

Vector Stores_FAISS

ALL Vector Store databases:

<https://python.langchain.com/v0.2/docs/integrations/vectorstores/>

Vectorstores |  LangChain

A vector store stores embedded data and performs similarity search.

 <https://python.langchain.com/v0.2/docs/integrations/vectorstores/>



What are Vector Stores?

- Databases optimized for storing and querying **vector embeddings**..
- These allow the system to find semantically similar documents or text chunks based on their meaning, not just exact keyword matches.

Top VectorStores in LangChain (2024)

Name	Type	Best For	Example
FAISS	Local	Fast prototyping	<code>FAISS.from_documents(docs, embeddings)</code>
Chroma	Local/Server	Persistent storage	<code>Chroma.from_documents(docs, embeddings, persist_dir="./db")</code>
Pinecone	Cloud	Production-scale	<code>Pinecone.from_documents(docs, embeddings, index_name="my-index")</code>
Weaviate	Hybrid	Graph+Vector search	<code>Weaviate.from_documents(docs, embeddings, by_text=False)</code>

Why Are They Needed?

- Traditional search looks for **exact matches** of words.

- Vector stores let us search by **meaning** (semantic similarity), using **cosine similarity** or **dot product** of vectors.

For example:

Query: "**How to boil rice?**"

Vector search might return a chunk that says: "Cooking plain rice involves boiling it in water..."

Even though "boil" and "cooking" are not exact matches — the meanings are close.

How It Works Internally?

1. **Text Input:** You start with some documents or text chunks.
2. **Embedding:** These are converted into numerical vectors using an **embedding model**.
3. **Storing:** The resulting vectors are saved in a **Vector Store** (e.g., FAISS, Chroma, etc.).
4. **Querying:**
 - Your search/query is also embedded into a vector.
 - The vector store compares this query vector to all stored vectors.
 - It returns the most similar ones based on a mathematical score (e.g., cosine similarity).

Examples of Vector Stores LangChain Supports

Vector Store	Description
FAISS	Fast, local, and simple. Works offline. Ideal for learning or small apps.
Chroma	Lightweight and modern. Good for simple cloud/local hybrid use.
Pinecone	Fully managed cloud-based. Scalable, production-ready.

Vector Store	Description
Weaviate	Open-source with advanced semantic capabilities.
Qdrant	Optimized for neural search. Offers filtering and hybrid search.
Milvus	High-performance for large-scale vector databases.
Supabase	Not a native vector store, but can be used with extensions like pgvector.

Python Code for Vector Store

```

pip install langchain-chroma # For Chroma
# or
pip install faiss-cpu langchain-community # For FAISS

```

```

from langchain_community.vectorstores import FAISS
from langchain.embeddings import OpenAIEmbeddings # or use HuggingFace

# Step 1: Create embedding model
embedding_model = OpenAIEmbeddings()

# Step 2: Sample documents
texts = ["Apple is a fruit.", "Dogs are loyal animals.", "Quantum computing is powerful."]

# Step 3: Convert to vector store
vectorstore = FAISS.from_texts(texts, embedding_model)

# Step 4: Query
results = vectorstore.similarity_search("What is quantum?", k=1)

print(results[0].page_content) # likely: "Quantum computing is powerful."

```

Querying

```
# Similarity search
results = vectorstore.similarity_search("What is LangChain?", k=3)

# With score thresholds
results = vectorstore.similarity_search_with_relevance_scores("Query", score_
threshold=0.7)

# Hybrid search (if supported)
results = vectorstore.similarity_search("Query", filter={"category": "AI"})
```

Full Code:

```
from langchain_community.document_loaders import TextLoader
from langchain_community.vectorstores import FAISS
from langchain_ollama.embeddings import OllamaEmbeddings
from langchain_text_splitters import CharacterTextSplitter

loader=TextLoader("speech.txt")
documents=loader.load()
text_splitter=CharacterTextSplitter(chunk_size=1000,chunk_overlap=30)
docs=text_splitter.split_documents(documents)
docs
```

```
[Document(metadata={'source': 'speech.txt'}, page_content='The world must be made safe for democ
Document(metadata={'source': 'speech.txt'}, page_content='...\\n\\nIt will be all the easier for u
Document(metadata={'source': 'speech.txt'}, page_content='We have borne with their present gov
Document(metadata={'source': 'speech.txt'}, page_content='It is a distressing and oppressive d
Document(metadata={'source': 'speech.txt'}, page_content='To such a task we can dedicate our l
```

```
embeddings=OllamaEmbeddings()
db=FAISS.from_documents(docs,embeddings)
db
```

```
<langchain_community.vectorstores.faiss.FAISS at 0x1e2b05abc40>
```

Query:

```
query="How does the speaker describe the desired outcome of the war?"  
docs=db.similarity_search(query)  
docs[0].page_content
```

Output:

'It is a distressing and oppressive duty, gentlemen of the Congress, which I have performed in thus addressing you. There are, it may be, many months of fiery trial and sacrifice ahead of us. It is a fearful thing to lead this great peaceful people into war, into the most terrible and disastrous of all wars, civilization itself seeming to be in the balance. But the right is more precious than peace, and we shall fight for the things which we have always carried nearest our hearts—for democracy, for the right of those who submit to authority to have a voice in their own governments, for the rights and liberties of small nations, for a universal dominion of right by such a concert of free peoples as shall bring peace and safety to all nations and make the world itself at last free.'

When to Choose Which?

Use Case	Recommended Store
Quick prototyping	FAISS
Persistent local storage	Chroma
Large-scale production	Pinecone/Weaviate
Multi-modal data	Weaviate

FAISS. `from_texts` **and** **FAISS.** `from_documents`

FAISS. from_texts (texts, embedding) :

```
texts = ["Hello world", "This is another chunk"]  
db = FAISS.from_texts(texts, embedding)
```

Result:

- Embeddings are computed for each string.
- Stored in FAISS as vectors with **no associated metadata or source info**.

FAISS. from_documents (documents, embedding)

Each `Document` contains both:

- `.page_content` : the text content
- `.metadata` : a dictionary of metadata (e.g., filename, chunk number)

When to use?

- Working with document loaders (PDFs, web pages, etc.)
- Need to filter/search by metadata



TL;DR:

- `from_texts` = simpler, just text
- `from_documents` = richer, retains metadata

Retriever

- A **Retriever** is a component in LangChain that helps you **fetch relevant information (documents or chunks)** from a database (usually a Vector Store) based on a **query**. It's like a **search engine helper** that brings back only the useful chunks to help an LLM answer your question better.

Why Use a Retriever?

Without a retriever, your LLM wouldn't have access to your custom data (like PDFs, notes, CSVs, websites).

A retriever solves this by:


- Finding the most relevant chunks from your stored documents.
- Passing them to the LLM to use in answering.

✓ Used in **Retrieval-Augmented Generation (RAG)** pipelines.

User → Retriever → Relevant Docs → LLM → Answer

How It Works?

Here's a step-by-step analogy:

Step	Description
1. Embed	You embed your documents into vectors and store them in a Vector Store.
2. Query	The user asks a question (e.g., "What is quantum computing?").
3. Embed Q	The retriever embeds the question into a vector.
4. Search	It compares this vector to those in the Vector Store (e.g., via cosine sim).
5. Return	It returns the top  most similar chunks of text.

```
from langchain_community.vectorstores import FAISS
from langchain.embeddings import HuggingFaceEmbeddings
from langchain_core.documents import Document

# Embedder
embedder = HuggingFaceEmbeddings(model_name="all-MiniLM-L6-v2")
```

```
# Some sample documents
docs = ["LangChain is used for LLM applications.", "Quantum computing is a new field."]

# Step 1: Create Vector Store
vectorstore = FAISS.from_texts(docs, embedder)

# Step 2: Create Retriever from Vector Store
retriever = vectorstore.as_retriever()

# Step 3: Use Retriever
results = retriever.get_relevant_documents("Tell me about LangChain")

# Step 4: Output
for r in results:
    print(r.page_content)
```



Instead of `retriever.get_relevant_documents`, you can also use `retriever.invoke(query)`



`.as_retriever()` Explained

This is a **convenience method** that turns a Vector Store into a retriever by wrapping it with similarity search logic.

You can configure it:

```
retriever = vectorstore.as_retriever(search_type="similarity", search_kwargs=
{"k": 2})
```


Parameter	Purpose
<code>search_type</code>	"similarity" (default), "mmr" (max marginal relevance)
<code>search_kwargs</code>	For example, <code>{ "k": 3 }</code> to return top 3 results

✓ Summary: Why Use a Retriever When `similarity_search()` Exists?

- Using `.similarity_search()` is lower-level and manual.
- Using a retriever gives you a standardized interface that integrates cleanly into LangChain pipelines like RAG, Agents, Chains, etc.

⚙️ In Detail: Key Differences

Feature	<code>similarity_search()</code>	<code>as_retriever()</code> & <code>get_relevant_documents()</code>
✓ Low-level vector store access	Yes	No (wraps vectorstore internally)
✓ Simple search usage	Yes	Yes
🔌 Plug into LangChain Chains	✗ Not directly	✓ Required for LLM chains (e.g., <code>RetrievalQA</code>)
🔧 Advanced options (MMR, filters)	✗ Limited to similarity	✓ Supports filters, MMR, relevance tuning
🧱 Extendable	✗ Not easily	✓ You can subclass and create custom retrievers
🧠 Used in Agents, Tools, RAG	✗	✓ Standard component
✨ LangChain integration	✗ Manual	✓ Fully integrated into LangChain ecosystem

- Using `.similarity_search()` is like directly searching a database table with raw SQL.
- Using `.as_retriever()` is like using a smart, reusable API that works everywhere and can be swapped or enhanced later.

Similarity Search with score

- There are some FAISS specific methods. One of them is `similarity_search_with_score`, which allows you to return not only the documents but also the distance score of the query to them.
- The returned distance score is L2 distance. Therefore, a lower score is better.

```
docs_and_score=db.similarity_search_with_score(query)
docs_and_score
```

```
[ (Document(page_content='It is a distressing and oppressive duty, gentlemen of the Congress,
15552.862),
(Document(page_content='To such a task we can dedicate our lives and our fortunes, everything
15925.468),
(Document(page_content='...\n\nIt will be all the easier for us to conduct ourselves as bellig
16786.777),
(Document(page_content='We have borne with their present government through all these bitter
19336.307) ]
```

Pass vectors instead of sentences:

Embed Query

```
embedding_vector=embeddings.embed_query(query)
embedding_vector
```

```
[1.8624788522720337,  
-2.919260263442993,  
1.954888105392456,  
1.3011348247528076,  
-0.8985604047775269,  
0.6351550817489624,  
1.5126235485076904,  
-0.5887193083763123,  
0.8871709704399109,  
-1.8674403429031372,  
1.2215391397476196,  
-1.9168273210525513,  
0.5738440810707034]
```

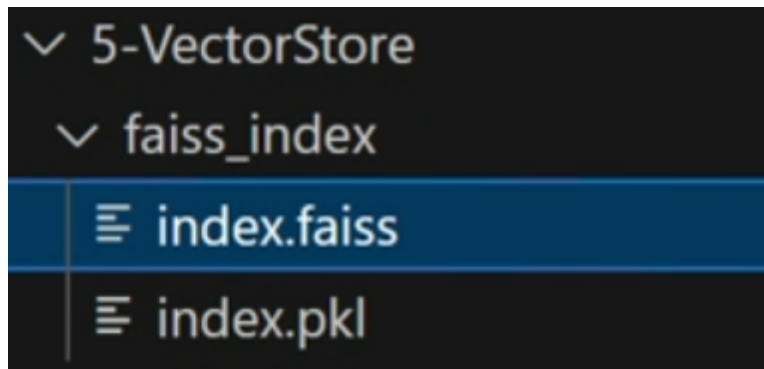
```
docs_score=db.similarity_search_by_vector(embedding_vector)  
docs_score
```

```
[Document(page_content='It is a distressing and oppressive duty, gentlemen of the C  
Document(page_content='To such a task we can dedicate our lives and our fortunes, o  
Document(page_content='...\n\nIt will be all the easier for us to conduct ourselves  
Document(page_content='We have borne with their present government through all the
```

✅ Create and Save FAISS:

```
db=FAISS.from_documents(docs,embeddings) #CREATING FAISS
```

```
### Saving And Loading  
db.save_local("faiss_index")
```



Load the saved model:

```
new_db=FAISS.load_local("faiss_index",embeddings,allow_dangerous_deserialization=True)
docs=new_db.similarity_search(query)
```

- **FAISS.load_local()** : This function loads a previously saved FAISS index from a specified file path (`"faiss_index"`).
- **embeddings** : This is used to specify the embedding model to use. When loading the FAISS index, LangChain needs the embedding model to interpret the vectors. You will pass the **same embedding model** that was used when creating the index (e.g., `OllamaEmbeddings`).
- **allow_dangerous_deserialization=True** : This flag is a safety mechanism to prevent deserialization of malicious or corrupted data. By enabling it, you are allowing the deserialization of the FAISS index even if it's potentially risky. You should only use this with trusted data to avoid potential vulnerabilities (e.g., if the index was tampered with).