Hacking pgvector for performance

sednai

Daniel Krefl
Sednai

$1.$2 @ sedn.ai

(feedback link)

@ PGonf.EU 2025

sednai

AERO

Intro

Aero aims to complement the efforts of the *EU Processor Initiative (EPI)* project by developing the open-source software ecosystem required to not only improve the efficiency of the EPI hardware but also accelerate and ease the processor's integration into the cloud.

Funded by the European Union

UK Research and Innovation

Project funded by

Schweizerische Eidgenossenschaft
Confédération suisse
Confederazione Svizzera
Confederaziun svizra

Swiss Confederation

Federal Department of Economic Affairs, Education and Research EAER
State Secretariat for Education, Research and Innovation SERI

Funded by the European Union. Views and opinions are however those of the author(s) only and do not necessarily reflect those of the European Union or the HaDEA. Neither the European Union nor the granting authority can be held responsible for them. Project number: 101092850.

AERO has also received funding from UKRI under grants no. 10048318 and 10048915, and the Swiss State Secretariat for Education, Research, and Innovation.

Intro

AERO

Aero aims to complement the efforts of the *EU Processor Initiative (EPI)* project by developing the open-source software ecosystem required to not only improve the efficiency of the EPI hardware but also accelerate and ease the processor’s integration into the cloud.

Target platform:

RHEA 1

HPC and AI processor

European high-performance energy-efficient processor (ARM based), dedicated to high performance computing, and designed to work with third-party accelerators, see [https://sipearl.com/]

RHEA images kindly provided by SIPEARL

Intro

Aero aims to complement the efforts of the *EU Processor Initiative (EPI)* project by developing the open-source software ecosystem required to not only improve the efficiency of the EPI hardware but also accelerate and ease the processor's integration into the cloud.

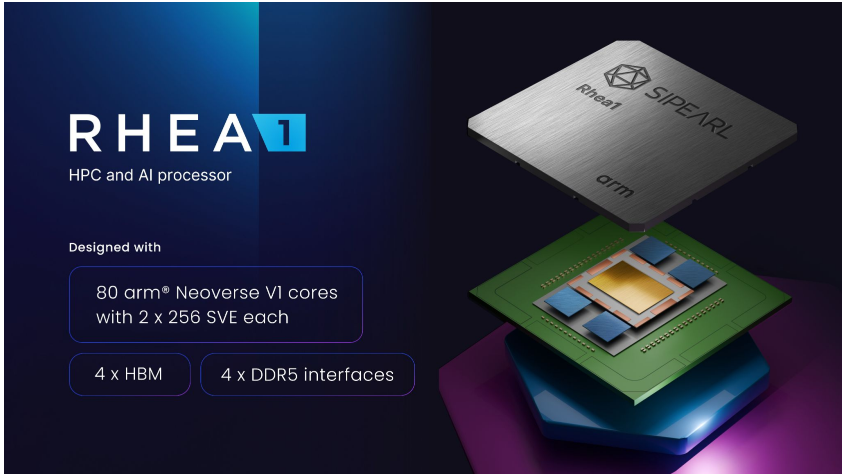
Target platform:

RHEA 1

HPC and AI processor

[https://sipearl.com/]

RHEA images kindly provided by SIPEARL



Intro

sednai

AERO

Aero aims to complement the efforts of the *EU Processor Initiative (EPI)* project by developing the open-source software ecosystem required to not only improve the efficiency of the EPI hardware but also accelerate and ease the processor's integration into the cloud.

Objectives: [https://aero-project.eu/about/]

- *Managed Programming Languages*
- *Native Programming Languages & Runtimes*
- *OS, drivers & virtualization support*
- *State-of-the-art cloud deployments*
- *Hardware acceleration for performance & security*
- *Adoption of the EU cloud ecosystem*

Intro

sednai

AERO

Aero aims to complement the efforts of the *EU Processor Initiative (EPI)* project by developing the open-source software ecosystem required to not only improve the efficiency of the EPI hardware but also accelerate and ease the processor's integration into the cloud.

Use cases / pilots:

- *Automotive Digital Twins with IoT-Cloud Interoperability*
- *High Performance Algorithms for Space Exploration (Gaia)*
- *HPC/Cloud Database Acceleration for Scientific Computing*

Intro

Aero aims to complement the efforts of the *EU Processor Initiative (EPI)* project by developing the open-source software ecosystem required to not only improve the efficiency of the EPI hardware but also accelerate and ease the processor's integration into the cloud.

Use cases / pilots:

- *Automotive Digital Twins with IoT-Cloud Interoperability*
- *High Performance Algorithms for Space Exploration (Gaia)*
- *HPC/Cloud Database Acceleration for Scientific Computing*

Leadership / Coordination of pilots

Intro

sednai

AERO

Aero aims to complement the efforts of the *EU Processor Initiative (EPI)* project by developing the open-source software ecosystem required to not only improve the efficiency of the EPI hardware but also accelerate and ease the processor's integration into the cloud.

Use cases / pilots:

- *Automotive Digital Twins with IoT-Cloud Interoperability*
- *High Performance Algorithms for Space Exploration (Gaia)*
- *HPC/Cloud Database Acceleration for Scientific Computing*

ICCS UBITECH SIPEARL PIERER INNOVATION Red Hat FORTH Università di Pisa Virtual Open Systems Université de Genève sednai Manchester 1824 The University of Manchester codeplay

… GAIA …

Gaia

Gaia was a space based astronomy telescope of ESA operational 2014-2025.

Gaia has made more than three trillion observations of two billion stars and other objects throughout our Milky Way galaxy and beyond, mapping their motions, luminosity, temperature and composition.

Scientific objectives:

- *First 3d map of our galaxy*
- *Insides on the origin and formation of our galaxy*
- *Detection of diverse variable phenomena*
- *Many more … see* [https://www.cosmos.esa.int/web/gaia/science]

Gaia

sednai

Gaia was a space based astronomy telescope of ESA operational 2014-2025.

Gaia has made more than three trillion observations of two billion stars and other objects throughout our Milky Way galaxy and beyond, mapping their motions, luminosity, temperature and composition.

Scientific objectives:

- *First 3d map of our galaxy*
- *Insides on the origin and formation of our galaxy*
- *Detection of diverse variable phenomena*
- *Many more … see* [https://www.cosmos.esa.int/web/gaia/science]

Gaia

sednai

Gaia was a space based astronomy telescope of ESA operational 2014-2025.

Gaia has made more than three trillion observations of two billion stars and other objects throughout our Milky Way galaxy and beyond, mapping their motions, luminosity, temperature and composition.

Data and compute challenge:

- *Petabyte scale*
- *~ 10 Billion photometric time series*
- *~ 5 Billion spectra time series*

We solve this with Postgres !

Intro

Aero aims to complement the efforts of the *EU Processor Initiative (EPI)* project by developing the open-source software ecosystem required to not only improve the efficiency of the EPI hardware but also accelerate and ease the processor's integration into the cloud.

Intertwined Gaia+SED pilots:

- *Process efficiently constantly increasing volumes of data.*
  - *Enable GPU computations directly within the database*
  - *Optimize data and compute pipeline for RHEA*

Intro

Aero aims to complement the efforts of the *EU Processor Initiative (EPI)* project by developing the open-source software ecosystem required to not only improve the efficiency of the EPI hardware but also accelerate and ease the processor's integration into the cloud.

Intertwined Gaia+SED pilots:

- *Process efficiently constantly increasing volumes of data.*

- *Enable GPU computations directly within the database*

Example: Hack to GPU accelerate a Postgres vector index

(WARNING: This will be technical …)

... VECTOR SEARCH ...

Vector Search

sednai

Motivation

Many data analysis algorithms require a nearest neighbour search in a D-dim space.
Often just referred to as *Vector Search*.

sednai

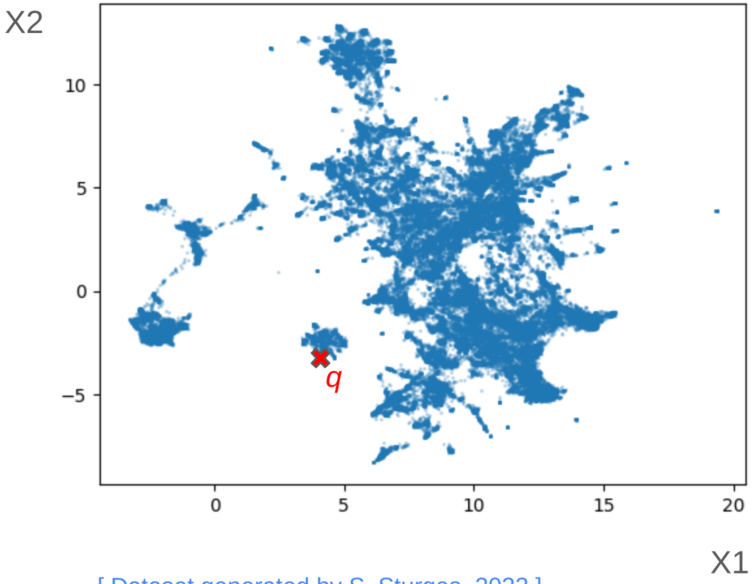
Motivation

Many data analysis algorithms require a nearest neighbour search in a D-dim space.
Often just referred to as *Vector Search*.

For a query point *q :*

What are its *k* nearest neighbors?

Wikipedia - OpenAI embeddings
1536d -> 2d projection (UMAP for 100k subset)



[Dataset generated by S. Sturges, 2023]
(https://www.kaggle.com/datasets/stephanst/wikipedia-simple-openai-embeddings)

sednai

Motivation

Many data analysis algorithms require a nearest neighbour search in a D-dim space.
Often just referred to as *Vector Search*.

For a query point *q* :

What are its *k* nearest neighbors?

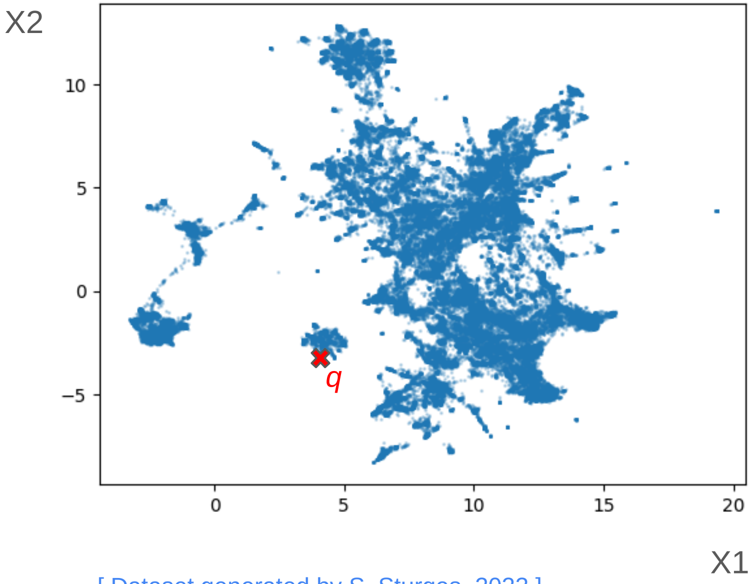
Note:

Growing interest due to ML / AI generated vector embeddings.

For instance:

Retrieval-Augmented Generation

Wikipedia - OpenAI embeddings
1536d -> 2d projection (UMAP for 100k subset)



[Dataset generated by S. Sturges, 2023]
(https://www.kaggle.com/datasets/stephanst/wikipedia-simple-openai-embeddings)

sednai

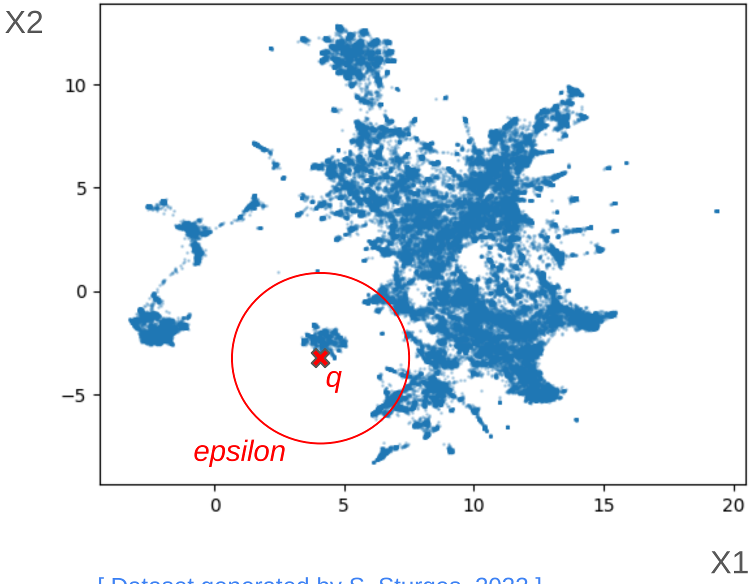
Motivation

Many data analysis algorithms require a nearest neighbour search in a D-dim space.
Often just referred to as *Vector Search*.

For a query point *q* :

What points are close by ?

Wikipedia - OpenAI embeddings
1536d -> 2d projection (UMAP for 100k subset)



[Dataset generated by S. Sturges, 2023]
(https://www.kaggle.com/datasets/stephanst/wikipedia-simple-openai-embeddings)

Vector Search

sednai

Motivation

Many data analysis algorithms require a nearest neighbour search in a D-dim space.
Often just referred to as *Vector Search*.

For a query point *q* :

What points are close by ?

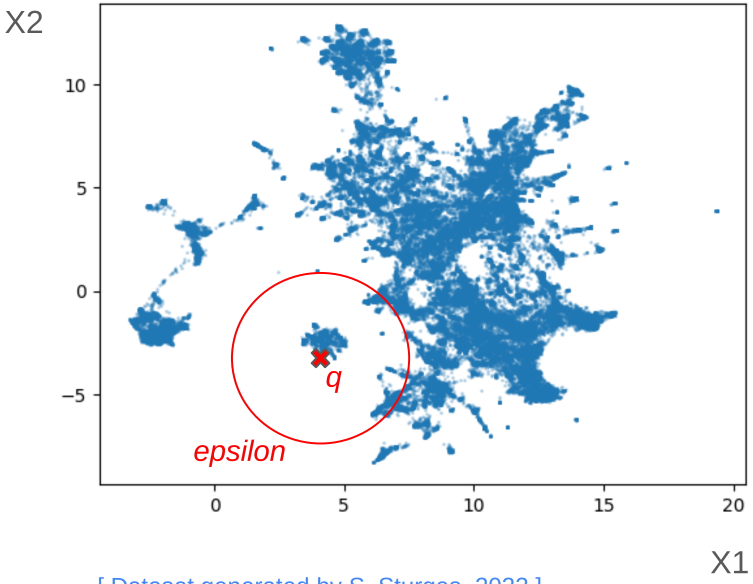
Note:

Of interest for classical unsupervised learning algorithms.

kNN, *DBSCAN*, ...

→ *Application to Gaia data*

Wikipedia - OpenAI embeddings
1536d -> 2d projection (UMAP for 100k subset)



[Dataset generated by S. Sturges, 2023]
(https://www.kaggle.com/datasets/stephanst/wikipedia-simple-openai-embeddings)

Vector Search

Motivation

Many data analysis algorithms require a nearest neighbour search in a D-dim space.
Often just referred to as *Vector Search*.

BUT:

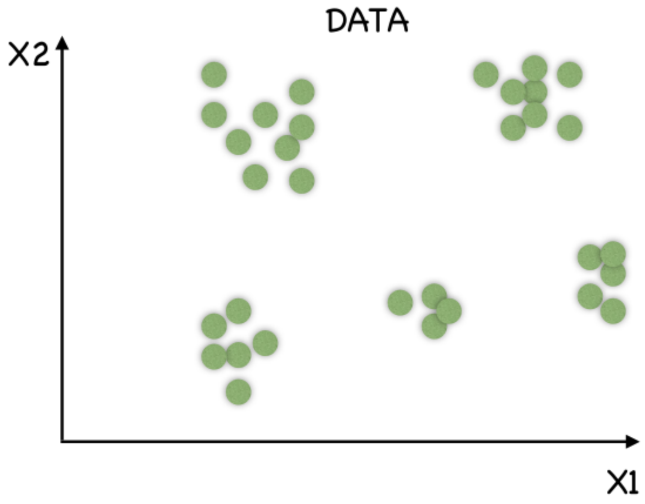
Requires for each query point *N* distance calculations + ranking.
(With *N* the number of datapoints in the dataset)

How to scale to large datasets ?

sednai

Approximate Nearest Neighbour search

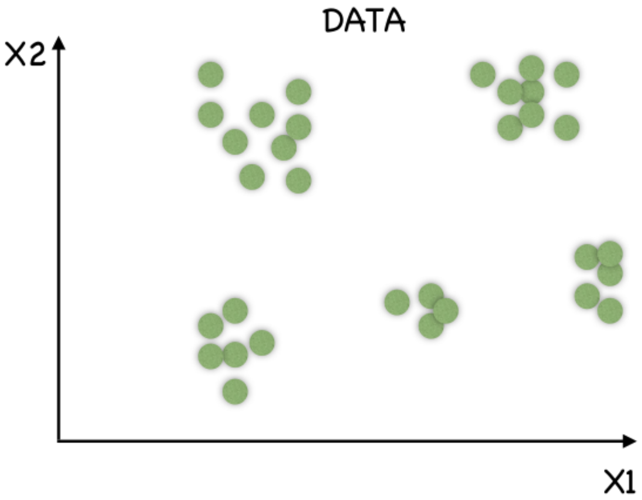
Several flavours exist, but *IVFFLAT* is the conceptually simplest algorithm.



[Approximate Nearest Neighbour search]

Several flavours exist, but *IVFFLAT* is the conceptually simplest algorithm.

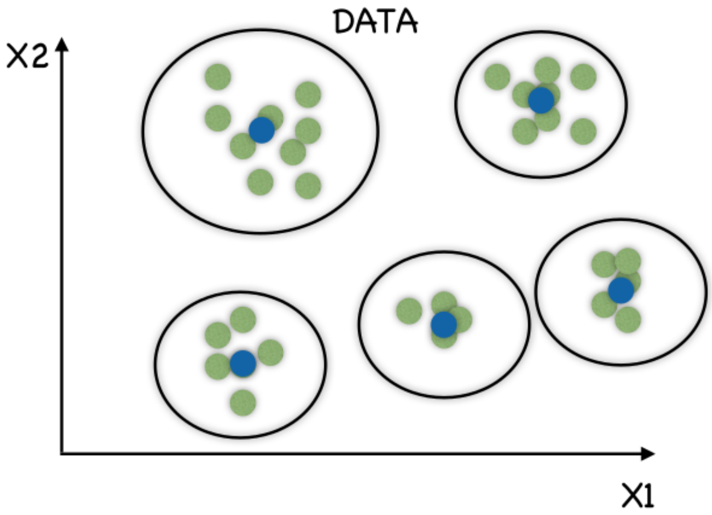
InVerted File Flat Index



IVFFLAT

Several flavours exist, but *IVFFLAT* is the conceptually simplest algorithm.

- Build vector index on dataset via K-means clustering
- Index each datapoint to the closest cluster (centroid)



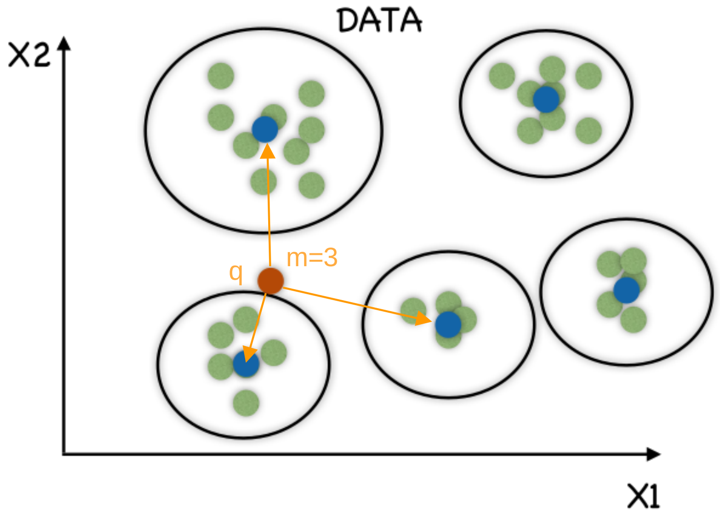
sednai

IVFFLAT

Several flavours exist, but *IVFFLAT* is the conceptually simplest algorithm.

- Build vector index on dataset via K-means clustering
- Index each datapoint to the closest cluster (centroid)
- Evaluate query point *q* only against cluster members of the *m* nearest centroids.

→ Recall / Performance tradeoff



BUT:

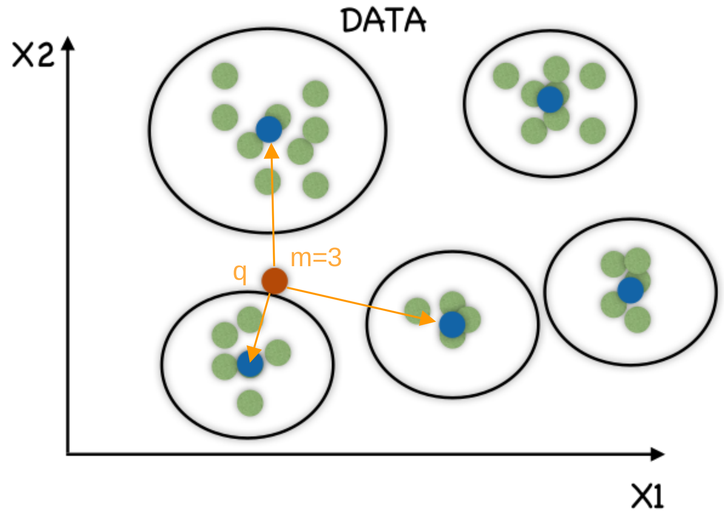
- Strongly depends on distribution of data
- High recall may effectively result in brute-force scan

sednai

IVFFLAT

Several flavours exist, but *IVFFLAT* is the conceptually simplest algorithm.

- Build vector index on dataset via K-means clustering
- Index each datapoint to the closest cluster (centroid)
- Evaluate query point *q* only against cluster members of the *m* nearest centroids.



→ Recall / Performance tradeoff

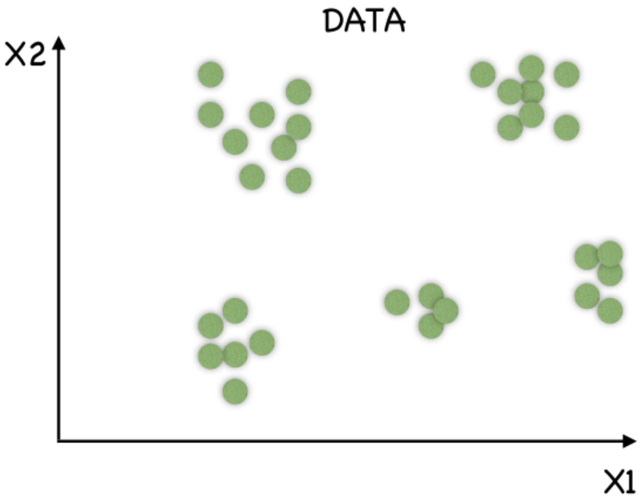
GOOD:

- Relatively fast and cheap to build index
- Very suitable for parallelization

sednai

Approximate Nearest Neighbour search

A bit more conceptually challenging is *HNSW*. But nowadays often preferred algorithm due to generally better recall / performance behavior.



[Malkov and Yashunin, 2016]

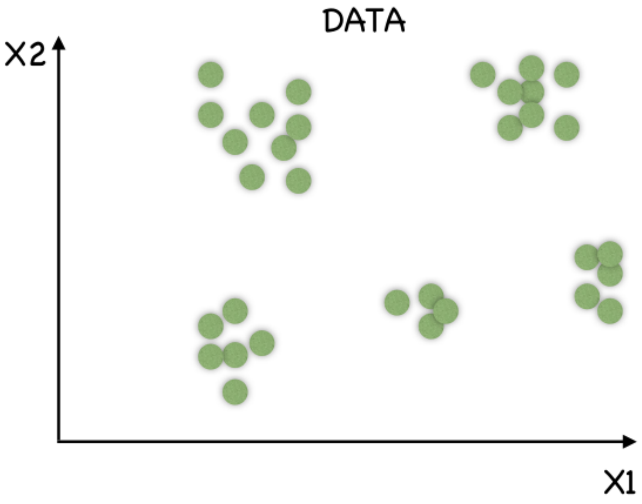
sednai

Approximate Nearest Neighbour search

A bit more conceptually challenging is *HNSW*. But nowadays often preferred algorithm due to generally better recall / performance behavior.

Hierarchical navigable small world

This talk: Mainly inference will be discussed

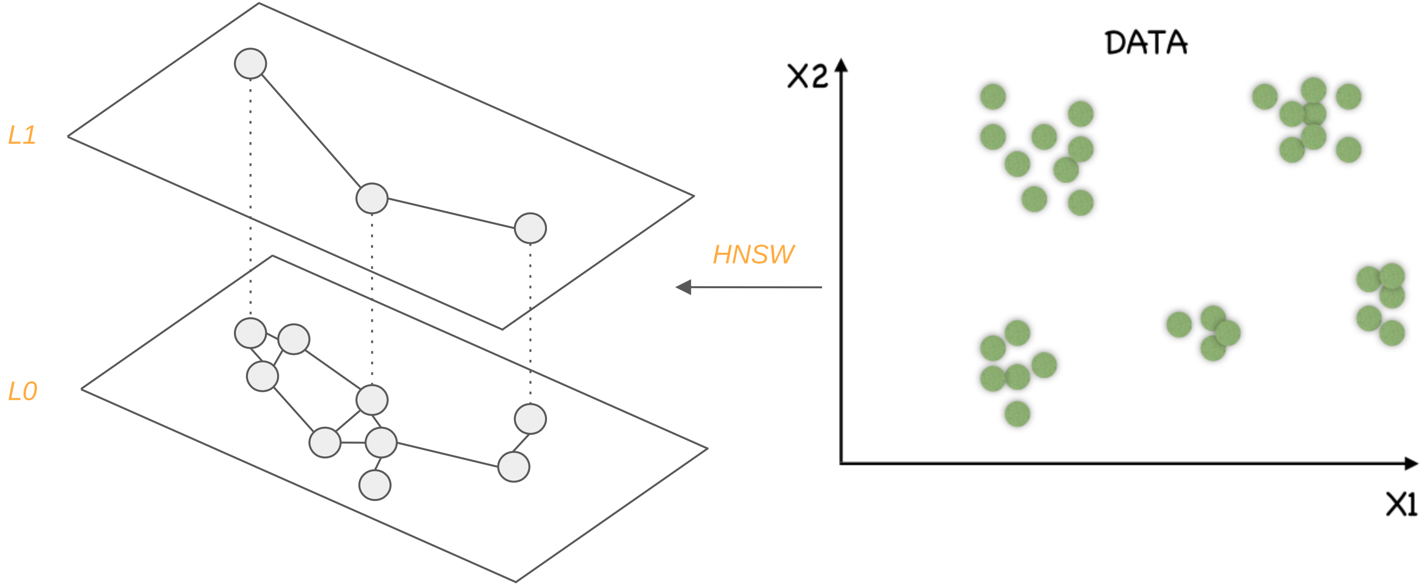


[Malkov and Yashunin, 2016]

sednai

HNSW inference

A bit more conceptually challenging is *HNSW*. But nowadays often preferred algorithm due to generally better recall / performance behavior.



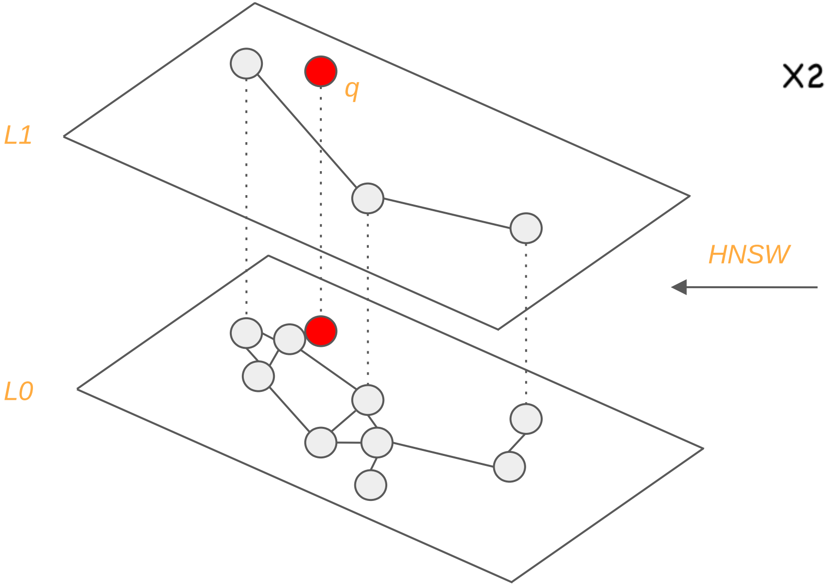
Multi-layered graph
(figure is simplified, usually many more layers)

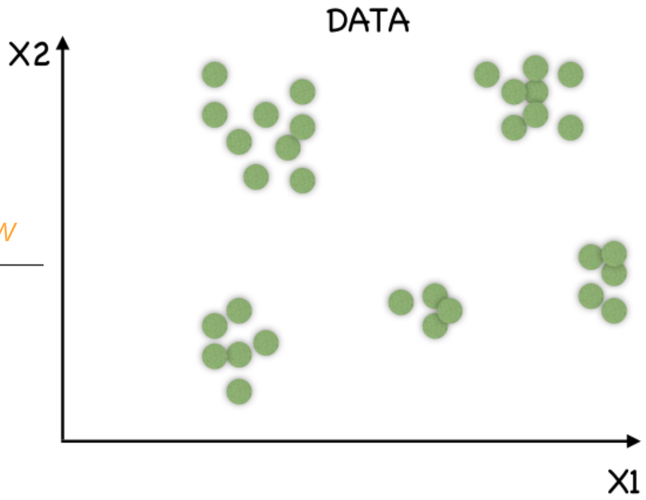
[Malkov and Yashunin, 2016]

sednai

HNSW inference

A bit more conceptually challenging is *HNSW*. But nowadays often preferred algorithm due to generally better recall / performance behavior.





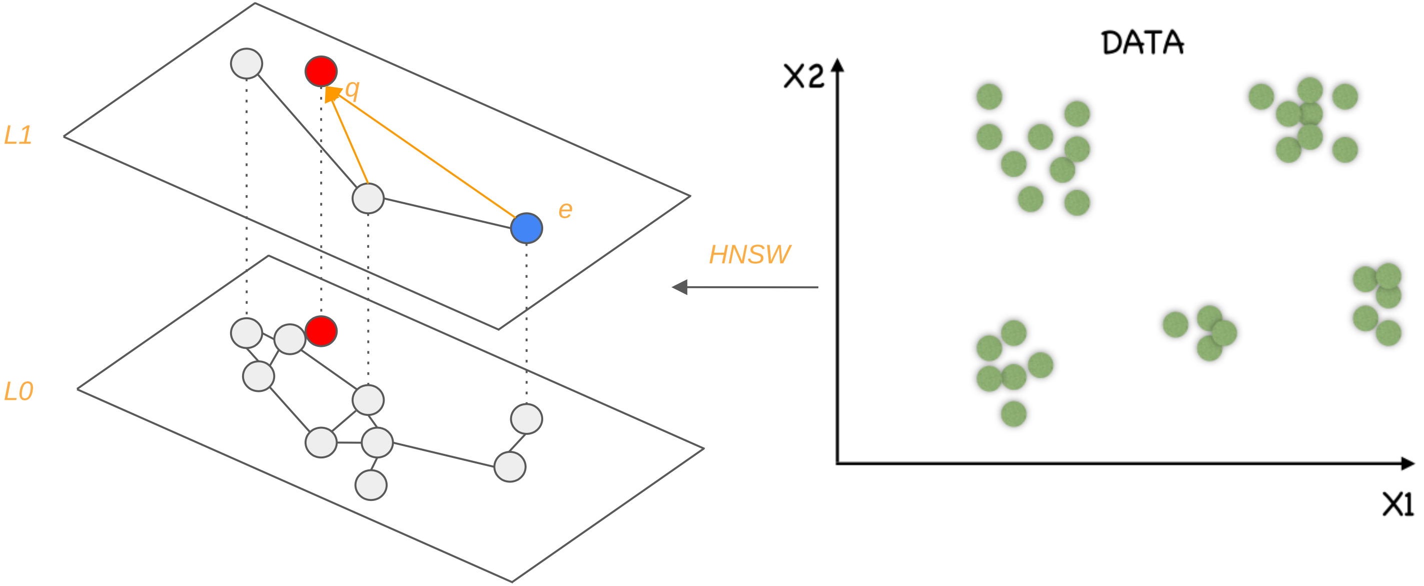
Search via greedy graph traversal
(keeping closest k visited nodes along the way)

[Malkov and Yashunin, 2016]

sednai

HNSW inference

A bit more conceptually challenging is *HNSW*. But nowadays often preferred algorithm due to generally better recall / performance behavior.



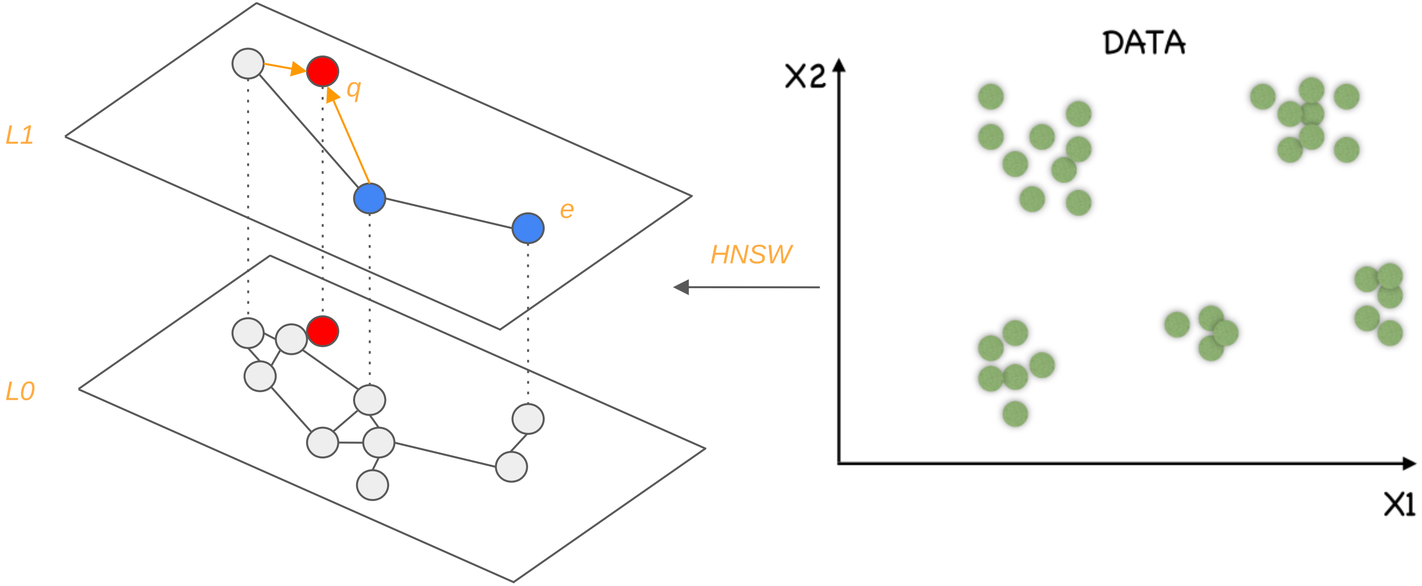
Search via greedy graph traversal
(keeping closest k visited nodes along the way)

[Malkov and Yashunin, 2016]

Vector Search

HNSW inference

A bit more conceptually challenging is *HNSW*. But nowadays often preferred algorithm due to generally better recall / performance behavior.



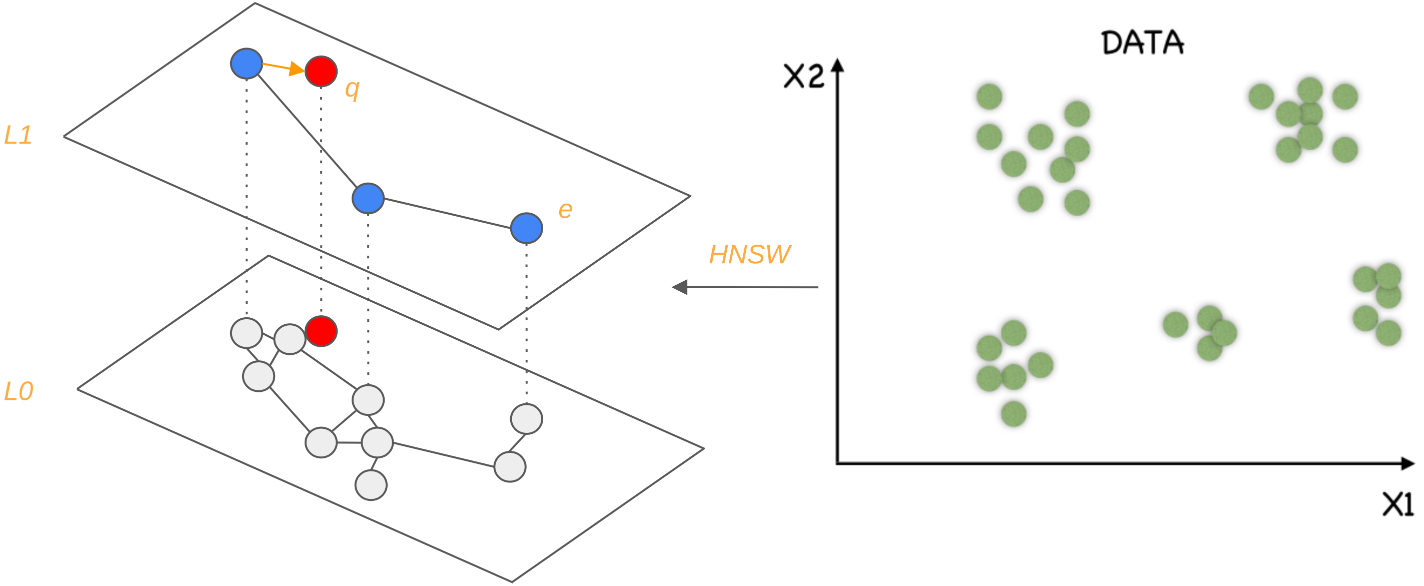
Search via greedy graph traversal
(keeping closest k visited nodes along the way)

[Malkov and Yashunin, 2016]

sednai

HNSW inference

A bit more conceptually challenging is *HNSW*. But nowadays often preferred algorithm due to generally better recall / performance behavior.

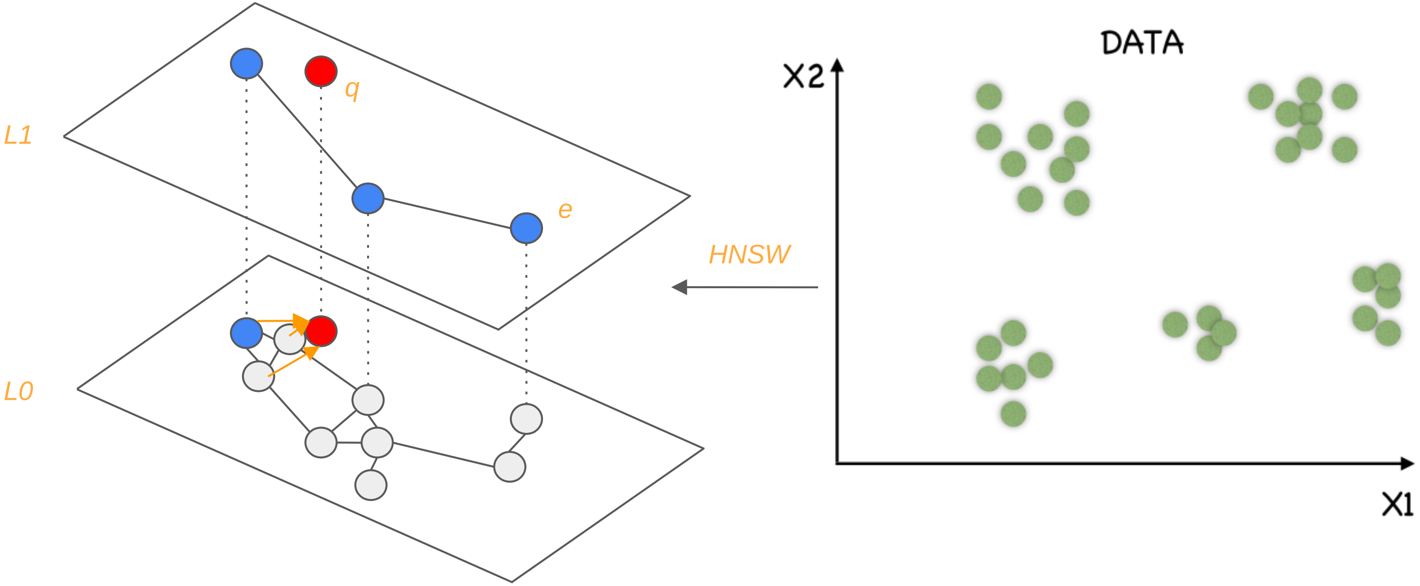


Search via greedy graph traversal
(keeping closest k visited nodes along the way)

[Malkov and Yashunin, 2016]

HNSW inference

A bit more conceptually challenging is *HNSW*. But nowadays often preferred algorithm due to generally better recall / performance behavior.



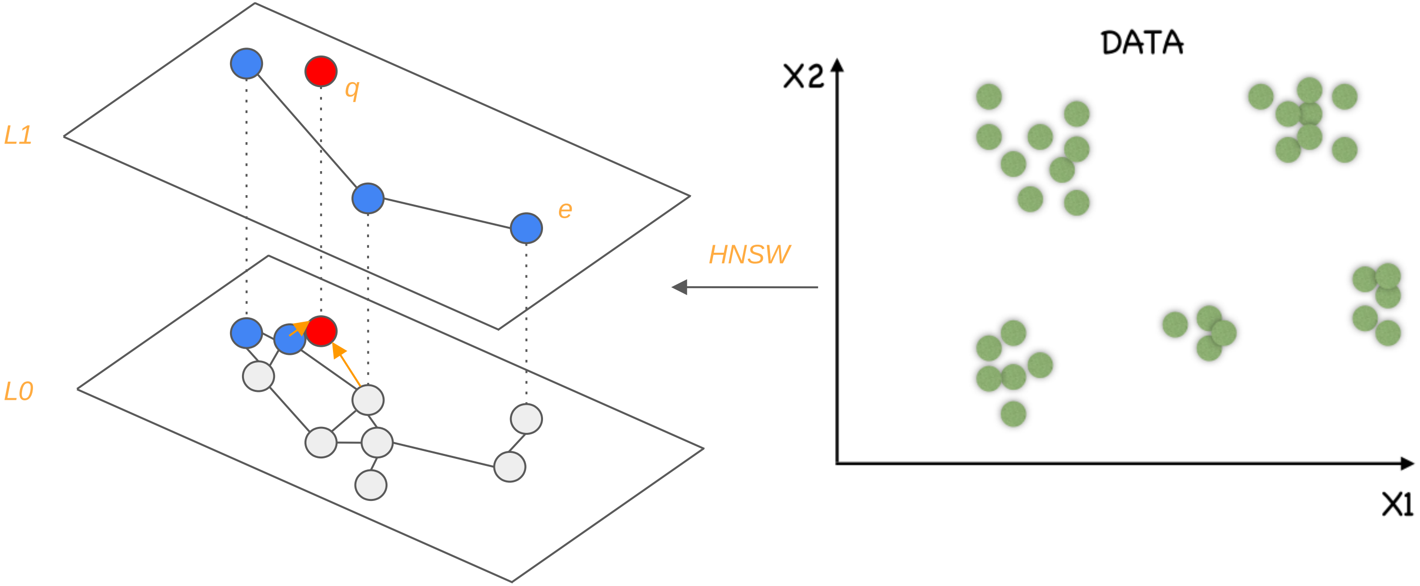
Search via greedy graph traversal
(keeping closest k visited nodes along the way)

[Malkov and Yashunin, 2016]

sednai

HNSW inference

A bit more conceptually challenging is *HNSW*. But nowadays often preferred algorithm due to generally better recall / performance behavior.



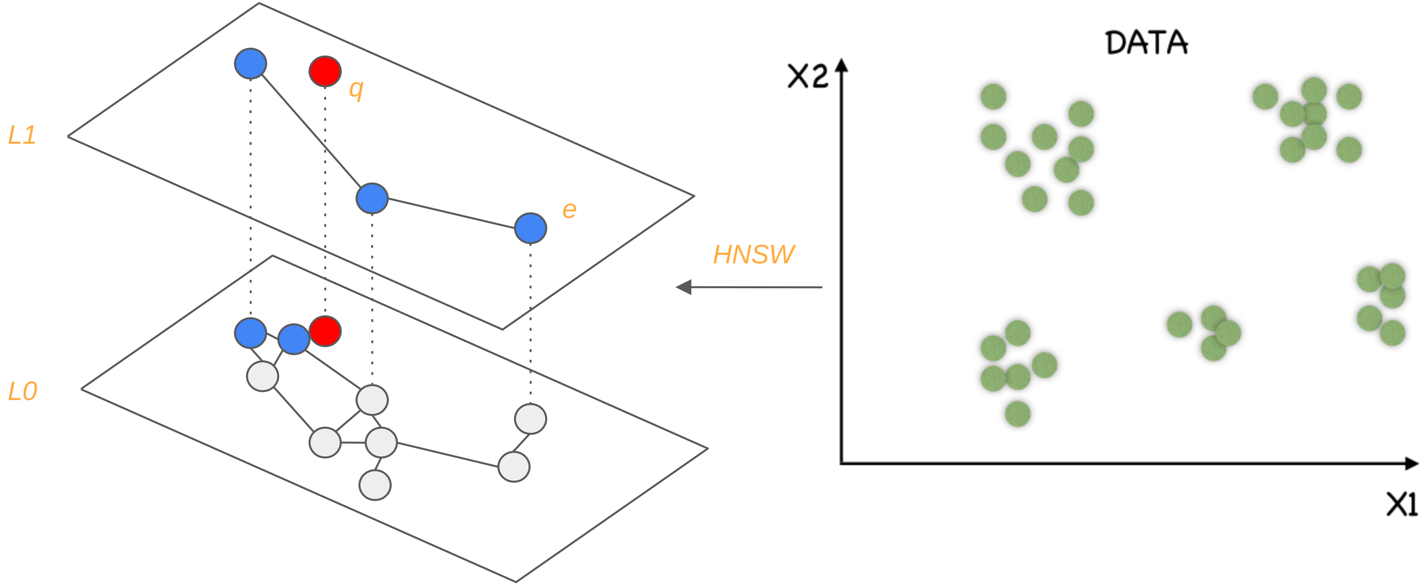
Search via greedy graph traversal
(keeping closest k visited nodes along the way)

[Malkov and Yashunin, 2016]

sednai

HNSW inference

A bit more conceptually challenging is *HNSW*. But nowadays often preferred algorithm due to generally better recall / performance behavior.



Search via greedy graph traversal
(keeping closest k visited nodes along the way)

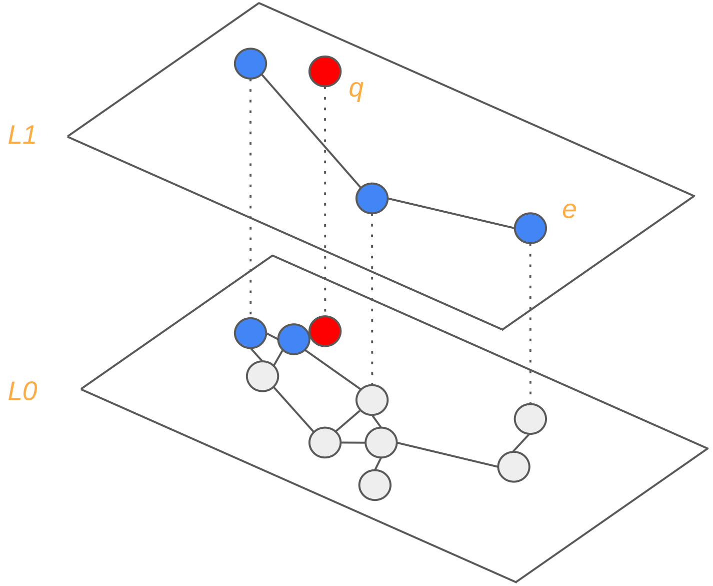
[Malkov and Yashunin, 2016]

Vector Search

sednai

HNSW inference

A bit more conceptually challenging is *HNSW*. But nowadays often preferred algorithm due to generally better recall / performance behavior.



Search via greedy graph traversal
(keeping closest k visited nodes along the way)

BUT:

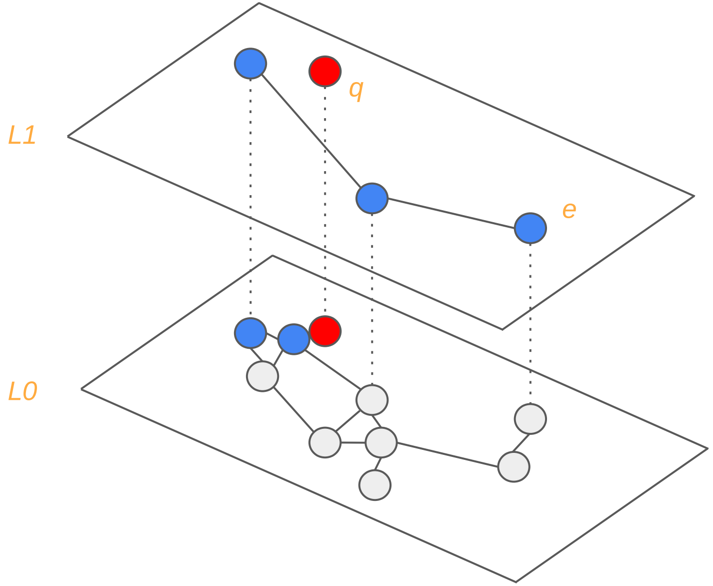
- Graph construction can be very compute and memory intensive
- Difficult to parallelize and distribute
- Large top *k* with filters tricky

[Malkov and Yashunin, 2016]

Vector Search

HNSW inference

A bit more conceptually challenging is *HNSW*. But nowadays often preferred algorithm due to generally better recall / performance behavior.



Search via greedy graph traversal
(keeping closest k visited nodes along the way)

BUT:

- Graph construction can be very compute and memory intensive
- Difficult to parallelize and distribute
- Large top *k* with filters tricky

[Malkov and Yashunin, 2016]

Vector Search

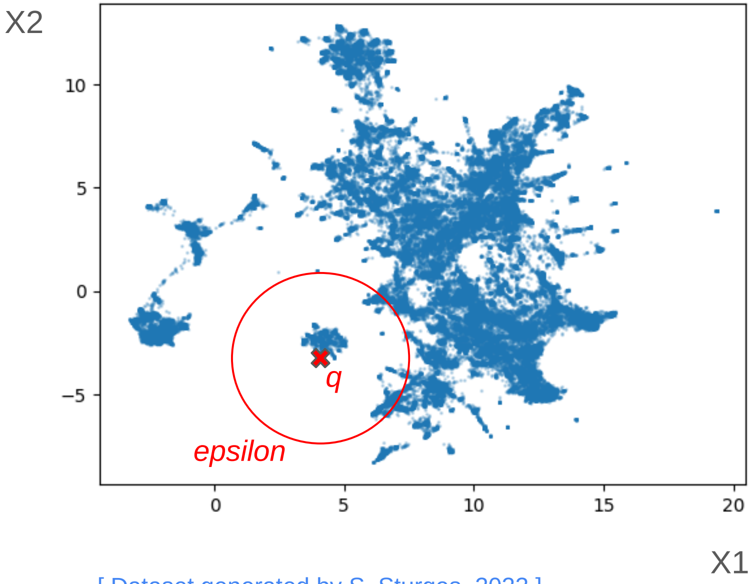
HNSW inference

“Large top k with filters tricky”

For a query point *q* :

What points are close by ?

Wikipedia - OpenAI embeddings
1536d -> 2d projection (UMAP for 100k subset)



[Dataset generated by S. Sturges, 2023]
(https://www.kaggle.com/datasets/stephanst/wikipedia-simple-openai-embeddings)

Vector Search

sednai

HNSW inference

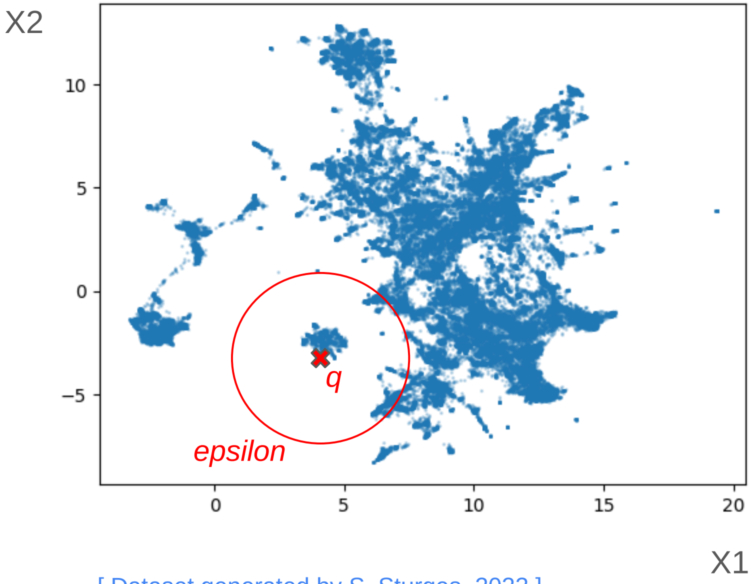
"Large top k with filters tricky"

For a query point *q* :

What points are close by ?

→ *IVFFLAT of main interest for this talk.*

Wikipedia - OpenAI embeddings
1536d -> 2d projection (UMAP for 100k subset)



[Dataset generated by S. Sturges, 2023]
(https://www.kaggle.com/datasets/stephanst/wikipedia-simple-openai-embeddings)

Approximate Nearest Neighbour search

Remark:

There are also newer, more specialized, algorithms which gain in popularity.

For example:

DiskANN: Applicable to large scale problems (beyond RAM). [Subramanya et al., 2019]

CAGRA: HNSW variant optimized for GPU execution. [Ootomo et al., 2024]

Vector Search

Postgres implementation

Most well-known and popular: *pgvector*
(https://github.com/pgvector/pgvector)

Introduces new PG type (vector), distance operators acting on vectors, and can build indices for vector columns.

Exact, and *HNSW* or *IVFFLAT* based approximate vector search. Many different metrics, but here we are mainly interested in euclidean distance (<->) and cosine distance (<=>).

Query example:

*select * from table where embedding <-> '[...]' < 10 order by embedding <-> '[...]' limit 10000*

Vector Search

sednai

Postgres implementation

Most well-known and popular: *pgvector*
(https://github.com/pgvector/pgvector)

Introduces new PG type (vector), distance operators acting on vectors, and can build indices for vector columns.

Exact, and *HNSW* or *IVFFLAT* based approximate vector search. Many different metrics, but here we are mainly interested in euclidean distance (<->) and cosine distance (<=>).

Query example:

*select * from table where embedding <-> '[...]' < 10 order by embedding <-> '[...]' limit 10000*

Column of vector type

Metric operator (L2) acting on vector types

Vector type
for example: '[3,0.1,-1.2,5]'

Vector Search

sednai

[Postgres implementation]()

Most well-known and popular: *pgvector*
(https://github.com/pgvector/pgvector)

Introduces new PG type (vector), distance operators acting on vectors, and can build indices for vector columns.

Exact, and *HNSW* or *IVFFLAT* based approximate vector search. Many different metrics, but here we are mainly interested in euclidean distance (<->) and cosine distance (<=>).

Query example:

Index on vector column

*select * from table where embedding <-> '[...]' < 10 order by embedding <-> '[...]' limit 10000*

Reduce number of required distance calculations via "index" / approx vector search.
(Worst case: For all rows)

Vector Search

sednai

Postgres implementation

Newcomer: *pgvectorscale* *(timescale)*
(https://github.com/timescale/pgvectorscale/)

DiskANN based approximate vector search.

Offers euclidean distance, cosine distance and inner product.

Interesting feature: Label filter push down into index scan.

Vector Search

sednai

Postgres implementation

Newcomer: *pgvectorscale* *(timescale)*
(https://github.com/timescale/pgvectorscale/)

DiskANN based approximate vector search.

Offers euclidean distance, cosine distance and inner product.

Interesting feature: Label filter push down into index scan.

Note:

Mentioned only for completeness. In this talk I will not dive further into *pgvectorscale.*

... PGVECTOR ...

Vector Search

sednai

pgvector

PostgreSQL plan generation (simplified)

FROM

↓

WHERE

↓

ORDER BY

↓

LIMIT

Vector Search

sednai

[pgvector]

PostgreSQL plan generation (simplified)

FROM ← Sequential or index scan (depending on cost estimate)

↓

WHERE

↓

ORDER BY

↓

LIMIT

Vector Search

sednai

[pgvector]

PostgreSQL plan generation (simplified)

FROM

↓

WHERE

↓

ORDER BY

↓

LIMIT

Pgvector implements INDEXAM
(Index Access Method Interface)

Vector Search

sednai

pgvector

PostgreSQL plan generation (simplified)

FROM

WHERE

ORDER BY

LIMIT

Pgvector implements INDEXAM
(Index Access Method Interface)

Core index functions:

ambuild
aminsert
ambeginscan
amrescan
amendscan
amgettuple
…

Vector Search

[pgvector]

PostgreSQL plan generation (simplified)

FROM

↓

WHERE

↓

ORDER BY

↓

LIMIT

One or both will be pushed down to INDEXAM implementation

More on this later …

Pgvector implements INDEXAM
(Index Access Method Interface)

Core index functions:

ambuild
aminsert
ambeginscan
amrescan
amendscan
amgettuple
…

Vector Search

pgvector

PostgreSQL plan generation (simplified)

FROM

WHERE

ORDER BY

LIMIT

One or both will be pushed down to INDEXAM implementation

More on this later …

Pgvector implements INDEXAM (Index Access Method Interface)

Core index functions:

ambuild
aminsert
ambeginscan
amrescan
amendscan
amgettuple
…

PostgreSQL execution ROW based

Has to return *ItemPointerData* on each call (*xs_heaptid*)

Vector Search

sednai

[pgvector]

PostgreSQL plan generation (simplified)

FROM

WHERE

ORDER BY

LIMIT

One or both will be pushed down to INDEXAM implementation

More on this later …

Pgvector implements INDEXAM (Index Access Method Interface)

Core index functions:

ambuild
aminsert
ambeginscan
amrescan
amendscan
amgettuple
…

Pointer to an item within a disk page of a known file (block + offset)

PostgreSQL execution ROW based

Has to return *ItemPointerData* on each call (*xs_heaptid*)

Vector Search

sednai

pgvector

PostgreSQL plan generation (simplified)

FROM

WHERE

ORDER BY

LIMIT

One or both will be pushed down to INDEXAM implementation

More on this later …

Pgvector implements INDEXAM
(Index Access Method Interface)

Core index functions:

ambuild
aminsert
ambeginscan
amrescan
amendscan
amgettuple
…

First call:
Pgvector performs approx vector search.

Subsequent calls:
Found *ItemPointers* are returned.

PostgreSQL execution ROW based

Vector Search

[pgvector]

What about performance ?

Hardware:

- i9-13900H + RTX 4070 [35W]

Software:

- Ubuntu 22.04
- gcc/g++-14; icpx
- Python 3.10.12 + psycopg2 2.9.6
- Standard Postgres v15
  (no special compile flags besides -g)
- PGvector master on Mar 24, 2025
  (commit 05182479a2a62e04300386b4da18be02fcb819b5)
  (compiled with -O3 -march=native -g)

Vector Search

sednai

pgvector

What about performance ?

Settings:

- Queried locally via python+psycopg2
  (on persistent connection)
- IVFFLAT: 200 clusters
- Recall computation for IVFFLAT via variation of # probes.
  (Smallest # probes for fixed recall + median at fixed recall)
- Recall computation for HNSW via variation of *ef_search* parameter.
  (Smallest parameter for fixed recall + median at fixed recall)

sednai

[pgvector]

What about performance ?

Settings:

- Queried locally via python+psycopg2
  (on persistent connection)
- IVFFLAT: 200 clusters
- Recall computation for IVFFLAT via variation of # probes.
  (Smallest # probes for fixed recall + median at fixed recall)
- Recall computation for HNSW via variation of *ef_search* parameter.
  (Smallest parameter for fixed recall + median at fixed recall)

Variation in discrete steps

IVFFLAT: 10 probes given by cutting points of [1,100] into equal size ranges
HNSW: [100, 200,..., 1000]

Vector Search

sednai

[pgvector]

What about performance ?

GIST-960 dataset [Jégou, Douze and Schmid, 2011] (http://corpus-texmex.irisa.fr/)

- 1M vectors
- 960d
- 1k test vectors
- Pre-computed 100 nearest neighbors (euclidean metric)

(Vectors given by global GIST descriptors of image dataset. GIST summarizes the gradient information for different parts of an image.)

DEEP1B dataset [Babenko and Lempitsky, 2016] (https://www.tensorflow.org/datasets/catalog/deep1b)

- 10M vectors (subset of 1B)
- 96d
- 10k test vectors (we use first 1k)
- Pre-computed 100 nearest neighbors (cosine metric)

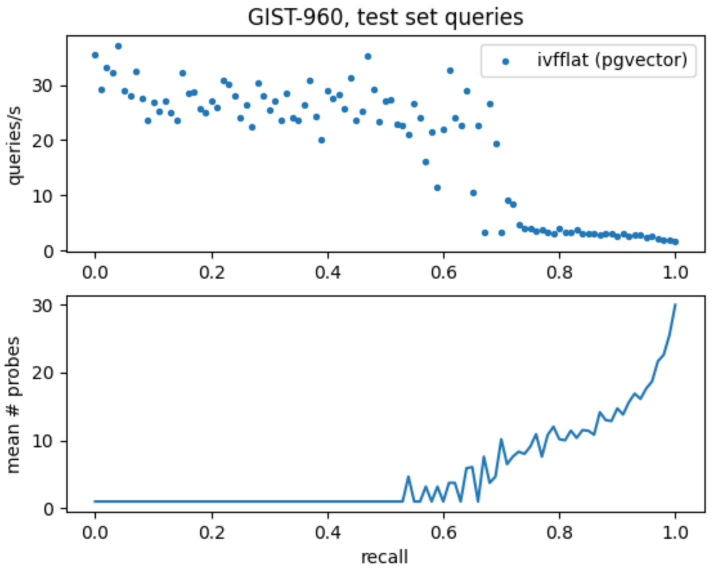
(Vectors given by PCA of 1B image embeddings produced as outputs from the last fully-connected layer of a GoogLeNet CNN model)

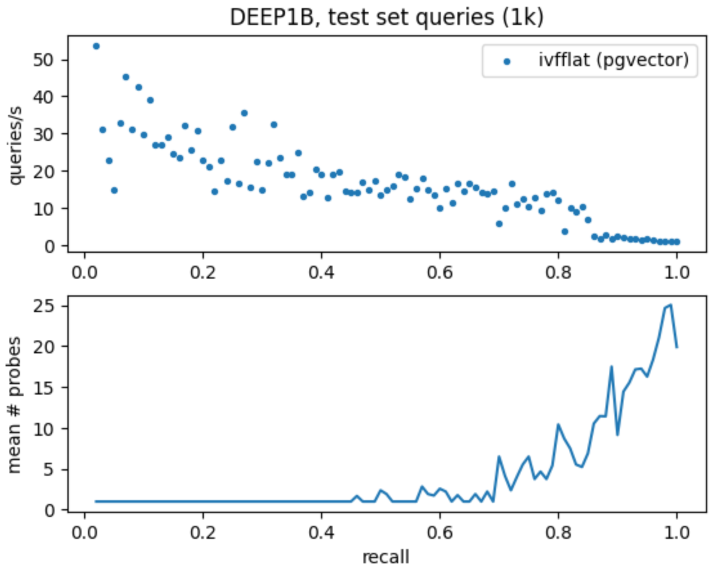
Vector Search

sednai

pgvector

What about performance ?





select id from gist order by embedding <-> "+q+" limit 100

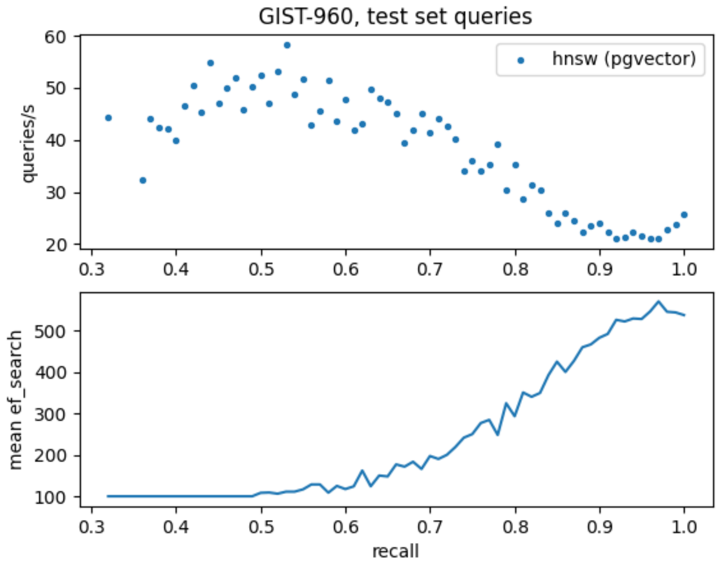
select id from deep1b order by embedding <=> "+q+" limit 100

Vector Search

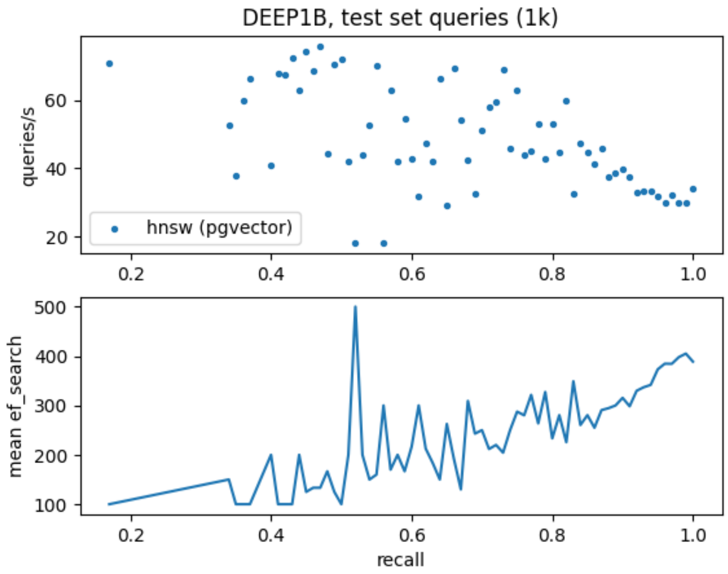
sednai

pgvector

What about performance ?



select id from gist order by embedding <-> "+q+" limit 100

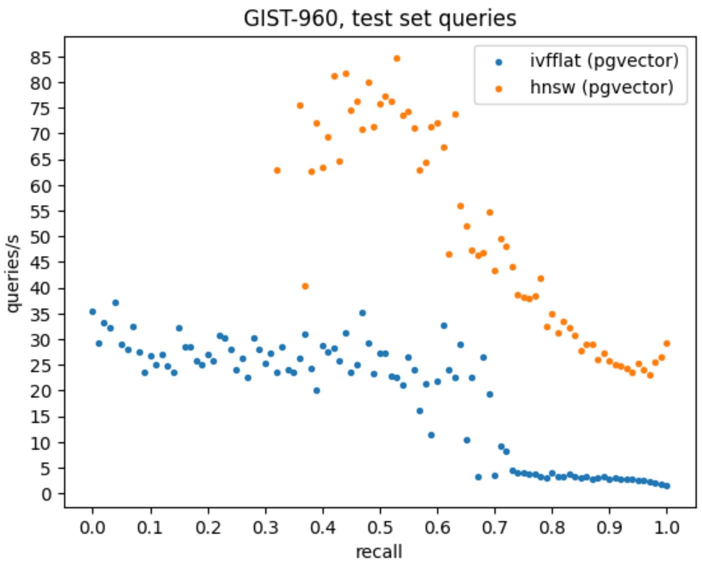


select id from deep1b order by embedding <=> "+q+" limit 100

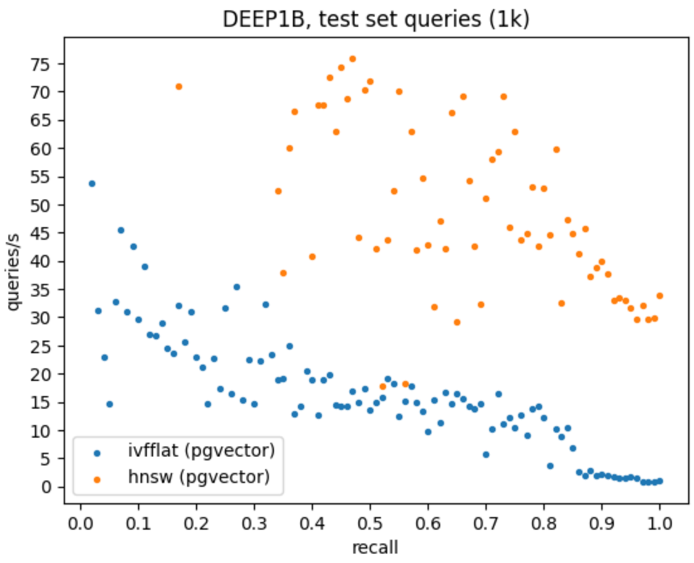
Vector Search

[pgvector]

What about performance ?



select id from gist order by embedding <-> "+q+" limit 100



select id from deep1b order by embedding <=> "+q+" limit 100

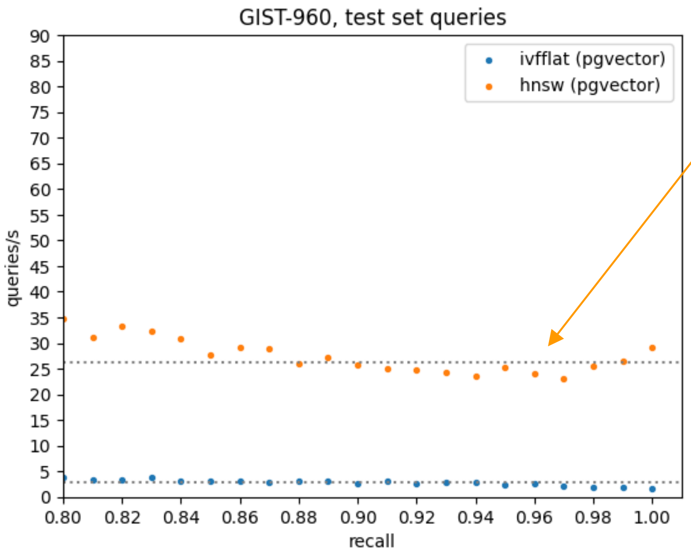
Vector Search

sednai

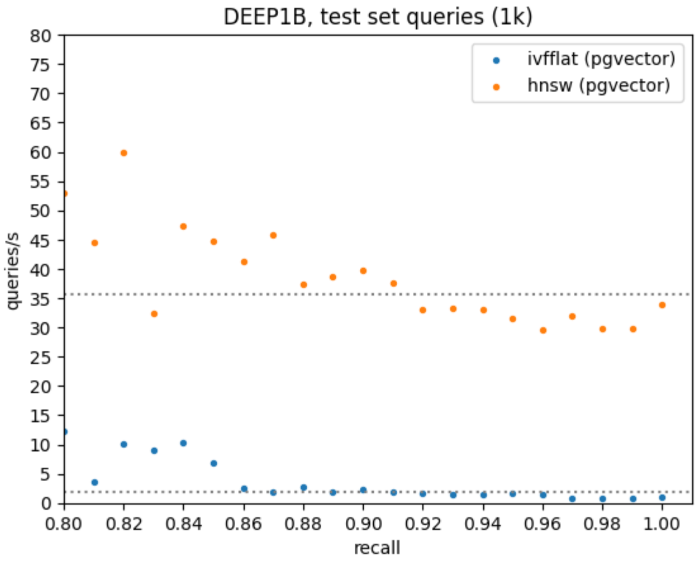
pgvector

What about performance ?

Median in range [0.8, 1.0]



~ 9x



~ 19x

select id from gist order by embedding <-> "+q+" limit 100

select id from deep1b order by embedding <=> "+q+" limit 100

Vector Search

[pgvector (ivfflat)]

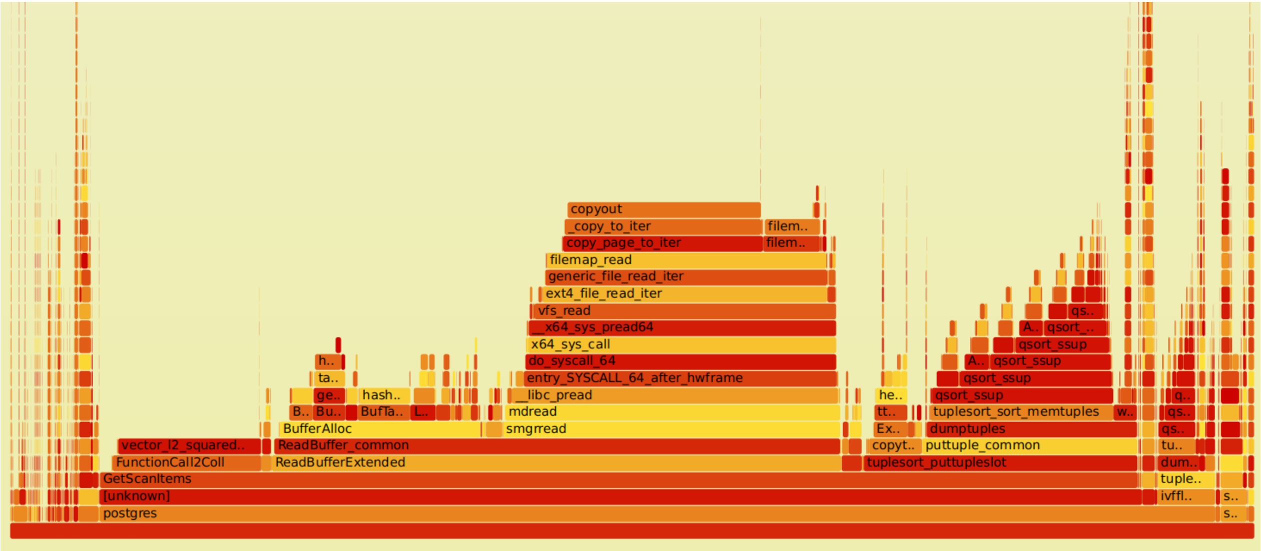
What about performance ?

GIST-960 dataset (1M, 960d)

Time thieves:

- ReadBuffers
- Tuplesort
- Distance calculation

Ivfflat inference perf analysis



[Generated with FlameGraph] (https://github.com/brendangregg/FlameGraph)

Vector Search

sednai

[pgvector (ivfflat)]

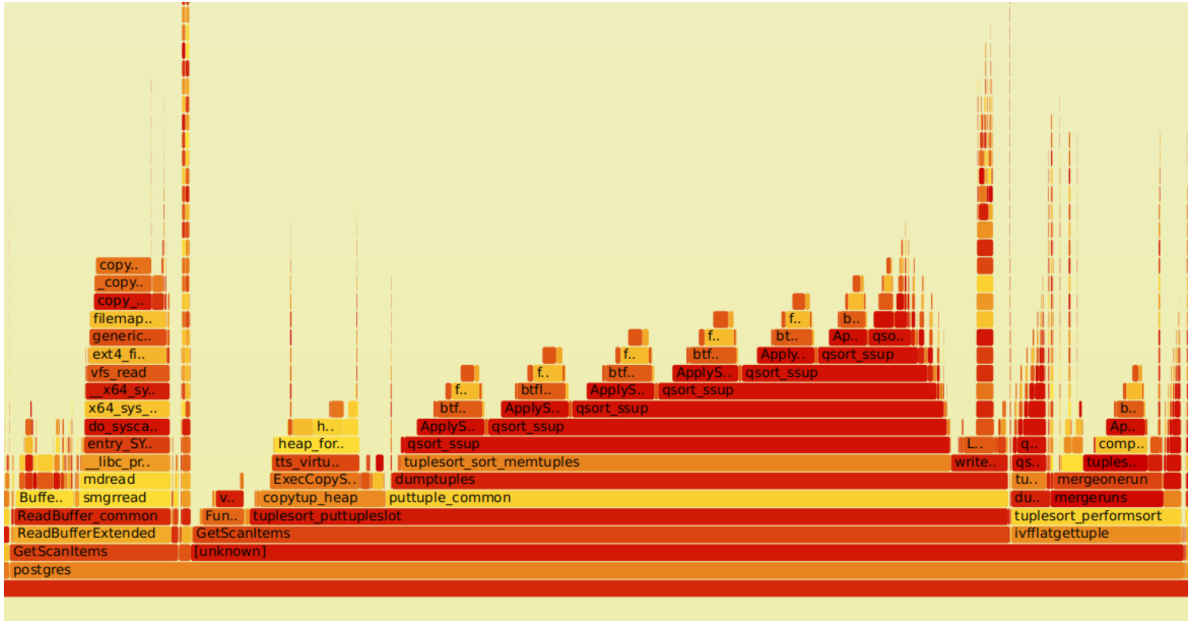
What about performance ?

DEEP-1B dataset (10M, 96d)

Time thieves:

- Tuplesort
- ReadBuffers
- Distance calculation

Ivfflat inference perf analysis



[Generated with FlameGraph] (https://github.com/brendangregg/FlameGraph)

Vector Search

sednai

[pgvector (ivfflat)]

What about performance ?

Time thieves:

- ReadBuffers
- Tuplesort
- Distance calculation

Portions Copyright (c) 1996-2024, PostgreSQL Global Development Group

Portions Copyright (c) 1994, The Regents of the University of California

Permission to use, copy, modify, and distribute this software and its documentation for any purpose, without fee, and without a written agreement is hereby granted, provided that the above copyright notice and this paragraph and the following two paragraphs appear in all copies.

IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING LOST PROFITS, ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF THE UNIVERSITY OF CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE SOFTWARE PROVIDED HEREUNDER IS ON AN "AS IS" BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATIONS TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

License of pgvector, 2025

```c
buf = ReadBufferExtended(scan->indexRelation, MAIN_FORKNUM, searchPage, RBM_NORMAL, so->bas);
LockBuffer(buf, BUFFER_LOCK_SHARE);
page = BufferGetPage(buf);
maxoffno = PageGetMaxOffsetNumber(page);

for (OffsetNumber offno = FirstOffsetNumber; offno <= maxoffno; offno = OffsetNumberNext(offno))
{
    IndexTuple  itup;
    Datum       datum;
    bool        isnull;
    ItemId      itemid = PageGetItemId(page, offno);

    itup = (IndexTuple) PageGetItem(page, itemid);
    datum = index_getattr(itup, 1, tupdesc, &isnull);

    /*
     * Add virtual tuple
     *
     * Use procinfo from the index instead of scan key for
     * performance
     */
    ExecClearTuple(slot);
    slot->tts_values[0] = so->distfunc(so->procinfo, so->collation, datum, value);
    slot->tts_isnull[0] = false;
    slot->tts_values[1] = PointerGetDatum(&itup->t_tid);
    slot->tts_isnull[1] = false;
    ExecStoreVirtualTuple(slot);

    tuplesort_puttupleslot(so->sortstate, slot);
}
```

All points in cluster(s) are read again from buffers on each call !

pgvector github, ivfscan.c, 2025

… HACKING PGVECTOR …

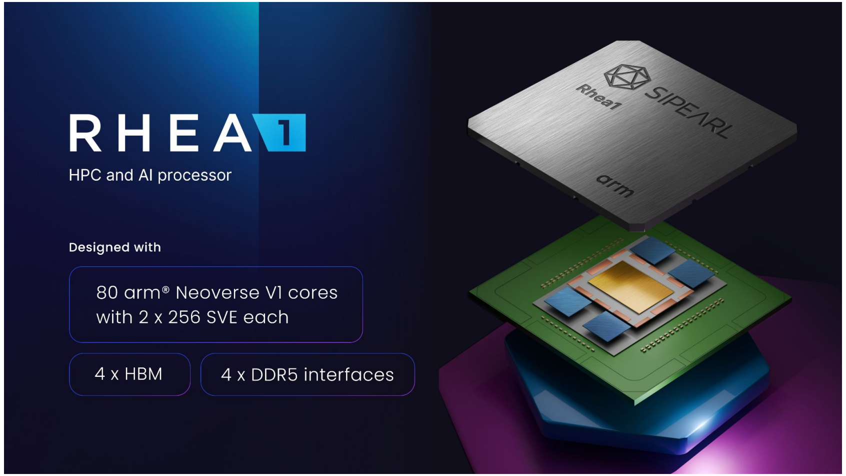
Vector Search

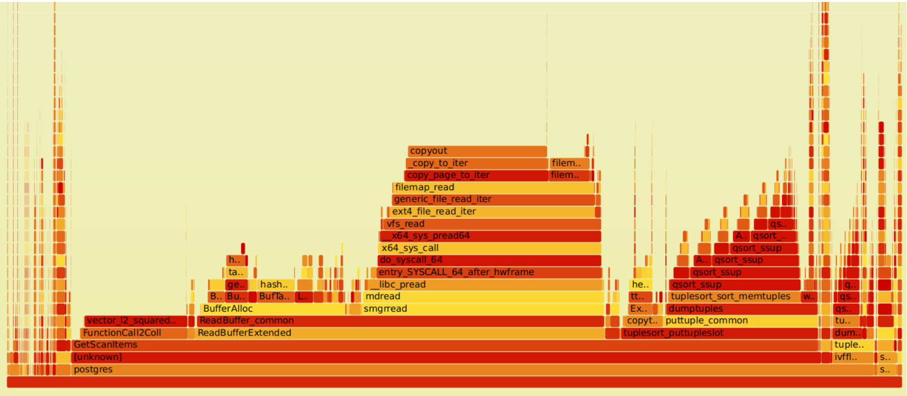
[pgvector hack (ivfflat)]()

Can we do better ? (https://github.com/Sednai/pgvector/tree/AERO_v2)

Keep index data persistent in Non-Postgres controlled memory:

- No Buffers but continuous arrays
- No TupleSort
- Possibility for better optimization for hardware





Vector Search

sednai

pgvector hack (ivfflat)

Can we do better ? (https://github.com/Sednai/pgvector/tree/AERO_v2)

Keep index data persistent in Non-Postgres controlled memory:

- No Buffers but continuous arrays
- No TupleSort
- Possibility for better optimization for hardware

→ *"In-memory vector database"*
(instead of in-memory buffer db)

Proof-of-concept
for
a next generation pgvector

sednai

pgvector hack (ivfflat)

Implementation

Own Postgres *background process* with task queue in shared memory.

```
BackgroundWorker worker;
BackgroundWorkerHandle *handle;
BgwHandleStatus status;
pid_t       pid;

memset(&worker, 0, sizeof(worker));
worker.bgw_flags = BGWORKER_SHMEM_ACCESS | BGWORKER_BACKEND_DATABASE_CONNECTION;
worker.bgw_start_time = BgWorkerStart_RecoveryFinished;
worker.bgw_restart_time = BGW_NEVER_RESTART; // Time in s to restart if crash. Use BGW_NEVER_RESTART for no restart;

char* WORKER_LIB = GetConfigOption("ivfflat.lib",true,true);

sprintf(worker.bgw_library_name, WORKER_LIB);
sprintf(worker.bgw_function_name, "pgv_gpuworker_main");

snprintf(worker.bgw_name, BGW_MAXLEN, "%s",buf);

worker.bgw_notify_pid = MyProcPid;

if (!RegisterDynamicBackgroundWorker(&worker, &handle))
    elog(ERROR,"Could not register background worker");

status = WaitForBackgroundWorkerStartup(handle, &pid);
```

Added bonus: Resources can be managed over all user sessions (via queuing system).

Vector Search

sednai

[pgvector hack (ivfflat)]

Implementation

Own Postgres *background process* with task queue in shared memory.

Re-routing the index tuple scan as task to background worker during scan:

```
/*
 * Fetch the next tuple in the given scan
 */
bool
ivfflatgettuple(IndexScanDesc scan, ScanDirection dir)
```

own process

```
/*
    Exec task
*/
dlist_node* dnode = dlist_pop_head_node(&worker_head->exec_list);
worker_exec_entry* entry = dlist_container(worker_exec_entry, node, dnode);

SpinLockRelease(&worker_head->lock);

load_index(entry->nodeid, entry->tupdesc, entry->usetriangle);

// Compute
if(!entry->usegpu) {
    entry->returns = exec_query_cpu(entry, worker_head);
}
else
```

sednai

[pgvector hack (ivfflat)]

Implementation

For an indexed table, we store the index vectors and corresponding location info (*ItemPointerData*) as raw native arrays in Non-Postgres memory.

(The data will be persistent over the lifetime of the background process. Have not implemented active memory management yet. Pgvector background process will crash if you run out of memory !)

```
/*
 * Fetch the next tuple in the given scan
 */
bool
ivfflatgettuple(IndexScanDesc scan, ScanDirection dir)
```

own process

```
/*
    Exec task
*/
dlist_node* dnode = dlist_pop_head_node(&worker_head->exec_list);
worker_exec_entry* entry = dlist_container(worker_exec_entry, node, dnode);

SpinLockRelease(&worker_head->lock);

load_index(entry->nodeid, entry->tupdesc, entry->usetriangle);

// Compute
if(!entry->usegpu) {
    entry->returns = exec_query_cpu(entry, worker_head);
}
else
```

C/C++

sednai

pgvector hack (ivfflat)

Implementation

For a query point (vector), we compute its *distance* against all index vectors and distance *sort*. Location infos are returned to the user process.
(More precisely, the corresponding page number and ItemPointerData are returned.)

```
/*
 * Fetch the next tuple in the given scan
 */
bool
ivfflatgettuple(IndexScanDesc scan, ScanDirection dir)
```

own process

```
/*
    Exec task
*/
dlist_node* dnode = dlist_pop_head_node(&worker_head->exec_list);
worker_exec_entry* entry = dlist_container(worker_exec_entry, node, dnode);

SpinLockRelease(&worker_head->lock);

load_index(entry->nodeid, entry->tupdesc, entry->usetriangle);

// Compute
if(!entry->usegpu) {
    entry->returns = exec_query_cpu(entry, worker_head);
}
else
```

C/C++

Vector Search

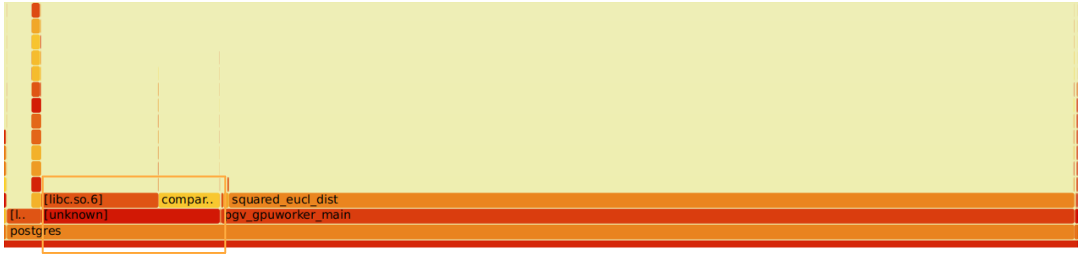
[pgvector hack (ivfflat)]

What about performance ?

Note:

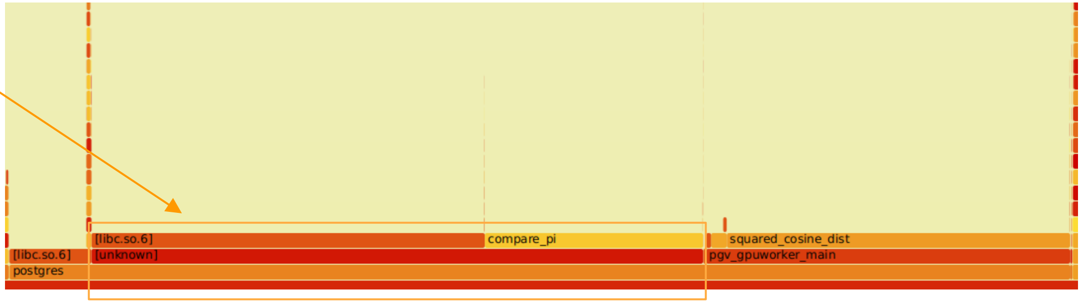
- No special tricks
- No multi-threading
- No manual vector instructions
- No HBM memory

GIST-960 inference perf analysis



std::qsort

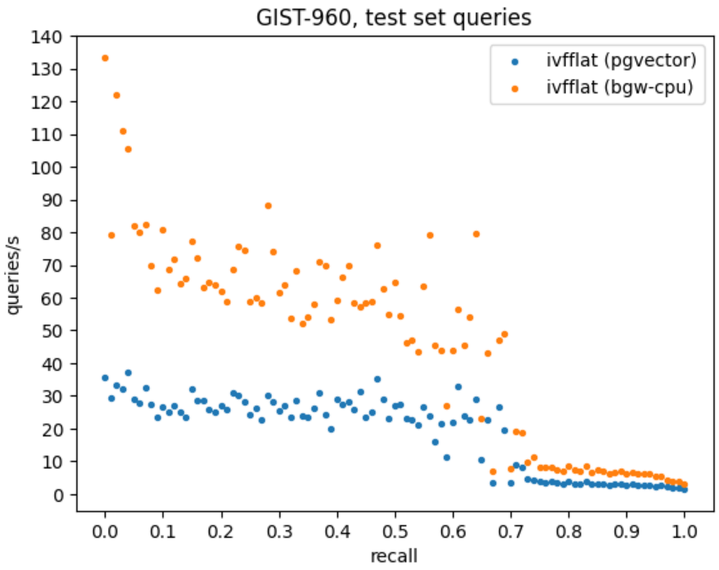
DEEP-1B inference perf analysis



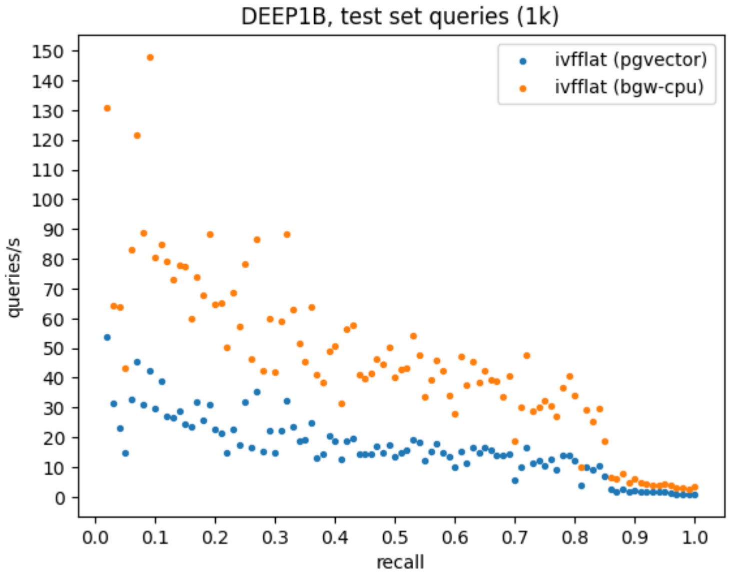
sednai

pgvector

What about performance ?



select id from gist order by embedding <-> "+q+" limit 100

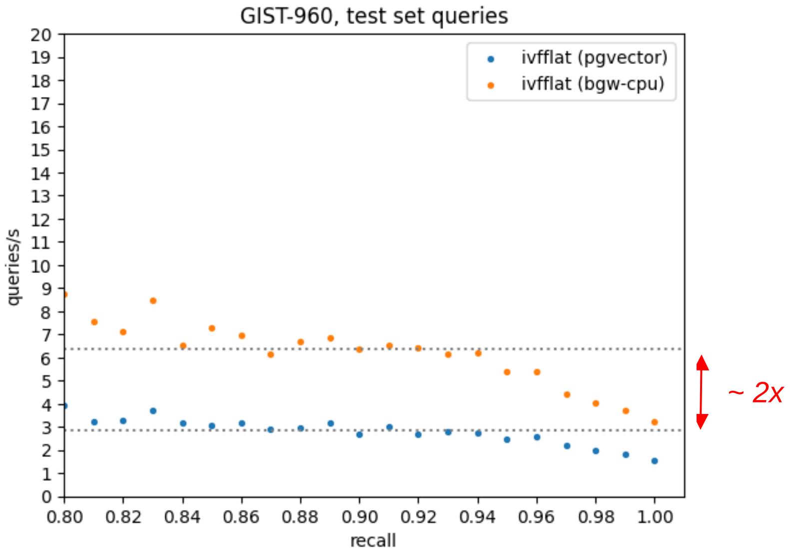


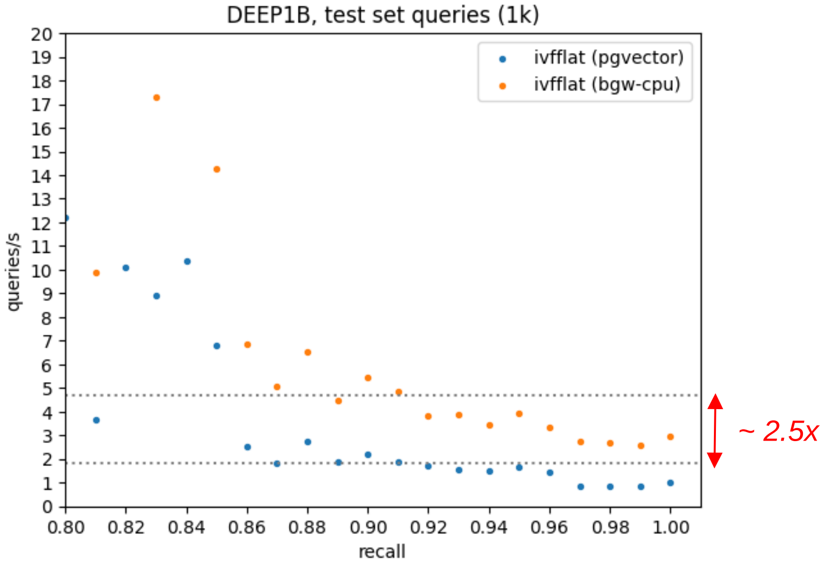
select id from deep1b order by embedding <=> "+q+" limit 100

Vector Search

pgvector

What about performance ?





select id from gist order by embedding <-> "+q+" limit 100

select id from deep1b order by embedding <=> "+q+" limit 100

Vector Search

sednai

[pgvector hack (ivfflat)]

Can we do better ?

Since we have already a setup to keep index persistent in Non-Postgres memory, we can easily go one step further and offload the index and compute to a GPU !

→ Compute of *distances* and *sort* on device.
Return only sorted index ids from device
(Mapping to location info on CPU)

Vector Search

sednai

pgvector hack (ivfflat)

Can we do better ?

Since we have already a setup to keep index persistent in Non-Postgres memory, we can easily go one step further and offload the index and compute to a GPU !

→ Compute of *distances* and *sort* on device.
Return only sorted index ids from device
(Mapping to location info on CPU)

Example: Nvidia A100

→ 80 GB in additional memory !
(for FP32 vectors of 100d that is enough to keep >200M index points persistent)

→ ~20 TFLOPS in FP32 compute power !

… GPU ACCELERATED VECTOR SEARCH …

sednai

pgvector hack (ivfflat)

Implementation

For an indexed table, we store now the index vectors on a GPU device.

(The data will be persistent over the lifetime of the background process. Have not implemented active memory management yet. Background process will crash if you run out of memory !)

```
/*
 * Fetch the next tuple in the given scan
 */
bool
ivfflatgettuple(IndexScanDesc scan, ScanDirection dir)
```

own process

```
/*
    Exec task
*/
dlist_node* dnode = dlist_pop_head_node(&worker_head->exec_list);
worker_exec_entry* entry = dlist_container(worker_exec_entry, node, dnode);

SpinLockRelease(&worker_head->lock);

load_index(entry->nodeid, entry->tupdesc, entry->usetriangle);

// Compute
if(!entry->usegpu) {
    entry->returns = exec_query_cpu(entry, worker_head);
}
else
```

C++ / CUDA
(essentially one big cudaMemcpy)

Vector Search

sednai

[pgvector hack (ivfflat)]

Implementation

For a query point (vector), we compute its *distance* against all index vectors and distance *sort* on device. Only ordered index ids are returned from GPU.

```
/*
 * Fetch the next tuple in the given scan
 */
bool
ivfflatgettuple(IndexScanDesc scan, ScanDirection dir)
```

own process

```
        load_index(entry->nodeid, entry->tupdesc, entry->usetriangle);

        // Compute
        if(!entry->usegpu) {
            entry->returns = exec_query_cpu(entry, worker_head);
        }
        else
#ifdef GPU
            entry->returns = exec_query_gpu(entry, worker_head);
#else
            entry->returns = exec_query_cpu(entry, worker_head);
#endif
```

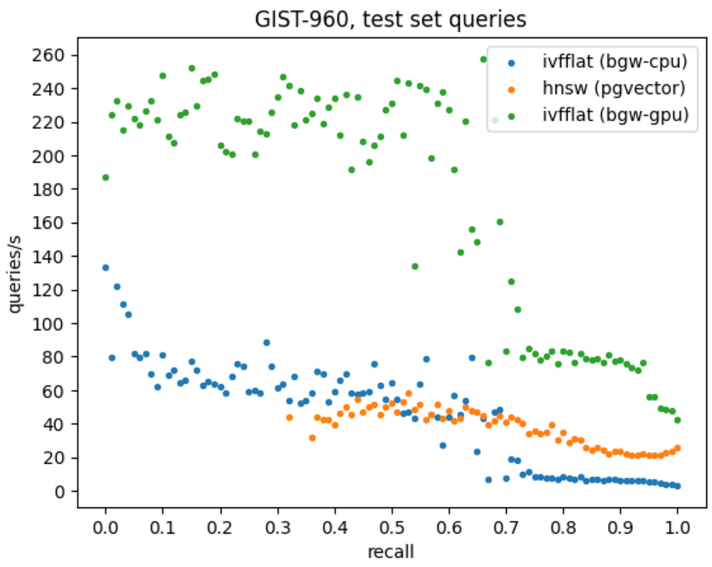
C++ / CUDA (custom kernels)

Vector Search

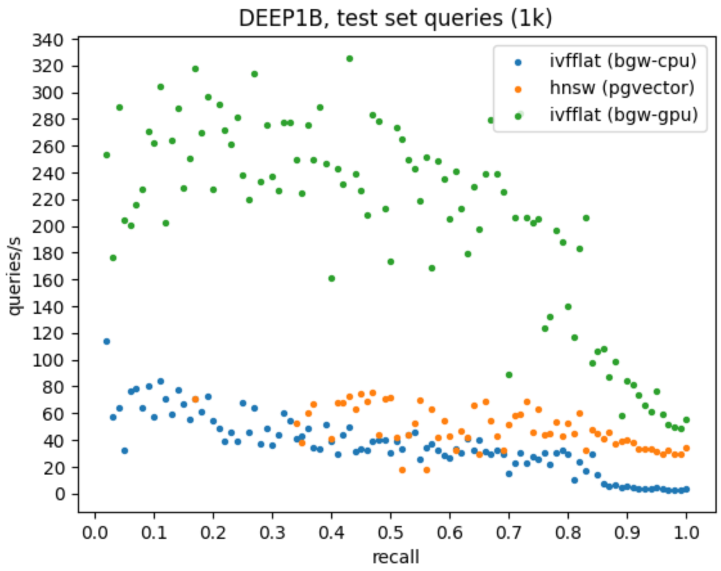
sednai

[pgvector hack (ivfflat)]

What about performance ?



select id from gist order by embedding <-> "+q+" limit 100

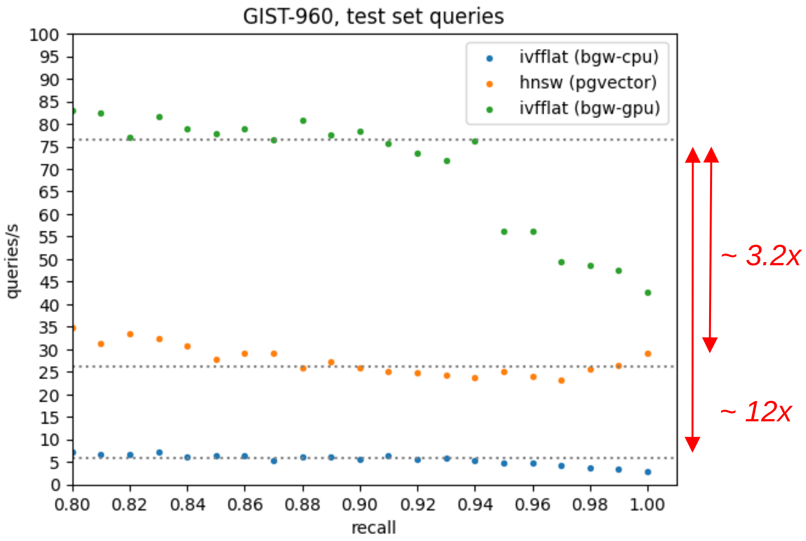


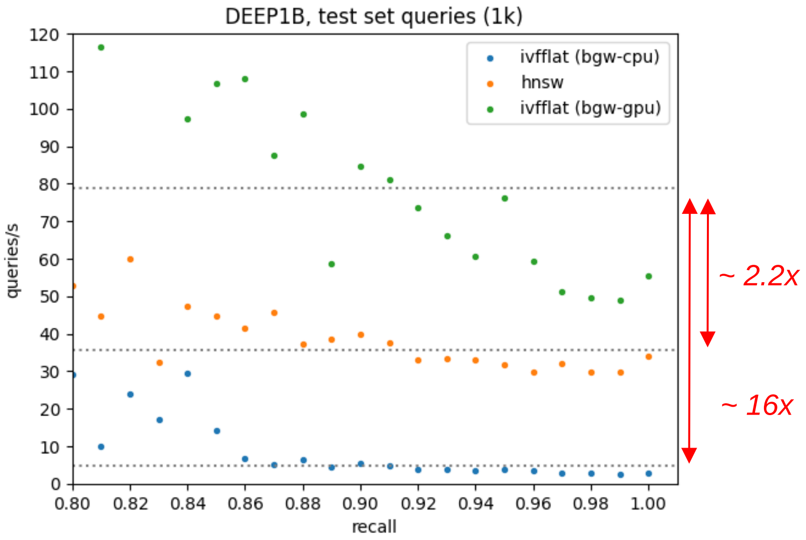
select id from deep1b order by embedding <=> "+q+" limit 100

sednai

pgvector hack (ivfflat)

What about performance ?





select id from gist order by embedding <-> "+q+" limit 100

select id from deep1b order by embedding <=> "+q+" limit 100

Vector Search

sednai

[pgvector hack]

Since we have now a basic vector search PG background worker:

Hack in Nvidia RAFT and cuVS library ?
(https://github.com/rapidsai/raft) (https://github.com/rapidsai/cuvs)

- Fullstack
  (HNSW, IFFLAT, CAGRA, DiskANN; Index build + inference)

- As GPU backend available for FAISS, Milvus, Redis and others …
  (according to https://big-ann-benchmarks.com/neurips23_slides/NVIDIA_Corey.pdf)

Vector Search

sednai

pgvector hack

Since we have now a basic vector search PG background worker:

Hack in Nvidia RAFT and cuVS library ?
(https://github.com/rapidsai/raft) (https://github.com/rapidsai/cuvs)

- Fullstack
  (HNSW, IFFLAT, CAGRA, DiskANN; Index build + inference)

- As GPU backend available for FAISS, Milvus, Redis and others …
  (according to https://big-ann-benchmarks.com/neurips23_slides/NVIDIA_Corey.pdf)

May be a low effort way to get something new and complete into PG …

BUT: Do we really want to become vendor dependent ?

oneAPI

Aero aims to complement the efforts of the *EU Processor Initiative (EPI)* project by developing the open-source software ecosystem required to not only improve the efficiency of the EPI hardware but also accelerate and ease the processor's integration into the cloud.

Heterogeneous hardware

PROBLEM: Divers set of accelerators from different vendors (NVIDIA, AMD, INTEL,...)

oneAPI

Aero aims to complement the efforts of the *EU Processor Initiative (EPI)* project by developing the open-source software ecosystem required to not only improve the efficiency of the EPI hardware but also accelerate and ease the processor's integration into the cloud.

Heterogeneous hardware

Open, cross-industry, standards-based, unified, multi-architecture, multi-vendor programming model, adopted by Intel.

oneAPI

sednai

Aero aims to complement the efforts of the *EU Processor Initiative (EPI)* project by developing the open-source software ecosystem required to not only improve the efficiency of the EPI hardware but also accelerate and ease the processor's integration into the cloud.

Heterogeneous hardware

oneAPI

Open, cross-industry, standards-based, unified, multi-architecture, multi-vendor programming model, adopted by Intel.

Intel oneAPI base toolkit plugins for NVIDIA and AMD

sednai

pgvector hack (ivfflat)

Implementation

For a query point (vector), we compute its *distance* against all index vectors and distance *sort* on device. Only ordered index ids are returned from GPU to CPU process.

```
/*
 * Fetch the next tuple in the given scan
 */
bool
ivfflatgettuple(IndexScanDesc scan, ScanDirection dir)
```

own process

```
        load_index(entry->nodeid, entry->tupdesc, entry->usetriangle);

        // Compute
        if(!entry->usegpu) {
            entry->returns = exec_query_cpu(entry, worker_head);
        }
        else
#ifdef GPU
            entry->returns = exec_query_gpu(entry, worker_head);
#else
            entry->returns = exec_query_cpu(entry, worker_head);
#endif
```

C++ / oneAPI (custom kernels)

Vector Search

sednai

pgvector hack (ivfflat)

Implementation

For a query point (vector), we compute its *distance* against all index vectors and distance *sort* on device. Only ordered index ids are returned from GPU to CPU process.

```
void calc_squared_euclidean_distances(float* M, float* V, sort_item* C, int* p, int N, int L, int probe) {

    Q->parallel_for(range<1>(N),
    [=](id<1> k){
        int pos = *p;

        float tmp = 0;
        for(int i = 0; i < L; i++) {
            tmp += (M[L*k+i] - V[i])*(M[L*k+i] - V[i]);
        }

        C[pos+k].distance = tmp;
        C[pos+k].probe = probe;
        C[pos+k].pos = k;

    });

    Q->wait();
}
```

oneAPI™

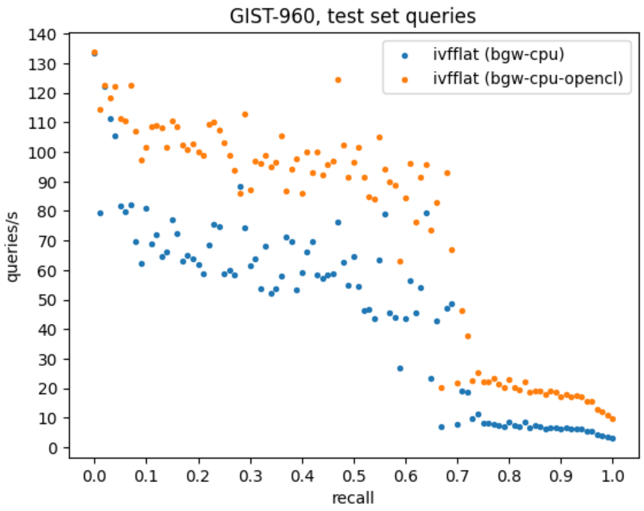
Vector Search

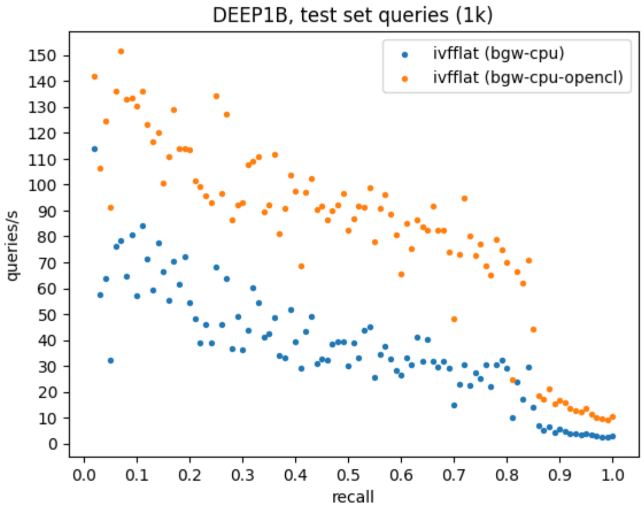
sednai

[pgvector hack (ivfflat)]

What about performance ?

With OpenCL CPU oneAPI backend (all cores)



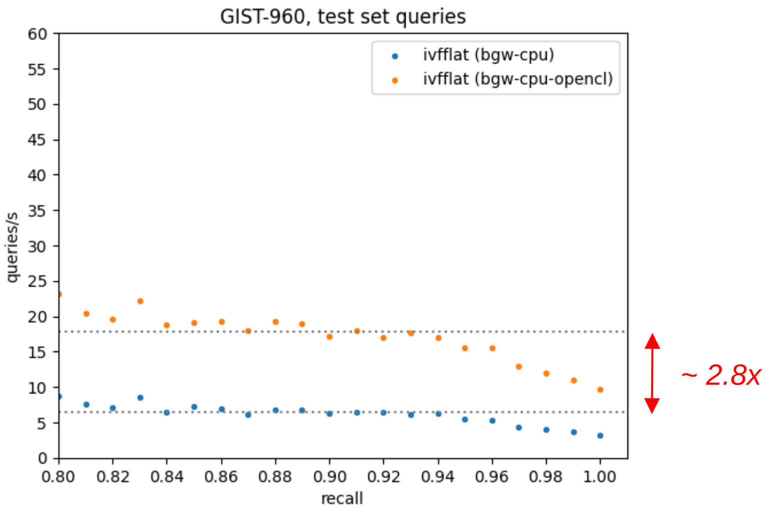


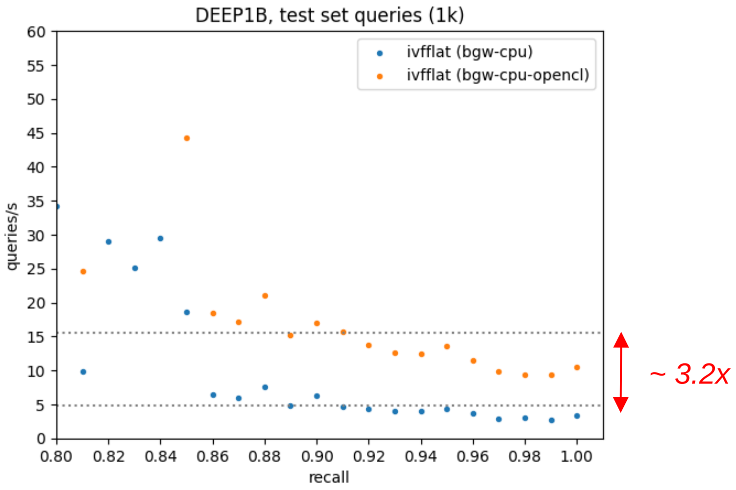
Vector Search

[pgvector hack (ivfflat)]

What about performance ?

With OpenCL CPU oneAPI backend (all cores)



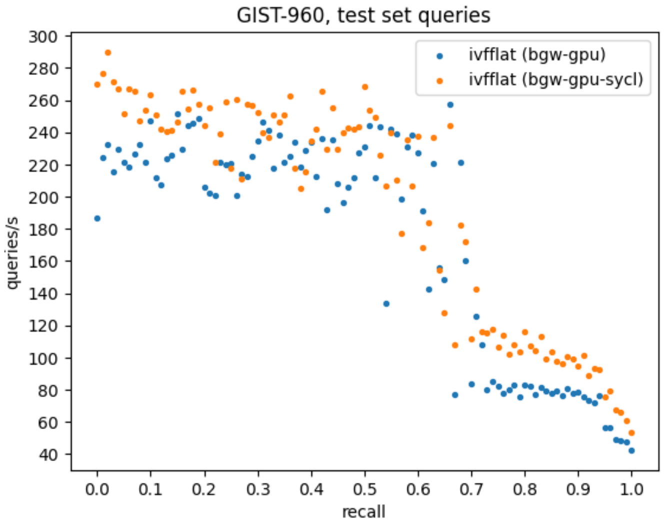


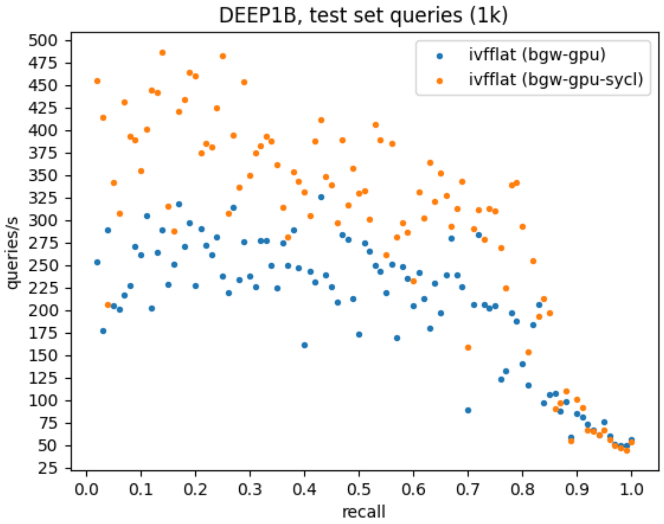
sednai

[pgvector hack (ivfflat)]

What about performance ?

With Nvidia GPU oneAPI backend





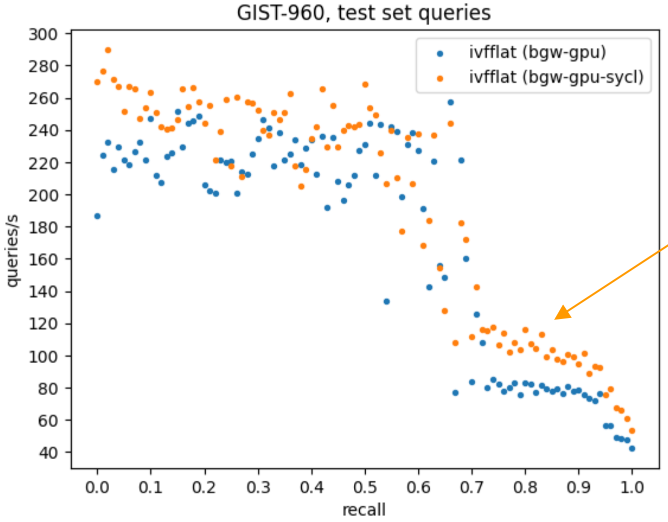
sednai

pgvector hack (ivfflat)

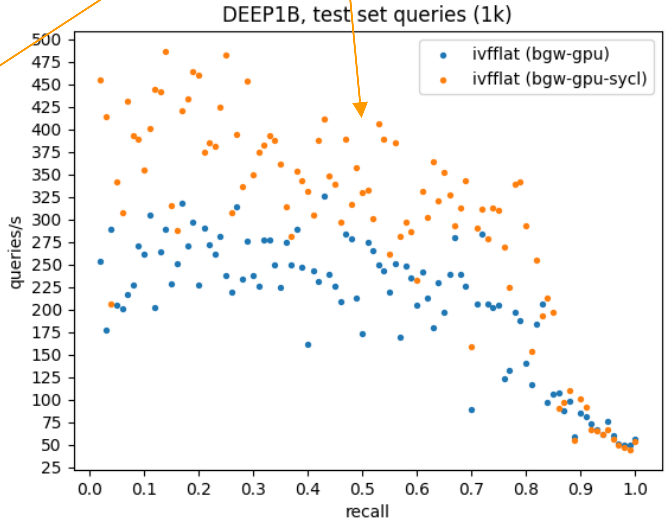
What about performance ?

Seems the automatic kernels are not bad …
Artefact of using unified memory ?

With Nvidia GPU oneAPI backend



~ 1.25x



… A SPECIAL CASE …

Vector Search

sednai

pgvector

What we want:

Fast way to retrieve (most) points up to a max distance from a query point.

Vector Search

[**pgvector**]()

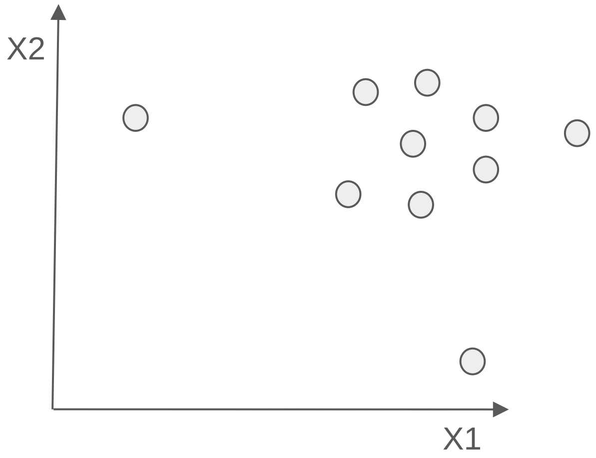
What we want:

Fast way to retrieve (most) points up to a max distance from a query point.

Why ?

Core ingredient to density based clustering algorithms.

Simplified:



Vector Search

[pgvector]

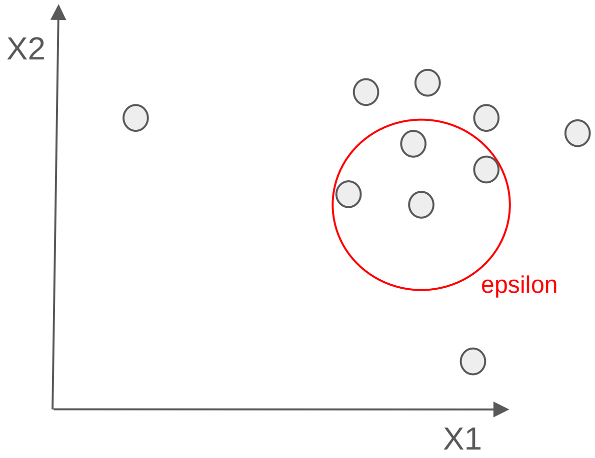
What we want:

Fast way to retrieve (most) points up to a max distance from a query point.

Why ?

Core ingredient to density based clustering algorithms.

Simplified:



Vector Search

pgvector

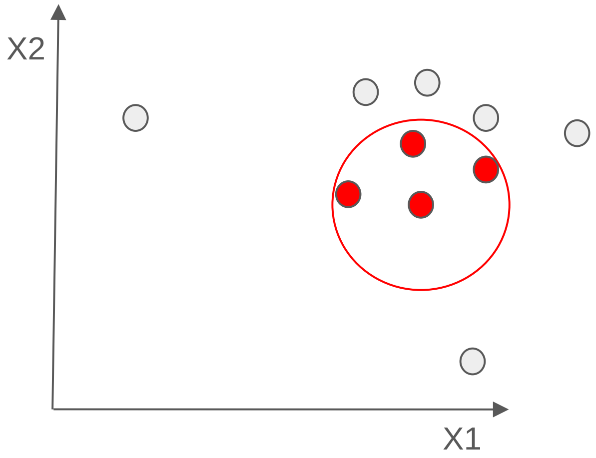
What we want:

Fast way to retrieve (most) points up to a max distance from a query point.

Why ?

Core ingredient to density based clustering algorithms.

Simplified:



Vector Search

pgvector

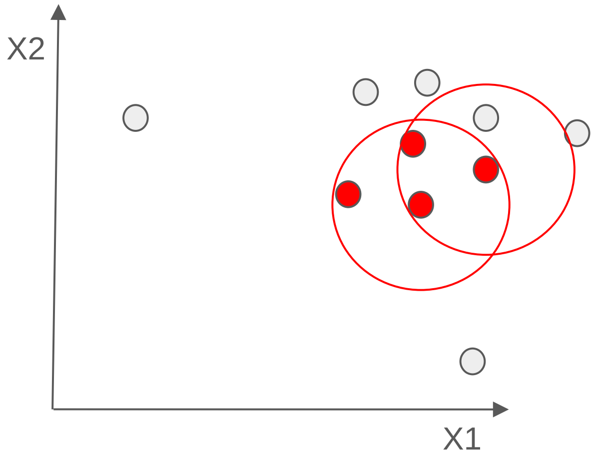
What we want:

Fast way to retrieve (most) points up to a max distance from a query point.

Why ?

Core ingredient to density based clustering algorithms.

Simplified:



Vector Search

sednai

pgvector

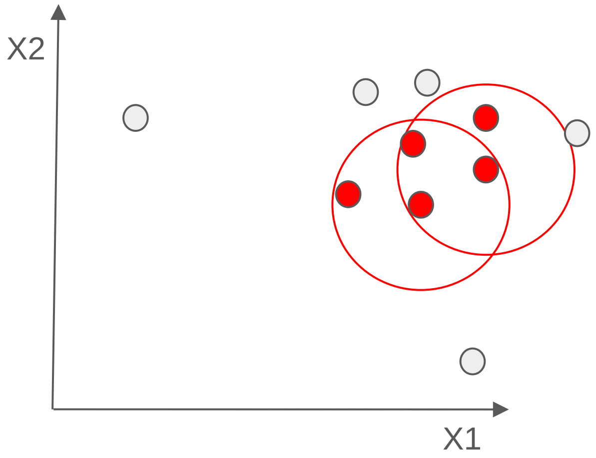
What we want:

Fast way to retrieve (most) points up to a max distance from a query point.

Why ?

Core ingredient to density based clustering algorithms.

Simplified:



Vector Search

sednai

[pgvector]

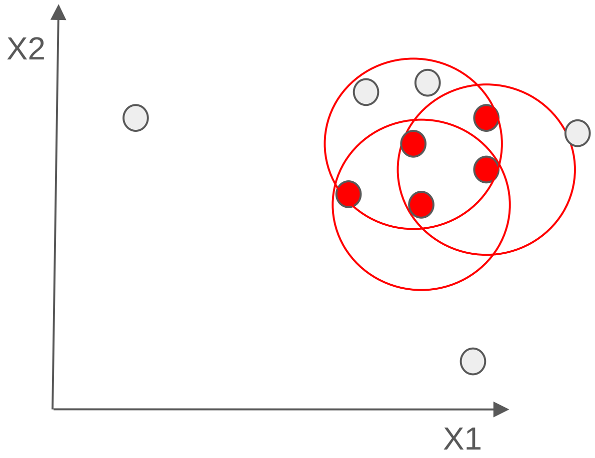
What we want:

Fast way to retrieve (most) points up to a max distance from a query point.

Why ?

Core ingredient to density based clustering algorithms.

Simplified:



Vector Search

[pgvector]

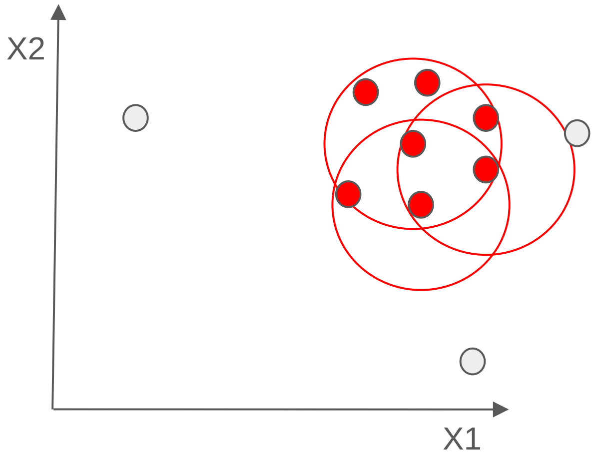
What we want:

Fast way to retrieve (most) points up to a max distance from a query point.

Why ?

Core ingredient to density based clustering algorithms.

Simplified:



Vector Search

[pgvector]

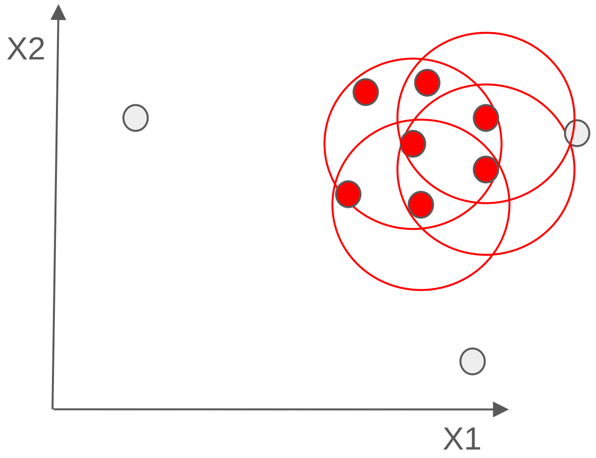
What we want:

Fast way to retrieve (most) points up to a max distance from a query point.

Why ?

Core ingredient to density based clustering algorithms.

Simplified:



sednai

pgvector

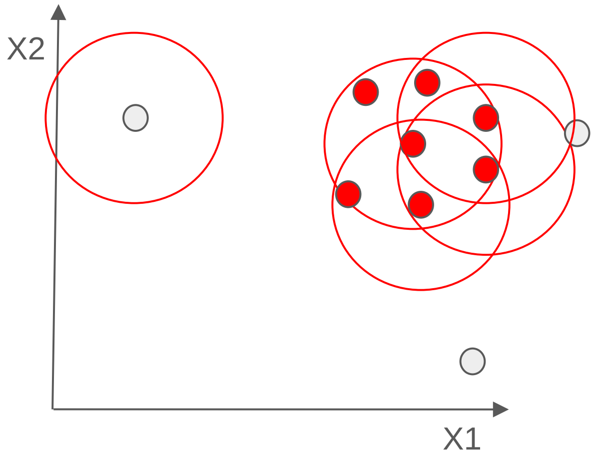
What we want:

Fast way to retrieve (most) points up to a max distance from a query point.

Why ?

Core ingredient to density based clustering algorithms.

Simplified:



Vector Search

[pgvector]

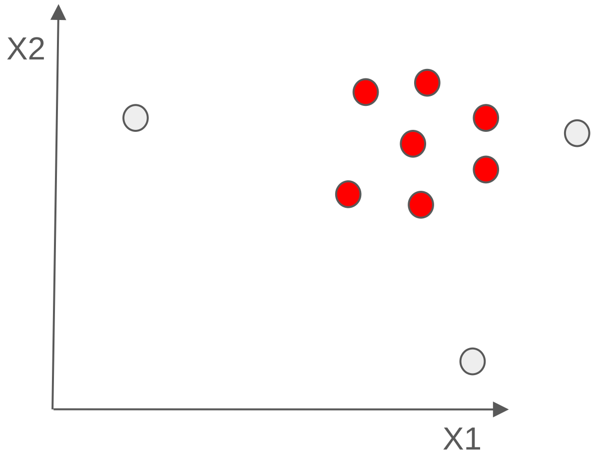
What we want:

Fast way to retrieve (most) points up to a max distance from a query point.

Why ?

Core ingredient to density based clustering algorithms.

Simplified:



Vector Search

pgvector

What we want:

Fast way to retrieve (most) points up to a max distance from a query point.

Get rows within a certain distance

```
SELECT * FROM items WHERE embedding <-> '[3,1,2]' < 5;
```

Note: Combine with `ORDER BY` and `LIMIT` to use an index

(From pgvector github README.md, 2025)

sednai

pgvector

What we want:

Fast way to retrieve (most) points up to a max distance from a query point.

Get rows within a certain distance

Operator for L2 distance

```
SELECT * FROM items WHERE embedding <-> '[3,1,2]' < 5;
```

Note: Combine with `ORDER BY` and `LIMIT` to use an index

(From pgvector github README.md, 2025)

Need quite large limit !

Vector Search

pgvector

What we want:

Fast way to retrieve (most) points up to a max distance from a query point.

Get rows within a certain distance

```
SELECT * FROM items WHERE embedding <-> '[3,1,2]' < 5;
```

Note: Combine with `ORDER BY` and `LIMIT` to use an index

(From pgvector github README.md, 2025)

Need quite large limit !

PROBLEM: Very very slow …

sednai

pgvector

What about performance ?

- 2M row Gaia dataset, 79 float8 features
- 40 ivfflat clusters, ivfflat.probes = 20
- Retrieve points up to a max distance from a query point (sparse return)
- After warmup

Original pgvector

```
 Limit  (cost=0.00..9416.48 rows=10000 width=16) (actual time=113.688..563.274 rows=2 loops=1)
   ->  Index Scan using lorenzo_attrs_idx on lorenzo  (cost=0.00..569379.89 rows=604663 width=16) (actual time=113.685..563.269 rows=2 loops=1)
         Order By: (attrs <-> '[-0.36719987,0.8524608,-0.5427666,-1.6615063,1.1010165,0.06038815,-0.8972259,-1.1181297,0.23769811,1.6662519,-1.5
473,-0.042955805,0.17839357,0.08050123,0.27791676,-0.425645,0.11280374,0.84778684,0.08167486,1.8496727,0.8007245,0.5793525,-0.5038844,0.2299031
,0.12975255,-1.9553635,0.9473572,-2.2433414,-0.30360684,0.33857238,-0.21312521,2.3237233,-0.060708717,0.339421,-2.0196183,-0.35616732,1.8636712,
749731,-1.5635711,-1.2368783,-0.96010184,-0.6722383,0.8274576,-0.7714504,-0.16363333,-0.96023947,-0.16326201,-1.0754527,-0.6974341,-2.3611114,-1
03672114,1.2685906,-0.22232309,-0.17129573,-0.30436236,-1.1221358,0.6857615,-0.60302067,0.22385728,-1.0727525,-1.6519747,-0.9824103,-1.5251932,-
1835048,-2.293227,1.9016955,-2.8030064,-0.045054823,-0.14567287]'::vector)
         Filter: ((attrs <-> '[-0.36719987,0.8524608,-0.5427666,-1.6615063,1.1010165,0.06038815,-0.8972259,-1.1181297,0.23769811,1.6662519,-1.55
73,-0.042955805,0.17839357,0.08050123,0.27791676,-0.425645,0.11280374,0.84778684,0.08167486,1.8496727,0.8007245,0.5793525,-0.5038844,0.22990316,
0.12975255,-1.9553635,0.9473572,-2.2433414,-0.30360684,0.33857238,-0.21312521,2.3237233,-0.060708717,0.339421,-2.0196183,-0.35616732,1.8636712,2
49731,-1.5635711,-1.2368783,-0.96010184,-0.6722383,0.8274576,-0.7714504,-0.16363333,-0.96023947,-0.16326201,-1.0754527,-0.6974341,-2.3611114,-1.
3672114,1.2685906,-0.22232309,-0.17129573,-0.30436236,-1.1221358,0.6857615,-0.60302067,0.22385728,-1.0727525,-1.6519747,-0.9824103,-1.5251932,-2
835048,-2.293227,1.9016955,-2.8030064,-0.045054823,-0.14567287]'::vector) < '6'::double precision)
         Rows Removed by Filter: 578577
 Planning Time: 0.204 ms
 Execution Time: 563.326 ms
(7 rows)

pgv2=#
```

Vector Search

sednai

pgvector

Why ?

Let us look with GDB what actually happens for a query of type

*select * from table where embedding <-> '[...]' < 10 order by embedding <-> '[...]' limit 10000*

by setting a breakpoint at ivfscan.c:ivfflatgettuple :

```
(gdb) break ivfscan.c:353
Breakpoint 1 at 0x7c106d843e7b: file src/ivfscan.c, line 358.
(gdb) c
Continuing.

Breakpoint 1, ivfflatgettuple (scan=0x60700e9406f8, dir=ForwardScanDirection) at src/ivfscan.c:360
360             if (so->first)
(gdb) p* scan
$1 = {heapRelation = 0x7c106d7925e8, indexRelation = 0x7c106d796818, xs_snapshot = 0x60700e8a8a68, numberOfKeys = 0, numberOfOrderBys = 1, keyData = 0x0,
  orderByData = 0x60700e940808, xs_want_itup = false, xs_temp_snap = false, kill_prior_tuple = false, ignore_killed_tuples = true, xactStartedInRecovery = false,
  opaque = 0x60700e940898, xs_itup = 0x0, xs_itupdesc = 0x0, xs_hitup = 0x0, xs_hitupdesc = 0x0, xs_heaptid = {ip_blkid = {bi_hi = 0, bi_lo = 8680}, ip_posid = 4},
  xs_heap_continue = false, xs_heapfetch = 0x60700e9409a8, xs_recheck = false, xs_orderbyvals = 0x0, xs_orderbynulls = 0x0, xs_recheckorderby = false,
  parallel_scan = 0x0}
(gdb)
```

Vector Search

pgvector

Why ?

Let us look with GDB what actually happens for a query of type

*select * from table where embedding <-> ‘[...]’ < 10 order by embedding <-> ‘[...]’ limit 10000*

by setting a breakpoint at ivfscan.c:ivfflatgettuple :

```
(gdb) break ivfscan.c:353
Breakpoint 1 at 0x7c106d843e7b: file src/ivfscan.c, line 358.
(gdb) c
Continuing.

Breakpoint 1, ivfflatgettuple (scan=0x60700e9406f8, dir=ForwardScanDirection) at src/ivfscan.c:360
360             if (so->first)
(gdb) p* scan
$1 = {heapRelation = 0x7c106d7925e8, indexRelation = 0x7c106d796818, xs_snapshot = 0x60700e8a8a68, numberOfKeys = 0, numberOfOrderBys = 1, keyData = 0x0,
  orderByData = 0x60700e940808, xs_want_itup = false, xs_temp_snap = false, kill_prior_tuple = false, ignore_killed_tuples = true, xactStartedInRecovery = false,
  opaque = 0x60700e940898, xs_itup = 0x0, xs_itupdesc = 0x0, xs_hitup = 0x0, xs_hitupdesc = 0x0, xs_heaptid = {ip_blkid = {bi_hi = 0, bi_lo = 8680}, ip_posid = 4},
  xs_heap_continue = false, xs_heapfetch = 0x60700e9409a8, xs_recheck = false, xs_orderbyvals = 0x0, xs_orderbynulls = 0x0, xs_recheckorderby = false,
  parallel_scan = 0x0}
(gdb)
```

IndexScanDesc

No scan keys are pushed down !

Vector Search

sednai

[pgvector]

ExecInitBuildScanKeys: quals are Null

Why no scan keys ?

iss_NumScanKeys = 0 already in IndexScanState

We have to dig deeper …

```
(gdb) bt
#0  ivfflatgettuple (scan=0x60700e9406f8, dir=ForwardScanDirection) at src/ivfscan.c:360
#1  0x000060700c44b319 in index_getnext_tid (scan=0x60700e9406f8, direction=ForwardScanDirection) at indexam.c:575
#2  0x000060700c44b529 in index_getnext_slot (scan=0x60700e9406f8, direction=ForwardScanDirection, slot=0x60700e9586d0) at indexam.c:667
#3  0x000060700c6b234d in IndexNextWithReorder (node=0x60700e958430) at nodeIndexscan.c:264
#4  0x000060700c68ad18 in ExecScanFetch (node=0x60700e958430, accessMtd=0x60700c6b2158 <IndexNextWithReorder>, recheckMtd=0x60700c6b2675 <IndexRecheck>)
    at execScan.c:132
#5  0x000060700c68adbd in ExecScan (node=0x60700e958430, accessMtd=0x60700c6b2158 <IndexNextWithReorder>, recheckMtd=0x60700c6b2675 <IndexRecheck>) at execScan.c:198
#6  0x000060700c6b2b73 in ExecIndexScan (pstate=0x60700e958430) at nodeIndexscan.c:533
#7  0x000060700c686cce in ExecProcNodeFirst (node=0x60700e958430) at execProcnode.c:464
#8  0x000060700c6b5207 in ExecProcNode (node=0x60700e958430) at ../../../src/include/executor/executor.h:262
#9  0x000060700c6b53f7 in ExecLimit (pstate=0x60700e958140) at nodeLimit.c:96
#10 0x000060700c686cce in ExecProcNodeFirst (node=0x60700e958140) at execProcnode.c:464
#11 0x000060700c67b20a in ExecProcNode (node=0x60700e958140) at ../../../src/include/executor/executor.h:262
#12 0x000060700c67dac5 in ExecutePlan (queryDesc=0x60700e965c68, operation=CMD_SELECT, sendTuples=true, numberTuples=0, direction=ForwardScanDirection,
    dest=0x60700e9531e0) at execMain.c:1640
#13 0x000060700c67b71c in standard_ExecutorRun (queryDesc=0x60700e965c68, direction=ForwardScanDirection, count=0, execute_once=false) at execMain.c:362
#14 0x000060700c67b621 in ExecutorRun (queryDesc=0x60700e965c68, direction=ForwardScanDirection, count=0, execute_once=false) at execMain.c:311
#15 0x000060700c8b9b66 in PortalRunSelect (portal=0x60700e8f4328, forward=true, count=0, dest=0x60700e9531e0) at pquery.c:922
#16 0x000060700c8b97d1 in PortalRun (portal=0x60700e8f4328, count=9223372036854775807, isTopLevel=true, run_once=true, dest=0x60700e9531e0, altdest=0x60700e9531e0,
```

=> Already before execution level no scan keys !

Vector Search

sednai

pgvector

Why no scan keys ?

We have to dig deeper …

Let us look into indxpath.c:

```
/*-------------------------------------------------------------------------
 *
 * indxpath.c
 *    Routines to determine which indexes are usable for scanning a
 *    given relation, and create Paths accordingly.
 *
 * Portions Copyright (c) 1996-2022, PostgreSQL Global Development Group
 * Portions Copyright (c) 1994, Regents of the University of California
 *
 *
 * IDENTIFICATION
 *    src/backend/optimizer/path/indxpath.c
 *
```

```
/*
 * match_clause_to_indexcol()
 *    Determine whether a restriction clause matches a column of an index,
 *    and if so, build an IndexClause node describing the details.
 *
 *    To match an index normally, an operator clause:
 *
 *    (1)  must be in the form (indexkey op const) or (const op indexkey);
 *         and
 *    (2)  must contain an operator which is in the index's operator family
 *         for this column; and
 *    (3)  must match the collation of the index, if collation is relevant.
 *
```

Vector Search

[pgvector]()

Why no scan keys ?

We have to dig deeper …

Let us look into indxpath.c:

Query:

*select * from table where embedding <-> '[...]' < 10 …*

```
/*
 * match_clause_to_indexcol()
 *    Determine whether a restriction clause matches a column of an index,
 *    and if so, build an IndexClause node describing the details.
 *
 *    To match an index normally, an operator clause:
 *
 *    (1)  must be in the form (indexkey op const) or (const op indexkey);
 *         and
 *    (2)  must contain an operator which is in the index's operator family
 *         for this column; and
 *    (3)  must match the collation of the index, if collation is relevant.
 *
```

Indexkey can not be matched !

op(indexkey, '[...']) op const

Vector Search

sednai

pgvector

Why no scan keys ?

We have to dig deeper …

Let us look into indxpath.c:

Query:

*select * from table where embedding <-> '[...]' < 10 …*

→ *Indexkey can not be matched !*

op(indexkey, '[...']) op const

=> Looks like Postgres enhancement required !

BUT: May take ages to get upstream …

Vector Search

sednai

pgvector hack

Quicker to production:

Let us introduce a new operator, thereby hacking the *WHERE* into the *ORDER BY*:

New operator for `WHERE` clause directly in index scan (only for euclidean metric so far):

```
vector <!> vector_adv
```

with

```
vector_adv = (vector,int,float,int)
```

`int` specifies the filter operator, `float` the condition value

```
2: >=
1: >
0: ==
-1: <
-2: <=
-100: no filter
```

Vector Search

pgvector hack

Quicker to production:

Let us introduce a new operator, thereby hacking the *WHERE* into the *ORDER BY*:

New operator for `WHERE` clause directly in index scan (only for euclidean metric so far):

```
vector <!> vector_adv
```

with

Will come back to the last *int* later in this talk

```
vector_adv = (vector,int,float,int)
```

`int` specifies the filter operator, `float` the condition value

```
2: >=
1: >
0: ==
-1: <
-2: <=
-100: no filter
```

Vector Search

sednai

pgvector hack

Quicker to production:

Let us introduce a new operator, thereby hacking the *WHERE* into the *ORDER BY*:

Query:

*select * from table order by embedding <!> (‘[...]’, -1, 10.0, 0) limit 10000*

Vector Search

sednai

pgvector hack

Quicker to production:

Let us introduce a new operator, thereby hacking the *WHERE* into the *ORDER BY*:

Query:

*select * from table order by embedding <!> $\boxed{\text{('[...]', -1, 10.0, 0)}}$ limit 10000*

Will be evaluated inside of pgvector !

-> Can also be executed on GPU !

Vector Search

sednai

[pgvector hack]

What about performance ?

- 2M row Gaia dataset, 79 float8 features
- 40 ivfflat clusters, ivfflat.probes = 20
- Retrieve points up to a max distance from a query point (sparse return)
- After warmup

BGW with filter on CPU

```
-----------------------------------------------------------------------------------------
 Limit  (cost=0.00..3105.50 rows=10000 width=16) (actual time=18.928..18.932 rows=1 loops=1)
   ->  Index Scan using lorenzo_attrs_idx on lorenzo  (cost=0.00..563333.26 rows=1813988 width=16) (actual time=18.925..18.927 rows=1 loops=1)
         Order By: (attrs <!> '("[-0.36719987,0.8524608,-0.5427666,-1.6615063,1.1010165,0.06038815,-0.8972259,-1.1181297,0.23769811,1.6662519,-1
18473,-0.042955805,0.17839357,0.08050123,0.27791676,-0.425645,0.11280374,0.84778684,0.08167486,1.8496727,0.8007245,0.5793525,-0.5038844,0.229903
75,0.12975255,-1.9553635,0.9473572,-2.2433414,-0.30360684,0.33857238,-0.21312521,2.3237233,-0.060708717,0.339421,-2.0196183,-0.35616732,1.863671
.0749731,-1.5635711,-1.2368783,-0.96010184,-0.6722383,0.8274576,-0.7714504,-0.16363333,-0.96023947,-0.16326201,-1.0754527,-0.6974341,-2.3611114,
0.03672114,1.2685906,-0.22232309,-0.17129573,-0.30436236,-1.1221358,0.6857615,-0.60302067,0.22385728,-1.0727525,-1.6519747,-0.9824103,-1.525193
1.1835048,-2.293227,1.9016955,-2.8030064,-0.045054823,-0.14567287]",-1,4)'::vector_adv)
 Planning Time: 0.188 ms
 Execution Time: 18.980 ms
(5 rows)

pgv2=#
```

Vector Search

sednai

pgvector hack

What about performance ?

- 2M row Gaia dataset, 79 float8 features
- 40 ivfflat clusters, ivfflat.probes = 20
- Retrieve points up to a max distance from a query point (sparse return)
- After warmup

BGW with filter on GPU

```
-----------------------------------------------------------------------------------------
 Limit  (cost=0.00..3105.50 rows=10000 width=16) (actual time=2.482..2.488 rows=1 loops=1)
   ->  Index Scan using lorenzo_attrs_idx on lorenzo  (cost=0.00..563333.26 rows=1813988 width=16) (actual time=2.478..2.482 rows=1 loops=1)
         Order By: (attrs <!> '("[-0.36719987,0.8524608,-0.5427666,-1.6615063,1.1010165,0.06038815,-0.8972259,-1.1181297,0.23769811,1.6662519,-1
18473,-0.042955805,0.17839357,0.08050123,0.27791676,-0.425645,0.11280374,0.84778684,0.08167486,1.8496727,0.8007245,0.5793525,-0.5038844,0.229903
75,0.12975255,-1.9553635,0.9473572,-2.2433414,-0.30360684,0.33857238,-0.21312521,2.3237233,-0.060708717,0.339421,-2.0196183,-0.35616732,1.863671
.0749731,-1.5635711,-1.2368783,-0.96010184,-0.6722383,0.8274576,-0.7714504,-0.16363333,-0.96023947,-0.16326201,-1.0754527,-0.6974341,-2.3611114,
0.03672114,1.2685906,-0.22232309,-0.17129573,-0.30436236,-1.1221358,0.6857615,-0.60302067,0.22385728,-1.0727525,-1.6519747,-0.9824103,-1.525193
1.1835048,-2.293227,1.9016955,-2.8030064,-0.045054823,-0.14567287]",-1,4)'::vector_adv)
 Planning Time: 0.199 ms
 Execution Time: 2.548 ms
(5 rows)

pgv2=#
```

Vector Search

sednai

[pgvector hack]()

What about performance ?

- 2M row Gaia dataset, 79 float8 features
- 40 ivfflat clusters, ivfflat.probes = 20
- Retrieve points up to a max distance from a query point (sparse return)
- After warmup

BGW with filter on GPU

```
------------------------------------------------------------------------------------------
 Limit  (cost=0.00..3105.50 rows=10000 width=16) (actual time=2.482..2.488 rows=1 loops=1)
   ->  Index Scan using lorenzo_attrs_idx on lorenzo  (cost=0.00..563333.26 rows=1813988 width=16) (actual time=2.478..2.482 rows=1 loops=1)
         Order By: (attrs <!> '("[-0.36719987,0.8524608,-0.5427666,-1.6615063,1.1010165,0.06038815,-0.8972259,-1.1181297,0.23769811,1.6662519,-1
18473,-0.042955805,0.17839357,0.08050123,0.27791676,-0.425645,0.11280374,0.84778684,0.08167486,1.8496727,0.8007245,0.5793525,-0.5038844,0.229903
75,0.12975255,-1.9553635,0.9473572,-2.2433414,-0.30360684,0.33857238,-0.21312521,2.3237233,-0.060708717,0.339421,-2.0196183,-0.35616732,1.863671
.0749731,-1.5635711,-1.2368783,-0.96010184,-0.6722383,0.8274576,-0.7714504,-0.16363333,-0.96023947,-0.16326201,-1.0754527,-0.6974341,-2.3611114,
0.03672114,1.2685906,-0.22232309,-0.17129573,-0.30436236,-1.1221358,0.6857615,-0.60302067,0.22385728,-1.0727525,-1.6519747,-0.9824103,-1.525193
1.1835048,-2.293227,1.9016955,-2.8030064,-0.045054823,-0.14567287]",-1,4)'::vector_adv)
 Planning Time: 0.199 ms
 Execution Time: 2.548 ms
(5 rows)

pgv2=#
```

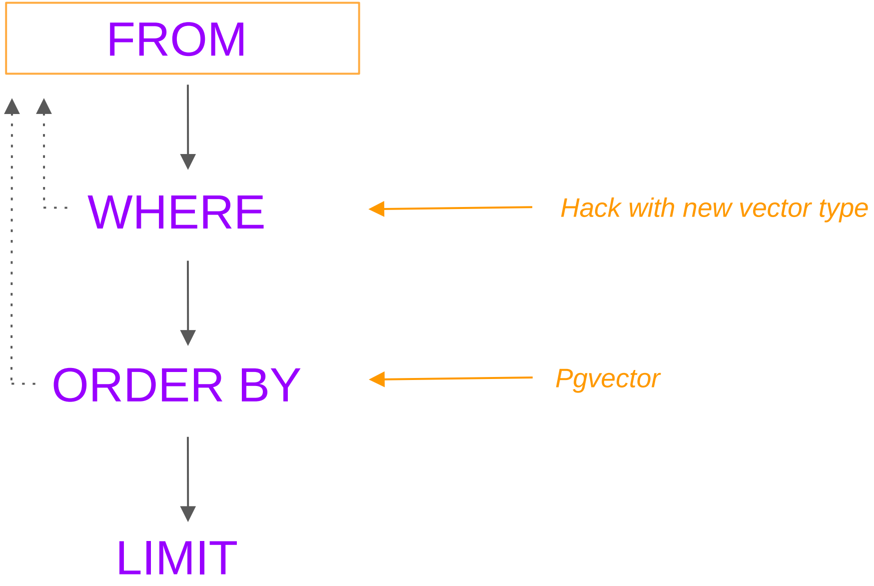
=> 200x speedup !

… LIMIT TRICKS …

Vector Search

[pgvector hack](#)

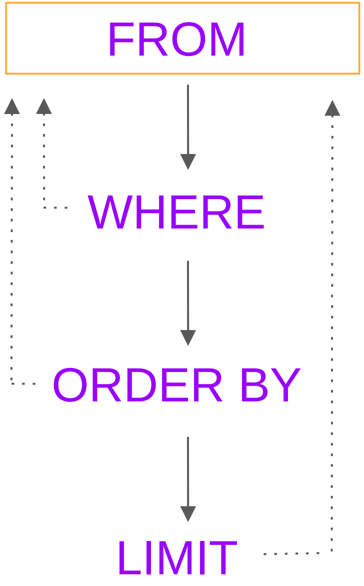
PostgreSQL plan generation (simplified)



Vector Search

[pgvector hack]

PostgreSQL plan generation (simplified)



New operator for `WHERE` clause directly in index scan (only for euclidean metric so far):

```
vector <!> vector_adv
```

with

Limit push down

```
vector_adv = (vector,int,float,int)
```

`int` specifies the filter operator, `float` the condition value

```
2: >=
1: >
0: ==
-1: <
-2: <=
-100: no filter
```

Hack with new vector type
(WARNING: Not fail safe if you do not know what you are doing !)

Vector Search

sednai

pgvector hack

Why do I want to do that ?

FROM

WHERE

ORDER BY

LIMIT

We can do instead of a full sort a partial sort (kth-element) + a small sort !

→ O(N) + O(k log k) scaling instead of O(N log N)

CPU: `std::nth_element`
OneAPI: `oneapi::dpl::nth_element`

Nvidia: Missing. Instead: `raft::matrix::select_k`

Hack with new vector type
(WARNING: Not fail safe if you do not know what you are doing !)

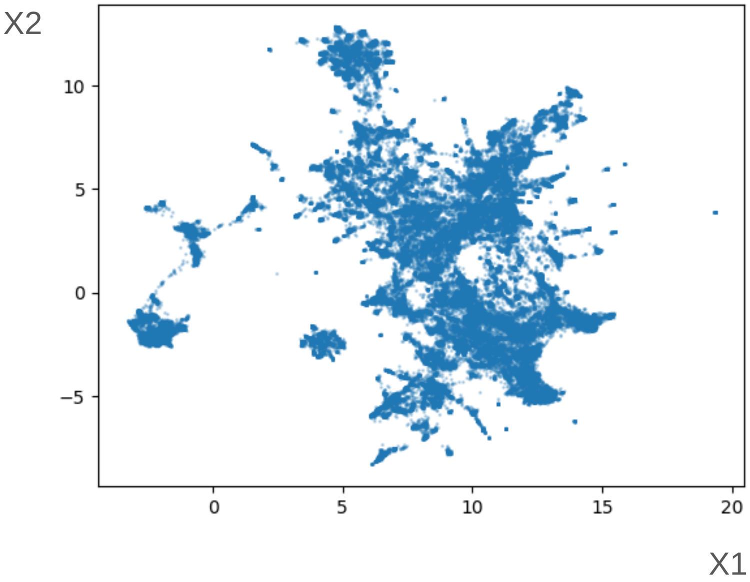
Vector Search

sednai

pgvector hack

Example:

Wikipedia - OpenAI embeddings
(~220k vectors; 1536d)



PGVECTOR (index with 200 ivfflat clusters)

```
pgv=# set ivfflat.probes = 20;
SET
pgv=# explain analyze select id from wiki order by vec <-> (select vec from wiki limit 1) limit 20;
                                          QUERY PLAN
--------------------------------------------------------------------------------------------------------------------
 Limit  (cost=0.09..56.84 rows=20 width=16) (actual time=68.925..69.208 rows=20 loops=1)
   InitPlan 1 (returns $0)
     ->  Limit  (cost=0.00..0.09 rows=1 width=18) (actual time=0.015..0.016 rows=1 loops=1)
           ->  Seq Scan on wiki wiki_1  (cost=0.00..19949.82 rows=224482 width=18) (actual time=0.014..0.014 rows=1 loops=1)
   ->  Index Scan using wiki_vec_idx on wiki  (cost=0.00..637017.14 rows=224482 width=16) (actual time=68.922..69.202 rows=20 loops=1)
         Order By: (vec <-> $0)
 Planning Time: 0.253 ms
 Execution Time: 69.402 ms
(8 rows)

pgv=#
```

~ 1.7x

BGW-CPU

```
pgv=# set ivfflat.bgw = 1;
SET
pgv=# explain analyze select id from wiki order by vec <!> ((select vec from wiki limit 1),-100,0,0) limit 20;
                                          QUERY PLAN
--------------------------------------------------------------------------------------------------------------------
 Limit  (cost=0.09..56.84 rows=20 width=16) (actual time=39.741..40.176 rows=20 loops=1)
   InitPlan 1 (returns $0)
     ->  Limit  (cost=0.00..0.09 rows=1 width=18) (actual time=0.027..0.028 rows=1 loops=1)
           ->  Seq Scan on wiki wiki_1  (cost=0.00..19949.82 rows=224482 width=18) (actual time=0.025..0.026 rows=1 loops=1)
   ->  Index Scan using wiki_vec_idx on wiki  (cost=0.00..637017.14 rows=224482 width=16) (actual time=39.737..40.167 rows=20 loops=1)
         Order By: (vec <!> ROW(($0)::vector, '-100'::integer, '0'::real, 0))
 Planning Time: 0.432 ms
 Execution Time: 40.277 ms
(8 rows)

pgv=#
```

~ 1.25x

~ 2.2x

BGW-CPU with LMT

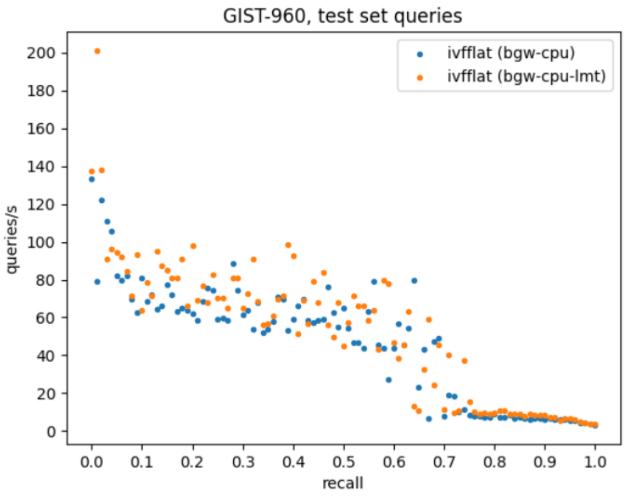
```
pgv=# explain analyze select id from wiki order by vec <!> ((select vec from wiki limit 1),-100,0,20) limit 20;
                                          QUERY PLAN
--------------------------------------------------------------------------------------------------------------------
 Limit  (cost=0.09..56.84 rows=20 width=16) (actual time=31.550..31.887 rows=20 loops=1)
   InitPlan 1 (returns $0)
     ->  Limit  (cost=0.00..0.09 rows=1 width=18) (actual time=0.007..0.008 rows=1 loops=1)
           ->  Seq Scan on wiki wiki_1  (cost=0.00..19949.82 rows=224482 width=18) (actual time=0.007..0.007 rows=1 loops=1)
   ->  Index Scan using wiki_vec_idx on wiki  (cost=0.00..637017.14 rows=224482 width=16) (actual time=31.548..31.882 rows=20 loops=1)
         Order By: (vec <!> ROW(($0)::vector, '-100'::integer, '0'::real, 20))
 Planning Time: 0.124 ms
 Execution Time: 31.934 ms
(8 rows)

pgv=#
```

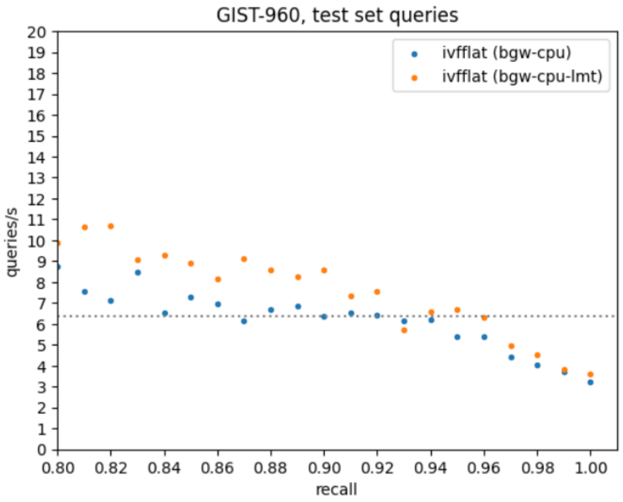
Vector Search

[pgvector hack]

Example:

GIST-960 with CPU-BGW backend





… REMARKS & OUTLOOK …

Vector Search

sednai

[pgvector hack]

General remarks:

- Proof-of-concept
- No active memory management
  (memory freed only upon killing the worker)
- Enough shared memory needs to be reserved for number of expected returns
- As more sparse the return, as better will be the speedup

Vector Search

[pgvector hack]

Can we do more ?

- Improvements of code (GPU kernels) are possible.
- Faster initial loading via Nvidia GPUDirect (NVMe <-> GPU DMA)
- Product quantization
- Multi-threading on the worker level
- For significant performance improvement, more *vectorization* …
  (for instance, to query for several points at once)
- Port other algorithms (HNSW, DiskANN, …)
- Use Nvidia RAFT and cuVS instead ?!

sednai

… THANK YOU …

(feedback link)

Funded by the European Union

UK Research and Innovation

Project funded by

Schweizerische Eidgenossenschaft
Confédération suisse
Confederazione Svizzera
Confederaziun svizra

Swiss Confederation

Federal Department of Economic Affairs,
Education and Research EAER
State Secretariat for Education,
Research and Innovation SERI

Funded by the European Union. Views and opinions are however those of the author(s) only and do not necessarily reflect those of the European Union or the HaDEA. Neither the European Union nor the granting authority can be held responsible for them. Project number: 101092850.

AERO has also received funding from UKRI under grants no. 10048318 and 10048915, and the Swiss State Secretariat for Education, Research, and Innovation.