[COMSE6998-015] Fall 2024 Introduction to Deep Learning and LLM based Generative AI Systems

Lecture 8

Parijat Dube, Chen Wang

Agenda

RAG & Agent

- RAG
  - Semantic Search, Keyword Search, Embeddings, Hybrid Search, ReRank, Generating Answers
- VectorDB
  - KNN, ANN, HNSW
- Functions, Tools and Agents
- **Guest Lecture, Bo Wen from IBM Research**
- LangChain & LlamaIndex Notebooks

Retrieval Augmented Generation

- What is RAG?
  - “**Retrieval Augmented Generation (RAG)** is a sophisticated approach that enhances the capabilities of large language models (LLMs) by integrating retrieval mechanisms with generative models. This synergy allows the model to access a wealth of external knowledge, significantly improving the relevance and accuracy of generated responses [1]”.
- Why we need RAG?
  - Improves accuracy and relevance of LLM outputs
  - Allows access to current information beyond LLM training data
  - Cost-effective alternative to fine-tuning or retraining LLMs

[1] https://www.restack.io/p/retrieval-augmented-generation-answer-rag-vs-semantic-cat-ai

Semantic Search

Traditional Semantic Search vs. RAG

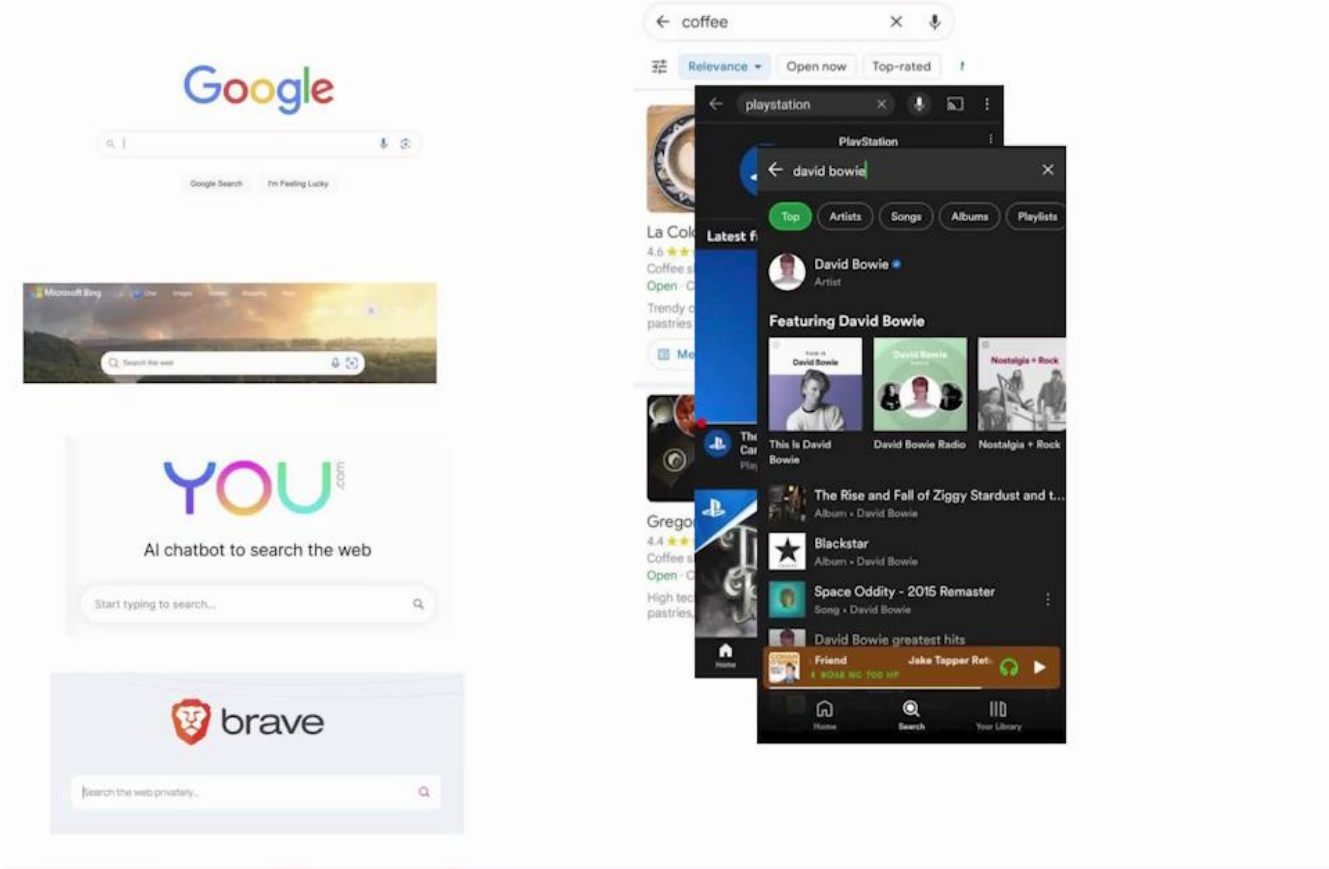
- Semantic Search: Retrieves relevant documents based on meaning and intent
- RAG: Retrieves relevant information AND uses it to generate new, contextual responses

Key Differences

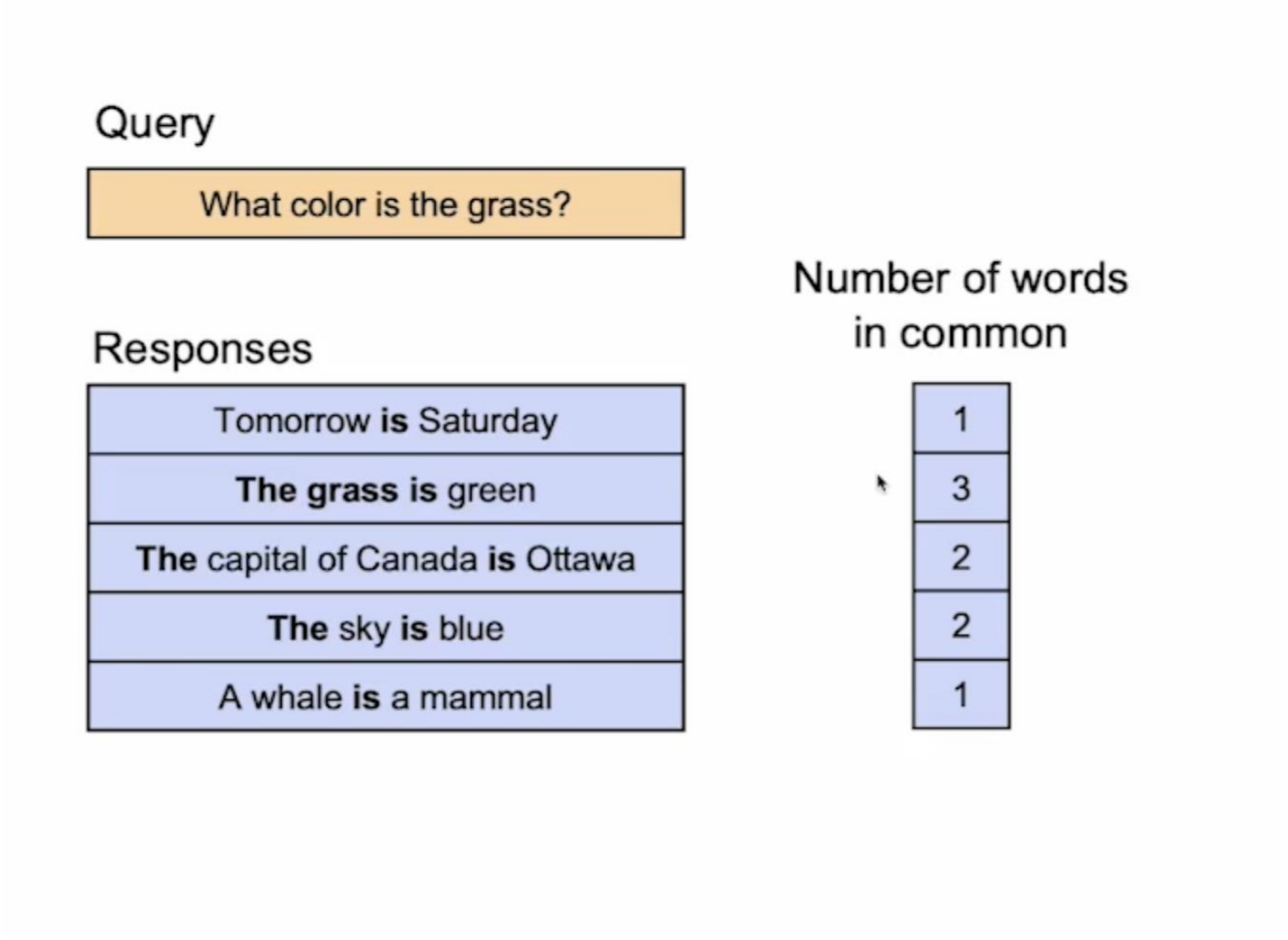
- Semantic Search focuses on retrieval;
- RAG combines retrieval with generation; RAG provides more dynamic and tailored responses

Semantic Search

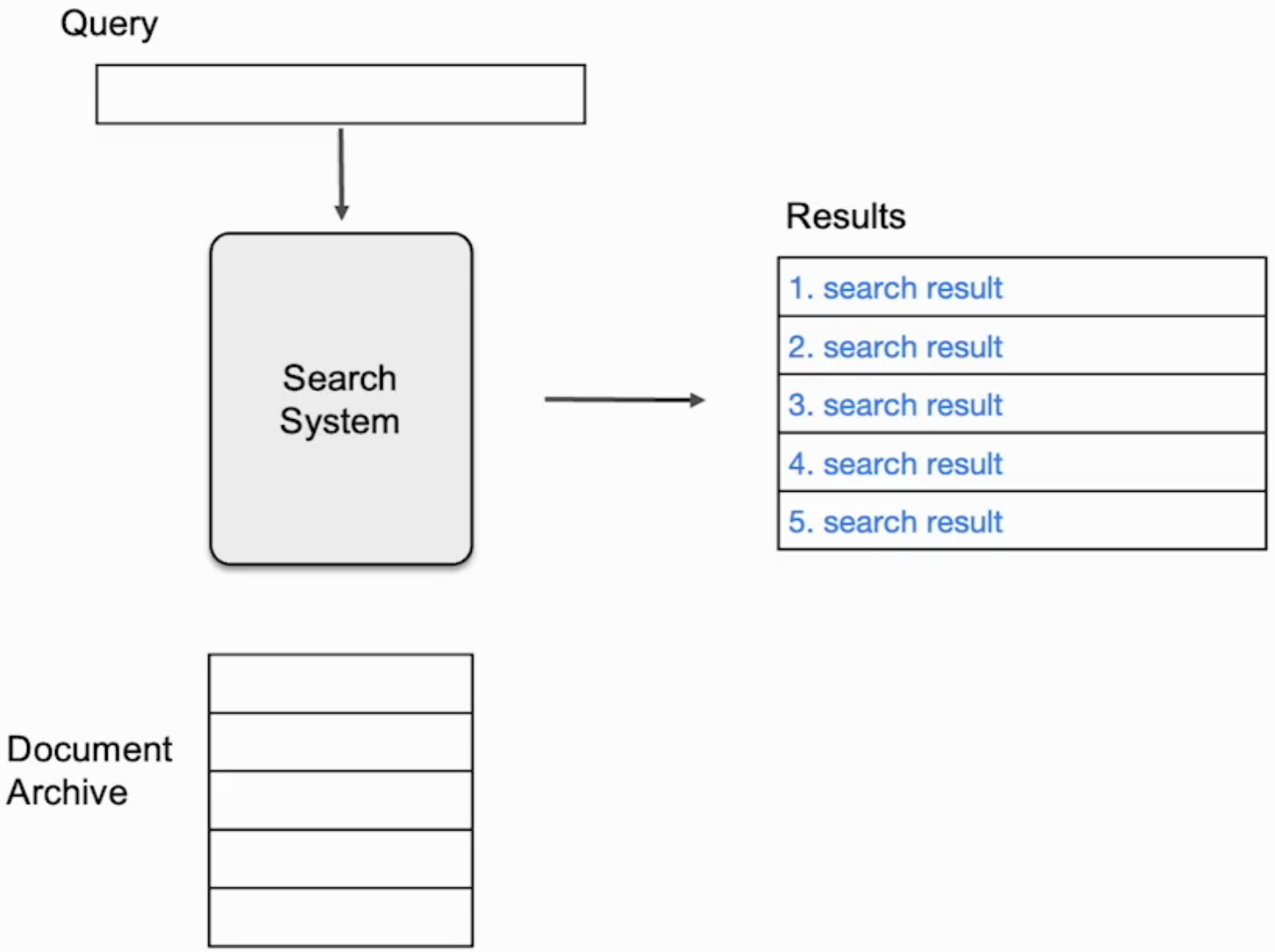
Search is crucial to how we navigate the world



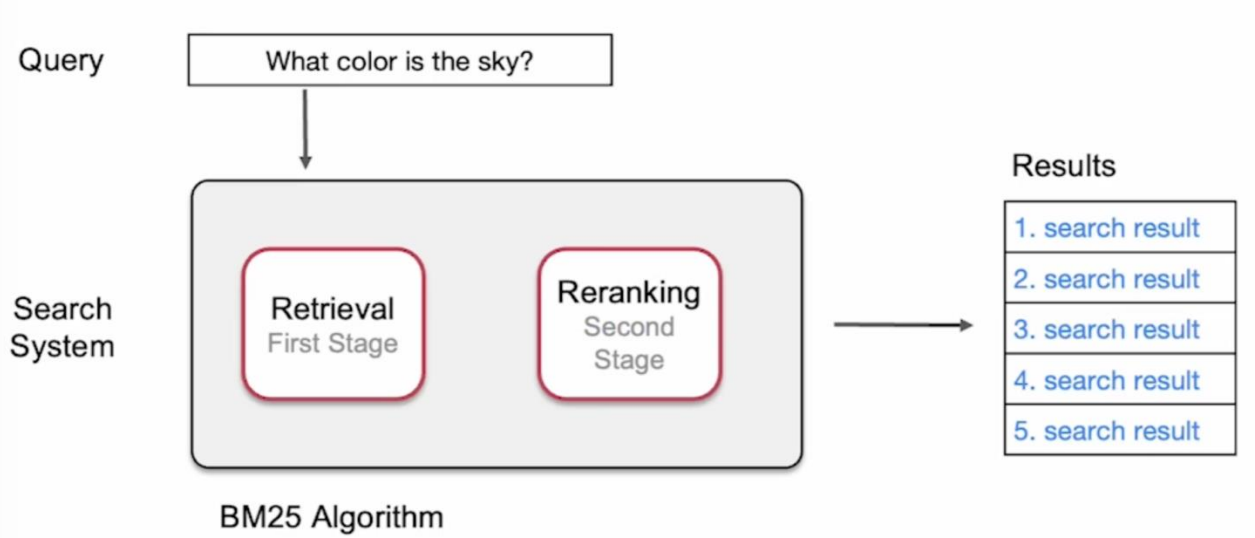
Keyword Search

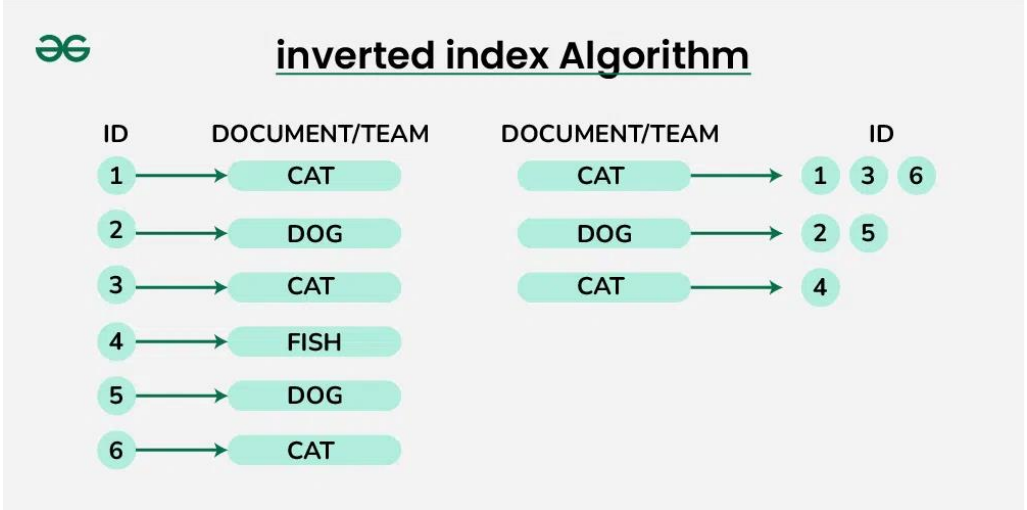


Search at a high level



Keyword Search (Internals)



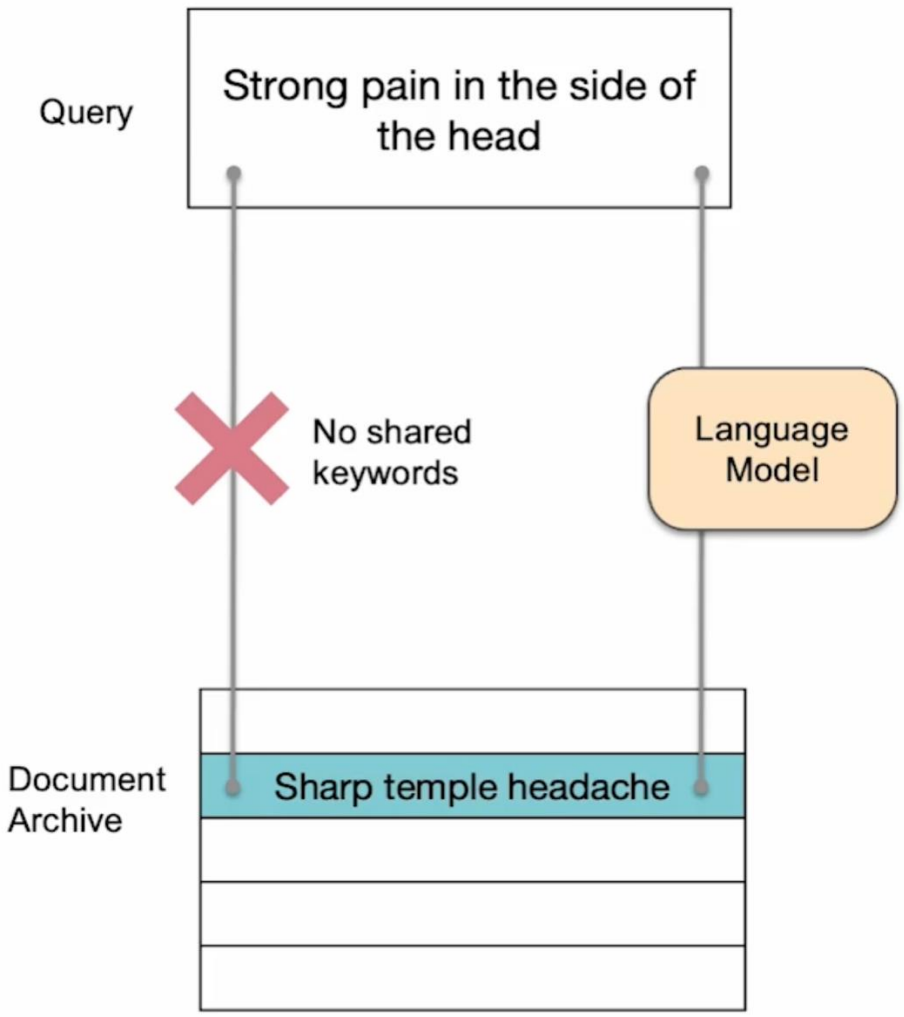


Inverted Index

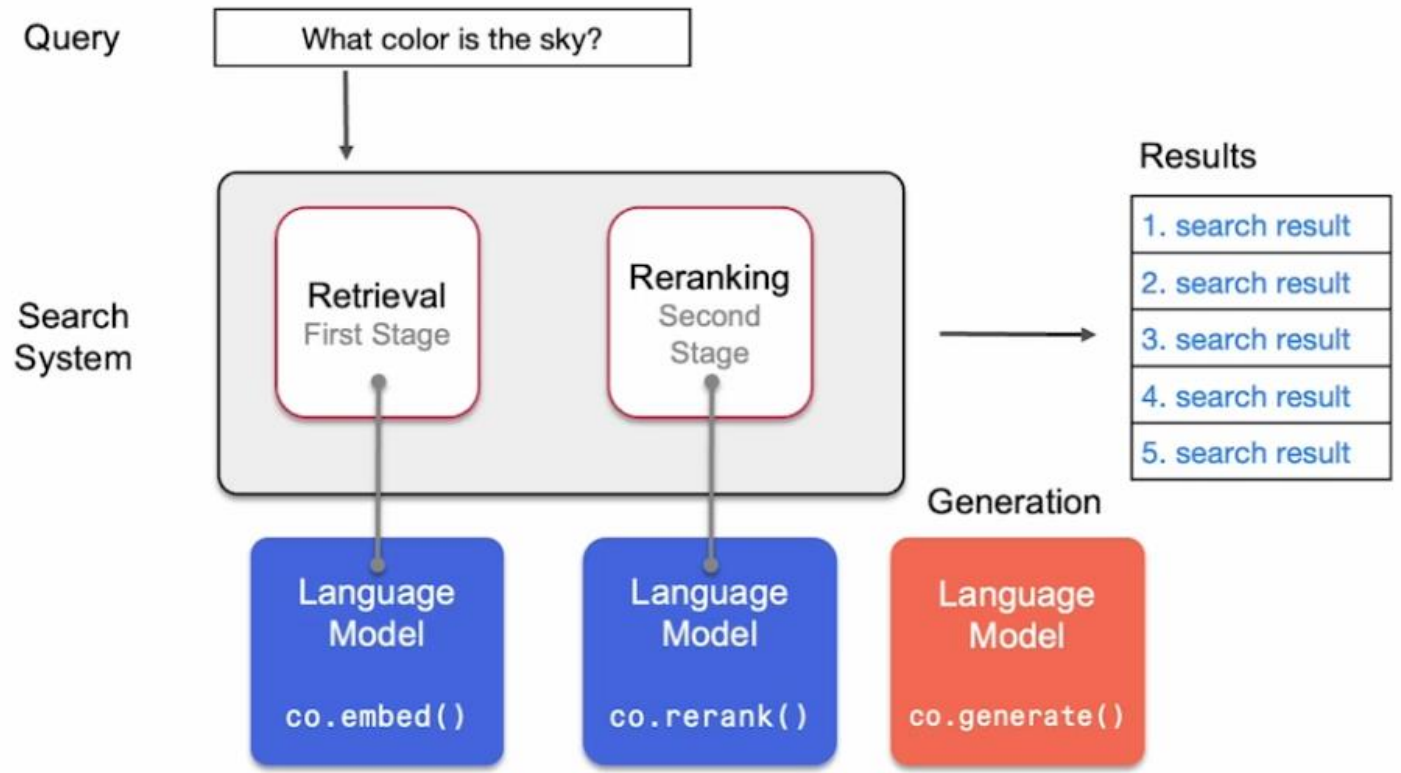
Keyword	Document IDs
abacus	1, 38, 18
...	
Color	23, 804
...	
Sky	804, 922

BM25 Algorithm, https://www.geeksforgeeks.org/keyword-searching-algorithms-for-search-engines/

Limitation of keyword matching

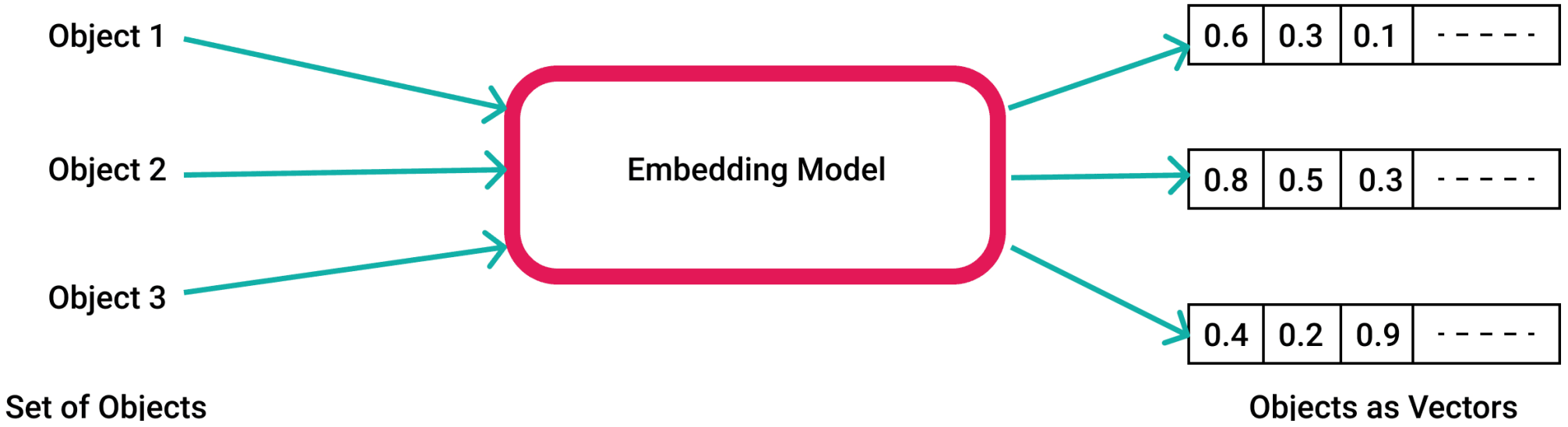


Language Models Can Improve Both Search Stages



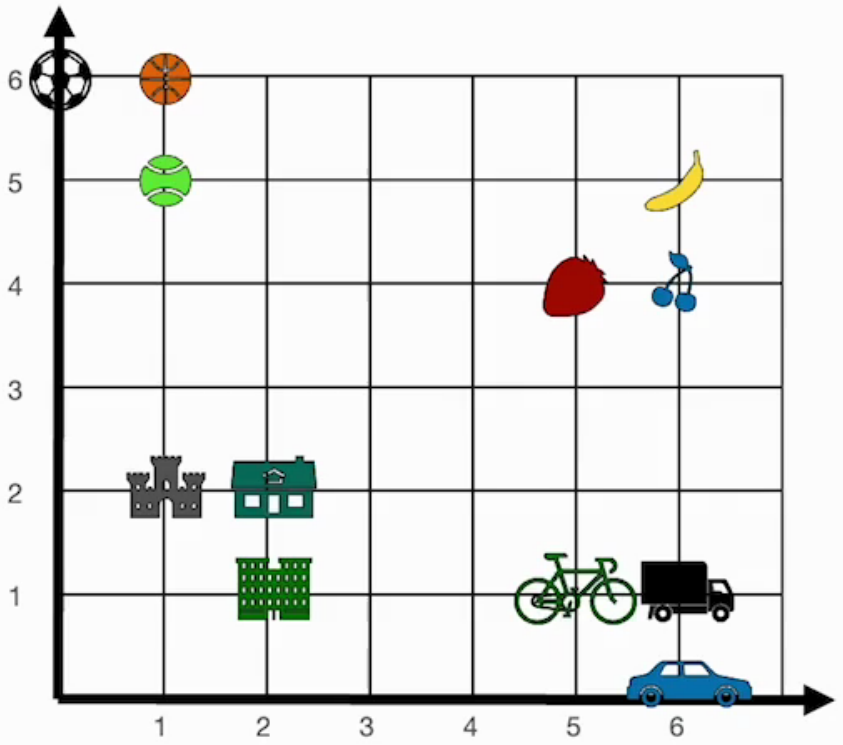
Embeddings

- Embeddings
  - are dense numerical representations of various types of data such as text, images, or audio, transforming them into a low-dimensional space suitable for machine learning.
- Key Points:
  - Dimensionality Reduction: They map high-dimensional data to a lower-dimensional space, capturing important features.
  - Semantic Similarity: Similar items have embeddings close together, allowing tasks like nearest neighbor search.
  - Applications: Used in NLP, computer vision, recommendation systems.
  - Learning from Data: Generated through algorithms like neural networks that learn from input data.



Embeddings

Quiz: Where would you put the word “apple”?



Word	Numbers	
Apple	?	?
Banana	6	5
Strawberry	5	4
Cherry	6	4
Soccer	0	6
Basketball	1	6
Tennis	1	5
Castle	1	2
House	2	2
Building	2	1
Bicycle	5	1
Truck	6	1
Car	6	0

Word Embeddings

Word	Numbers	
Apple	5	5
Soccer	0	6
House	2	2
Car	6	0

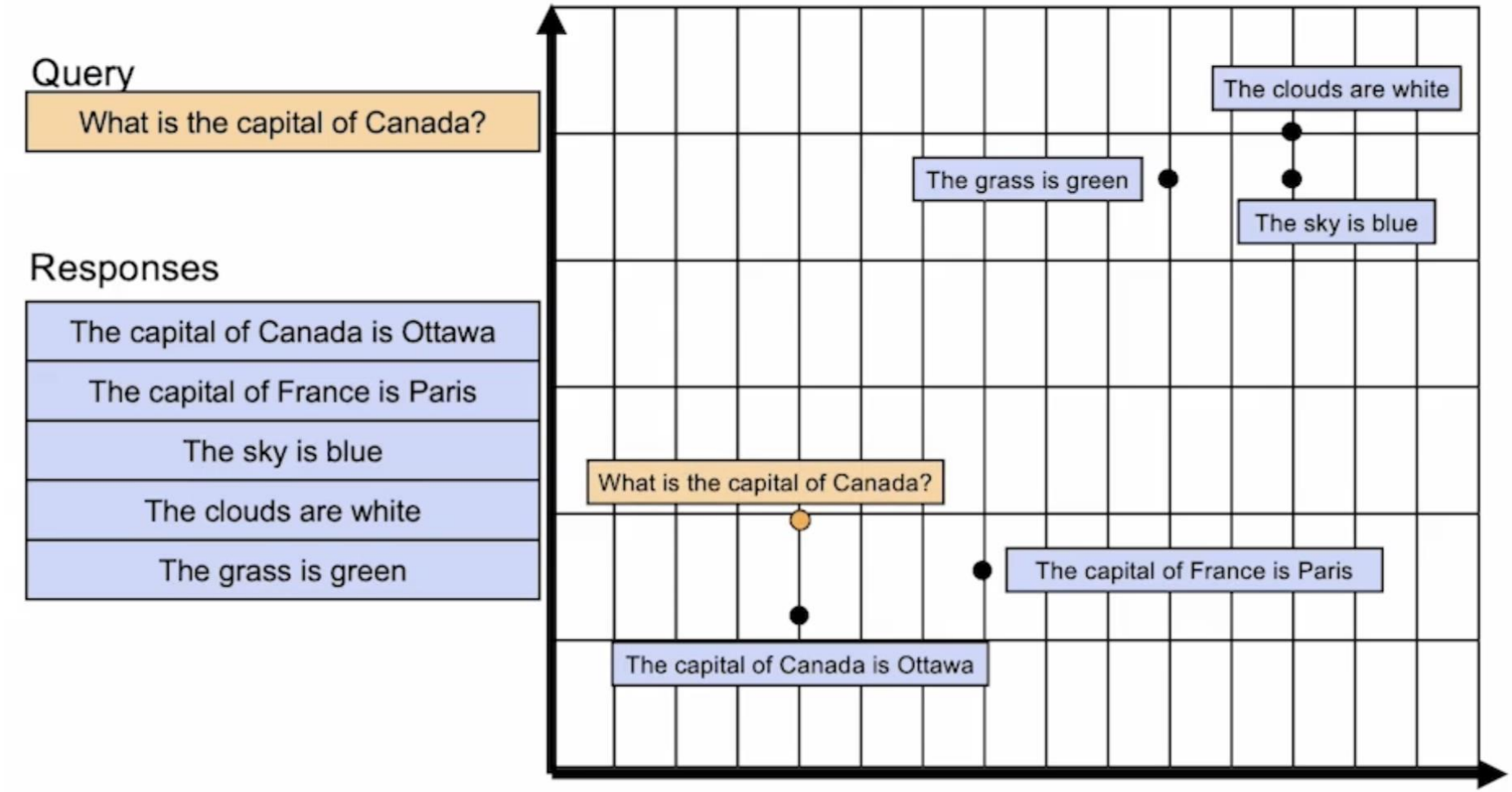
Word	Numbers			
A	-0.82	-0.32	…	0.23
Aardvark	0.42	1.28	…	-0.06
…	…	…	…	…
Zygote	-0.74	-1.02	…	1.35

← Hundreds →

Text Embeddings

Sentence	Numbers				
Hello, how are you?	0.39	0.49	…	-1.01	-0.72
I'm going to school today	-0.79	-0.05	…	-0.94	2.71
…	…	…	…	…	…
Once upon a time	3.23	-0.23	…	-1.45	0.82
Hi, how's it going?	0.41	0.48	…	-0.98	-0.66

Dense Retrieval



Chunking

Query

Do transformers have a recurrent structure?

Transformer (machine learning model)

Article Talk

From Wikipedia, the free encyclopedia

A **Transformer** is a deep learning architecture that relies on the attention mechanism.[1] It is notable for requiring less training time compared to previous recurrent neural architectures, such as long short-term memory (LSTM),[2] and has been prevalently adopted for training large language models on large (language) datasets, such as the Wikipedia Corpus and Common Crawl, by virtue of the parallelized processing of input sequence.[3] More specifically, the model takes in tokenized (byte pair encoding) input tokens, and at each layer, contextualizes each token with other (unmasked) input tokens in parallel via attention mechanism. Though the Transformer model came out in 2017, the core attention mechanism was proposed earlier in 2014 by Bahdanau, Cho, and Bengio for machine translation.[4][5] This architecture is now used not only in natural language processing, computer vision,[6] but also in audio,[7] and multi-modal processing. It has also led to the development of pre-trained systems, such as generative pre-trained transformers (GPTs)[8] and BERT[9] (Bidirectional Encoder Representations from Transformers).

Background [edit]

Before transformers, most state-of-the-art NLP systems relied on gated RNNs, such as LSTMs and gated recurrent units (GRUs), with various attention mechanisms added to them. Unlike RNNs, transformers do not have a recurrent structure. Provided with enough training data, their attention mechanisms alone can match the performance of RNNs with attention added.[1]

Previous work [edit]

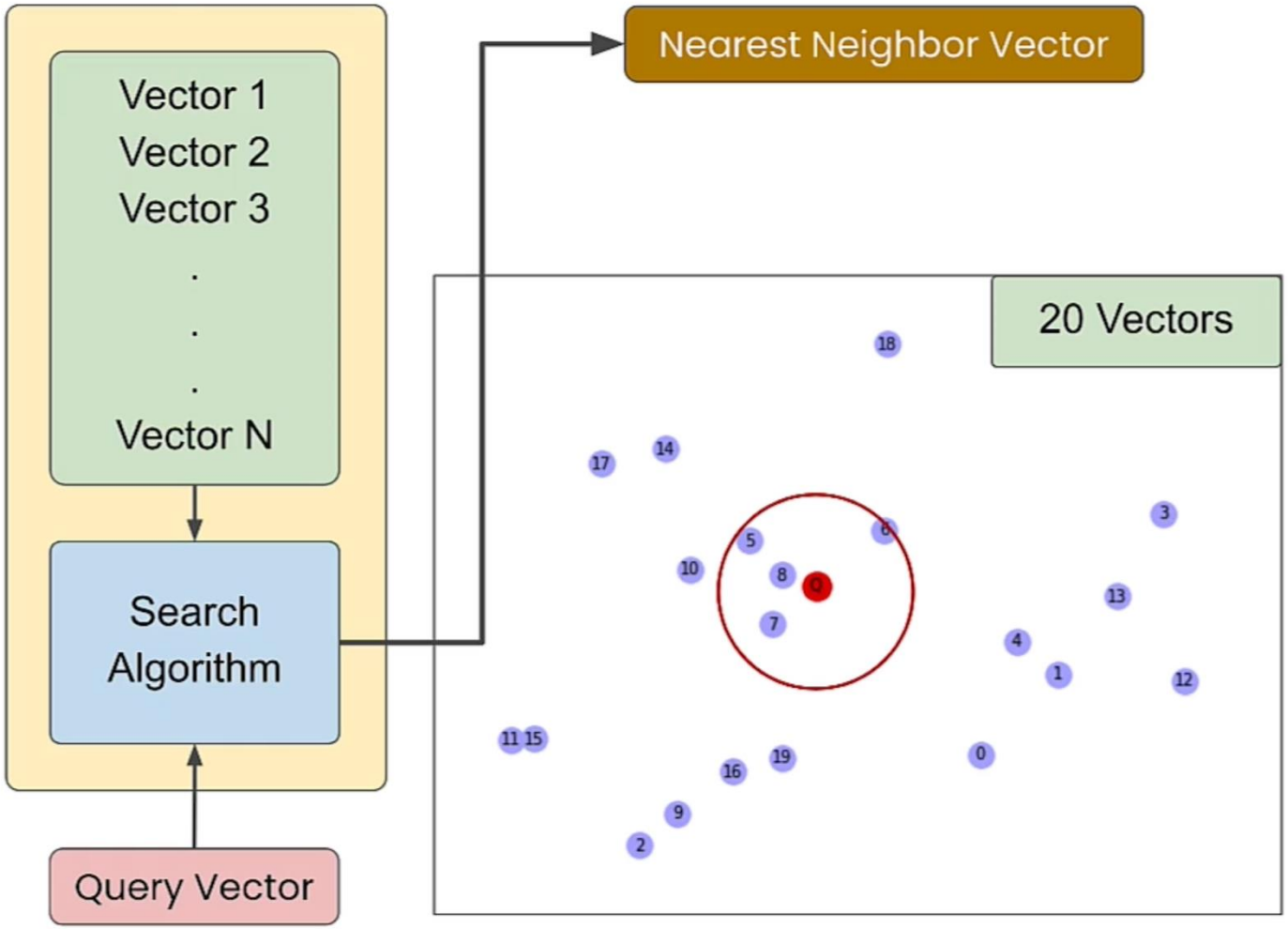
In 1992, Jürgen Schmidhuber published the fast weight controller as an alternative to RNNs that can learn "internal spotlights of attention,"[10] and experimented with using it to learn variable binding.[11]

In a fast weight controller, a feedforward neural network ("slow") learns by gradient descent to control the weights of another neural network ("fast") through outer products of self-generated activation patterns called "FROM" and "TO" which corresponds to "key" and "value" in the attention mechanism.[12] This fast weight is applied to queries. The attention mechanism may be obtained by interposing a softmax operator and three linear operators (one for each of query, key, and value).[12][13]

Flawed sequential processing [edit]

Models, used before the transformer models prevailed, processed the input tokens sequentially, maintaining a state vector representing all the tokens up to the current token in the input. To process the nth token, such a model combined the vector representing the sentence up to token $n-1$ with the information of the new token to create a new state, representing the sentence up to token n. Theoretically, the information from one token can propagate arbitrarily far down the sequence, if at every point the state continues to encode contextual information about the token.

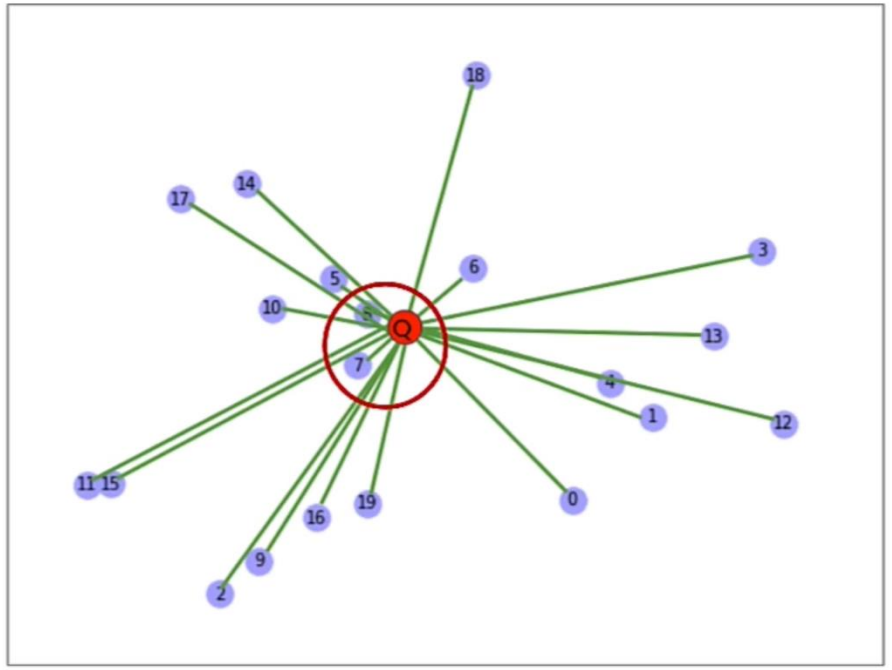
Search for Similar Vectors



Find “K” nearest neighbors to a search query - K-NN

‘brute force’ search algorithm

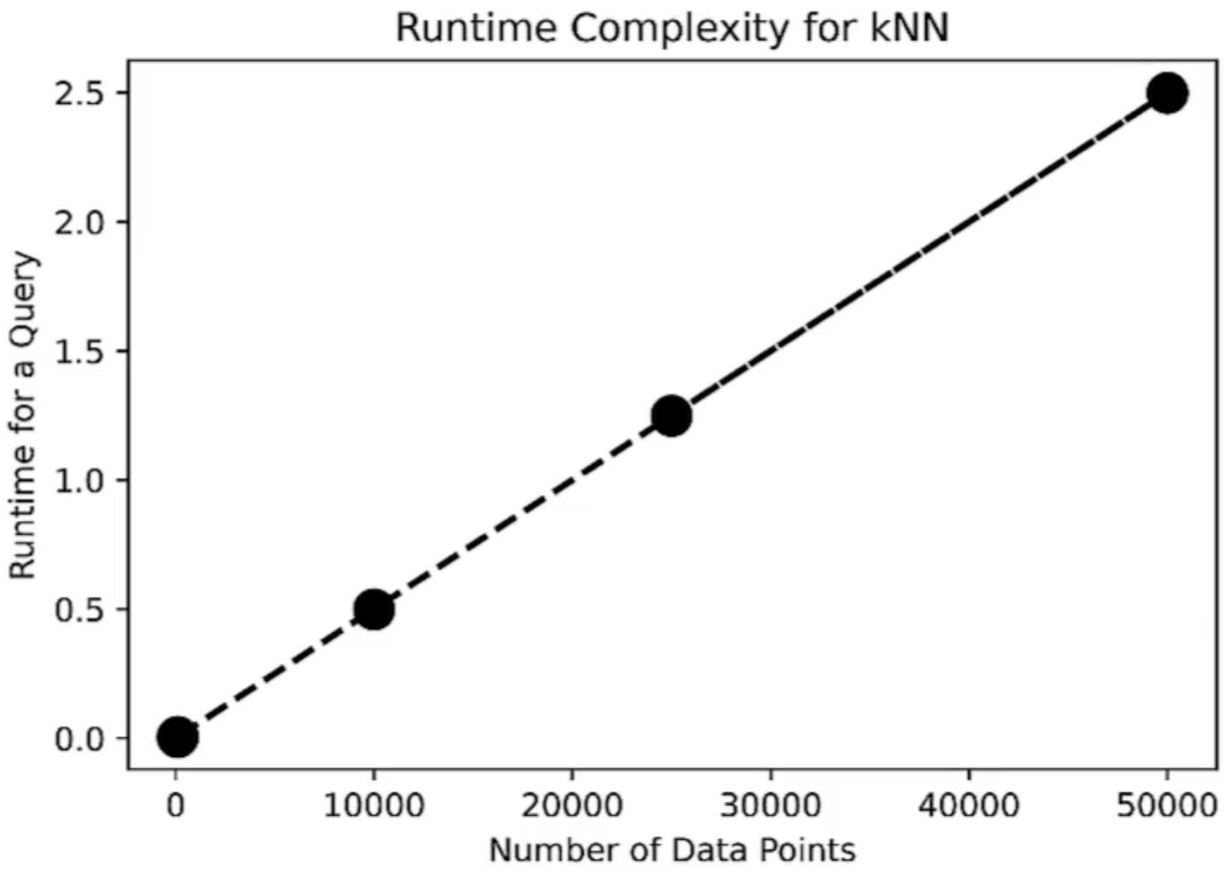
‘brute force’ search algorithm



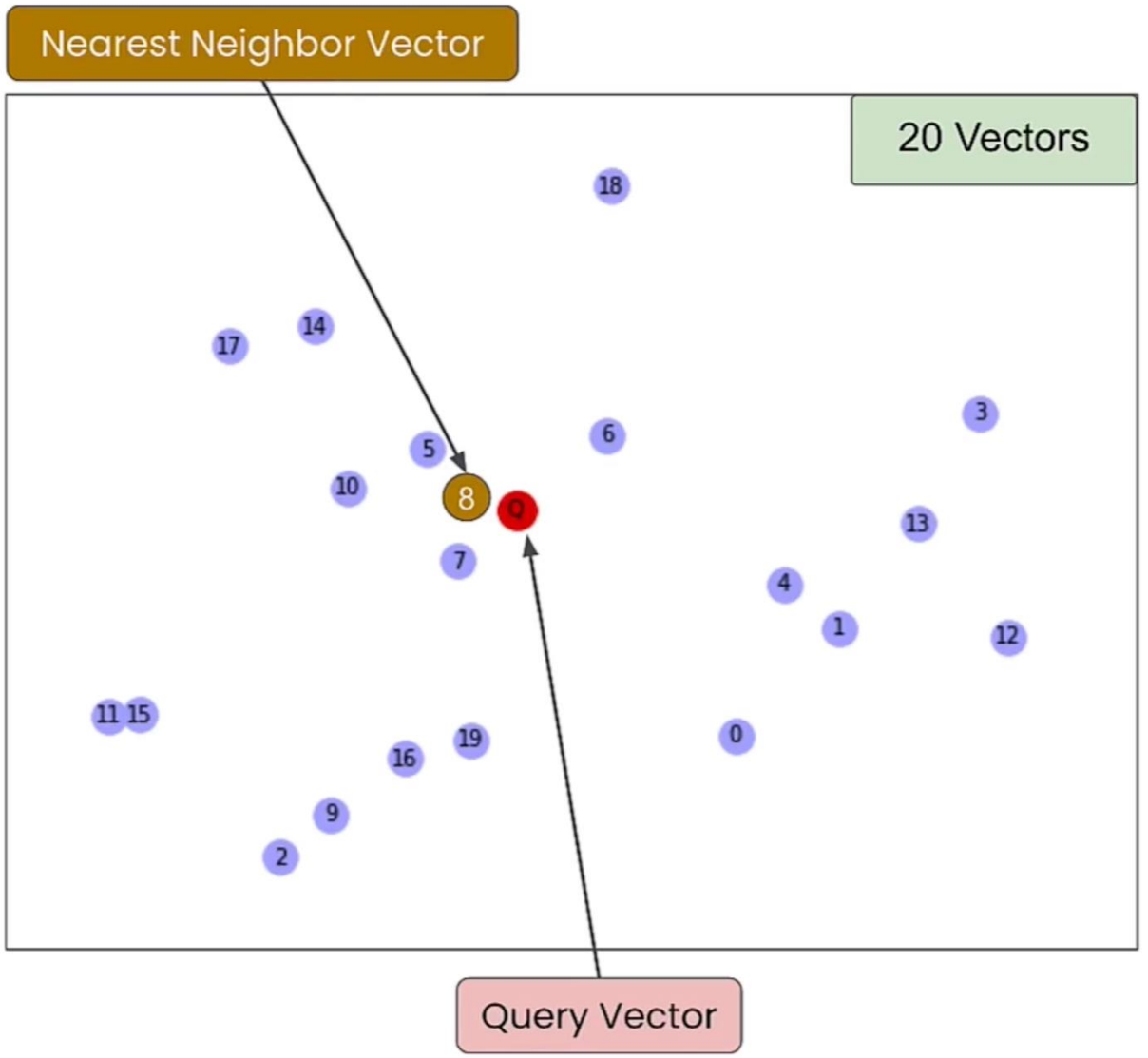
1. Measure the L2 distance between the Query and **each** vector
2. Sort **all** those distances
3. Return the top k matches. These are the most semantically similar points.

Runtime Complexity of KNN

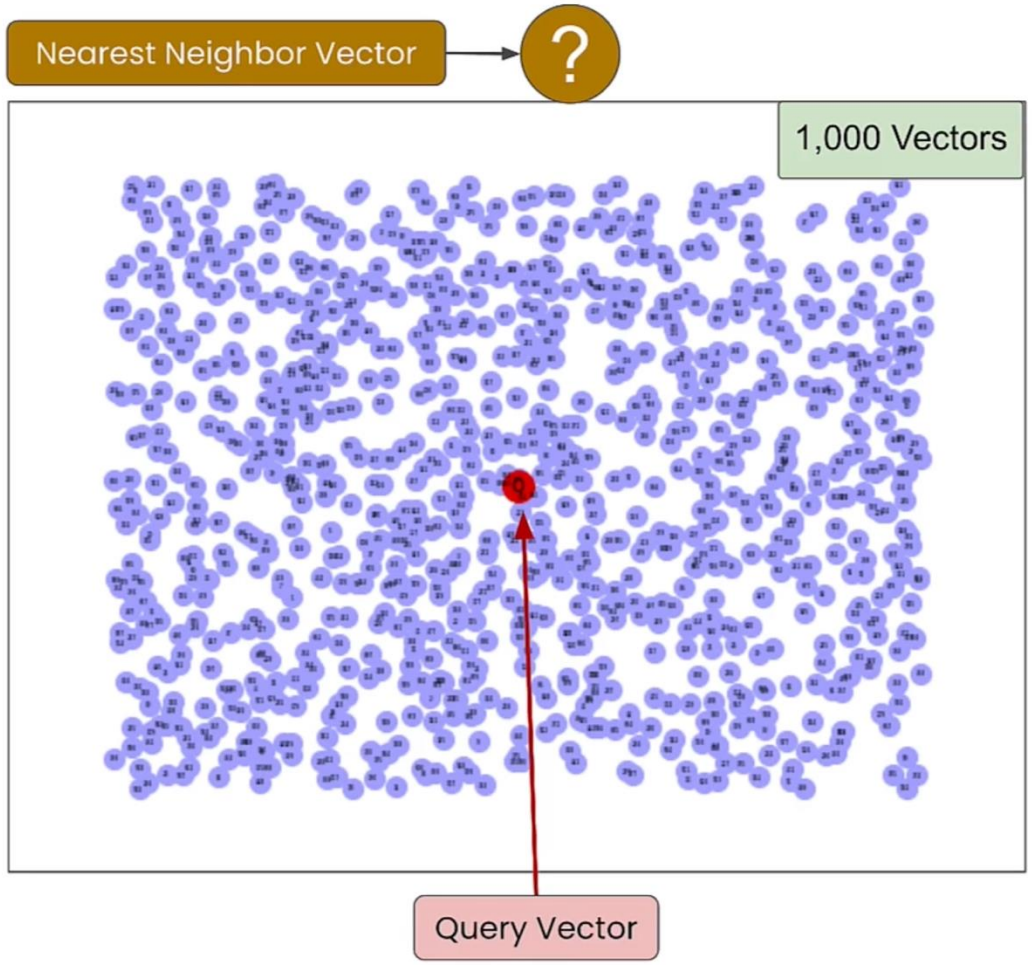
- ***O(dN)* runtime complexity for search**



Not a big problem at small scale

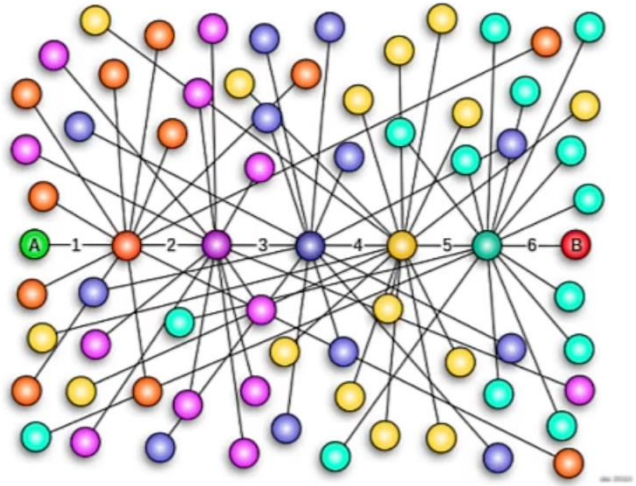


Big problem at scale



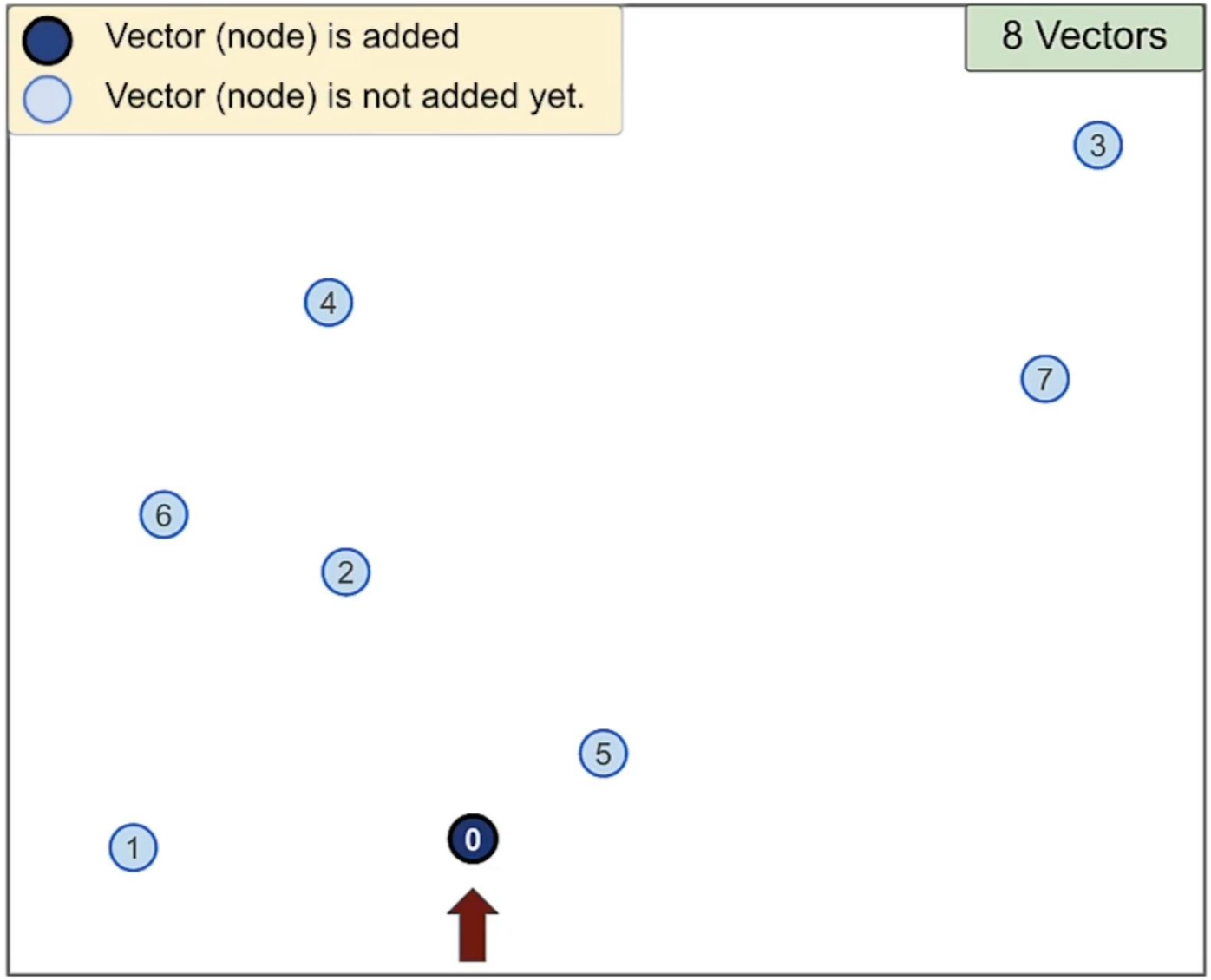
Small World

small world

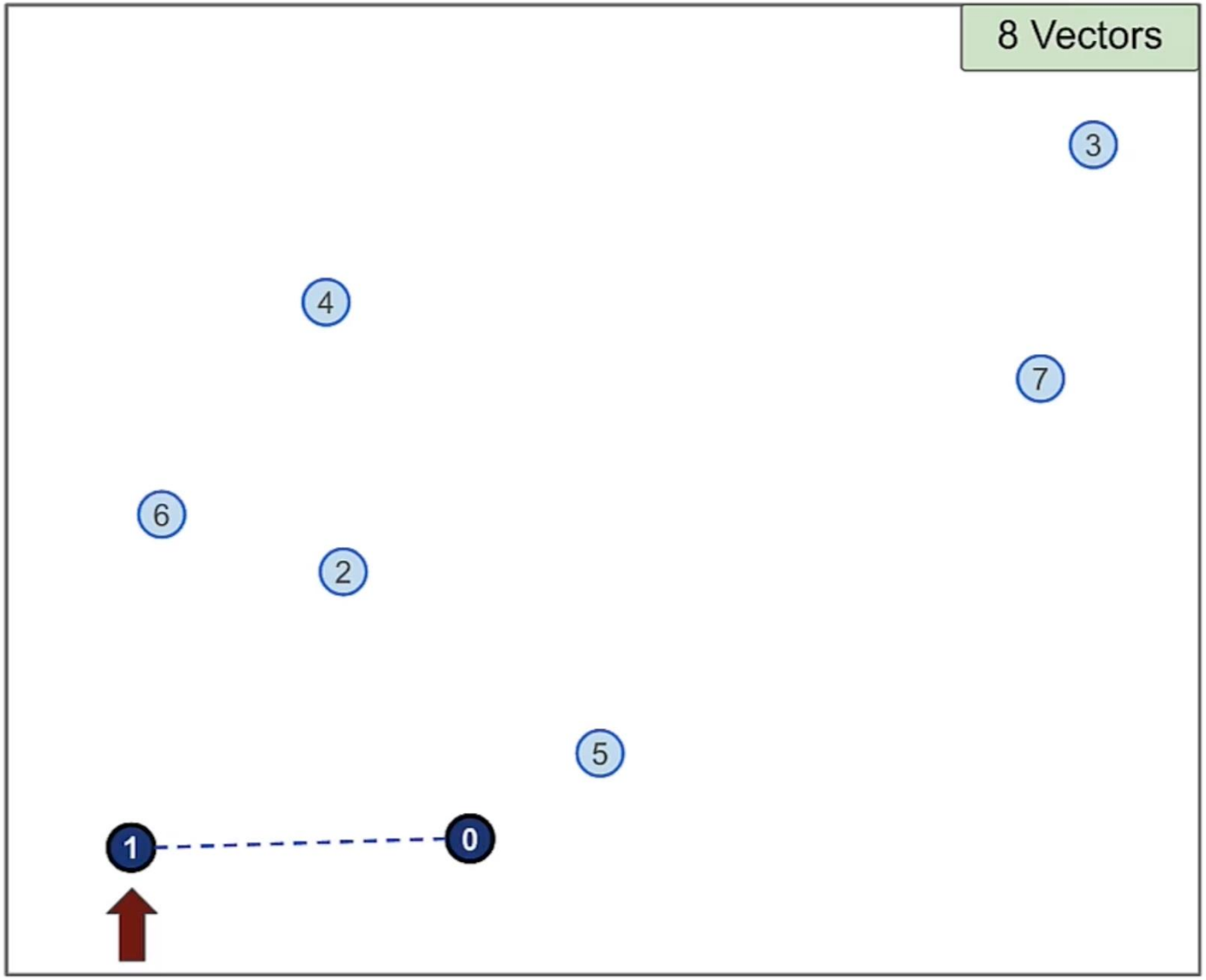


- Observed phenomenon in human social networks where everyone is closely connected.
- **Six Degrees of Separation:** Popularized concept that any two individuals are on average separated by six acquaintance links.
- **High Clustering:** Nodes form tightly-knit groups, with members of groups connecting to one another.
- **Short Path Lengths:** Despite high clustering, only a few steps are needed to connect any two nodes.

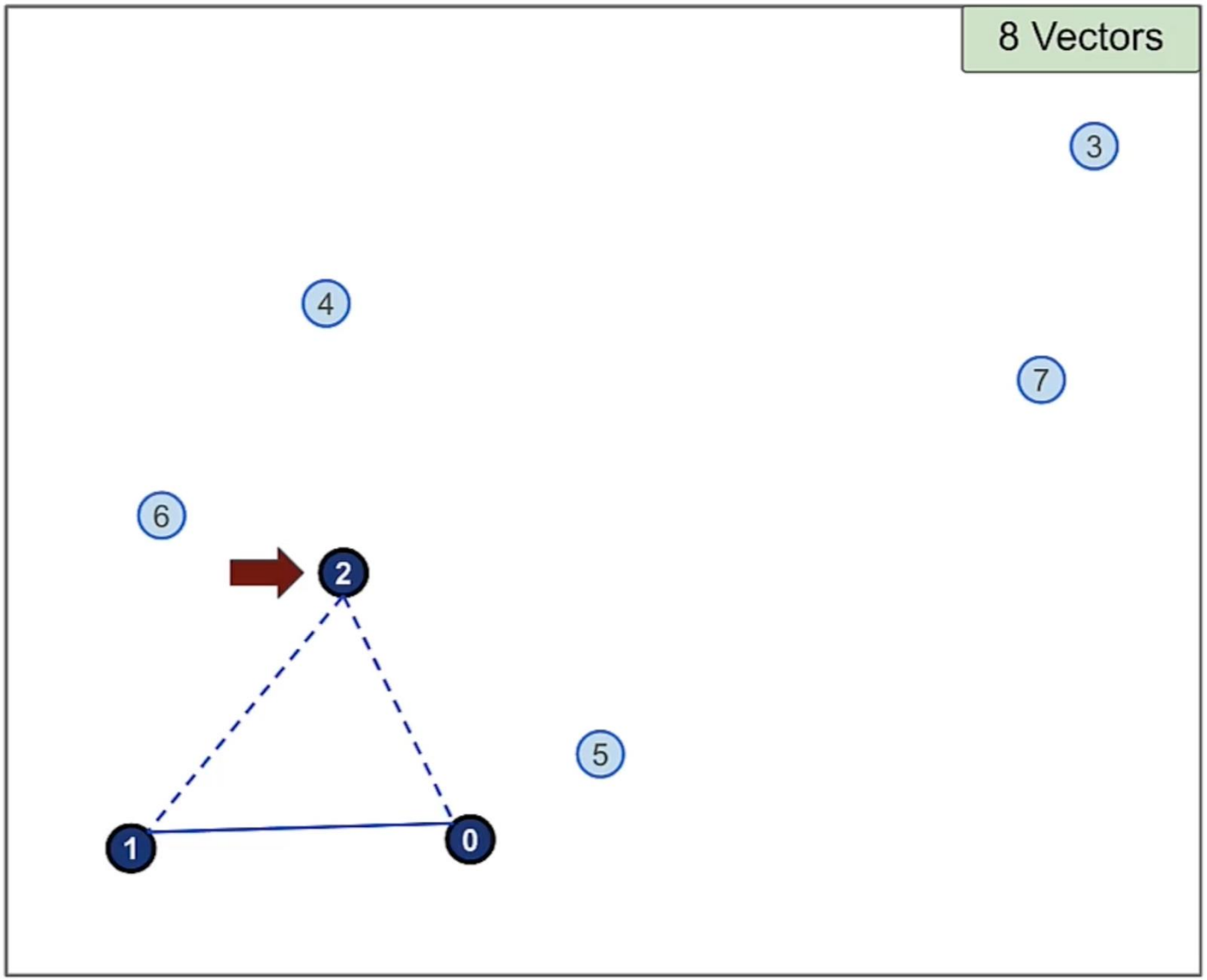
Navigate Small World (Constructing)



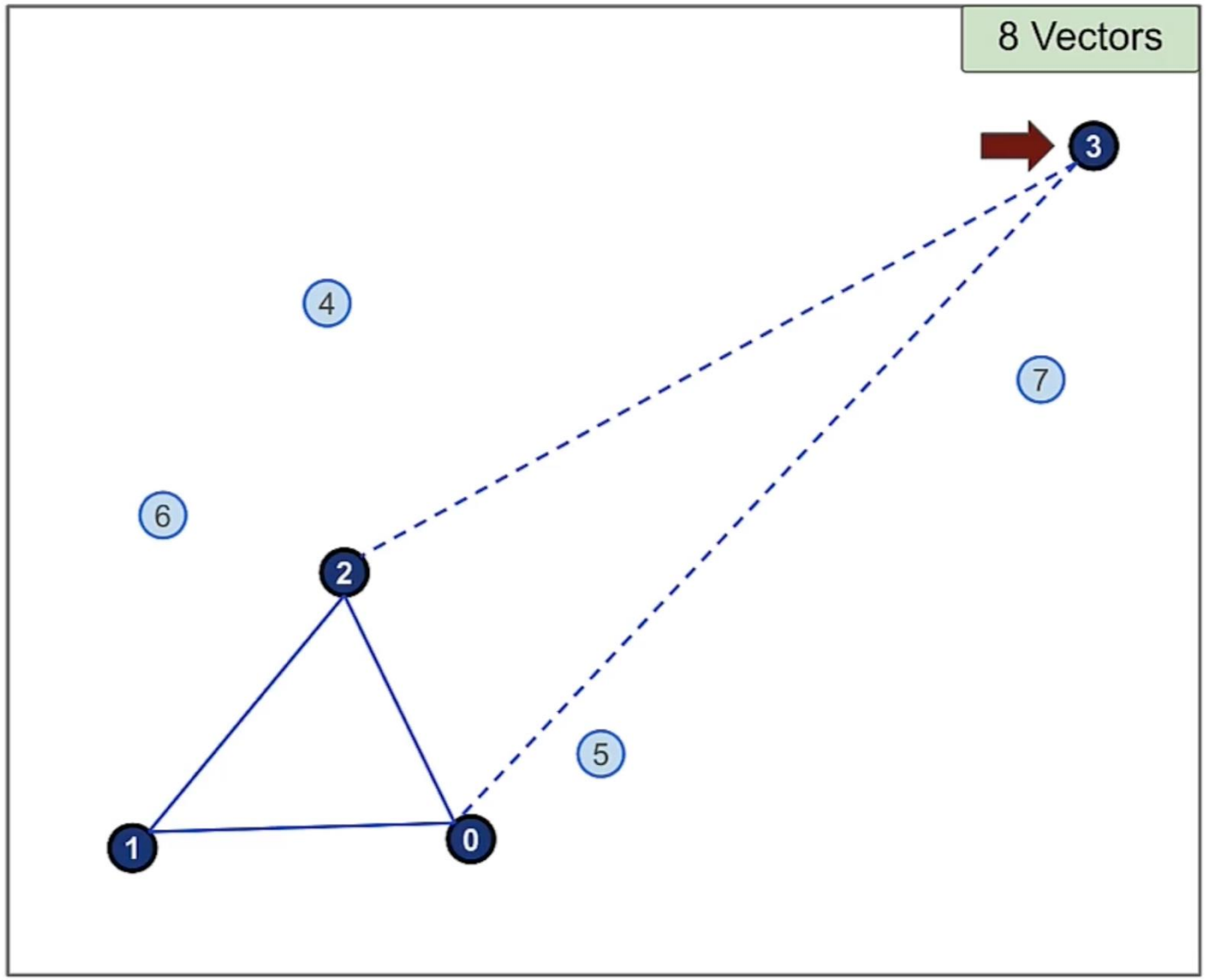
Navigate Small World (Constructing)



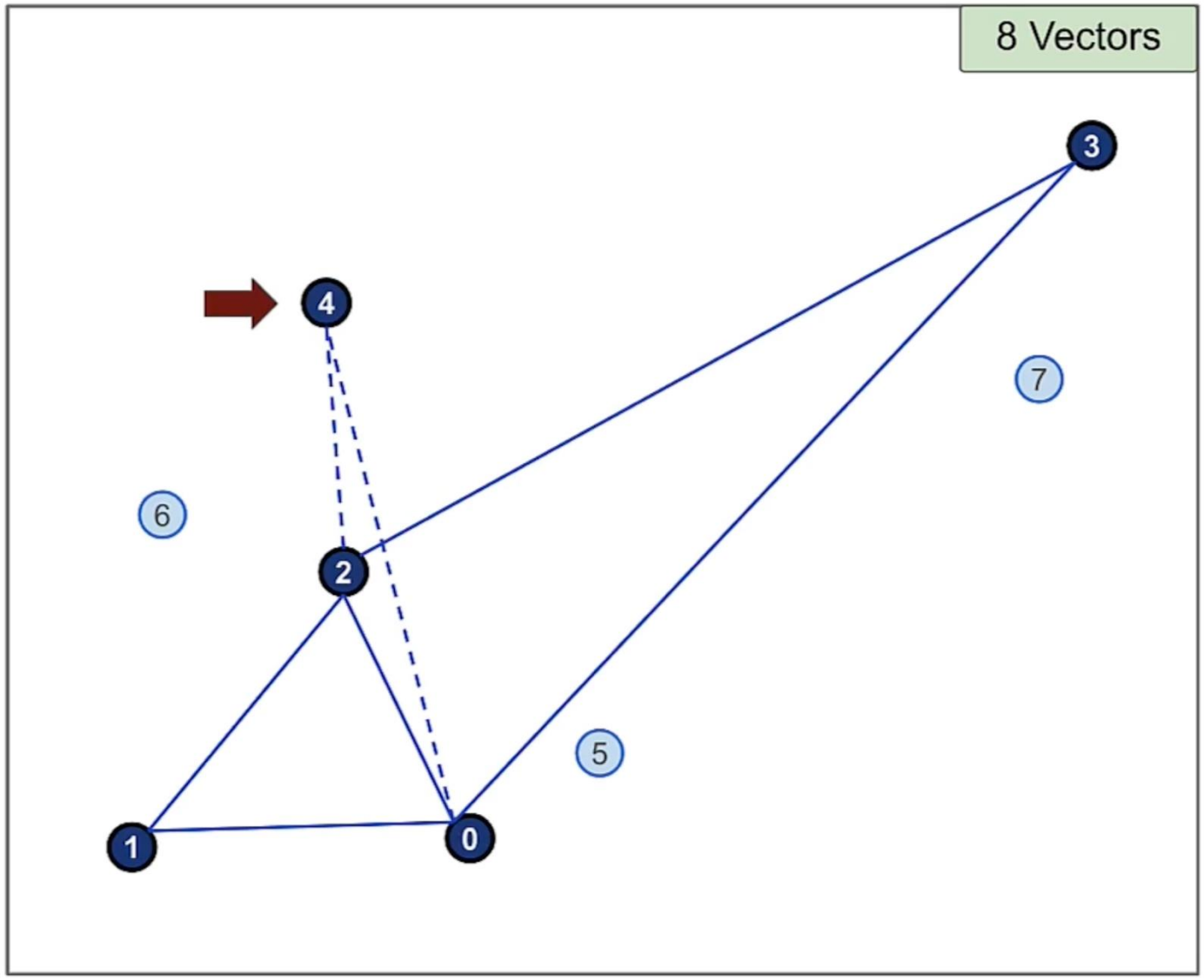
Navigate Small World (Constructing)



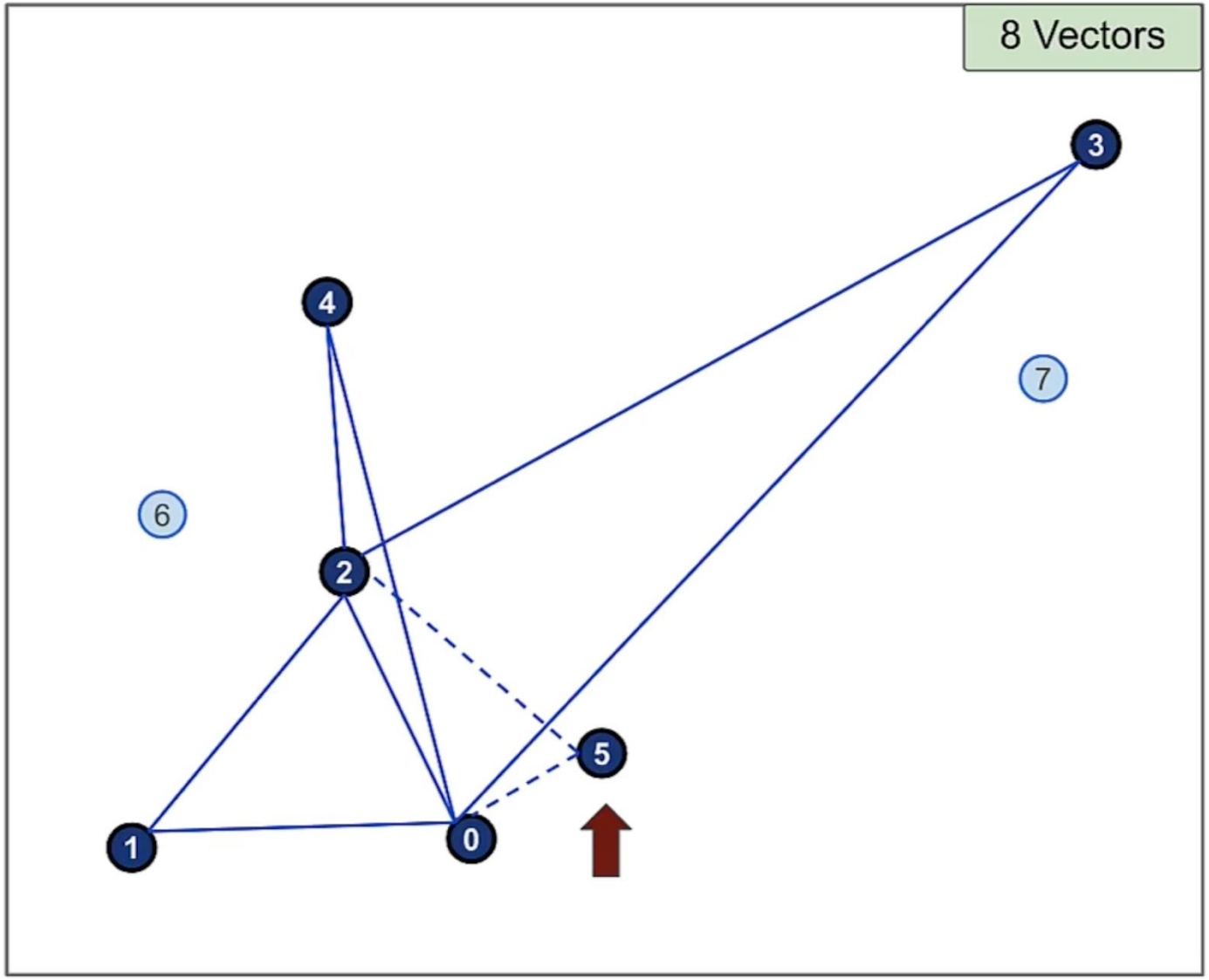
Navigate Small World (Constructing)



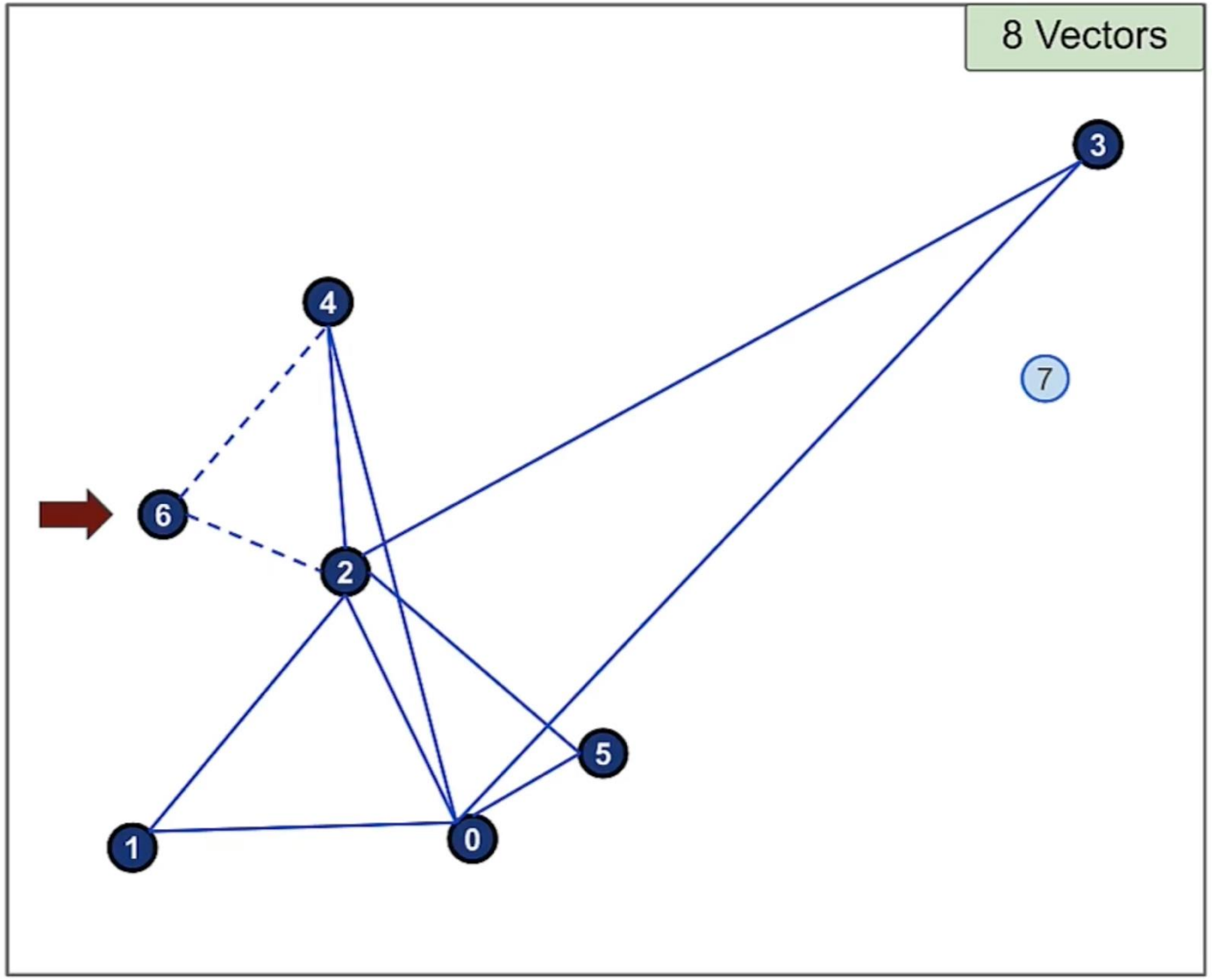
Navigate Small World (Constructing)



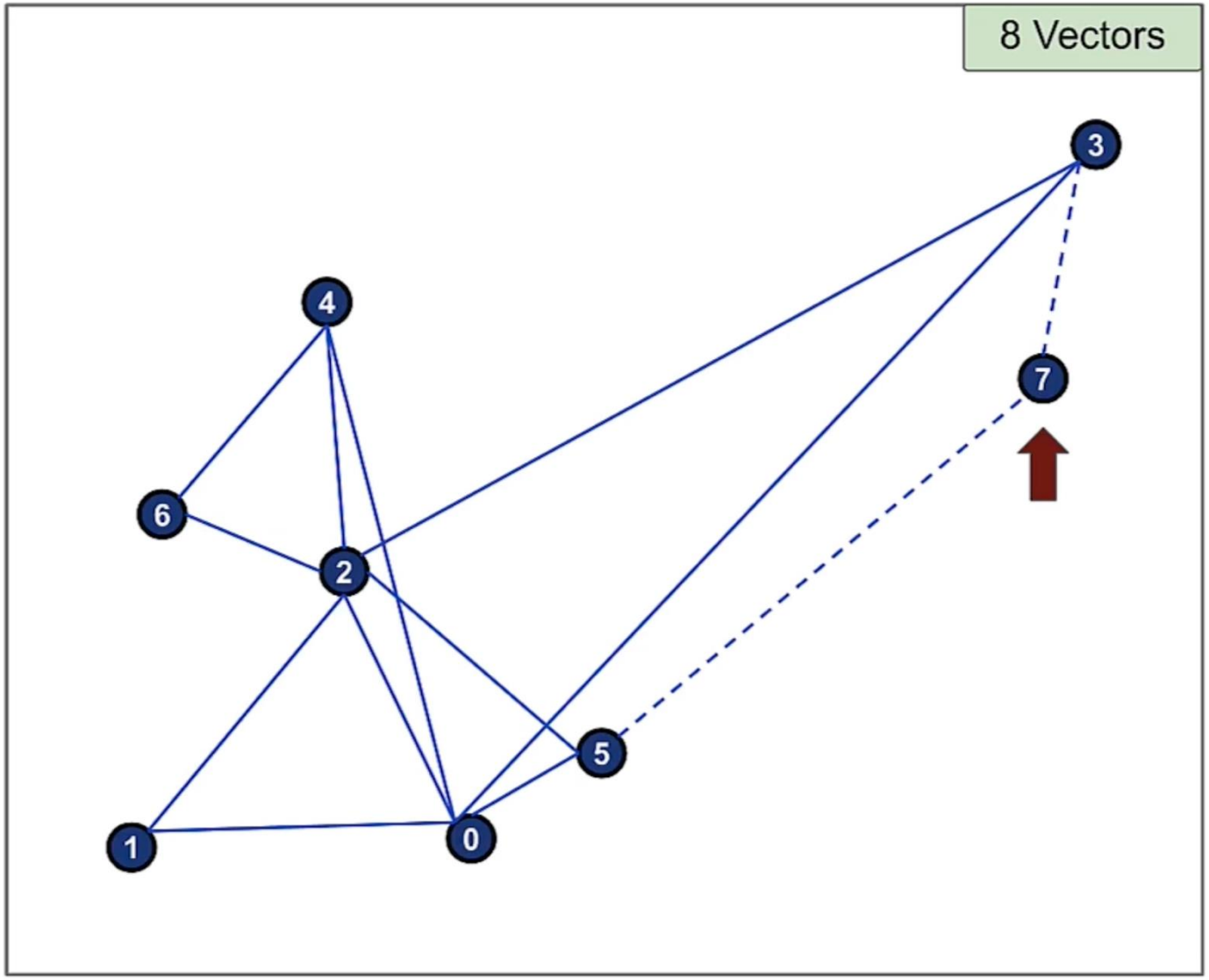
Navigate Small World (Constructing)



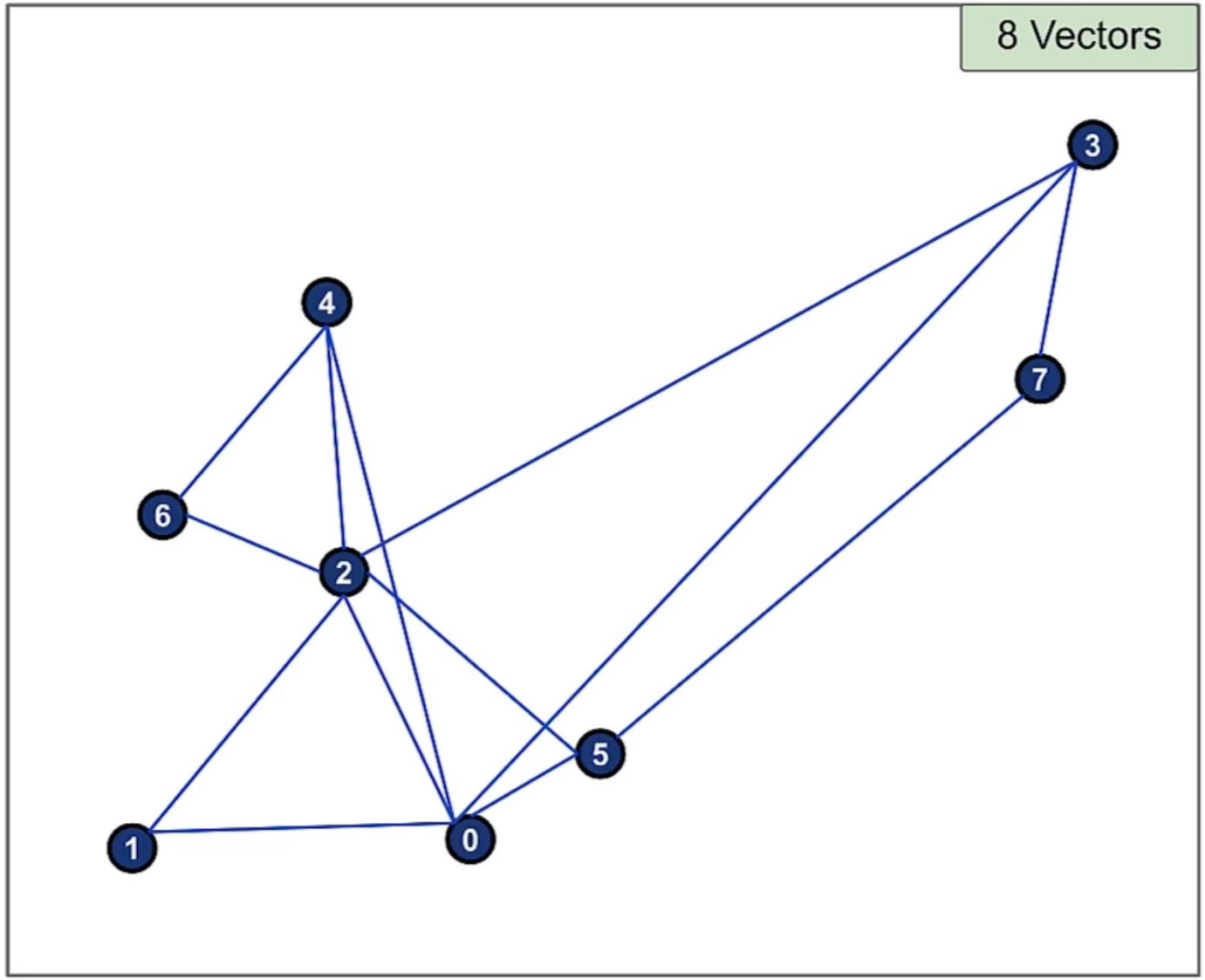
Navigate Small World (Constructing)



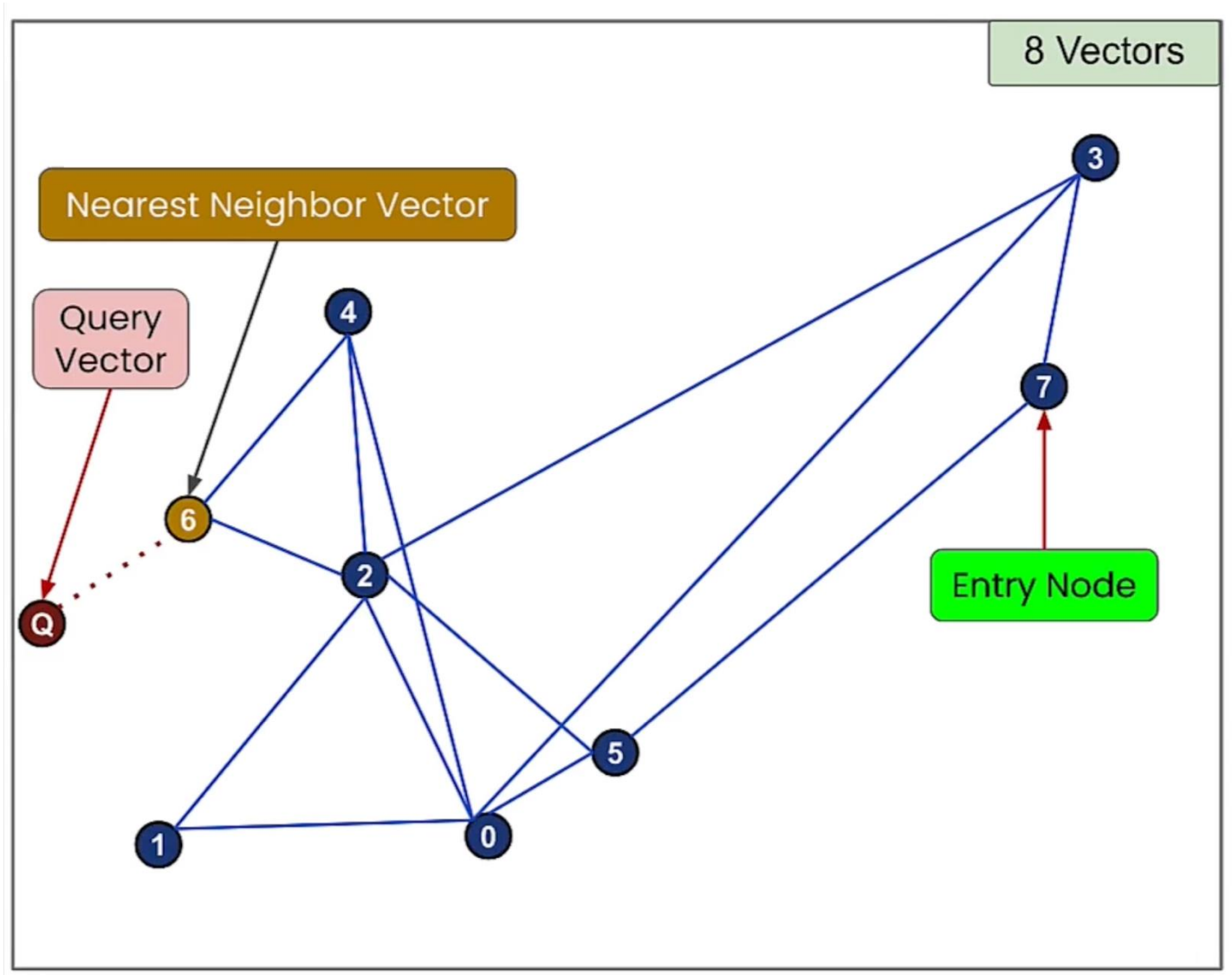
Navigate Small World (Constructing)



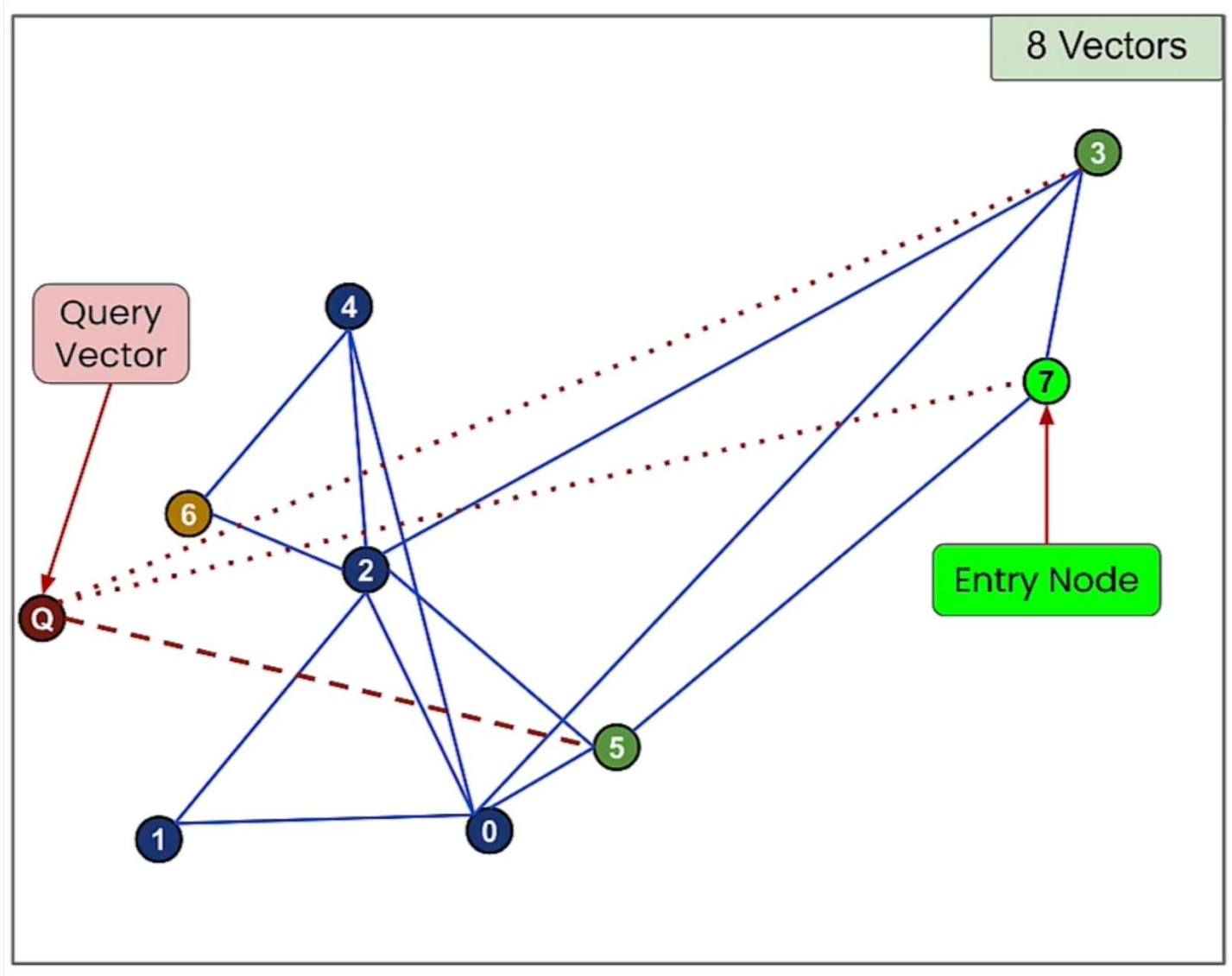
Navigate Small World (Constructing)



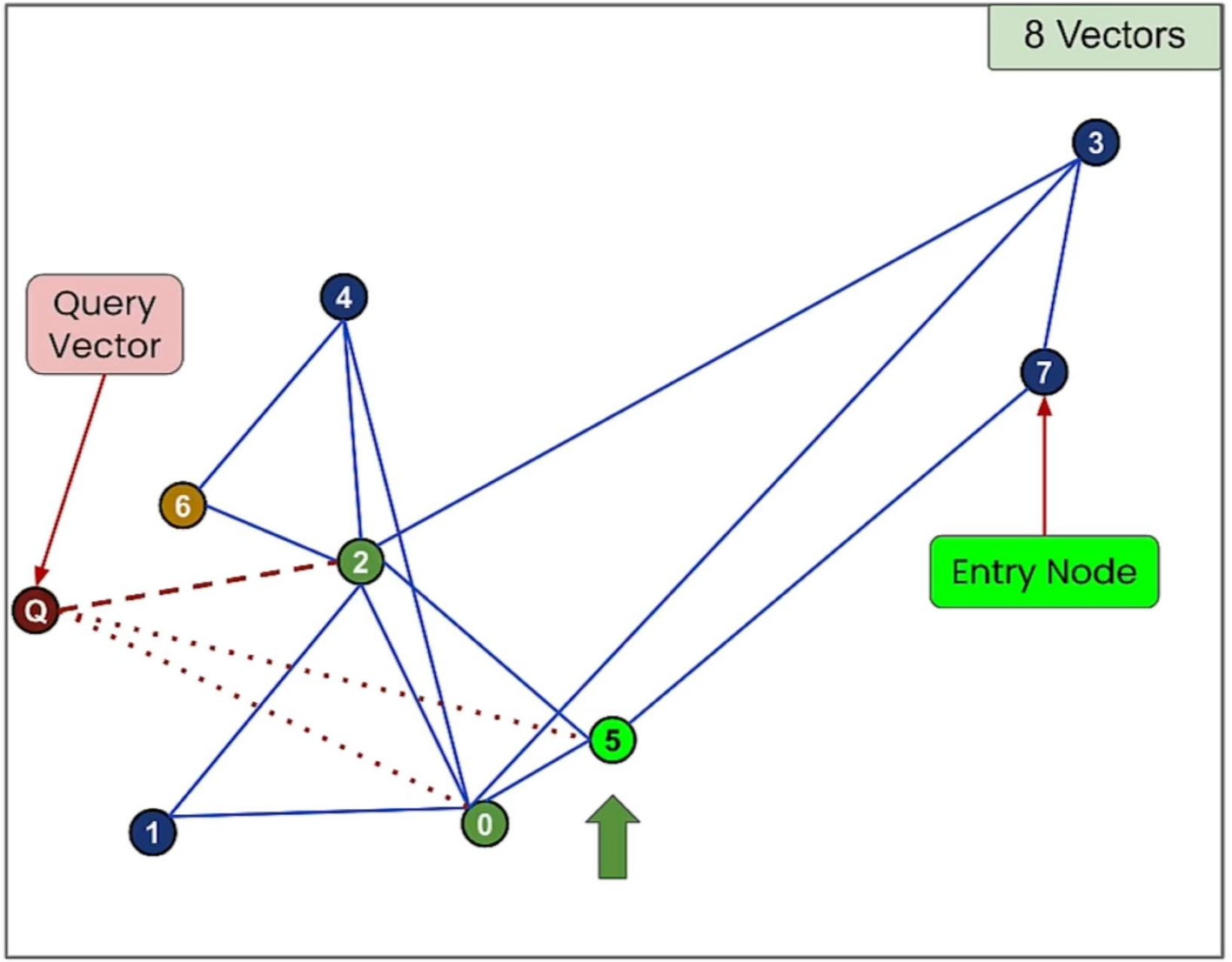
NSW - Search 1: Nearest Neighbor



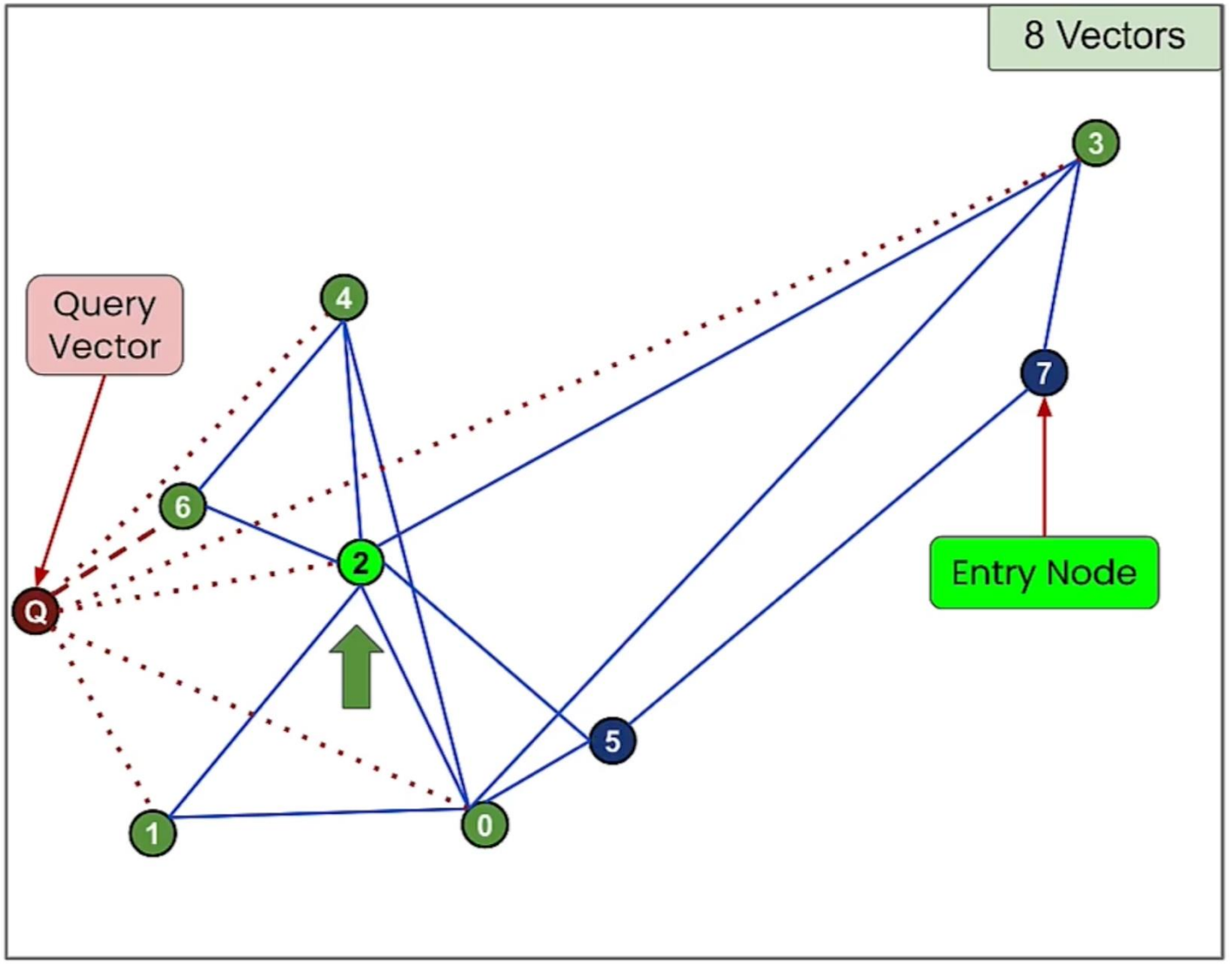
NSW - Search



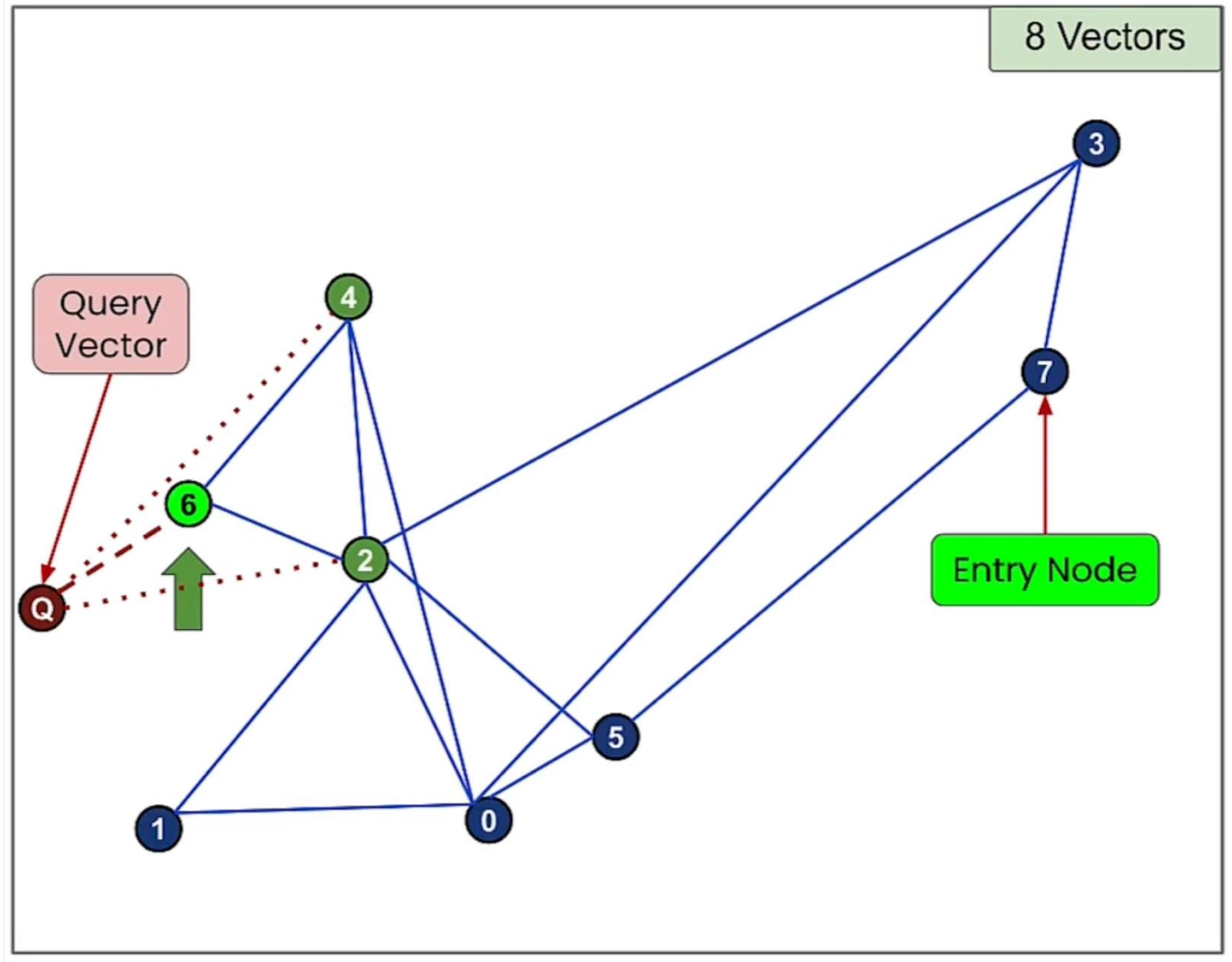
NSW - Search



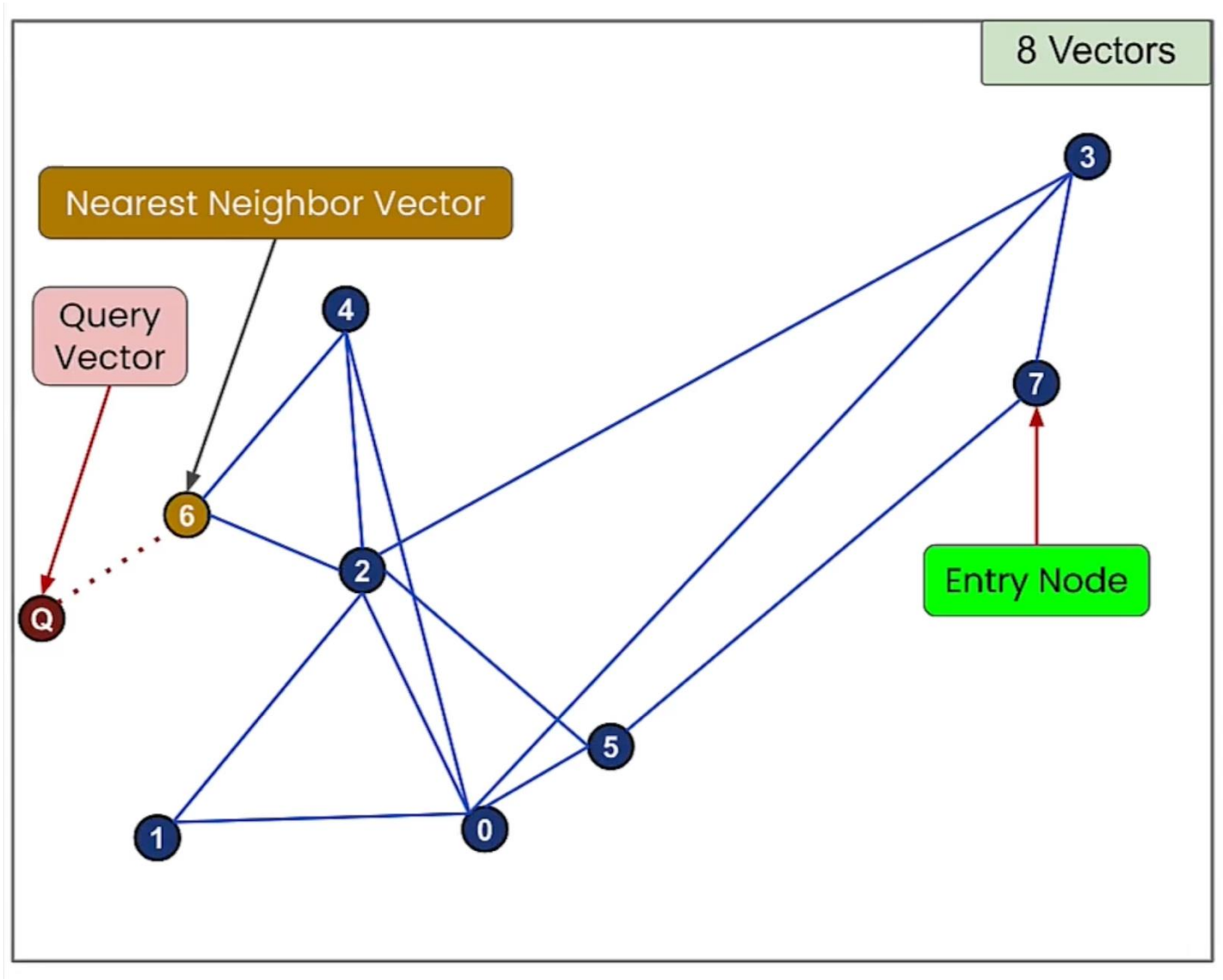
NSW - Search



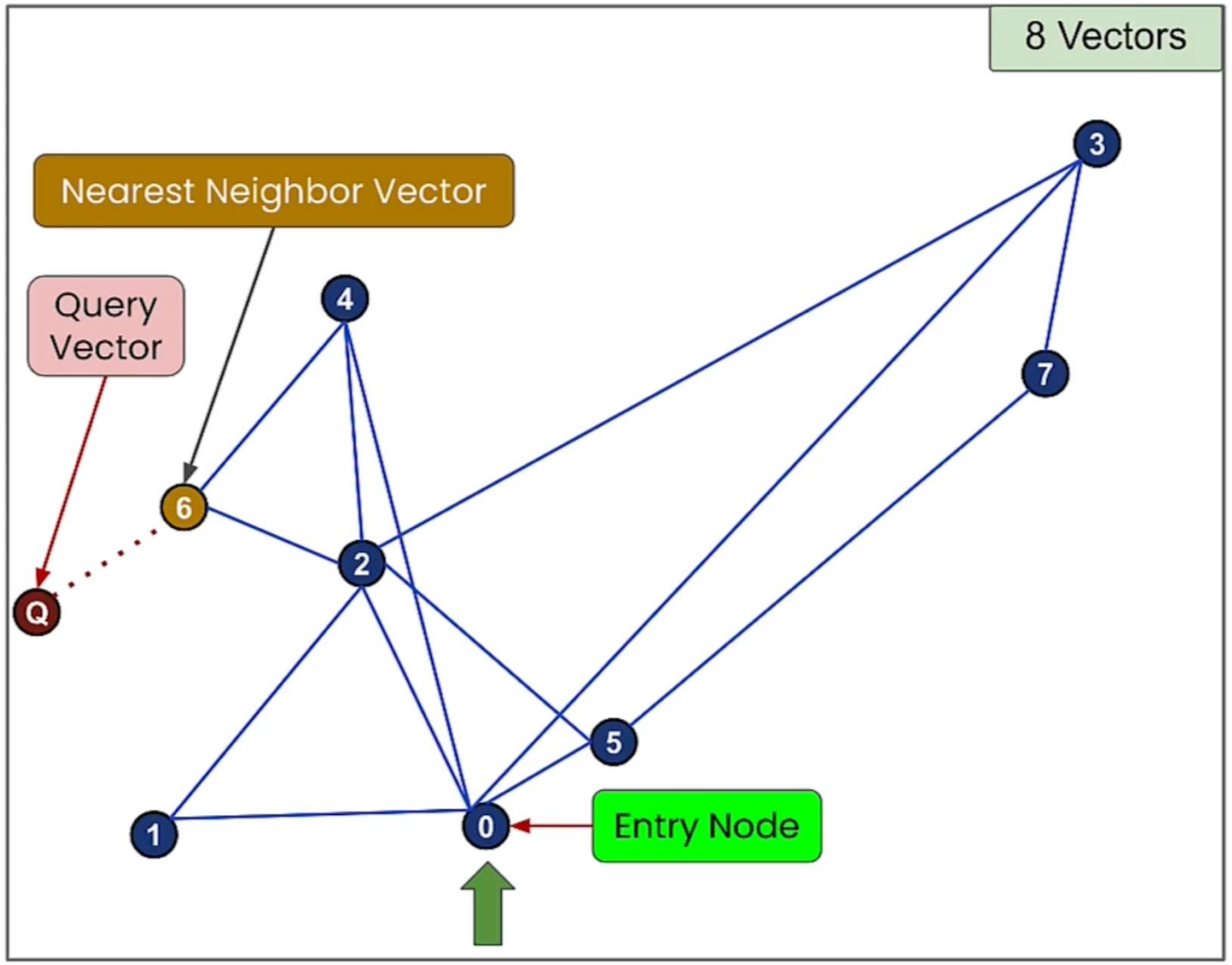
NSW - Search



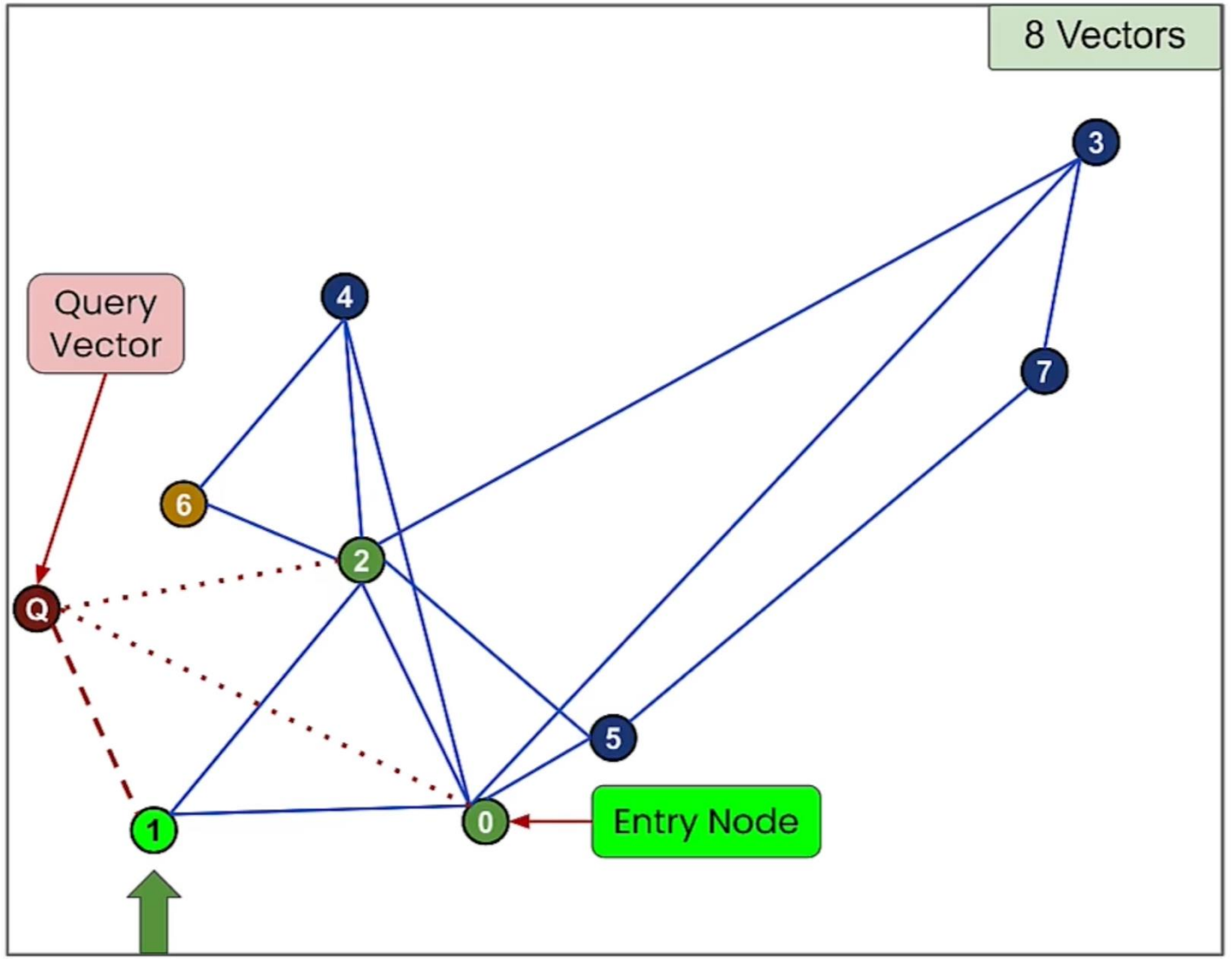
NSW - Search


8 Vectors
Nearest Neighbor Vector
Query Vector
Entry Node
Q
0
1
2
3
4
5
6
7

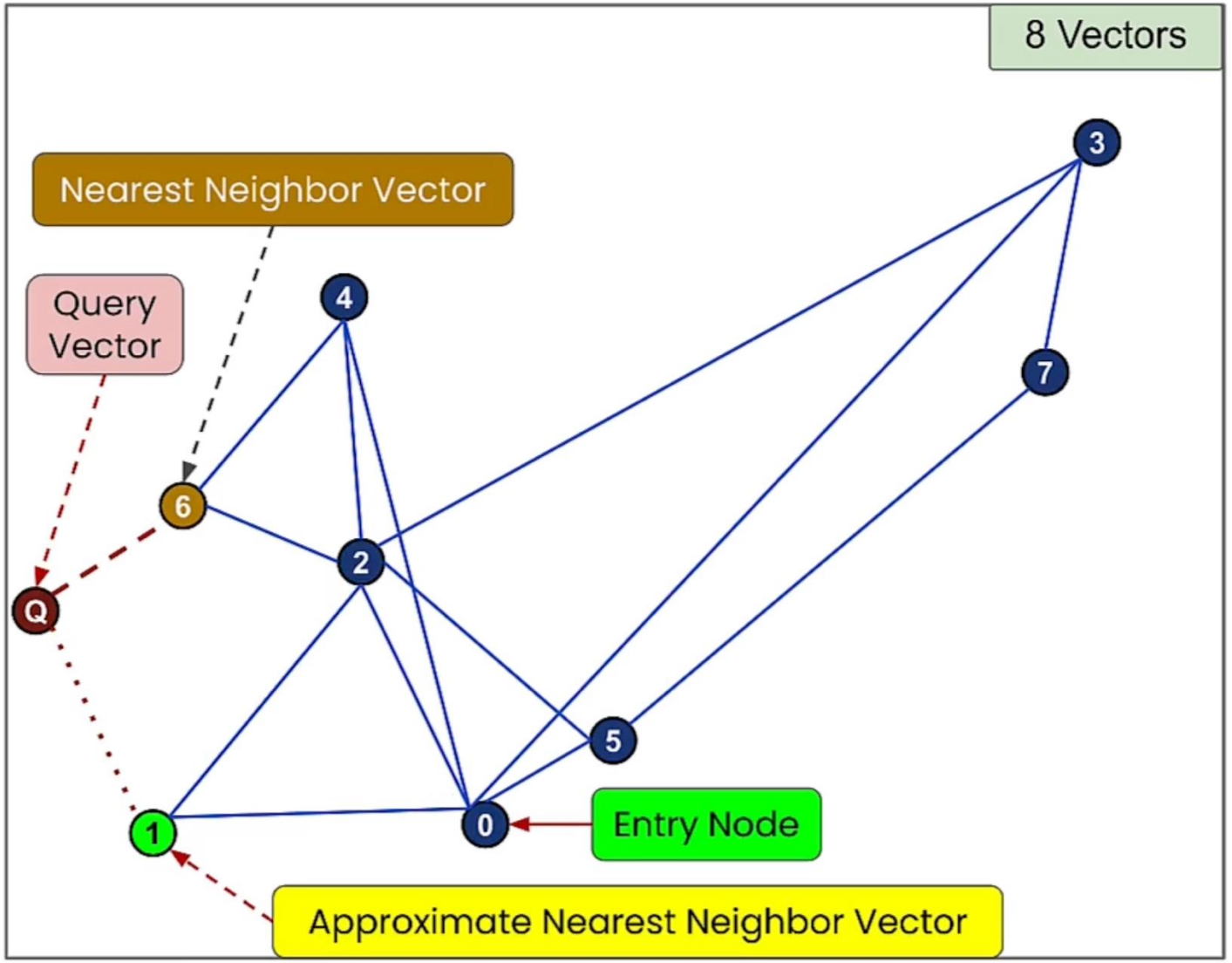
NSW – Search 2 Approximate Nearest Neighbor



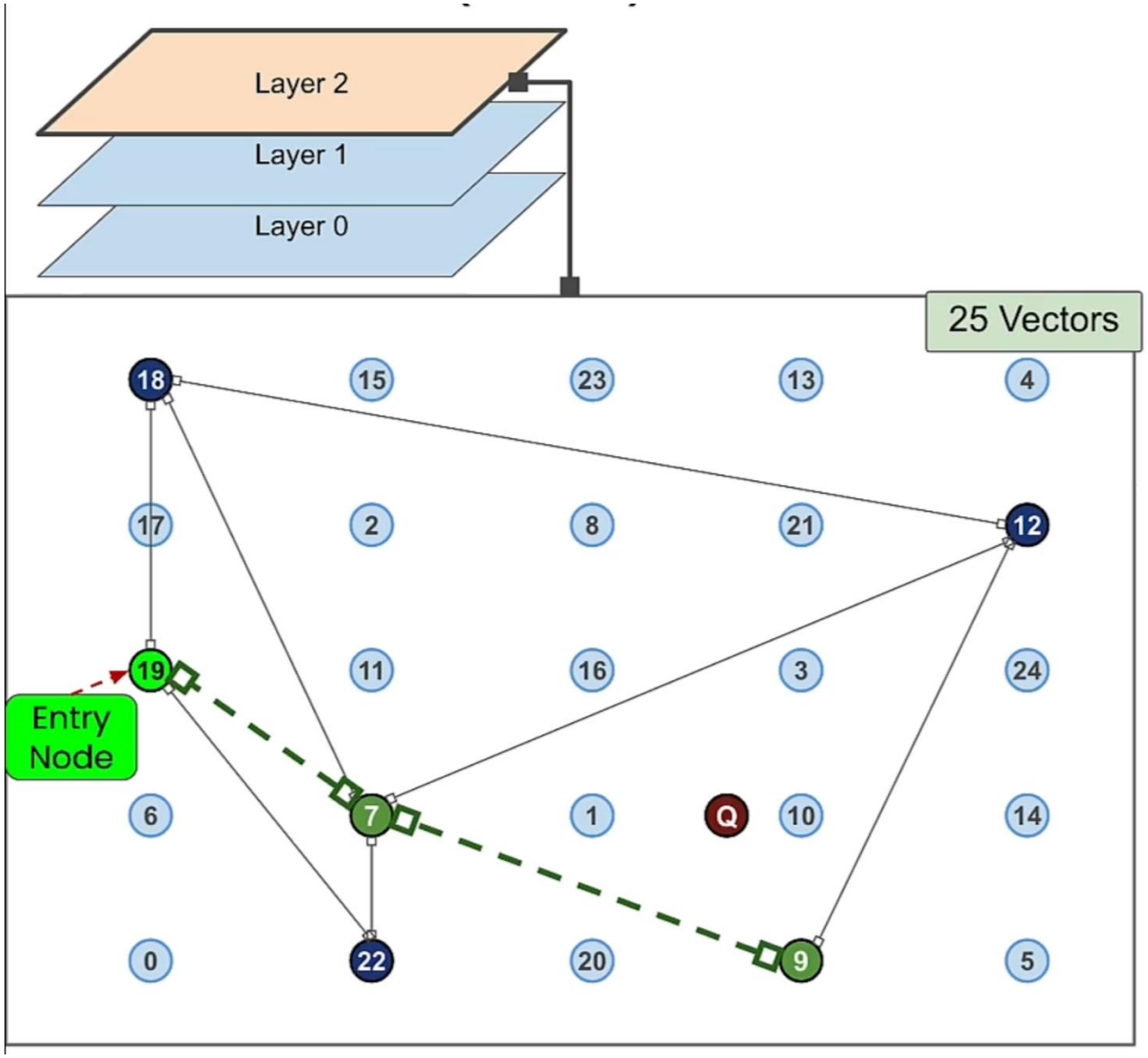
NSW – Search 2 Approximate Nearest Neighbor



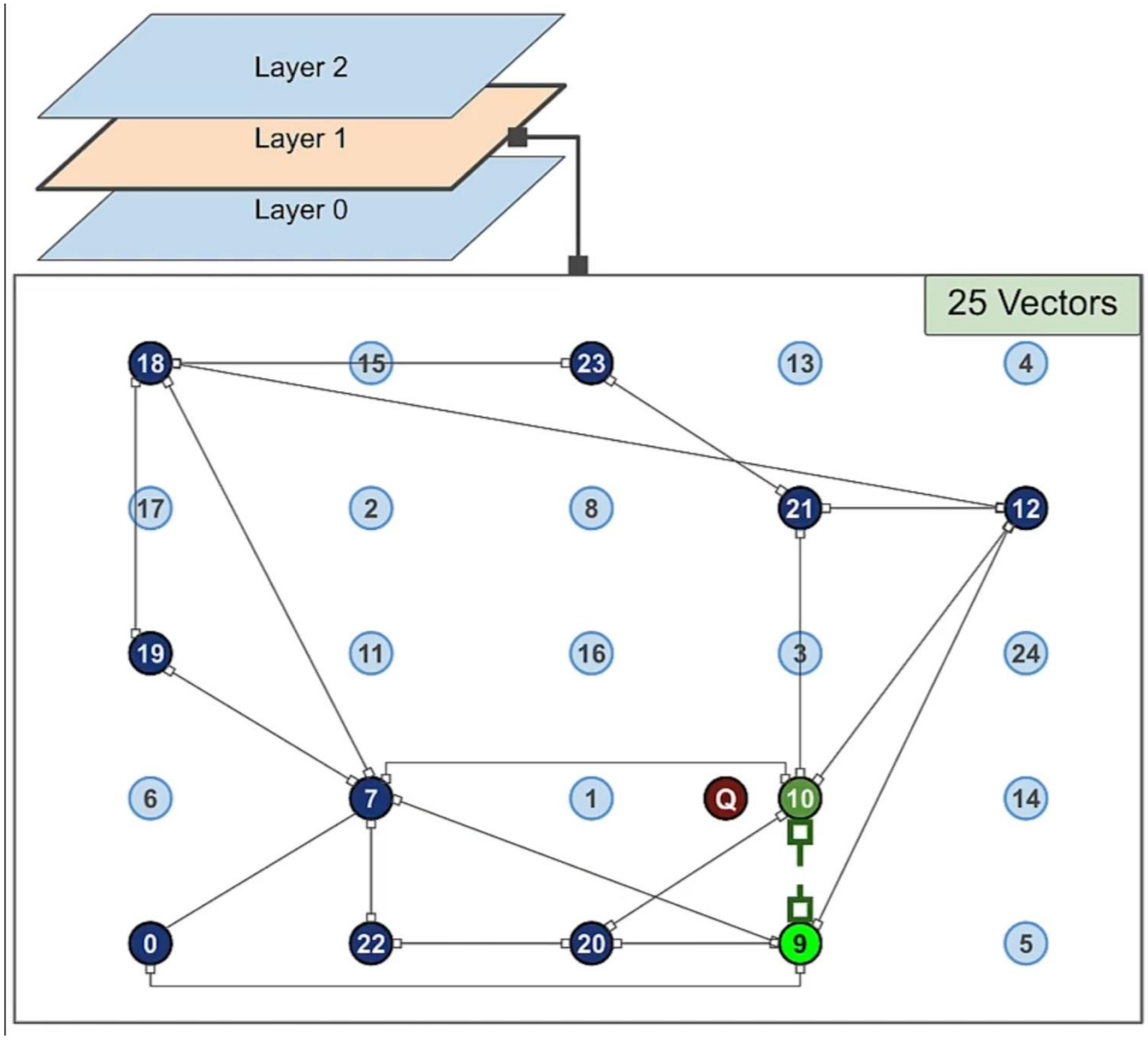
NSW – Search 2 Approximate Nearest Neighbor



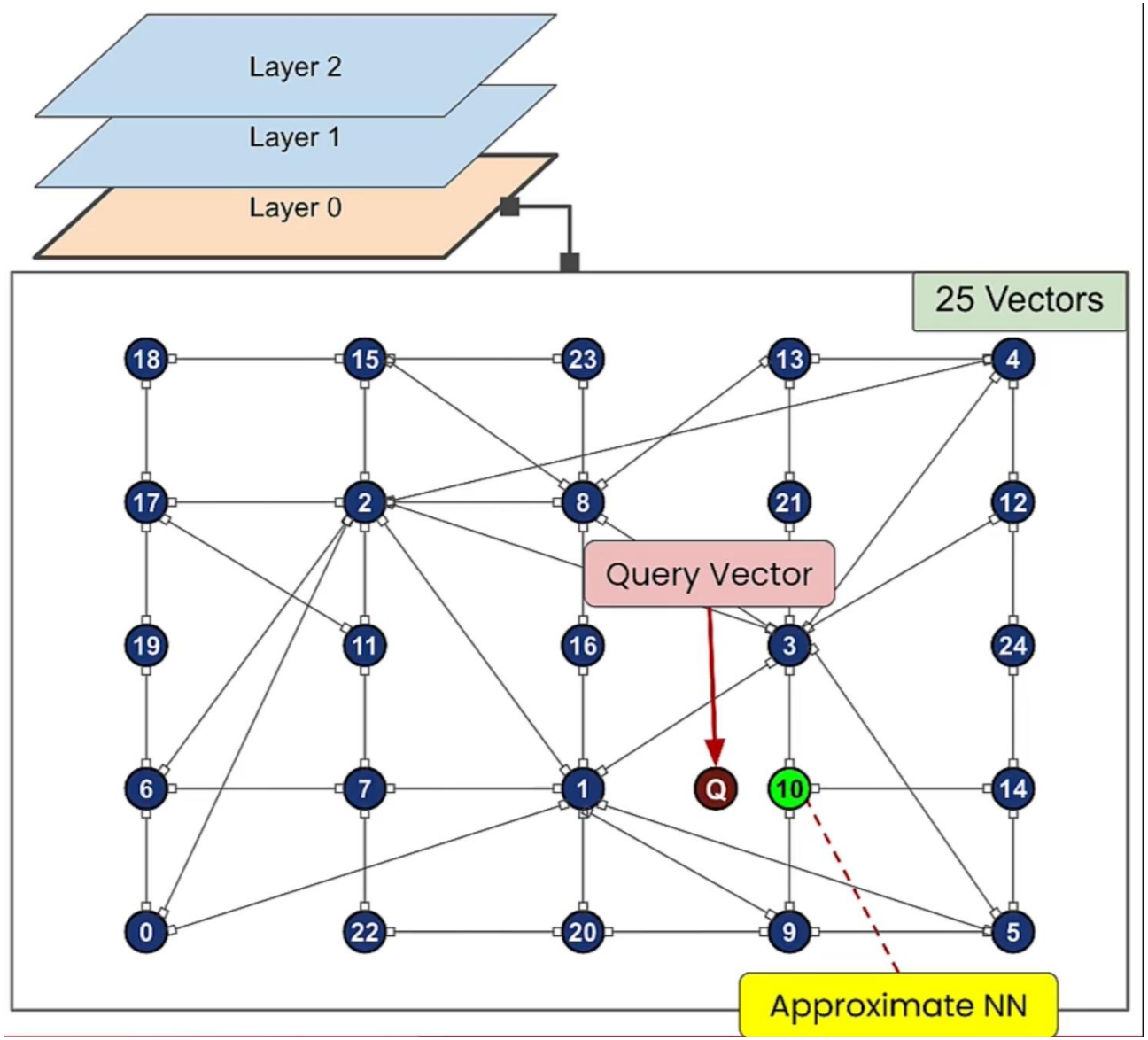
Hierarchical Navigable Small World (HNSW) - Search



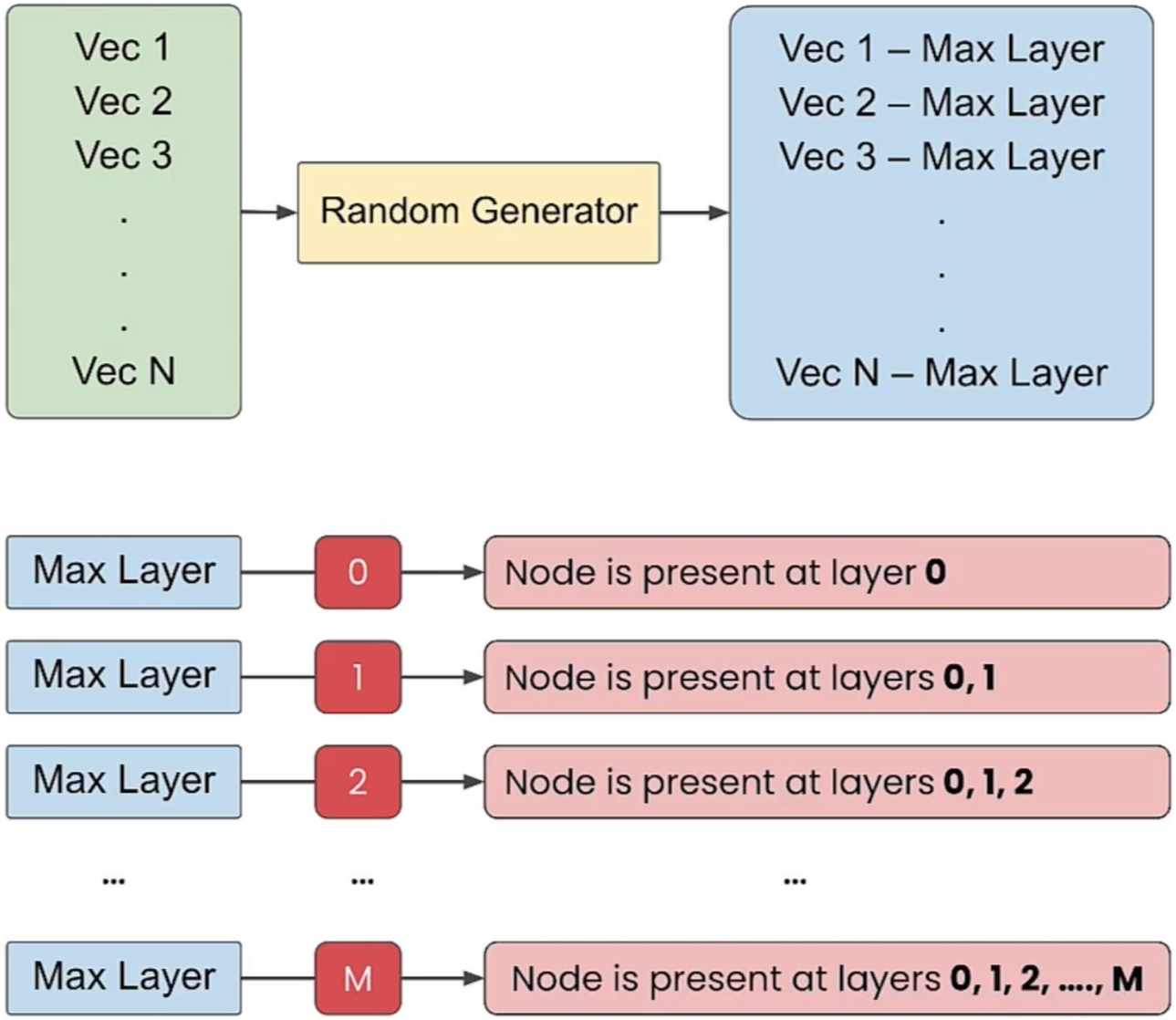
Hierarchical Navigable Small World (HNSW) - Search



Hierarchical Navigable Small World (HNSW) - Search

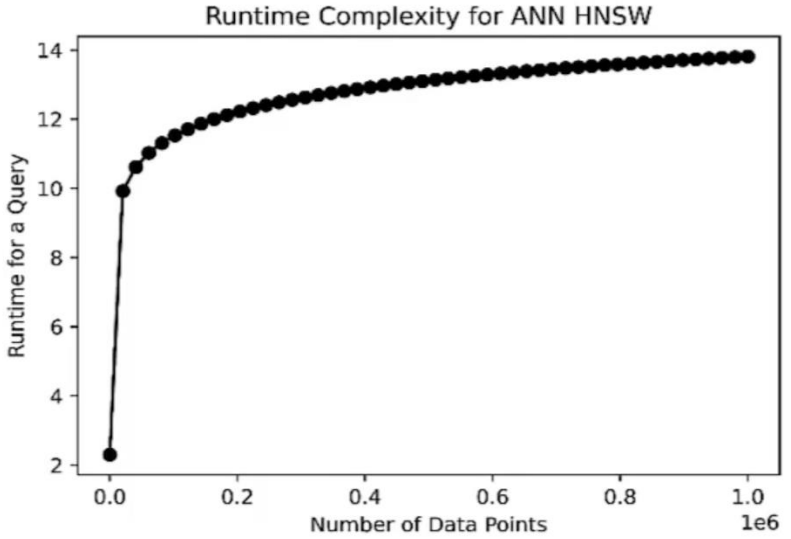


HNSW - Construction



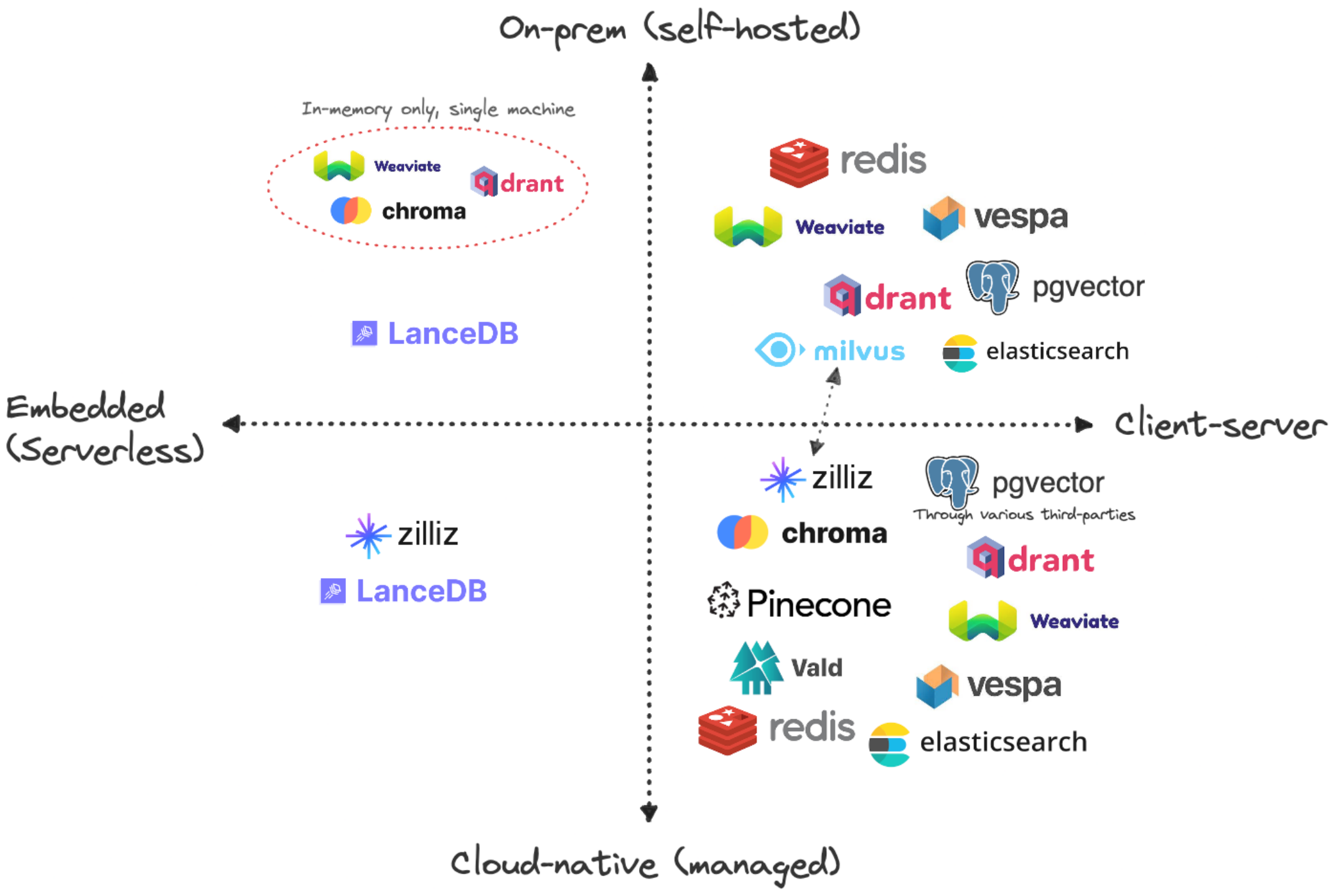
HNSW Runtime

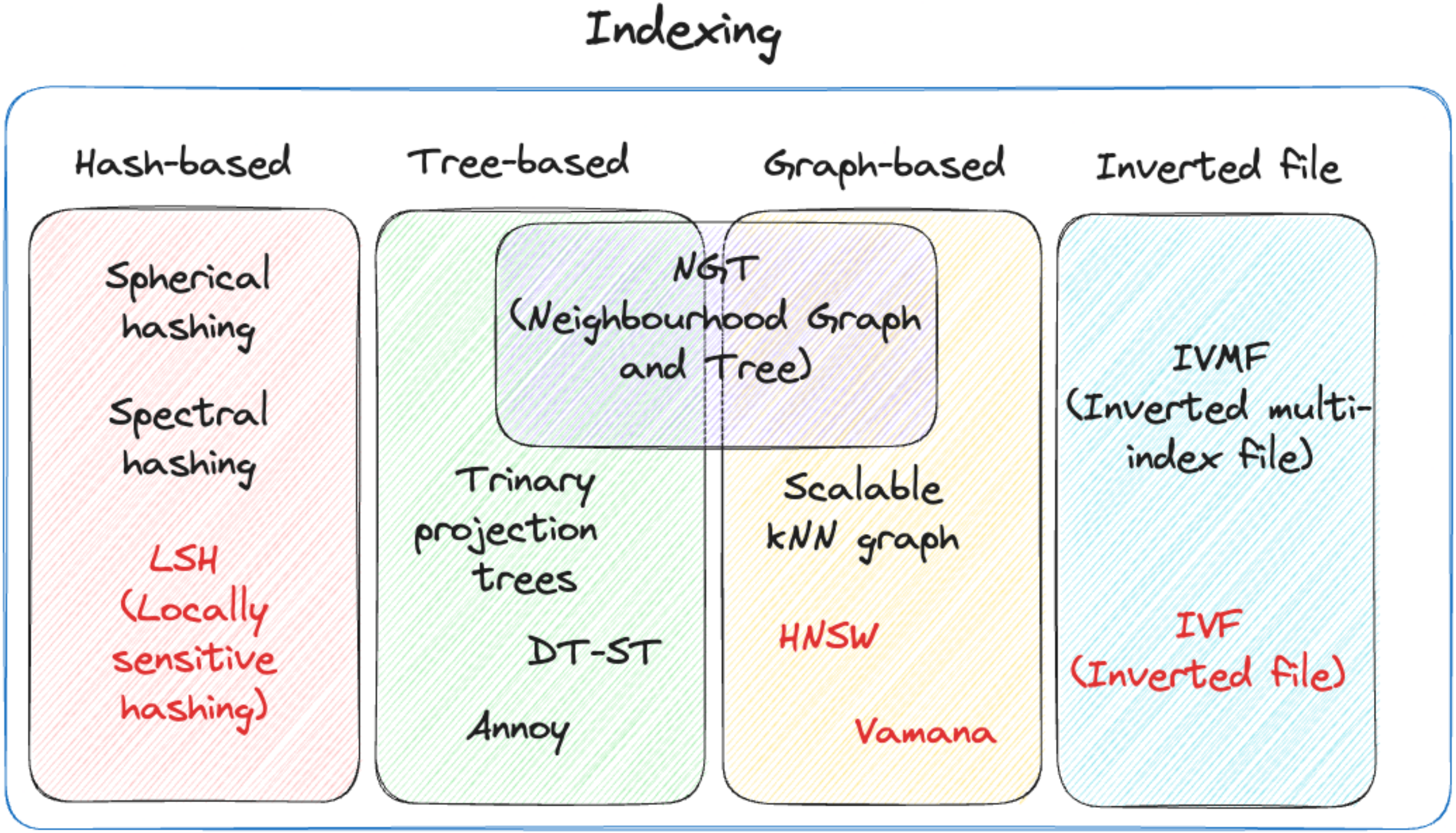
- **Low likelihood in Higher Levels**
  - The likelihood of a vector being found on the higher up levels of our HNSW graph goes down exponentially
- **Query time increases Logarithmically**
  - This means that as the number of datapoints increases the number of comparisons to perform vector search only goes up logarithmically.
- ***O(log(N))* runtime complexity**
  - As a results we have a *O(log(N))* runtime complexity for the HNSW algorithm



VectorDB

- A vector database is a specialized database designed to store, manage, and query high-dimensional vector data efficiently
  - Optimized for storing and retrieving vector embeddings, which are crucial for AI and machine learning applications
- Why Vector Databases are Useful for Dense Retrieval and RAG
  - Efficient storage and retrieval of high-dimensional data
  - Optimized for similarity search operations
  - Scalability to handle large datasets
  - Low-latency queries for real-time applications
  - Seamless integration with AI and machine learning workflows


On-prem (self-hosted)
In-memory only, single machine
Weaviate
qdrant
chroma
LanceDB
redis
Weaviate
vespa
qdrant
pgvector
milvus
elasticsearch
Embedded (Serverless)
Client-server
zilliz
LanceDB
zilliz
pgvector
Through various third-parties
chroma
qdrant
Pinecone
Weaviate
Vald
vespa
redis
elasticsearch
Cloud-native (managed)


Indexing
Hash-based
Tree-based
Graph-based
Inverted file
Spherical hashing
Spectral hashing
LSH (Locally sensitive hashing)
NGT (Neighbourhood Graph and Tree)
Trinary projection trees
DT-ST
Annoy
Scalable kNN graph
HNSW
Vamana
IVMF (Inverted multi-index file)
IVF (Inverted file)

Sparse vs Dense Search

- **Dense Search (Semantic Search)**
  - Uses vector embedding representation of data to perform search.
    (as described in the previous lessons)
  - This types of search allows one to capture and return semantically similar objects.

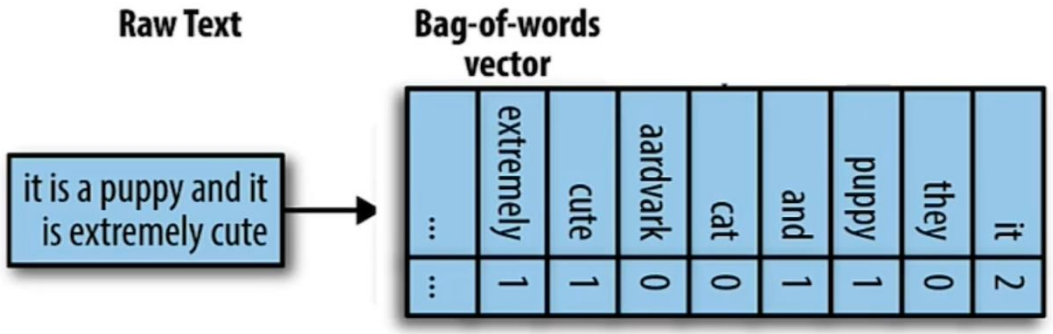
Dense Search Challenge

- **Product with a Serial Number**
  - Searching for seemingly random data (like serial numbers) will also yield poor results.

- **String or Word Matching**
  - Here we would be better off doing exact string or word matching to see where which product has the same serial number as the input serial number.
- This is known as **keyword search or sparse search!**

Sparse vs Dense Search

- **Bag of Words**
  - The easiest way to do keyword matching is using Bag of Words - to count how many times a word occurs in the query and the data vector and then return objects with the highest matching word frequencies.
- **This is known as Sparse Search**
  - because the text is embedded into vectors by counting how many times every unique word in your vocabulary occurs in the query and stored sentences.
- **Mostly Zeroes (Sparse Embedding)**
  - Since the likelihood of any given sentence containing every word in your vocabulary is quite low the embeddings is mostly zeroes and thus is known as a sparse embedding.



BM25

Best Matching 25 (BM25): In practice when performing keyword search we use a modification of simple word frequencies called *best matching 25*

$$\text{score}(D,Q) = \sum_{i=1}^{n} \text{IDF}(q_i) \cdot \frac{f(q_i, D) \cdot (k_1 + 1)}{f(q_i, D) + k_1 \cdot \left(1 - b + b \cdot \frac{|D|}{\text{avgdl}}\right)}$$

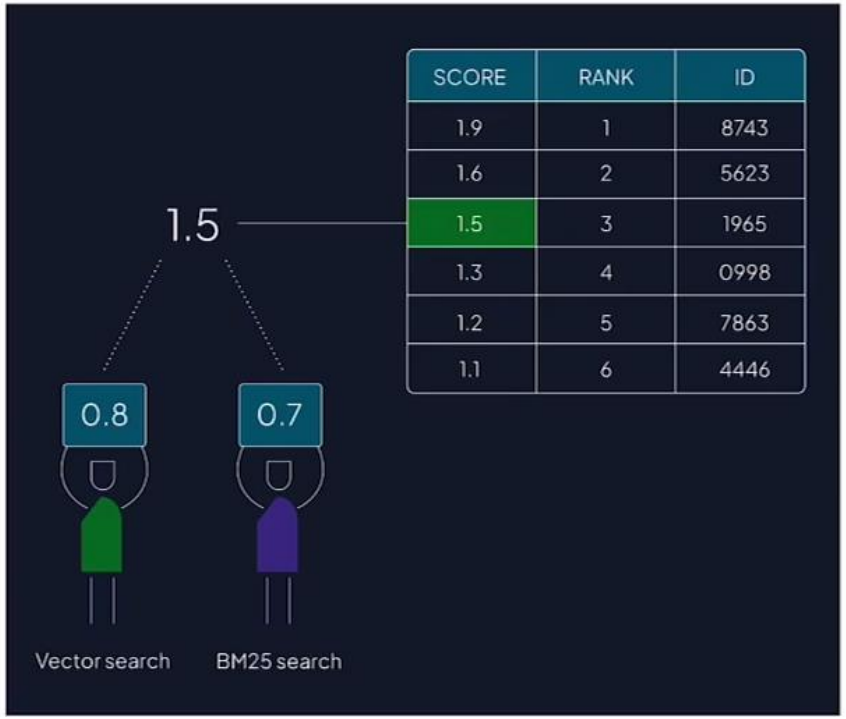
Learn more about BM25: https://en.wikipedia.org/wiki/Okapi_BM25

"If you can look into the seeds of time, And say which grain will grow and which will not."

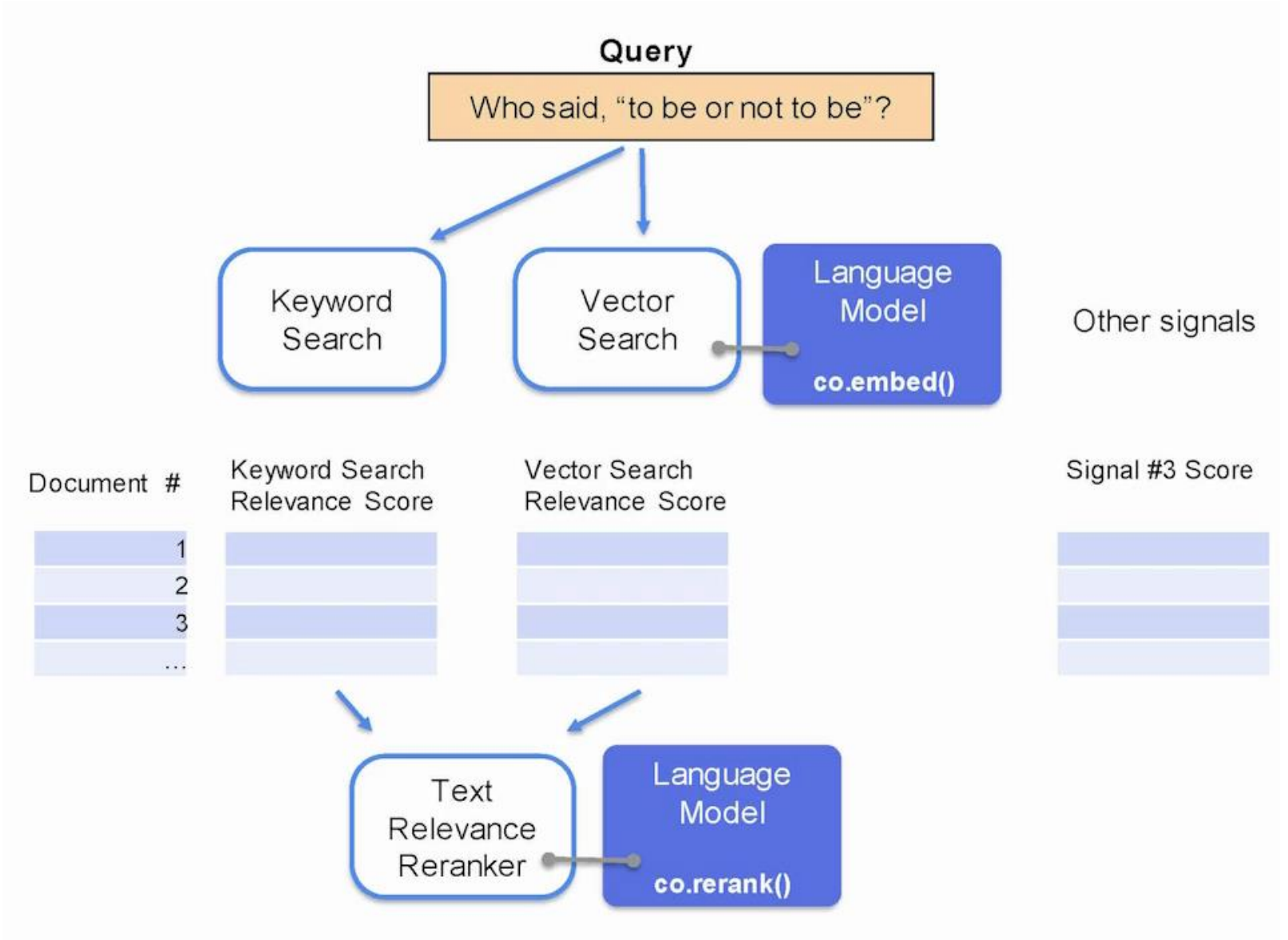
[0.,0.,0.,0.33052881,0.,.,0.,0.20961694,0.,0.,.,0.,0.,0.,
0.,.,0.,0.,0.20961694,0.20961694,.,0.20961694,0.209
61694,0.,0.20961694,.20961694,0.20961694,0.,0.,0.,.
,0.,0.,0.,0.20961694,0.,0.20961694,0.10938682,.,0.,0.
20961694,0.41923388,0.41923388,0.,0.,0.20961694]

Hybrid Search

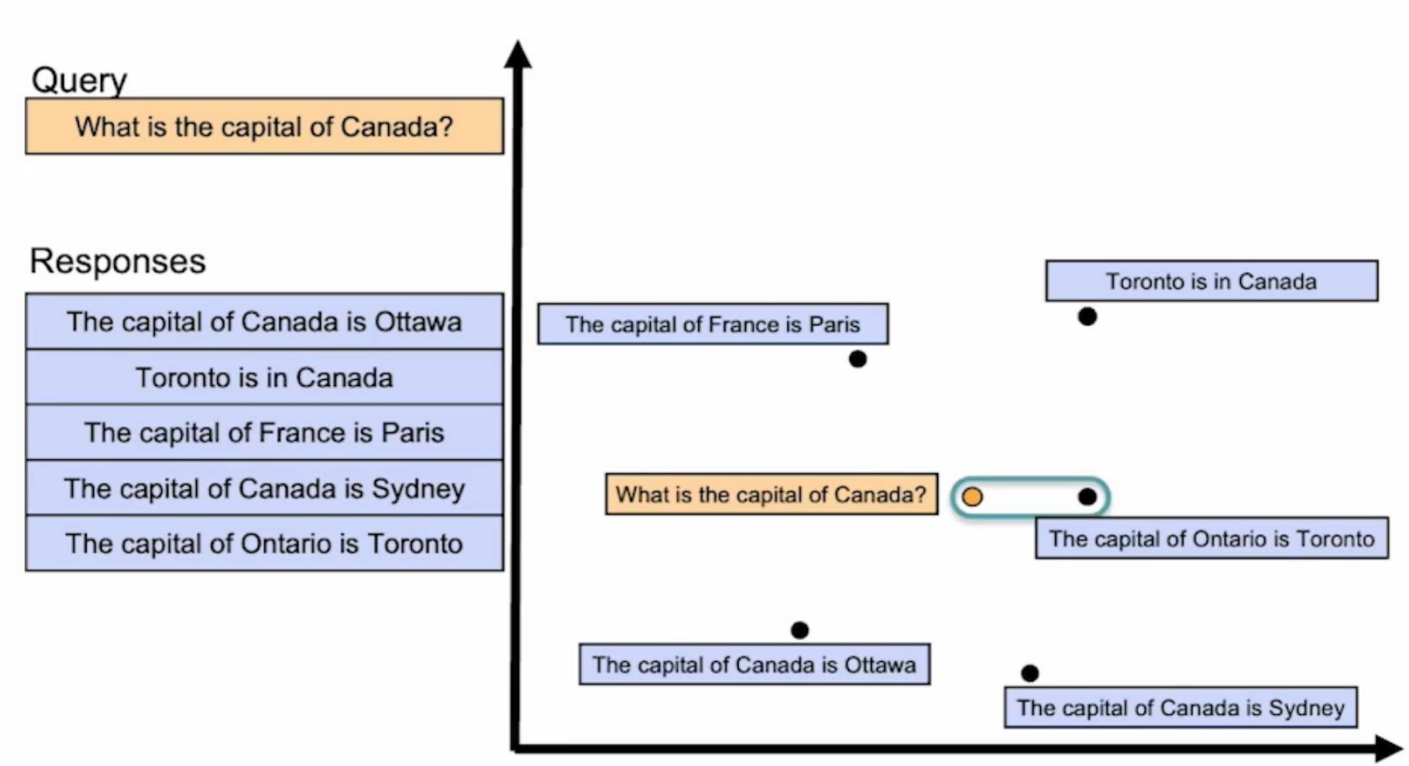
- **What is Hybrid Search?**
  - Hybrid search is the process of performing both vector/dense search and keyword/sparse search and then combining the results
- **Combination based on a Scoring System**
  - This combination can be done based on a scoring system that measures how well each objects matches the query using both dense and sparse searches.



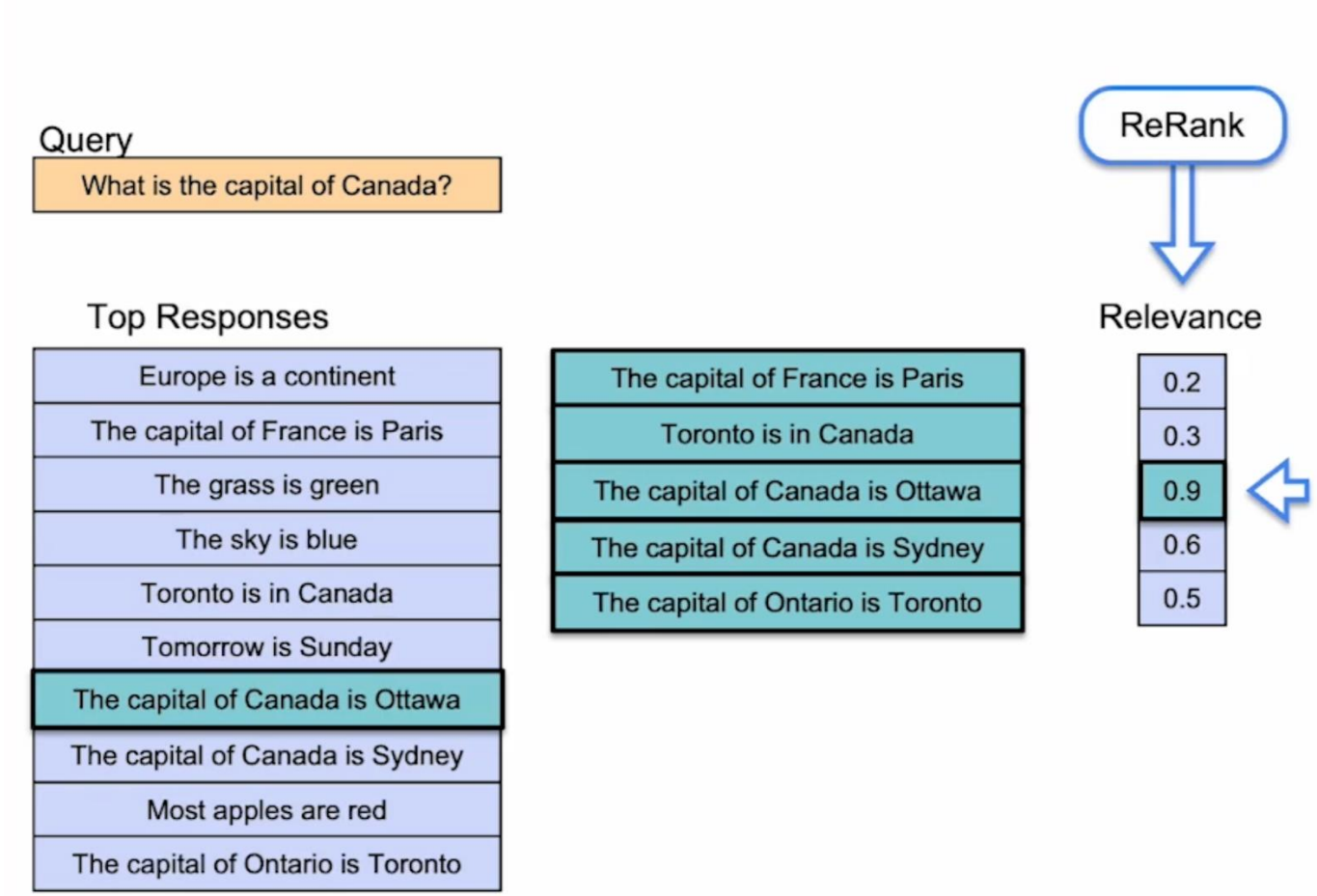
Hybrid Search



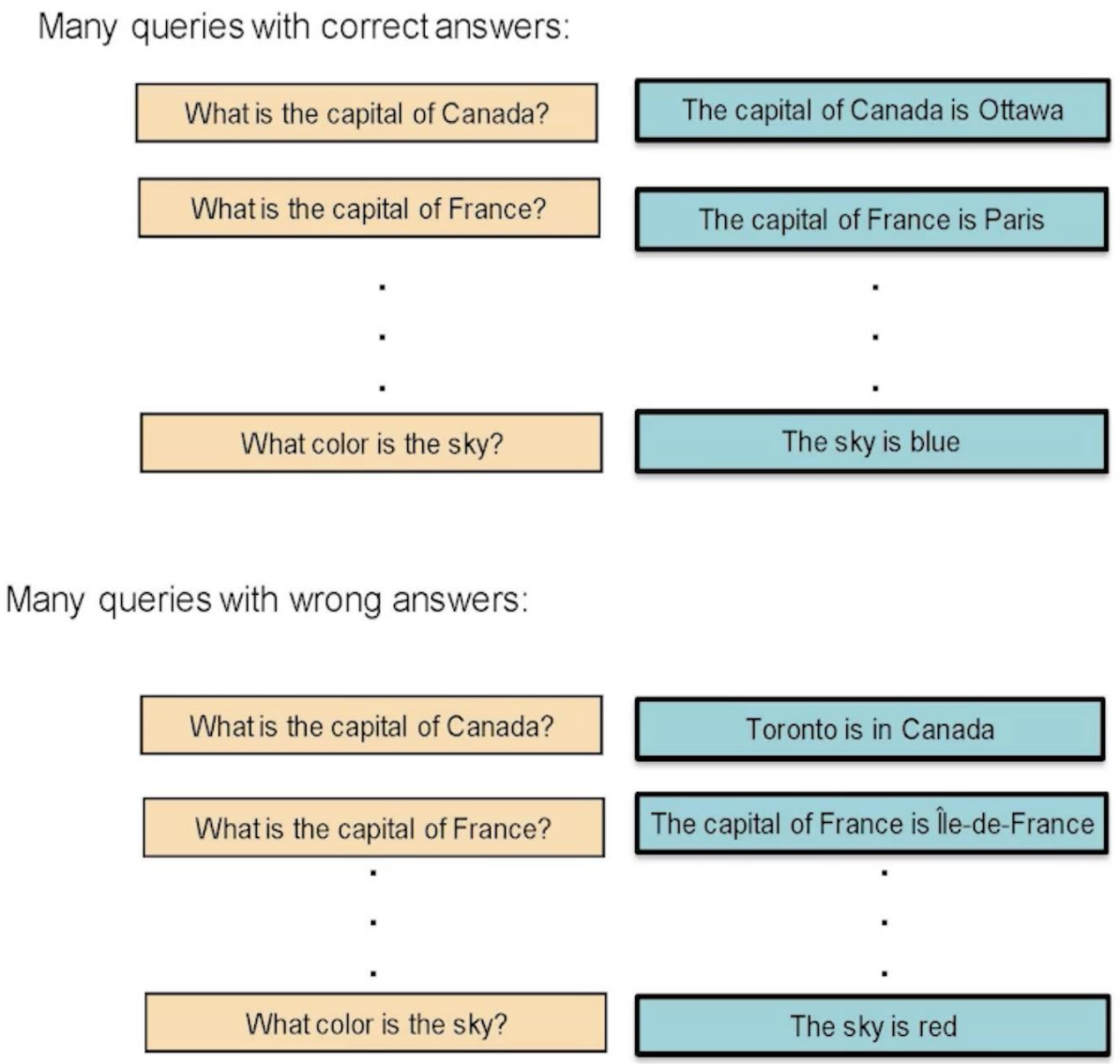
Dense Retrieval is not Perfect



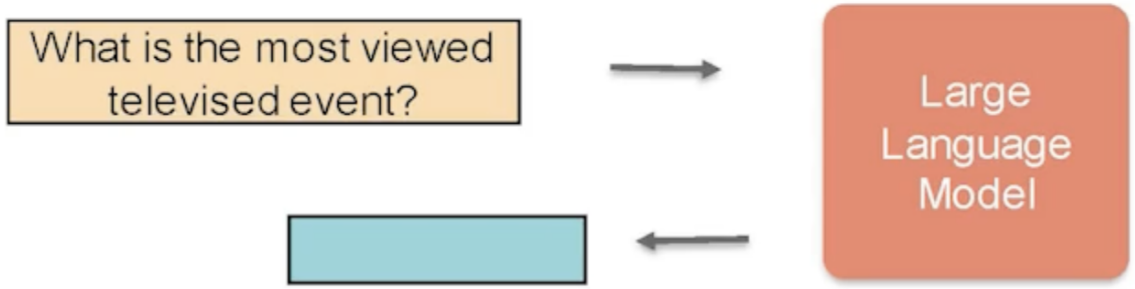
Solution: ReRank



ReRank is trained on



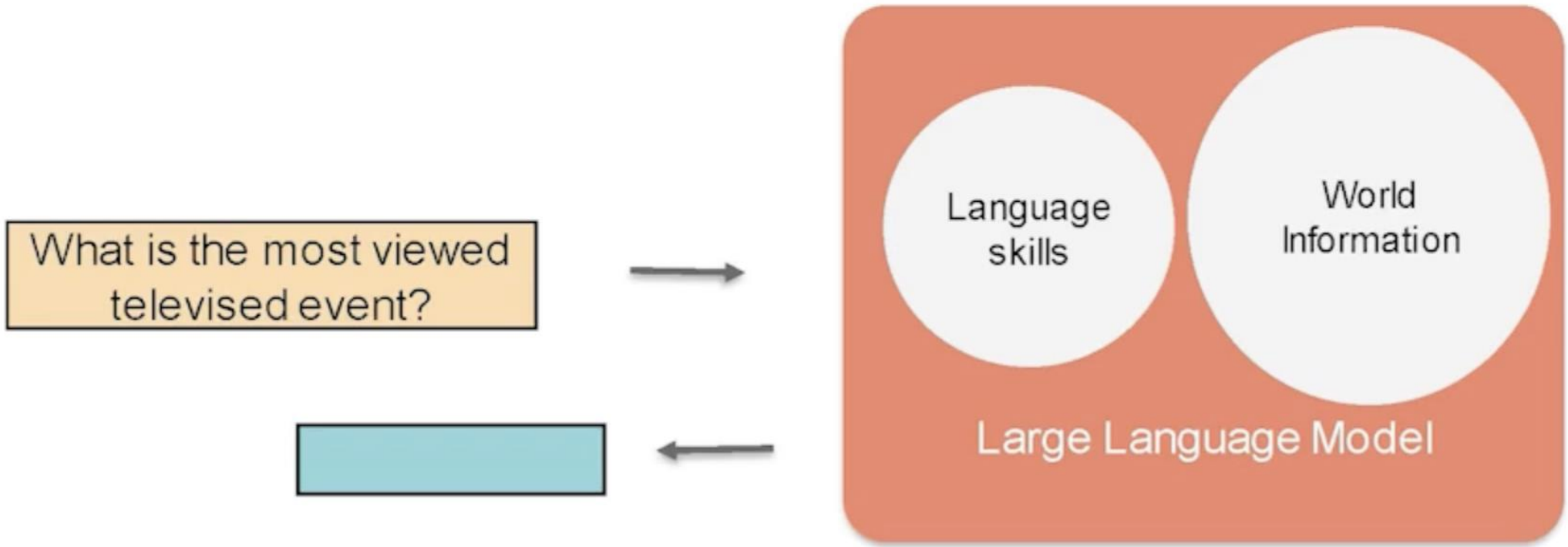
Search can help LLMs in multiple ways



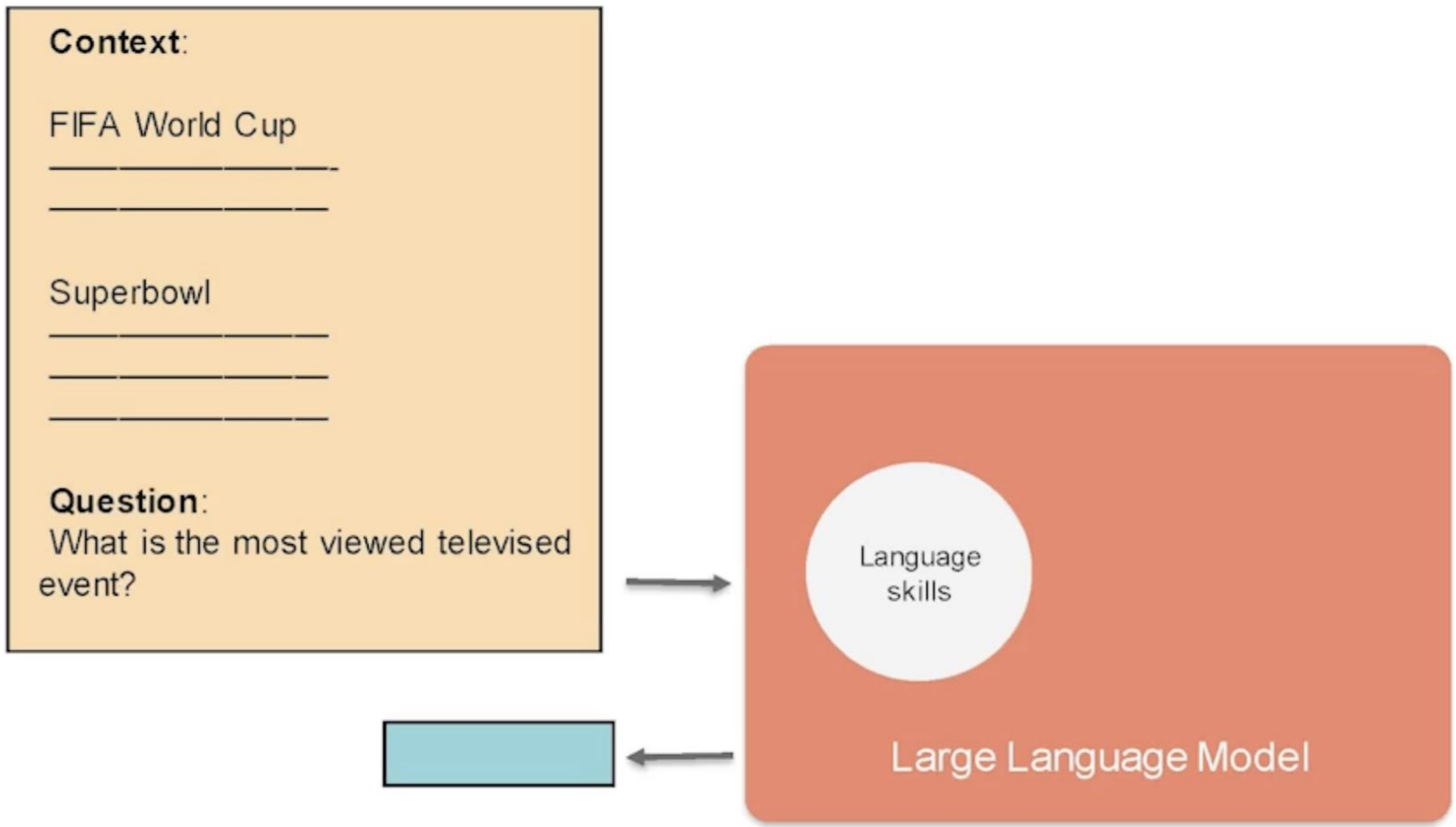
Retrieval can help LLMs with:

- Factual information
- Private information
- Updated information

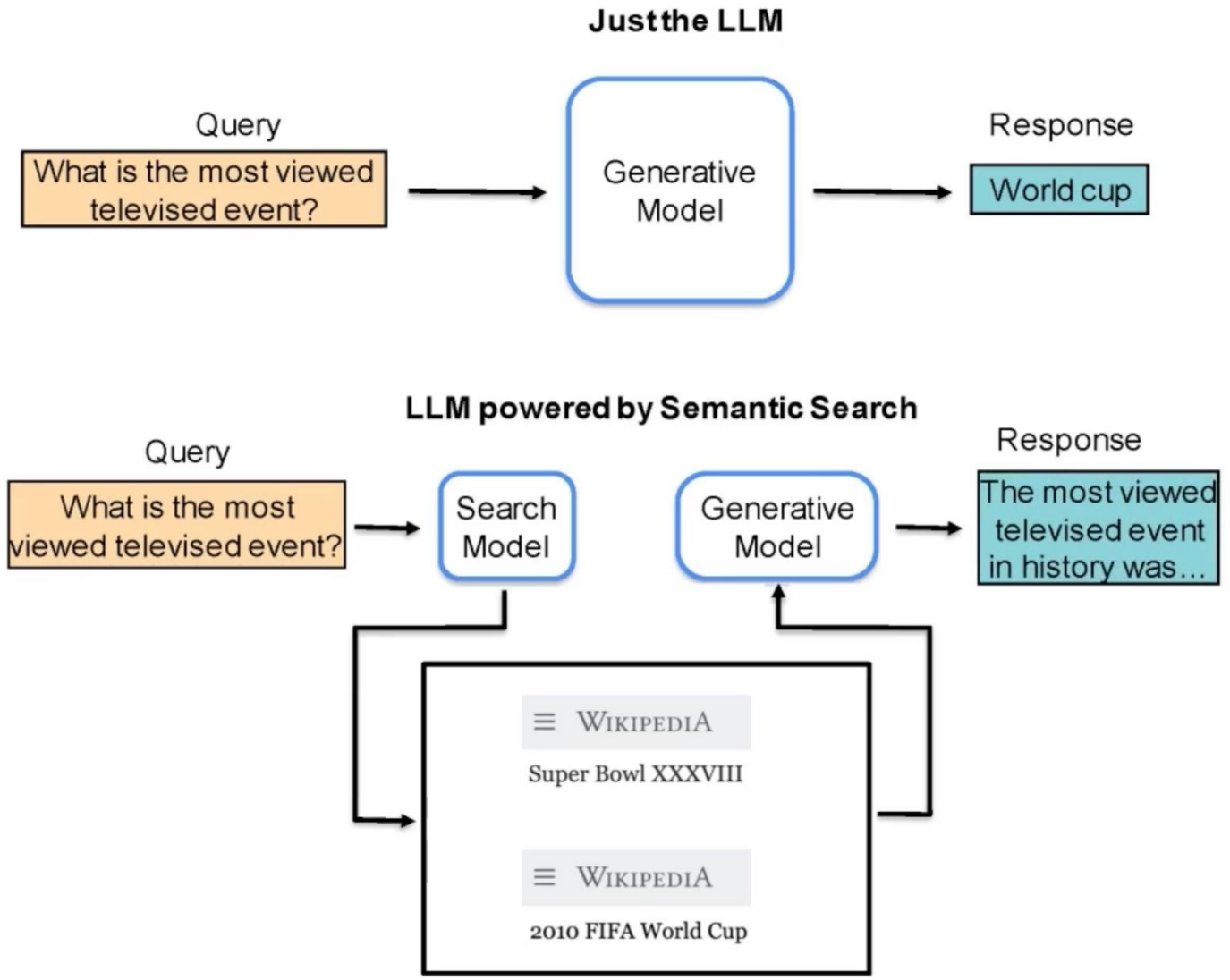
Where is the information stored?



Search can add some context



Generating Answers



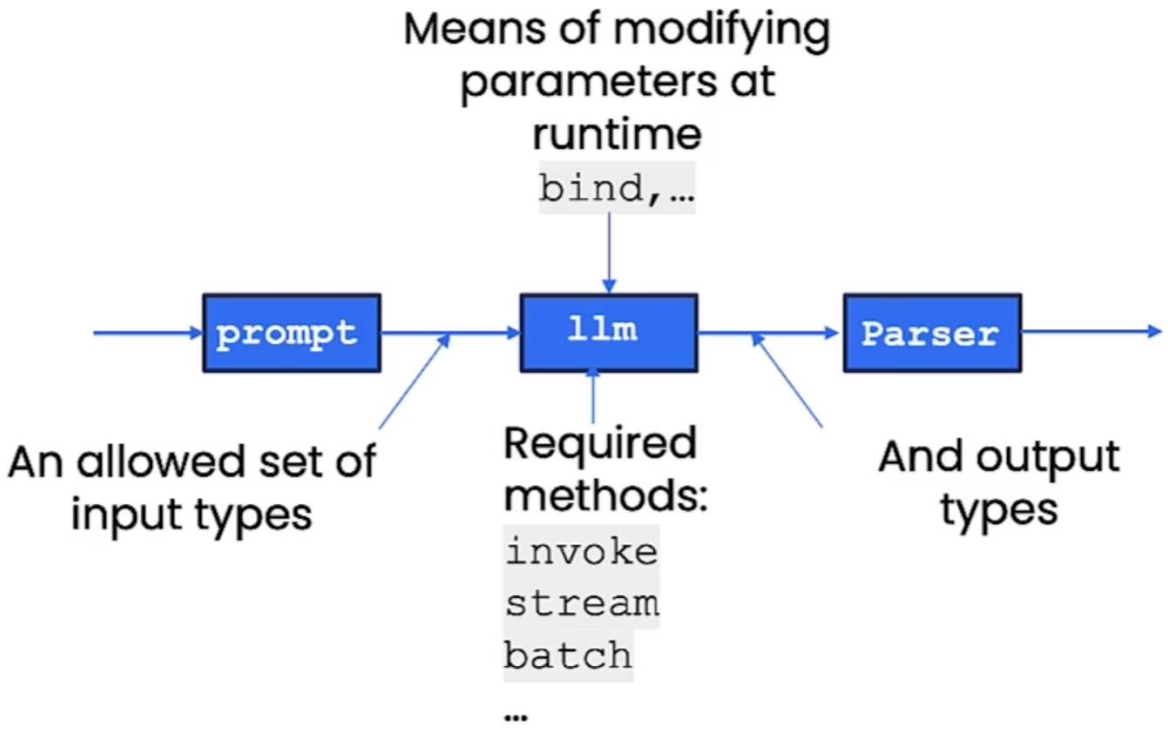
Function Call

- OpenAI has fine tuned GPT-3.5-turbo and GPT-4 models to
  - Accept additional arguments through which users can pass in descriptions of functions.
  - If it is relevant, return the name of the function to use, along with a JSON object with the appropriate input parameters.
- Notebook example:
  - https://drive.google.com/file/d/1U-Cb_ED3anGIfUvqm19ZKvVdBTBJb4kM/view?usp=sharing

LangChain Expression Language (LCEL)

LangChain composes chains of components

LCEL and the runnable protocol define:



Composition can now use the Linux pipe syntax:

```
Chain = prompt | llm | OutputParser
```

Interface

- Components implement "**Runnable**" protocol
- Common methods include:
  - `invoke [ainvoke]`
  - `stream [astream]`
  - `batch [abatch]`
- Common properties
  - `input_schema, output_schema`
- Common I/O

Component	Input Type	Output Type
Prompt	Dictionary	Prompt Value
Retriever	Single String	List of Documents
LLM	String, list of messages or Prompt Value	String
ChatModel	String, list of messages or Prompt Value	ChatMessage
Tool	String/Dictionary	Tool dependent
Output Parser	Output of LLM or ChatModel	Parser dependent

LangChain Expression Language (LCEL)

- Runnables support:
  - Async, Batch and Streaming Support
  - Fallbacks
  - Parallelism
    - LLM calls can be time consuming
    - Any components that can be run in parallel are!
  - Logging is built in

Pydantic

Pydantic is a 'data validation library' for python.

- Works with python type annotations. But rather than static type checking, they are actively used at runtime for data validation and conversion.
- Provides built-in methods to serialize/deserialize models to/from JSON, dictionaries, etc.
- LangChain leverages Pydantic to create JSON Scheme describing function.

Pydantic

```python
class pUser(BaseModel):
    name: str
    age: int
    email: str
```

```python
foo_p = pUser(name="Jane", age=32, email="jane@gmail.com")
```

```python
foo_p.name
```

```
'Jane'
```

```python
foo_p = pUser(name="Jane", age="bar", email="jane@gmail.com")
```

```
---------------------------------------------------------------------------
ValidationError                           Traceback (most recent call last)
Cell In[20], line 1
----> 1 foo_p = pUser(name="Jane", age="bar", email="jane@gmail.com")

File ~/Documents/GitHub/SC-langchain-openai/.venv_LC3/lib/python3.9/site-packages/pydantic/main.py:341, in pydantic.main.BaseModel.__init__()

ValidationError: 1 validation error for pUser
age
  value is not a valid integer (type=type_error.integer)
```

In normal python you would create a class like this:

```python
class User:
    def __init__(self, name: str, age: int, email: str):
        self.name = name
        self.age = age
        self.email = email
```

Using Pydantic, this is:

```python
from pydantic import BaseModel,

class User(BaseModel):
    name: str
    age: int
    email: EmailStr
```

Pydantic with LangChain for OpenAI function calls

Using multiple functions

Even better, we can pass a set of function and let the LLM decide which to use based on the question context.

```
class ArtistSearch(BaseModel):
    """Call this to get the names of songs by a particular artist"""
    artist_name: str = Field(description="name of artist to look up")
    n: int = Field(description="number of results")
```

```
functions = [
    convert_pydantic_to_openai_function(WeatherSearch),
    convert_pydantic_to_openai_function(ArtistSearch),
]
```

```
model_with_functions = model.bind(functions=functions)
```

```
model_with_functions.invoke("what is the weather in sf?")
```

```
AIMessage(content='', additional_kwargs={'function_call': {'name': 'Weat
herSearch', 'arguments': '{\n  "airport_code": "SFO"\n}'}})
```

```
model_with_functions.invoke("what are three songs by taylor swift?")
```

```
AIMessage(content='', additional_kwargs={'function_call': {'name': 'Arti
stSearch', 'arguments': '{\n  "artist_name": "Taylor Swift",\n  "n": 3
\n}'}})
```

Tagging and Extraction Using OpenAI functions

```
import os
import openai

from dotenv import load_dotenv, find_dotenv
_ = load_dotenv(find_dotenv()) # read local .env file
openai.api_key = os.environ['OPENAI_API_KEY']
```

```
from typing import List
from pydantic import BaseModel, Field
from langchain.utils.openai_functions import convert_pydantic_to_openai_f
```

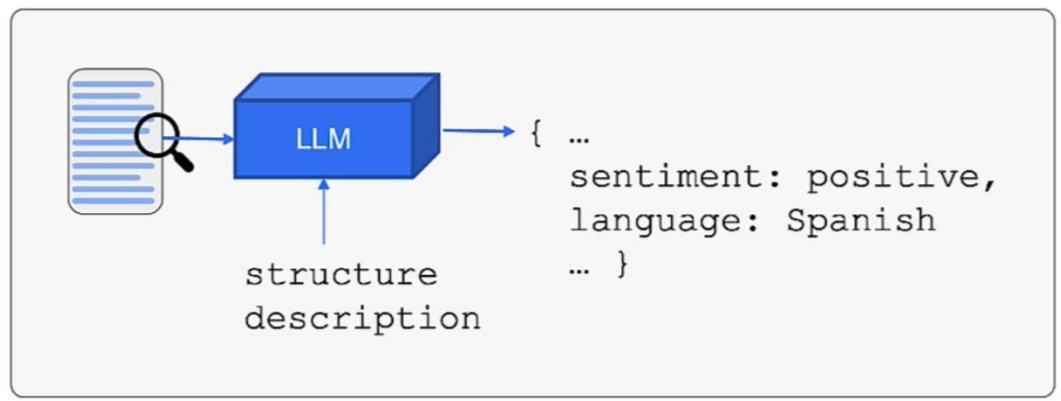
```
class Tagging(BaseModel):
    """Tag the piece of text with particular info."""
    sentiment: str = Field(description="sentiment of text, should be `pos
    language: str = Field(description="language of text (should be ISO 63
```

```
convert_pydantic_to_openai_function(Tagging)
```

```
{'name': 'Tagging',
 'description': 'Tag the piece of text with particular info.',
 'parameters': {'title': 'Tagging',
  'description': 'Tag the piece of text with particular info.',
  'type': 'object',
  'properties': {'sentiment': {'title': 'Sentiment',
    'description': 'sentiment of text, should be `pos`, `neg`, or `neutr
al`',
    'type': 'string'},
   'language': {'title': 'Language',
    'description': 'language of text (should be ISO 639-1 code)',
    'type': 'string'}},
  'required': ['sentiment', 'language']}}
```

Tagging

- We have seen the LLM, given a function description, select arguments from the input text generate a structured output forming a function call
- More generally, the LLM can *evaluate* the input text and generate structured output.



```
from langchain.prompts import ChatPromptTemplate
from langchain.chat_models import ChatOpenAI
```

```
model = ChatOpenAI(temperature=0)
```

```
tagging_functions = [convert_pydantic_to_openai_function(Tagging)]
```

```
prompt = ChatPromptTemplate.from_messages([
    ("system", "Think carefully, and then tag the text as instructed"),
    ("user", "{input}")
])
```

```
model_with_functions = model.bind(
    functions=tagging_functions,
    function_call={"name": "Tagging"}
)
```

```
tagging_chain = prompt | model_with_functions
```

```
tagging_chain.invoke({"input": "I love langchain"})
```

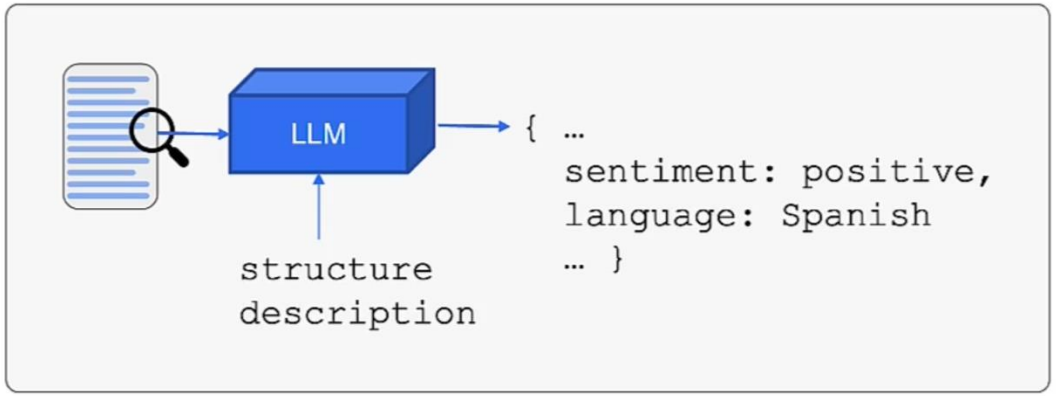
```
AIMessage(content='', additional_kwargs={'function_call': {'name': 'Tagg
ing', 'arguments': '{\n  "sentiment": "pos",\n  "language": "en"\n}'}})
```

```
tagging_chain.invoke({"input": "non mi piace questo cibo"})
```

```
AIMessage(content='', additional_kwargs={'function_call': {'name': 'Tagg
ing', 'arguments': '{\n  "sentiment": "neg",\n  "language": "it"\n}'}})
```

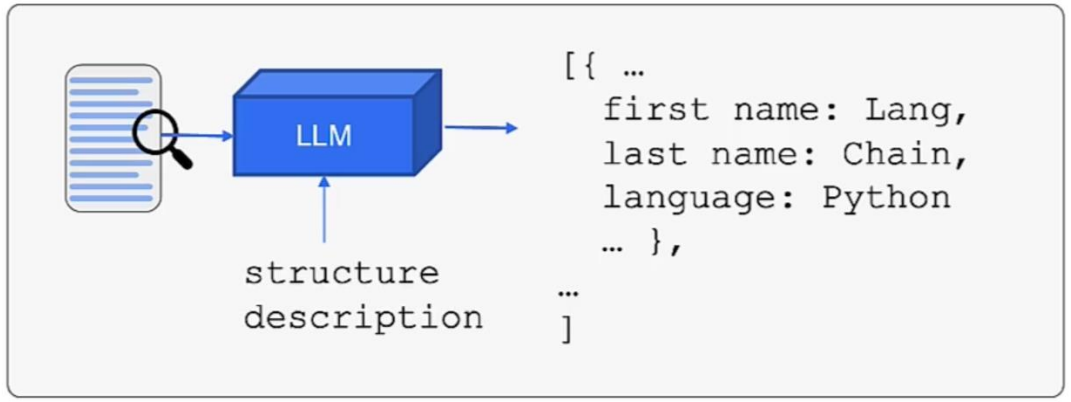
Tagging

- We have seen the LLM, given a function description, select arguments from the input text generate a structured output forming a function call
- More generally, the LLM can *evaluate* the input text and generate structured output.



Extraction

- When given an input JSON schema, the LLM has been fine tuned to find and fill in the parameters of that schema.
- The capability is not limited to function schema.
- This can be used for general purpose extraction.



Extraction

Extraction is similar to tagging, but used for extracting multiple pieces of information.

```
from typing import Optional
class Person(BaseModel):
    """Information about a person."""
    name: str = Field(description="person's name")
    age: Optional[int] = Field(description="person's age")
```

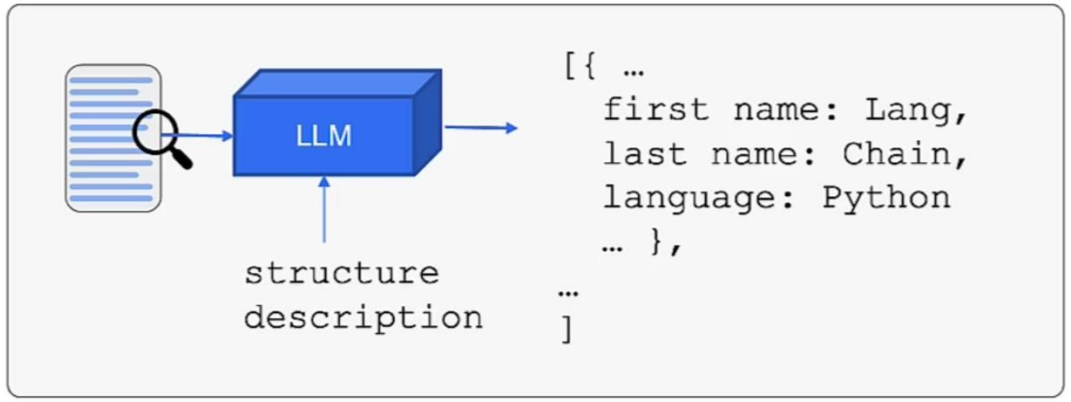
```
class Information(BaseModel):
    """Information to extract."""
    people: List[Person] = Field(description="List of info about people")
```

```
convert_pydantic_to_openai_function(Information)
```

```
{'name': 'Information',
 'description': 'Information to extract.',
 'parameters': {'title': 'Information',
  'description': 'Information to extract.',
  'type': 'object',
  'properties': {'people': {'title': 'People',
    'description': 'List of info about people',
    'type': 'array',
    'items': {'title': 'Person',
     'description': 'Information about a person.',
     'type': 'object',
     'properties': {'name': {'title': 'Name',
       'description': "person's name",
       'type': 'string'},
      'age': {'title': 'Age',
       'description': "person's age",
       'type': 'integer'}},
     'required': ['name']}}},
  'required': ['people']}}
```

Extraction

- When given an input JSON schema, the LLM has been fine tuned to find and fill in the parameters of that schema.
- The capability is not limited to function schema.
- This can be used for general purpose extraction.




```
    'type': 'object',
    'properties': {'name': {'title': 'Name',
      'description': "person's name",
      'type': 'string'},
     'age': {'title': 'Age',
      'description': "person's age",
      'type': 'integer'}},
    'required': ['name']}}},
  'required': ['people']}}
```

```
extraction_functions = [convert_pydantic_to_openai_function(Information)]
extraction_model = model.bind(functions=extraction_functions, function_ca
```

```
extraction_model.invoke("Joe is 30, his mom is Martha")
```

```
AIMessage(content='', additional_kwargs={'function_call': {'name': 'Info
rmation', 'arguments': '{\n  "people": [\n    {\n      "name": "Joe",\n
"age": 30\n    },\n    {\n      "name": "Martha",\n      "age": 0\n    }
\n  ]\n}'}})
```

```
prompt = ChatPromptTemplate.from_messages([
    ("system", "Extract the relevant information, if not explicitly provi
    ("human", "{input}")
])
```

```
extraction_chain = prompt | extraction_model
```

```
extraction_chain.invoke({"input": "Joe is 30, his mom is Martha"})
```

```
AIMessage(content='', additional_kwargs={'function_call': {'name': 'Info
rmation', 'arguments': '{\n  "people": [\n    {\n      "name": "Joe",\n
"age": 30\n    },\n    {\n      "name": "Martha"\n    }\n  ]\n}'}})
```

```
extraction_chain = prompt | extraction_model | JsonOutputFunctionsParser()
```

```
extraction_chain.invoke({"input": "Joe is 30, his mom is Martha"})
```

Tools and Routing

- Functions and services an LLM can utilize to extend its capabilities are named “tools” in LangChain
- LangChain has many tools available
  - Search tools
  - Math tools
  - SQL tools
  - …

```python
import requests
from pydantic import BaseModel, Field
import datetime

# Define the input schema
class OpenMeteoInput(BaseModel):
    latitude: float = Field(..., description="Latitude of the location to
    longitude: float = Field(..., description="Longitude of the location

@tool(args_schema=OpenMeteoInput)
def get_current_temperature(latitude: float, longitude: float) -> dict:
    """Fetch current temperature for given coordinates."""

    BASE_URL = "https://api.open-meteo.com/v1/forecast"

    # Parameters for the request
    params = {
        'latitude': latitude,
        'longitude': longitude,
        'hourly': 'temperature_2m',
        'forecast_days': 1,
    }

    # Make the request
    response = requests.get(BASE_URL, params=params)

    if response.status_code == 200:
        results = response.json()
    else:
        raise Exception(f"API Request failed with status code: {response.

    current_utc_time = datetime.datetime.utcnow()
    time_list = [datetime.datetime.fromisoformat(time_str.replace('Z', '+
    temperature_list = results['hourly']['temperature_2m']

    closest_time_index = min(range(len(time_list)), key=lambda i: abs(tim
    current_temperature = temperature_list[closest_time_index]
```

```python
import wikipedia
@tool
def search_wikipedia(query: str) -> str:
    """Run Wikipedia search and get page summaries."""
    page_titles = wikipedia.search(query)
    summaries = []
    for page_title in page_titles[: 3]:
        try:
            wiki_page =  wikipedia.page(title=page_title, auto_suggest=Fa
            summaries.append(f"Page: {page_title}\nSummary: {wiki_page.su
        except (
            self.wiki_client.exceptions.PageError,
            self.wiki_client.exceptions.DisambiguationError,
        ):
            pass
    if not summaries:
        return "No good Wikipedia Search Result was found"
    return "\n\n".join(summaries)
```

```python
search_wikipedia.name
```

```
'search_wikipedia'
```

```python
search_wikipedia.description
```

```
'search_wikipedia(query: str) -> str - Run Wikipedia search and get page
summaries.'
```

```python
format_tool_to_openai_function(search_wikipedia)
```

```
{'name': 'search_wikipedia',
 'description': 'search_wikipedia(query: str) -> str - Run Wikipedia sea
rch and get page summaries.',
 'parameters': {'title': 'search_wikipediaSchemaSchema',
  'type': 'object',
  'properties': {'query': {'title': 'Query', 'type': 'string'}},
```

```python
functions = [
    format_tool_to_openai_function(f) for f in [
        search_wikipedia, get_current_temperature
    ]
]
model = ChatOpenAI(temperature=0).bind(functions=functions)
```

```python
model.invoke("what is the weather in sf right now")
```

```
AIMessage(content='', additional_kwargs={'function_call': {'name': 'get_
current_temperature', 'arguments': '{\n  "latitude": 37.7749,\n  "longit
ude": -122.4194\n}'}})
```

```python
model.invoke("what is langchain")
```

```
AIMessage(content='', additional_kwargs={'function_call': {'name': 'sear
ch_wikipedia', 'arguments': '{\n  "query": "langchain"\n}'}})
```

```python
from langchain.prompts import ChatPromptTemplate
prompt = ChatPromptTemplate.from_messages([
    ("system", "You are helpful but sassy assistant"),
    ("user", "{input}"),
])
chain = prompt | model
```

```python
chain.invoke({"input": "what is the weather in sf right now"})
```

```
AIMessage(content='', additional_kwargs={'function_call': {'name': 'get_
current_temperature', 'arguments': '{\n  "latitude": 37.7749,\n  "longit
ude": -122.4194\n}'}})
```

```
chain.invoke({"input": "what is the weather in sf right now"})
```

```
AIMessage(content='', additional_kwargs={'function_call': {'name': 'get_
current_temperature', 'arguments': '{\n  "latitude": 37.7749,\n  "longit
ude": -122.4194\n}'}})
```

```
from langchain.agents.output_parsers import OpenAIFunctionsAgentOutputPar
```

```
chain = prompt | model | OpenAIFunctionsAgentOutputParser()
```

```
result = chain.invoke({"input": "what is the weather in sf right now"})
```

```
type(result)
```

```
langchain.schema.agent.AgentActionMessageLog
```

```
result.tool
```

```
'get_current_temperature'
```

```
result.tool_input
```

```
{'latitude': 37.7749, 'longitude': -122.4194}
```

```
get_current_temperature(result.tool_input)
```

```
'The current temperature is 22.9°C'
```

```
{ output : Hello! How can I assist you today? }
```

```
from langchain.schema.agent import AgentFinish
def route(result):
    if isinstance(result, AgentFinish):
        return result.return_values['output']
    else:
        tools = {
            "search_wikipedia": search_wikipedia,
            "get_current_temperature": get_current_temperature,
        }
        return tools[result.tool].run(result.tool_input)
```

```
chain = prompt | model | OpenAIFunctionsAgentOutputParser() | route
```

```
result = chain.invoke({"input": "What is the weather in san francisco rig
```

```
result
```

```
'The current temperature is 22.9°C'
```

```
result = chain.invoke({"input": "What is langchain?"})
```

```
result
```

```
'Page: LangChain\nSummary: LangChain is a framework designed to simplify
the creation of applications using large language models (LLMs). As a la
nguage model integration framework, LangChain\'s use-cases largely overl
ap with those of language models in general, including document analysis
and summarization, chatbots, and code analysis.\n\nPage: Prompt engineer
ing\nSummary: Prompt engineering is the process of structuring text that
can be interpreted and understood by a generative AI model. A prompt is
natural language text describing the task that an AI should perform.A pr
ompt for a text-to-text model can be a query such as "what is Fermat\'s
little theorem?", a command such as "write a poem about leaves falling",
a short statement of feedback (for example, "too verbose", "too formal",
"rephrase again", "omit this word") or a longer statement including cont
```

Agent Basics

- Agents
  - Are a combination of LLMs and code
  - LLMs reason about what steps to take and call for actions.
- Agent loop
  - Choose a tool to use
  - Observe the output of the tool
  - Repeat until a stopping condition is met
- Stopping conditions can be:
  - LLM determined
  - Hardcoded rules