AWS re:Invent

DECEMBER 2 – 6, 2024 | LAS VEGAS, NV

DAT423

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

Jonathan Katz

(he/him)
Principal Product Manager – Technical
AWS

Agenda

01 Overview of generative AI and the role of databases

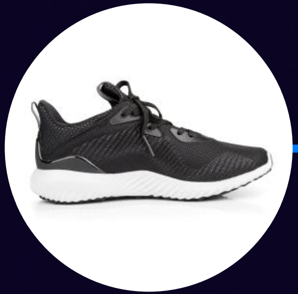
02 PostgreSQL as a vector store

03 pgvector strategies and best practices

04 Amazon Aurora features for vector search

05 Looking ahead: pgvector roadmap

Overview of generative AI and the role of databases



CUSTOMER

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

DEVELOPER CREATED AGENT

Of course, do you have your order number?

Tags

Human: You are an agent who manages orders and
Input, create an orchestration plan for executing

Emphasis (capitalized)

Valid "api" values are GetOrderHistory::GetProduct
- DO NOT return an api if all required parameter
- DO NOT replace the placeholders in the api_name
- Return available parameters in api_inputs ONLY,

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

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

Convergence criteria

DO NOT go into a loop and return exact same

Format (JSON)

Provide only ONE action per $JSON_BLOB, as shown

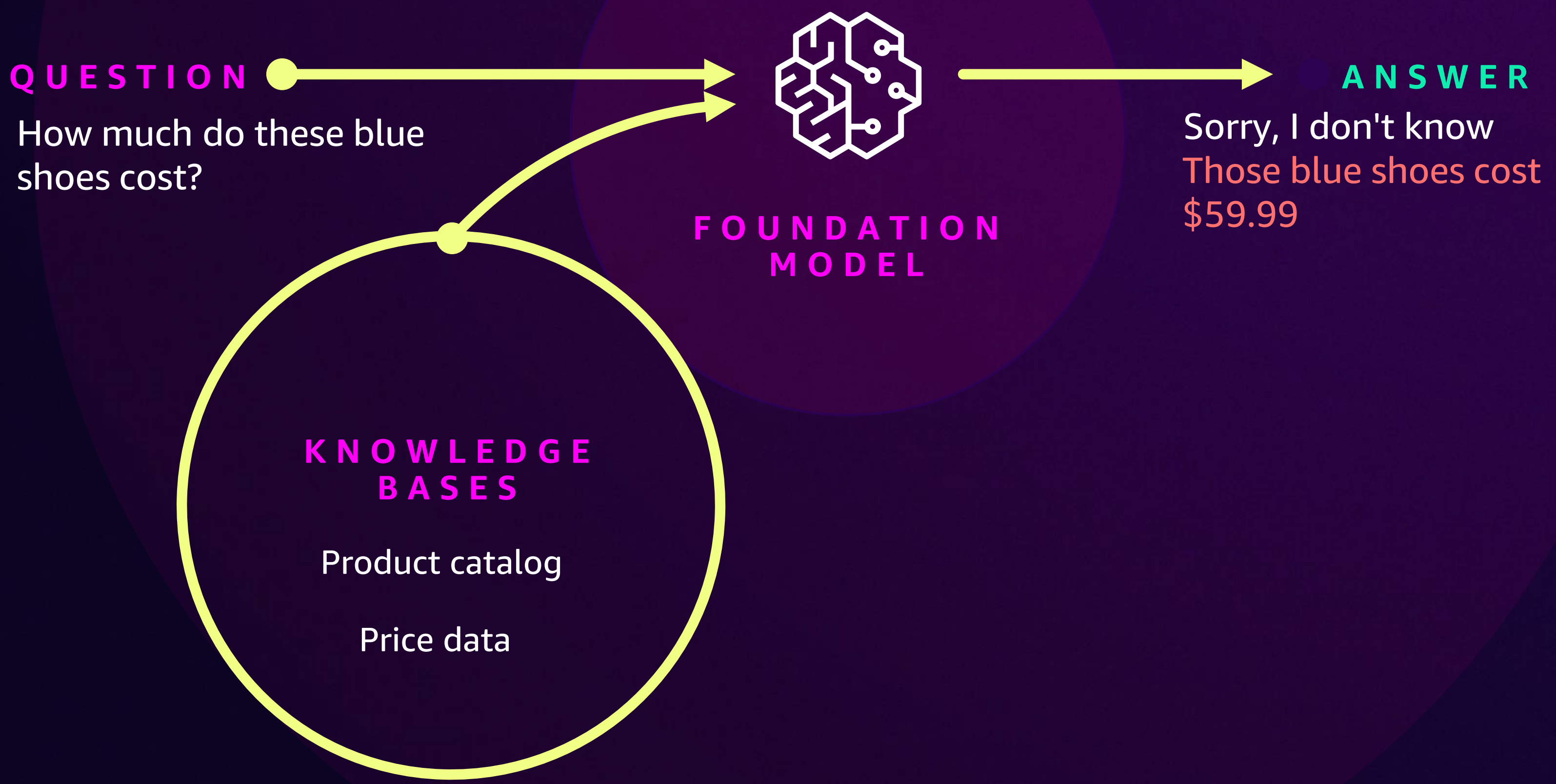
{ "api": $API_NAME, "verb": $HTTP_VERB, "api_input

History format

Conversation History: Below is the history of the

Retrieval Augmented Generation (RAG)

Configure foundation model to interact with your 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…

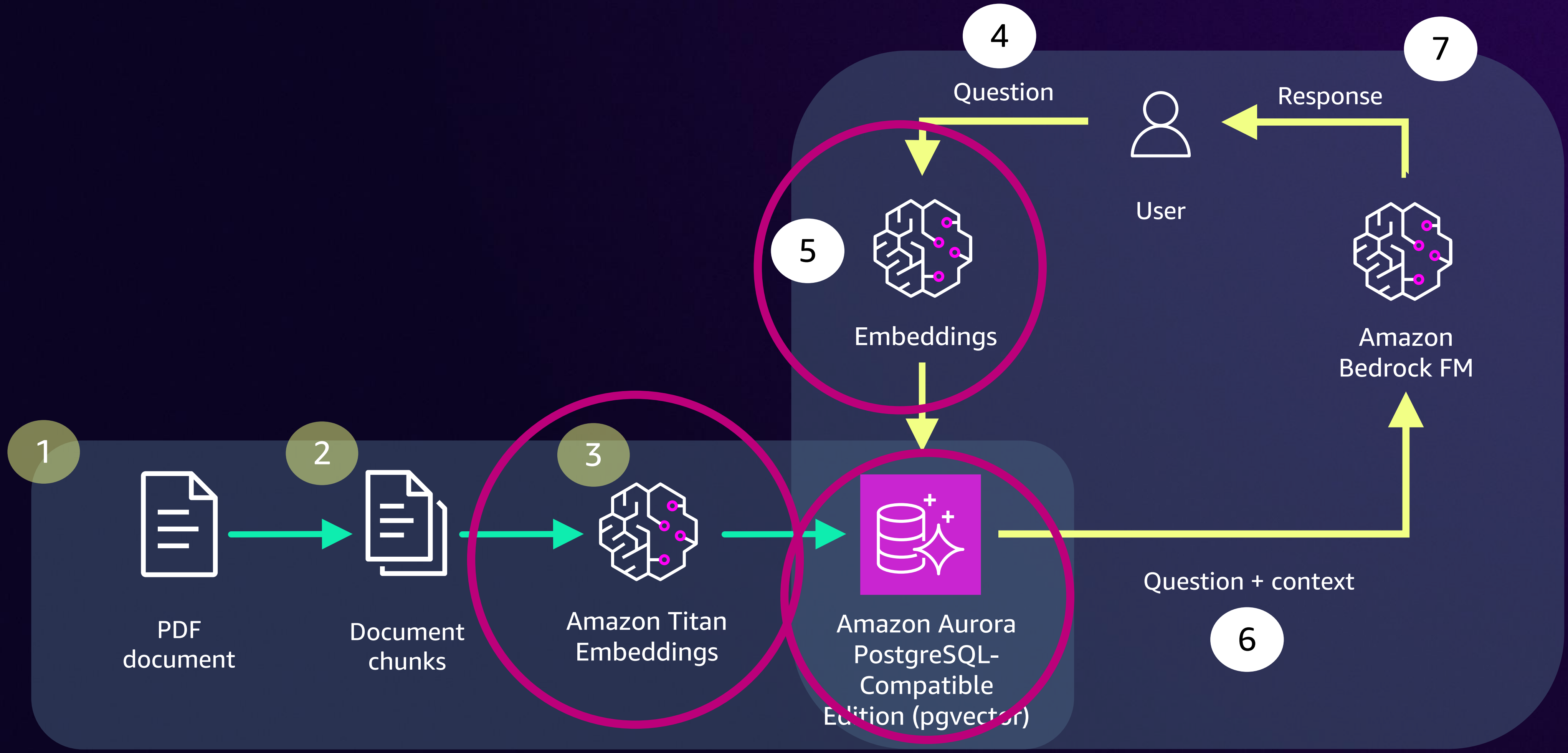
0.35 0.1 0 0.8 001.0 00 0001.0 0 0…

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

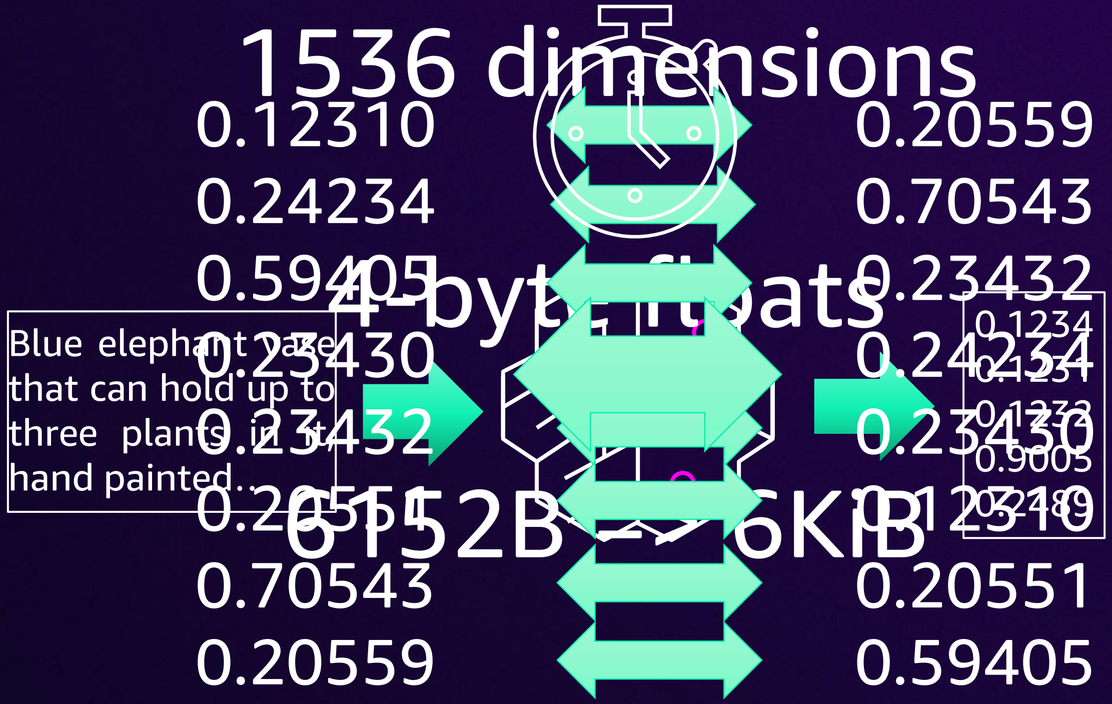
Embeddings: When vector elements are semantic, used in generative AI

How vector embeddings are used



Challenges with vectors

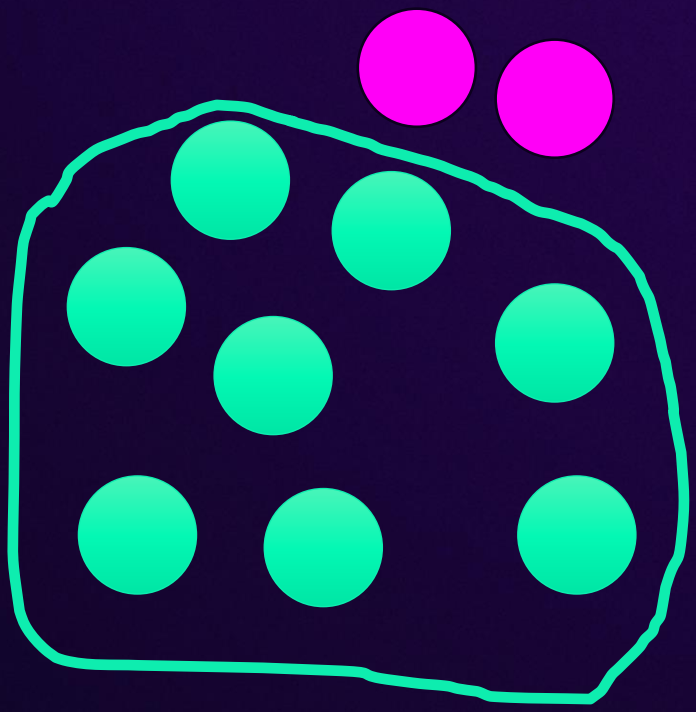
- Time to generate embeddings
- Embedding size
- ~~Compression~~
- Query time



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

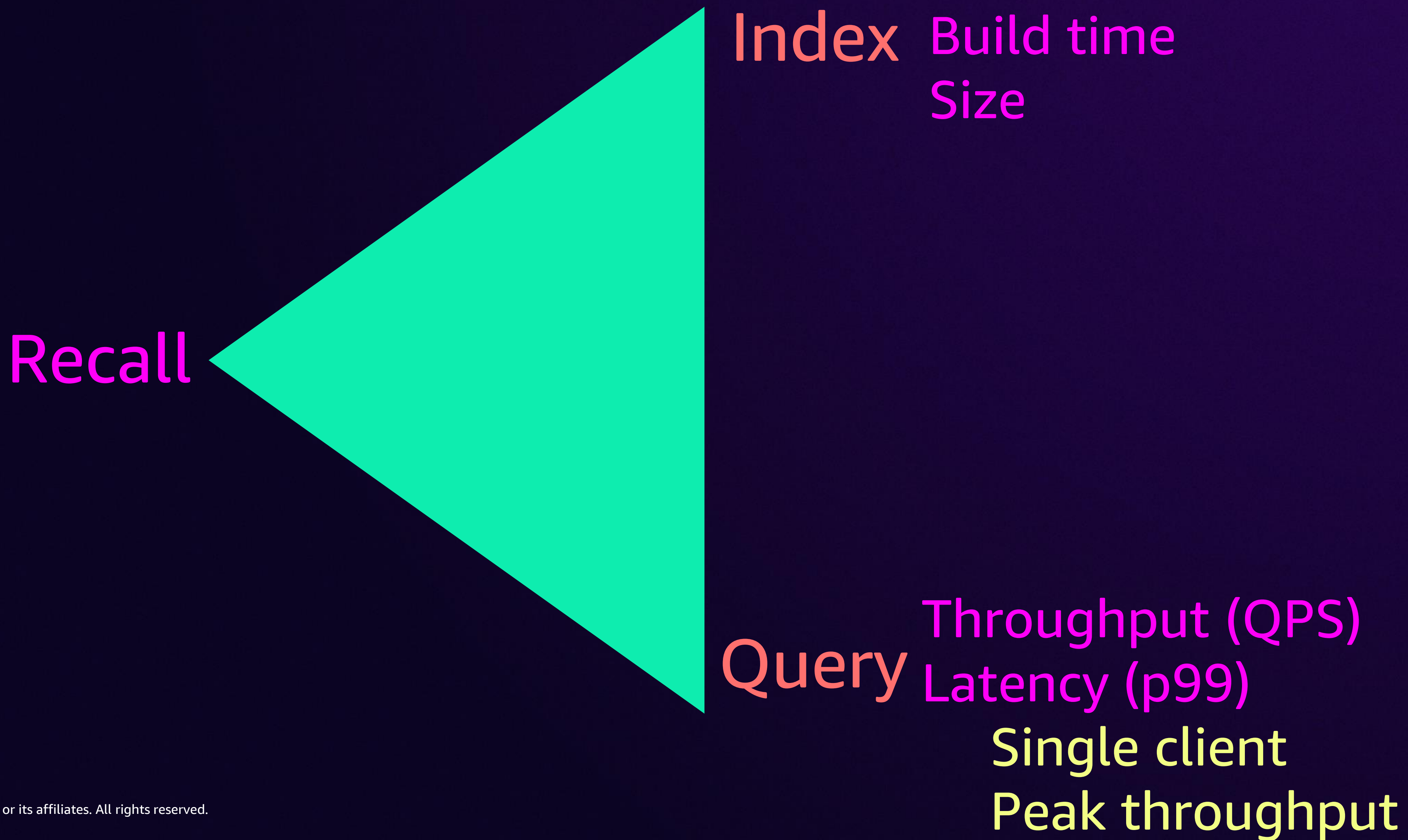


Recall: 80%

Criteria that impact selection of vector storage

- Cost
- Choice of embedding / foundation model
- Ease of development
- Query performance targets
- Data ingestion patterns

Metrics for evaluating a vector storage system



What impacts vector storage performance metrics?

- Choice of embedding model
- Number of vector dimensions
- Choice of indexing algorithm
- Query patterns (full search, filtering, hybrid search)
- Quantization
- Ingestion / modification patterns
- Infrastructure resources/utilization

Enabling vector search across our services

Amazon OpenSearch Service

Amazon OpenSearch Serverless

Amazon Aurora PostgreSQL-Compatible

Amazon RDS for PostgreSQL

Amazon DocumentDB

Amazon DynamoDB via zero-ETL

Amazon MemoryDB

Amazon Neptune

PostgreSQL as a vector store

Why PostgreSQL?

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
  - 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?

- **A**tomicity: "All or nothing" stored in transaction (bulk loads)
- **C**onsistency: Follows rules for other data stored in database
- **I**solation: Correctness in returned results; committed transactions "immediately available"
- **D**urability: Once 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

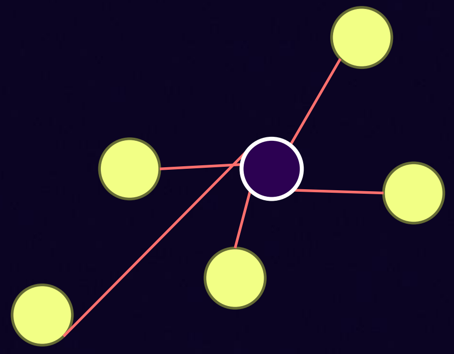
- High-performance vector searches
- Support for larger vectors
- Sustained, rapid improvements
- Better support in developer tools

pgvector: Year-in-review timeline

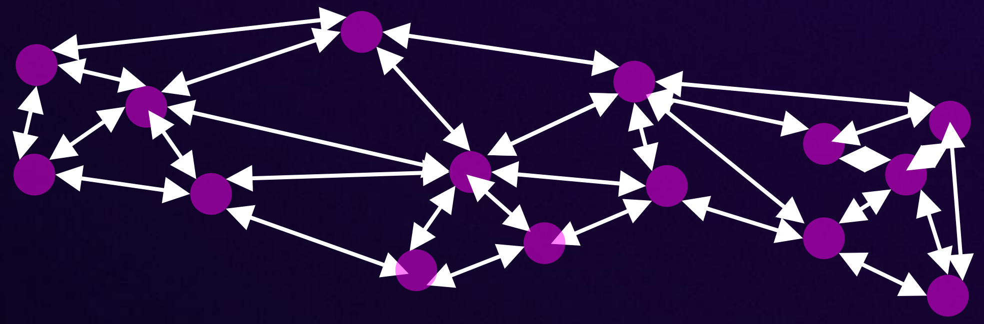
- v0.4.x (1/2023 – 6/2023)
  - Improved IVFFlat cost estimation
  - Store higher dimensional vectors
- v0.5.x (8/2023 – 10/2023)
  - Add HNSW index + distance function performance improvements
  - Parallel IVFFlat builds
- v0.6.x (1/2024 – 3/2024)
  - Parallel HNSW index builds + in-memory build optimizations
- v0.7.x (4/2024 – 9/2024)
  - halfvec (2-byte float), bit(n) index support, sparsevec (up to 1B dim)
  - Quantization (scalar/binary), Jaccard/Hamming distance, explicit SIMD
- v0.8.x (10/2024)
  - Iterative index scans
  - HNSW search memory reduction / insert speedups
  - Improved HNSW cost estimation

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
- Filtering: Depends on your selectivity
  - High selectivity (most results filtered out): B-tree / GIN / GiST / BRIN / no index
  - Low selectivity: HNSW (+ iterative scan)

pgvector strategies and best practices

Best practices for pgvector

01 Storage configuration

02 HNSW build and search parameters

03 Filtering

04 Quantization

Best practices: Storage configuration

How does PostgreSQL store vectors?

- Page: PostgreSQL atomic storage unit
  - 8192 bytes = 8K = 8KiB
- Vector indexes have “wasted space”
  - 1,536-dim vector (6 KiB) has 2KiB “empty space”

(1,[1,2,3]),(2,[2,3,4]),(3,[3,4
,5]),(4,[4,5,6]),(5,[5,6,7]),(6,
[6,7,8]),(7,[7,8,9]),(8,[8,9,1
0]),(9,[9,10,11]),(10,[10,11,
12]),(11,[11,12,13]),(12,[12
,13,14]),(13,[13,14,15]),(14
,[14,15,16]),(15,[15,16,17])
,(16,[16,17,18]),(17,[17,18,
19]),(18,[18,19,20]),(19,[19
,20,21])

How vector size impacts space utilization

Dimensions	Vectors / page	Wasted space (B)
128	15	308
256	7	916
384	5	428
512	3	1,988
768	2	2,000
1,024	1	4,060
1,536	1	2,012
2,000	1	156

```
PAGE_SIZE - PAGE_HEADER - (VECTORS * 4) - VECTORS * (4 * DIMS + 8)
```

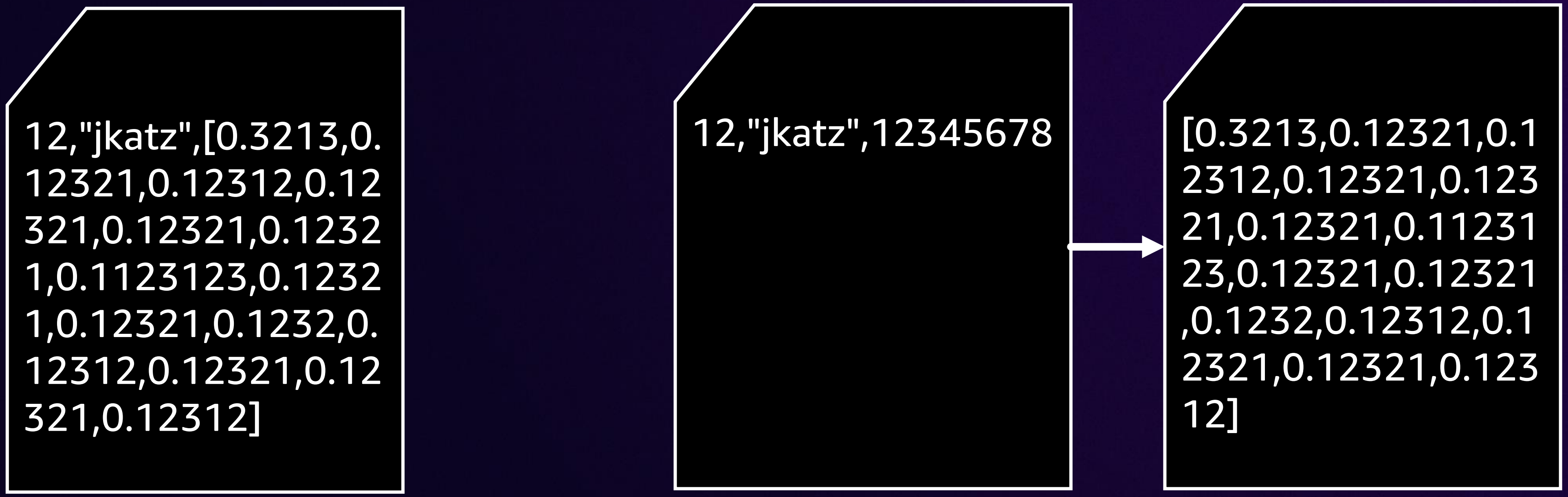
Vectors and page sizes

- 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 (<u>T</u>he <u>O</u>versized-<u>A</u>ttribute <u>S</u>torage <u>T</u>echnique) 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
  - <u>EXTERNAL</u>: Data stored in TOAST table when threshold exceeded
    - pgvector default 0.6.0+
  - EXTENDED: Data stored/compressed in TOAST table when threshold exceeded
    - pgvector default before 0.6.0

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

How storage selection impacts QPS

5,000 cosine distance operations (<=>) – single connection (r7i.16xlarge)				
	PLAIN		EXTERNAL	
Dimensions	p50 (ms)	QPS	p50 (ms)	QPS
128	1.2	863	1.2	814
256	1.5	655	1.5	658
384	1.7	591	1.7	587
512	2.0	491	9.8	102
768	2.4	410	10.7	93
1,024	3.5	288	12.0	83
1,536	4.1	246	16.2	62

```
SELECT $1 <=> embedding AS distance FROM embeddings ORDER BY distance LIMIT 5000;
```

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)
```

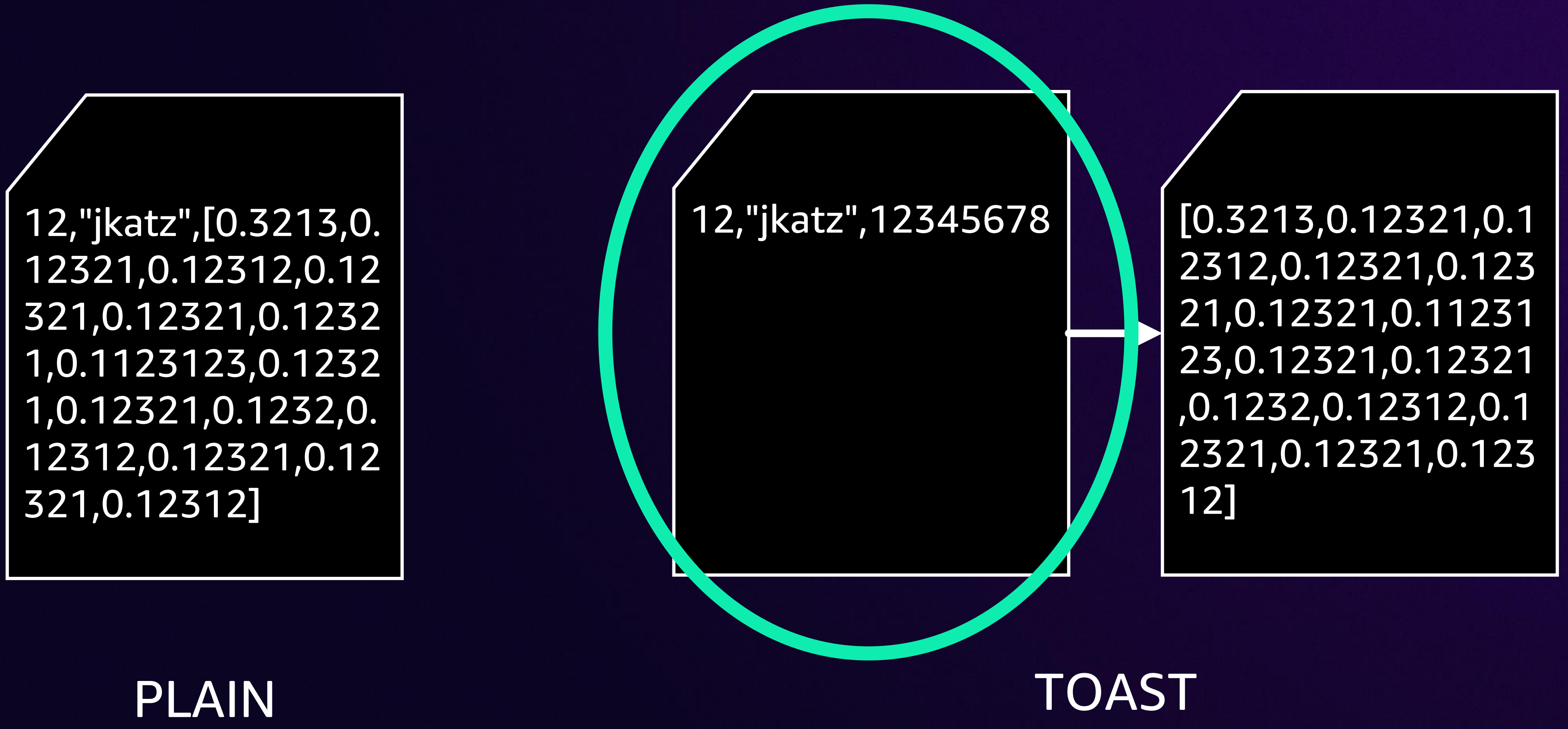
128 dimensions

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

Parallel worker underestimation



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`

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

Best practices: HNSW build and storage parameters

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

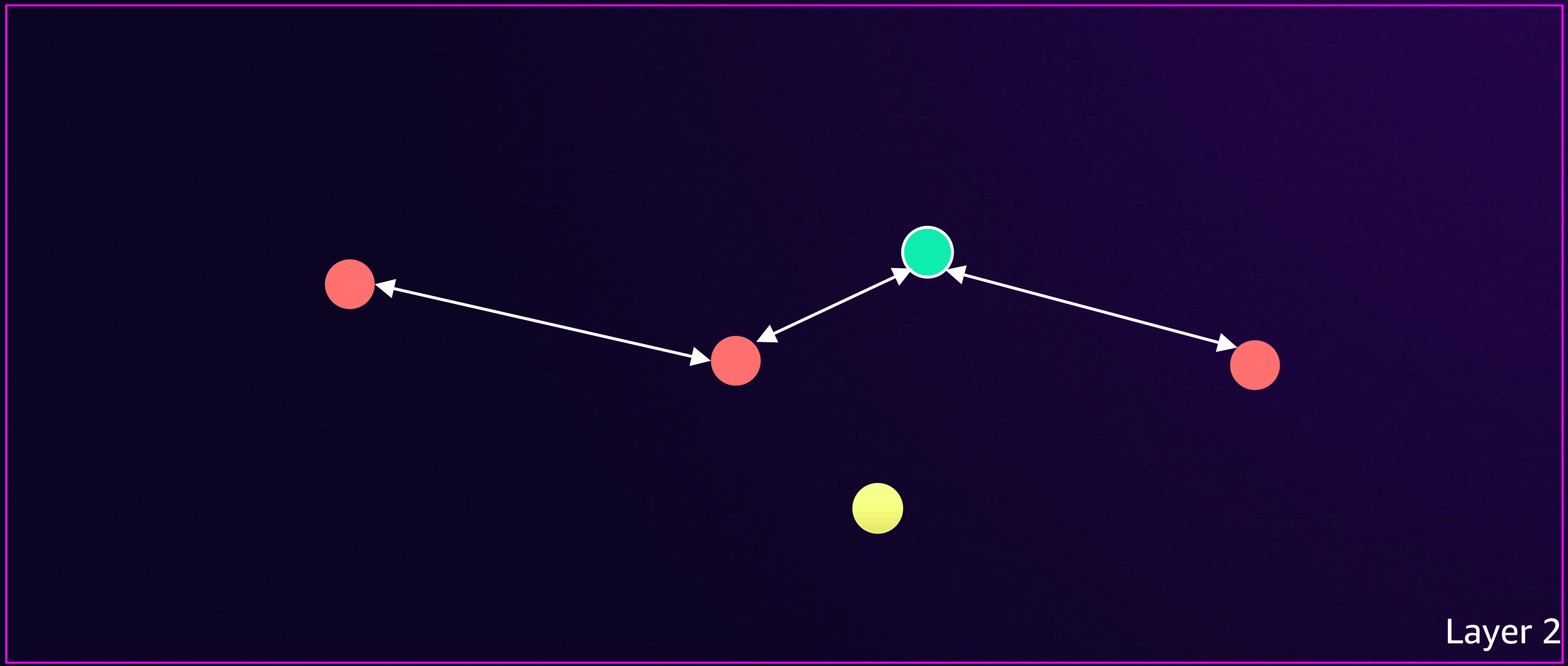
Recommendation: 64 or 256*

HNSW query parameters

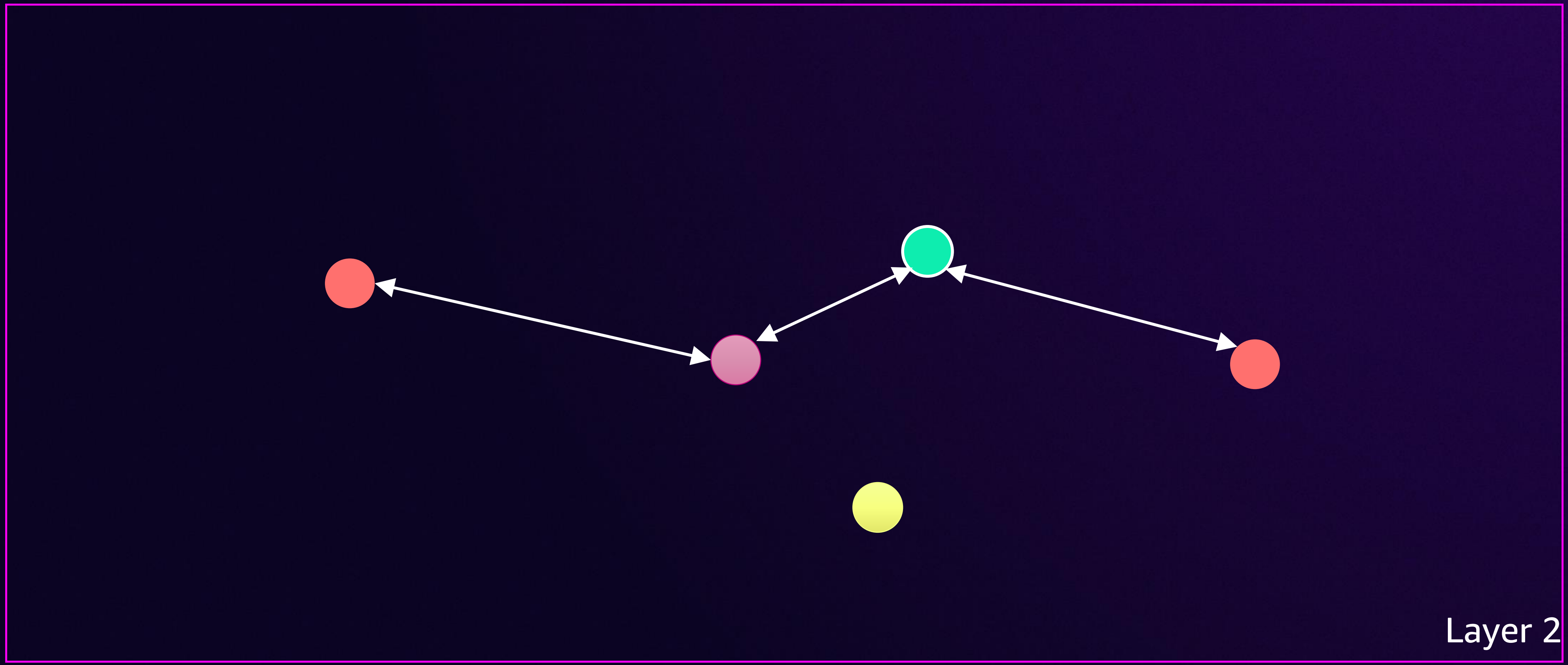
`hnsw.ef_search`

- Number of vectors to maintain in “nearest neighbor” list
- Before v0.8, must be greater than or equal to `LIMIT`
- v0.8+, can use `hnsw.iterative_search` to satisfy unmet `LIMIT`

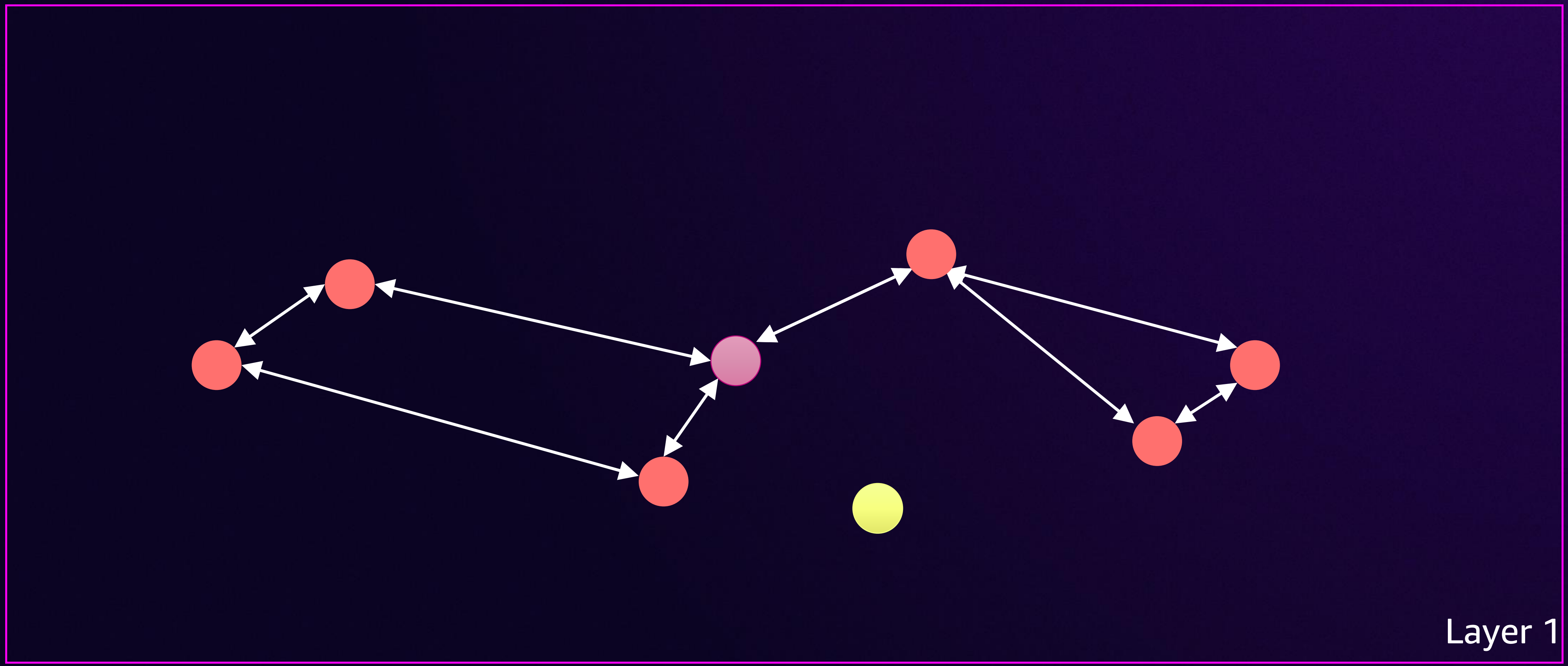
Querying an HNSW index



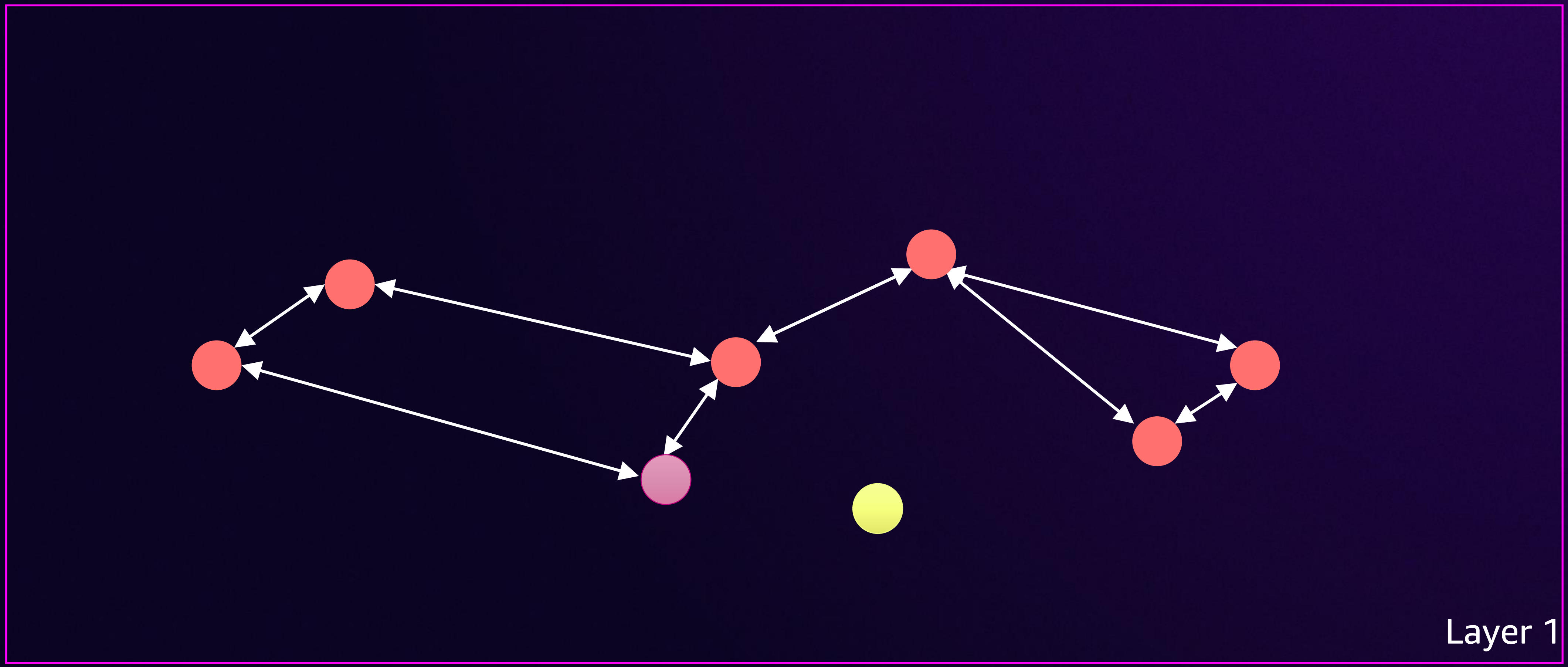
Querying an HNSW index



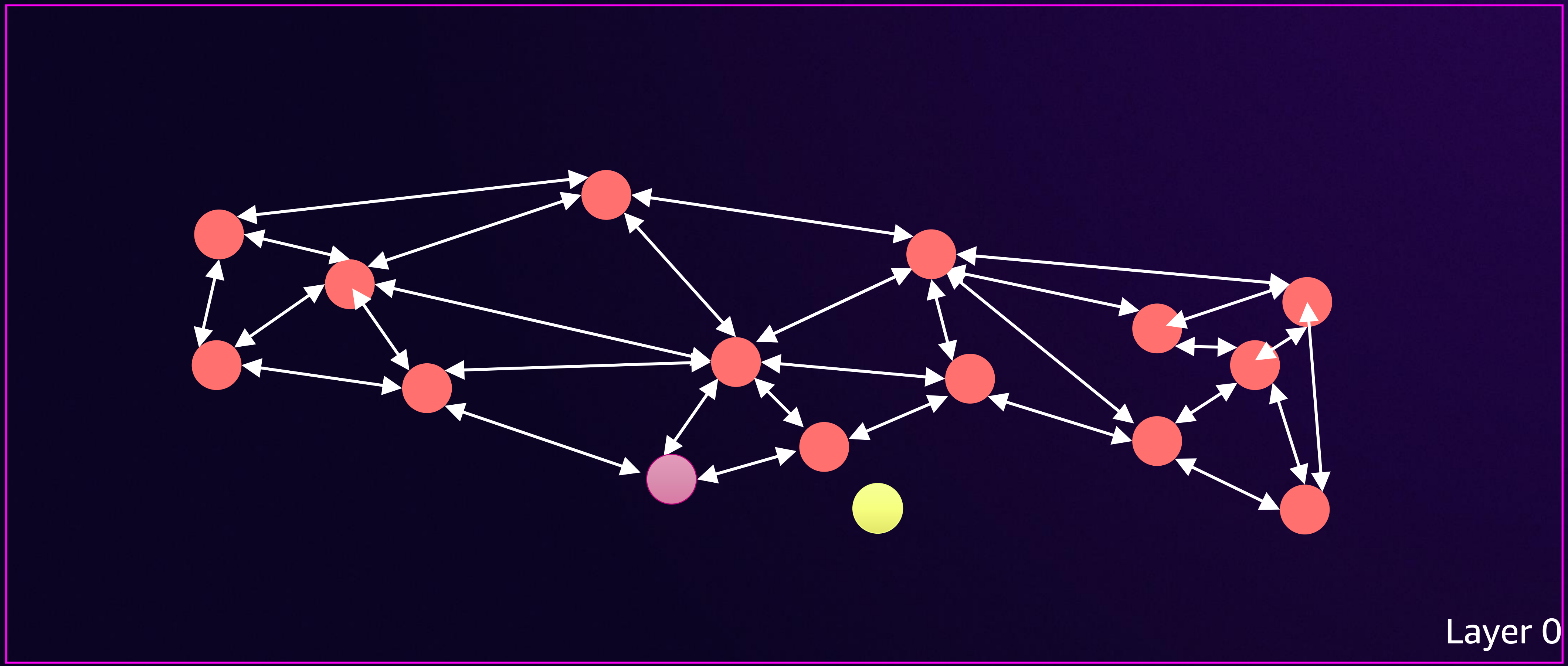
Querying an HNSW index



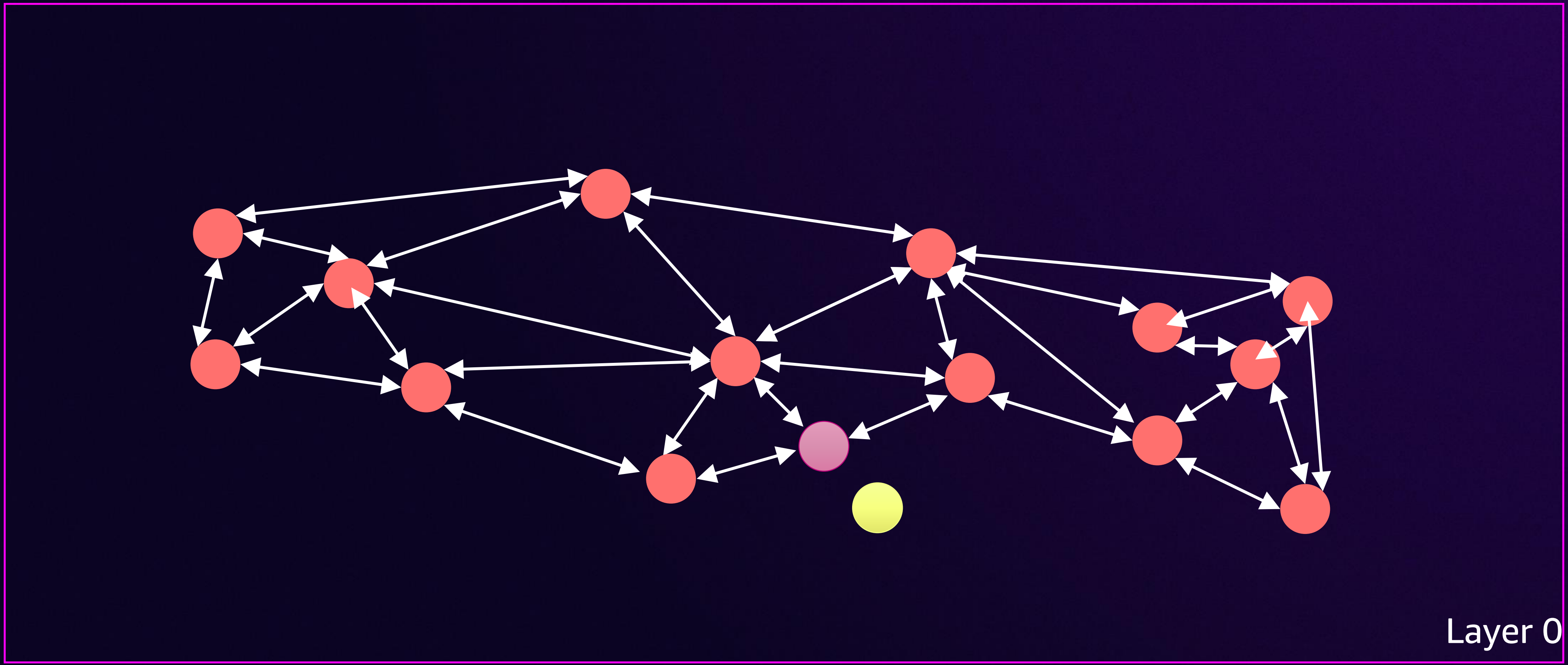
Querying an HNSW index



Querying an HNSW index



Querying an HNSW index

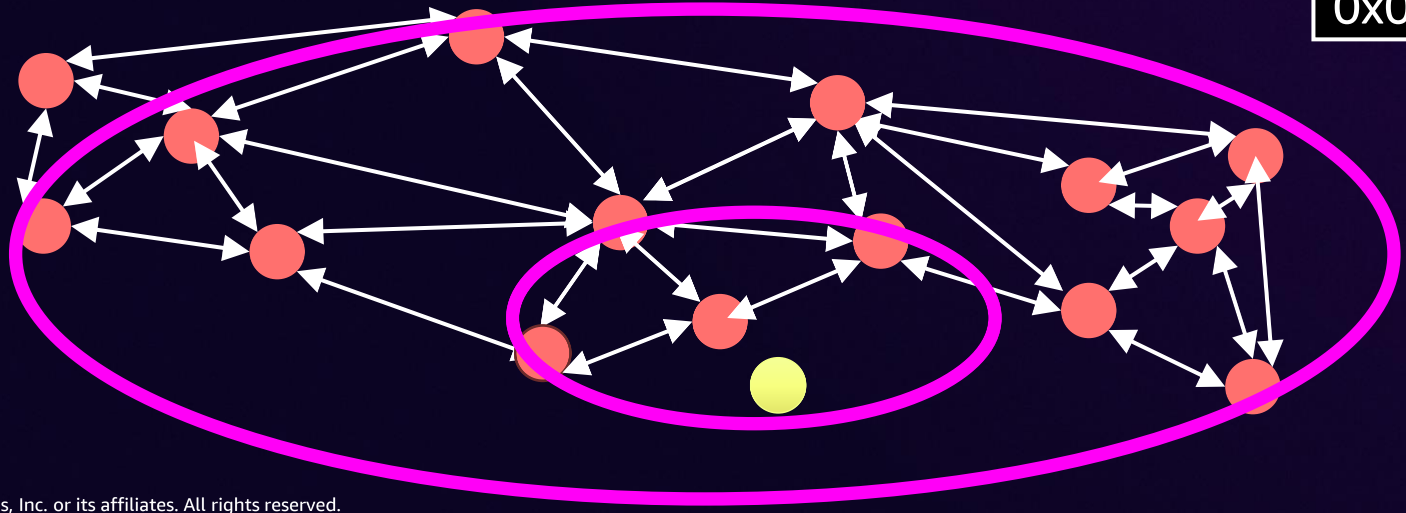


What happens internally searching a HNSW index?

- Maintain a list of visited
- Maintain an ordered list of candidates with distances
- `ef_search` is 1 at Layer 1+
- `ef_search` is `ef_search` (default 40) at Layer 0

Visited
0x0102030405060708
0x0102030405060709
0x0102030405060710

Candidates	
0x0102030405060708	0.0123
0x0102030405060709	0.0434
0x0102030405060710	0.0845



HNSW entry level distribution

m	Layer 1 Entry Level	Layer 0 Entry Level
2	25%	50%
4	19%	75%
8	11%	87%
12	8%	92%
16	6%	94%
20	5%	95%
24	4%	96%
32	3%	97%
36	3%	97%
48	2%	98%
64	2%	98%

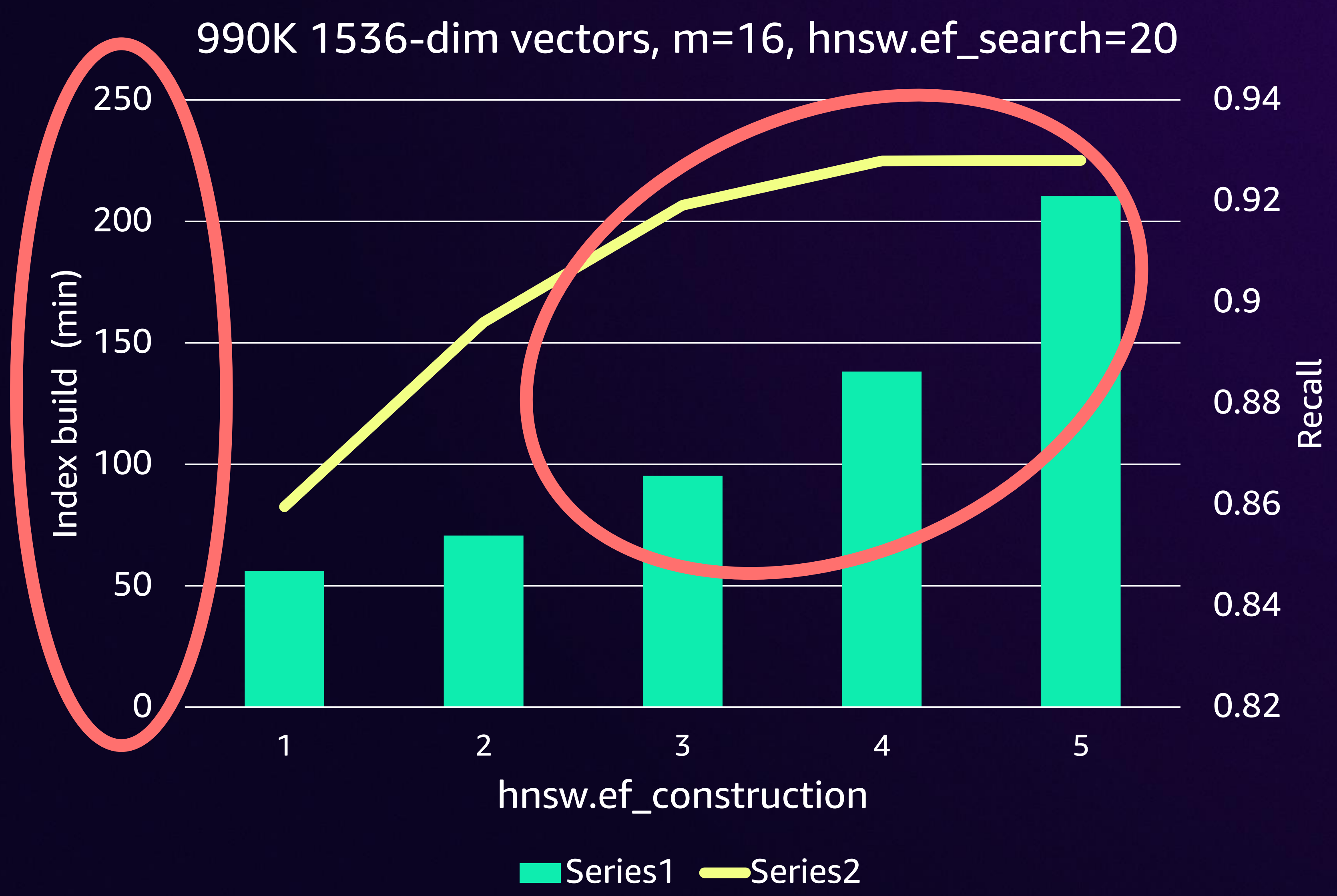
How “m” impacts query time via vectors compared

m=16, ef_construction=64 **# vectors compared**					
ef	SIFT (N=1M)	GIST (N=1M)	GLoVE25 (N=1.1M)	1536d (N=5M)	768d (N=10M)
10	427	512	438	456	498
20	643	779	652	650	695
40	1044	1272	1049	1005	1050
80	1774	2212	1761	1629	1762
120	2438	3099	2420	2214	2449
200	3638	4755	3629	3328	3833
400	6247	8402	6303	5836	7190
800	10619	14706	10938	10563	13258

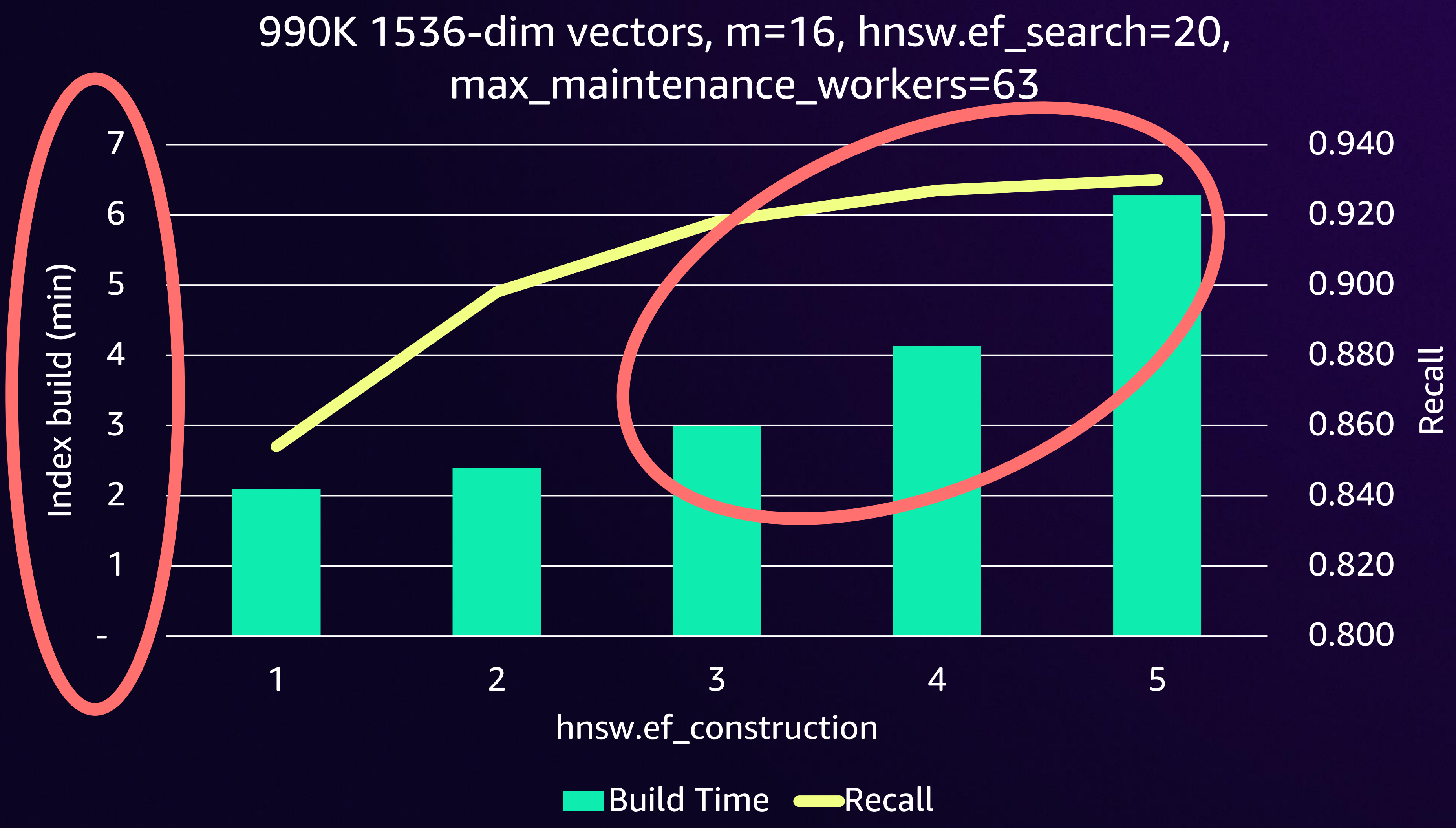
How “m” impacts query time via vectors compared

ef	1536 (N=5M,m=16)	1536d (N=5M,m=64)		768d (N=10M,m=16)	768d (N=10M,m=64)
10	456	605		498	1425
20	650	1257		695	2038
40	1005	2292		1050	3246
80	1629	4049		1762	5691
120	2214	5728		2449	8046
200	3328	8601		3833	12664
400	5836	15158		7190	23284
800	10563	27249		13258	42200

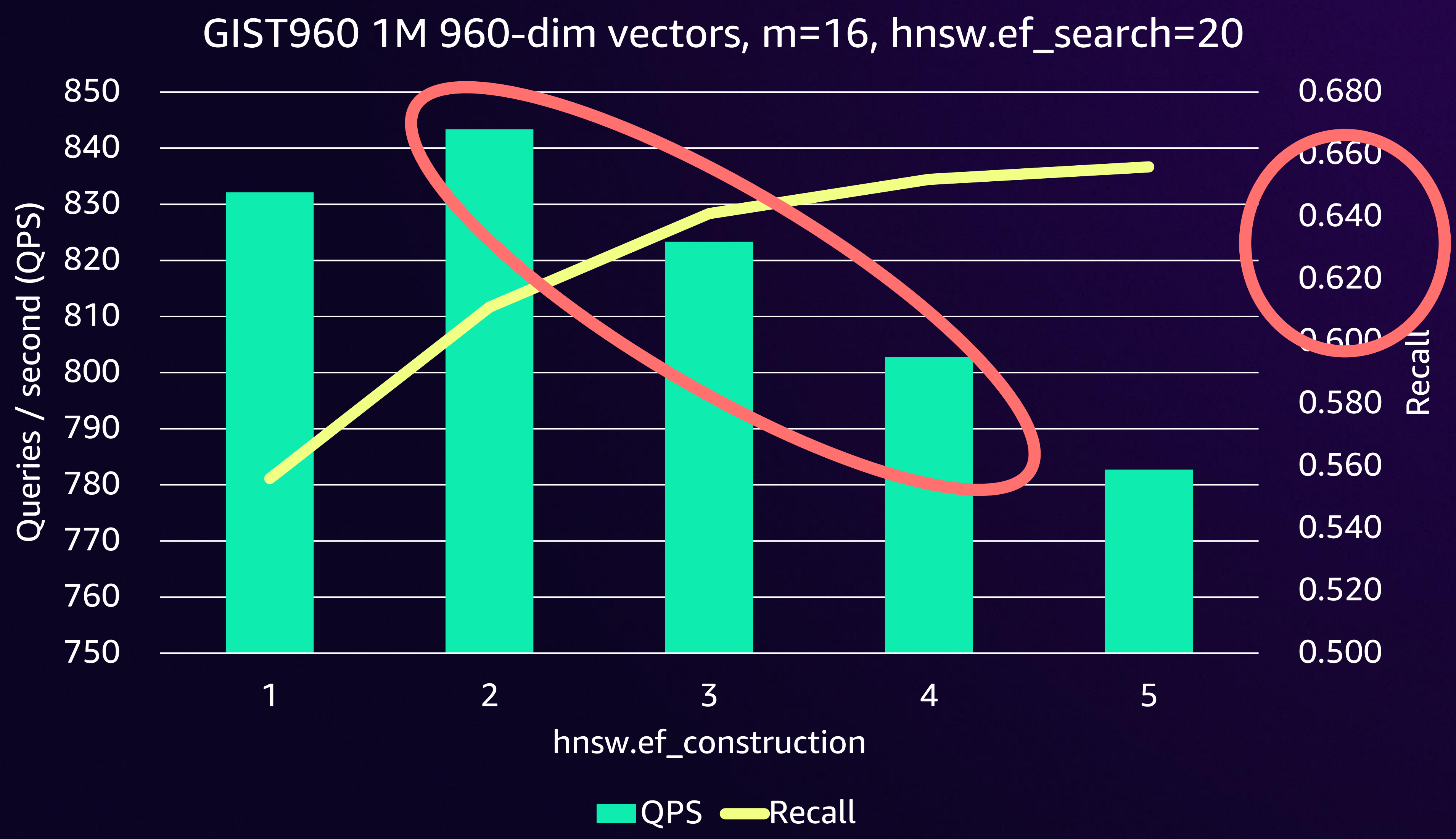
Why index build speed matters (serial build)



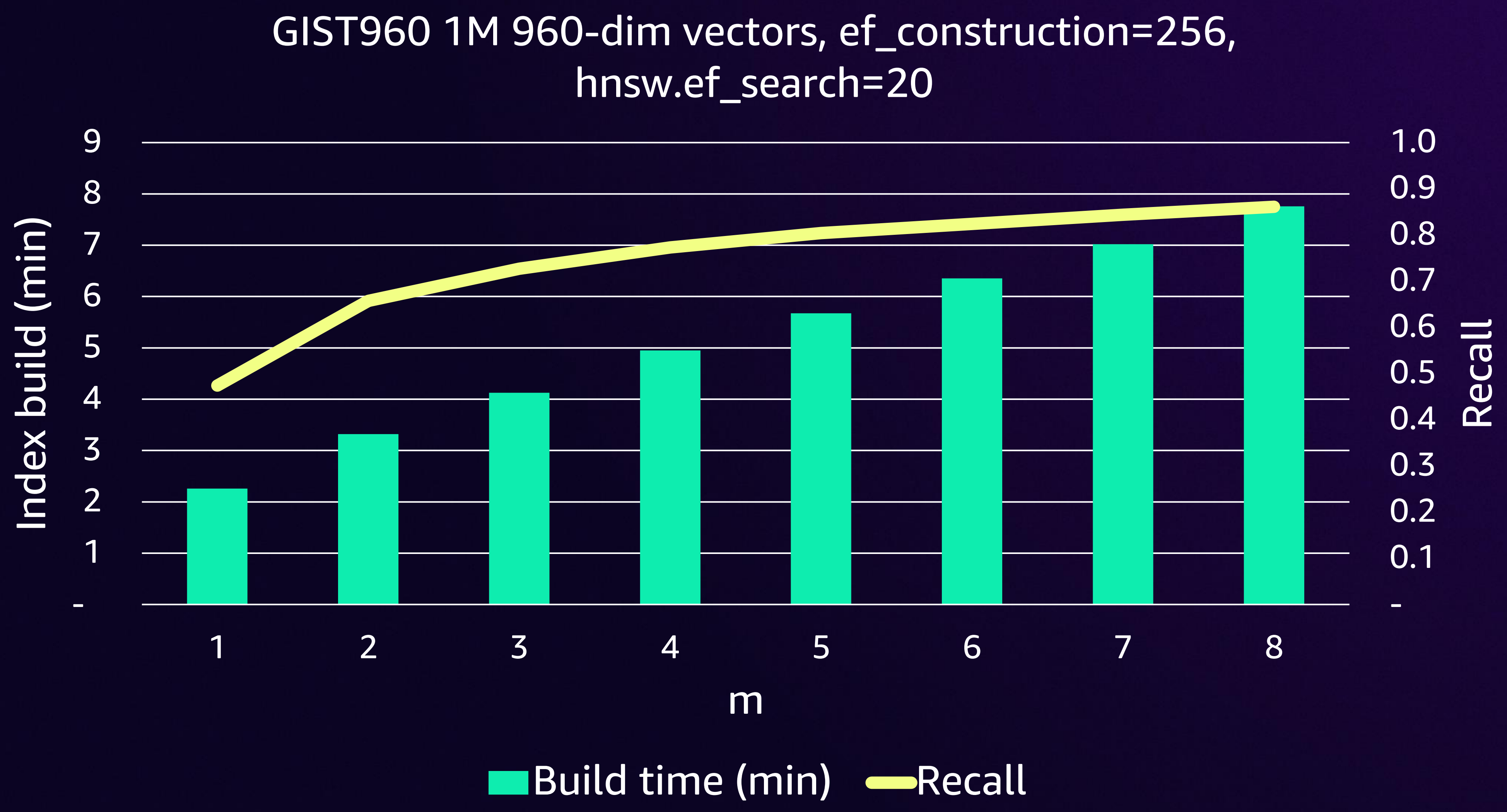
Why index build speed matters (parallel build)



How hnsw.ef_construction impacts query performance



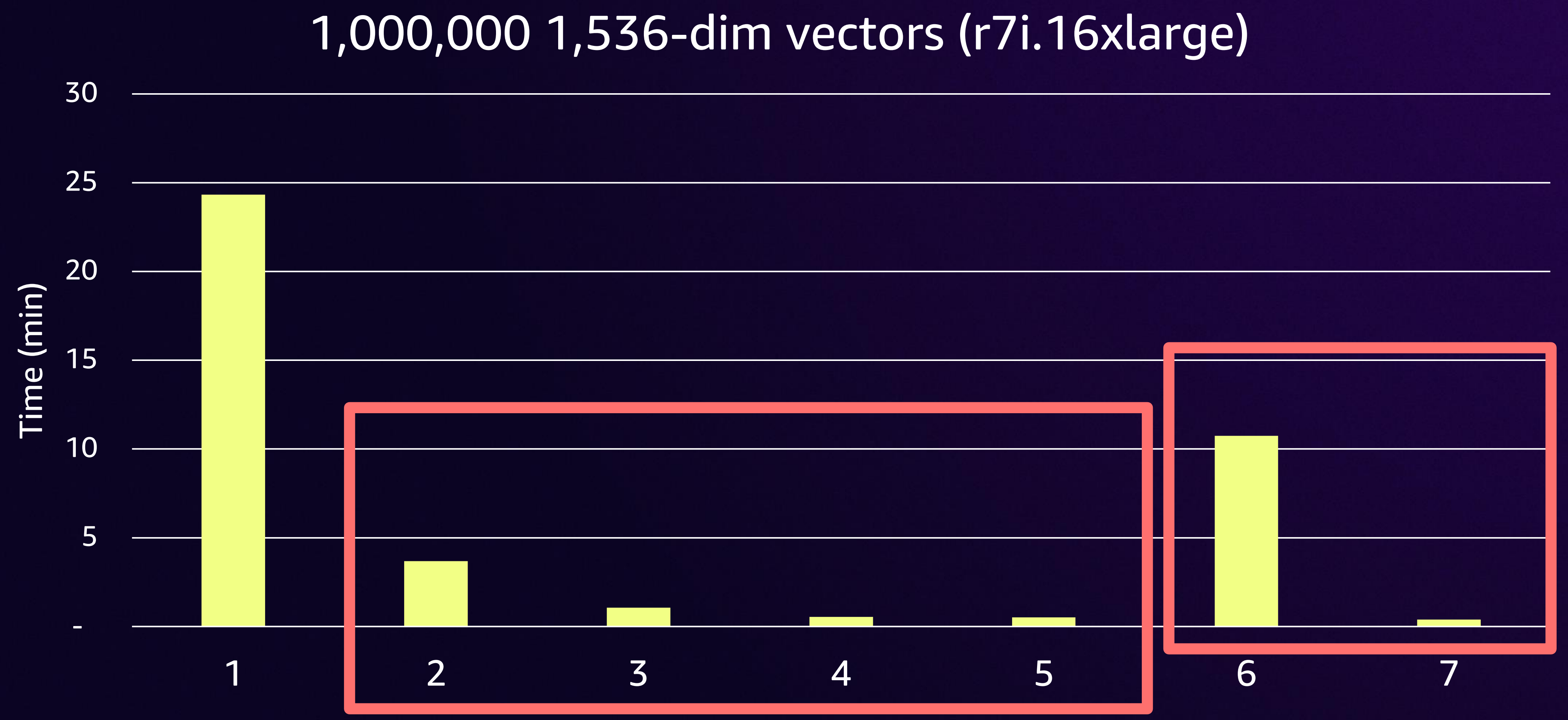
How “m” impacts index build time & search quality



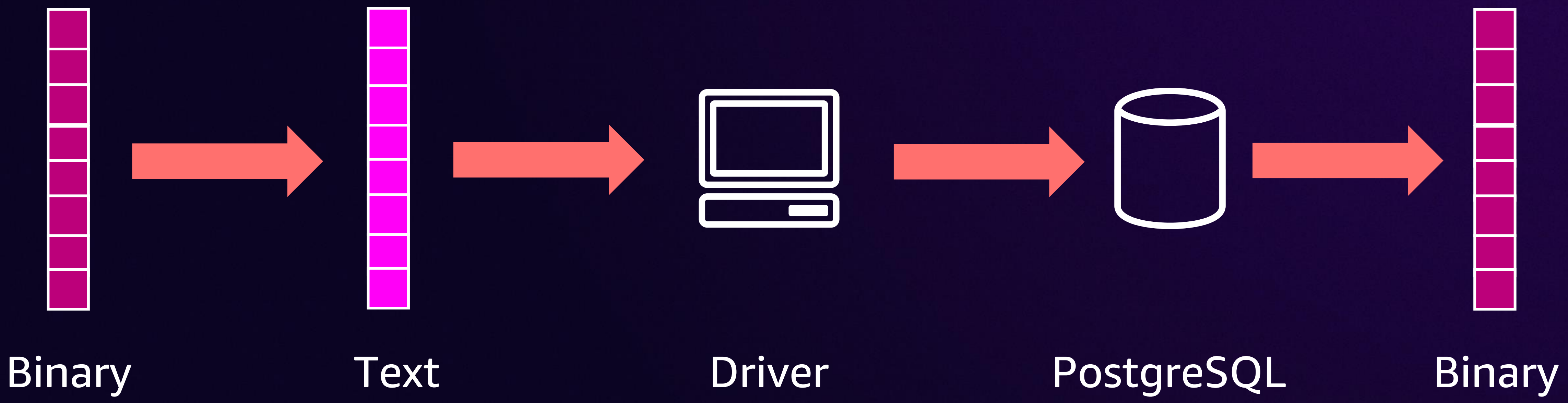
Choosing a data ingestion method

- How you're ingesting vectors dictates methodology
  - Bulk load vectors, then build index
  - Iteratively add vectors to index
- Loading technique impacts write performance
  - Single vs. bulk inserts
  - `INSERT` vs. `COPY` vs. `COPY BINARY`
- Parallel index builds vs. concurrent inserts

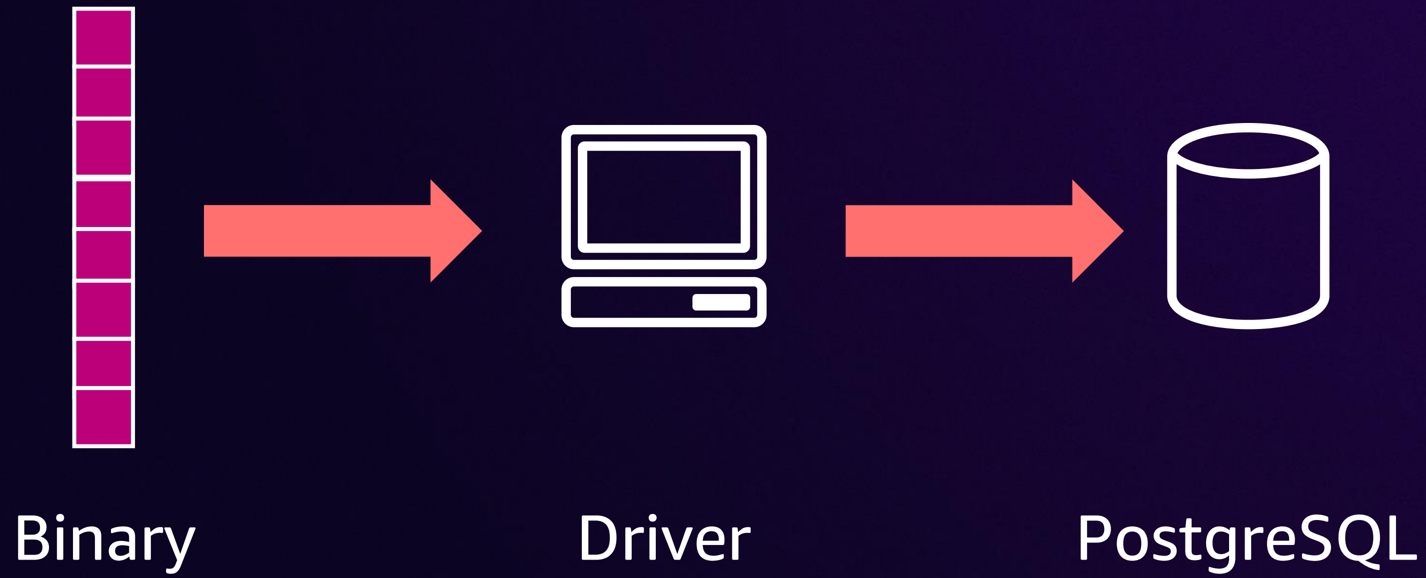
Comparison of ingestion methods – no index



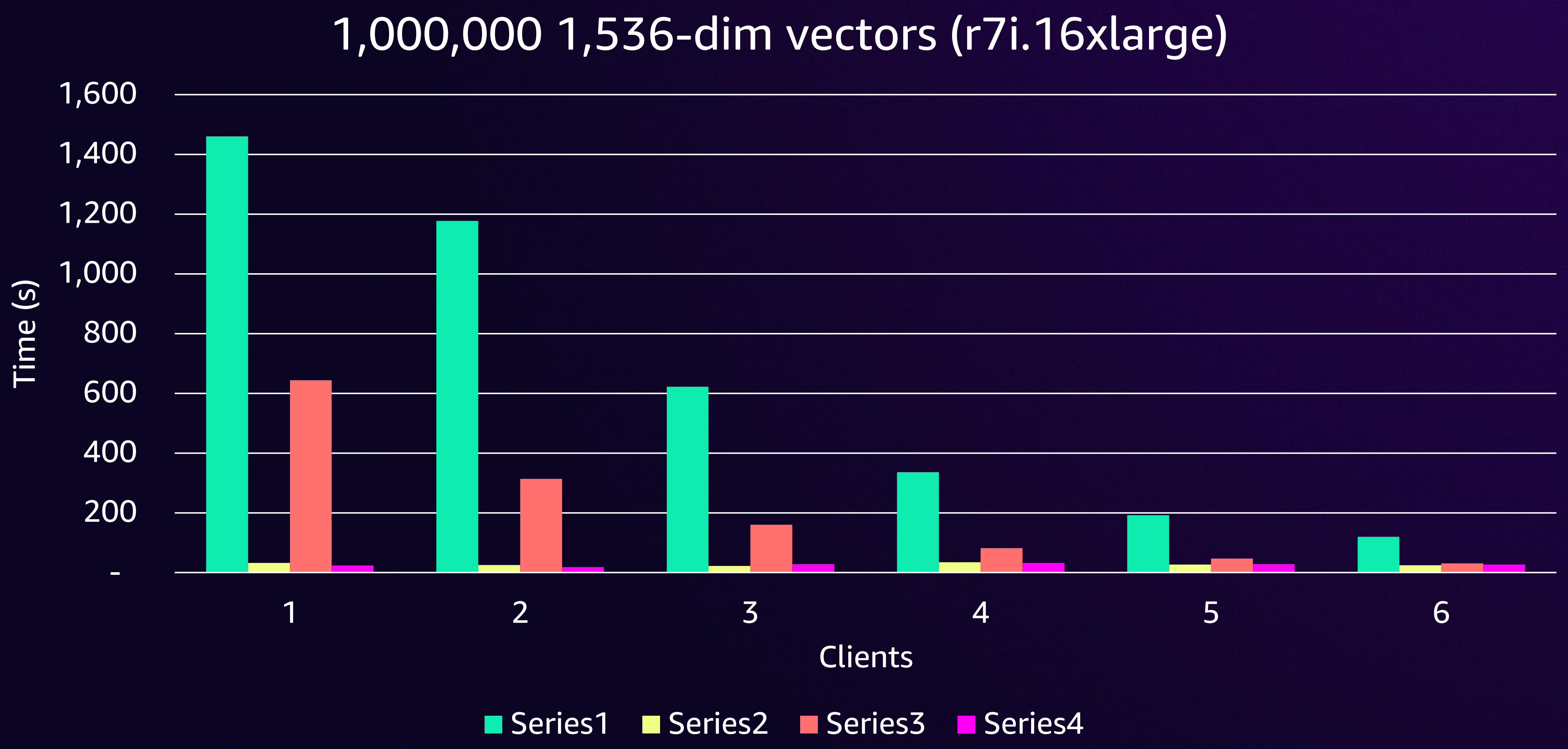
Ingestion with COPY



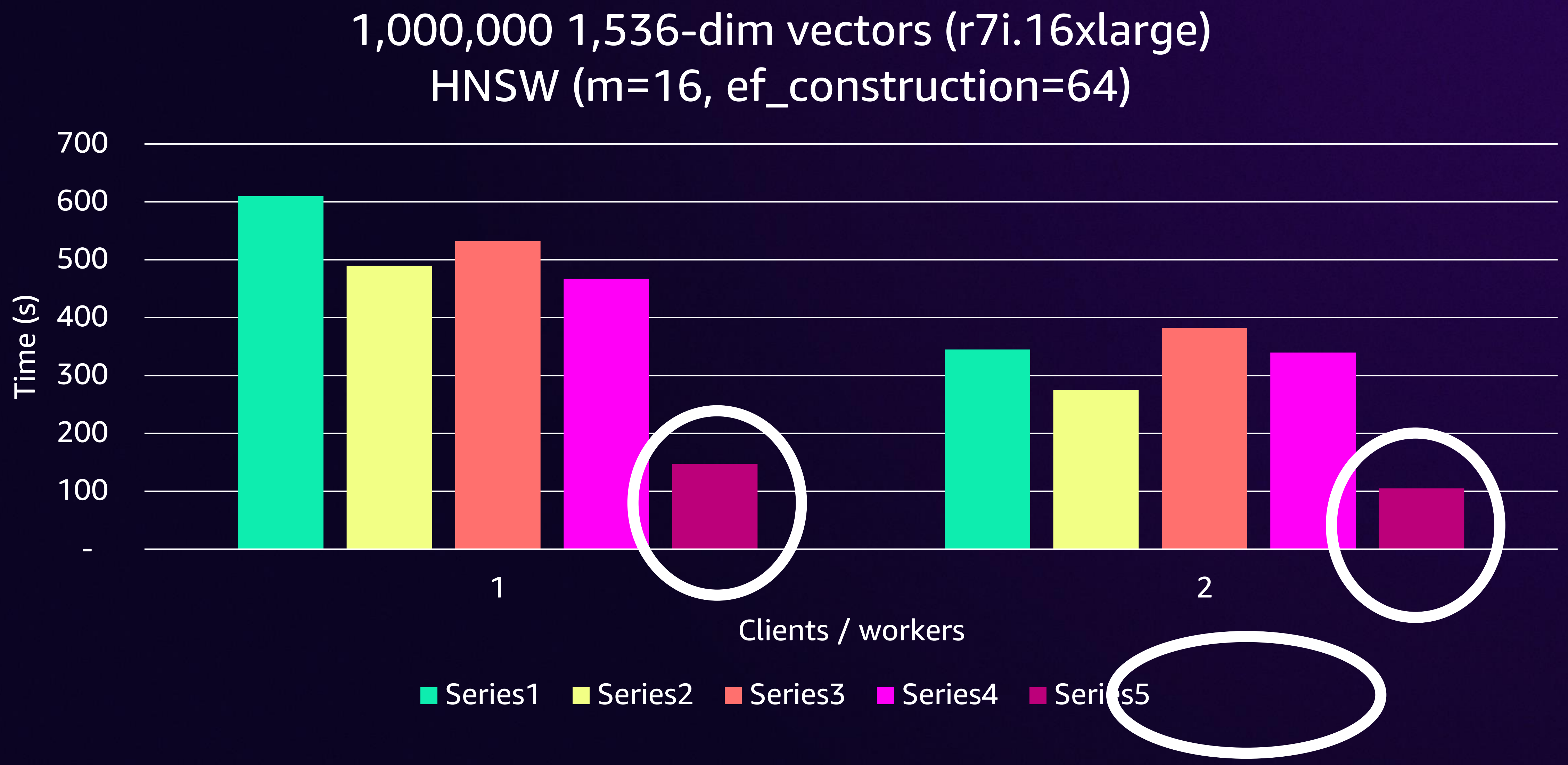
Ingestion with COPY BINARY



Vector ingestion and concurrency – no index



Parallel index build vs. concurrent inserts



Best practices for building HNSW indexes

Start with the defaults (`m=16`, `ef_construction=64`)

- Better recall: `ef_construction` up to `256`

Use `PLAIN` storage to maximize performance

Ingestion

- Full table: parallel build (`max_parallel_maintenance_workers`)
- Iterative inserts: Bulk `INSERT` / `COPY BINARY`

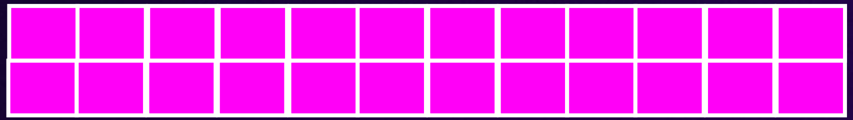
Quantization can help reduce storage

Best practices: Quantization

What is quantization?

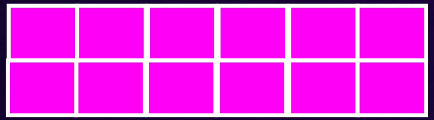
Flat

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



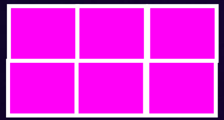
Scalar quantization (2-byte float)

[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;
```

pgvector and binary quantization

```
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 40 -- set to hnsw.ef_search
) i
ORDER BY i.distance
LIMIT 10;
```

1536d 5MM (r7i.16xlarge, m=16, ef_construction=256)			
	Flat	2-byte float	Binary (rerank)
Index Size (GB)	38.15	19.07	2.34
Index build time (min)	21	13	4
Recall @ ef_search = 40	0.931	0.929	0.811
QPS @ ef_search = 40	24,216	27,084	33,984
Recall @ ef_search = 80	0.965	0.961	0.900
QPS @ ef_search = 80	11,057	12,759	20,410
Recall @ ef_search = 220	0.989	0.983	0.963
QPS @ ef_search = 220	5,242	5,983	7,856

1536d 5MM (r7i.16xlarge, m=16, ef_construction=256)			
	Flat	2-byte float	Binary (rerank)
Index Size (GB)	38.15	19.07	2.34
Index build time (min)	21	13	4
Recall @ ef_search = 40	0.931	0.929	0.811
QPS @ ef_search = 40	24,216	27,084	33,984
Recall @ ef_search = 80	0.965	0.961	0.900
QPS @ ef_search = 80	11,057	12,759	20,410
Recall @ ef_search = 220	0.989	0.983	0.963
QPS @ ef_search = 220	5,242	5,983	7,856

1536d 5MM (r7i.16xlarge, m=16, ef_construction=256)			
	Flat	2-byte float	Binary (rerank)
Index Size (GB)	38.15	19.07	2.34
Index build time (min)	21	13	4
Recall @ ef_search = 40	0.931	0.929	0.811
QPS @ ef_search = 40	24,216	27,084	33,984
Recall @ ef_search = 80	0.965	0.961	0.900
QPS @ ef_search = 80	11,057	12,759	20,410
Recall @ ef_search = 220	0.989	0.983	0.963
QPS @ ef_search = 220	5,242	5,983	7,856

1536d 5MM (r7i.16xlarge, m=16, ef_construction=256)			
	Flat	2-byte float	Binary (rerank)
Index Size (GB)	38.15	19.07	2.34
Index build time (min)	21	13	4
Recall @ ef_search = 40	0.931	0.929	0.811
QPS @ ef_search = 40	24,216	27,084	33,984
Recall @ ef_search = 80	0.965	0.961	0.900
QPS @ ef_search = 80	11,057	12,759	20,410
Recall @ ef_search = 220	0.989	0.983	0.963
QPS @ ef_search = 220	5,242	5,983	7,856

Best practices for quantization

- Quantizing can reduce space, but may lose information
  - Consult your data science team if this is an acceptable tradeoff
- Binary quantization is best for vectors with many bits (“bit diversity”)
- Recall decreases as you store more vectors

Best practices: Filtering

What is filtering?

```
SELECT id
FROM products
WHERE products.category_id = 7
ORDER BY $1 <=> products.embedding
LIMIT 10;
```

How filtering impacts ANN queries

- PostgreSQL may choose not to use an ANN index
- PostgreSQL uses an ANN index, but doesn't return enough results
- Filtering occurs after using the index

Considerations for filtering strategy

- **Query patterns**: Distribution of filtered vs. unfiltered queries
- **Selectivity**: how many rows do your filters remove?
  - High selectivity: removes "most" rows
- **# of vector distance comparisons per query**

Remember: Speed of 5,000 distance operations

5,000 cosine distance operations (<=>) – single connection (r7i.16xlarge)				
	PLAIN		EXTERNAL	
Dimensions	p50 (ms)	QPS	p50 (ms)	QPS
128	1.2	863	1.2	814
256	1.5	655	1.5	658
384	1.7	591	1.7	587
512	2.0	491	9.8	102
768	2.4	410	10.7	93
1,024	3.5	288	12.0	83
1,536	4.1	246	16.2	62

Pre-v0.8.0 filtering strategies

Partial indexing

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

Partitioning

```
CREATE TABLE docs_cat7
  PARTITION OF docs
  FOR VALUES IN (7);

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

pgvector v0.8.0+ changes for filtering

- HNSW cost estimation provide more options for query planner
  - Alternative index selection / sequential scan

- Iterative scans: keep scanning index `LIMIT` satisfied / `hnsw.max_scan_tuples` reached
  - Helps "overfiltering" problem
  - `hnsw.iterative_scan`:
    - `relaxed_order` (better recall)
    - `strict_order` (no reordering required)
    - `off` (default)
  - `hnsw.max_scan_tuples` (default: 20,000)

Example: Index strategy when using filters

r7i.16xlarge, 5,000,000 512-dim vectors, k=10, single connection

```
CREATE TABLE embeddings (
  filter_10 int,
  filter_1 int,
  filter_01 int,
  filter_001 int,
  embedding vector (512)
);
CREATE INDEX ON embeddings USING
  hnsw (embedding vector_cosine_ops);
```

No iterative scan (ef_search=40)		
Selectivity	QPS	Rows returned (avg)
10%	485	3.98
1%	486	0.40
0.1%	485	0.04
0.01%	451	0.00

Example: Index strategy when using filters

r7i.16xlarge, 5,000,000 512-dim vectors, k=10, single connection

	No iterative scan (ef_search=40)			No iterative scan (ef_search=1000)	
Selectivity	QPS	Rows returned (avg)	Selectivity	QPS	Rows returned (avg)
10%	485	3.98	10%	27	10.00
1%	486	0.40	1%	26	8.77
0.1%	485	0.04	0.1%	25	1.00
0.01%	451	0.00	0.01%	25	0.10

```
SELECT $1 <=> embedding AS distance FROM embeddings
WHERE filter_10 = $2
ORDER BY distance LIMIT 10
```

aws

Example: Index strategy when using filters

r7i.16xlarge, 5,000,000 512-dim vectors, k=10, single connection

	No iterative scan (ef_search=1000)		Iterative scan (ef_search=40, iterative_scan=relaxed_order, max_tuples_scanned=20000)	
Selectivity	QPS	Rows returned (avg)	QPS	Rows returned (avg)
10%	27	10.00	196	10.00
1%	26	8.77	37	10.00
0.1%	25	1.00	26	9.99
0.01%	25	0.10	16	2.06

```
SET hnsw.iterative_scan TO relaxed_order;
SET hnsw.max_tuples_scanned TO 20000;
```

Example: Index strategy when using filters

r7i.16xlarge, 5,000,000 512-dim vectors, k=10, single connection

	Iterative scan (ef_search=40, iterative_scan=relaxed_order, max_tuples_scanned=20000)		B-tree	
Selectivity	QPS	Rows returned (avg)	QPS	Rows returned (avg)
10%	196	10.00	1	10.00
1%	37	10.00	12	10.00
0.1%	26	9.99	125	9.88
0.01%	16	2.06	900	10.00

```
CREATE INDEX ON embeddings (filter_10);
```

Example: Index strategy when using filters

r7i.16xlarge, 5,000,000 512-dim vectors, k=10, single connection

	Iterative scan (relaxed_order)		B-tree (multicolumn)		GIN (JSONB)	
Selectivity	QPS	Rows returned (avg)	QPS	Rows returned (avg)	QPS	Rows returned (avg)
10% + 1%	20	9.99	114	10.00	26	10.00
10% + 0.1%	16	2.08	1,138	10.00	162	10.00

```
CREATE INDEX ON embeddings (filter_10, filter_01);

ALTER TABLE embeddings ADD COLUMN filters;
UPDATE embeddings SET filters = jsonb_build_object('filter_10', filter_10, ...);
CREATE INDEX ON embeddings USING gin(filters jsonb_path_ops);
```

Example: Index strategy when using filters

r7i.16xlarge, 5,000,000 512-dim vectors, k=10, single connection

	Index size (GB)	**x smaller**
HNSW	12.72	-
B-tree	0.03	424x
B-tree (multicolumn)	0.03	424x
GIN	0.07	181x

Best practices for filtering

- When possible, avoid using an ANN index (HNSW / IVFFlat)
  - B-tree: Known, fixed set of filters
  - GIN: Store filters in `JSONB`
  - Other indexes for specialized data types (`GiST`, `BRIN`)
- If mix of selectivity, use both ANN and non-ANN indexes
- Partitioning / partial indexes can segment data with low selectivity

Amazon Aurora features for vector search

Amazon Aurora

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 & scalability

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

Availability & 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

Fully managed

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

Amazon Aurora features for vector workloads

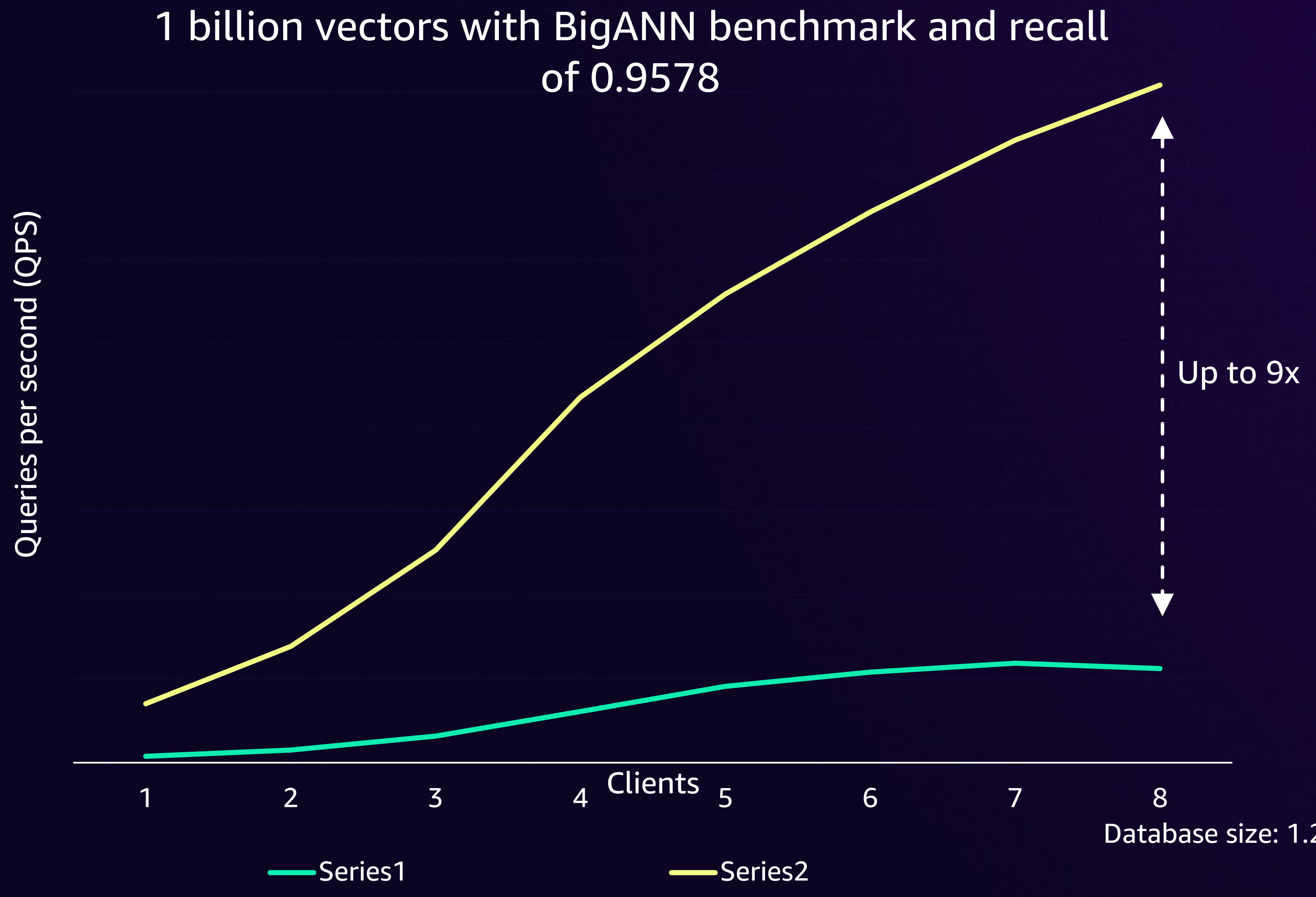
- Aurora PostgreSQL-Compatible with Optimized Reads
  - NVMe caching

- Higher memory instances (r7g / r7i)
- Amazon Aurora PostgreSQL Limitless Database: automated horizontal scaling

- Aurora as an Amazon Bedrock knowledge base
- AuroraML: Generate embeddings directly from Aurora

- Compatibility with frameworks like LangChain and LlamaIndex

Scale further with Aurora Optimized Reads



Amazon Aurora Optimized Reads and pgvector increase 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

Amazon Aurora Limitless Database

Scaling *Managed*

Serverless

Distributed

Single interface

Transactionally **consistent**

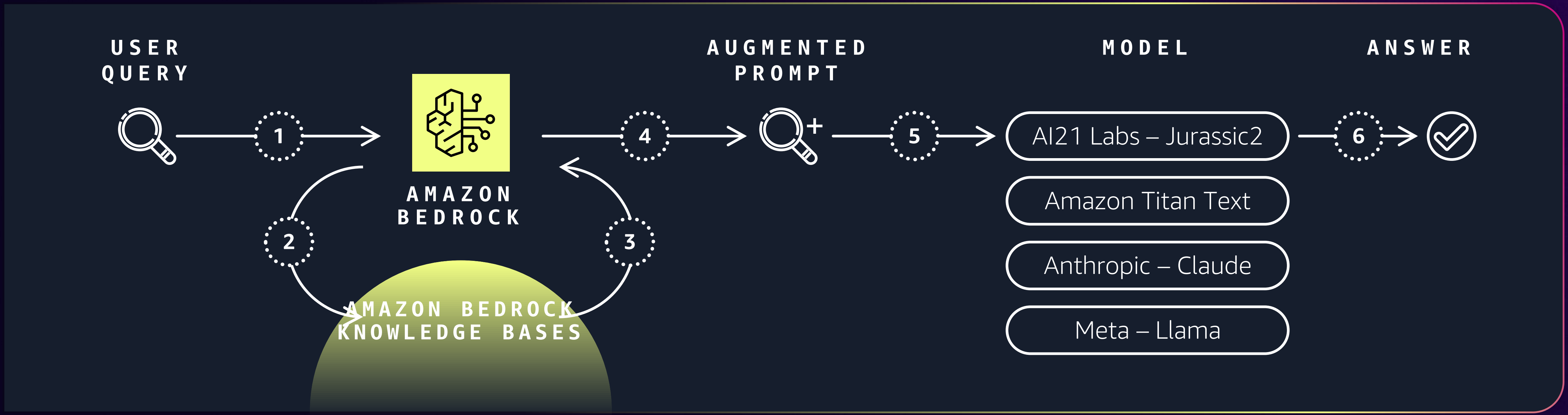
Millions of transactions

Petabytes of data

Amazon Aurora

Amazon Bedrock Knowledge Bases

NATIVE SUPPORT FOR RAG



Securely connect FMs to data sources for RAG to deliver more relevant responses

Fully managed RAG workflow including ingestion, retrieval, and augmentation

Built-in session context management for multiturn conversations

Automatic citations with retrievals to improve transparency

Amazon Bedrock Agents

ENABLE GENERATIVE AI APPLICATIONS TO EXECUTE MULTISTEP TASKS USING COMPANY SYSTEMS AND DATA SOURCES

Decompose into steps using available actions and Amazon Bedrock Knowledge Bases → **Execute action or search knowledge base** → **Until final answer**

Execute action or search knowledge base
↓
Observe results
↓
Think about next step

Looking ahead

pgvector roadmap

- Filtering enhancements, e.g., index-based prefiltering (**in progress**)
- More data types per dimension (fp8, uint8) (**in progress**)
- Streaming I/O
- Additional quantization techniques (statistical)
- Parallel query

Conclusion

- Primary design decision: **query performance** and **recall**
- Determine where to invest: **storage**, **compute**, **indexing strategy**

Amazon Aurora PostgreSQL-Compatible features help you scale your vector workloads

- 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

@jkatz05