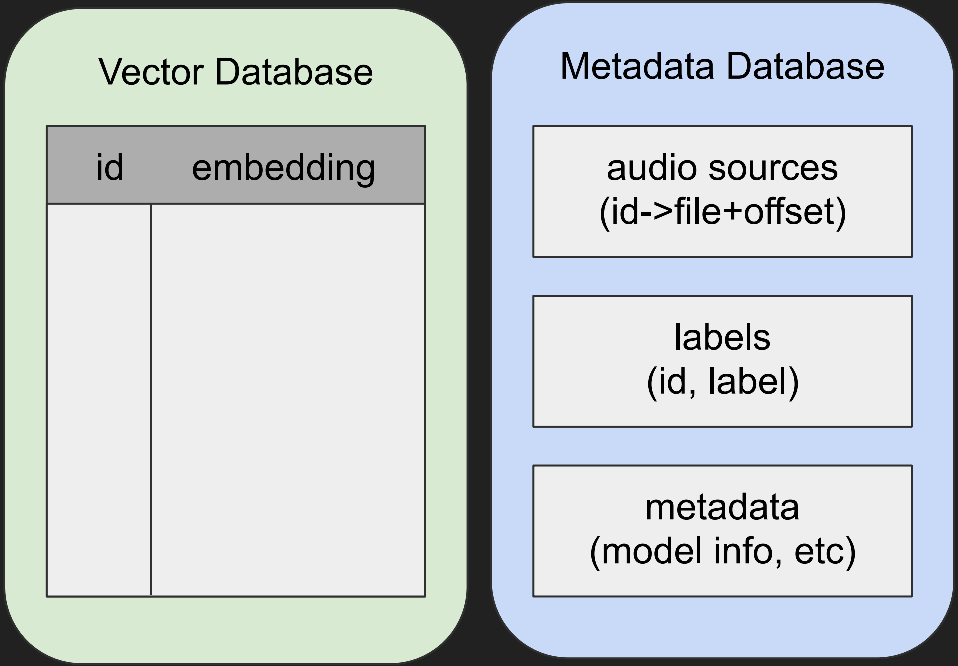
vector databases

db: The beating heart of everything…

A database for managing a combination of embedding vectors and metadata connecting the embedding vectors back to actual audio.

Two essential parts: **Vector database** and **metadata database**.



Similarity Metrics

Given a **query vector**, we need to find similar vectors as efficiently as possible.

We need to choose a **similarity metric**.

Usual choices are:

- Maximum inner product (MIP),
- Cosine similarity,
- (negative) Euclidean distance.

Gory detail:
It is annoying to have to switch between 'up is good' and 'down is good' depending on the metric the user has chosen.

As a result, it is typical to choose a 'good' direction and then redefine some metrics as their negatives.

Brute Force Search

Computing the similarity between the query vector and a vector in the database takes some amount of time.

Brute-force search is the baseline: it finds all of the most similar metrics according to the chosen metric, but may be very slow when working with millions (or billions) of vectors.

Try it out!

Plot the time to compute the dot product of a 1024-dim'l embedding with N vectors across various values of N.

```
query = np.random.normal(size=[1024])
vecs = np.random.normal(
     size=[N, 1024])

st = time.time()
np.dot(vecs, query)
elapsed = time.time() - st
```

Tricks to speed up ~~brute force~~ search…

- **Threading**!
  Numpy can already use some threading in its low-level computations.
  But adding a little bit of threading can still speed things up.

- Careful handling of the **TopKResults**…

- Only search a **random subset** of the data.

- Use **float16** instead of float32 (or float 64):
  ```
  query = np.random.normal(size=[1024], dtype=np.float16)
  ```

The last two only give approximate answers!

Recall vs Latency

If we only work on (say) a 10% subset of the data, then we will run 10x faster, but miss 90% of the relevant results.

Recall@K: Proportion of approximate search results appearing in brute-force Top-K results.

The core metric for vector search is **Recall@Latency**: How quickly can we produce a given level of recall?

Other strategies for approximate nearest neighbor search?

We want approaches which can be computed very quickly while still giving pretty good results…

Two classes of approaches

- Cluster-based
  (which you probably guessed)

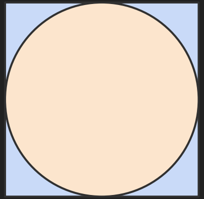
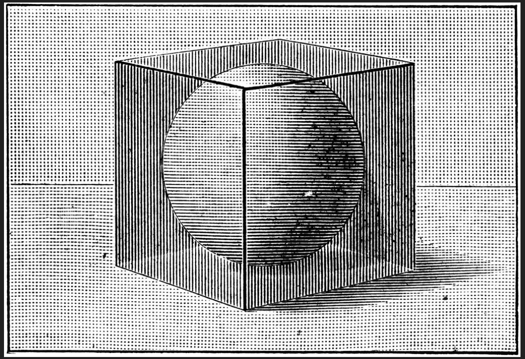
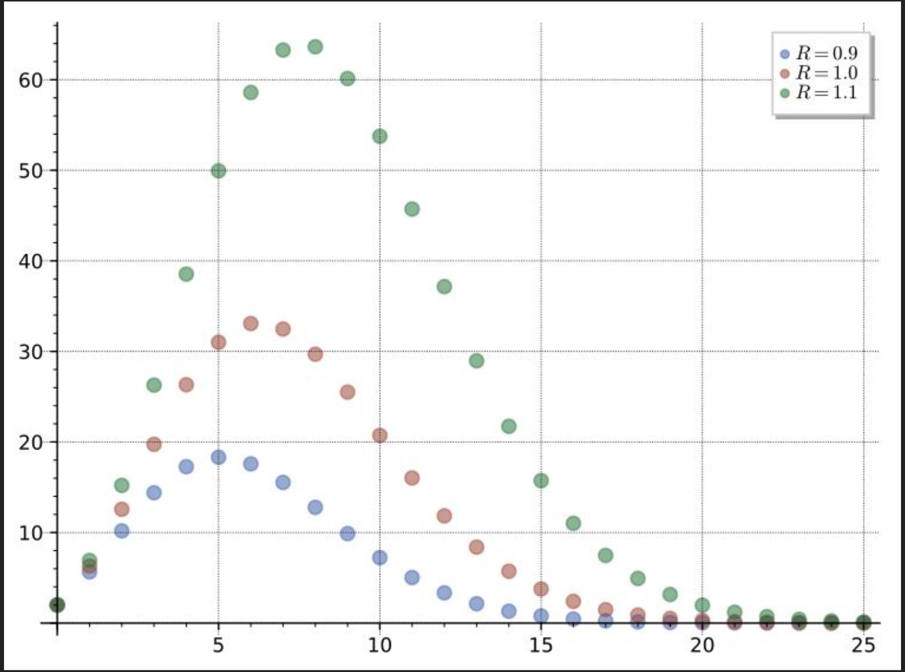
- Graph-based
  (which is considerably more complicated)

In both cases, we spend time **indexing** (ie, arranging) the data before the search is performed to make it easier to get good results quickly.

Core problem: Curse of Dimensionality

When we talk about Euclidean nearest neighbors, the curse of dimensionality rears its head.

The relative volume of a sphere to a cube goes to zero as the dimension increases…

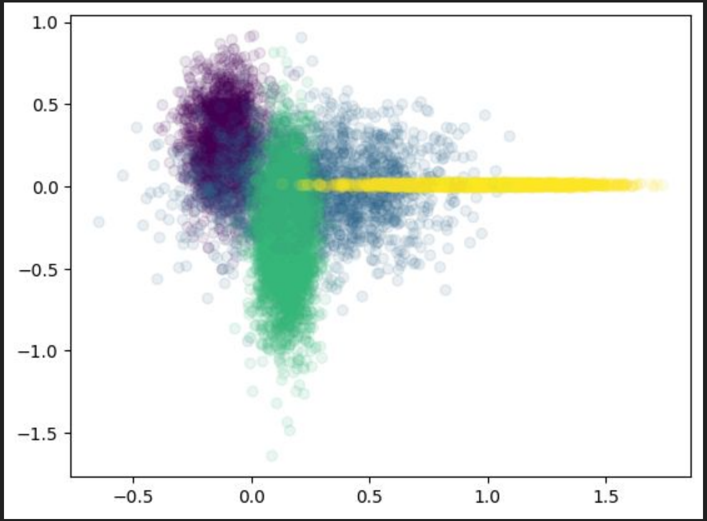
A Basic Clustering Approach

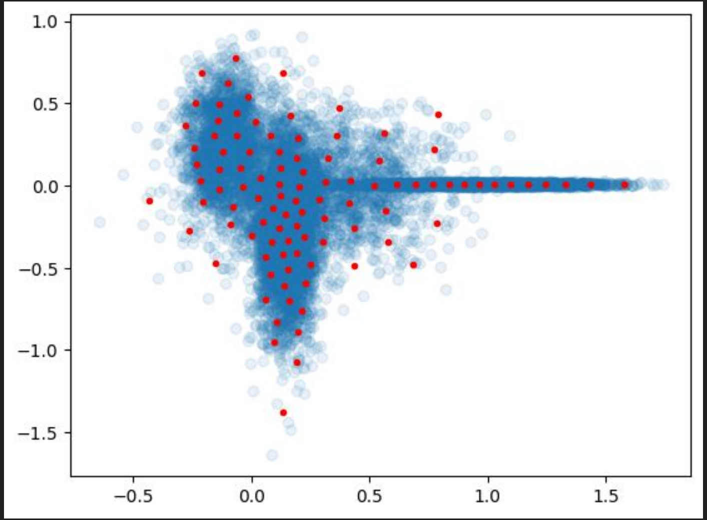
If we have many data points, we can cluster them to try to group together similar things. (In the picture, I generate a mixture of Gaussians to play with.)

Assign each data point to the nearest cluster centroid.

Inference: Find the nearest centroid to the query, then check all assigned points.

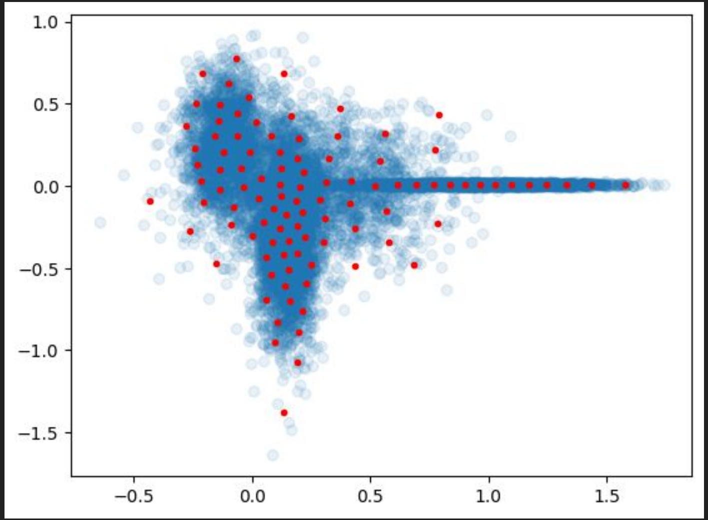
Time: `num_clusters * bucket_size`



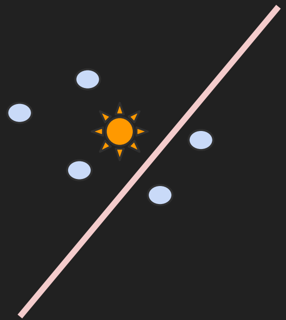


Problems

- Clusters may have very uneven size.

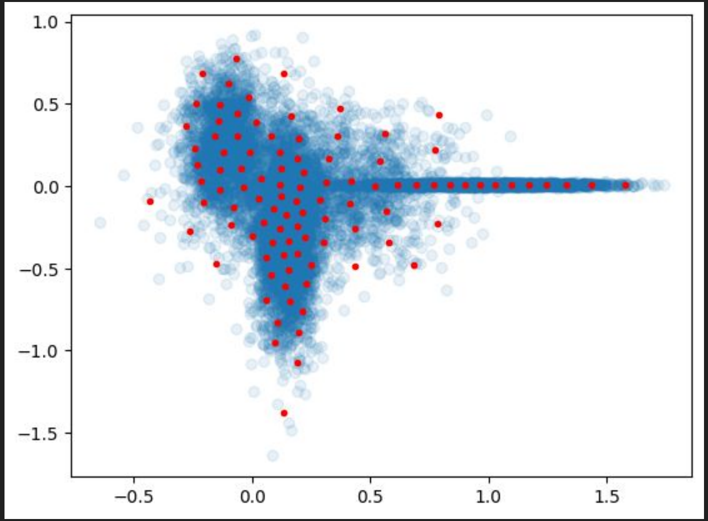


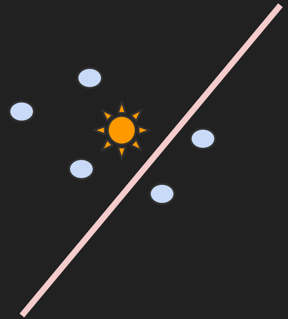
- If the query is near the **boundary** between two clusters, its nearest neighbors will be in the wrong cluster.



Problems

- Clusters may have very uneven size.
  - Many possible fixes, often through modifying the clustering algorithm…
  - Ex: Assign each cluster the TopK', instead of just the nearest. Then each has the same size, but may lose some examples in very dense regions…
- If the query is near the **boundary** between two clusters, its nearest neighbors will be in the wrong cluster.
  - Can search the m clusters nearest to the query.





Recursive K-Means (SCANN)

Once we have billions of vectors to work with, the number of centroids may itself be too high to search quickly.

Then we can cluster the centroids and search those…

Same problems as before, but now happening at each layer!

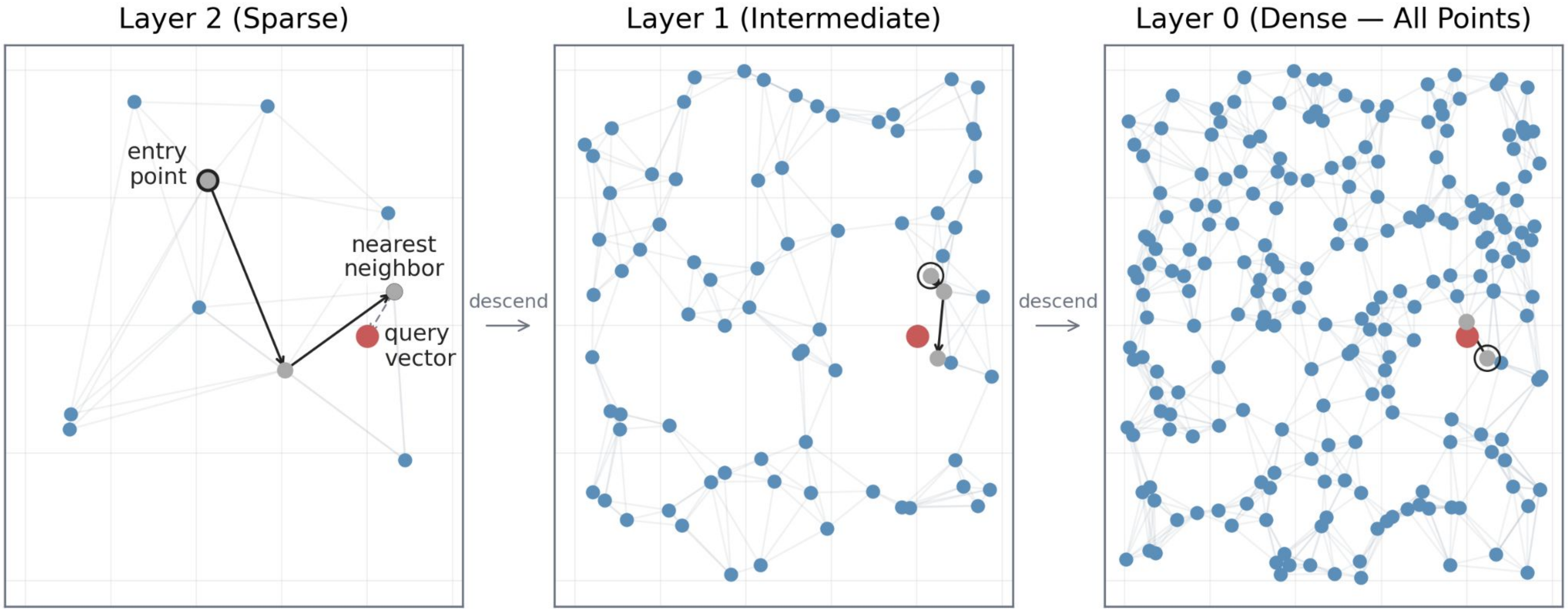
Graph-based Approaches

Graph-based approaches to ANN Search involves placing a (directed) graph structure on the data.

At each point, you can compare a query to its neighbors, and 'walk' in the direction of the neighbor which looks nearest to the query.

The problem is then finding good strategies to choose the graph edges.

Hierarchical Navigable Small World (HNSW)



Problems in Vector Search

- Many of the most popular libraries are **extremely memory intensive**.
  - This lets them operate very quickly, but can be very expensive.
  - **usearch** allows in-memory or disk-based search, and is awesome.

- Most algorithms need extra work to incorporate **metadata queries**.
  - Pre-filtering: Make many indices, depending on metadata characteristics. Only search the relevant ones.
  - Post-filtering: Search with a very large K, then filter out irrelevant results.

- For large datasets, **indexing time** can become quite large.
  - Especially for graph-based algorithms.

- For extremely large datasets, you may need **multiple indexes**.
  - Adds more software engineering overhead.

A couple last points…

- CLIP allows **multimodal search**!
  - Align the embeddings from two different modalities, then use embedding from text to search for things in the other modality.

- Despite the rough edges,
  ANN Search works quite well in 2025.
  - Can sometimes find interesting new ideas by assuming it's a tool that 'just works.'
  - A few software companies (eg, Pinecone) try to sand off all the rough edges for business use-cases.