PostgresPro

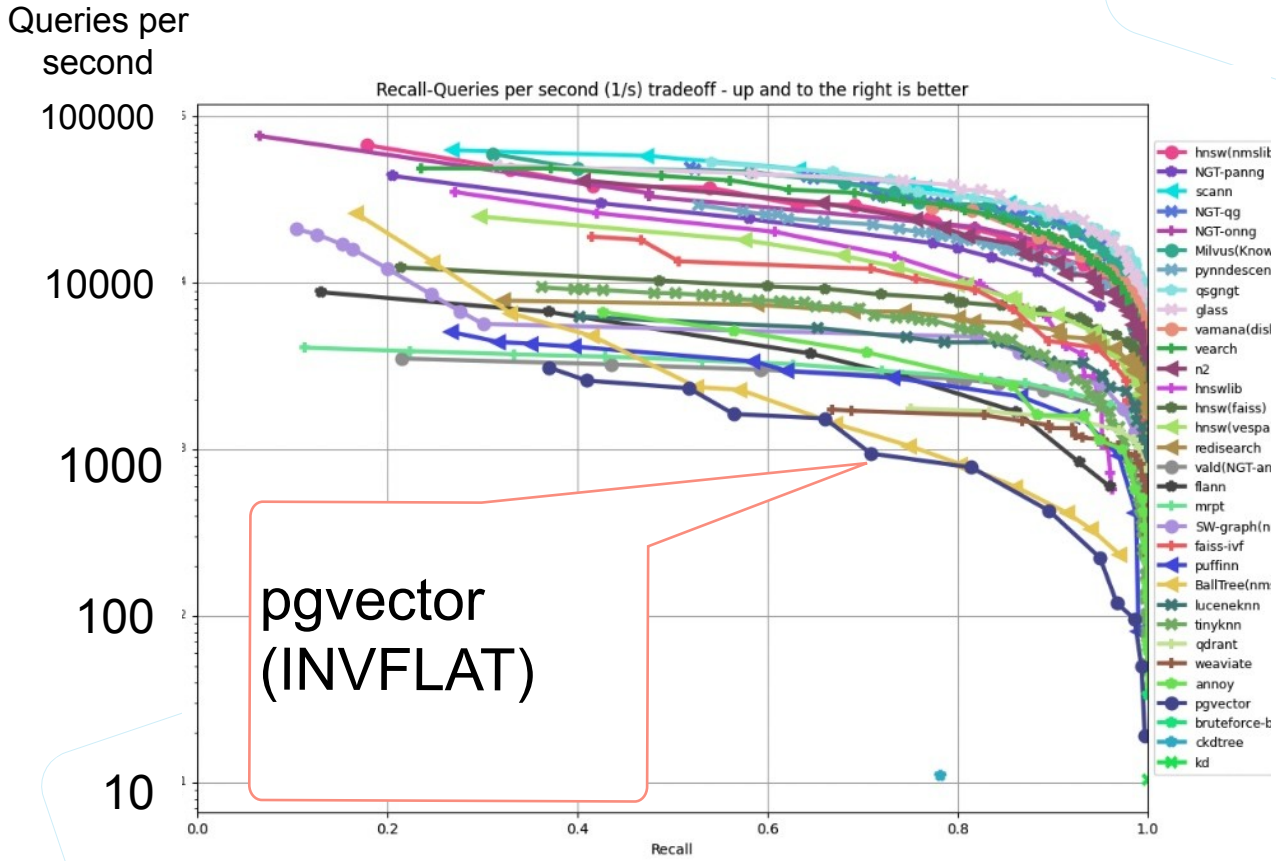
PostgreSQL for AI

Real Vector Search in PostgreSQL

FOSSASIA PGDay 2025

Bangkok, 14 of March 2025

Comparison of Pgvector with Vector Databases in 2023



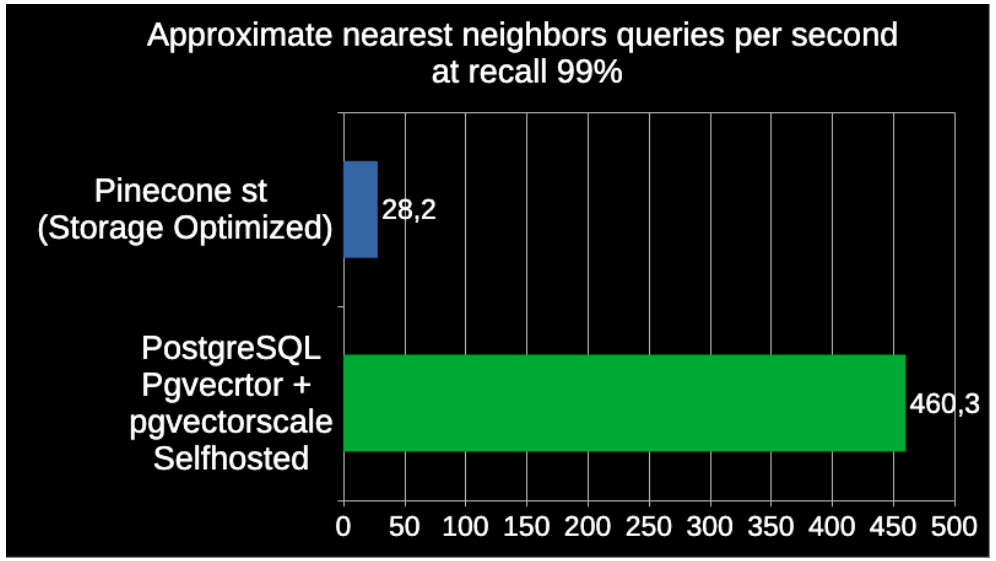
recall 20%

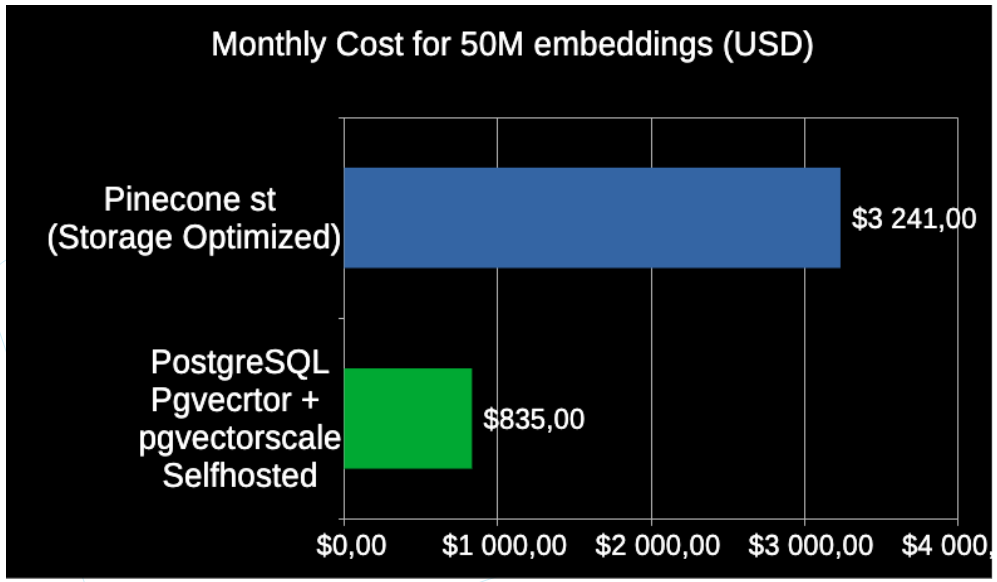
recall 99%

- Old version of pgvector
- Algorithm of pgvector — INVFLAT
- Vector databases are loaded into memory
- Most of vector databases have algorithm HNSW

source https://ann-benchmarks.com/index.html

Comparison of Postgres with Vector Databases in 2024



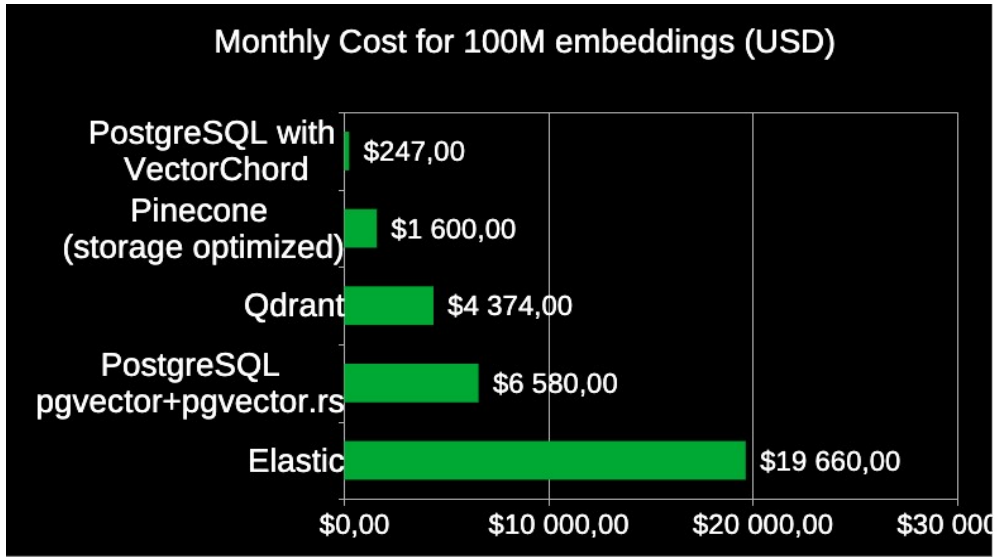


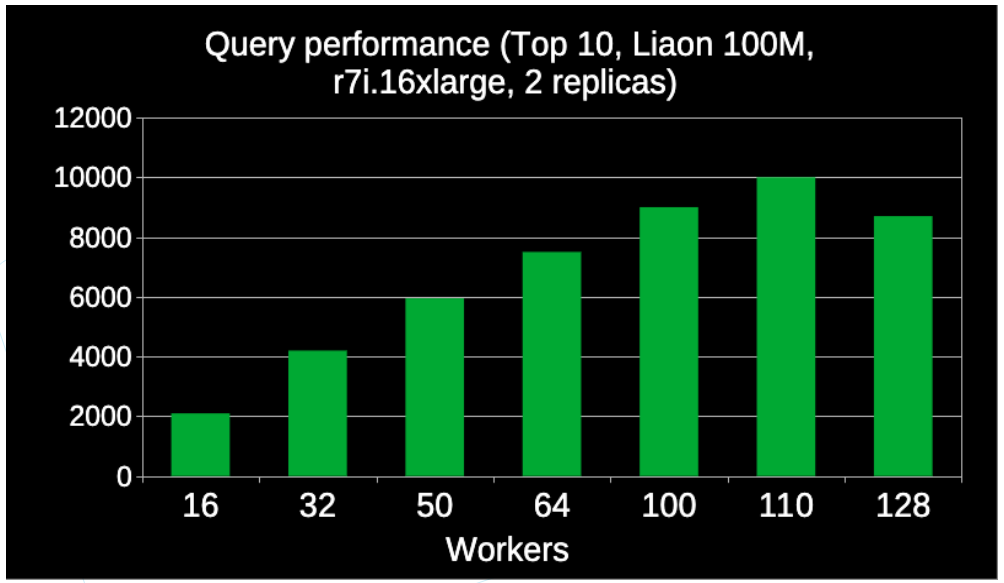
- Timescaledb-benchmark (50 mln vectors, dimension 768) compares **Postgres** (pgvector+pgvectorscale) with **Pinecone**
- Postgres results:

- faster by 16 times
- cost by 75% lower

source https://www.timescale.com/blog/pgvector-vs-pinecone

Comparison of Postgres with Vector Databases in 2025





- VectorChord.ai compares the cost of the querying the dataset LIAON (100 mln vectors, dimensions 768).
- VectorChord.ai demonstrates scalability to reach 10000 QPS using 2 readonly replicas (128 vcpu, 1TB RAM in total)
- Algorithm is based on INV and uses bit quantization

source https://blog.vectorchord.ai/vector-search-at-10000-qps-in-postgresql-with-vectorchord

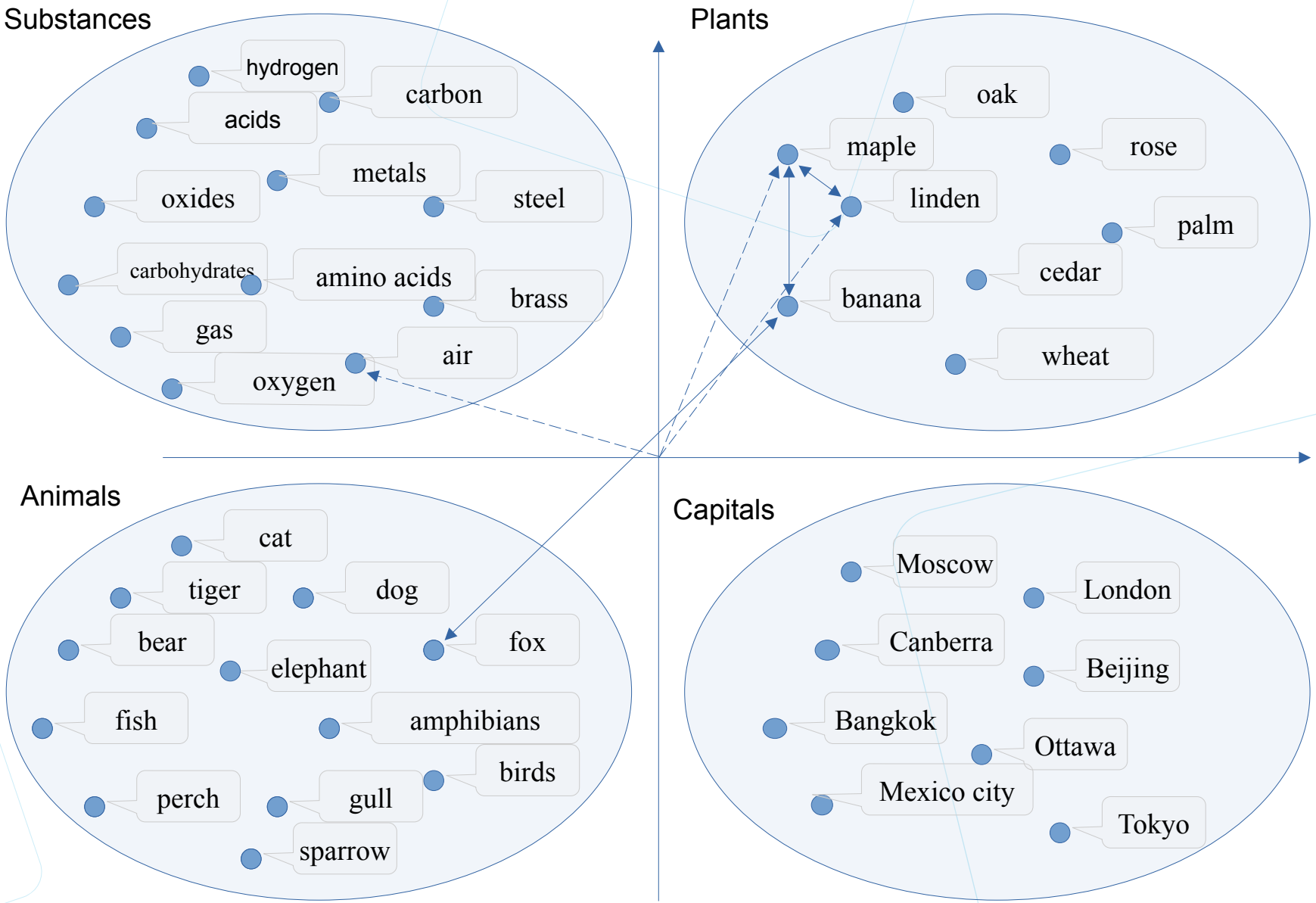
Short Introduction to Vectors

- Growing use with the development of algorithms for converting words to vectors (2010)
  - word2vec https://code.google.com/archive/p/word2vec/
  - GloVe https://github.com/stanfordnlp/GloVe
- Transformers models convert any data to vectors (2017)
  - texts
  - images, video, audio
  - sequences (measurements)
  - anomaly detection (monitoring, fraud detection etc)
  - and more

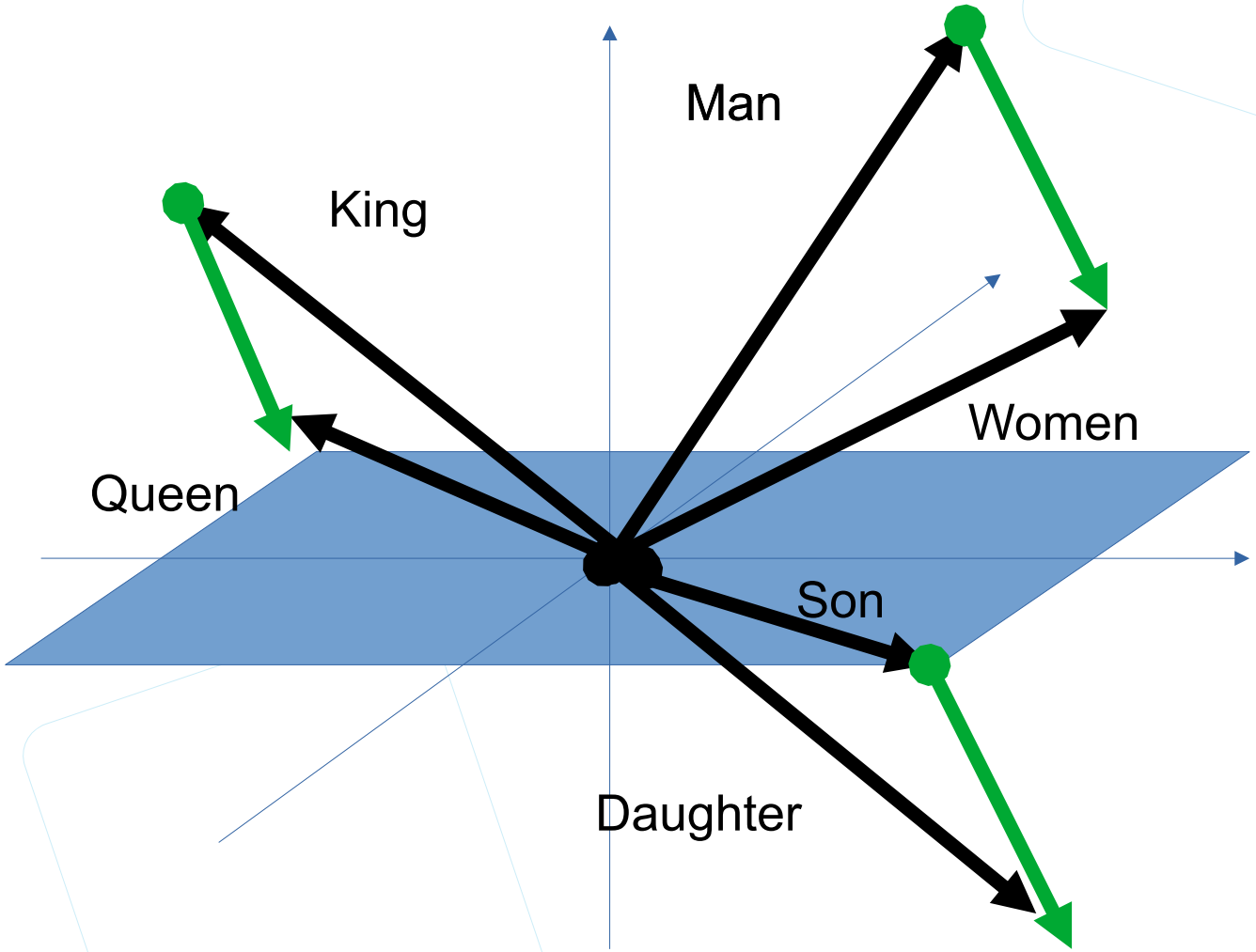
Vectors — Euclidean and Angular Distances

Euclidean Distance – continuous arrows

Angular Distance – dash arrows



Vectors Properties



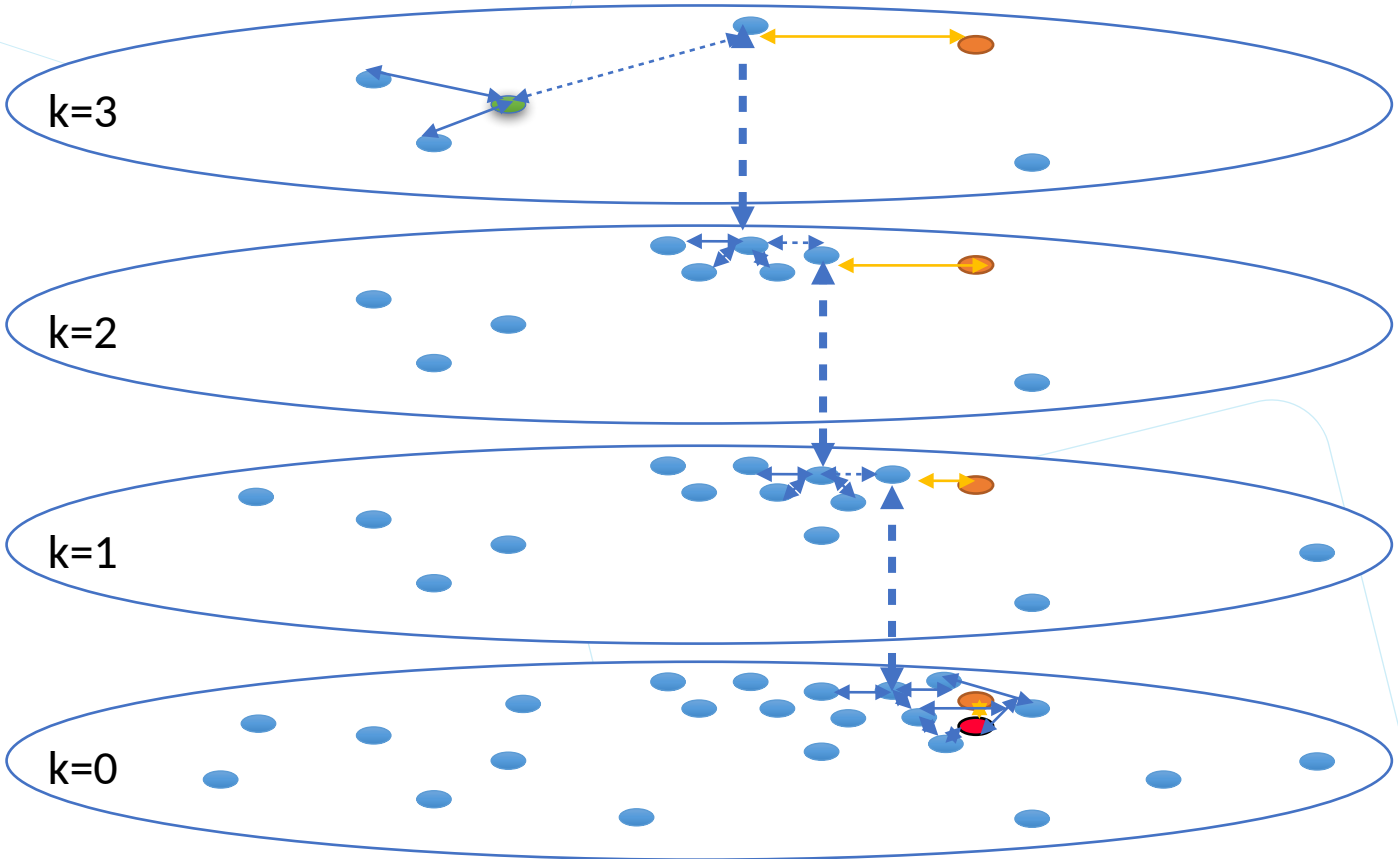
- Vectors behave as geometric vectors (valid add and subtract operations)
- Direction in N-dimension space is «concept» - color, gender etc
- The more dimensions in space, the more concepts represented in vectors

Pgvector — Two Vector Search Algorithms

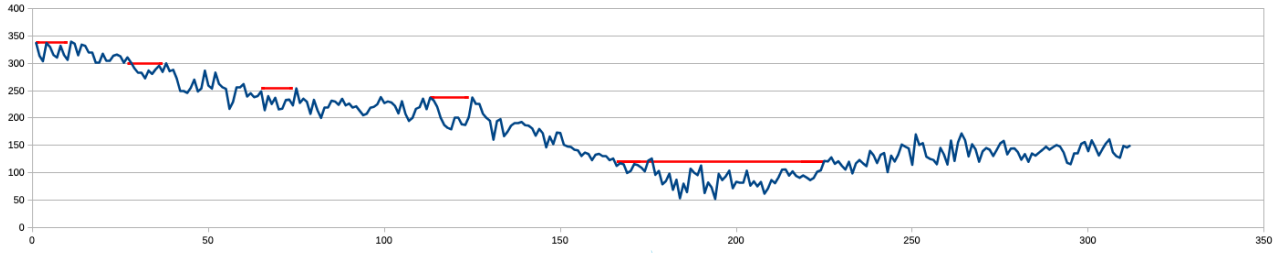
- INVFLAT
  - It is first
  - Actually improved bruteforce
  - Recommended in small range of applications
  - For some applications it is choice #1
- HNSW
  - Universal algorithm
  - High rate
  - Parameters to fit recall (accuracy)

HNSW — Hierarchical Worlds

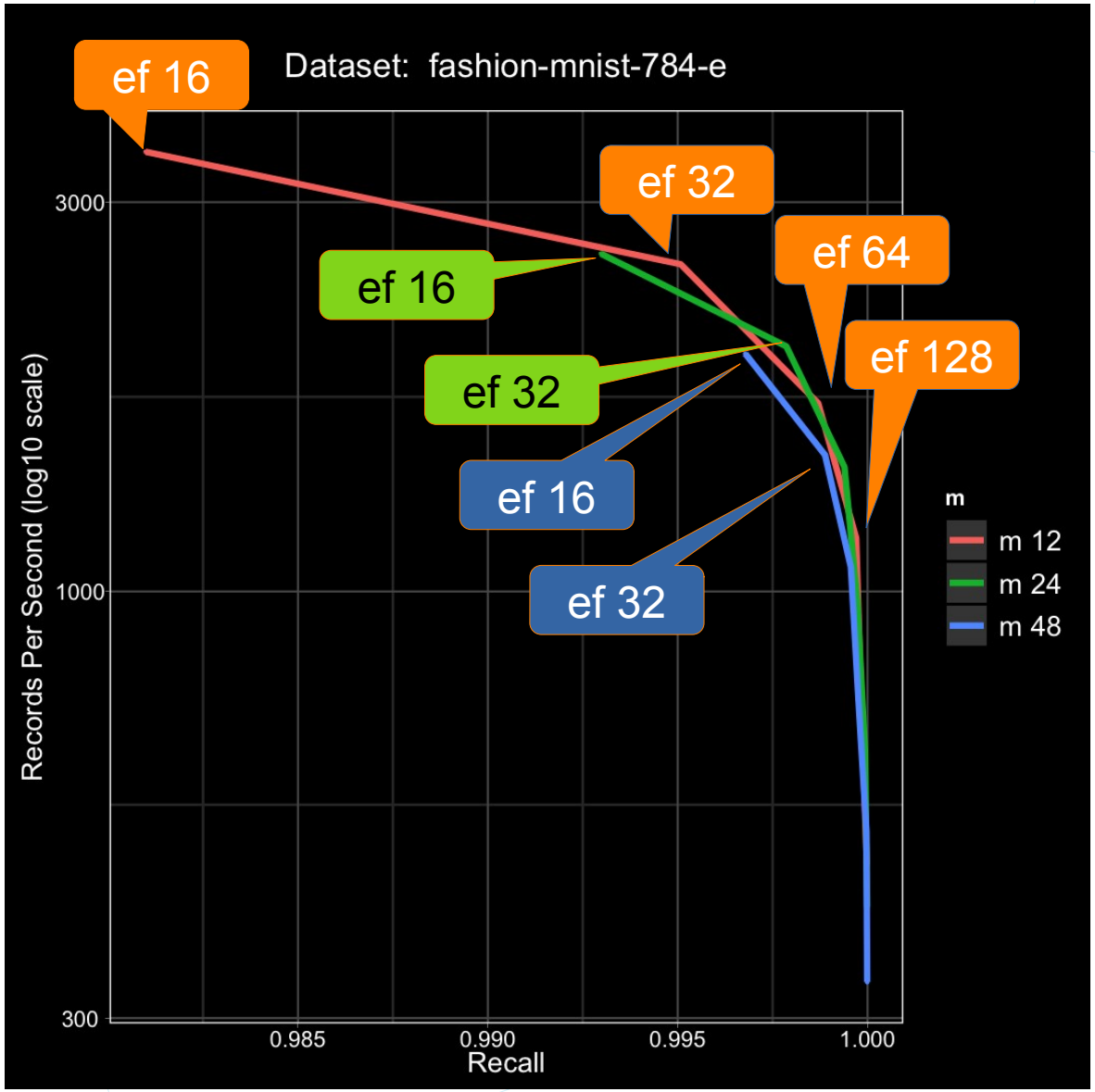
- M — neighbors number of every vector
- Vectors are distributed to level with decreasing quantity per level (for M=20):
  - 0 — 1 000 000
  - 1st — 50 000
  - 2nd — 2 500
  - 3rd — 125 etc.
- Entry point at highest level



Entry Point
Lookup Vector
Found Nearest Vector



Influence of Parameters on Speed and Accuracy



- The influence of the neighbors quantity **M** and the search depth **ef_construction** on the speed and recall

	Value	Index Build Time	Search Rate	Accuracy / Recall
M	↑ Higher	↘ Slower	↘ Slower	↗ Higher
	↓ Lower	↗ Faster	↗ Faster	↘ Lower
ef	↑ Higher	↘ Slower	↗ Slower	↗ Higher
	↓ Lower	↗ Faster	↘ Faster	↘ Lower

- The influence of the search depth **ef_search** on the speed and recall

	Value	Search Rate	Accuracy / Recall
ef	↑ Higher	↘ Slower	↗ Higher
	↓ Lower	↗ Faster	↘ Lower

Create a Table and Insert Data

1. Install extension in the current database

```
CREATE EXTENSION vector;
```

2. Create a table with a column of type vector

```
CREATE TABLE primer (id INT4, v vector(5));
```

3. Insert vector data

- SQL

```
INSERT INTO primer (id, v) VALUES (1,'[1,2,3,4,5]');
```

- Python

```
embedding = np.array([1, 2, 3, 4, 5])
conn.execute('INSERT INTO primer (id, v) VALUES
(1,%s)', (embedding,))
```

Search of Nearest Vectors (without filters)

- Set the number of the nearest neighbors to search for

```
SET hnsw.ef_search = 100;
```

- Search for 10 nearest neighbors

```
SELECT id
FROM primer
ORDER BY v <-> '[2,1,4,3,5]'
LIMIT 10;
```

PostgresPro

Increase of Search Rate — the Decrease of the Vector Elements Size to 2 Bytes

1. halfvec type — float 2 bytes
2. The data does not need to be converted to halfvec type:

```
CREATE INDEX primer_idx ON primer  USING hnsw
( (v::halfvec(5)) halfvec_l2_ops)
WITH (m=20,ef_construction=100);
```

3. Transform expression in a query:

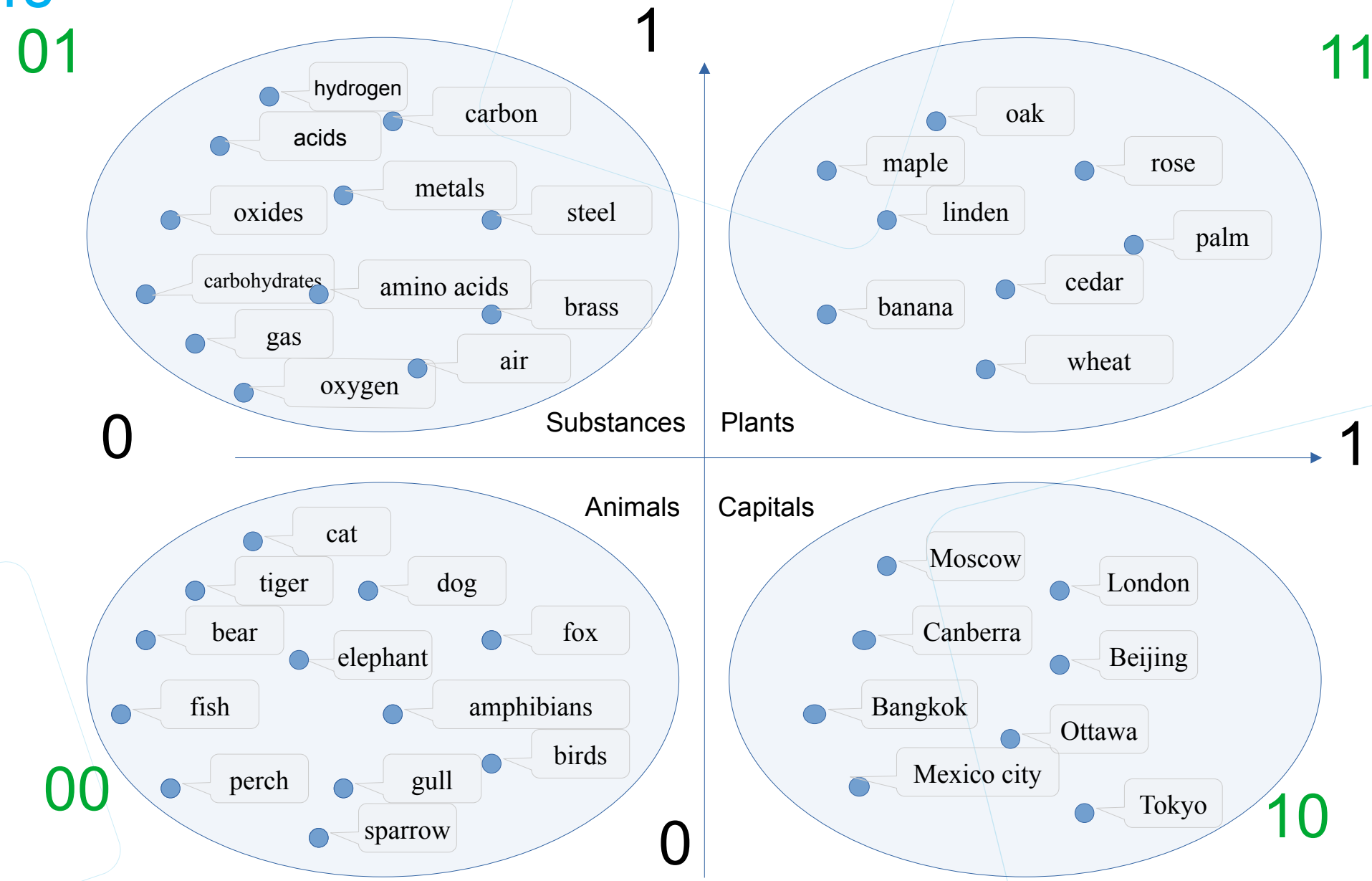
```
SET hnsw.ef_search = 100;
SELECT id FROM primer
ORDER BY v::halfvec(5) <-> '[2,1,4,3,5]' LIMIT 10;
```

PostgresPro

Increase of Search Rate — the Decrease of the Vector Elements Size to 1 bit

- The value x in the vector is replaced:
- by 1 if x > 0
- by 0 if x <= 0
- Reduces memory requirements by 32 times
- Works better on vectors with dimensions greater than 1000
- Supported by pgvector и pgvectorscale

Vectors — Hamming Distance



Increase of Search Rate — the Decrease of the Vector Elements Size to 1 bit

- Type bit - 1 bit per vector element
- Data do not need to be converted to bit:

```
CREATE INDEX primer_idx
ON primer
USING hnsw
( (binary_quantize(v)::bit(5)) bit_hamming_ops)
WITH (m=20,ef_construction=100);
```

PostgresPro

Increase of Search Rate — the Decrease of the Vector Elements Size to 1 bit

- Option 1 - Transform the expression in the search query:

```
SET hnsw.ef_search = 100;
SELECT id FROM primer
ORDER BY binary_quantize(v)::bit(5) <~>
binary_quantize('[2,1,4,3,5]'::vector(5) ) LIMIT 10;
```

- Option 2 - Search for the nearest and exact distances in a subquery (rerank)

```
SET hnsw.ef_search = 800;
SELECT i.id FROM (
SELECT id, v <-> '[2,1,4,3,5]' as distance FROM primer
ORDER BY binary_quantize(v)::bit(5) <~>
binary_quantize('[2,1,4,3,5]'::vector(5)) LIMIT 800) as i
ORDER BY i.distance LIMIT 10;
```

PostgresPro

Table and Index Size

Dataset Name	Vectors qty	Dimensions	Table Size MB	Index Size **vector** MB	Index Size **halfvec** MB	Index Size **bit** MB
sift-128-euclidean	1000000	128	488	793	544	304
gist-960-euclidean	1000000	960	3662	7680	2603	405
dbpedia-openai-1000k-angular	1000000	1536	5859	7734	3867	473

source https://jkatz05.com/post/postgres/pgvector-scalar-binary-quantization/

PostgresPro

Search Rate and Accuracy

Dataset Name	Vectors qty	Dimen-sions	Ef search	Time ms Accuracy	vector	halfvec	bit	bit rerank
sift-128-euclidean	1M	128	40	time	1.19	1.20	1.25	1.34
			40	accuracy	95.4%	95.4%	2.42%	4.19%
sift-128-euclidean	1M	128	800	time	15.07	14.8	16.05	17.28
			800	accuracy	100%	100%	2.52%	15.68%
gist-960-euclidean	1M	960	40	time	2.64	2.40	0.24	1.00
			40	accuracy	78%	78.1%	0%	0%
gist-960-euclidean	1M	960	800	time	29.20	25.4	0.28	1.82
			800	accuracy	99.6%	99.6%	0%	0%
dbpedia-openai	1M	1536	40	time	2.7	2.61	1.79	1.91
			40	accuracy	96.8%	96.8%	66.8%	91.6%
dbpedia-openai	1M	1536	800	time	30.50	28.49	19.57	21.34
			800	accuracy	99.9%	99.9%	68.6%	99.8%

Neighbors Search Queries with a Filter

- Need to find the closest ones that satisfy the condition, for example: language code is 1

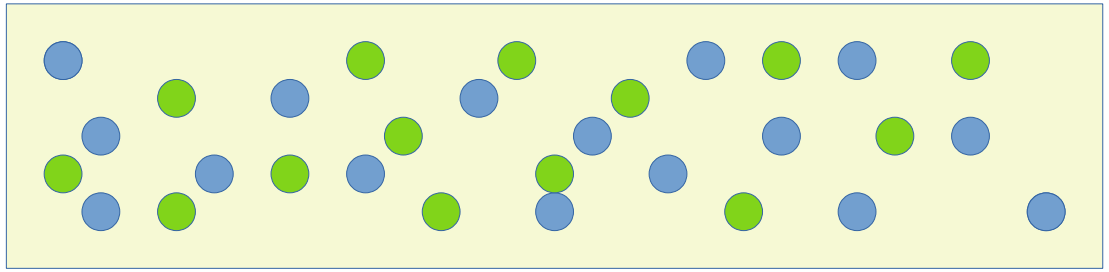
```
SELECT id FROM documents
WHERE language = 1
ORDER BY v <-> '[2,1,4,3,5]'
LIMIT 10;
```

- It is important to know the filter properties and adapt indexes and queries
  - Is the list of values fixed?
  - Number of values (2-5 or thousands)

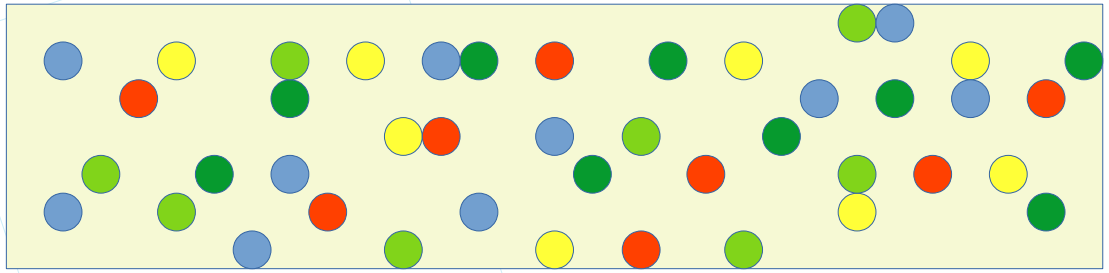
Selectivity and Correlation in Datasets

- Selectivity or specificity is the fraction of indexed data points filtered by predicate

High selectivity (50%)

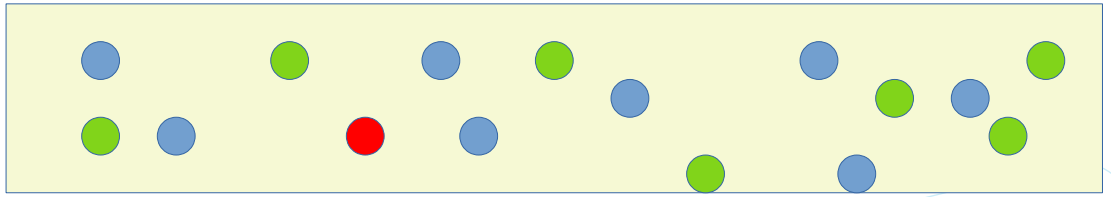


Lower selectivity (~20%)

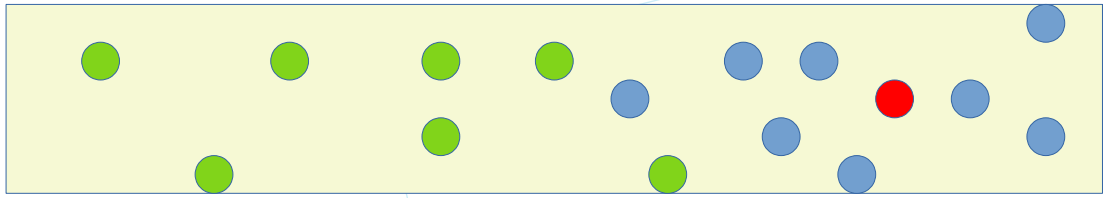


- Correlation is the difference between the distribution of query vectors and predicate vectors.

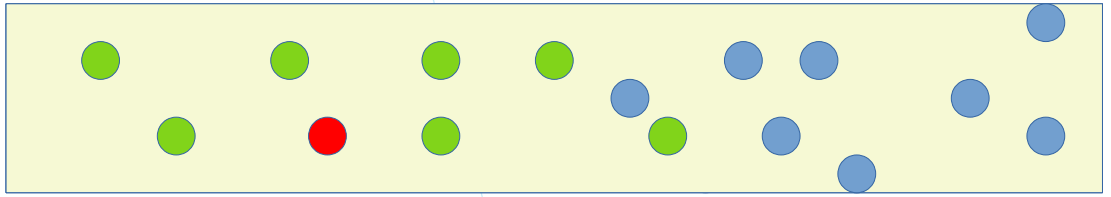
No correlation



Positive correlation



Negative correlation



PostgresPro

Filter Search Patterns

Filter values \ Rows	No filter	Ten	Hundred	Thousand
Less Rows in a Table	Regular vector search	Every 10th matches the condition. Acceptable rate with ef_search 100 and LIMIT 10	Every 100th matches the condition. With LIMIT 10 you need to increase ef_search to 1000 or more (query rate decreases)	Only every 1000th vector meets the condition (ef_search 10000 does not work). **BUT you can try a regular BTREE index**
More Rows in a Table	Regular vector search, query time grows as log(Rows)	With ef_search 100 and LIMIT 10 time grows as log(Rows)/selectivity	The time for the request increases as log(Rows)/selectivity	**BTREE** index inefficient, vector index with pre-filter is required.

Filters — Partial Index

- Partial index - an index with a condition. As a result, only a part of the rows are in the index:

```
CREATE INDEX primer_part_idx ON primer USING
hnsw  (v vector_l2_ops)
WITH (m=20, ef_construction=100) WHERE lang = 1;
```

- Search with partial index

```
SELECT id FROM primer
WHERE lang = 1
ORDER BY v <-> '[2,1,4,3,5]'
LIMIT 10;
```

- Works with any vector index

Filters — Streaming or Iterative Scan

- Pgvector from version 0.8.0

```
SET hnsw.iterative_scan = relaxed_order;

SELECT id FROM primer
WHERE id = 1
ORDER BY v <-> '[2,1,4,3,5]'
LIMIT 10;
```

Filters — Streaming or Iterative Scan

- Pgvectorscale
  - Create index:

```
CREATE INDEX primer_part_idx ON primer
USING diskann (v)
WITH (storage_layout=plain,
num_neighbors=50, search_list_size=200)
```

  - Query:

```
SELECT id FROM primer
WHERE lang = 1
ORDER BY v <=> '[2,1,4,3,5]' LIMIT 10;
```

When a Regular Index is Faster

- If there are many unique filter values, then each one corresponds to a few vectors. We make a REGULAR btree index by the filter condition:

```
CREATE INDEX on documents
(language, authorid, date);
```

- Search with the same command (without vector index):

```
SELECT id from documents
WHERE language = 1 AND authorid = 42
ORDER BY v <=> '[2,1,4,3,5]' LIMIT 10;
```

Filters — Multi-column index

- Multi-column index demonstrates good query rate under all filter conditions
- The problem - there are no multi-column vector indexes available in the open source yet.
- For example, a multi-column index from the PostgresPro package
  - Add extension to a database

```
CREATE EXTENSION mc_hnsw CASCADE;
```

  - Create index:

```
CREATE INDEX on documents USING gann (v mc_hnsw,
language, date, authorid) WITH (m=16,
ef_construction=100);
```

  - Search with filters:

```
SELECT id from documents
WHERE language = 1 AND authorid = 42
ORDER BY v <=> '[2,1,4,3,5]'LIMIT 10;
```

Filter Search Patterns — conclusions

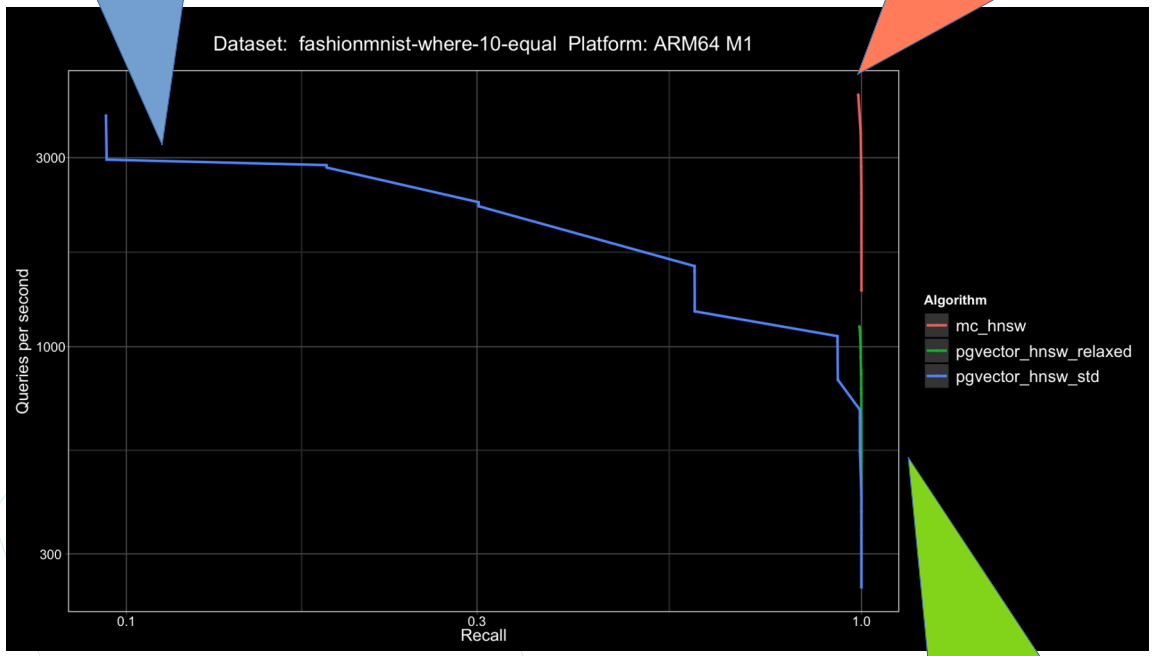
Filter values \ Algorithms	No filter	Tens	Hundred	Thousand
Regular vector search	Fast, good accuracy	Accuracy decreases, query rate drops slightly	Accuracy drops to 1-10%, query rate doesn't matter	BTREE index by filter conditions (prefiltering and seqscan in table with vectors)
Streaming Vector search	Fast, good accuracy	Good accuracy, query rate decreases	Good accuracy, query rate drops by 10-100 times	Good accuracy, very low query rate
Multicolumn Vector Index	Fast, good accuracy	Fast, good accuracy	Fast, good accuracy for = operator, slower for < and > operators	Fast, good accuracy for = operator, slower for < and > operators

Filter Search Patterns — conclusions

Selectivity 10%

Regular HNSW

Multi-column index with HNSW



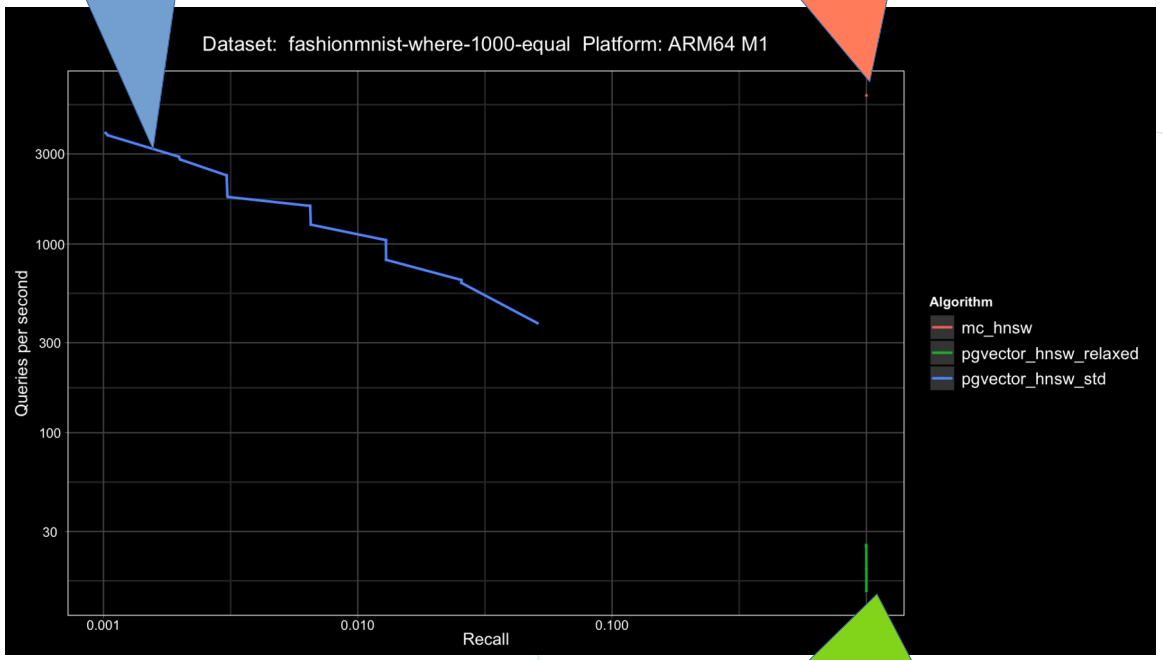
0.1 0.3 1.0

HNSW with streaming

Selectivity 0.1%

Regular HNSW

Multi-column index with HNSW



0.001 0.01 0.1

HNSW with streaming

PostgresPro

Recommendations

- For storage you can choose the vector type
- To speed up your search, use the HNSW index
- Select parameters to start with:
  Small m, for example 12-16
  Relatively small ef_coefficient (100-200)
  Increase precision when querying by increasing ef_search
- If the accuracy is not enough
  Increase ef_coefficient when building a new index
- If the accuracy is not enough
  Increase m (this will increase the index)

Recommendations

- If the accuracy is sufficient, but the index size is large
  Create an index with halfvec and fix the query for halfvec
- If the index is large and the vector dimension is 1000 or more, it is recommended to try an index with the bit type and additional sorting
- If the query has filters
- If there are a lot of values in the filter, first try the BTREE index by filter without a vector index
- If the speed is not enough, create a vector index as recommended above
- Use the option SET hnsw.iterative_scan = relaxed_order

PostgresPro

Conclusions

- Postgres allows you to store large amounts of vector data
  - The advantages of the database are standard API, backup, user management, replication, etc.
- Many extensions for vector search
  - Pgvector — the most mature
  - Pgvectorscale — innovative but still in development
  - Pgvecto.rs — there is support for streaming, written in Rust
  - pgpro_vector — multicolumn vector index
- It is possible to control vector types and index formats depending on the task
- Postgres is comparable in performance to vector databases, while retaining the advantages of a relational database
- Postgres needs to improve support for multi-column indexes and improve cost estimation procedures

PostgresPro

Thank you for attention!

Answers to Questions