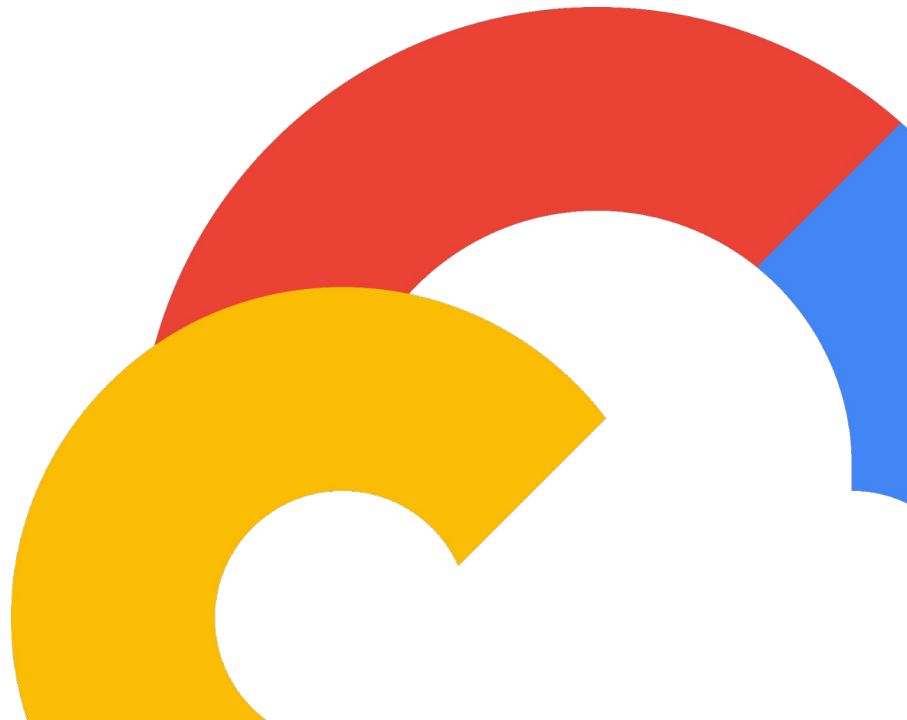


# Vector data in Postgres

Size, TOAST, filters and  
performance

Oct 2025

Google Cloud



# Contents

- 01** **Vectors**
- 02** **Storing Vectors**
- 03** **KNN search**
- 04** **ANN search**
- 05** **Indexes**
- 06** **Vectors and Filters**
- 07** **DML**
- 08** **Q/A**

# Speaker introduction



**Gleb  
Otochkin**

Cloud Advocate  
Databases

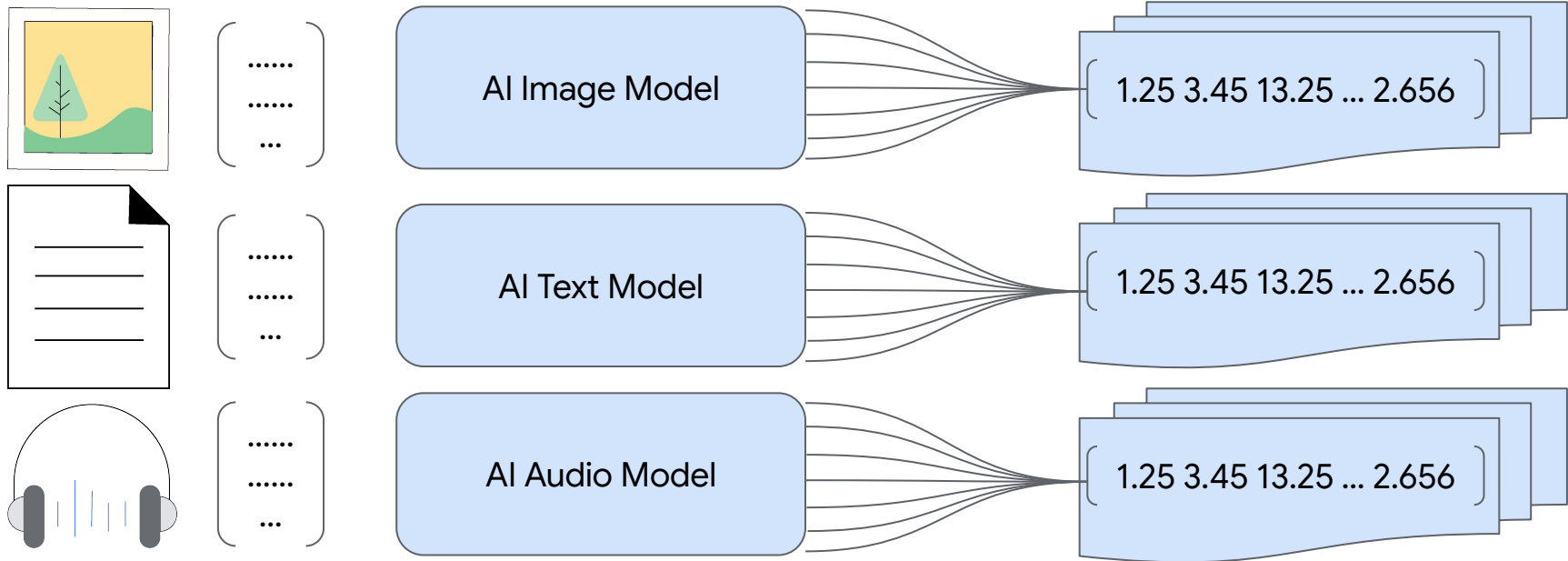
- Live in Ottawa, Canada
- Earned a degree in oceanology and participated in expeditions dedicated to oceanic research in the Pacific.
- Runner. Next Marathons - Florence and the Boston 2026)



# 01 Vectors

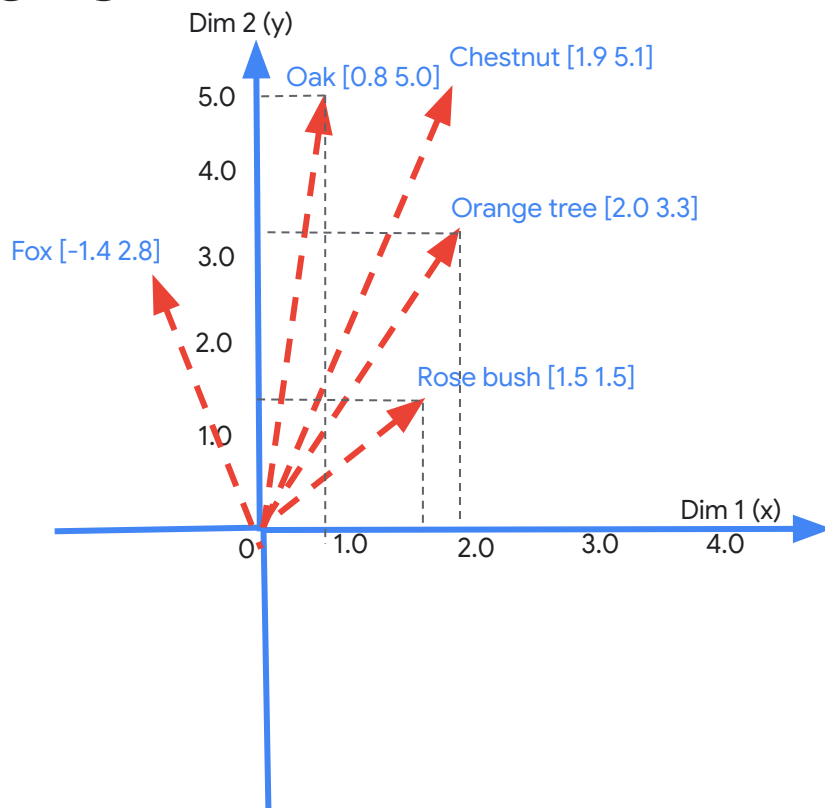
Why are we talking about vectors

# AI and Vectors



# Vectors and dimensions

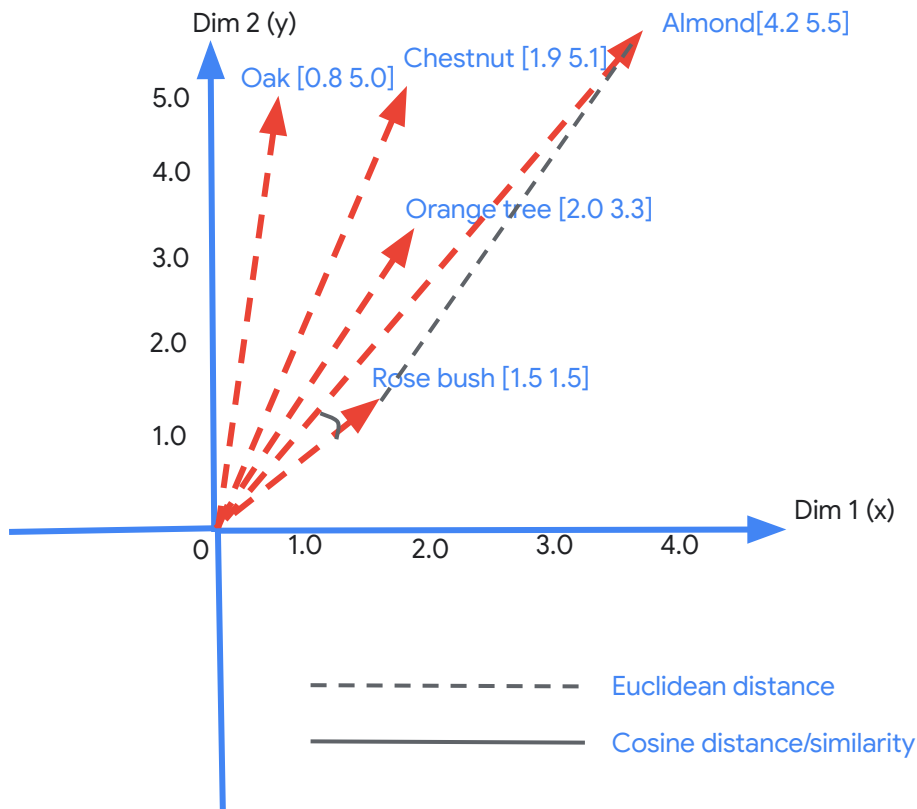
- Vector defined by coordinates in n-dimensional space
- Example for 2-dimensional space:
  - Coordinates in 2 dimensional space
  - Dimension 1 (x) - the first coordinate
  - Dimension 2 (y) - the second coordinate
  - Vector is direction from the center of coordinates



# Distance between vectors

- Euclidean distance - straight line
- Cosine distance - based on angle
- Other ways( L2, Inner product)

$$distance = \sqrt{(x_R - x_A)^2 + (y_R - y_A)^2}$$





# 02 Vector Data in PostgreSQL

How we store vectors in database

**Vector** is an ordered list of numbers.  
These numbers represent magnitudes  
along different dimensions.

**Vector embedding** is a vector that  
represents a piece of data

**Vector embedding size** depends on  
dimensions and doesn't depend on size of  
the source data

# Vector embeddings

- Dimensions - from 25 to 3000 (or more)
- Precision for each value (dimension) - float32
- Stored in postgres as vector data type
  - As single-precision, half-precision, binary, and sparse vectors
  - Single-precision (real) - 4 bytes

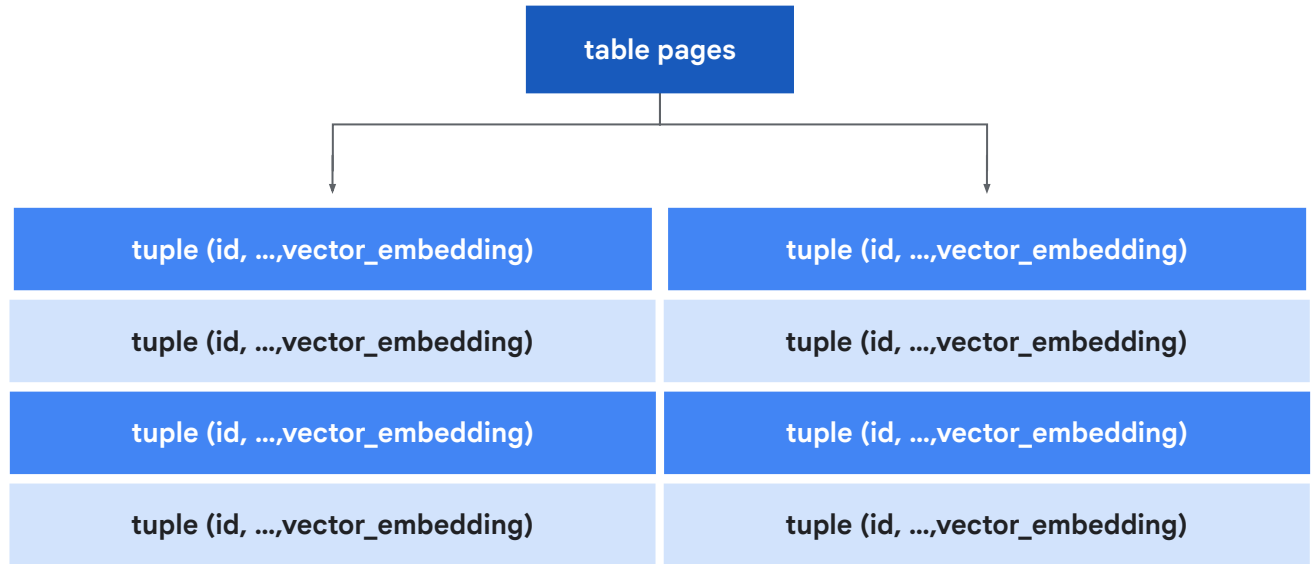
# Vector - stored inline

Up to 500 dimensions is stored inline

4 bytes \* 500 = 2000

Vector size is less than

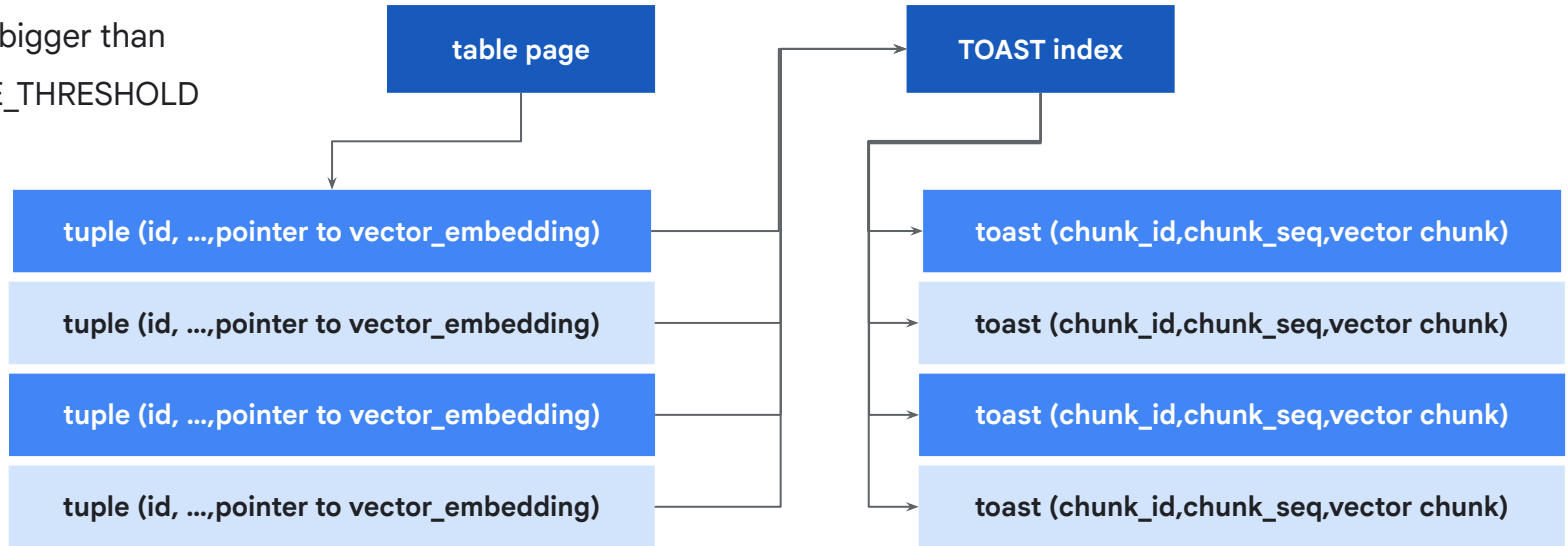
TOAST\_TUPLE\_THRESHOLD



# Vector - using TOAST

More than 500 dimensions is stored in TOAST chunks

Vector size is bigger than  
TOAST\_TUPLE\_THRESHOLD





# 03

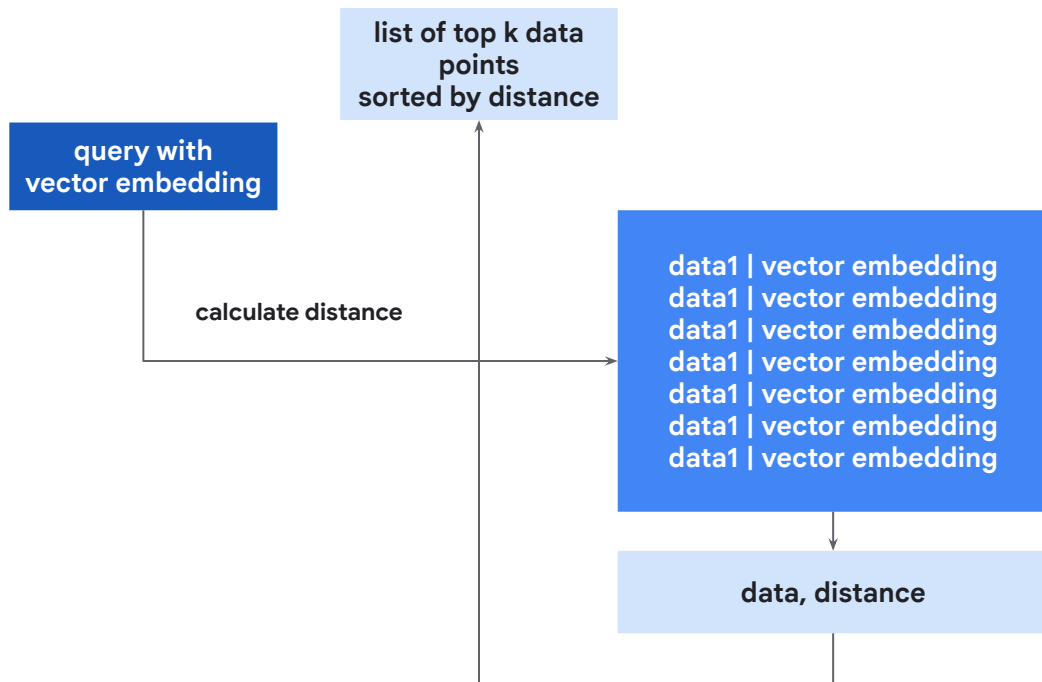
# Vector Search - KNN

Compare vectors and finding similarities

# KNN

## What is KNN or kNN?

- It stands for k-nearest neighbor search
- Sometimes called exact search
- Linear - depends on the size of a dataset
- Uses different measurements:
  - L2 distance
  - Cosine distance
  - inner product
  - Others -  
<https://github.com/pgvector/pgvector>



# KNN - how it works

## What is KNN or kNN?

- Compares a vector to every other vector
- Sort it by distance
- Get top k values
- Keeping in memory on k top results
- Memory depends on K

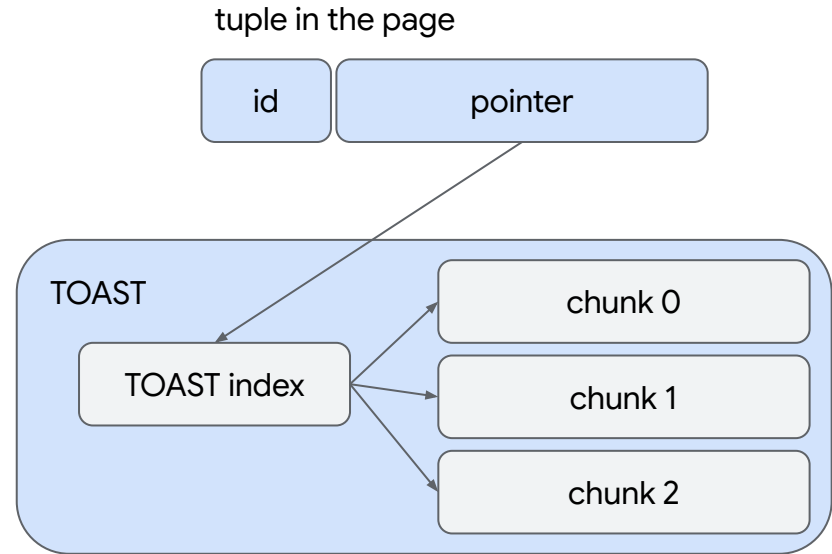
```
EXPLAIN ANALYZE
SELECT
  ID, 1 - (D <=> :'SAMPLE_VECTOR':VECTOR) AS SIMILARITY
FROM TV500
ORDER BY D <=> :'SAMPLE_VECTOR':VECTOR LIMIT 5;
```

```
Limit (cost=265212.38..265212.97 rows=5 width=24) (actual time=881.220..916.180
rows=5 loops=1)
  -> Gather Merge (cost=265212.38..362441.47 rows=833334 width=24) (actual
time=878.233..913.190 rows=5 loops=1)
    Workers Planned: 2
    Workers Launched: 2
    -> Sort (cost=264212.36..265254.03 rows=416667 width=24) (actual
time=803.573..803.575 rows=4 loops=3)
      Sort Key: ((d <=>
'[0.31021905,0.42329416,0.7588454,....,0.5502057]':vector))
      Sort Method: top-N heapsort Memory: 25kB
      Worker 0: Sort Method: top-N heapsort Memory: 25kB
      Worker 1: Sort Method: top-N heapsort Memory: 25kB
      -> Parallel Seq Scan on tv500 (cost=0.00..257291.67 rows=416667
width=24) (actual time=8.299..630.664 rows=333333 loops=3)
        Planning Time: 0.121 ms
        Execution Time: 1045.325 ms
```

# KNN - TOAST

## How TOAST impact KNN search?

- For each vector additional IO
- At least 3 IO buffers to get data



# KNN - TOAST

## How does TOAST impact KNN search?

- Execution longer
- IO is higher

```
Limit (cost=22565.38..22565.97 rows=5 width=24) (actual time=9797.457..9845.396
rows=5 loops=1)
  -> Gather Merge (cost=22565.38..119794.47 rows=833334 width=24) (actual
time=9797.454..9845.391 rows=5 loops=1)
    Workers Planned: 2
    Workers Launched: 2
      -> Sort (cost=21565.36..22607.03 rows=416667 width=24) (actual
time=9721.322..9721.323 rows=4 loops=3)
        Sort Key: ((d <=> '[0.15708542,0.61898744, ...
,0.46348935]':vector))
        Sort Method: top-N heapsort Memory: 25kB
        Worker 0: Sort Method: top-N heapsort Memory: 25kB
        Worker 1: Sort Method: top-N heapsort Memory: 25kB
      -> Parallel Seq Scan on tv501 (cost=0.00..14644.67 rows=416667
width=24) (actual time=0.285..9514.106 rows=333333 loops=3)
        Planning Time: 0.120 ms
        Execution Time: 9845.434 ms
```



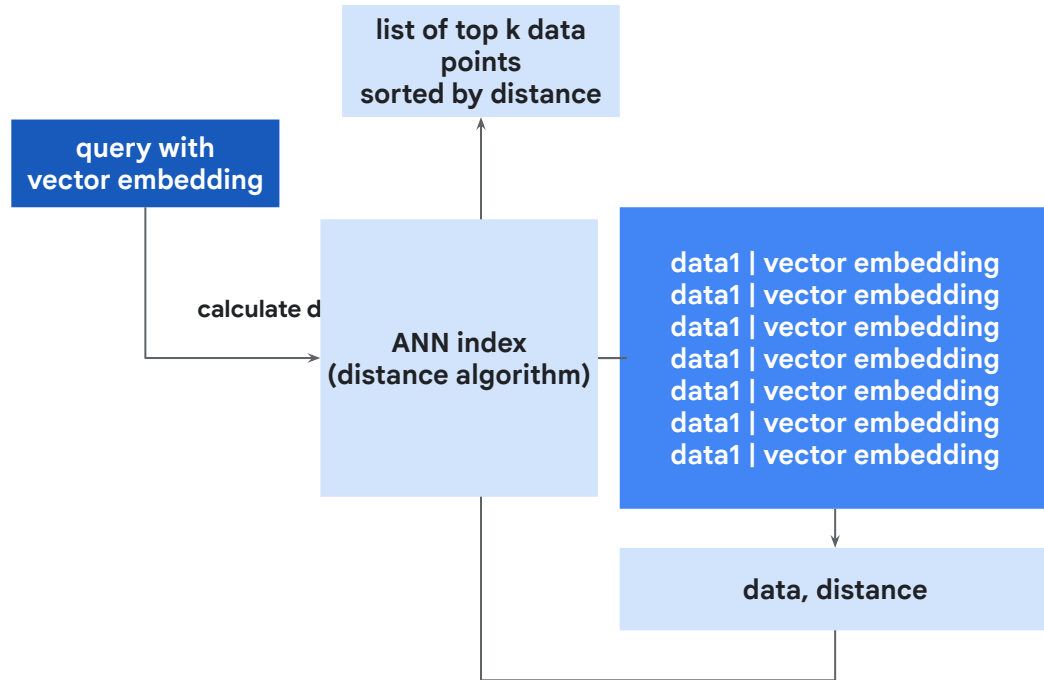
# 05 Vector Search - ANN

Compare vectors and finding approximate similarities

# ANN - what it is

## ANN - Approximate Nearest Neighbours

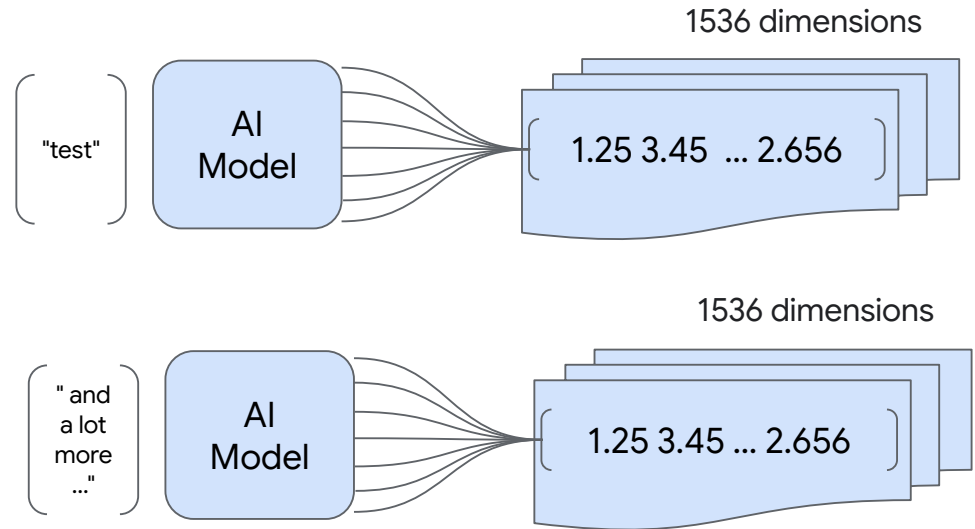
- KNN is too expensive and slow
- ANN trades recall to speed
- Enabled by using ANN index for vectors
- Search top-k value using distance algorithm
- Distance algorithm is part of the index



# ANN and dimensions

## More dimensions - bigger size for vector

- Data size in a row >2k - TOAST
- What about index ?
- Size limited by 2k for index value
- Over 2000 dimensions limit in pgvector?
  - Half-Precision Indexing (halfvec)
  - Quantization - changing precision (and size)





# 04 Indexes

What kind of indexes we can use with the vectors



**Index reduces scan area and as result  
reduces number of work you need to  
do to retrieve and compare the data”**

# Vector Indexes



## IVFFlat

---

- Tree-based
- Fast rebuild



## HNSW

---

- Graph-based
- Fast quality recall



## AlloyDB ScaNN

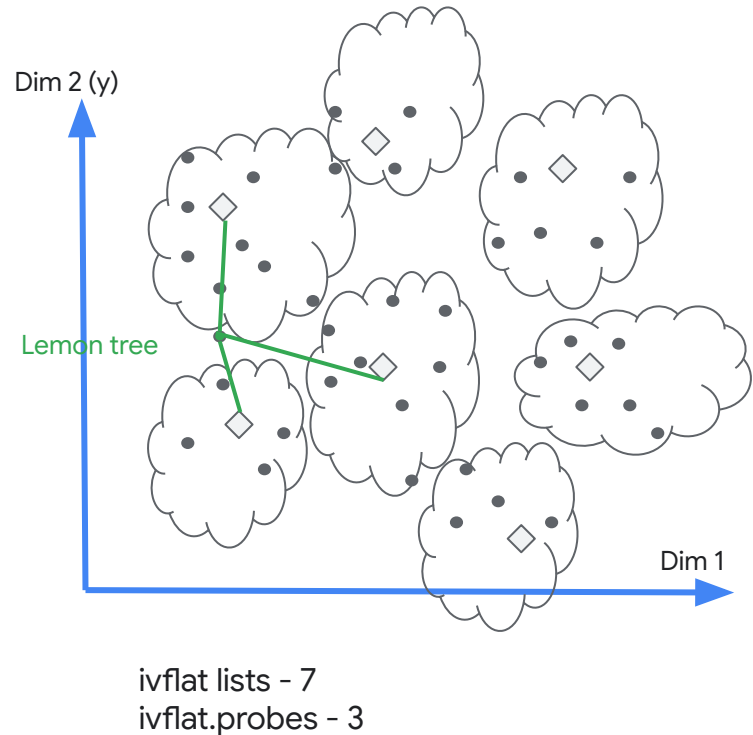
---

- Tree-based
- Uses Google algorithm

# IVFFlat

## IVFFlat - Inverted File with Flat Compression

- Tree-based with centroids
- Define number of lists
- Create centroids for each list
- Assign each vector to a centroid
- Search only for probed lists
- Parameters:
  - lists - number of centroids (during creation)
  - ivfflat.probes - number of lists to try (execution)



# IVFFlat - execution

## IVFFlat - vector inline

- Lists = 100
- Cosine distance
- Default parameters

```
EXPLAIN ANALYZE
SELECT
  ID, 1 - (D <=> :'SAMPLE_VECTOR'::VECTOR) AS SIMILARITY
FROM tvtest
ORDER BY D <=> :'SAMPLE_VECTOR'::VECTOR LIMIT 10;
```

```
Limit (cost=634.12..650.58 rows=10 width=24) (actual time=0.905..0.946 rows=10
loops=1)
  -> Index Scan using tvtest_d_idx on tvtest (cost=634.12..165162.50 rows=100000
width=24) (actual time=0.903..0.942 rows=10 loops=1)
    Order By: (d <=> '[0.5356429,0.07264433, ..., 0.7928642]':vector)
Planning Time: 0.108 ms
Execution Time: 0.973 ms
```

# IVFFlat - execution

## IVFFlat - vector in TOAST

- Lists = 100
- Cosine distance
- Default parameters

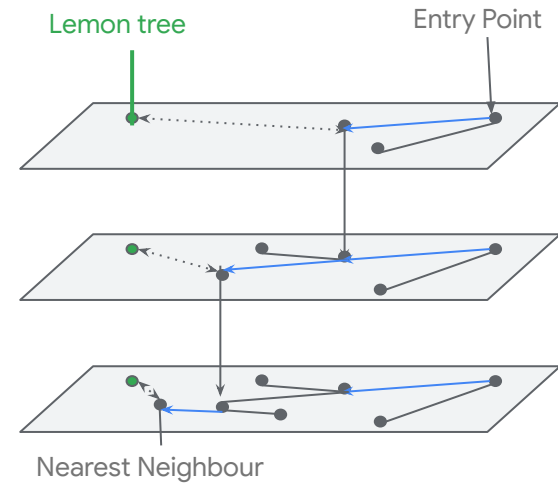
```
EXPLAIN ANALYZE
SELECT
  ID, 1 - (D <=> :'SAMPLE_VECTOR'::VECTOR) AS SIMILARITY
FROM tvtest
ORDER BY D <=> :'SAMPLE_VECTOR'::VECTOR LIMIT 10;
```

```
Limit (cost=633.95..640.70 rows=10 width=24) (actual time=1.537..1.672 rows=10
loops=1)
  -> Index Scan using tvtest_d_idx on tvtest (cost=633.95..68089.00 rows=100000
width=24) (actual time=1.535..1.668 rows=10 loops=1)
    Order By: (d <=> '[0.65339184,0.92753774, ... ,0.7215546]'::vector)
    Planning Time: 0.101 ms
    Execution Time: 1.698 ms
```

# HNSW

## HNSW - Hierarchical Navigable Small Worlds

- Split to layers
- Long links on the top
- Shorter links to bottom
- Starting on top from long distances
- Down in layers - shorter distances
- Graph based - handles updates



# HNSW - execution

## HNSW - vector inline

- Cosine distance
- Default parameters

```
EXPLAIN ANALYZE
SELECT
  ID, 1 - (D <=> :'SAMPLE_VECTOR'::VECTOR) AS SIMILARITY
FROM tvtest
ORDER BY D <=> :'SAMPLE_VECTOR'::VECTOR LIMIT 10;
```

```
Limit (cost=921.81..945.30 rows=10 width=24) (actual time=2.460..2.493 rows=10
loops=1)
  -> Index Scan using tvtest_d_idx on tvtest (cost=921.81..235840.00 rows=100000
width=24) (actual time=2.458..2.489 rows=10 loops=1)
    Order By: (d <=> '[0.5356429,0.07264433, ..., 0.7928642]':vector)
    Planning Time: 0.103 ms
    Execution Time: 2.518 ms
```

# HNSW - execution

## HNSW - vector in TOAST

- Cosine distance
- Default parameters

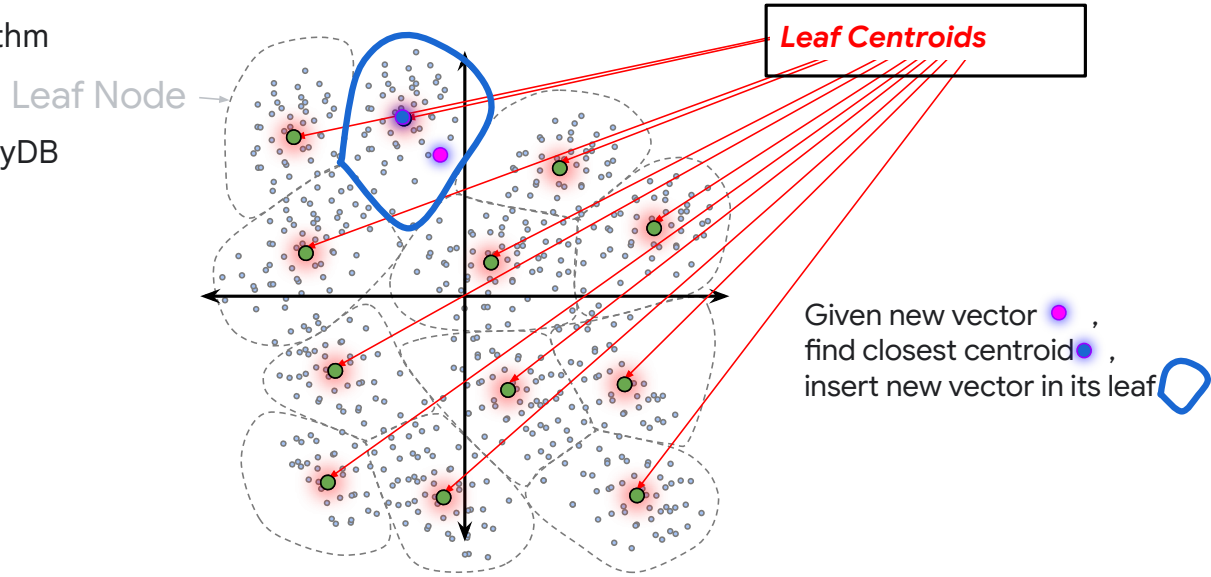
```
EXPLAIN ANALYZE
SELECT
  ID, 1 - (D <=> :'SAMPLE_VECTOR'::VECTOR) AS SIMILARITY
FROM tvtest
ORDER BY D <=> :'SAMPLE_VECTOR'::VECTOR LIMIT 10;
```

```
Limit (cost=921.81..935.59 rows=10 width=24) (actual time=3.677..3.806 rows=10
loops=1)
  -> Index Scan using tvtest_d_idx on tvtest (cost=921.81..138784.00 rows=100000
width=24) (actual time=3.675..3.802 rows=10 loops=1)
    Order By: (d <=> '[0.65339184,0.92753774, ... ,0.7215546]'::vector)
    Planning Time: 0.114 ms
    Execution Time: 3.832 ms
```

# ScaNN for AlloyDB

## AlloyDB - Scalable Nearest Neighbors

- ScaNN vector search algorithm from Google Research
- Natively integrated with AlloyDB
- Fast build
- Quality recall
- Auto-maintenance



- Algorithm open-sourced in 2019 (<https://github.com/google-research/google-research/tree/master/scann>)

# AlloyDB ScaNN - execution

## ScaNN - vector inline

- Cosine distance
- Default parameters

```
EXPLAIN ANALYZE
SELECT
  ID, 1 - (D <=> :'SAMPLE_VECTOR'::VECTOR) AS SIMILARITY
FROM tvtest
ORDER BY D <=> :'SAMPLE_VECTOR'::VECTOR LIMIT 10;
```

```
Limit (cost=320.00..320.47 rows=10 width=24) (actual time=0.774..0.808 rows=10
loops=1)
  -> Index Scan using tvtest_d_idx on tvtest (cost=320.00..5004.00 rows=100000
width=24) (actual time=0.771..0.803 rows=10 loops=1)
    Order By: (d <=> '[0.9267122,0.2705454, ...,0.3225359]'::vector)
    Limit: 10
Planning Time: 0.146 ms
Execution Time: 0.855 ms
```

# AlloyDB ScaNN - execution

## ScaNN - vector in TOAST

- Cosine distance
- Default parameters

```
EXPLAIN ANALYZE
SELECT
  ID, 1 - (D <=> :'SAMPLE_VECTOR'::VECTOR) AS SIMILARITY
FROM tvtest
ORDER BY D <=> :'SAMPLE_VECTOR'::VECTOR LIMIT 10;
```

```
Limit (cost=342.50..342.81 rows=10 width=24) (actual time=0.935..1.088 rows=10
loops=1)
  -> Index Scan using tvtest_d_idx on tvtest (cost=342.50..3486.50 rows=100000
width=24) (actual time=0.931..1.083 rows=10 loops=1)
    Order By: (d <=> '[0.14669167,0.53945506, ... ,0.34823254]'::vector)
    Limit: 10
Planning Time: 0.186 ms
Execution Time: 1.160 ms
```

# Indexes Build Time

|                        | IVFFLAT  | HNSW       | ScaNN     |
|------------------------|----------|------------|-----------|
| Build on 498 dimension | 5,047 ms | 282,323 ms | 11,816 ms |
| Build on 501 dimension | 6,042 ms | 386,666 ms | 14,594 ms |

1. Numbers can be different on your system and shown only for comparison



# 06

# Vector Search & Filters

Vector search with filters on another columns

# Pre-Filtering

**When:** High selectivity on the filtered value

Use a standard B-tree index on the metadata column *\*before\** the vector search.

- **Filter First:** The database uses the B-tree to quickly find the small set of rows matching the filter.
- **Vector Search Second:** An KNN search is performed only on this small, pre-filtered subset of vectors.

# Post-Filtering

**When: Low selectivity on the filtered value**

Perform the ANN vector search first, then filter the results after:

- **Vector search First:** The database performs a full ANN search to find the top K nearest neighbors from the entire dataset.
- **Filter Second:** The metadata filter is applied to this small list of K candidate results.

# Inline-Filtering

When: Uniform distributions (mid-range) of the filtered values (Iterative index scan)

The filter is applied during the ANN index scan (e.g., HNSW graph traversal).

- **Integrated Search:** As the ANN index algorithm explores the graph...
- **Check As It Goes:** ...it checks if each node/vector also matches the metadata filter before adding it to the candidate list.
- Best for: A good "all-rounder" when selectivity is moderate. Supported by modern ANN indexes.

# How pgvector Handles This

## The pgvector approach

- Pre-filtering: The Postgres planner often chooses this. (with or without index on the column)
- Post-filtering: This is supposed to be a default
- Inline-filtering (Iterative Index Scans HNSW): pgvector (0.8.0+) supports this for HNSW indexes.
- Inline-filtering (Iterative Index Scans IVF): IVF indexes in pgvector have supported this.

Parameters:

- (ivfflat)hnsw.iterative\_scan = relaxed\_order
- hnsw.max\_scan\_tuples = 20000
- ivfflat.max\_probes = 100

***"With approximate indexes, filtering is applied after the index is scanned ..." - default behaviour from README***

<https://github.com/pgvector/pgvector?tab=readme-ov-file#filtering>

# Filtering Strategy

| Strategy         | Best For Selectivity | Data Access          | Key Benefit                                 |
|------------------|----------------------|----------------------|---|
| Pre-Filtering    | High                 | B-Tree Index → KNN   | Drastically reduces kNN search space        |
| Post-Filtering   | Low                  | ANN → Filter         | Fastest when ANN is already selective       |
| Inline-Filtering | Mid-range            | Iterative index scan | Balanced performance in a single operation. |



**The best filtering strategy depends entirely on your data's selectivity. ”**

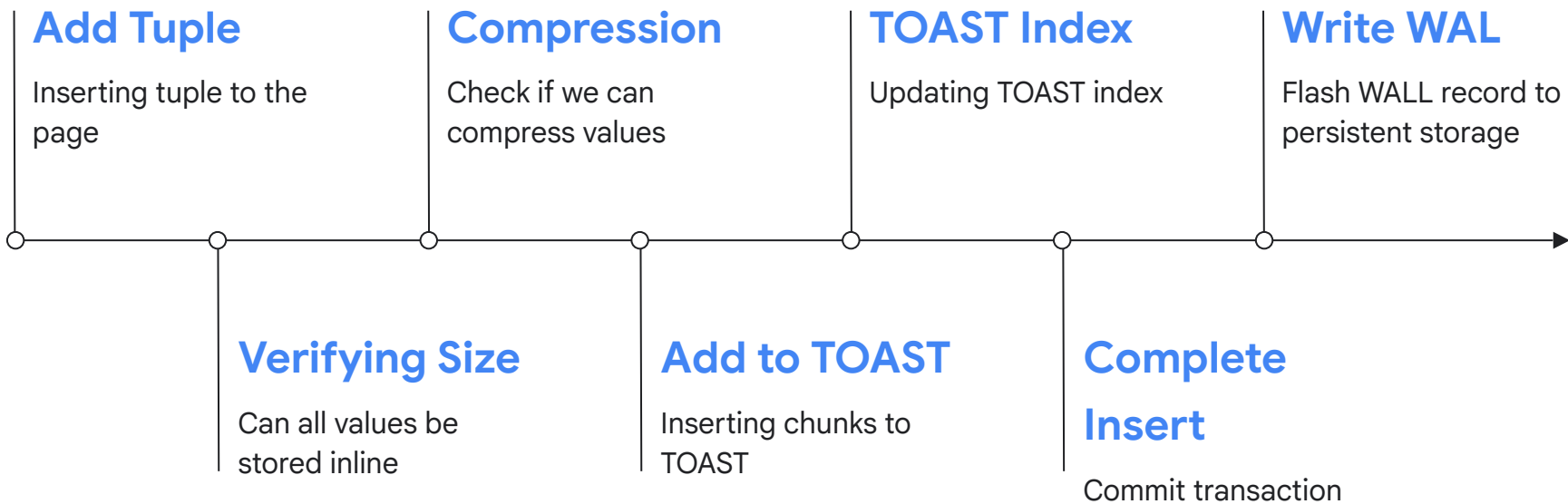


# 07

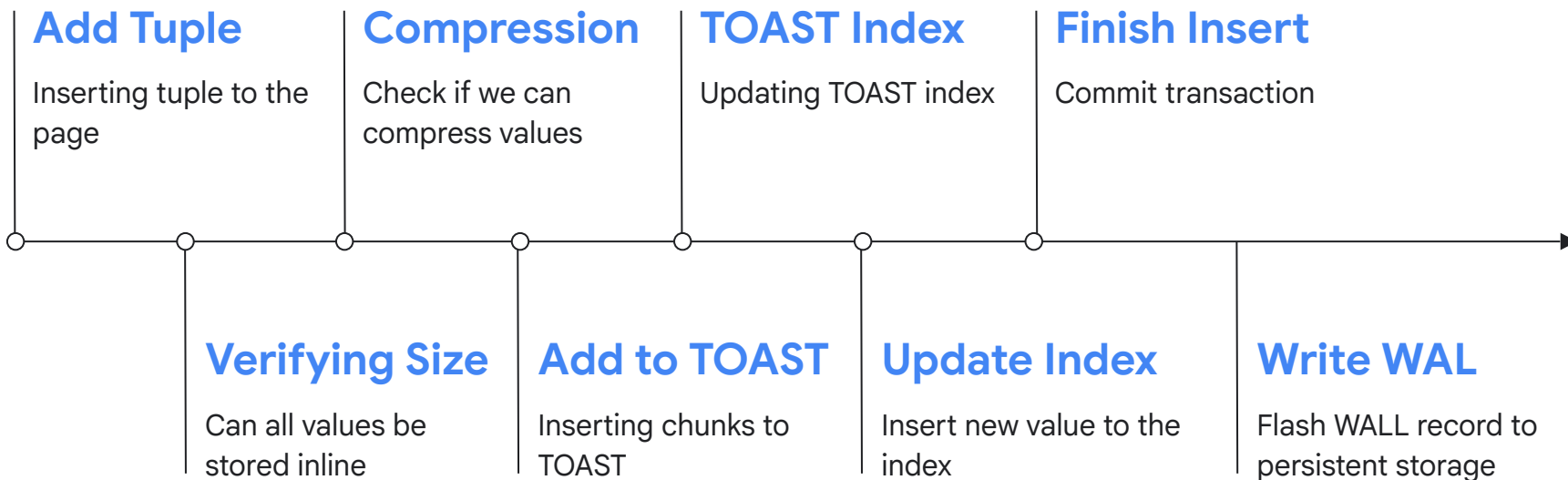
# DML on Vector Data

Inserting and Updating Vectors

# High Level Overview



# Insert with ANN Index



# DML on Vectors - Insert

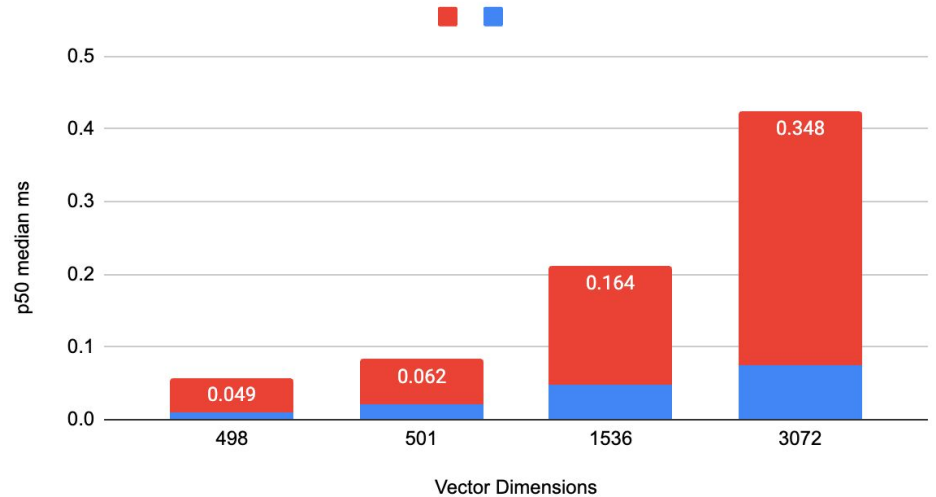
|                          | No Index | HNSW      | IVFFLAT  | TOAST chunks |
|--------------------------|----------|-----------|----------|--------------|
| Insert 1M 498 dimension  | 0.009 ms | 3.587 ms  | 0.049 ms | 0            |
| Insert 1M 501 dimension  | 0.021 ms | 11.335 ms | 0.062 ms | 2            |
| Insert 1M 1536 dimension | 0.047 ms | 11.842 ms | 0.164 ms | 4            |
| Insert 1M 3072 dimension | 0.076 ms | 21.435 ms | 0.348 ms | 7            |

1. Latency measured for p50 median
2. Numbers can be different on your system and shown only for comparison
3. For 3072 dimensions a casting to halfvec was used to build HNSW and IVFFlat indexes

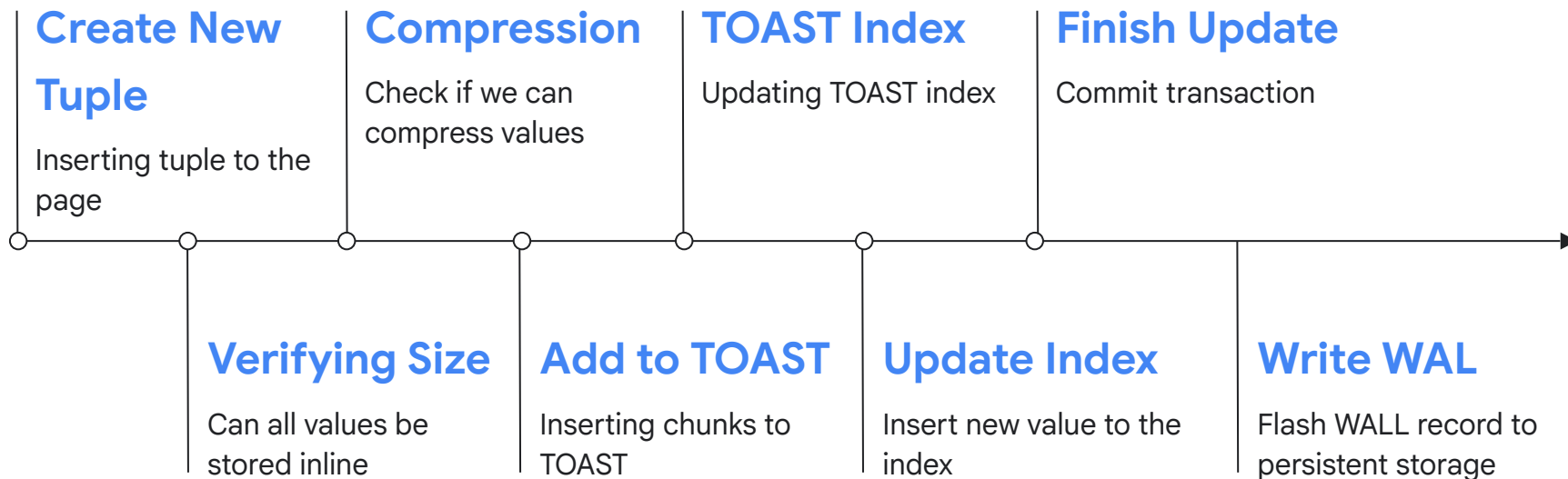
# DML - Insert

- Depends on TOAST - TOAST slows down inserts
- Depends on dimensions - how many TOAST chunks
- Indexes have significant performance impact
- HNSW is more expensive in overhead than IVFFlat

Vector DML, insert



# Update



# DML on Vectors - Updates

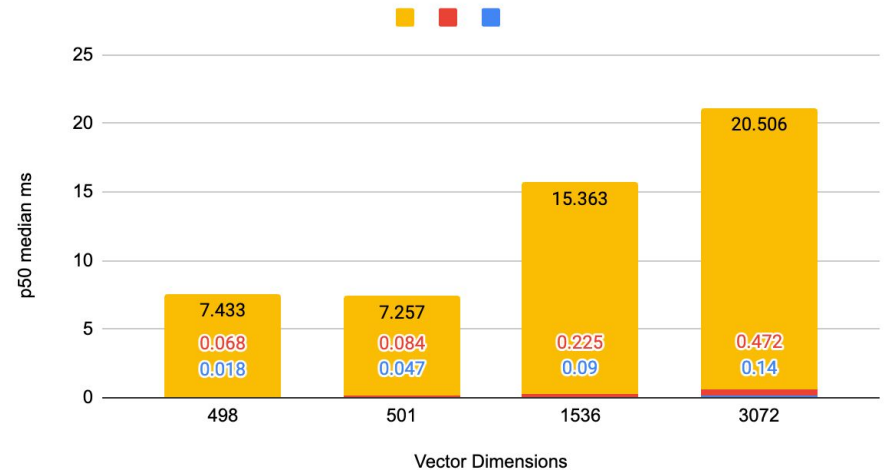
|                          | No Index | HNSW      | IVFFLAT  | TOAST chunks |
|--------------------------|----------|-----------|----------|--------------|
| Update 1M 498 dimension  | 0.018 ms | 7.433 ms  | 0.068 ms | 0            |
| Update 1M 501 dimension  | 0.047 ms | 7.257 ms  | 0.084 ms | 2            |
| Update 1M 1536 dimension | 0.090 ms | 15.363 ms | 0.225 ms | 4            |
| Update 1M 3072 dimension | 0.140 ms | 20.506 ms | 0.472 ms | 7            |

1. Latency measured for p50 median
2. Numbers can be different on your system and shown only for comparison
3. For 3072 dimensions a casting to halfvec was used to build HNSW and IVFFlat indexes

# DML - Update

- Depends on TOAST - TOAST slows down updates
- Depends on dimensions - how many TOAST chunks
- Indexes have significant performance impact
- HNSW is more expensive in overhead than IVFFlat
- Dead tuples for inline vectors and in TOAST
- Indexes fragmentation
- TOAST bloating

Vector DML, update



# Some Recommendations

- Keep vector data in separate table if it is feasible
- Dimensions and size matters - check if you can reduce it
- Indexes have significant performance impact on DML
- HNSW is more expensive in overhead than IVFFLAT
- But HNSW might be better choice if your data are static
- Use partitioning to reduce search area

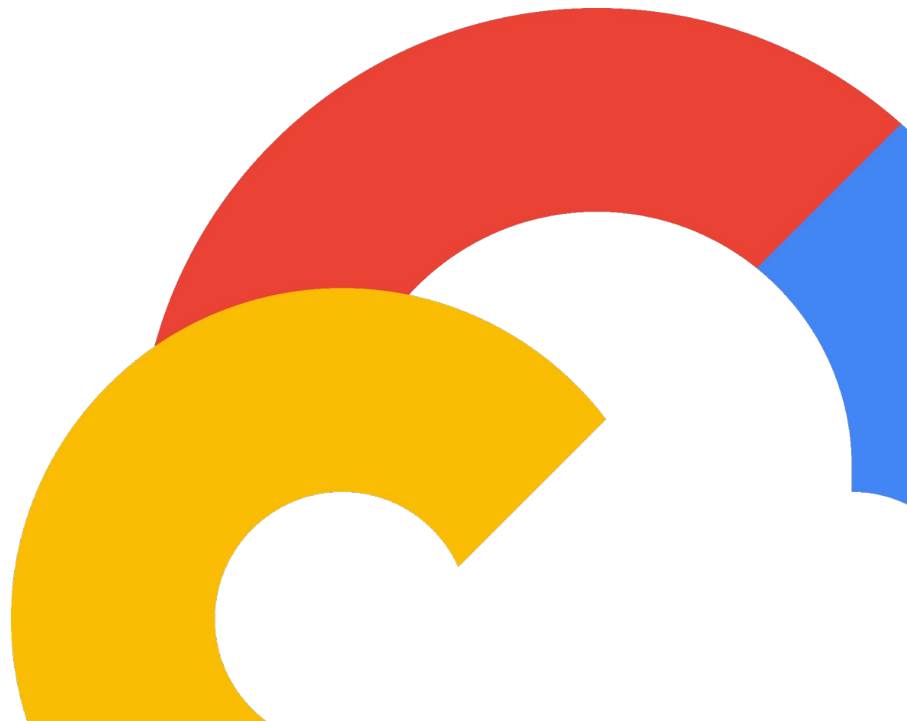
Here is supposed  
to be a very  
opinionated slide  
with general  
recommendations

...



# Q&A

Google Cloud



# Thank you

Google Cloud

