Vector Database

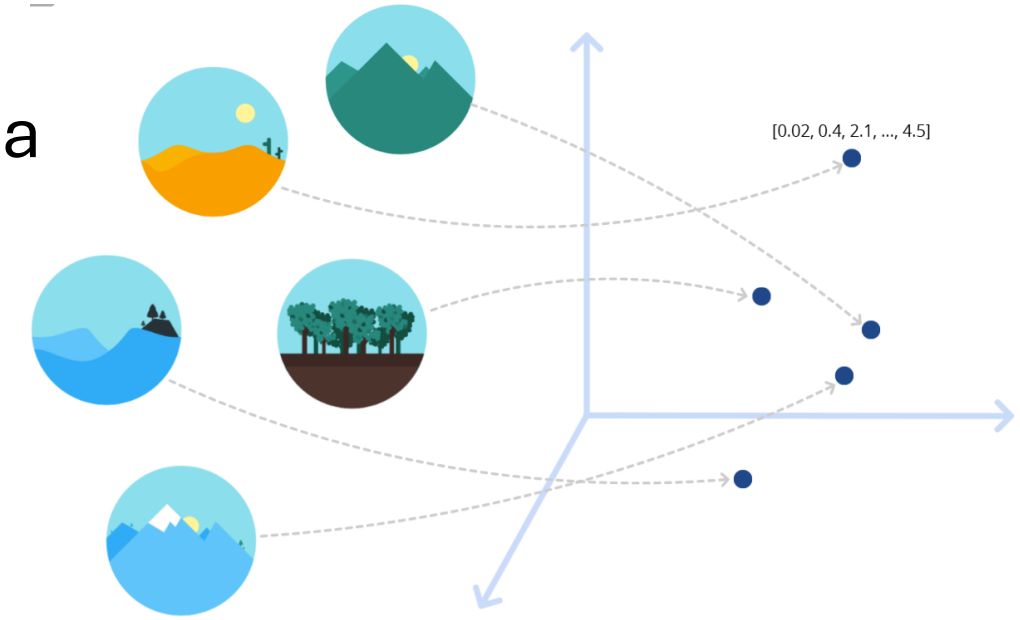
Powering AI Search and Semantic Understanding

What is a Vector Database?

- A **Vector Database** is a specialized data store designed to manage, index, and query **high-dimensional vector embeddings**.
- These vectors represent **semantic meaning** often generated by machine learning models (like embeddings from text, image, or audio).
- Enables **similarity search** rather than traditional exact matching.

What are embeddings?

- **Embeddings** are **numerical representations** of data (text, images, audio, etc.) in a **high-dimensional space**.
- Each piece of data is converted into a **vector,** a list of numbers that captures its **semantic meaning**.
- Similar meanings → vectors close together.
- Different meanings → vectors far apart.



A simple example with textual data

Words:

- “king” → [0.21, 0.89, 0.42, ...]
- “queen” → [0.20, 0.88, 0.43, ...]
- “car” → [0.85, 0.13, 0.09, ...]

The vectors for “king” and “queen” are **close together** because they’re semantically related (royalty).
The vector for “car” is **far away** because it belongs to a different concept.
We can think each column identifies a “concept”, depending on the embedding model selected.

Concept are learned by the embedding model: they don’t have a concrete meaning.

How embeddings are created

Embeddings come from **machine learning models** trained on large datasets.

Examples:

- **Word2Vec –** early model for word embeddings
- **GloVe –** captures word relationships using co-occurrence statistics
- **BERT / OpenAI Embeddings –** modern models that capture **contextual meaning**, event on long sentences
- CLIP – Model that learns image embedding, aligned with their descriptions

Each model learns to **map meaning** into **numbers** based on context and similarity.

Why do we need Vector Databases?

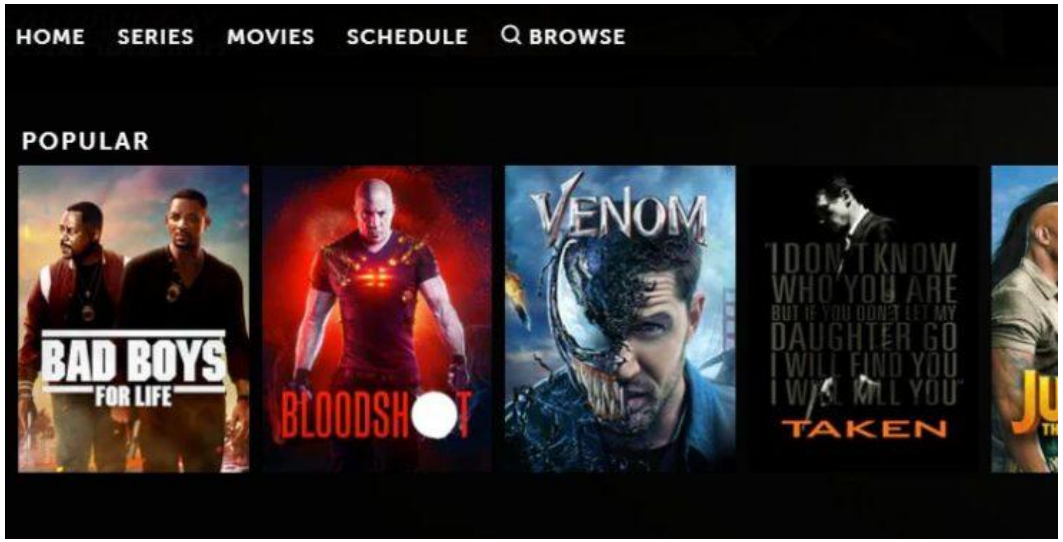
- Traditional databases struggle with **unstructured data** (e.g., text, images).
- Vector databases allow **semantic retrieval**, so “find things that mean the same”, not just “match keywords”
- Embedding similarity comparison made naively is **extremely slow**
- They are critical for building applications such as:
  - chatbots and recommendation systems
  - image and video search (not keyword-base)
  - document and knowledge retrieval

Popular Vector Databases

- **Pinecone** – Managed vector database with API integration
- **Qdrant** – Vector DB with hybrid and schema-based search
- **Milvus** – Scalable open-source vector engine
- **FAISS** (by Meta) – Library for efficient similarity search

Real-World use cases

- **Retrieval-Augmented Generation (RAG)**
- **Personalized Recommendations** (e.g., movies, products)
- **Semantic Document Search**
- **Image Similarity Search** (visual recognition, e-commerce)



How a Vector DB works

- **Data embedding:** convert data (text, images, etc.) into numerical vector representations using AI models.
- **Indexing:** store vectors in optimized structures (e.g., HNSW, IVF, or PQ indexes).
- **Querying:** compare **query** vectors using **similarity metrics** (cosine, Euclidean, etc.).
  - Query can be images (e.g., when finding images close to a target one), text (e.g., one question, when looking for a bunch of possible answers)
- **Results:** retrieve the **k most similar** items based on vector distance.

Similarity search types

- **k-Nearest Neighbors (k-NN)**
  - Returns the *k* most similar items.
  - Most common search type.
- **Range search**
  - Returns all items within a similarity threshold (e.g., distance < 0.5).
- **Hybrid search**
  - Combines **vector search** with **keyword filtering** or **metadata constraints**.

Example hybrid query: “Find articles about *Tesla* that are semantically similar to *electric car innovation*.”

To efficiently do this, Vector Databases use indexing.

Vector indexing

- **Indexing** is the process of organizing vectors to make **similarity search fast and efficient**.
- Without an index, finding the nearest vectors would require comparing every vector, which is **too slow** for large datasets.
- Vector indexes enable **Approximate Nearest Neighbor (ANN)** search by retrieving the most similar items quickly, even among **billions of vectors**.

Indexing basics – similarity search

Vector indexes rely on **distance metrics** to measure similarity

- **Cosine similarity:** measures angle between vectors (used for text).
- **Euclidean distance:** measures straight-line distance (used for images).
- **Dot product:** measures projection of one vector onto another.

The index stores and organizes vectors so these comparisons are **optimized**.

Cosine similarity

Measures the **angle** between two vectors.

Ignores the length (magnitude) and focuses on **directional similarity**.

Formula:

- Cosine *Similarity* $= \frac{A \cdot B}{\|A\|\|B\|}$

Values range from **-1 (opposite)** to **1 (identical)**.

Best for: text embeddings, NLP applications (semantic meaning).

The need for indexing

Imagine you have **100 million embeddings**.

A brute-force search checks all 100M vectors for every query, becoming **very slow**.

Indexing creates a **structure** that allows the database to skip most comparisons, at the price of small accuracy trade-off.

Types of index structures

Common **ANN (Approximate Nearest Neighbor)** techniques include

- **HNSW (Hierarchical Navigable Small World)**
  - Graph-based structure
  - Balances speed and accuracy
  - Used in Weaviate, Pinecone, Milvus
- **IVF (Inverted File Index)**
  - Clusters vectors into “buckets”
  - Speeds up search by checking only relevant buckets
- **PQ (Product Quantization)**
  - Compresses vectors to save memory
  - Enables large-scale search with minimal loss

HNSW

- **HNSW (Hierarchical Navigable Small World)** is a **graph-based indexing algorithm** used for **Approximate Nearest Neighbor (ANN)** search.
- It builds a **multi-layer network** of vectors, where each node connects to a small number of nearby vectors.
- Designed for **high recall**, **fast search**, and **efficient updates** ideal for AI-scale vector databases

The core idea of HNSW

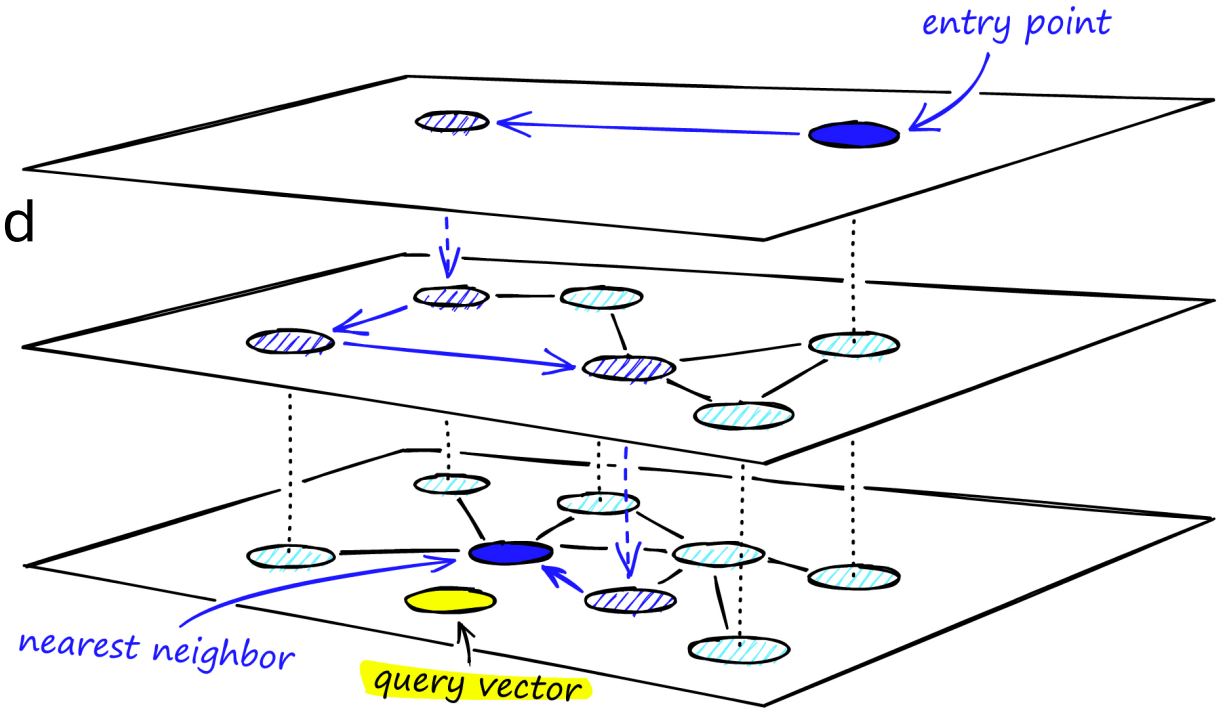
- Every vector is a **node** in a **navigable graph**
- Each node connects to several **neighbors** (similar vectors)
- The graph has **multiple layers**
  - Top layers have **fewer nodes** (for broad, fast navigation)
  - Lower layers have **denser connections** (for fine-grained search)
- The structure lets the search “zoom in” on the most relevant vectors step by step

HNSW Structure Overview

Visualize it like a **pyramid**

- **Top layer:** few entry points, global view of the space.
- **Middle layers:** increasingly detailed neighborhood connections.
- **Bottom layer:** full vector network.

Each new vector is placed into multiple layers, forming links to its **nearest neighbors**



How search works in HNSW

When a query comes in

- Start at the **top layer** with an entry node
- Move to the neighbor that's **closest** to the query vector
- Continue until no closer neighbor exists
- Descend to the **next layer** and repeat
- Once at the **bottom layer**, perform a **local nearest neighbor search** for exact ranking

This hierarchical descent drastically reduces the number of comparisons needed

Trade-offs in indexing

- **Speed vs. Accuracy:** faster search may skip some near-perfect matches
- **Memory vs. Compression:** smaller indexes can reduce precision
- **Dynamic vs. Static:**
  - *Static indexes* are optimized for fixed data
  - *Dynamic indexes* can handle frequent updates or inserts

Balancing these trade-offs is key to good performance

Example - text query

User query: “Best ways to reduce electricity consumption”

- Embedding model → vector: [0.12, 0.47, -0.56, …]
- Compare against all stored vectors (e.g., articles)
- Retrieve nearest vectors (highest cosine similarity)

Results

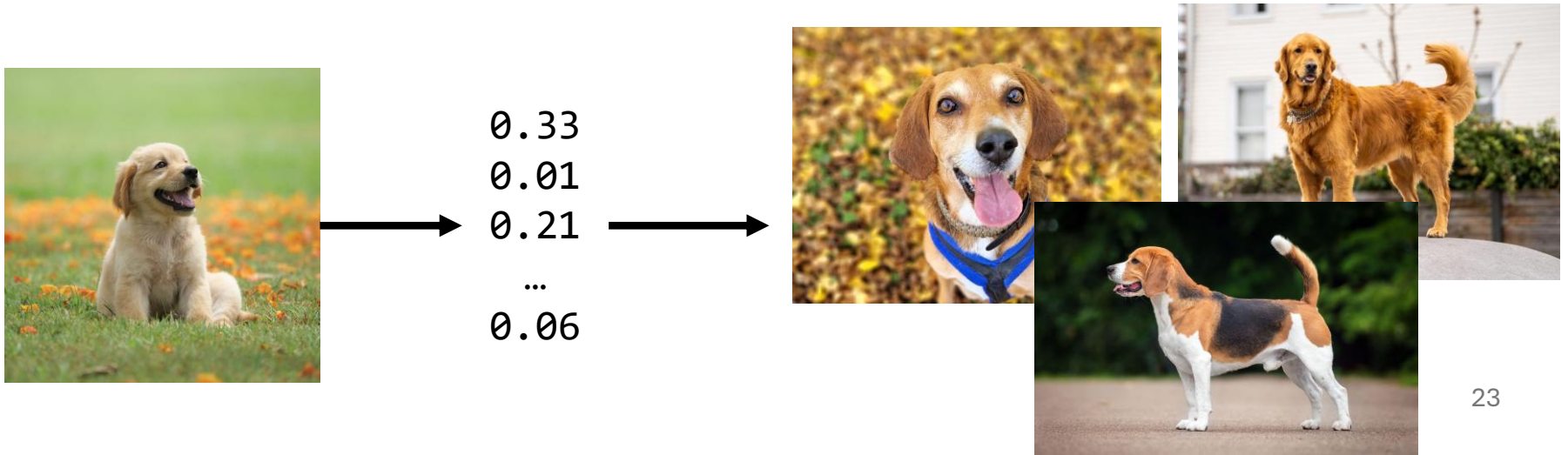
- “How to save power at home”
- “Tips for lowering energy bills”
- “Smart home energy management”

Even though wording differs, **meaning is the same**

Example — image query

- **User Query:** find a photo of a dog
  - Convert image into embedding
  - Search for similar image embeddings using **cosine distance**
  - Retrieve images that are **visually or semantically similar** (e.g., other dogs, similar breeds)

Used in e-commerce, visual search, and content moderation systems.



Querying and similarity search

- Queries in vector databases are **meaning-based**, not keyword-based
- Embeddings + Indexing + Similarity Metrics = **Semantic Search Engine**.
- Supports **multimodal** (text, image, audio) queries
- Core to modern **AI, recommendation, and knowledge retrieval** systems

RAG (Retrieval-Augmented Generation)

RAG is a technique that augments LLMs by combining **retrieval-based methods** (i.e., a Vector DB) with generation-based capabilities (an LLM).

Unlike standalone language models, RAG retrieves relevant information from an external knowledge base to

- enrich the LLM’s responses
- reduce hallucination.
- provide up-to-date answers

How does RAG work?

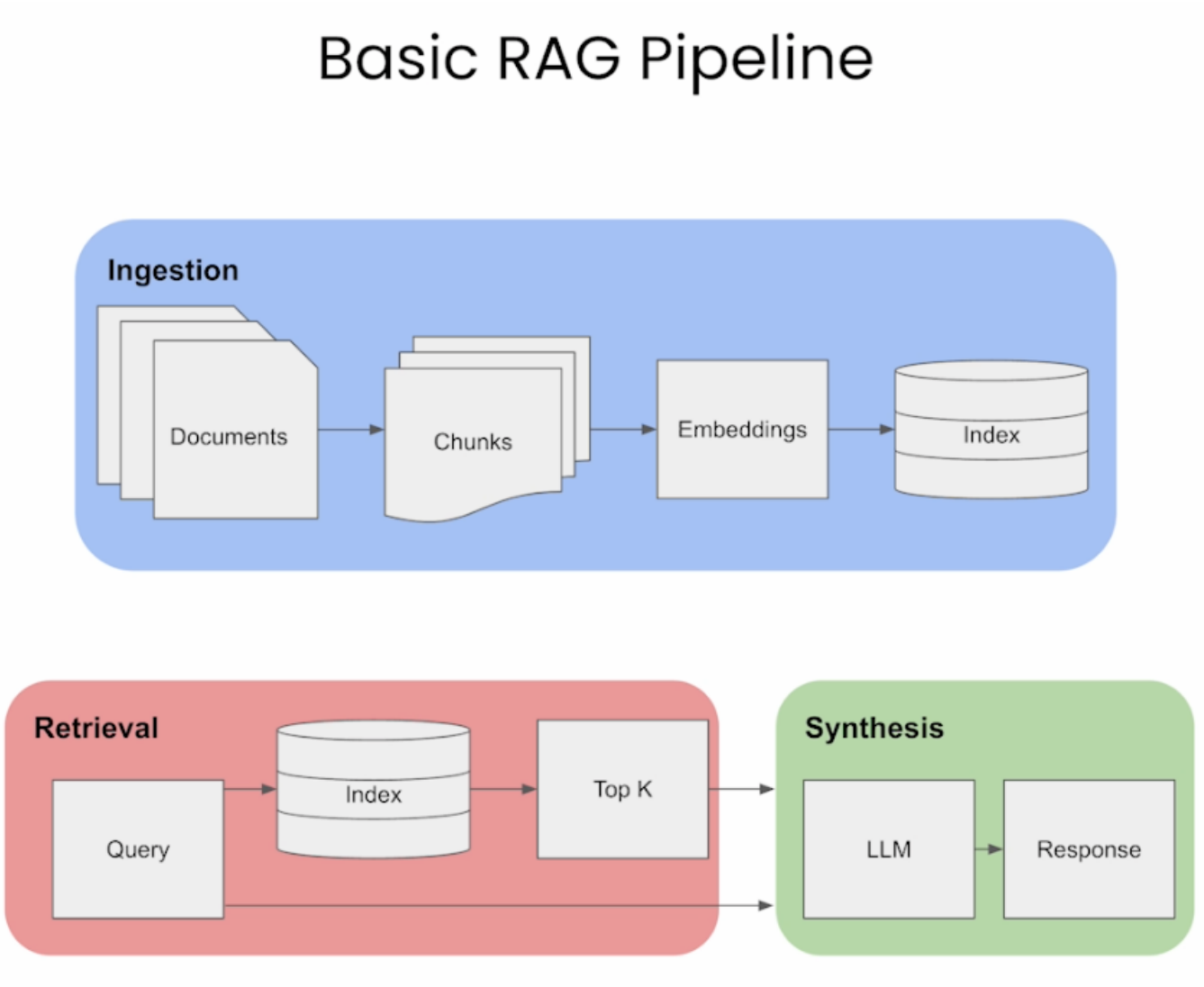
RAG consists of two main components

- **Retriever:** this component fetches relevant information from a large dataset based on the user's query
- **Generator:** after retrieving the information, the generator (usually an LLM) uses it to generate a more accurate, context-aware response.

How does RAG work?

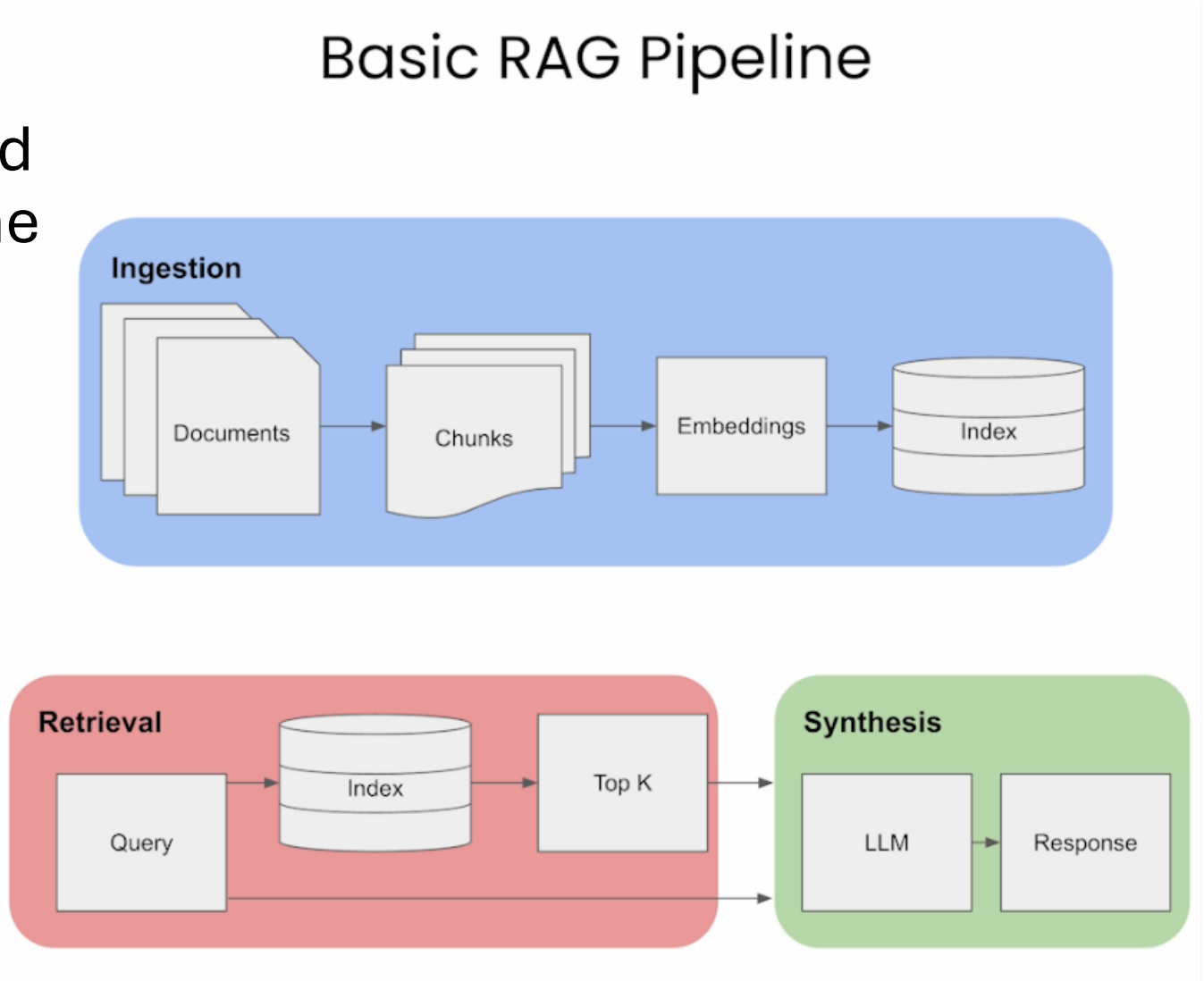
1 The knowledge base (books, documentation, etc.) is split into **chunks** to

- allow the creation of more aligned embeddings
- allow the generator to focus on more centred portion of the text



How does RAG work?

2 Get the **query embedding** and perform a Vector search on the database. The result is the **Context**.



How does RAG work?

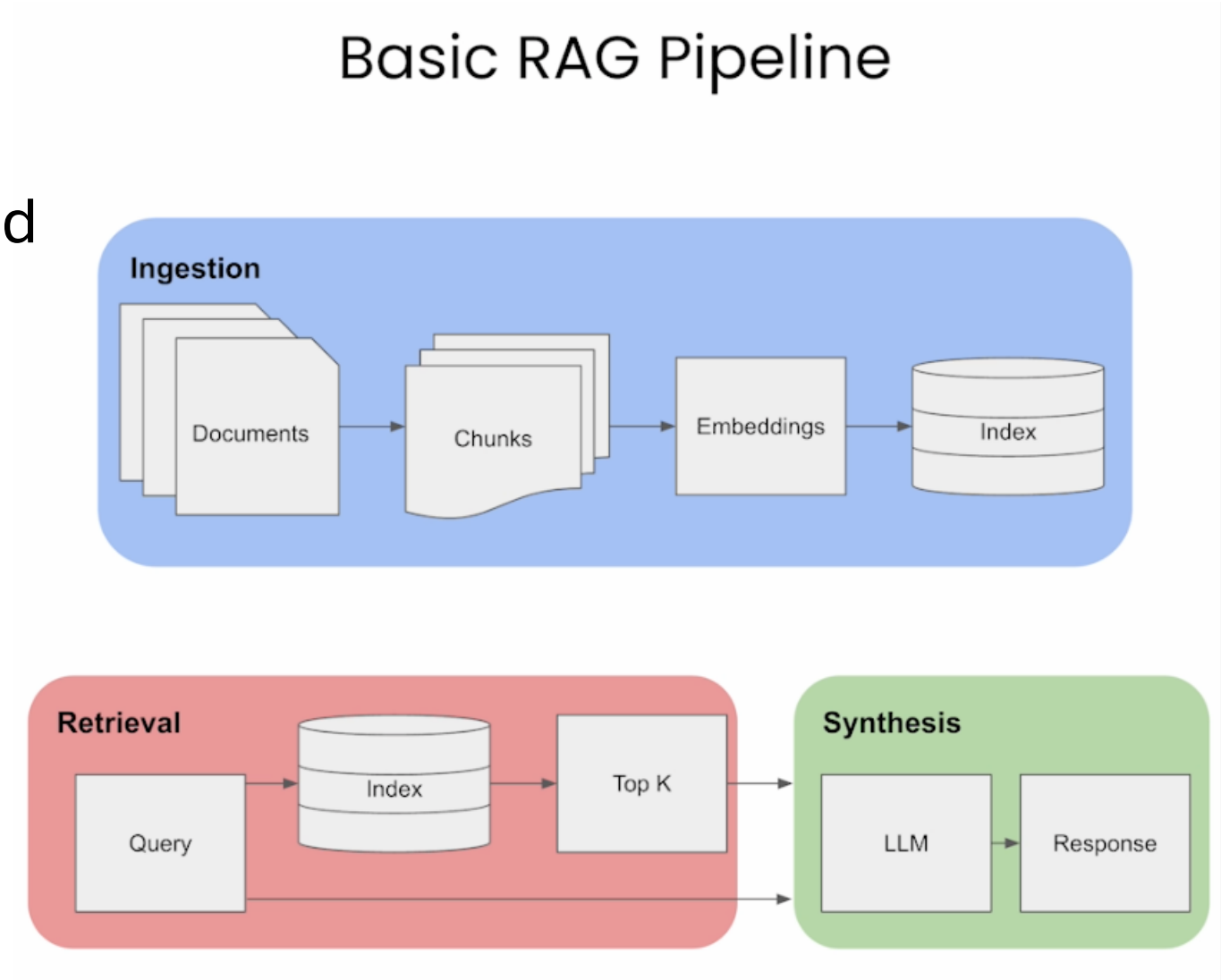
3 The generator produces an answer to the **query** grounded on the **context**, e.g.:

```
You are a helpful chatbot.

Use only the following pieces of
context to answer the question.
Don't make up any new information:

{CONTEXT}

to answer to: {QUERY}
```



MongoDB + Vector search

MongoDB can be used as a vector database too.

This utility allows to seamlessly search and index vector data alongside all other MongoDB data.

- You must create a MongoDB Vector Search index. MongoDB Vector Search indexes are separate from your other database indexes and are used to efficiently retrieve documents that contain vector embeddings at query-time.
- It supports both ANN with HNSW and ENN (Exact Nearest Neighbor)

Vector search pipeline

MongoDB Vector Search queries are **aggregation pipeline stages** where the $vectorSearch stage is the first stage.

- You select either ANN or ENN search and specify the **query vector**, that represents your search query
- MongoDB Vector Search finds vector embeddings in your data that are closest to the query vector

```
{
  "$vectorSearch": {
    "exact": true | false,
    "filter": {<filter-specification>},
    "index": "<index-name>",
    "limit": <number-of-results>,
    "numCandidates": <number-of-candidates>,
    "path": "<field-to-search>",
    "queryVector": [<array-of-numbers>],
    "explainOptions": {
      "traceDocumentIds": [<array-of-documentIDs>]
    }
  }
}
```