MongoDB.

MongoDB Developer

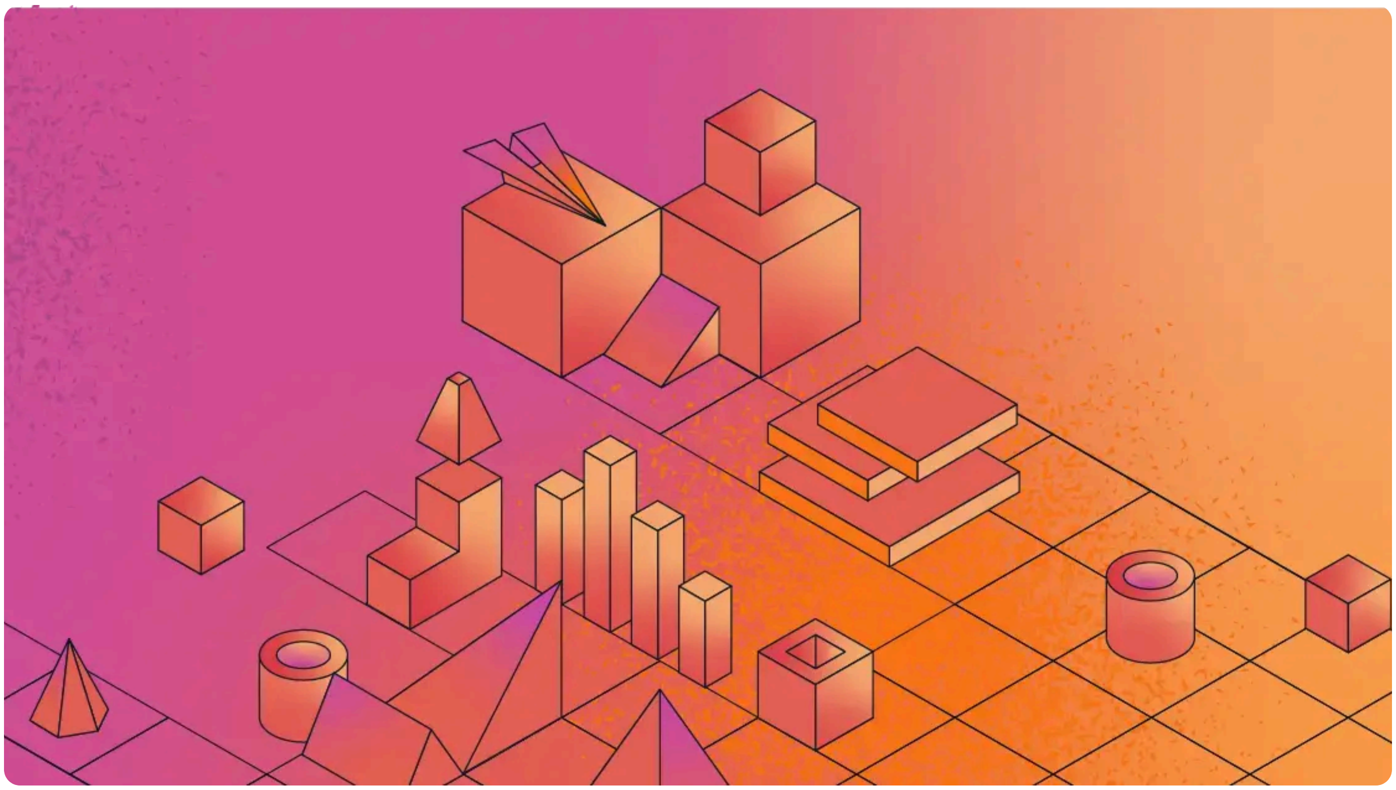
MONGODB DEVELOPER CENTER > DEVELOPER TOPICS > PRODUCTS > ATLAS > TUTORIALS

How to Choose the Right Chunking Strategy for Your LLM Application

Apoorva Joshi

16 min read • Published Jun 17, 2024 • Updated Jun 17, 2024

AI Python Atlas



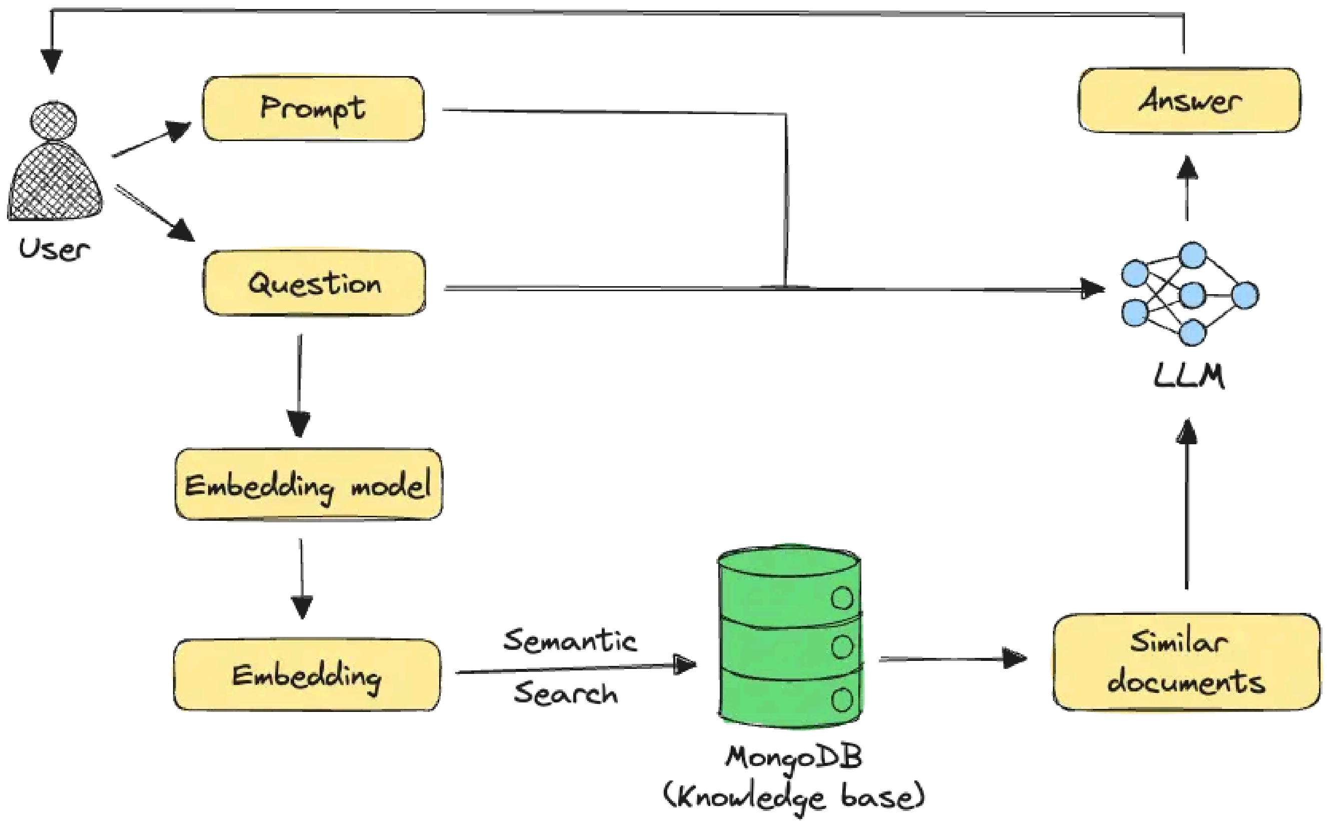
Rate this tutorial ☆ ☆ ☆ ☆ ☆

In Part 1 of this series on Retrieval Augmented Generation (RAG), we looked into choosing the right embedding model for your RAG application. While the choice of embedding model is an important consideration to ensure good quality retrieval for RAG, there is one key decision to be made before the embedding stage that can have a significant downstream impact – choosing the right chunking strategy for your data.

In this tutorial, we will cover the following:

- What is chunking and why is it important for RAG?
- Choosing the right chunking strategy for your RAG application
- Evaluating different chunking methodologies on a dataset

RAG — a very quick refresher



In a RAG application, the goal is to get more accurate responses from a large language model (LLM) on subjects that might not be well-represented in its parametric knowledge – for example, your organization's data. This is typically done by retrieving relevant information from a knowledge base using semantic search. This technique uses an embedding model to create vector representations of the user query and information in the knowledge base.

Given a user query and its embedding, it retrieves the most relevant documents from the knowledge base using vector similarity search. The retrieved documents, user query, and any user prompts are then passed as context to an LLM, to generate an answer to the user's question.

What is chunking and why is it important for RAG?

Most use cases for RAG currently are centered around using LLMs to answer questions about large repositories of data such as technical documentation, onboarding documents, etc. While LLM model providers are constantly increasing the context windows for their models, they also charge by the number of input tokens. This means trying to fit large documents into the model's context window can be expensive.

Passing large documents as input to the LLM also means that the LLM has to parse out the most relevant information from the document to answer user queries. This can be challenging given that LLMs are designed to process input sequentially, token by token. While they can capture long-range dependencies and context to some extent, their ability to do so effectively diminishes as the input sequence becomes longer. This is where chunking helps.

Chunking is the process of breaking down large pieces of text into smaller segments or chunks. In the context of RAG, embedding smaller chunks instead of entire documents to create the knowledge base means that given a user query, you only have to retrieve the most relevant document chunks, resulting in fewer input tokens and more targeted context for the LLM to work with.

Choosing the right chunking strategy for your RAG application

There is no “one size fits all” solution when it comes to choosing a chunking strategy for RAG – it depends on the structure of the documents being used to create the knowledge base and will look different depending on whether you are working with well-formatted text documents or documents with code snippets, tables, images, etc. The three key components of a chunking strategy are as follows:

- Splitting technique: Determines where the chunk boundaries will be placed – based on paragraph boundaries, programming language-specific separators, tokens, or even semantic boundaries
- Chunk size: The maximum number of characters or tokens allowed for each chunk
- Chunk overlap: Number of overlapping characters or tokens between chunks; overlapping chunks can help preserve cross-chunk context; the degree of overlap is typically specified as a percentage of the chunk size

As with choosing embedding or completion models for RAG, we recommend picking a few different options for each component; evaluating them on a small, manually curated dataset of anticipated user queries for your application; and choosing the one that gives you the best performance in terms of retrieval precision and recall on this evaluation dataset. Alternatively, you can start by creating an LLM-generated synthetic dataset based on your best guess of how users will use your application.

In this tutorial, we will evaluate different combinations of chunk sizes, overlaps, and splitting techniques available in LangChain. We choose LangChain here for the sake of continuity from Part 2 of this tutorial series. Most of these techniques are also supported in LlamaIndex, which is another popular framework for building RAG applications.

> LangChain and LlamaIndex support various text-splitting techniques but text splitters may not work well if you are working with documents containing tables, images, etc. Unstructured is a library that supports chunking for documents containing semi-structured, multimodal, purely tabular data, etc.

Before we begin

Data

For this tutorial, let’s assume we are building a chat application for new Python developers. We will use Python Enhancement Proposals (PEP) to construct the knowledge base for the RAG system. PEPs are documents that provide information about Python coding conventions and best practices to the Python community.

Tools

We will use LangChain for text splitting and for creating components of a RAG application as needed. We will use the Ragas framework which we introduced in Part 2 of this tutorial series to evaluate the best chunking strategy for our data.

Where’s the code?

The Jupyter Notebook for this tutorial can be found on GitHub.

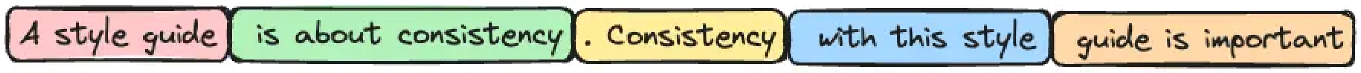
Chunking strategies

Looking at the PEP documents, we note that they contain mostly text and some Python code snippets. With this in mind, we will test out the following chunking strategies:

Fixed token without overlap

In this technique, we split the documents into chunks with a fixed number of tokens, with no token overlap between chunks. This can work well if there are hard contextual boundaries between chunks – i.e., the context varies drastically between adjacent chunks. In reality, this is rarely ever the case, but we will keep this as a baseline.

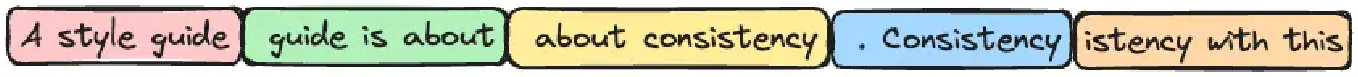
A visual representation of this strategy is as follows:



Fixed token with overlap

In this technique, we split documents into chunks with a fixed number of tokens, with some token overlap between chunks. Chunk overlap ensures that contextual information at the boundaries of chunks is not lost during chunking, thus improving the chances of the right information being retrieved during semantic search even if it spans across multiple chunks.

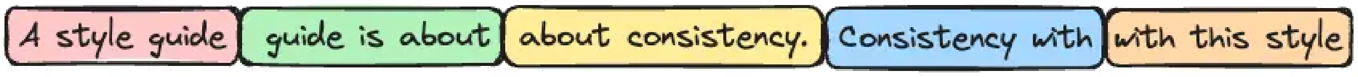
A visual representation of this strategy is as follows:



Recursive with overlap

In this technique, we first split the documents by a parameterized list of characters such as `\n\n`, `\n`, etc., and then recursively merge characters into tokens using a tokenizer as long as the chunk size (in terms of the number of tokens) is less than the specified chunk size. This has the effect of trying to keep all paragraphs (and then sentences, and then words) together as long as possible, as those would generically seem to be the strongest semantically related pieces of text.

A visual representation of this strategy is as follows:



Recursive Python splitter with overlap

Since the PEPs contain some Python code snippets, we will also include a Python-specific chunking technique in our evaluation. This technique is the same as recursive with overlap, except the character list for splitting also includes Python-specific separators such as `\nclass`, `\ndef`, etc.

Semantic

In this technique, documents are split based on semantic similarity. Documents are first split into sentence groups of three sentences using a sliding window. Embeddings are generated for each sentence group, and similar groups in the embedding space are merged to form chunks. The similarity threshold for merging is determined using metrics such as percentile, standard deviation, and interquartile distance. As a result, the chunk size can vary across chunks.

While this method is more computationally expensive than the ones above, it can be useful for chunking documents where the contextual boundaries are not obvious – for example, essays that might not have headings to indicate logical contextual breaks.

A visual representation of this strategy is as follows:

A style guide is about consistency. Consistency with this style guide is important.

Step 1: Install required libraries

We will require the following libraries for this tutorial:

- langchain: Python library to develop LLM applications using LangChain
- langchain-openai: Python package to use OpenAI models in LangChain
- langchain-mongodb: Python package to use MongoDB Atlas as a vector store with LangChain
- langchain-experimental: Python package for LangChain's experimental features, such as semantic chunking
- ragas: Python library for the Ragas framework
- pymongo: Python driver for interacting with MongoDB
- tqdm: Python module to show a progress meter for loops

```
! pip install -qU langchain langchain-openai langchain-mongodb langchain-experi
```

Step 2: Set up pre-requisites

In this tutorial, we will use MongoDB Atlas to create the knowledge base (vector store) for our RAG application. But first, you will need a MongoDB Atlas account with a database cluster. You'll also need to get the connection string to connect to your cluster. Follow these steps to get set up:

- Register for a free MongoDB Atlas account.
- Follow the instructions to create a new database cluster.
- Follow the instructions to obtain the connection string for your database cluster.

> Don't forget to add the IP of your host machine to the IP access list for your cluster.

```
import getpass
MONGODB_URI = getpass.getpass("Enter your MongoDB connection string:")
```

We will be using OpenAI's embedding and chat completion models, so you'll also need to obtain an OpenAI API key and set it as an environment variable for the OpenAI client to use:

```
import os
from openai import OpenAI
os.environ["OPENAI_API_KEY"] = getpass.getpass("Enter your OpenAI API Key:")
openai_client = OpenAI()
```

Step 3: Load the dataset

As mentioned previously, we will use PEPs to build the knowledge base for our RAG application. We will use the `WebBaseLoader` document loader from LangChain which loads all the text from HTML webpages into a more readable document format by stripping out HTML tags etc.

```
from langchain_community.document_loaders import WebBaseLoader

web_loader = WebBaseLoader(
    [
        "https://peps.python.org/pep-0483/",
        "https://peps.python.org/pep-0008/",
        "https://peps.python.org/pep-0257/",
    ]
)

pages = web_loader.load()
```

The `load` method of the document loaders in LangChain creates a list of LangChain document objects (we will refer to these as "documents" in this tutorial), each consisting of two attributes – namely, `page_content` and `metadata`. `page_content`, as the name suggests, corresponds to the content of the document, and `metadata` is some basic metadata extracted by the document loader, such as source, title, description, language, etc.

An example of a document object is as follows:

```
Document(
    page_content="\nThe goal of this PEP is to propose such a systematic way of
    metadata= {'source': 'https://peps.python.org/pep-0483/', 'title': 'PEP 483
)
```

If you are using LlamaIndex, use the SimpleWebPageReader class to load text from HTML web pages.

Step 4: Define chunking functions

Next, let's define functions to support each of the chunking strategies we want to experiment with.

```
from langchain.text_splitter import TokenTextSplitter

def fixed_token_split(
    docs: List[Document], chunk_size: int, chunk_overlap: int
) -> List[Document]:
    """
    Fixed token chunking

    Args:
        docs (List[Document]): List of documents to chunk
        chunk_size (int): Chunk size (number of tokens)
        chunk_overlap (int): Token overlap between chunks

    Returns:
        List[Document]: List of chunked documents
    """
    splitter = TokenTextSplitter(
        encoding_name="cl100k_base", chunk_size=chunk_size, chunk_overlap=chun
    )
    return splitter.split_documents(docs)
```

In the above code, the `fixed_token_split` function uses the `TokenTextSplitter` class in LangChain to split documents into chunks, each consisting of a fixed number of tokens. The `TokenTextSplitter` class takes the following arguments:

- chunk_size: Number of tokens in each chunk
- chunk_overlap: Number of overlapping tokens between adjacent chunks
- encoding_name: The model to use for generating tokens

We will use this function to create chunks with and without overlap. To create chunks without overlap, we will set the `chunk_overlap` argument to 0 when calling the function. To create chunks with overlap, we will set it to a non-zero value.

> If you are using LlamaIndex, use the TokenTextSplitter class for fixed token chunking.

Next, let's create a function for recursive chunking:

```
from langchain.text_splitter import (
    Language,
    RecursiveCharacterTextSplitter,
)

def recursive_split(
    docs: List[Document],
    chunk_size: int,
    chunk_overlap: int,
    language: Optional[Language] = None,
) -> List[Document]:
    """
    Recursive chunking

    Args:
        docs (List[Document]): List of documents to chunk
        chunk_size (int): Chunk size (number of tokens)
        chunk_overlap (int): Token overlap between chunks
        language (Optional[Language], optional): Programming language enum. De

    Returns:
        List[Document]: List of chunked documents
    """
    separators = ["\n\n", "\n", " ", ""]

    if language is not None:
        try:
            separators = RecursiveCharacterTextSplitter.get_separators_for_lan
                language
            )
        except (NameError, ValueError) as e:
            print(f"No separators found for language {language}. Using default

    splitter = RecursiveCharacterTextSplitter.from_tiktoken_encoder(
        encoding_name="cl100k_base",
        chunk_size=chunk_size,
        chunk_overlap=chunk_overlap,
        separators=separators,
    )
    return splitter.split_documents(docs)
```

In the above code, the `recursive_split` function uses the `tiktoken_encoder` method of the `RecursiveCharacterTextSplitter` class in LangChain to first split documents by a list of characters and then recursively merge splits into tokens using the `tiktoken` tokenizer.

In addition to `chunk_size` and `chunk_overlap`, the function takes an additional `language` parameter to support splitting based on language-specific separators for programming languages supported in the Language enum in LangChain. If no language or an unsupported language is specified, the default list of separators – i.e., `["\n\n", "\n", " ", ""]` – is used for splitting. Otherwise, a language-specific list is used. This list is obtained using the `get_separators_for_language` method of the `RecursiveCharacterTextSplitter` class. For Python, we will set the language parameter to `Language.PYTHON` in the function call later on.

If you are using LlamaIndex, use the SentenceSplitter class for recursive chunking and for splitting based on programming language-specific separators.

Finally, let's define a function for semantic chunking:

```python
from langchain_experimental.text_splitter import SemanticChunker
from langchain_openai.embeddings import OpenAIEmbeddings

def semantic_split(docs: List[Document]) -> List[Document]:
    """
    Semantic chunking

    Args:
        docs (List[Document]): List of documents to chunk

    Returns:
        List[Document]: List of chunked documents
    """
    splitter = SemanticChunker(
        OpenAIEmbeddings(), breakpoint_threshold_type="percentile"
    )
    return splitter.split_documents(docs)
```

In the above code, we define the `semantic_split` function that uses the `SemanticChunker` class in LangChain to split documents based on semantic similarity. We use an OpenAI embedding model (`text-embedding-ada-002` by default) to embed sentence groups and set the `breakpoint_threshold_type` parameter to `percentile`. In this method, the embedding distances between adjacent sentence groups are calculated, and splits are created at points where the embedding distance is greater than the 95th percentile of distances. Other options for `breakpoint_threshold_type` include `standard_deviation` and `interquartile`.

> If you are using LlamaIndex, use the SemanticSplitterNodeParser class for semantic chunking.

Step 5: Generate the evaluation dataset

As mentioned before, you can either use a manually curated dataset of anticipated user queries for evaluation or start by using an LLM-generated dataset. Let’s explore the latter using the synthetic data generation capabilities in the ragas library.

```
from ragas import RunConfig
from ragas.testset.generator import TestsetGenerator
from ragas.testset.evolutions import simple, reasoning, multi_context
from langchain_openai import ChatOpenAI, OpenAIEmbeddings

RUN_CONFIG = RunConfig(max_workers=4, max_wait=180)

# generator with openai models
generator_llm = ChatOpenAI(model="gpt-3.5-turbo-16k")
critic_llm = ChatOpenAI(model="gpt-4")
embeddings = OpenAIEmbeddings()

generator = TestsetGenerator.from_langchain(generator_llm, critic_llm, embeddi

# Change resulting question type distribution
distributions = {simple: 0.5, multi_context: 0.4, reasoning: 0.1}

testset = generator.generate_with_langchain_docs(
    pages, 10, distributions, run_config=RUN_CONFIG
)
```

The above code:

- Creates a runtime config for Ragas to override its default concurrency and retry settings – we do this to avoid running into OpenAI’s rate limits, but this might be a non-issue depending on your usage tier, or if you are not using OpenAI models for test set generation.
- Specifies the `generator_llm` – i.e., the model to use to generate the questions, answers, and ground truth for evaluation.
- Specifies the `critic_llm` which is the model to use to validate the data generated by the `generator_llm`.
- Specifies `embeddings` – i.e., the embedding model to build the vector store for the test set generator.

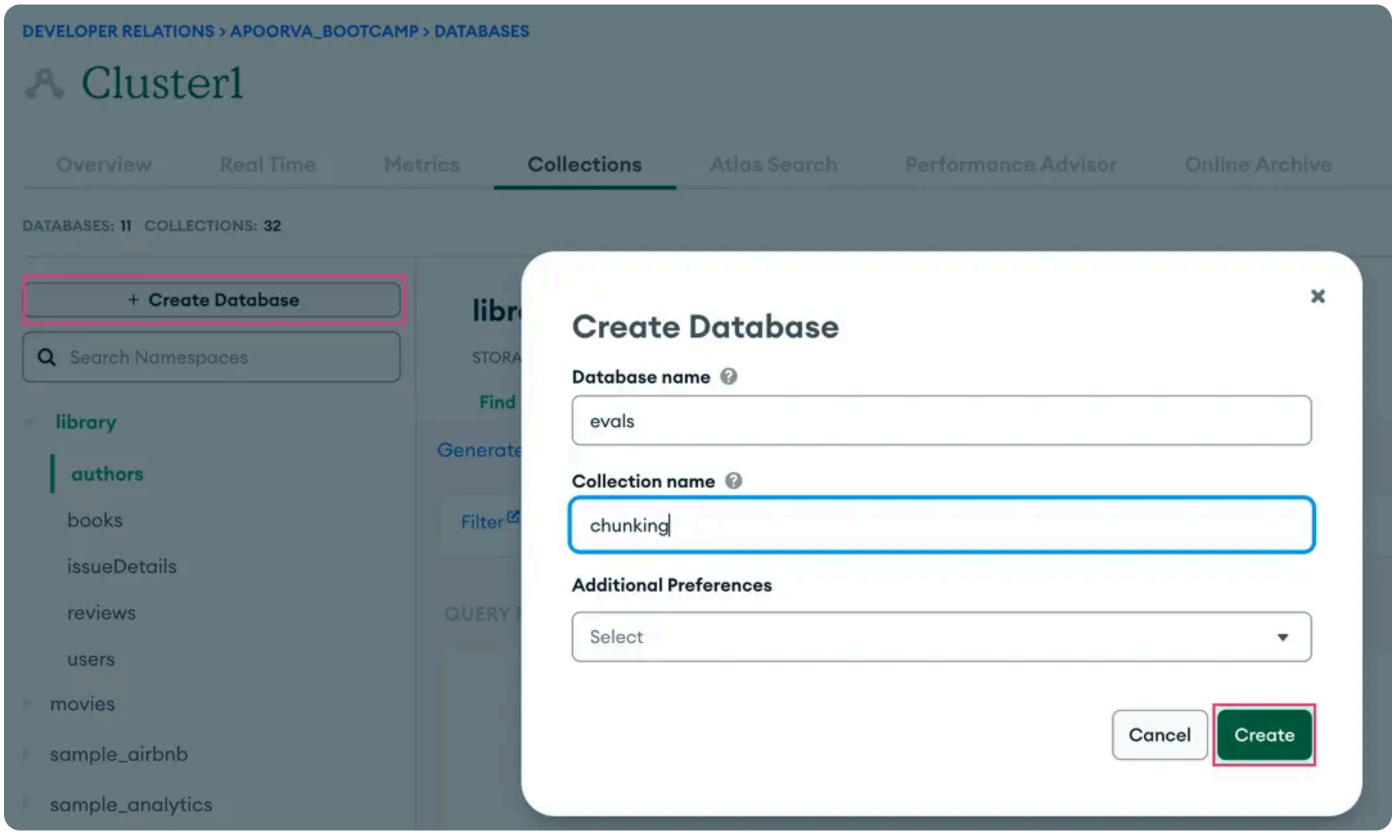
- Creates a test set generator using the specified models. We use the `from_langchain` method of the `TestsetGenerator` class since we are using OpenAI models through LangChain.
- Specifies the distribution of the types of questions to include in the test set: `simple` corresponds to straightforward questions that can be easily answered using the source data. `multi_context` stands for questions that would require information from multiple related sections or chunks to formulate an answer. `reasoning` questions are those that would require an LLM to reason to effectively answer the question. You will want to set this distribution based on your best guess of the type of questions you would expect from your users.
- Generates a test set from the dataset (`pages`) we created in Step 3. We use the `generate_with_langchain_docs` method since our dataset is a list of LangChain documents.

Ragas also supports test set generation using LlamaIndex. Refer to their documentation for more information.

Step 6: Evaluate chunking strategies

Now that we have defined our chunking functions and created our evaluation dataset, we are ready to evaluate the different chunking strategies to find the one that gives us the best retrieval quality on our evaluation dataset.

Let's start by creating a MongoDB collection with a vector store index for our evaluation. To do this, navigate to the Collections tab of the cluster you created in Step 2. Click on Create Database to create a database called `evals` with a collection named `chunking`.



Create Database

Database name: evals

Collection name: chunking

Additional Preferences: Select

Once that is done, create a vector index named `vector_index` on the `chunking` collection with the following index definition:

```
{
  "fields": [
    {
      "numDimensions": 1536,
      "path": "embedding",
      "similarity": "cosine",
      "type": "vector"
    }
  ]
}
```

> The number of embedding dimensions in the index definition is 1536 since we will be using OpenAI's text-embedding-3-small model to generate embeddings for our vector store.

Now, let's define a function to create a MongoDB Atlas vector store using the collection we created:

```python
from langchain_mongodb import MongoDBAtlasVectorSearch
from pymongo import MongoClient

client = MongoClient(MONGODB_URI)
DB_NAME = "evals"
COLLECTION_NAME = "chunking"
ATLAS_VECTOR_SEARCH_INDEX_NAME = "vector_index"
MONGODB_COLLECTION = client[DB_NAME][COLLECTION_NAME]

def create_vector_store(docs: List[Document]) -> MongoDBAtlasVectorSearch:
    """
    Create MongoDB Atlas vector store

    Args:
        docs (List[Document]): List of documents to create the vector store

    Returns:
        MongoDBAtlasVectorSearch: MongoDB Atlas vector store
    """
    vector_store = MongoDBAtlasVectorSearch.from_documents(
        documents=docs,
        embedding=OpenAIEmbeddings(model="text-embedding-3-small"),
        collection=MONGODB_COLLECTION,
        index_name=ATLAS_VECTOR_SEARCH_INDEX_NAME,
    )

    return vector_store
```

The above code:

- Creates a PyMongo client (`client`) to connect to our MongoDB Atlas cluster.
- Specifies the database (`DB_NAME`) and collection (`MONGODB_COLLECTION`) to connect to.
- Specifies the vector search index (`` ATLAS_VECTOR_SEARCH_INDEX_NAME`` ``) to use for vector search.
- Defines a `create_vector_store` function that takes a list of documents and returns a MongoDB Atlas vector store object. It uses the `from_documents` method of the `MongoDBAtlasVectorSearch` class

from the `langchain-mongodb` integration to create a MongoDB Atlas vector store directly using LangChain documents.

Next, let's define a function to evaluate different chunking strategies on the test dataset we created in Step 5:

```python
from tqdm import tqdm
from datasets import Dataset
from ragas import evaluate
from ragas.metrics import context_precision, context_recall
import nest_asyncio

# Allow nested use of asyncio (used by Ragas)
nest_asyncio.apply()

# Disable tqdm locks
tqdm.get_lock().locks = []

QUESTIONS = testset.question.to_list()
GROUND_TRUTH = testset.ground_truth.to_list()

def perform_eval(docs: List[Document]) -> Dict[str, float]:
    """
    Perform Ragas evaluation

    Args:
        docs (List[Document]): List of documents to create the vector store

    Returns:
        Dict[str, float]: Dictionary of evaluation metrics
    """
    eval_data = {
        "question": QUESTIONS,
        "ground_truth": GROUND_TRUTH,
        "contexts": [],
    }

    print(f"Deleting existing documents in the collection {DB_NAME}.{COLLECTIO
    MONGODB_COLLECTION.delete_many({})
    print(f"Deletion complete")
    vector_store = create_vector_store(docs)

    # Getting relevant documents for the evaluation dataset
    print(f"Getting contexts for evaluation set")
    for question in tqdm(QUESTIONS):
        eval_data["contexts"].append(
            [doc.page_content for doc in retriever.similarity_search(question,
        )
    # RAGAS expects a Dataset object
    dataset = Dataset.from_dict(eval_data)

    print(f"Running evals")
    result = evaluate(
        dataset=dataset,
        metrics=[context_precision, context_recall],
        run_config=RUN_CONFIG,
        raise_exceptions=False,
    )
    return result
```

The above code defines a `perform_eval` function that takes a list of chunked documents (`docs`) produced using a particular chunking strategy as input and returns a dictionary consisting of retrieval quality metrics for that strategy, on our evaluation dataset. A breakdown of what the function does is as follows:

- Creates a dictionary (`eval_data`) with `question`, `ground_truth`, and `contexts` as keys, corresponding to the questions in the evaluation dataset, their ground truth answers, and retrieved contexts
- Deletes any existing documents from the MongoDB collection we are using for testing
- Creates a MongoDB Atlas vector store using the current list of chunked documents
- Uses the `similarity_search` method to retrieve the top three most relevant chunks from the vector store for each question in the evaluation dataset and adds them to the `contexts` list in the `eval_data` dictionary
- Converts the `eval_data` dictionary to a dataset object
- Uses the `evaluate` method from the ragas library to calculate the `context_precision` and `context_recall` metrics for the evaluation dataset

Context precision evaluates the ability of the retriever to rank retrieved items in order of relevance to the ground truth answer. Context recall measures the extent to which the retrieved context aligns with the ground truth answer.

Finally, let's run the evaluation for the five chunking strategies that we short-listed for our PEP dataset:

```
for chunk_size in [100, 200, 500, 1000]:
    chunk_overlap = int(0.15 * chunk_size)
    print(f"CHUNK SIZE: {chunk_size}")
    print("------ Fixed token without overlap ------")
    print(f"Result: {perform_eval(fixed_token_split(pages, chunk_size, 0))}")
    print("------ Fixed token with overlap ------")
    print(
        f"Result: {perform_eval(fixed_token_split(pages, chunk_size, chunk_ove
    )
    print("------ Recursive with overlap ------")
    print(f"Result: {perform_eval(recursive_split(pages, chunk_size, chunk_ove
    print("------ Recursive Python splitter with overlap ------")
    print(
        f"Result: {perform_eval(recursive_split(pages, chunk_size, chunk_overl
    )
print("------ Semantic chunking ------")
print(f"Result: {perform_eval(semantic_split(pages))}")
```

A few things to note about the code above:

- We are evaluating different chunk sizes to find the most optimal chunk size for each chunking strategy.
- For chunking strategies with token overlap, we set the overlap to 15% of the chunk size. While we have kept the overlap percentage constant here, you can experiment with different values if needed. A chunk overlap between 5% and 20% of the chunk size is recommended for most datasets.
- There is no concept of chunk size for semantic chunking since it uses an embedding distance threshold to determine chunk boundaries.

The evaluation results for the different chunking strategies look as follows on our evaluation dataset:

Chunking Strategy	Chunk Size	Context Precision	Context Recall
Fixed token without overlap	100	0.8583	0.7833
	200	0.9	0.9
	500	0.8833	0.95
	1000	0.9	0.8909
Fixed token with overlap	100	0.9	0.95
	200	1.0	0.9383
	500	0.7	0.9
	1000	0.7833	0.8909
Recursive with overlap	100	0.9	0.9833
	200	0.9	0.9008
	500	0.5667	0.8236
	1000	0.7833	0.88
Recursive Python splitter with overlap	100	0.9833	0.9833
	200	1.0	0.8583
	500	0.6	0.88

Chunking Strategy	Chunk Size	Context Precision	Context Recall
	1000	0.8	0.8709
Semantic chunking	N/A	0.9	0.8187

Based on the above results, a recursive Python-specific splitting technique with a chunk overlap of 15 tokens and a chunk size of 100 is the best chunking strategy for our dataset.

Bear in mind that we have used a very small evaluation dataset of 10 samples for demonstration so the above results cannot be treated as fully conclusive. In reality, you will want to use a larger, more representative evaluation dataset.

Conclusion

In this tutorial, we learned how to choose the right chunking strategy for RAG. Breaking up large documents into smaller chunks for RAG results in fewer tokens passed as input to LLMs, and a more targeted context for LLMs to work with. Chunking strategies are composed of three key components – splitting technique, chunk size, and chunk overlap. Picking the right strategy involves experimenting with different combinations of the three components.

Now that you have a good understanding of chunking for RAG, take it up as a challenge to evaluate different chunking strategies on datasets that were used in some of our previous tutorials:

- Building an AI Agent with Memory Using MongoDB, Fireworks AI, and LangChain
- How to Build a RAG System Using Claude 3 Opus and MongoDB

If you have further questions about RAG, Vector Search, or AI in general, please reach out to us in our Generative AI community forums and stay tuned for the last tutorial in the RAG series. Previous tutorials from the series can be found below:

- Part 1: How to Choose the Right Embedding Model for Your RAG Application
- Part 2: How to Evaluate Your RAG Application

Top Comments in Forums

Sheersh_Saxena Sheersh Saxena **3 quarters ago**

in the final piece of code I am encountering SSL handshake error.

Reply

See More on Forums

Rate this tutorial

Related

TUTORIAL

Securely Connect MongoDB to Cloud-Offered Kubernetes Clusters

Sep 09, 2024 | 4 min read

NEWS & ANNOUNCEMENTS

Deprecating MongoDB Atlas GraphQL and Hosting Services

Mar 13, 2025 | 2 min read

TUTORIAL

Getting Started with MongoDB Atlas Local Search Experience Using Docker

Feb 11, 2025 | 7 min read

ARTICLE

Atlas Search is a Game Changer!

Sep 09, 2024 | 2 min read

Request a Tutorial

English

About

Support