Vector Storage Challenges

Nov 2024 Barcelona PUG

Dave Pitts - Database Engineer - Adyen (Madrid)

adyen

engineered
for ambition

Vector Storage Challenges

Nov 2024 Barcelona PUG

Dave Pitts - Database Engineer - Adyen (Madrid)

Redis

adyen

engineered
for ambition

The Adyen Formula

We include

different people

to sharpen our ideas

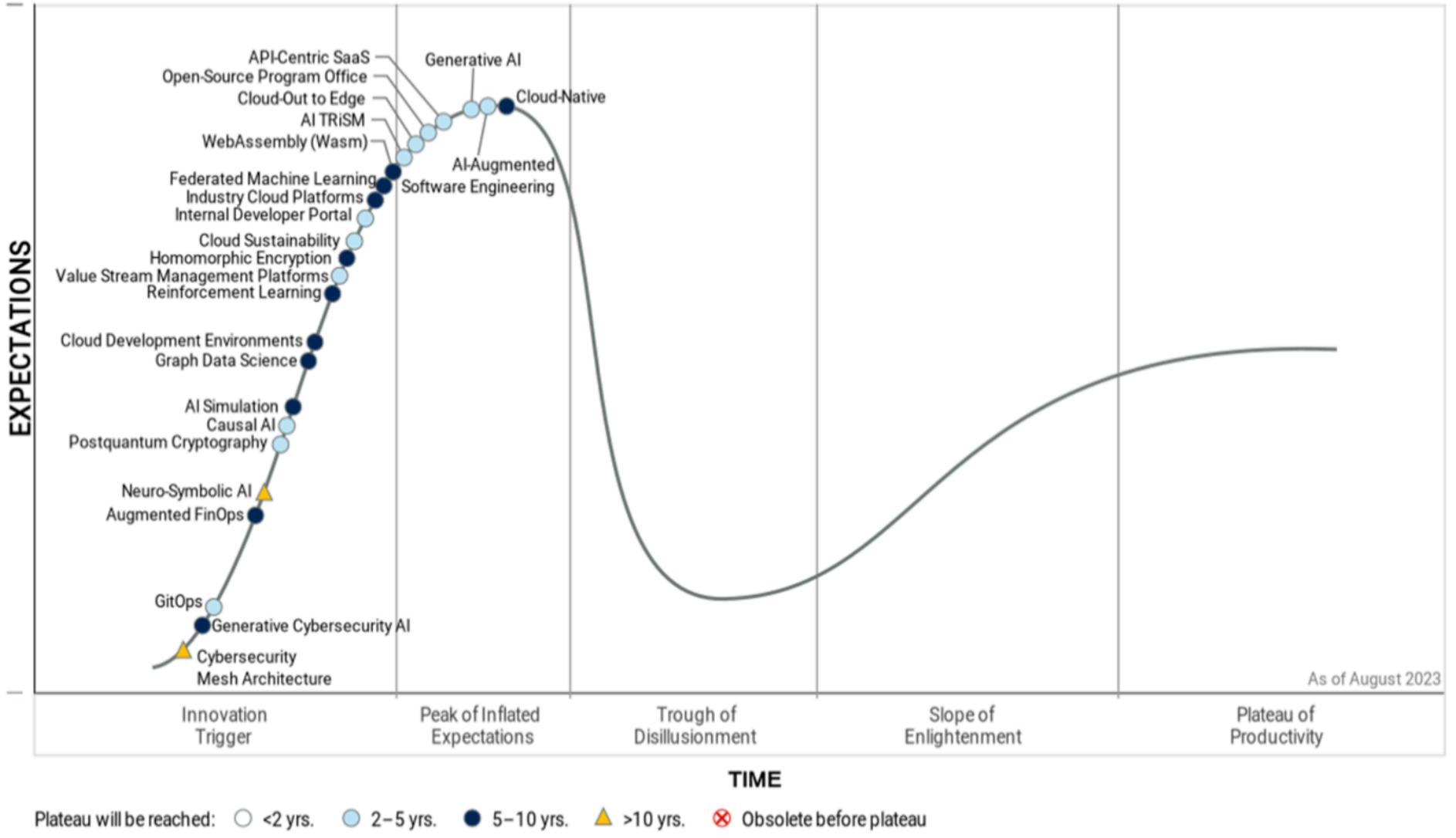
Neurodiversity ERG (Madrid co-Lead)

The AI HypeCycle?

adyen

engineered
for ambition

Figure 1. Hype Cycle for Emerging Technologies, 2023



Gartner

Background - what are vectors? pgVector based intro…

```
postgres=# \d v2
                                  Table "public.v2"
  Column   |       Type        | Collation | Nullable |              Default
-----------+-------------------+-----------+----------+-----------------------------------
 id        | integer           |           | not null | nextval('v2_id_seq'::regclass)
 name      | character varying |           | not null |
 embedding | vector(2)         |           |          |
Indexes:
    "v2_pkey" PRIMARY KEY, btree (id)
    "v2_embedding_ivfflat_idx" ivfflat (embedding) WITH (lists='100')
```

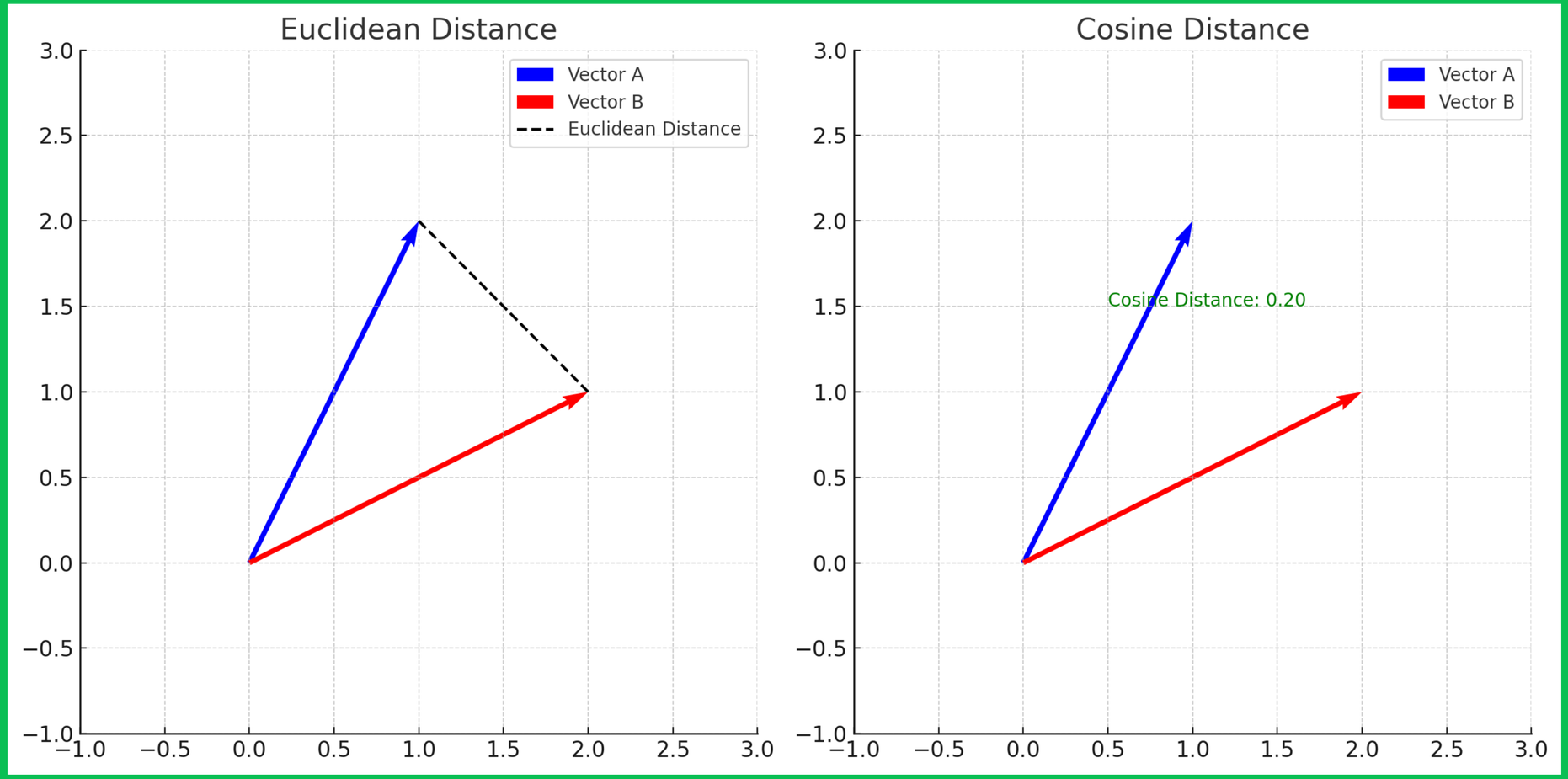
v2 - table with 2-dimensional arrays

```
postgres=# select * from v2 where id <=  10;
 id |     name     |          embedding
----+--------------+-------------------------
  1 | example_item | [0.066909604,0.9264157]
  2 | example_item | [0.36301365,0.5783217]
  3 | example_item | [0.9195596,0.7316905]
  4 | example_item | [0.5622496,0.9395603]
  5 | example_item | [0.638565,0.10896978]
  6 | example_item | [0.7050413,0.89062977]
  7 | example_item | [0.96956474,0.28005248]
  8 | example_item | [0.99984896,0.9242679]
  9 | example_item | [0.8908088,0.944833]
 10 | example_item | [0.2012447,0.018824905]
(10 rows)
```

v4 - table with 4-dimensional arrays

```
postgres=# select * from v4 where id <=  10;
 id |     name     |                    embedding
----+--------------+-----------------------------------------------
  1 | example_item | [0.25943446,0.71785164,0.4939812,0.13810147]
  2 | example_item | [0.30521932,0.8590337,0.05396965,0.31620467]
  3 | example_item | [0.53790855,0.610532,0.49912184,0.7980991]
  4 | example_item | [0.2628163,0.45041478,0.23879251,0.97646517]
  5 | example_item | [0.865146,0.4203888,0.25693193,0.004584638]
  6 | example_item | [0.5993715,0.8390484,0.110092215,0.767787]
  7 | example_item | [0.22387078,0.10384338,0.77448714,0.85737246]
  8 | example_item | [0.85913867,0.69988596,0.3159561,0.7109314]
  9 | example_item | [0.54125035,0.49513316,0.020857222,0.9136872]
 10 | example_item | [0.51117563,0.6467485,0.34206793,0.45778415]
(10 rows)
```

Vector Operations - **Euclidean vs Cosine Distance**



Vector Operations - **Euclidean Distance**

```
SELECT id, embedding
FROM v4
ORDER BY embedding <-> '[0.1, 0.2, 0.3, 0.4]’
LIMIT 10;
```

Vector Operations - **Cosine Distance**

```
SELECT id, embedding

FROM v4

ORDER BY embedding <=> '[0.1, 0.2, 0.3, 0.4]’

LIMIT 10;
```

And adding indexing

```
postgres=# CREATE INDEX IF NOT EXISTS v2_embedding_ivfflat_idx ON v2 USING ivfflat
(embedding) WITH (lists = 100);
CREATE INDEX

postgres=# CREATE INDEX IF NOT EXISTS v4_embedding_ivfflat_idx ON v4 USING ivfflat
(embedding) WITH (lists = 100);
CREATE INDEX
postgres=# CREATE INDEX IF NOT EXISTS v8_embedding_ivfflat_idx ON v8 USING ivfflat
(embedding) WITH (lists = 100);
CREATE INDEX
postgres=# CREATE INDEX IF NOT EXISTS v16_embedding_ivfflat_idx ON v16 USING ivfflat
(embedding) WITH (lists = 100);
CREATE INDEX
```

And adding indexing

```
postgres=# CREATE INDEX IF NOT EXISTS v2_embedding_ivfflat_idx ON v2 USING ivfflat
(embedding) WITH (lists = 100);
CREATE INDEX

postgres=# CREATE INDEX IF NOT EXISTS v4_embedding_ivfflat_idx ON v4 USING ivfflat
(embedding) WITH (lists = 100);
CREATE INDEX
postgres=# CREATE INDEX IF NOT EXISTS v8_embedding_ivfflat_idx ON v8 USING ivfflat
(embedding) WITH (lists = 100);
CREATE INDEX
postgres=# CREATE INDEX IF NOT EXISTS v16_embedding_ivfflat_idx ON v16 USING ivfflat
(embedding) WITH (lists = 100);
CREATE INDEX
```

for v2, v4, v8, v16, v32, … **v1024**

Hey **Dave** for our PoC we used Redis

- simple and lightweight?

- do we need ACID compliance and transactional consistency?

- what about indexing?

- what about backup and recovery?

Running pg_bench tests … setup

```
postgres@dc465c92a2a4:~$ cat pgbench_v4.sql
\set id random(1, 10000)
SELECT id, name, embedding <-> (select embedding from v4 where id = :id) as euclidean_distance
from v4 order by euclidean_distance limit 100;
```

```
postgres@dc465c92a2a4:~$ pgbench -f pgbench_v4.sql -c 1 -t 500
…
number of transactions actually processed: 500/500
number of failed transactions: 0 (0.000%)
latency average = 0.124 ms
initial connection time = 1.905 ms
tps = 8052.048441 (without initial connection time
```

Background - example query

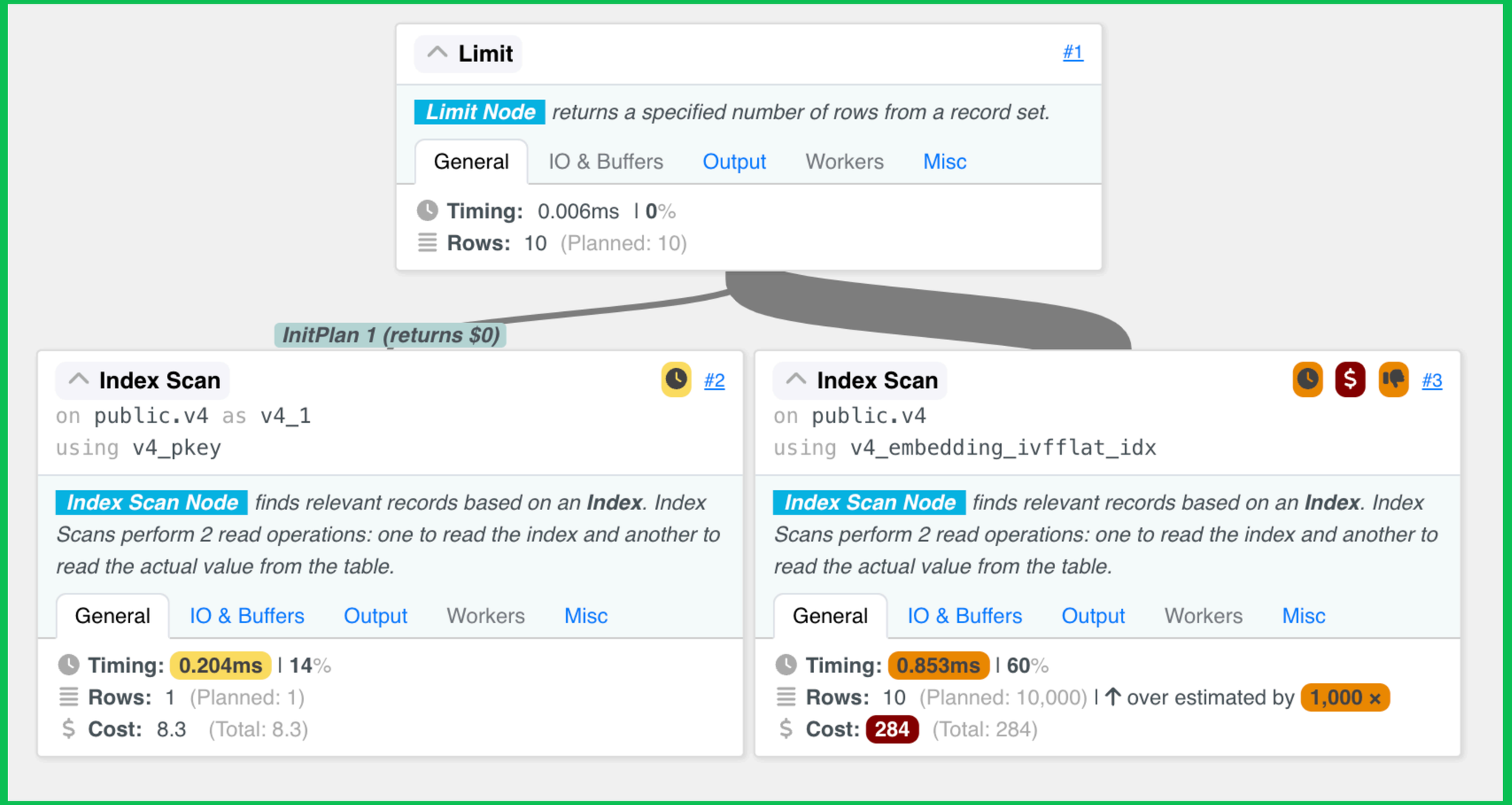
```
postgres=# SELECT id, name, embedding,
embedding <-> (select embedding from v4 where id = 42) as euclidean_distance
from v4 order by euclidean_distance limit 10;
  id  |     name     |                    embedding                    | euclidean_distance
------+--------------+-------------------------------------------------+--------------------
   42 | example_item | [0.41936868,0.21380356,0.9548757,0.44536898]    |                  0
 4403 | example_item | [0.4768776,0.22438358,0.98122853,0.41351837]    |  0.0716110658504272
 5881 | example_item | [0.3739942,0.24519081,0.95423734,0.39586428]    | 0.074129116368019 85
 4485 | example_item | [0.37156776,0.22567432,0.87409854,0.42469677]   | 0.0968407676187699 9
 4824 | example_item | [0.4483818,0.23014477,0.8792314,0.49804822]     | 0.0980100041855578 7
 3249 | example_item | [0.5110938,0.19154403,0.98203796,0.4875552]     |  0.106894568720873 2
 2886 | example_item | [0.49190125,0.20986946,0.9621258,0.5348985]     | 0.115518627498980 52
 7326 | example_item | [0.49487656,0.17229778,0.87750185,0.42601272]   | 0.117411879464941 03
 1555 | example_item | [0.46581516,0.11873428,0.89035386,0.47200975]   | 0.126760571512957 44
 1833 | example_item | [0.5326645,0.21787386,0.97018003,0.36689878]    |   0.138723820373035
(10 rows)
```

Background - query plan

```
postgres=# explain (analyze, buffers, verbose) SELECT id, name, embedding, embedding <->
(select embedding from v4 where id = 42) as euclidean_distance from v4 order by
euclidean_distance limit 10;
```

```
 Limit  (cost=14.05..14.33 rows=10 width=46) (actual time=0.397..0.859 rows=10 loops=1)
   Output: v4.id, v4.name, v4.embedding, ((v4.embedding <-> $0))
   Buffers: shared hit=16
   InitPlan 1 (returns $0)
     ->  Index Scan using v4_pkey on public.v4 v4_1  (cost=0.29..8.30 rows=1 width=21) (actual
time=0.202..0.204 rows=1 loops=1)
           Output: v4_1.embedding
           Index Cond: (v4_1.id = 42)
           Buffers: shared hit=3
   ->  Index Scan using v4_embedding_ivfflat_idx on public.v4  (cost=5.75..283.75 rows=10000
width=46) (actual time=0.394..0.853 rows=10 loops=1)
         Output: v4.id, v4.name, v4.embedding, (v4.embedding <-> $0)
         Order By: (v4.embedding <-> $0)
         Buffers: shared hit=16
```

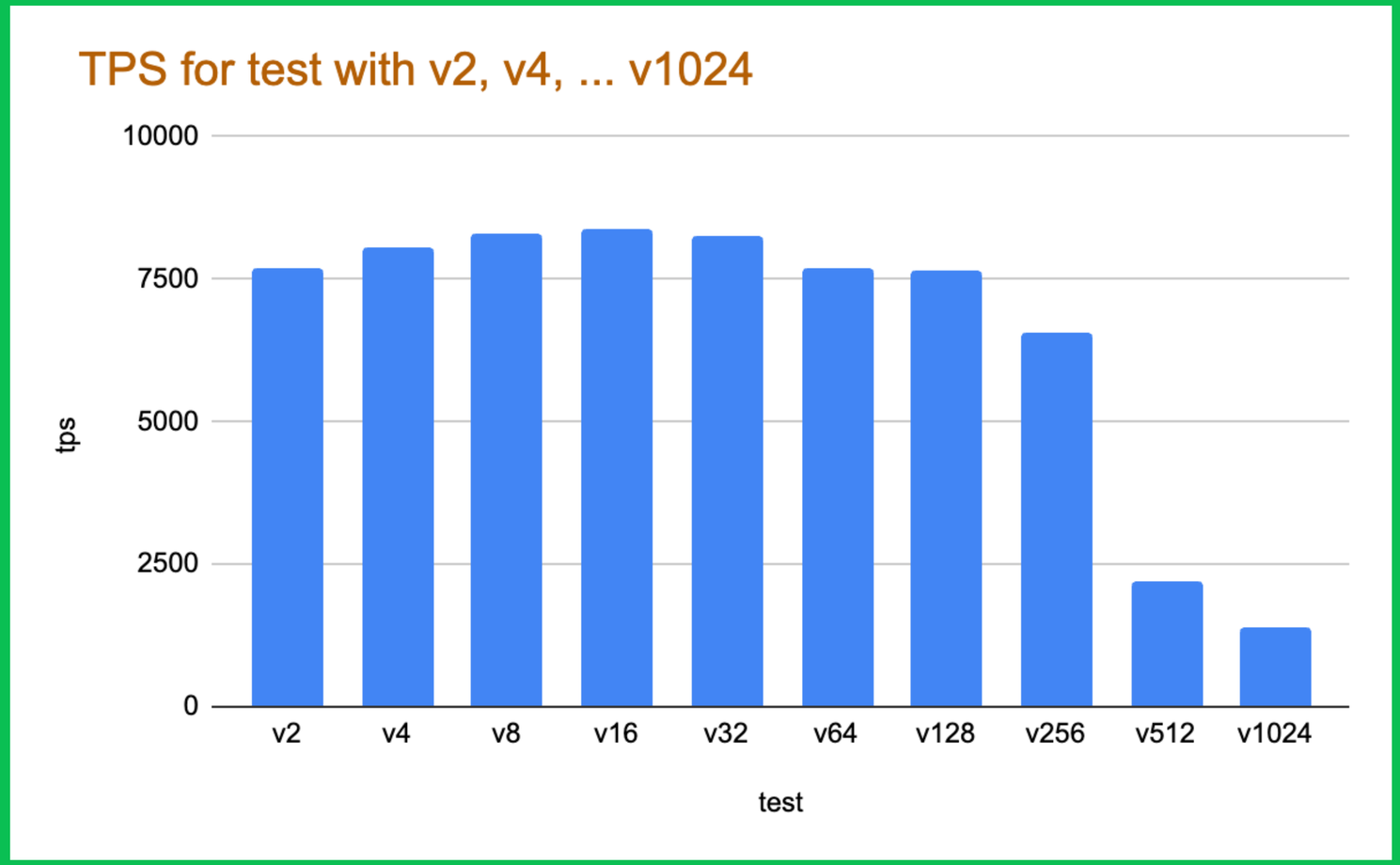
Background - query plan in pev2 (thanks dalibo!)



https://explain.dalibo.com/plan/f9981300a2252cd1

Running pg_bench tests … performance cliff for large vectors

test	tps
v2	7709
v4	8052
v8	8317
v16	8361
v32	8272
v64	7705
v128	7646
v256	6549
v512	2191
v1024	1373



What the problem - focus on the TOAST storage (overflow)

oid	table_schema	table_name	row_estimate	total_bytes	index_bytes	toast_bytes	table_bytes	table	index	total
26649	public	v1024	10000	139059200	82993152	55345152	720896	704 kB	79 MB	133 MB
16586	public	v512	10000	56672256	28139520	27811840	720896	704 kB	27 MB	54 MB
26607	public	v256	10000	24166400	12419072	8192	11739136	11 MB	12 MB	23 MB
26596	public	v128	10000	12058624	6160384	8192	5890048	5752 kB	6016 kB	12 MB
16574	public	v64	10000	6815744	3497984	8192	3309568	3232 kB	3416 kB	6656 kB
36668	public	v32	10000	3964928	2015232	8192	1941504	1896 kB	1968 kB	3872 kB
36691	public	v16	10000	2834432	1622016	8192	1204224	1176 kB	1584 kB	2768 kB
16554	public	v2	14977	2138112	1187840	8192	942080	920 kB	1160 kB	2088 kB
36700	public	v8	10000	1974272	1081344	8192	884736	864 kB	1056 kB	1928 kB
36709	public	v4	10000	1810432	1081344	8192	720896	704 kB	1056 kB	1768 kB

What the problem - focus on the TOAST storage (overflow)

oid	table_schema	table_name	row_estimate	total_bytes	index_bytes	toast_bytes	table_bytes	table	index	total
26649	public	v1024	10000	139059200	82993152	55345152	720896	704 kB	79 MB	133 MB
16586	public	v512	10000	56672256	28139520	27811840	720896	704 kB	27 MB	54 MB
26607	public	v256	10000	24166400	12419072	8192	11739136	11 MB	12 MB	23 MB
26596	public	v128	10000	12058624	6160384	8192	5890048	5752 kB	6016 kB	12 MB
16574	public	v64	10000	6815744	3497984	8192	3309568	3232 kB	3416 kB	6656 kB
36668	public	v32	10000	3964928	2015232	8192	1941504	1896 kB	1968 kB	3872 kB
36691	public	v16	10000	2834432	1622016	8192	1204224	1176 kB	1584 kB	2768 kB
16554	public	v2	14977	2138112	1187840	8192	942080	920 kB	1160 kB	2088 kB
36700	public	v8	10000	1974272	1081344	8192	884736	864 kB	1056 kB	1928 kB
36709	public	v4	10000	1810432	1081344	8192	720896	704 kB	1056 kB	1768 kB

TOAST storage (The Oversized-Attribute Storage Technique)

What about halfvec?

Is additional precision provided by 32-bit floating-point numbers necessary?

Many modern GPUs optimized for half-precision computations (FP16).

How about using Cassandra as a vector store

- native horizontal scaling (fully/natively distributed)

- do we need ACID compliance and transactional consistency?

- but need to integrate vector search (Weaviate or Milvus)

- **and** CQL vs SQL (doesn’t look like a regular database)

Links

Barcelona & Madrid PostgreSQL User Groups - LinkedIn

Barcelona PostgreSQL User Groups - Meetup

Madrid PostgreSQL User Groups - Meetup

Talking PostgreSQL Podcast - Tom Lane episode

Q & A ?

Maybe over a beer …

Dave Pitts - Database Engineer - Adyen (Madrid)

adyen

engineered
for ambition