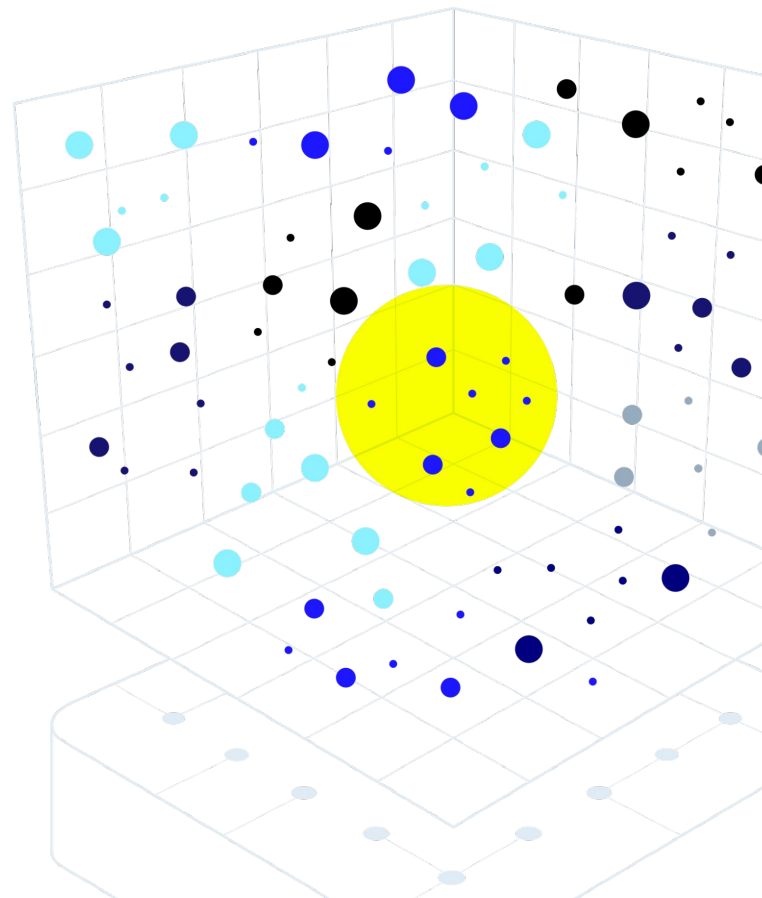


# Generative AI Vector Databases

Ram Sriharsha, VP R&D Pinecone

Presented by  Pinecone



# What is generative AI?

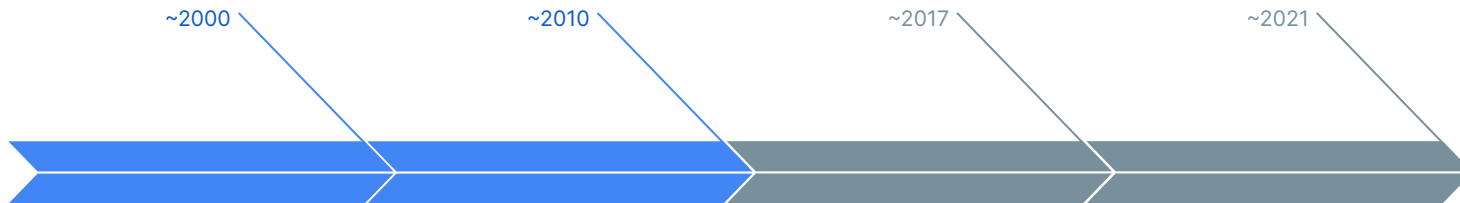


explain generative AI to me in a few sentences



Generative AI is a type of artificial intelligence that creates new data or content, such as images, texts, or sounds, by learning patterns and structures from existing data. It uses techniques like neural networks to generate content that is similar to what it has been trained on, allowing it to produce novel and often realistic outputs, making it a powerful tool for creative tasks and simulations.

# Some recent history on NLP



## Fully supervised w/ hand crafted features

Lots of domain knowledge  
and NLP expertise needed  
to extract salient features

## Fully supervised w/ Neural networks

Co-learn features w/ model  
training

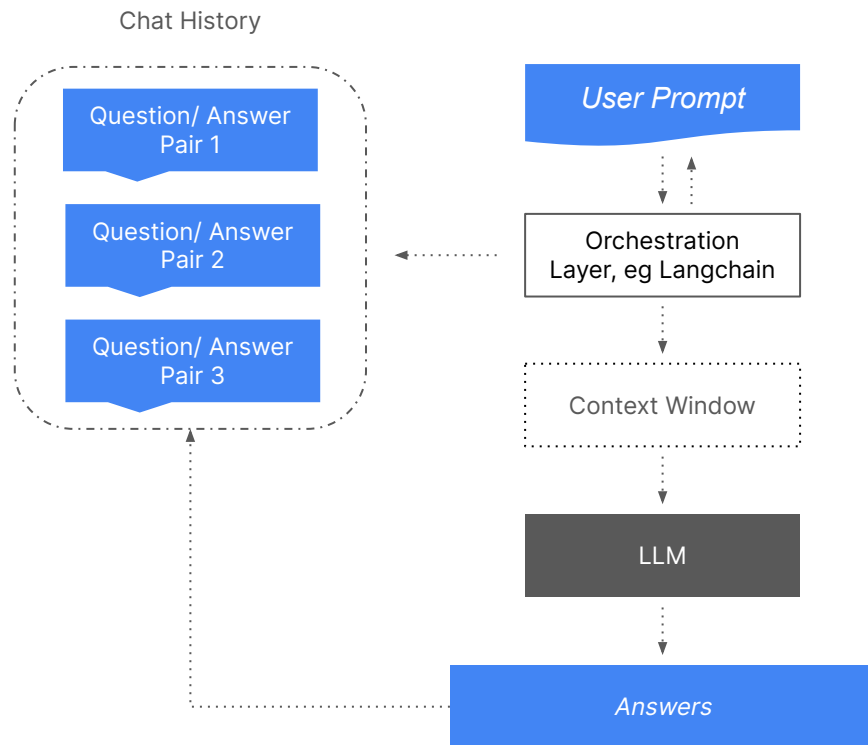
## Pre-train + fine tune

Train a language model (LM)  
to predict probability of text.  
Domain adaptation to  
downstream tasks by fine  
tuning

## Pre-train + Prompting with LLMs

Instead of adapting to  
downstream tasks, use  
prompting to cast the  
problem as original task

# Generative AI workflow



# LLM Challenges

| Architectural Limits   | Hallucination   | Attribution   | Privacy  |
|--|---|---|--|
| <ul style="list-style-type: none"><li>• costly to retrain to keep up with evolving knowledge bases</li><li>• Finite context window, so no long term memory</li></ul> | <ul style="list-style-type: none"><li>• Providing the incorrect answers confidently</li><li>• Longer context windows worsen the problem</li></ul> | <ul style="list-style-type: none"><li>• Attaching sources to relevant content</li></ul> | <ul style="list-style-type: none"><li>• The ability to retract (protected) data</li><li>• The ability to control who gets access to what knowledgebase</li></ul> |

# Incorporating knowledge using RAG

**User: What pinecone pod type should I use if I need high throughput?**

**GPT-3.5:** If you are using Pinecone, a vector similarity search engine, for high-throughput workloads, you should consider using the "large" pod type. The "large" pod type is specifically designed for high-throughput scenarios and offers more compute resources, allowing for faster indexing and querying of embeddings.

The "large" pod type in Pinecone provides higher CPU and memory resources compared to other pod types. It allows for parallel processing, ensuring efficient utilization of computing power during indexing and search operations.

Using the "large" pod type can significantly improve the overall throughput and reduce latency, especially if you have large-scale datasets or require fast response times for similarity search queries.

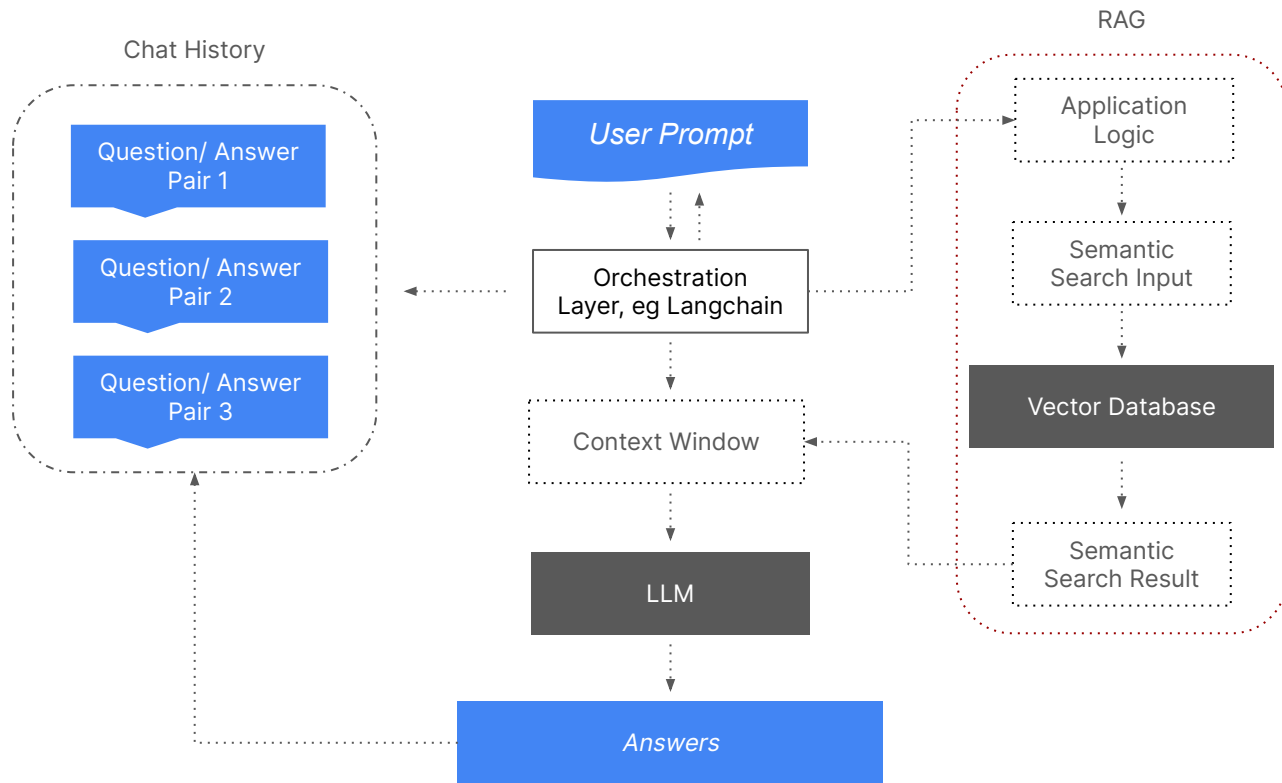
Remember to consider the specific needs of your application and workload when selecting the pod type, as the optimal choice may vary depending on factors like data size, query frequency, and desired response times.

**GPT-3.5 with context (RAG):** If you need high throughput with Pinecone, you should use the p2 pod type. The p2 pod type is designed to provide fast search speeds, with a throughput of around 200 queries per second per replica. It offers approximately 10 times faster speeds and higher throughput compared to the p1 pod type. However, it's important to note that the p2 pod type is currently in public preview and may not be suitable for production workloads.

Source: <<[Link to Source](#)>>

**Solves multiple problems in one shot: attribution, privacy, incorporation of knowledge**

# Generative AI workflow w/ RAG



# What is a vector search?

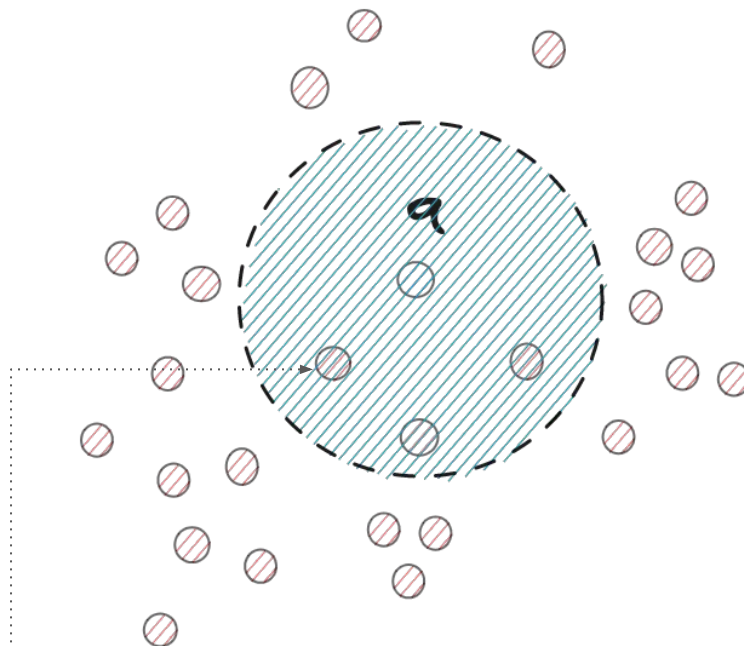
Semantic Search Input



Embedding Model  
Eg Open AI, ada



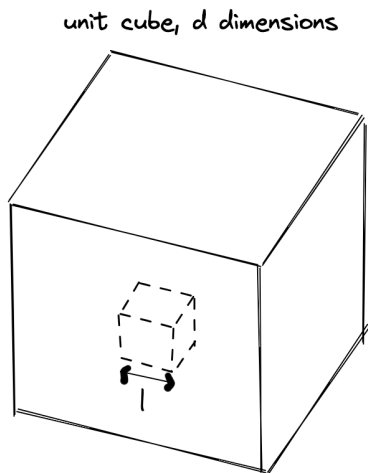
Vector Embedding  
[0.34, -1.2, 0.34, 1.3, ..., -0.03, 1.14]





# Can my database do this efficiently?

Traditional indexes fare very poorly with dimension and size of corpus due to the curse of dimensionality



data points drawn uniformly at random from  $U[0,1]^d$

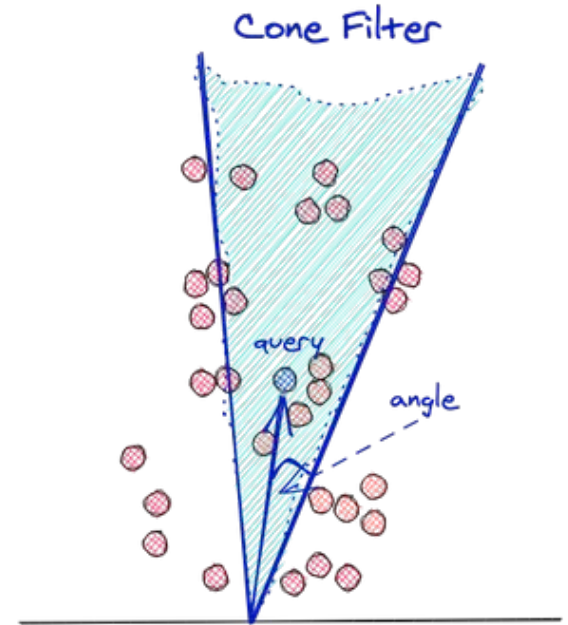
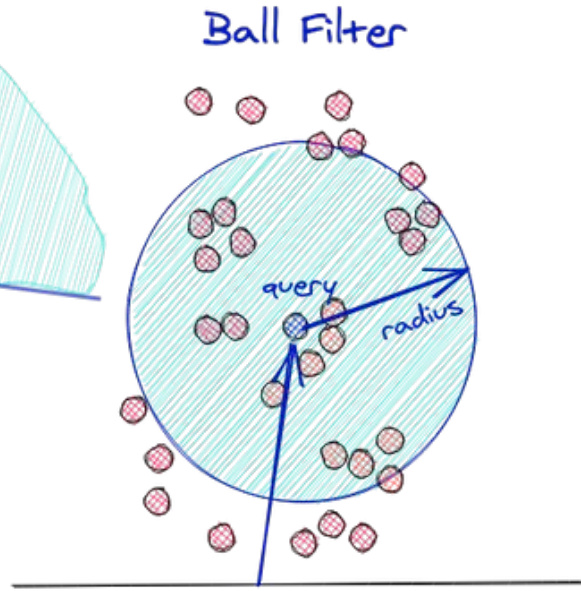
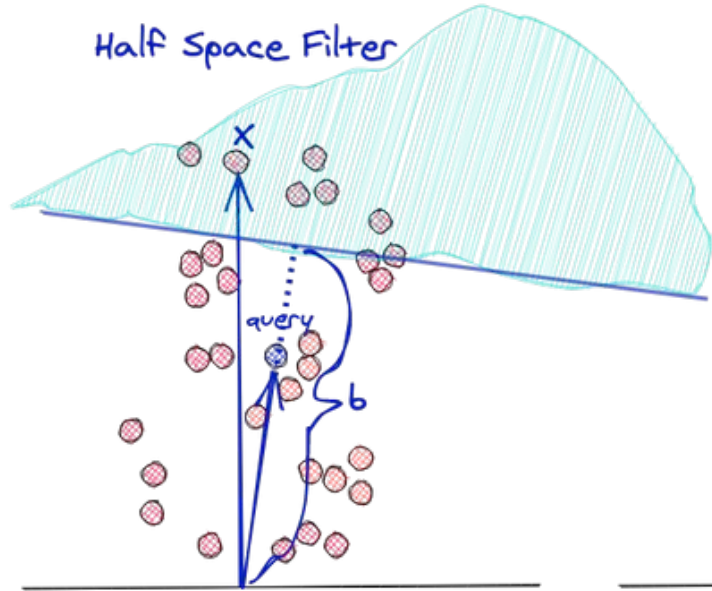
$l \sim (k/n)^{1/d}$  to contain  $k$  nearest neighbors

for  $d = 768$ ,

$n = 1\text{m}$ ,  $l \sim 0.98$

$n = 1\text{b}$ ,  $l \sim 0.97$

# Challenges w/ geometric filters



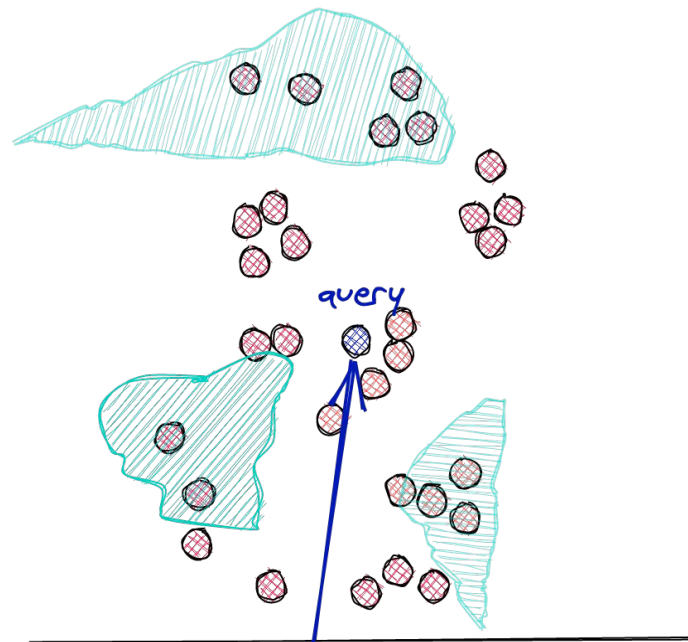
## Ok, maybe we use one of the vector search libraries?

- . Hashing based (LSH)
  - Convert a geometric problem into a hash bucket lookup problem
- . Compression based (PQ)
  - Compress data and perform as much computation on compressed data as possible
- . Spatial Partitioning based (Random Projection Trees, IVF, ...)
  - Partition space via preprocessing to reduce regions scanned
- . Graph based (HNSW, DiskANN etc)
  - Generate neighborhood graph in advance and greedy search
- . Multi-stage pipelines (IVF + PQ, ...)

# Challenges w/ metadata filters

```
“$and”:[  
  {“region”:{“$eq”：“...”},  
  {“price”:{“$lt”：10.0}]
```

Arbitrary Filter



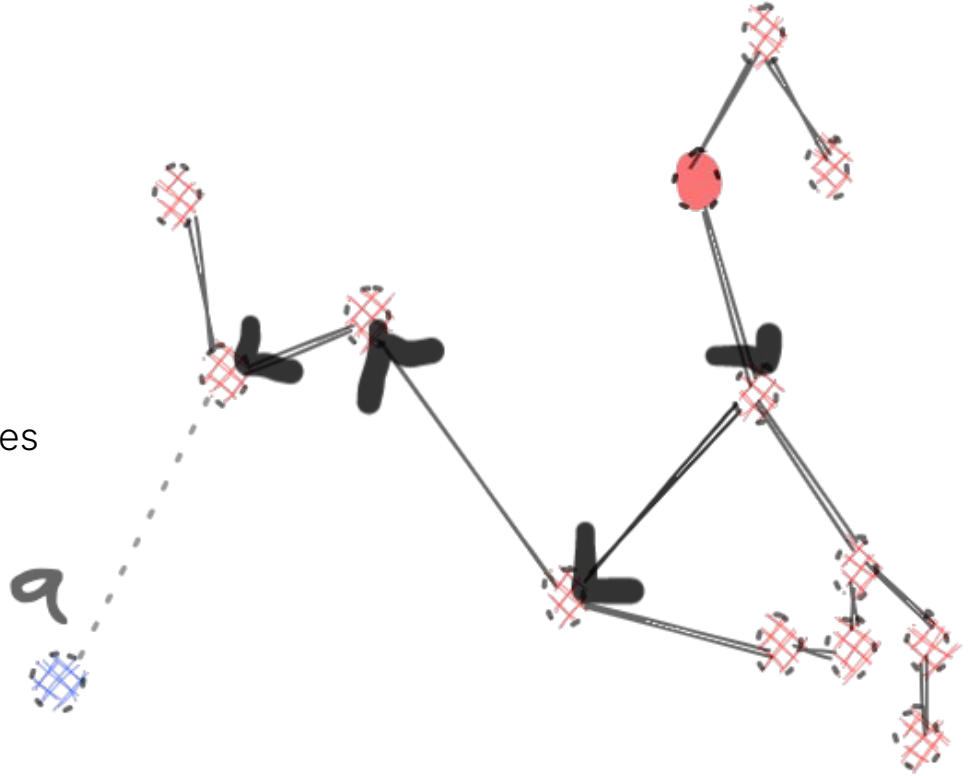
# Graph index

Preprocessing phase:

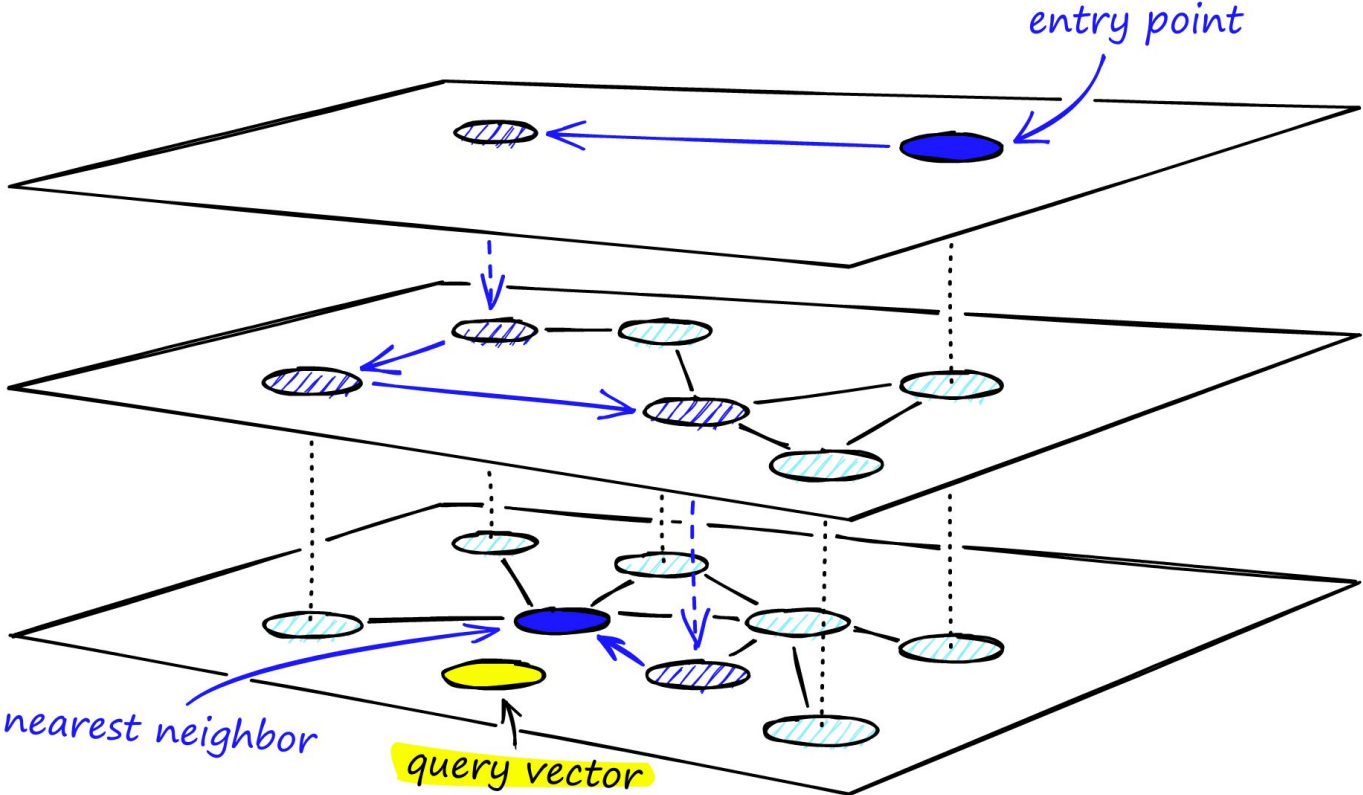
Build neighborhood graph (N)

Query phase:

- . Start at a random node in N
- . Greedy search through graph until stuck
- . Best first traversal to collect top-k matches

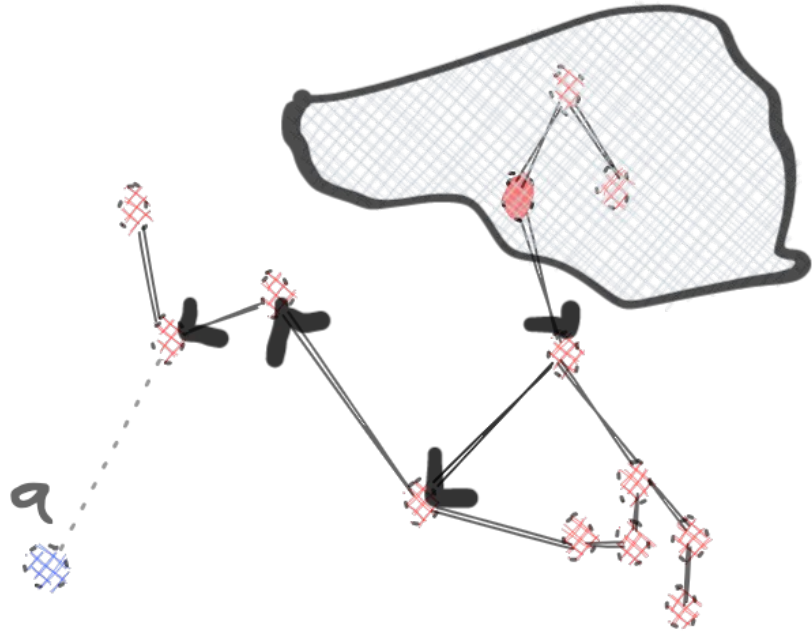


# HNSW, Hierarchical Navigable Small World Graphs



# Challenges w/ filters

Worst case, this can be even worse than brute force search!



# Challenges w/ index management

| <b>Forces Freshness/Cost tradeoff</b>   | <b>Challenges w/ concurrent workloads</b>  | <b>Expensive</b>  | <b>Hard to scale</b>  |
|---|--|---|---|
| <ul style="list-style-type: none"><li>. IVF-PQ for example requires build retraining for data shift</li><li>. HNSW is an incremental index but is slow to index</li></ul> | <ul style="list-style-type: none"><li>. Concurrent updates/ deletes alongside queries require significant effort to manage</li></ul> | <ul style="list-style-type: none"><li>. Usually RAM based and store large vectors in memory/ on local SSD causing costs to explode irrespective of workload</li></ul> | <ul style="list-style-type: none"><li>. Scaling out is entirely DIY</li><li>. Only effective mechanism for queries is scatter gather which introduces high tail latencies</li></ul> |



# Properties of a vector database

| <b>Geometry Aware</b>  | <b>Adaptive Indexes</b>   | <b>Controlled Tradeoff b/w Recall &amp; Cost</b>   |
|--|---|--|
| <ul style="list-style-type: none"><li>• Provide first class support for efficient geometric filters</li><li>• Perform efficient metadata filtering</li></ul> | <ul style="list-style-type: none"><li>• The indexes they construct are adaptive to updates, deletes etc</li></ul> | <ul style="list-style-type: none"><li>• Unlike traditional databases, they don't provide exact answers but aim for high recall for a given cost budget</li></ul> |

# Pinecone

- . Self-serve platform for vector search
- . Create, manage, scale indexes through API/Console
- . High freshness
- . Scalable, low latency queries (< 100ms)
- . Performance and cost optimized instances
- . No tuning / parameter tweaking required!

python curl

```
index query(  
  vector=[0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3],  
  top_k=3,  
  include_values=True  
)  
# Returns:  
# {'matches': [{'id': 'C',  
#               'score': 0.0,  
#               'values': [0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3]}]}
```

The screenshot shows the 'Create Index' interface in the Pinecone console. It features a 'NEW' button (highlighted in blue) and an 'EXAMPLE' button. The form includes the following fields:

- Index Name\***: A text input field containing 'gist-960'.
- Dimensions\***: A text input field containing '960'.
- Metric\***: A dropdown menu currently showing 'euclidean'. A dropdown menu is open below it, showing options 's1' and 'p1'.
- Pods\***: A text input field containing '1'.

At the bottom of the form, there is a blue '+ CREATE INDEX' button and a 'CANCEL' button. A 'Show cURL Command' link with a question mark icon is located at the bottom left of the interface.