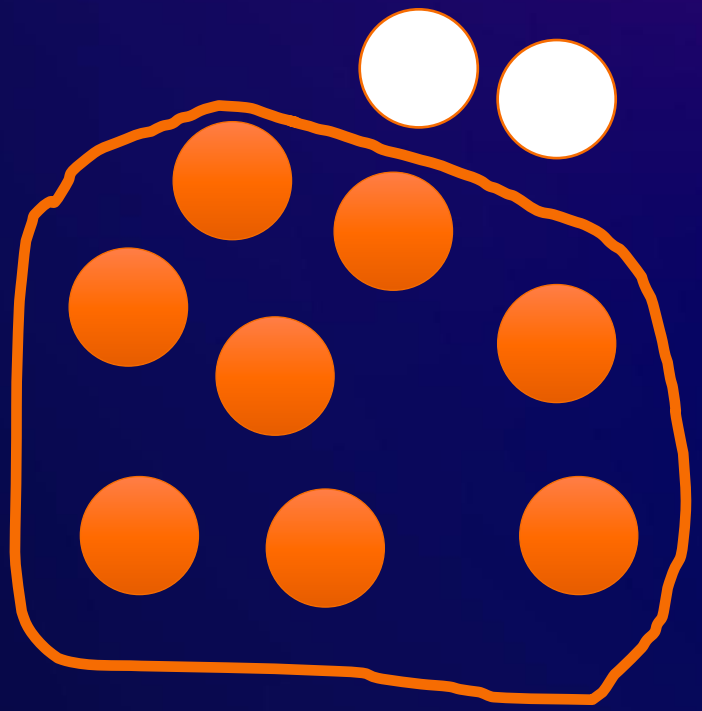
Approximate nearest neighbor (ANN)

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



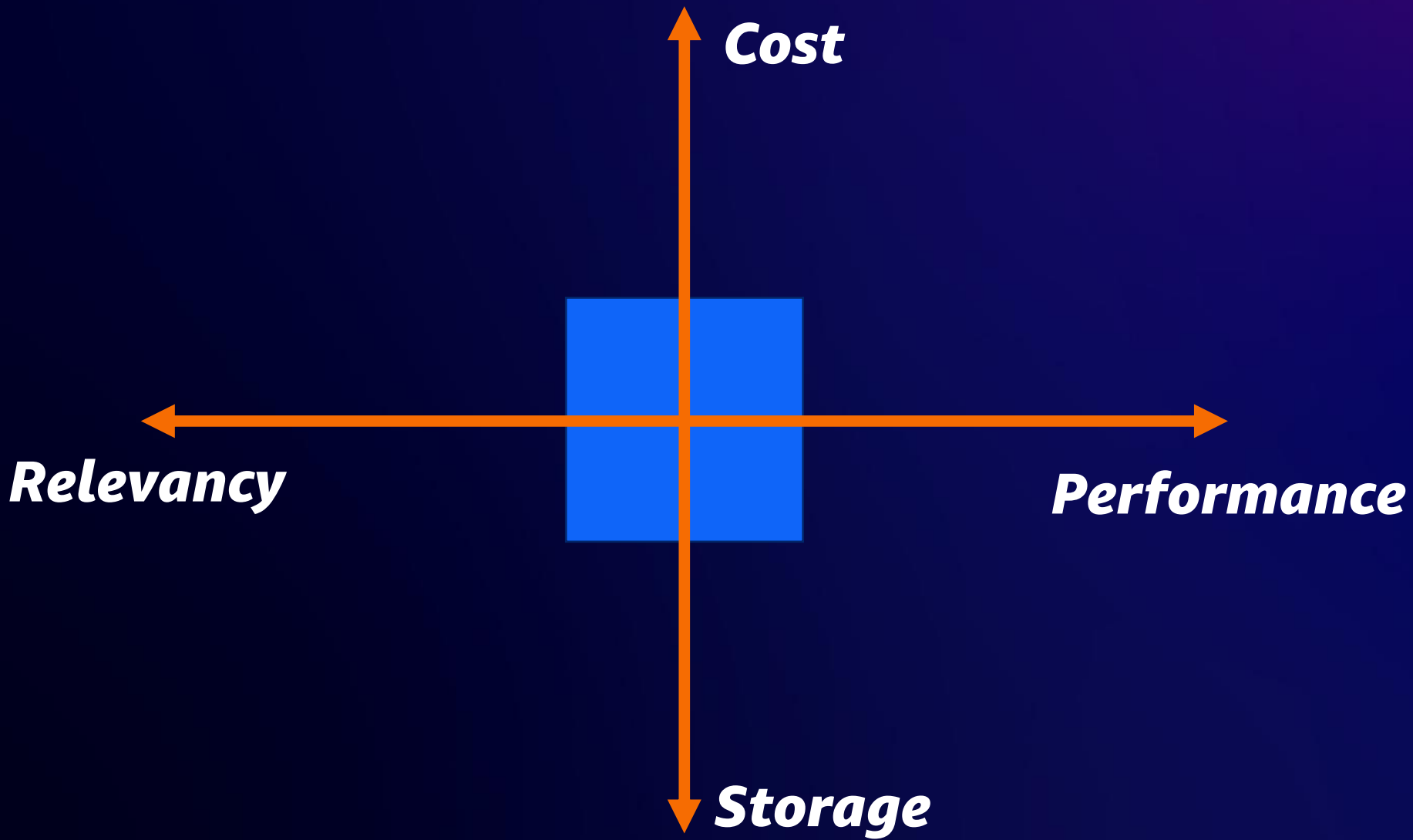
Approximate nearest neighbor (ANN)

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

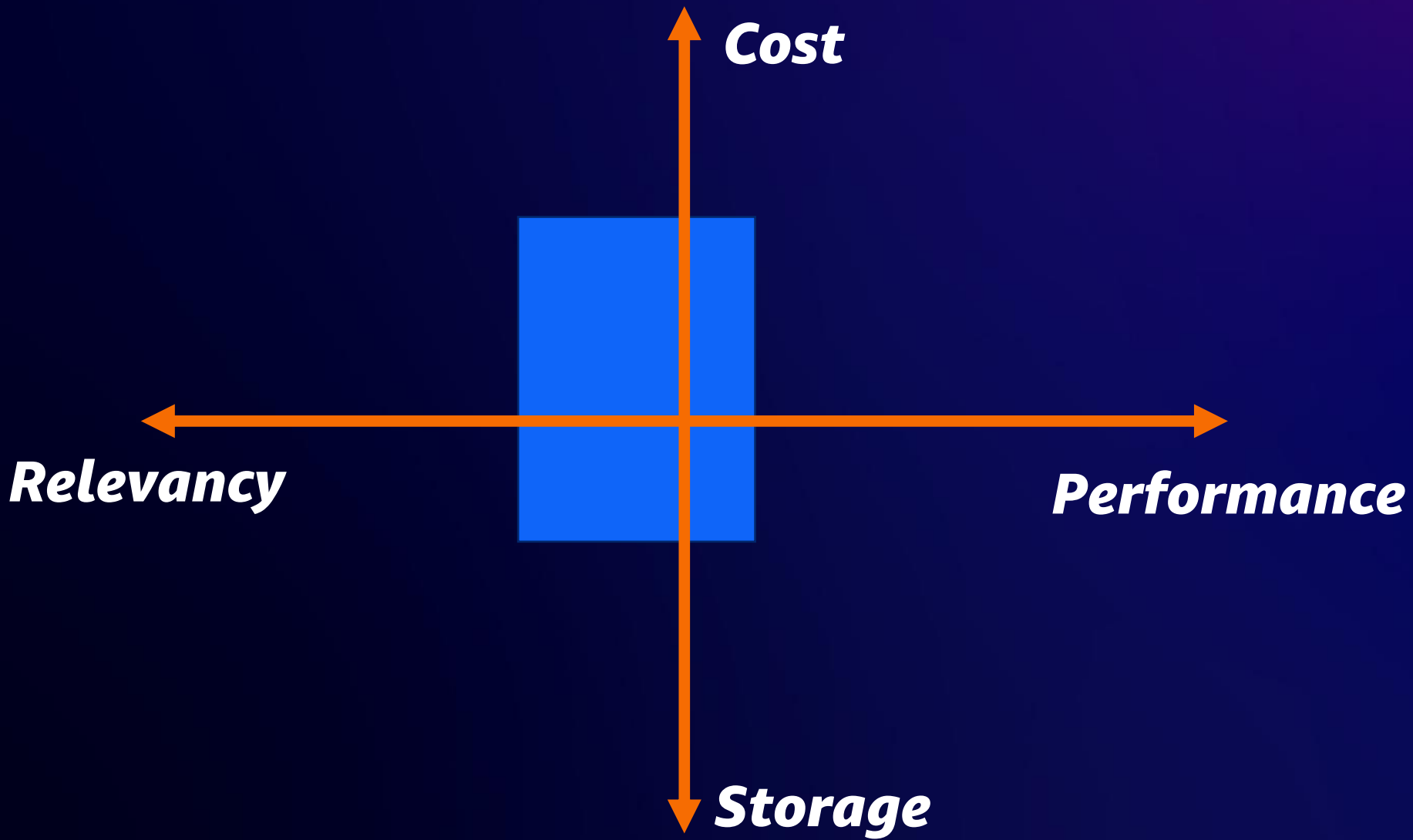


Recall: 80%

Considerations for vector storage



Considerations for vector storage



Questions for choosing a vector storage system

Questions for choosing a vector storage system

- Where does vector storage fit into my workflow?

Questions for choosing a vector storage system

- Where does vector storage fit into my workflow?
- How much data am I storing?

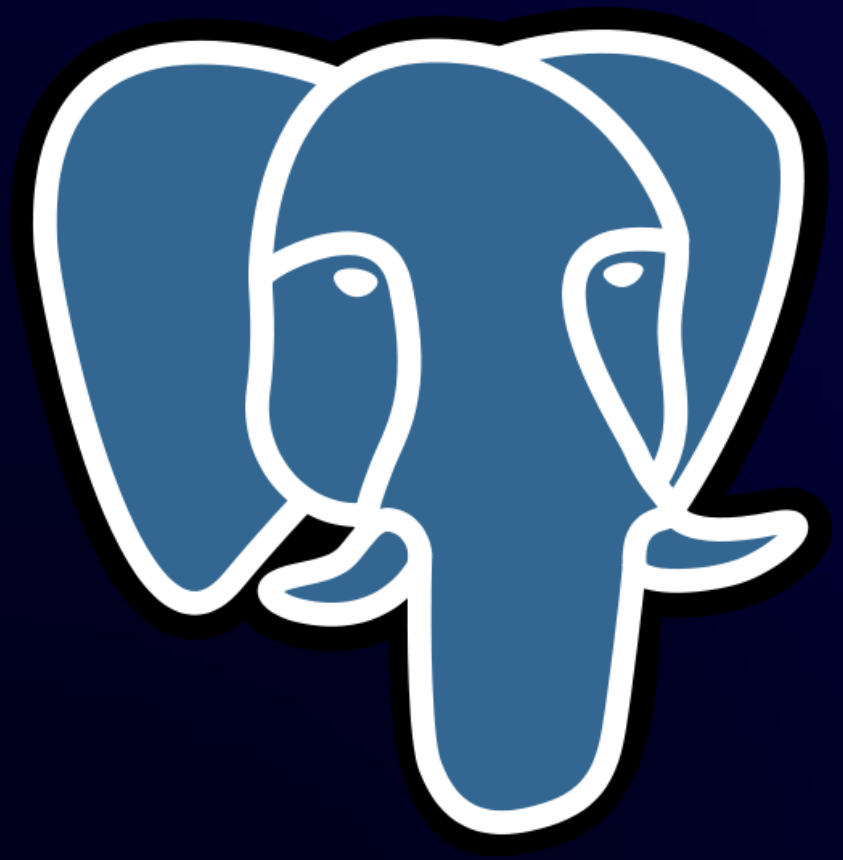
Questions for choosing a vector storage system

- Where does vector storage fit into my workflow?
- How much data am I storing?
- What matters to me: storage, performance, relevancy, cost?

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



Open source

- Active development for more than 35 years
- Controlled by a community, not a single company

Performance and scale

- Robust data type implementations
- Extensive indexing support
- Parallel processing for complex queries
- Native partitioning for large tables

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

What is pgvector?

An open source extension that:

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

github.com/pgvector/pgvector

What is pgvector?

An open source extension that:

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

`vector` data type

github.com/pgvector/pgvector

What is pgvector?

An open source extension that:

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

`vector` data type

Supports IVFFlat/HNSW indexing

github.com/pgvector/pgvector

What is pgvector?

An open source extension that:

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

`vector` data type

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

Supports IVFFlat/HNSW indexing

github.com/pgvector/pgvector

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

github.com/pgvector/pgvector

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

Indexing methods: IVFFlat and HNSW

- IVFFlat
- HNSW

Indexing methods: IVFFlat and HNSW

- IVFFlat
  - K-means based
- HNSW
  - Graph based

Indexing methods: IVFFlat and HNSW

- IVFFlat
  - K-means based
  - Organize vectors into lists
- HNSW
  - Graph based
  - Organize vectors into “neighborhoods”

Indexing methods: IVFFlat and HNSW

- IVFFlat
  - K-means based
  - Organize vectors into lists
  - Requires prepopulated data
- HNSW
  - Graph based
  - Organize vectors into “neighborhoods”
  - Iterative insertions

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?

Which search method do I choose?

- Exact nearest neighbors: No index

Which search method do I choose?

- Exact nearest neighbors: No index
- Fast indexing: IVFFlat

Which search method do I choose?

- Exact nearest neighbors: No index
- Fast indexing: IVFFlat
- Easy to manage: HNSW

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

Understanding TOAST in PostgreSQL

- TOAST (The Oversized-Attribute Storage Technique) is a mechanism for storing data larger than 8KB

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

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

PostgreSQL column storage types

- PLAIN*: Data stored inline with table*

PostgreSQL column storage types

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

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*

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
```

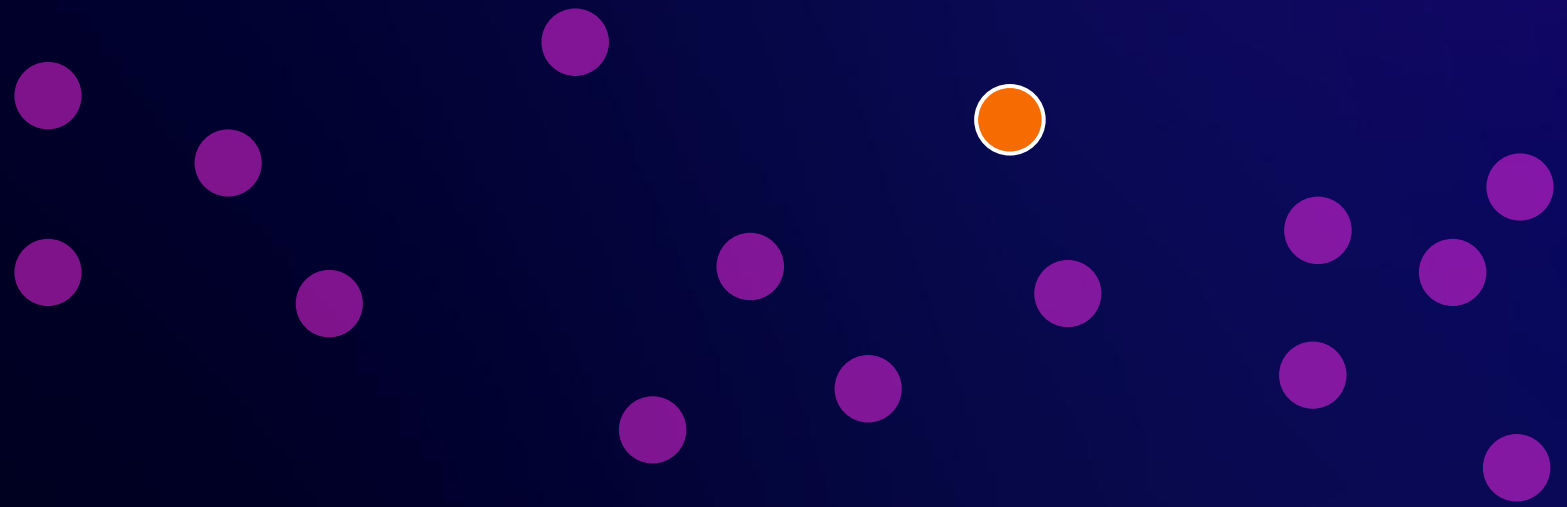
HNSW strategies

HNSW index building parameters

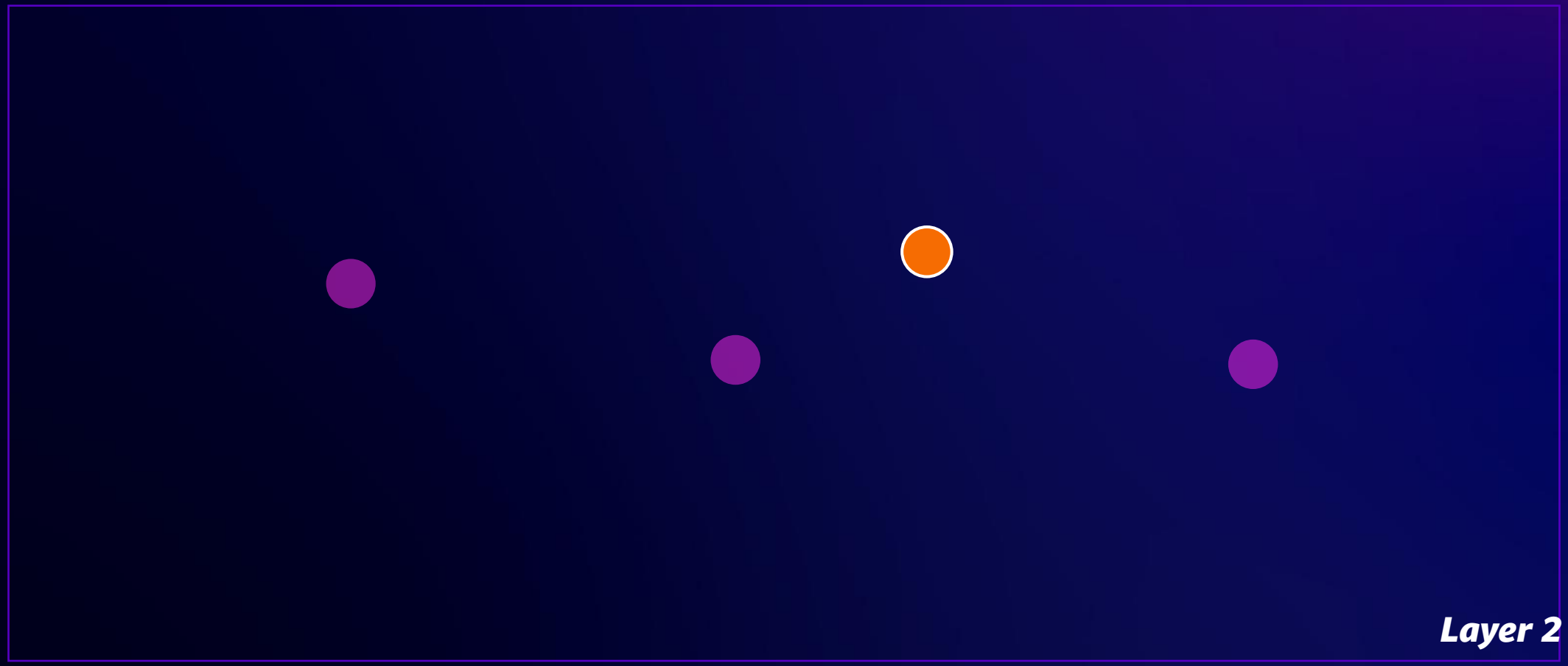
- `m`
  - ***Maximum number of bidirectional links between indexed vectors***
  - ***Default: 16***

- `ef_construction`
  - ***Number of vectors to maintain in “nearest neighbor” list***
  - ***Default: 64***

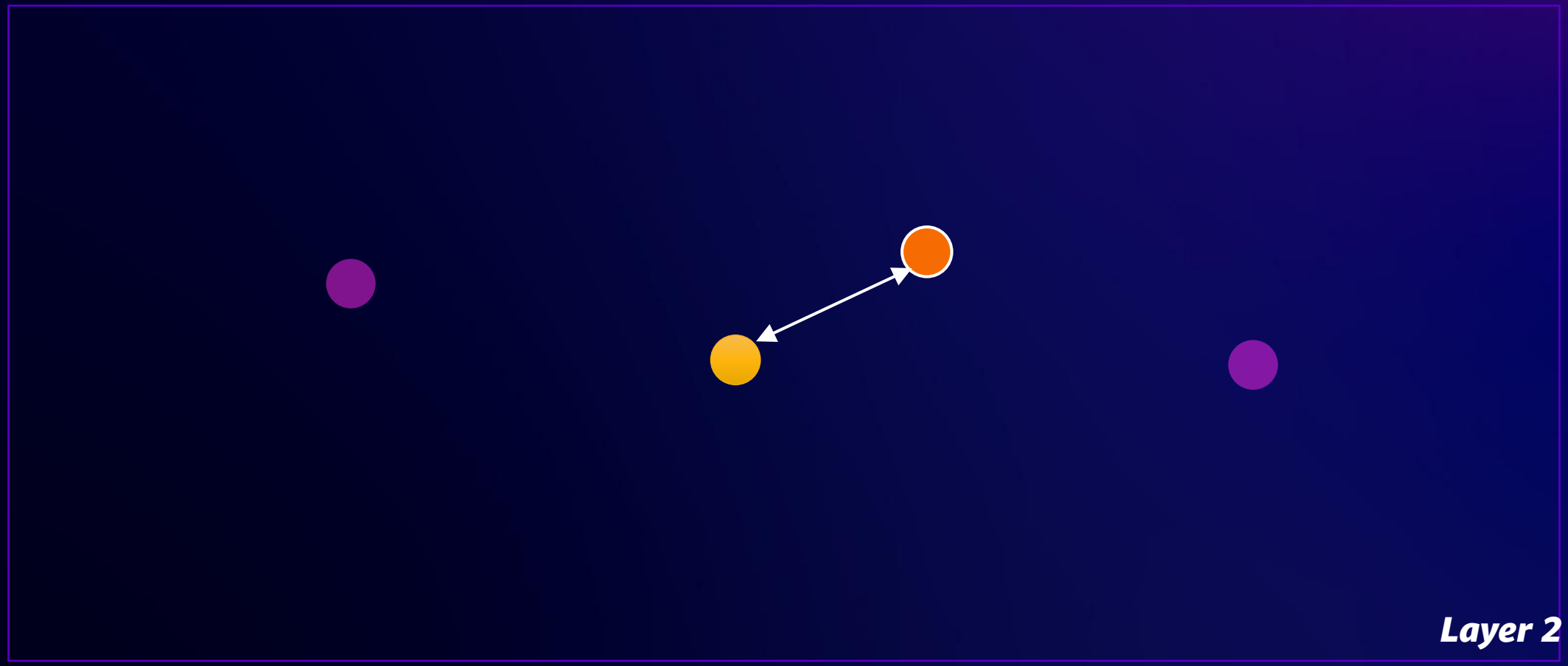
Building an HNSW index



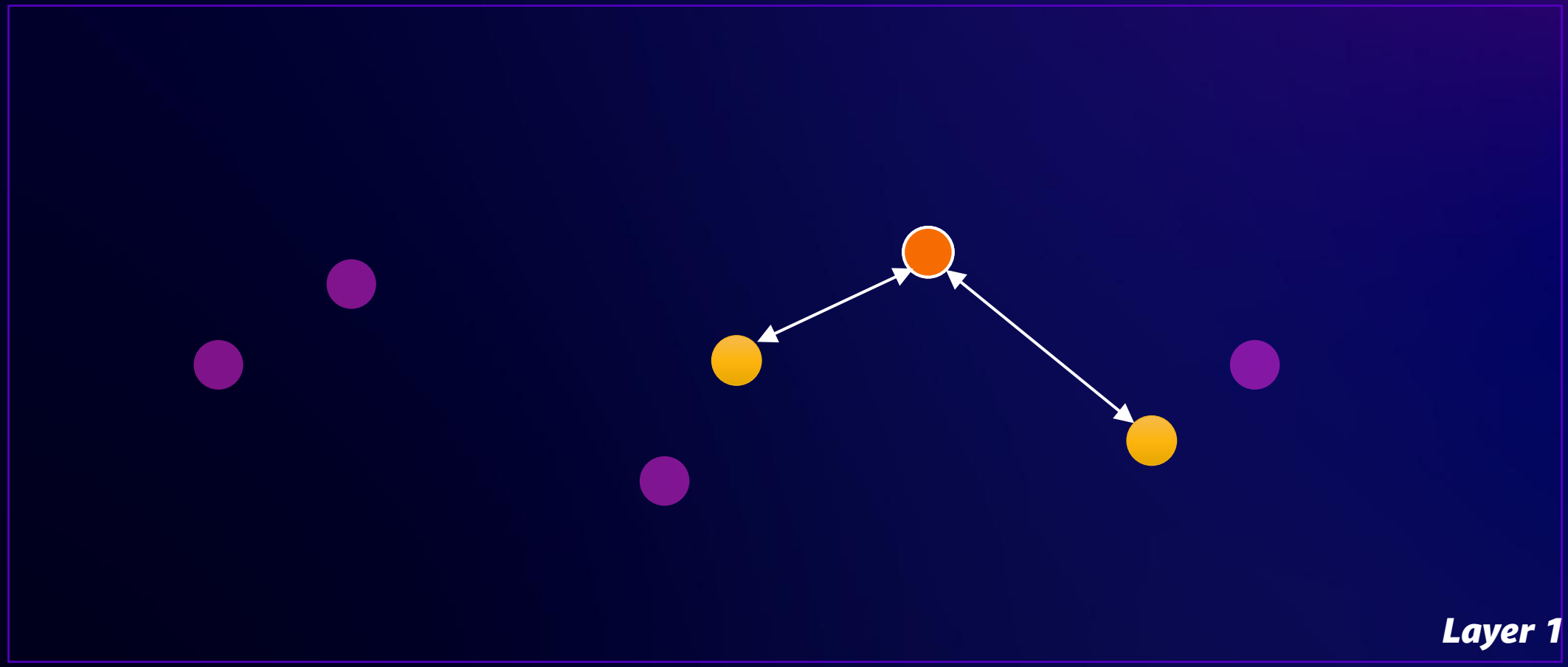
Building an HNSW index



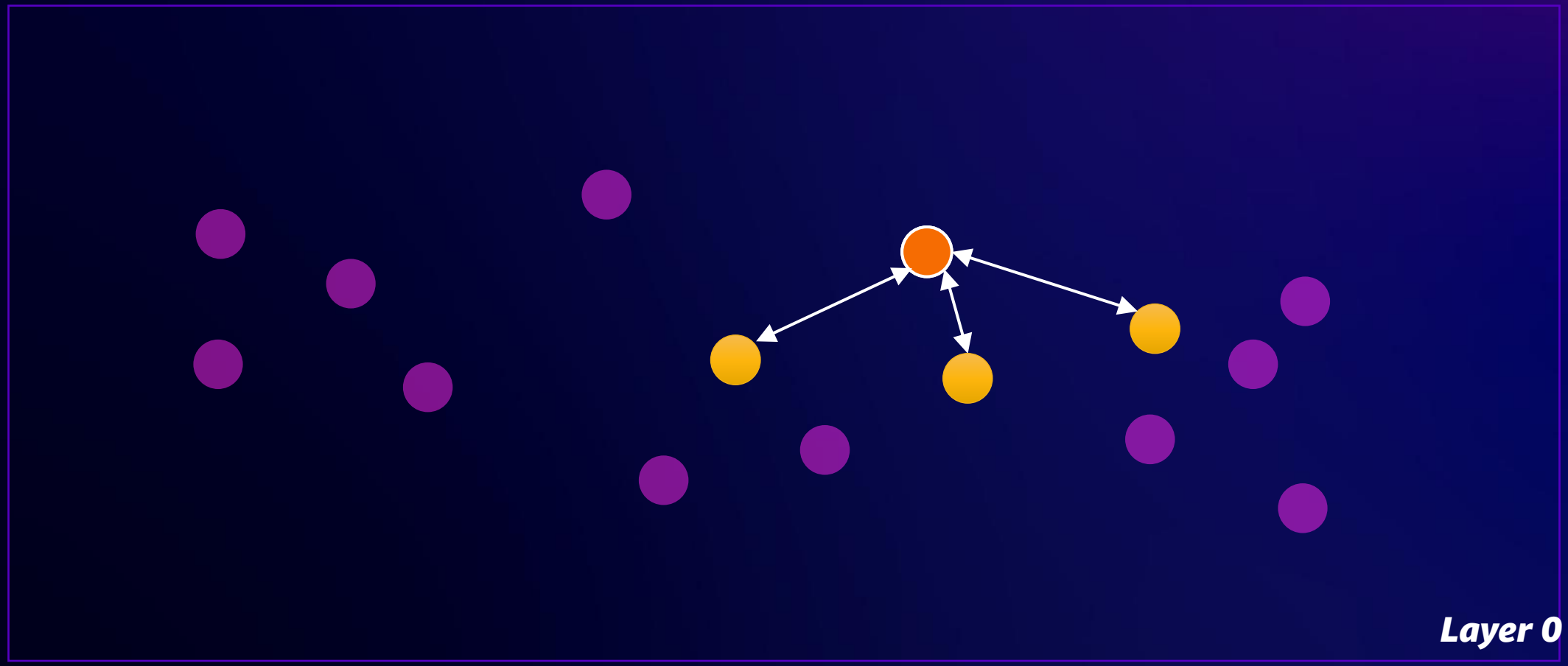
Building an HNSW index



Building an HNSW index



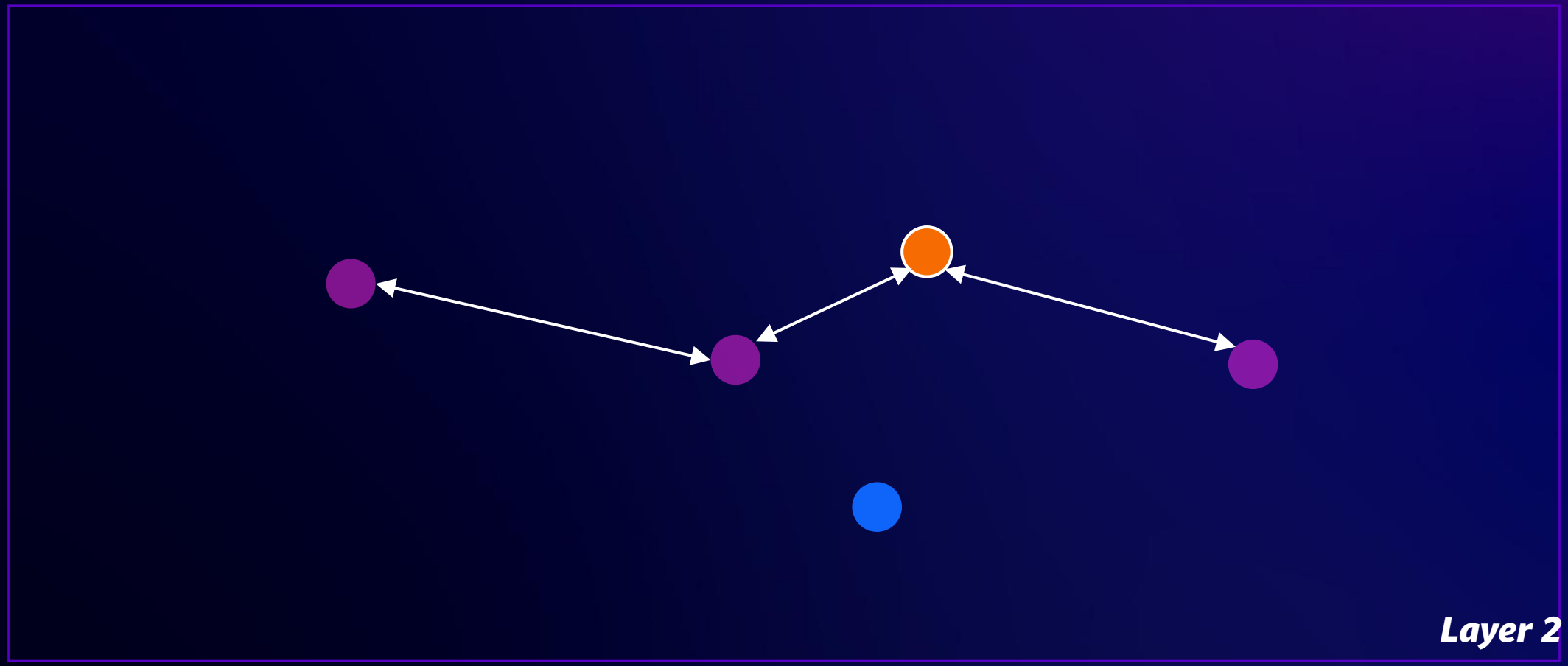
Building an HNSW index



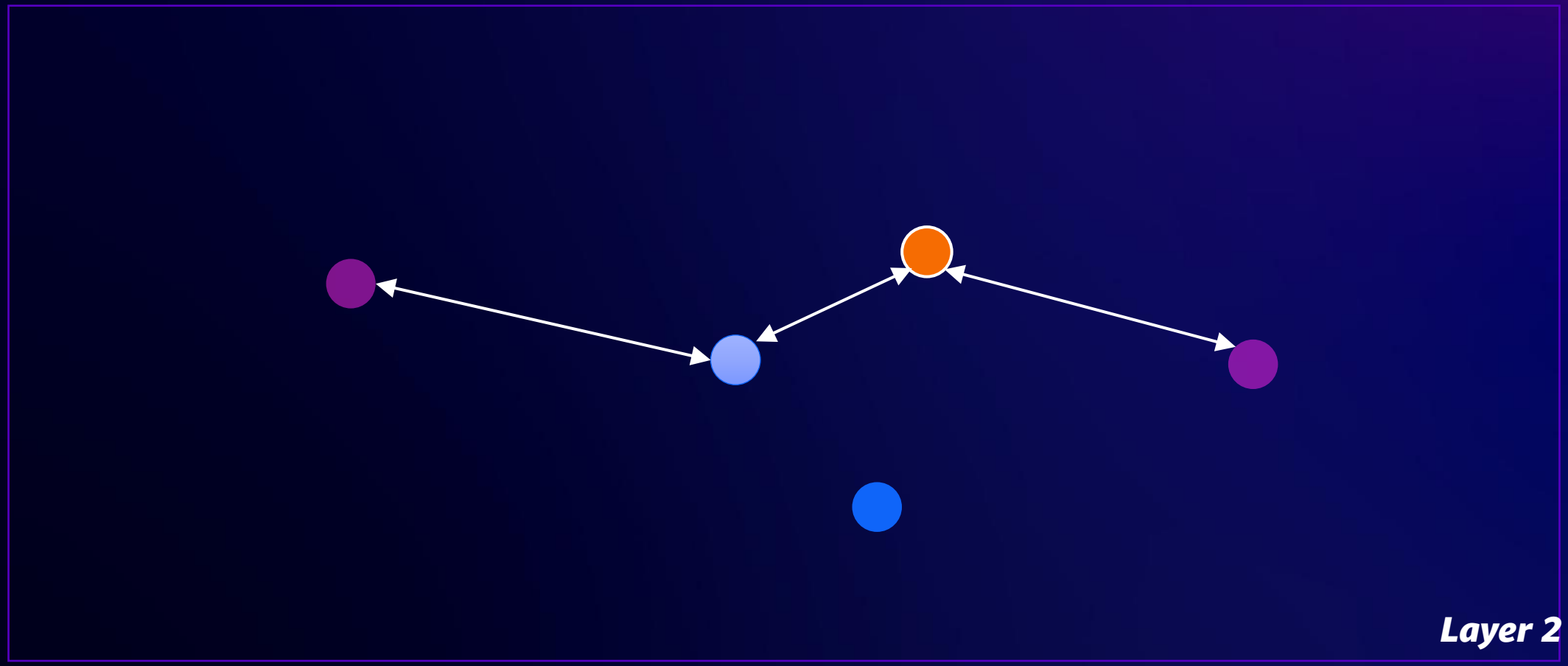
HNSW query parameters

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

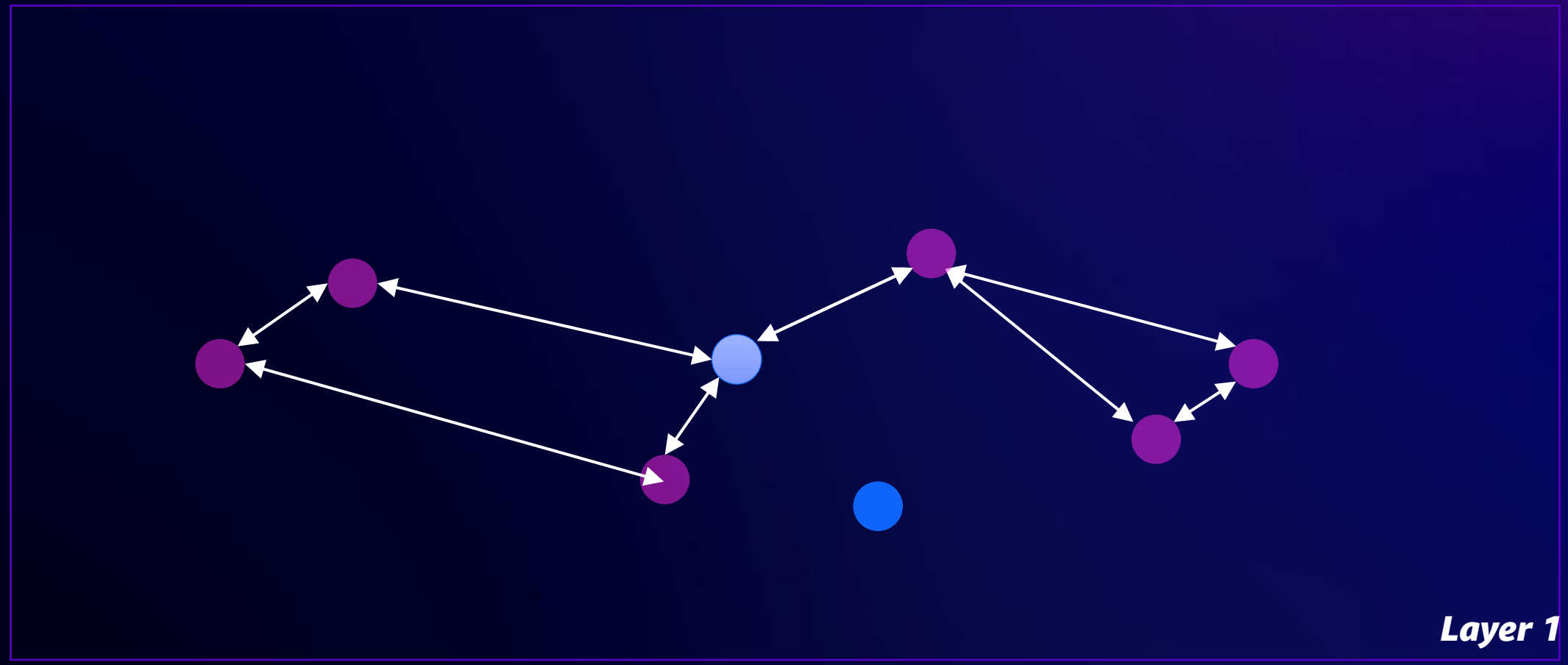
Querying an HNSW index



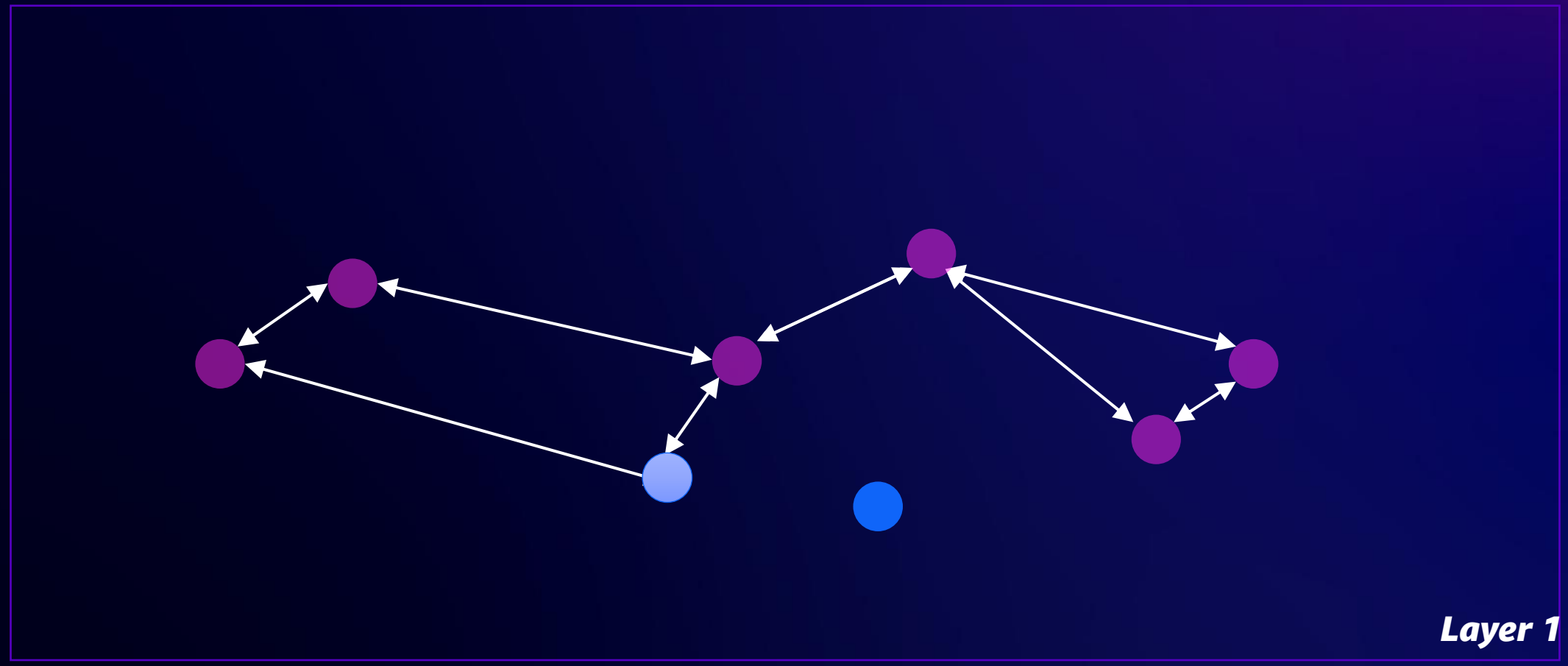
Querying an HNSW index



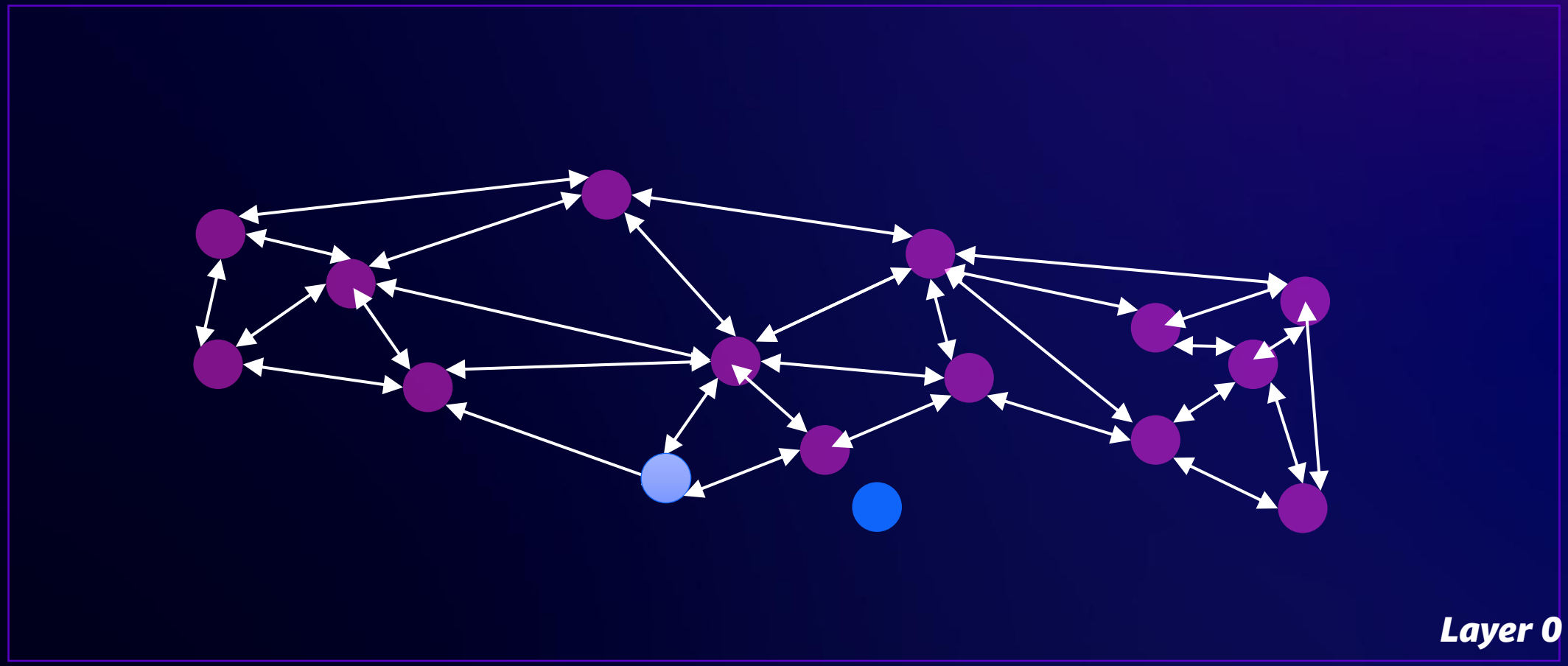
Querying an HNSW index



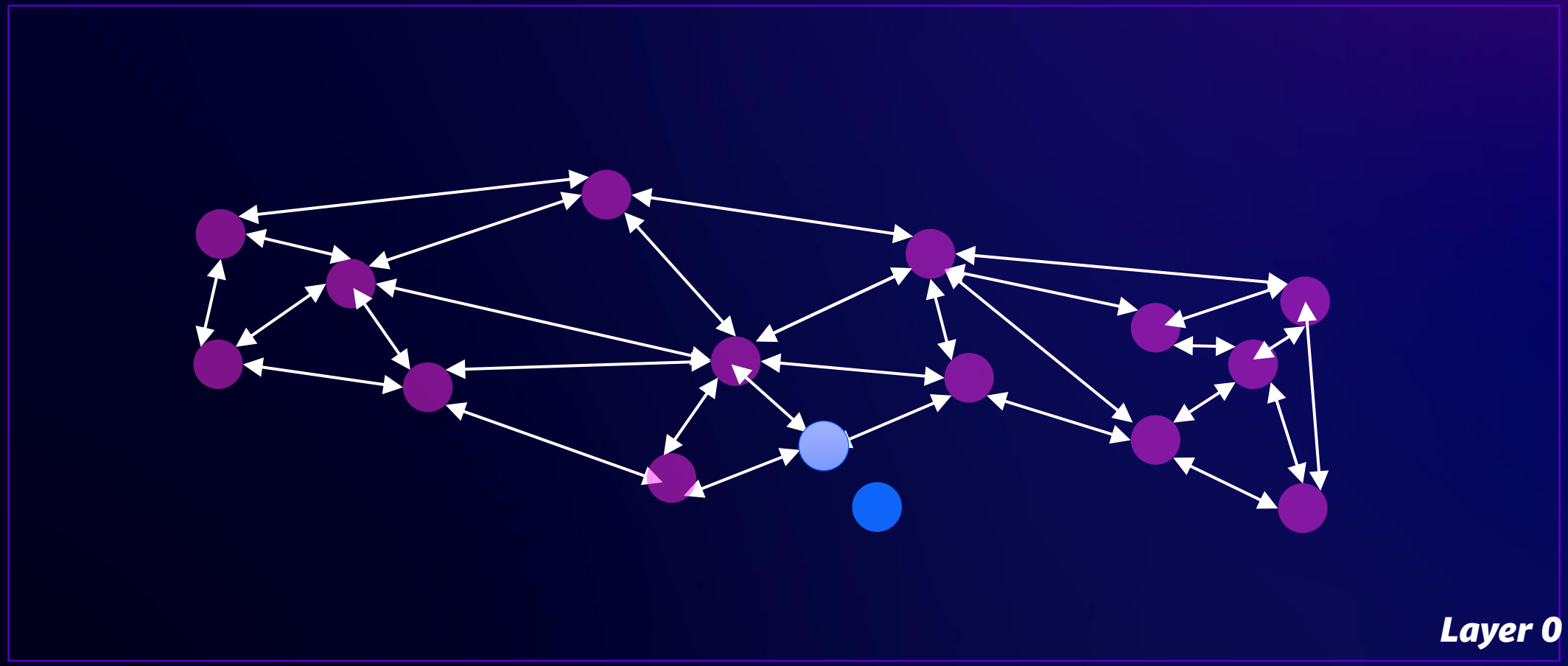
Querying an HNSW index



Querying an HNSW index



Querying an HNSW index



HNSW overview recap

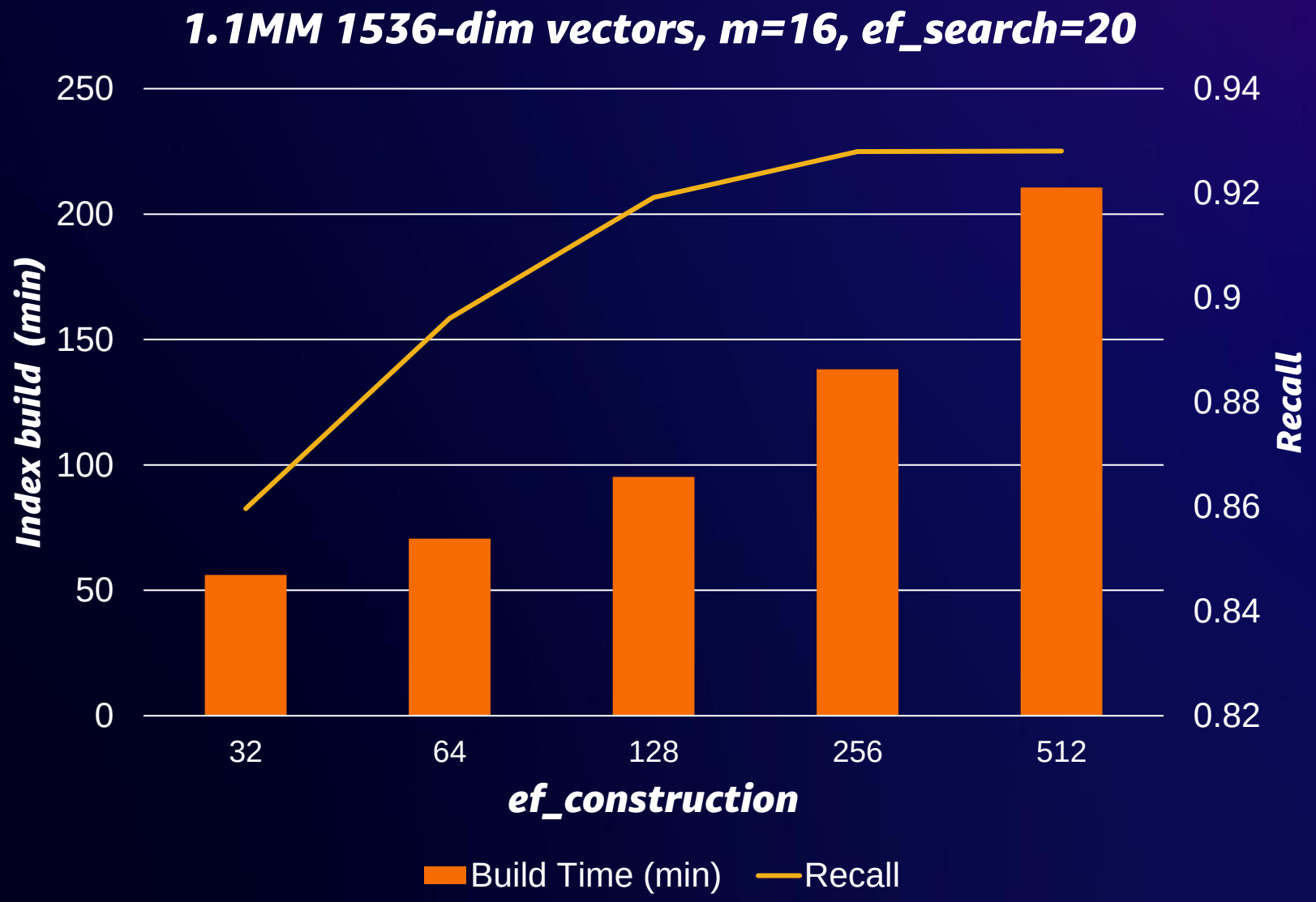
- HNSW does more upfront work in constructing index
- This increases probability that searches are in best neighborhood for nearest neighbors

Best practices for building HNSW indexes

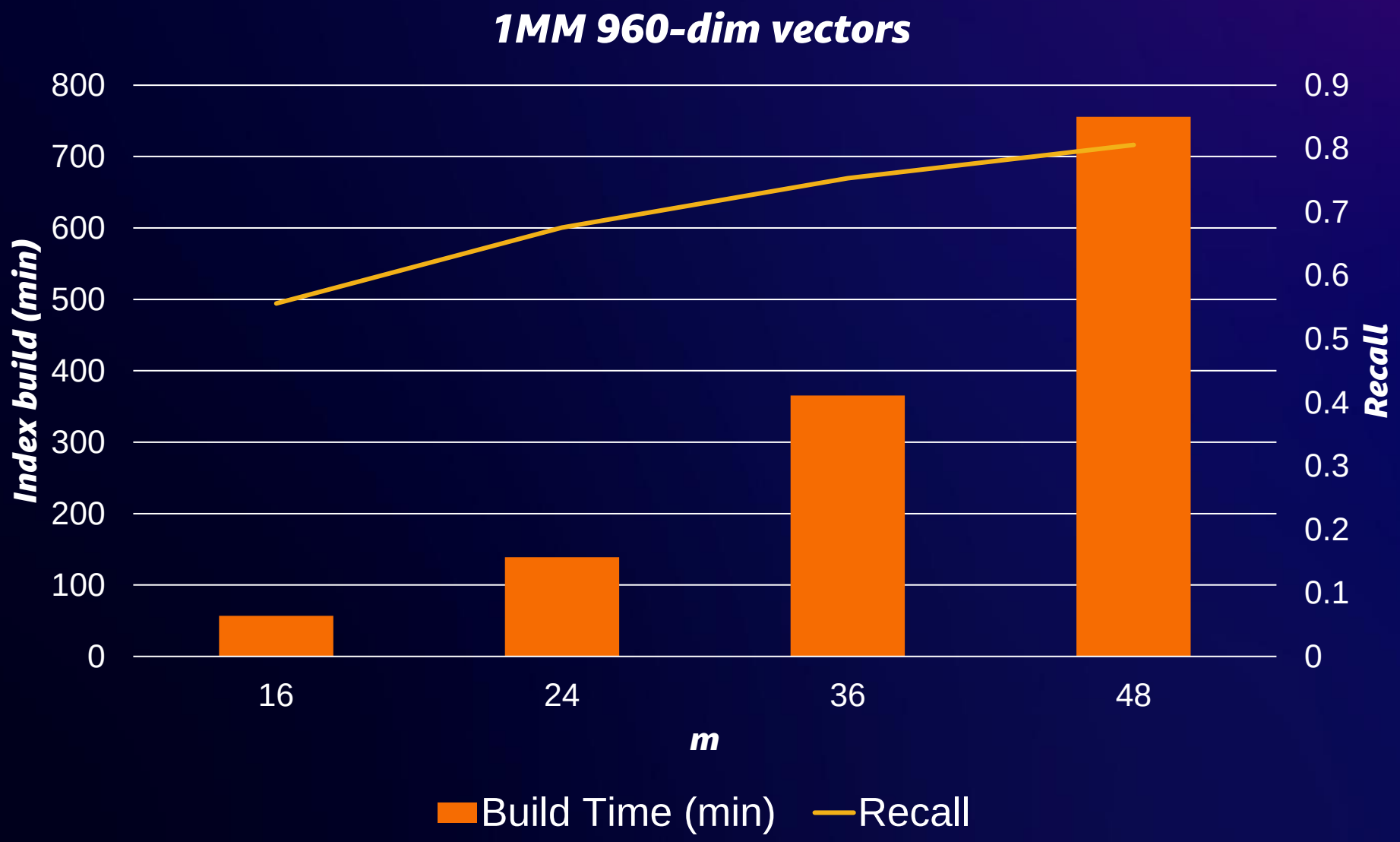
- ***Default values (***`M=16,ef_construction=64`***) usually work***
- ***(pgvector 0.5.1) Start with empty index and use concurrent writes to accelerate builds***
  - `INSERT` ***or*** `COPY`

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

IVFFlat strategies

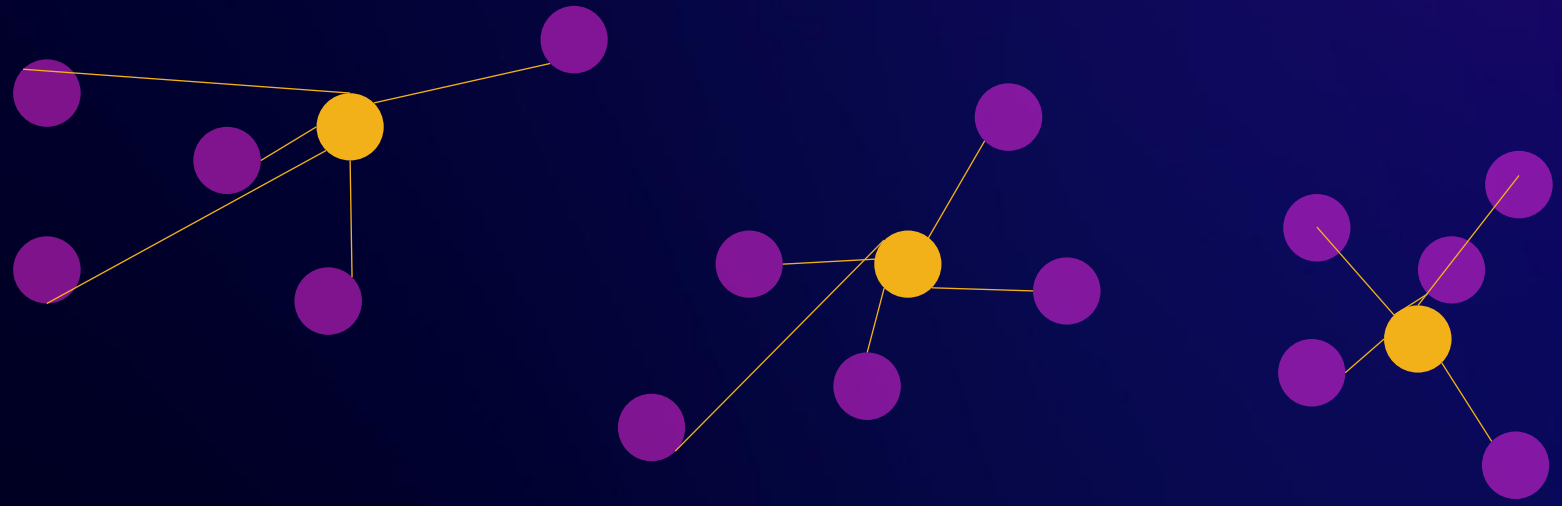
IVFFlat index building parameters

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

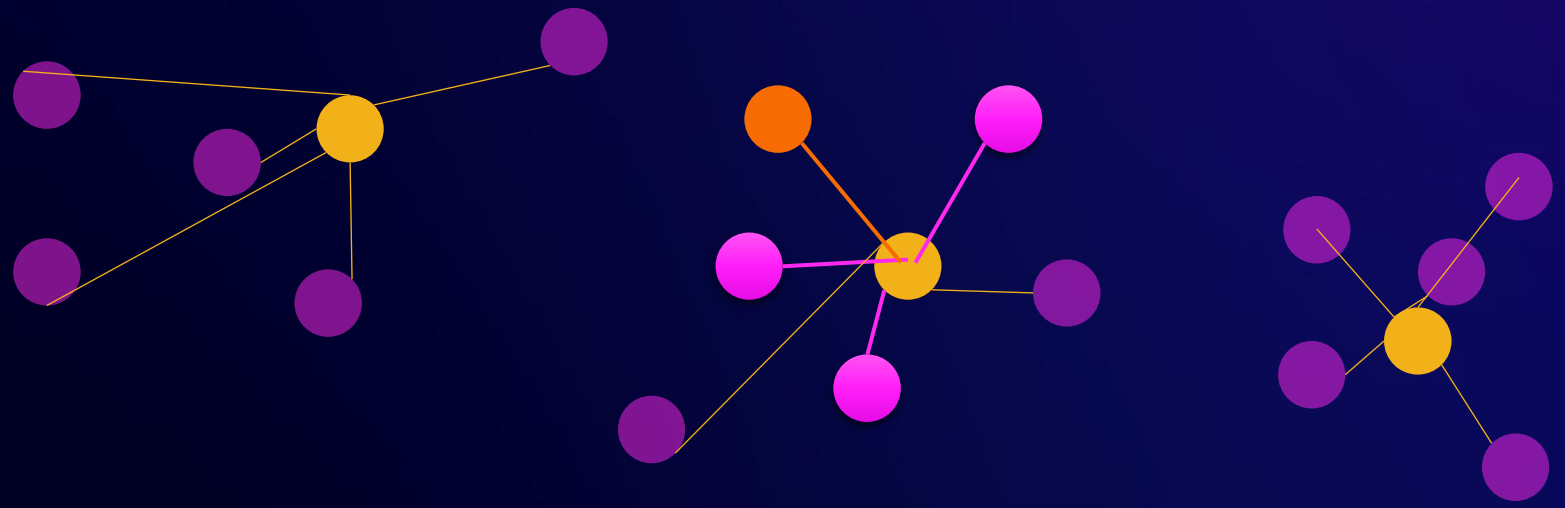
```
CREATE INDEX ON products
USING ivfflat(embedding) WITH (lists=3);
```

Building an IVFFlat index

Building an IVFFlat index: Assign lists



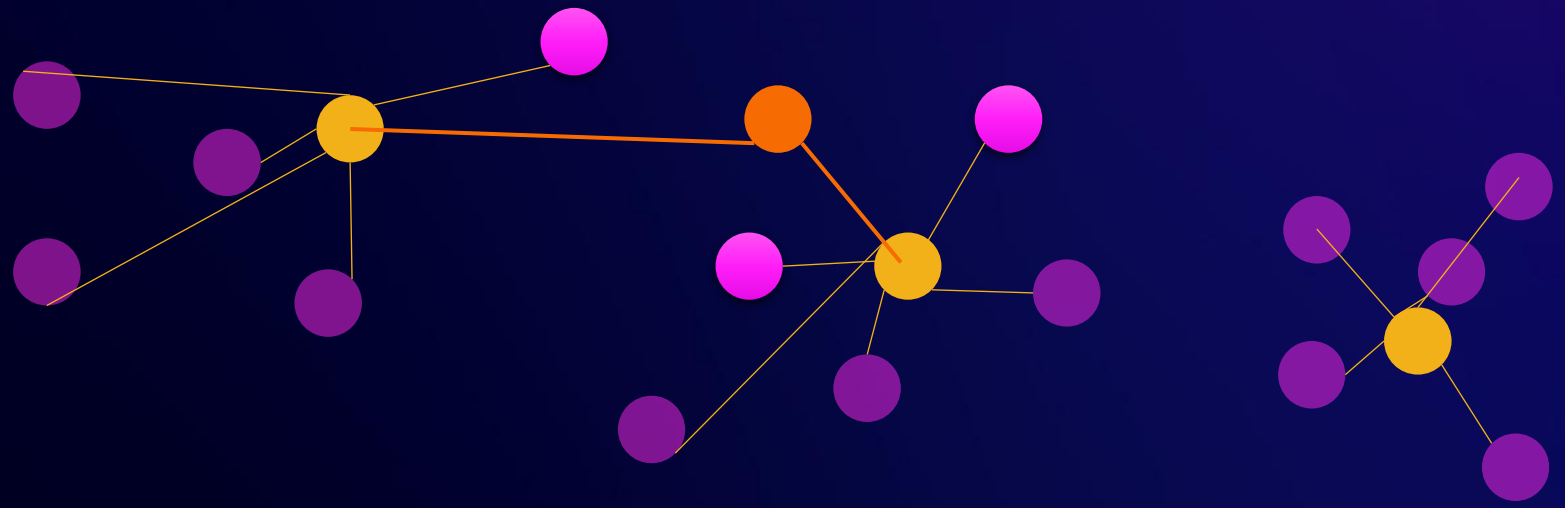
Querying an IVFFlat index



```
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

Performance strategies for IVFFlat queries

- ***Increasing*** `ivfflat.probes` ***increases recall, decreases performance***

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***

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***

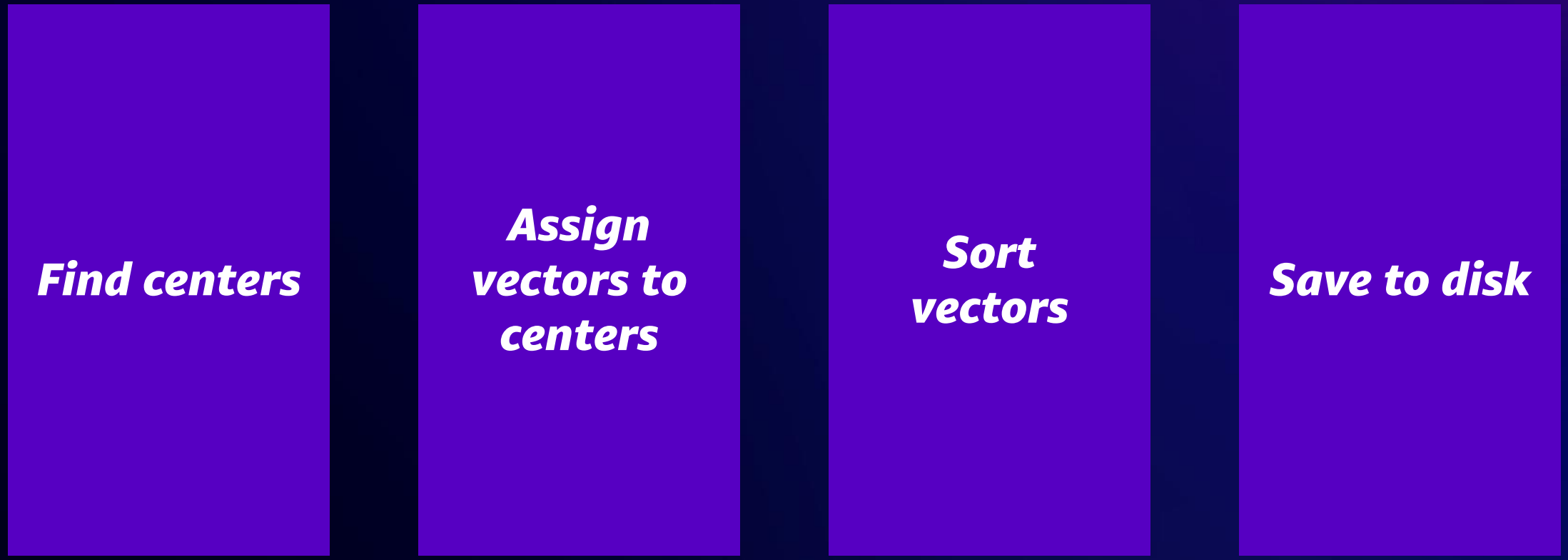
Performance strategies for IVFFlat queries

- ***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***

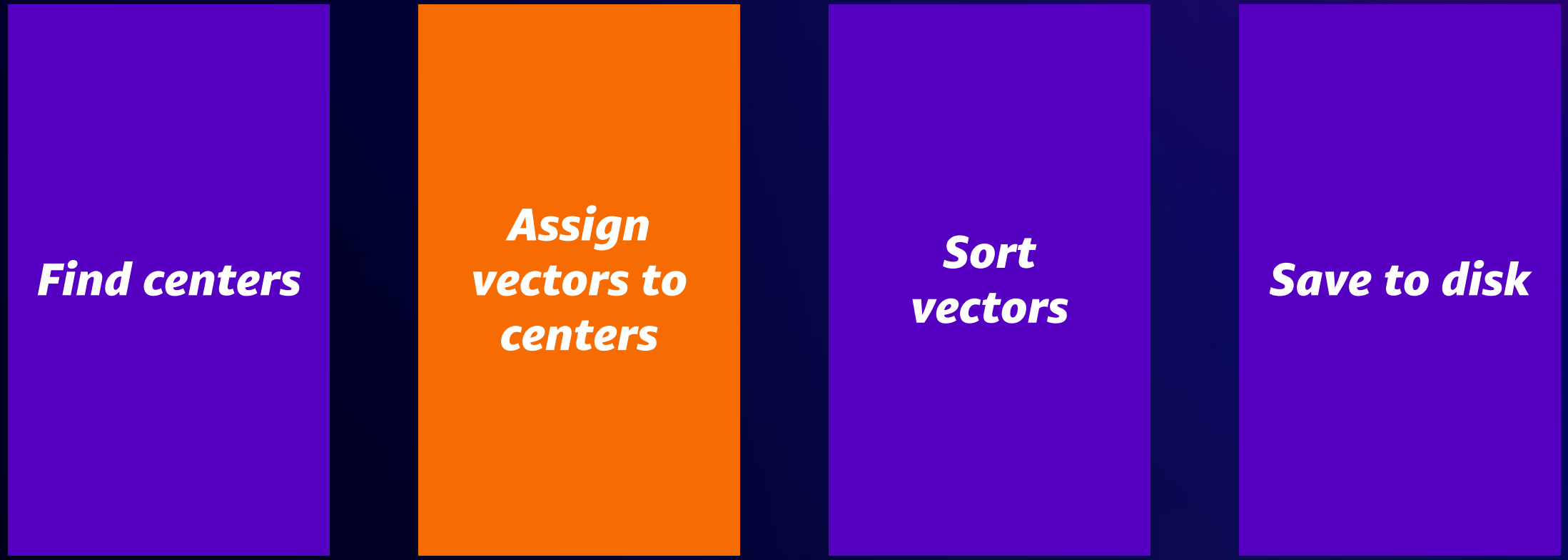
Best practices for building IVFFlat indexes

- 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

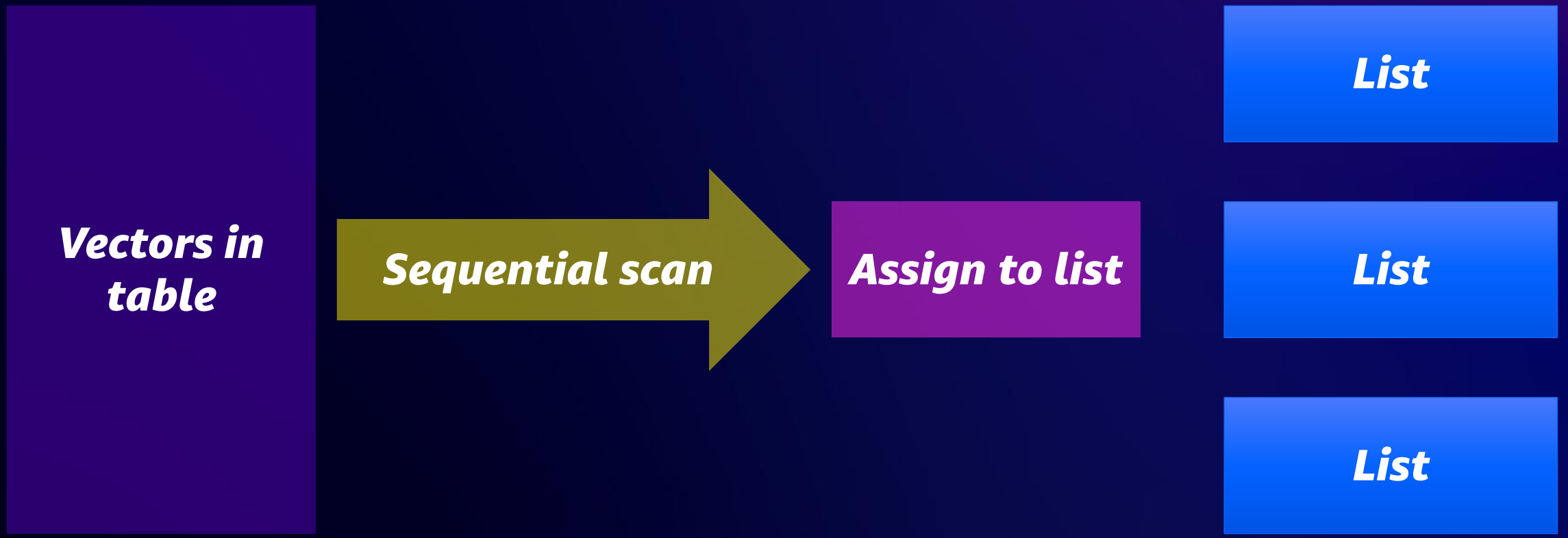
How parallelism works with pgvector IVFFlat



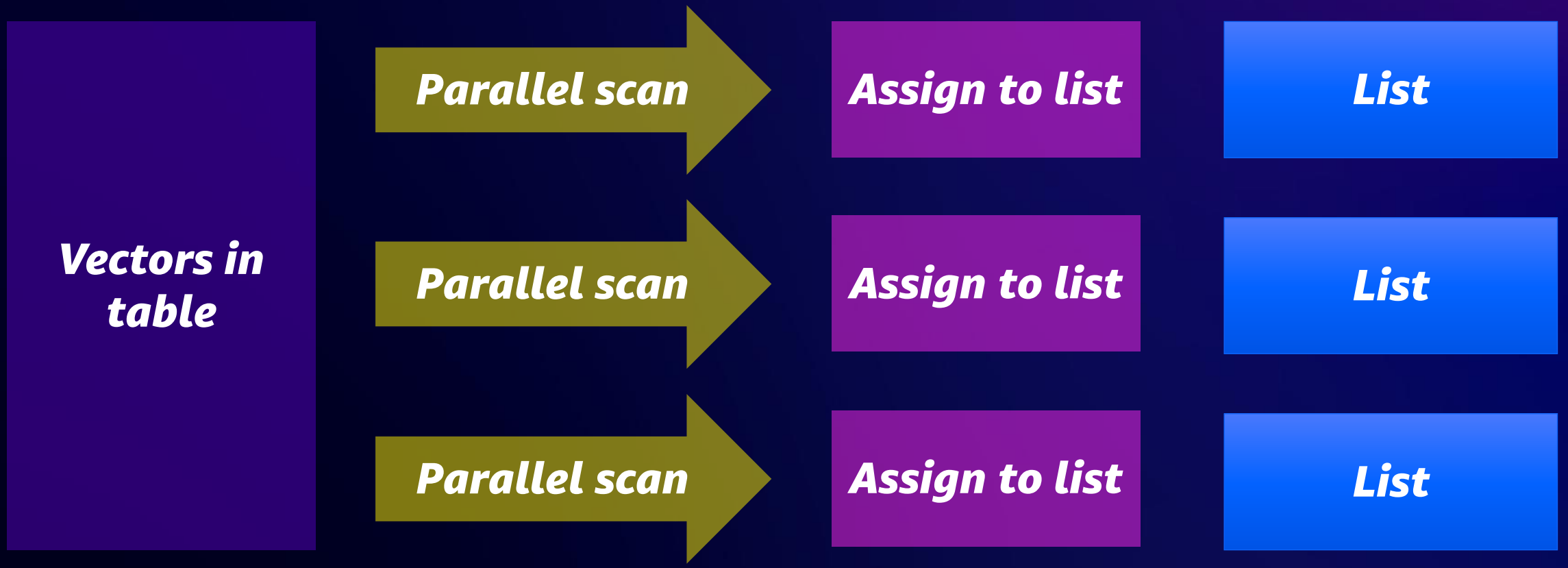
How parallelism works with pgvector IVFFlat



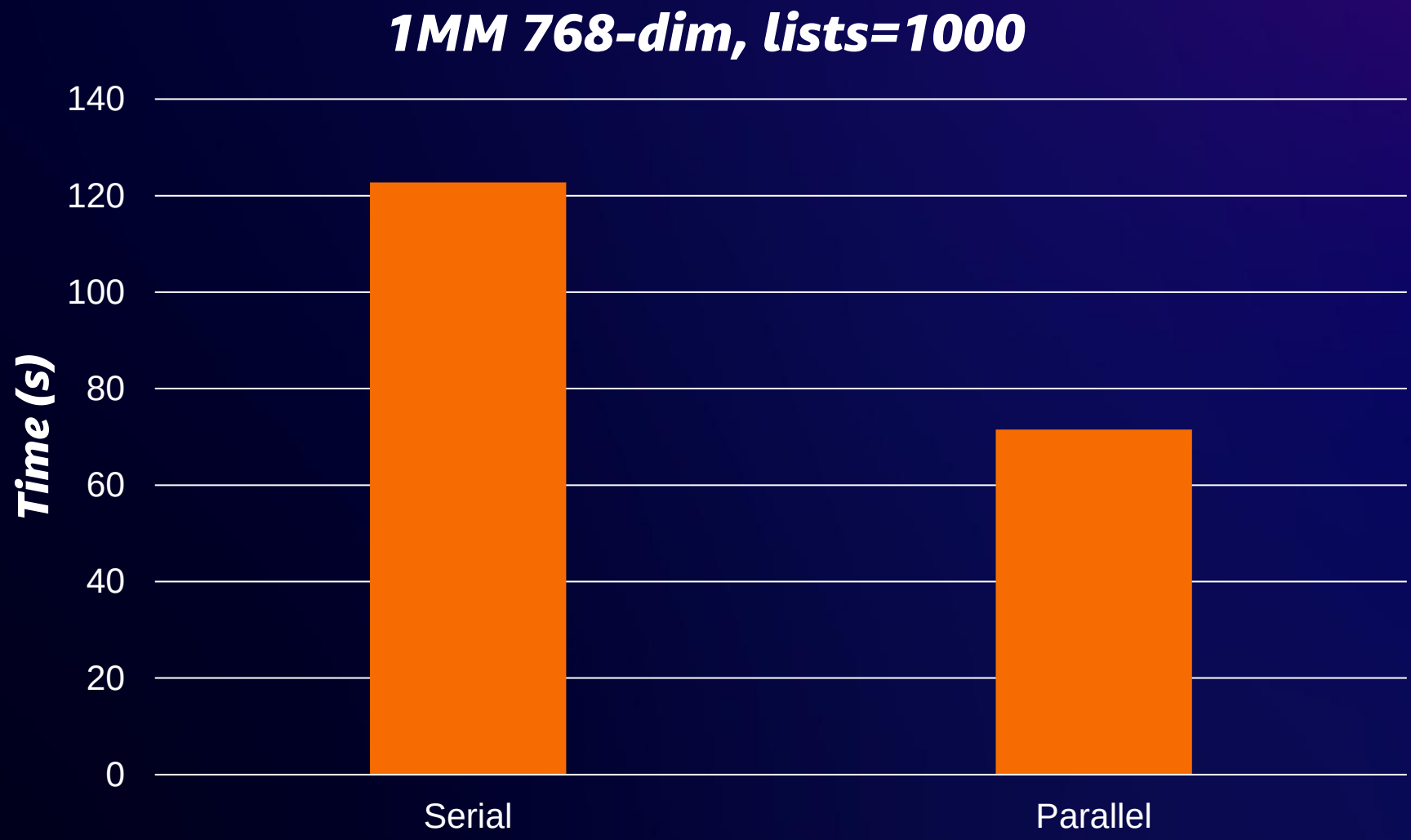
How ~~parallelism~~ works with pgvector IVFFlat



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

Filtering strategies

- **Partial index**

```
CREATE INDEX ON docs
  USING hnsw(embedding vector_l2_ops)
  WHERE category_id = 7;
```

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

```
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;
```

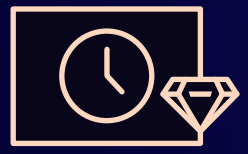
Amazon Aurora features

Amazon Aurora overview

Designed for unparalleled high performance and availability at global scale with full MySQL and PostgreSQL compatibility at 1/10th the cost of commercial databases

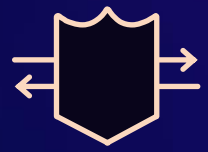
Performance and scalability

- 5x throughput of standard MySQL and 3x of standard PostgreSQL
- Scale out up to 15 read replicas in region and 90 read replicas using global database
- Decoupled storage and compute enabling cost optimization
- Fast database cloning
- Distributed, dynamically scaling storage subsystem



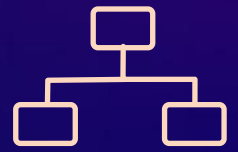
Availability and durability

- 99.99% availability with Multi-AZ
- Data is durable across 3 AZs (customers only pay for 1 copy)
- Automatic, continuous, incremental backups with point-in-time recovery (PITR)
- Failovers in < 10 seconds
- Fault-tolerant, self-healing, auto scaling storage
- Global database for disaster recovery



Highly secure

- Network isolation
- Encryption at rest/in transit
- Supports multiple secure authentication mechanisms and audit controls



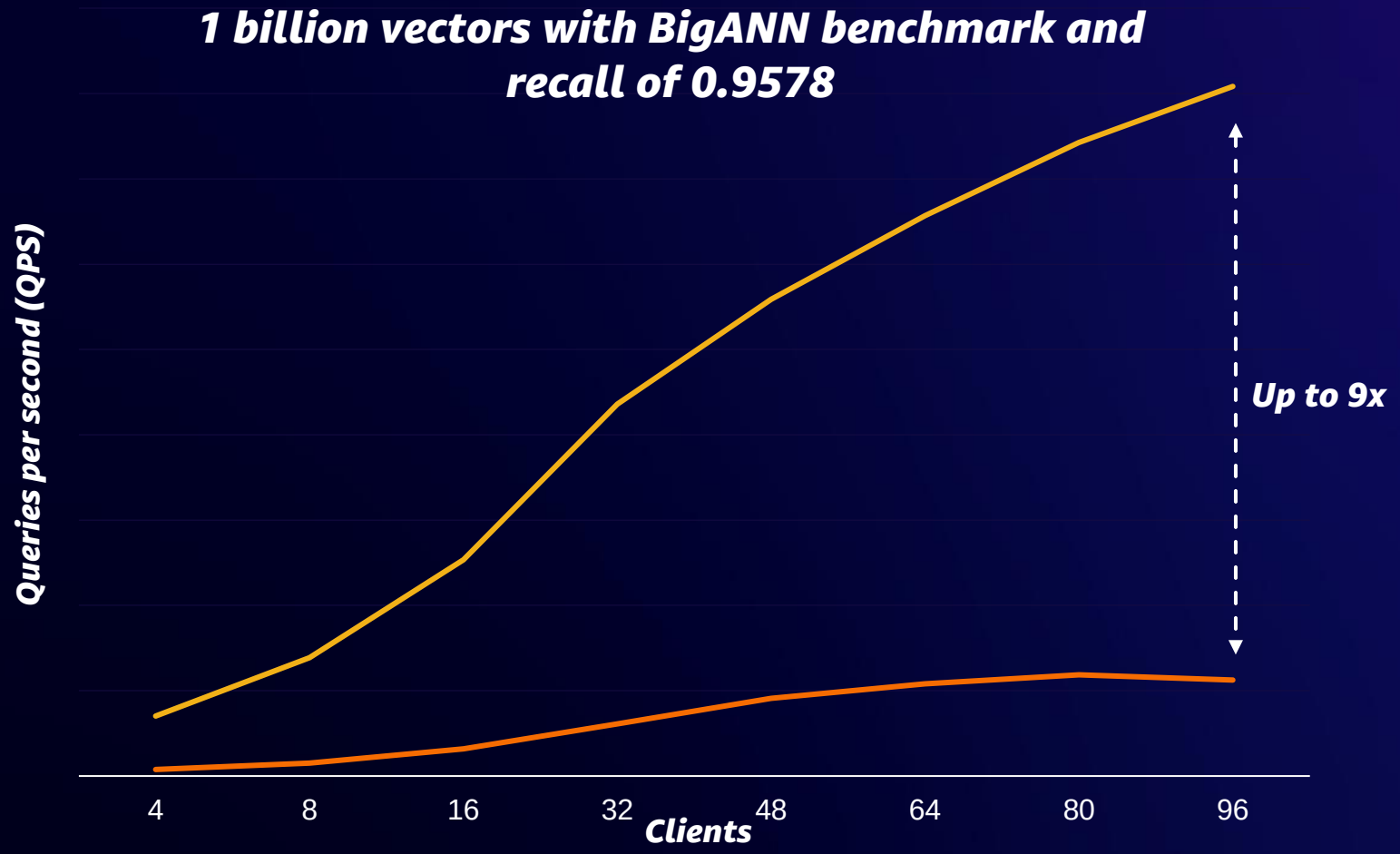
Manageability at scale

- Automates time-consuming management of administration tasks like hardware provisioning, database setup, patching, and backups
- Serverless configuration options

Amazon Aurora features for vector workloads

- Amazon Aurora PostgreSQL with Optimized Reads
  - NVMe caching
- Instance types (r7g)
- Compatibility with frameworks like LangChain

Vector performance with tiered caching

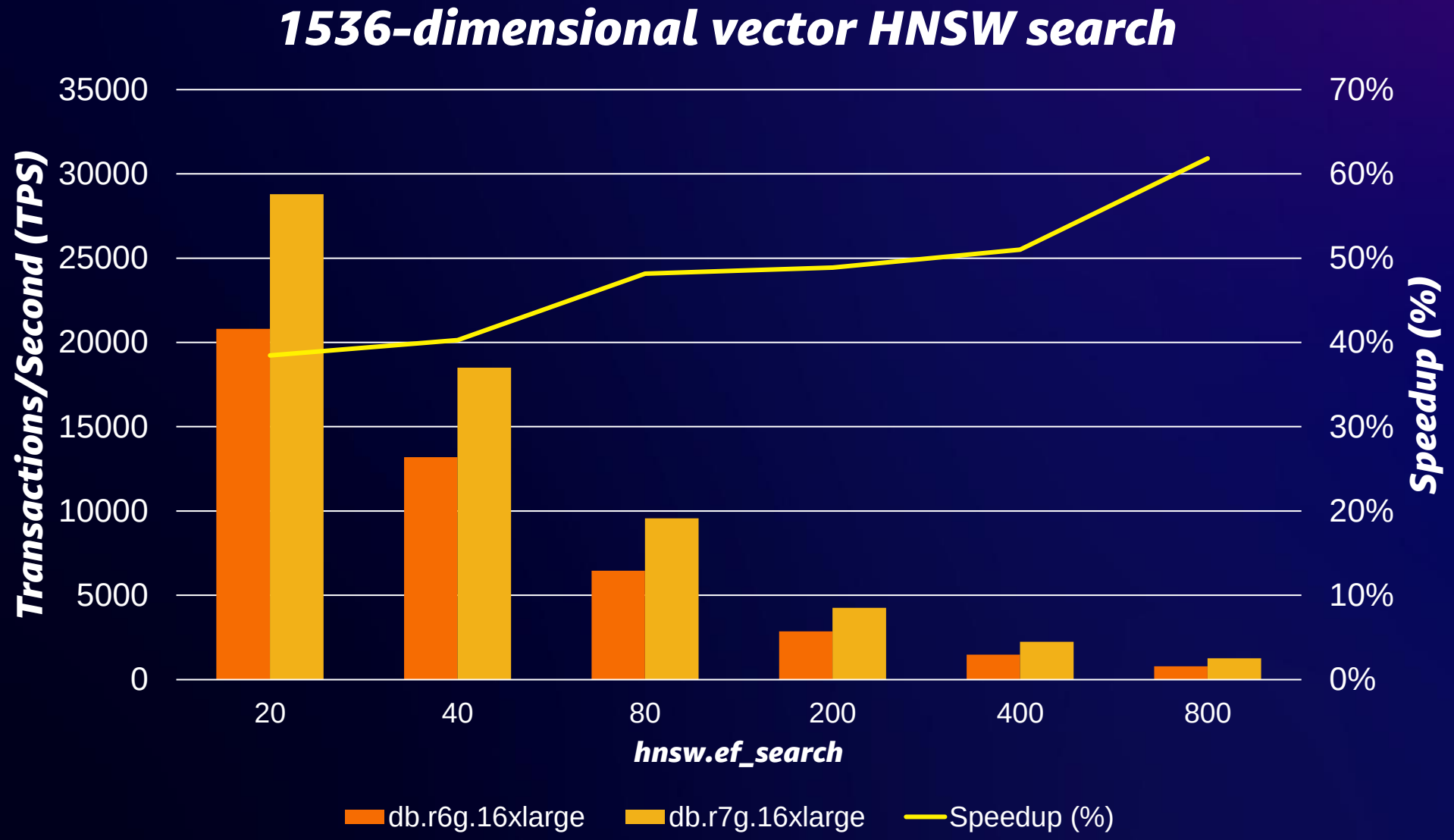


Amazon Aurora PostgreSQL I/O-Optimized with tiered caching and pgvector increases queries per second for vector search by up to 9x in workloads that exceed available instance memory

Database size: 1.28TB (Data: 560 GB, Index: 720GB)

pgvector v0.5: HNSW index

Accelerating queries with AWS Graviton3



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

- Primary design decision: query performance and recall
- Determine where to invest: storage, compute, indexing strategy
- Plan for today and tomorrow: pgvector is rapidly innovating

Continue your journey

- **DAT413-R1**: Using LangChain to build gen AI apps with Amazon Aurora and pgvector
  - Nov 28: 5:30pm-6:30pm; Wynn – Level 1 – Lafite 2

- **DAT212-INT**: Future-proofing your applications with AWS databases
  - Nov 29: 2:30pm-3:30pm; Venetian – Level 5 – Palazzo Ballroom B

- **DAT323-R / DAT323-R1**: Discovering the vector database power of Amazon Aurora & Amazon RDS
  - Nov 29: 4:30pm-5:30pm; Mandalay Bay – Level 1 North – South Pacific B
  - Dec 1: 8:30am-9:30am; Caesars Forum – Level 1 – Forum 104

Thank you!

Please complete the session survey in the mobile app

Jonathan Katz

jkatz@amazon.com