CYBERTEC
POSTGRESQL SERVICES & SUPPORT

# PostgreSQL for AI

Vectors today, and a way for your models to attach straight to the database's memory

**Hans-Jürgen Schönig — CYBERTEC**
www.cybertec-postgresql.com

# What we will cover

From what you can do today to what we are releasing

**01** **PostgreSQL already does AI**
Vectors in the database you already run.

**02** **A short look at PGEE GPU**
One commercial idea, in a few minutes.

**03** **What we are open sourcing**
The direct attach: a protocol and a client.

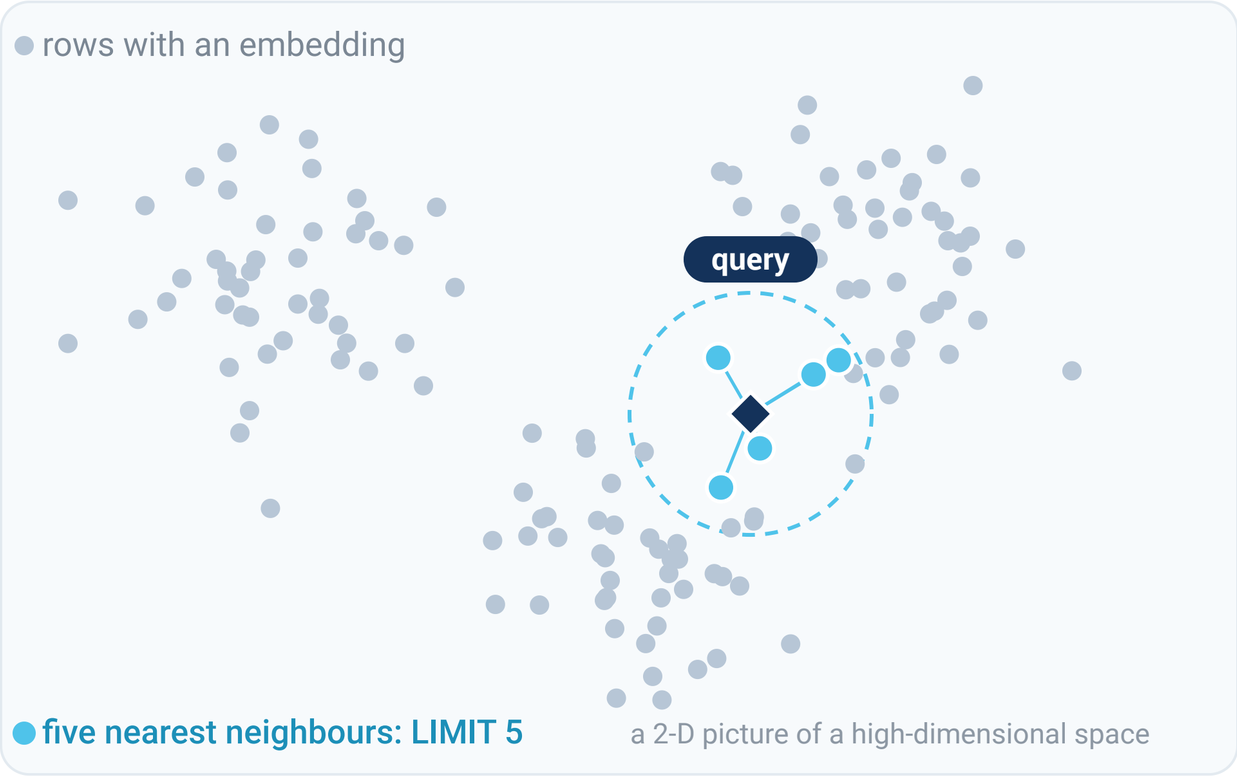
**04** **How a process attaches**
Five steps, and no copy of the data.

# PostgreSQL already does AI

Vectors, in the database you already run

# Similarity search is a query

pgvector turns “nearest” into an ORDER BY




## Embeddings are a column

Every row carries a vector. The question becomes a vector too, and “nearest” is a distance you sort by.

| | |
|---|---|
| `<->` | Euclidean distance |
| `<=>` | cosine distance |
| `<#>` | negative inner product |

Thank you to Andrew Kane and the pgvector contributors.

# Embeddings are a column

CYBERTEC
POSTGRESQL SERVICES & SUPPORT

pgvector

```
1.  CREATE EXTENSION vector;
2.
3.  CREATE TABLE docs (
4.      id         bigserial PRIMARY KEY,
5.      lang       text,
6.      body       text,
7.      embedding  vector(1536)
8.  );
9.
10. -- an approximate index, when approximate is good enough
11. CREATE INDEX ON docs USING hnsw (embedding vector_cosine_ops);
12.
13. -- nearest neighbours, filtered, in one ordinary query
14. SELECT id, body FROM docs WHERE lang = 'en' ORDER BY embedding <=> $1 LIMIT 5;
```

# Why it works so well

Nothing new to run, nothing new to secure

### One system

Rows and their embeddings share a table, a transaction and a backup.

### Joins and filters

WHERE, JOIN and ORDER BY … LIMIT in a single plan.

### Real indexes

HNSW or IVFFlat when approximate is enough. An exact scan when it is not.

### Everything inherited

Replication, point-in-time recovery and row-level security all apply.

### An extension, not a fork

Added with CREATE EXTENSION to the PostgreSQL you already run.

### Built in the open

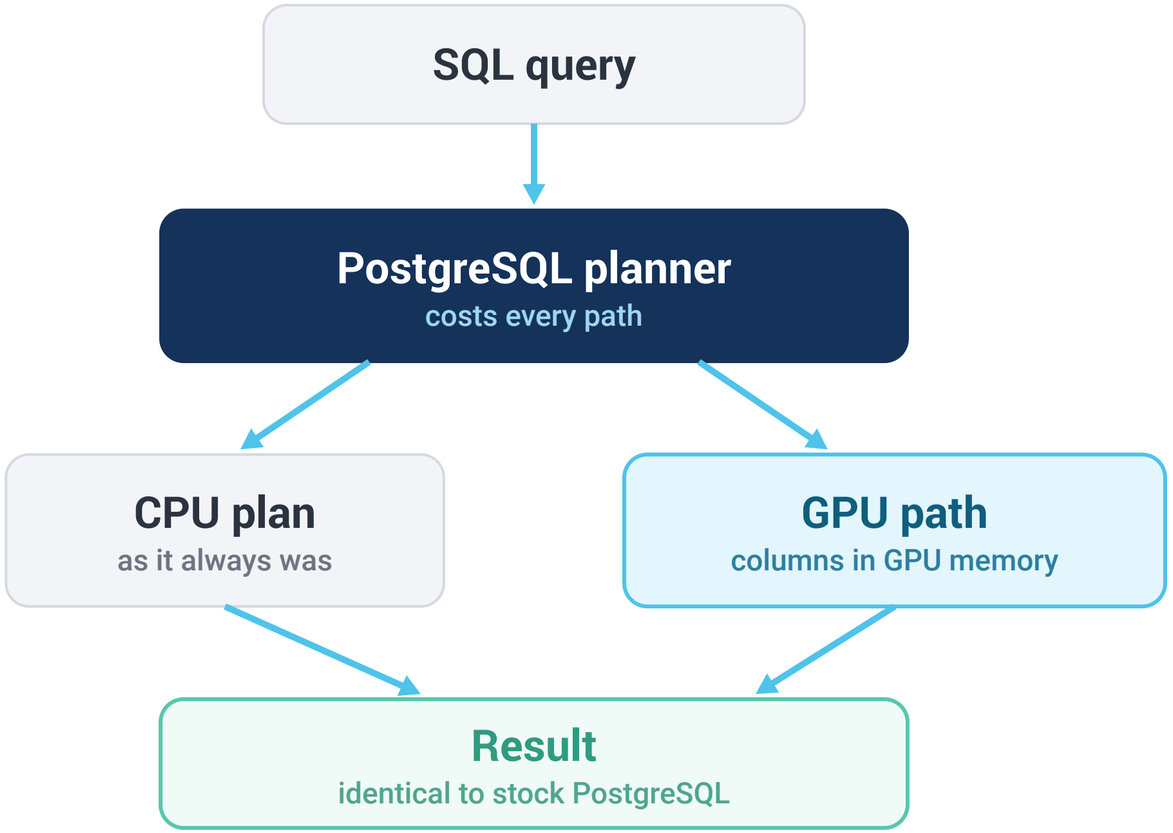
Developed by a community, under the PostgreSQL licence.

# A short look at PGEE GPU

More speed for our clients

# Where it sits

One more candidate path in the planner




## Costed like any other path

There is no hint and no session flag. If the GPU path is not cheaper, it is not chosen.

If a concurrent writer touched part of the data, that part is finished on the CPU in the same pass. The result is a smaller GPU share, never a stale answer.

MEASURED ON 4 × NVIDIA H200

# 2.18 trillion

rows scanned per second

| 1.10 ms | 11,182× | 100% |
| --- | --- | --- |
| to count 2.4 billion rows | faster than 32 CPU cores | answers identical to PostgreSQL |

Measured: count(*) over a 2.4-billion-row table, warm, best of three. A count is the best case for this engine, not a general mix.

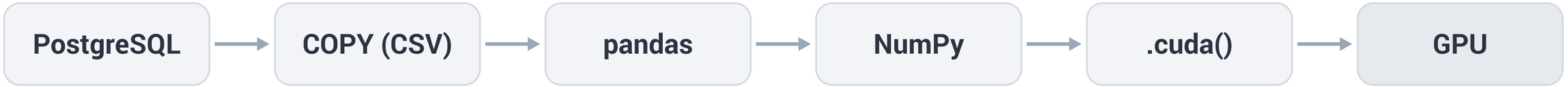
# What we are open sourcing

Attach PyTorch directly to database memory

# The detour every pipeline takes

Getting database rows onto the GPU that already holds them

Today




18.1 s

Attached



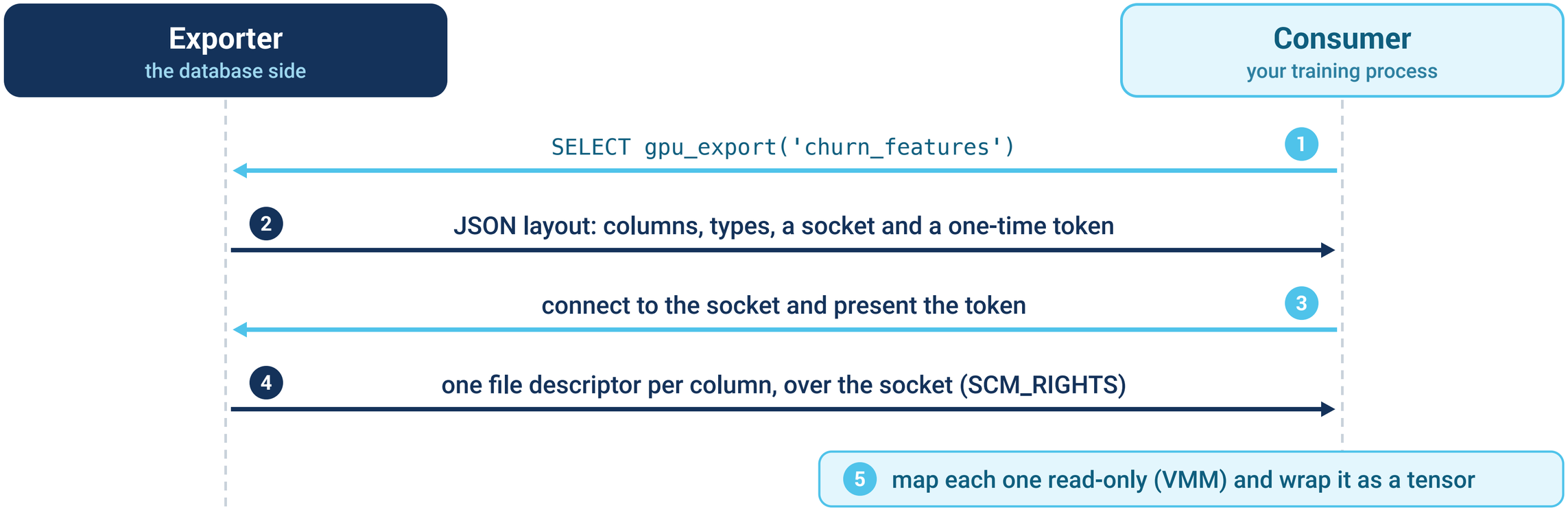

0.15 s

## 124× faster

Eight columns by 20 million rows · NVIDIA GB10 · measured · the results are identical

# The attach protocol

Five steps that any other consumer could implement




**Only descriptors move. The bytes stay where the database put them.**

# The layout is the contract

layout (abridged)

```
{
  "transport": "vmm",
  "view": "churn_features",
  "generation": 7,
  "rows": 20000000,
  "socket": "/run/postgresql/…",
  "token": "9f2c…",
  "columns": [
    { "name": "tenure_months",   "dtype": "int32",        "pgtype": "integer" },
    { "name": "monthly_charges", "dtype": "scaled_int64", "scale": 2, "pgtype": "numeric(10,2)" },
    { "name": "churned",         "dtype": "int8",         "pgtype": "boolean" }
  ]
}
# every column also carries col_index, values_off, nullmask_off, bytes, dict_off and dim
```

# Attach from PyTorch

attach

```
import torch
from pgee_gpu import PsqlSession, export

sess = PsqlSession(host="/var/run/postgresql", port=5432, dbname="analytics")

with export(sess, "churn_features") as ex:        # the view the exporter offers
    tenure  = ex["tenure_months"].values            # a read-only view (VMM transport)
    charges = ex["monthly_charges"].decoded()       # exact numeric, as float64

    X = torch.stack([tenure.float(), charges.float()], dim=1)   # a new tensor: ours to change
    y = ex["churned"].values.float()
    train(model, X, y)                              # tenure += 1 would fault
```

# What arrives on the GPU

Types that survive the trip

### Exact numerics

numeric arrives as scaled integers plus a scale, never as a float.

### Encoded text

Text arrives as dictionary ids: categorical features, already encoded.

### Nulls stay nulls

A null mask always travels with the column. Missing is never silently zero.

### A frozen snapshot

You read one generation of the data until you let go, while the data moves on.

### Read-only

Each column is mapped read-only on the VMM transport. A write faults.

### No copy

The tensors are views of memory the database already holds.

# Where it fits, and where it does not

Said before you find out

## It fits

- Training and feature retrieval on the database host.
- Features that are already SQL, in a view.
- Integer, float, numeric and text columns.

## It does not, today

- Remote training: use COPY for that.
- Live data: you read a snapshot, and export again to move it.
- The GPU path needs an NVIDIA GPU and CUDA.

**No GPU? There is a way in.**

A shared-memory transport exports the same columns on a Mac, or on any host without a GPU manager.

CYBERTEC
POSTGRESQL SERVICES & SUPPORT

# Coming soon, as open source

The direct attach to GPU memory. Not the GPU engine.

Announcement and source: **www.cybertec-postgresql.com**

**Built and supported by CYBERTEC**