Advanced RAG with Vector Databases and Retrievers

What is a LangChain retriever?

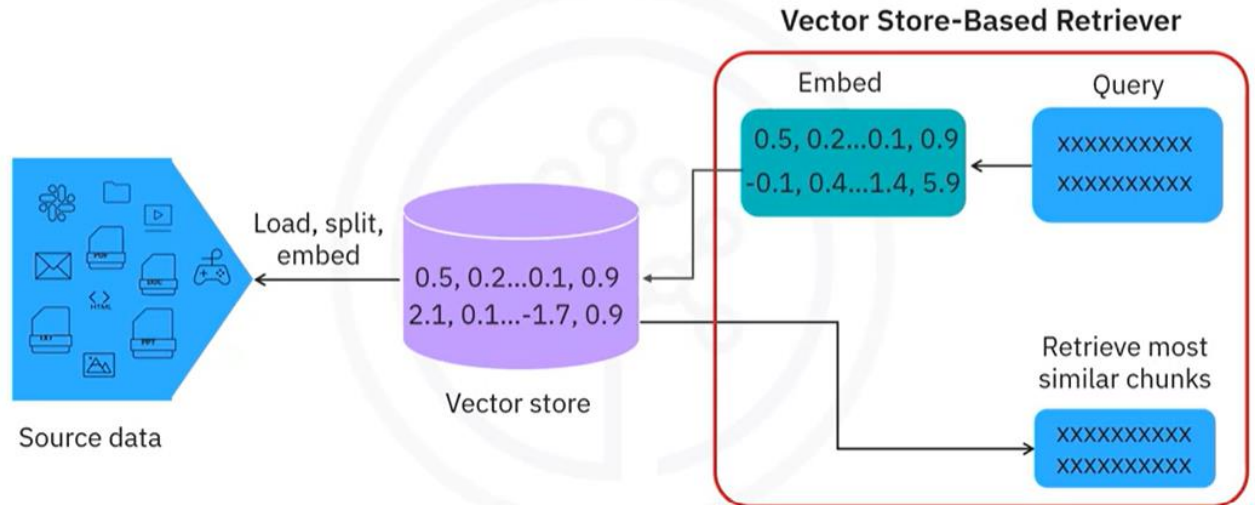
An interface that returns documents based on an unstructured query

More general than a vector store

Retrieves documents or their chunks

Accepts a string query as input and returns a list of documents or chunks as output

Vector Store-Based Retriever



Vector Store-Based Retriever

Simple to understand because it queries an existing vector store

Does not require an LLM to retrieve the most similar chunks

Vector Store-Based Retriever: Maximum marginal relevance (MMR) retrieval

Technique used to balance the relevance and diversity of retrieved results

Selects documents relevant to the query and minimally similar to previous documents

Avoids redundancy and ensures comprehensive coverage of query

Vector Store-Based Retriever:
Maximum marginal relevance (MMR) retrieval

```
retriever = vectordb.as_retriever(search_type="mmr")
docs = retriever.invoke("email policy")
```

```
[Document(metadata={'source': 'companypolicies.txt'}, page_content='to this policy. Non-compliance may lead to appropriate disciplinary action,
which could include'),
 Document(metadata={'source': 'companypolicies.txt'}, page_content='of our employees, customers, and the public. We diligently comply with all r
elevant health and'),
 Document(metadata={'source': 'companypolicies.txt'}, page_content="Data Privacy: We are committed to protecting the privacy of candidates' pers
onal information and"),
 Document(metadata={'source': 'companypolicies.txt'}, page_content='workplace safety measures.')]
```

Recap

- A LangChain retriever is an interface that returns documents based on an unstructured query
- Vector Store-Based Retriever retrieves documents from a vector database using similarity search or MMR
- Similarity search is when the retriever accepts a query and retrieves the most similar data
- MMR is a technique used to balance the relevance and diversity of retrieved results

Multi-Query Retriever

- A LangChain retriever returns documents based on unstructured query
- A Vector Store-Based Retriever retrieves documents from a vector database

Multi-Query Retriever

- Uses an LLM to create different versions of the query to generate a richer set of documents
- Overcomes differences in results due to changes in query wording or poor embeddings

Multi-Query Retriever

```
from ibm_watsonx_ai.foundation_models import ModelInference
from ibm_watsonx_ai.metanames import GenTextParamsMetaNames as GenParams
from ibm_watsonx_ai import Credentials
from ibm_watsonx_ai.foundation_models.extensions.langchain import WatsonxLLM

def llm():
    model_id = 'mistralai/mixtral-8x7b-instruct-v01'
    parameters = {
        GenParams.MAX_NEW_TOKENS: 256,  # this controls the maximum number of tokens in the generated output
        GenParams.TEMPERATURE: 0.5, # this randomness or creativity of the model's responses
    }
    credentials = {
        "url": "https://us-south.ml.cloud.ibm.com"
    }
    project_id = "skills-network"
    model = ModelInference(
        model_id=model_id,
        params=parameters,
        credentials=credentials,
        project_id=project_id
    )
    mixtral_llm = WatsonxLLM(model = model)
    return mixtral_llm
```

```
from langchain.retrievers.multi_query import MultiQueryRetriever

retriever = MultiQueryRetriever.from_llm(
    retriever=vectordb.as_retriever(), llm=llm()
)

docs = retriever.invoke("email policy")
```

Self-Query Retriever

- Converts the query into two components:
  - A string to look up semantically
  - Metadata filter to go along with it

Self-Query Retriever: Setup

```
vectordb = Chroma.from_documents(docs, watsonx_embedding)
```

```
metadata_field_info = [
    AttributeInfo(
        name="genre",
        description="The genre of the movie. One of ['science fiction', 'comedy', 'drama', 'thriller', 'romance', 'action', 'animated']",
        type="string",
    ),
    AttributeInfo(
        name="year",
        description="The year the movie was released",
        type="integer",
    ),
    AttributeInfo(
        name="director",
        description="The name of the movie director",
        type="string",
    ),
    AttributeInfo(
        name="rating", description="A 1-10 rating for the movie", type="float"
    ),
]
```

Self-Query Retriever: Usage

```
document_content_description = "Brief summary of a movie."

retriever = SelfQueryRetriever.from_llm(
    llm(),
    vectordb,
    document_content_description,
    metadata_field_info,
)

retriever.invoke("I want to watch a movie rated higher than 8.5")
```

```
[Document(metadata={'director': 'Andrei Tarkovsky', 'genre': 'thriller', 'rating': 9.9, 'year': 1979}, page_content='Three men walk into the Zon
e, three men walk out of the Zone'),
 Document(metadata={'director': 'Satoshi Kon', 'rating': 8.6, 'year': 2006}, page_content='A psychologist / detective gets lost in a series of d
reams within dreams within dreams and Inception reused the idea')]
```

Parent Document Retriever

- Splitting documents involves conflict between small documents for accuracy, and long for context
- Parent Document Retriever fetches small chunks, looks up their parent IDs, and returns large documents of the small chunks

Parent Document Retriever: Setup

```
from langchain.retrievers import ParentDocumentRetriever
from langchain_text_splitters import CharacterTextSplitter
from langchain.storage import InMemoryStore

# Set two splitters. One is with big chunk size (parent) and one is with small chunk size (child)
parent_splitter = CharacterTextSplitter(chunk_size=2000, chunk_overlap=20, separator='\n')
child_splitter = CharacterTextSplitter(chunk_size=400, chunk_overlap=20, separator='\n')
```

```
vectordb = Chroma(
    collection_name="split_parents", embedding_function=watsonx_embedding
)

# The storage layer for the parent documents
store = InMemoryStore()
```

```
retriever = ParentDocumentRetriever(
    vectorstore=vectordb,
    docstore=store,
    child_splitter=child_splitter,
    parent_splitter=parent_splitter,
)
retriever.add_documents(data)
```

Parent Document Retriever: Usage

```
retriever.invoke("smoking policy")
```

```
[Document(metadata={'source': 'companypolicies.txt'}, page_content='Smoking Restrictions: Smoking inside company buildings, offices, meeting roo
ms, and other enclosed spaces is strictly prohibited. This includes electronic cigarettes and vaping devices.\nCompliance with Applicable Laws:
All employees and visitors must adhere to relevant federal, state, and local smoking laws and regulations.\nDisposal of Smoking Materials: Prope
rly dispose of cigarette butts and related materials in designated receptacles. Littering on company premises is prohibited.\nNo Smoking in Comp
any Vehicles: Smoking is not permitted in company vehicles, whether they are owned or leased, to maintain the condition and cleanliness of these
vehicles.\nEnforcement and Consequences: All employees and visitors are expected to adhere to this policy. Non-compliance may lead to appropriat
e disciplinary action, which could include fines, or, in the case of employees, possible termination of employment.\nReview of Policy: This poli
cy will be reviewed periodically to ensure its alignment with evolving legal requirements and best practices for maintaining a healthy and safe
workplace.\nWe appreciate your cooperation in maintaining a smoke-free and safe environment for all.\n6.\tDrug and Alcohol Policy\nPolicy Object
ive: The Drug and Alcohol Policy is established to establish clear expectations and guidelines for the responsible use of drugs and alcohol with
in the organization. This policy aims to maintain a safe, healthy, and productive workplace.\nProhibited Substances: The use, possession, distri
bution, or sale of illegal drugs or unauthorized controlled substances is strictly prohibited on company premises or during work-related activit
ies. This includes the misuse of prescription drugs.\nAlcohol Consumption: The consumption of alcoholic beverages is not allowed during work hou
rs, on company property, or while performing company-related duties. Exception may be made for company-sanctioned events.')]
```

Recap

- The Multi-Query Retriever uses an LLM to create different versions of the query to generate richer documents
- The Self-Query Retriever converts the query into a string and a metadata filter
- The Parent Document Retriever has a parent splitter to split text into large chunks and a child splitter for small chunks

Index types in LlamaIndex

Three core index types:

VectorStoreIndex
Semantic search based on meaning

DocumentSummaryIndex
Generated summaries to identify relevant documents

KeywordTableIndex
Exact keyword matching for rule-based or hybrid search

The VectorStoreIndex

The DocumentSummaryIndex

Generates and stores summaries of documents

Filters documents before full retrieval

Useful for large, diverse document sets

The KeywordTableIndex

Extracts keywords from documents

Enables exact keyword matching

Maps keywords to specific chunks of content

Useful for hybrid or rule-based search

Core and advanced retrievers – Vector Index Retriever

- Uses embeddings to find relevant content
- Ideal for general-purpose search
- Common in retrieval-augmented generation (RAG)

Core and advanced retrievers – Understanding TF-IDF

TF-IDF

TERM FREQUENCY-INVERSE DOCUMENT FREQUENCY

Term Frequency (TF): Word frequency in a document	Inverse Document Frequency (IDF): Rarity across documents	TF-IDF = TF x IDF

Core and advanced retrievers – BM25 Retriever

- Keyword-based retrieval for ranking docs
- Retrieves content based on exact keyword matches (not semantic similarity)

- Improves on TF-IDF
  - Adds term frequency saturation
  - Normalizes for document length

Core and advanced retrievers – Document Summary Index Retriever

Uses summaries to filter documents

Returns original documents, not their summaries

Two versions:

- LLM-based
  - Time-consuming
  - Expensive
- Embedding-based
  - Uses semantic similarity
  - Efficient for large collections

Core and advanced retrievers – Auto Merging Retriever

Preserves context in long docs using a hierarchical structure

Retrieves parent node if enough child nodes match

Uses hierarchical chunking to break docs into parent and child nodes

Consolidates related content and preserves broader context

Core and advanced retrievers – Recursive Retriever

Follows node relationships using references:

- Citations
- Metadata links

Supports chunk and metadata linking references

Retrieves related content across documents or abstraction layers

Core and advanced retrievers – Query Fusion Retriever

Combines results from multiple different retrievers

Supports multiple query variations

Uses fusion strategies to improve recall

Fusion strategies supported by the Query Fusion Retriever

Relative Score Fusion

Normalizes scores within each result set

Reciprocal Rank Fusion

Combines rankings across queries

Distribution-Based Fusion

Uses statistical normalization

Recommended retrievers by use case

Use Case	Primary Retriever	Secondary/Hybrid
General Q&A	Vector Index Retriever	+ BM25
Technical docs	BM25 Retriever	+ Vector
Long documents	Auto Merging Retriever	-
Research papers	Recursive Retriever	-
Large document sets	Document Summary Index Retriever	+ Vector

Recap

- The core LlamaIndex index types are the VectorStoreIndex, the DocumentSummaryIndex, and the KeywordTableIndex
- The VectorStoreIndex stores vector embeddings for each document chunk, is best suited for semantic retrieval, and is commonly used in pipelines that involve large language models
- The DocumentSummaryIndex generates and stores summaries of documents, which are used to filter documents before retrieving the full content, and is useful when working with large and diverse document sets
- The KeywordTableIndex extracts keywords from documents and maps them to specific content chunks, and is useful in hybrid or rule-based search scenarios

Recap

- The Vector Index Retriever uses vector embeddings to find semantically related content, and is ideal for general-purpose search and RAG pipelines
- The BM25 Retriever is a keyword-based method for ranking documents, and it retrieves content based on exact keyword matches rather than semantic similarity
- The Document Summary Index Retriever uses document summaries instead of the actual documents to find relevant content
- There are two versions of the Document Summary Index Retriever, one uses LLM, and the other uses semantic similarity

Recap

- The Auto Merging Retriever preserves context in long documents using a hierarchical structure, and uses hierarchical chunking to break documents into parent and child nodes
- The Recursive Retriever follows relationships between nodes using references such as citations in academic papers or metadata links
- The Query Fusion Retriever combines results from different retrievers using fusion strategies
- The fusion strategies supported by the Query Fusion Retriever are Reciprocal Rank Fusion, Relative Score Fusion, and Distribution-Based Fusion

What is FAISS?

- Library created by Meta for fast vector search
- Runs on a single machine (CPU or GPU)
- You write code to use it - no built-in database storage or server
- Great for custom, high-performance systems

What is Chroma DB?

- A vector database designed for AI applications
- Stores vectors and metadata together
- Can run locally or as a server
- Easy to use with tools such as LangChain

FAISS vs Chroma DB: Key differences

Feature	FAISS	Chroma DB
Type	Library	Database
Deployment	Local, single node	Local, single node, or distributed
Indexing options	Flat, IVF, LSH, HNSW	HNSW only
Metadata support	None	Storage, filtering
LangChain/ LlamaIndex support	Yes	Yes

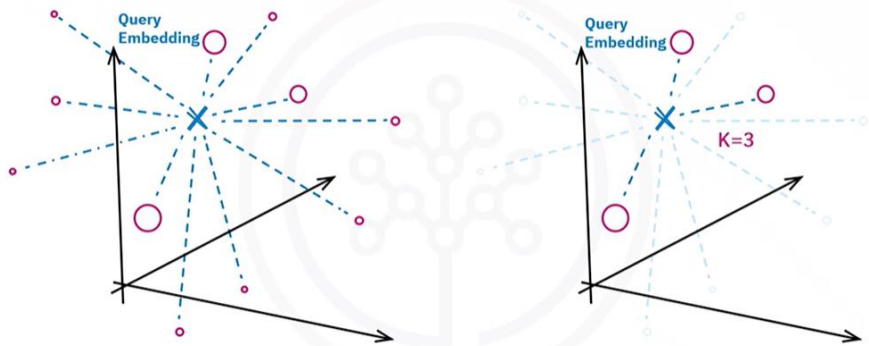
What is an Index?

FAISS offers several index types

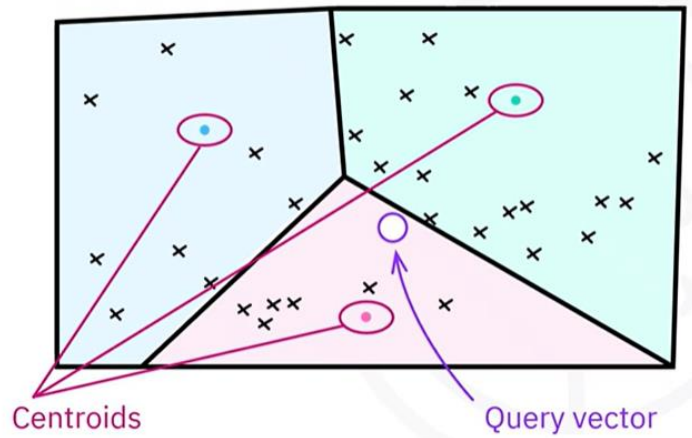
Helps you search vectors faster

Each index balances speed, memory, and accuracy

Flat Index (IndexFlat)



Inverted File Index (IVF)



- Speeds up vector search
  - Clustering vectors using methods like k-means
  - Forming Voronoi cells around centroids
- Each cell contains vectors closest to its centroid
- When a query vector is introduced, the search is limited to vectors in the nearest cell
- Faster than a flat index
  - May slightly reduce accuracy

Locality-Sensitive Hashing (LSH)

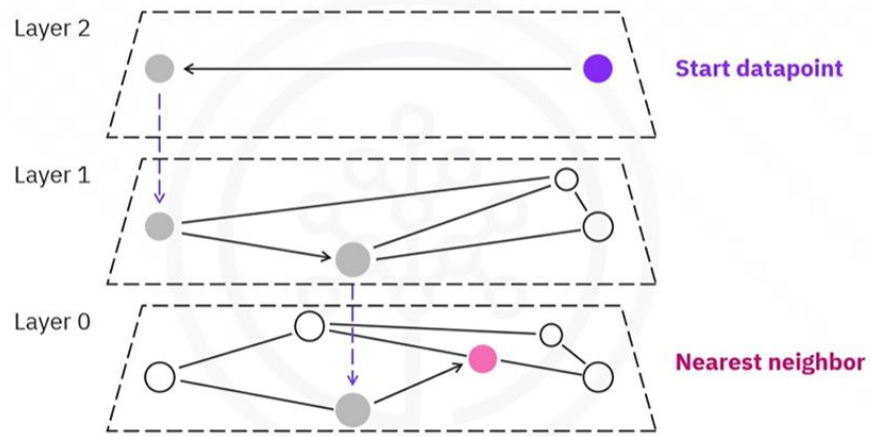
Uses hash functions to group similar vectors

Best for high-dimensional, sparse data

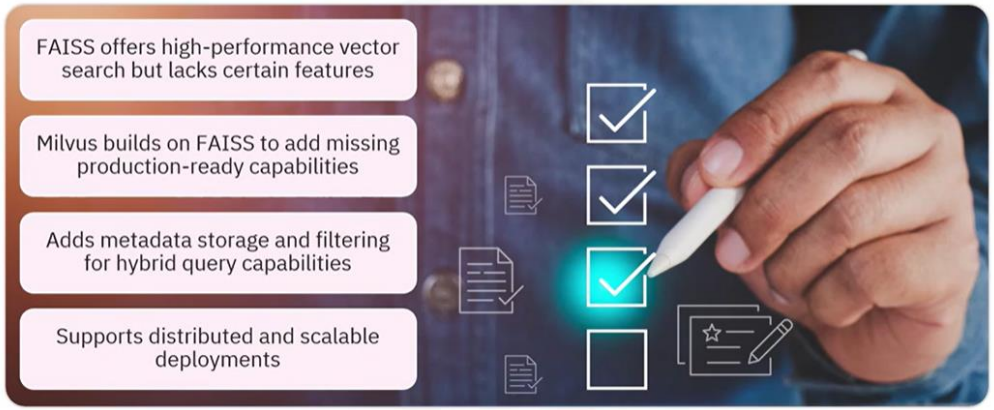
Fast and memory-efficient

Not the fastest or most accurate method

Hierarchical Navigable Small World (HNSW)



Extending FAISS with Milvus



FAISS, Chroma DB, and Milvus: When to use which?

- **Use FAISS for:**
  - Custom, high-performance systems
  - GPU acceleration
  - Local-only deployments
- **Use Chroma DB for:**
  - Fast prototyping
  - Metadata-rich queries
- **Use Milvus for:**
  - Distributed, production-scale systems
  - FAISS-style indexing with database features
  - Hybrid search (vector + metadata)

Recap

- FAISS and Chroma DB serve different needs
- FAISS gives you full control over indexing algorithms, but doesn't support metadata or distributed search on its own
- Chroma DB offers easy setup and metadata support, but fewer indexing options
- Extend FAISS with Milvus to add metadata support and distributed capabilities
- Extend FAISS and Chroma DB with LangChain and LlamaIndex for RAG pipelines
- Choose the best tool for your project's scale, complexity, and infrastructure