Hello, Welcome to MongoDB Learning Bytes, where you can pick up knowledge on MongoDB in less than 20 minutes.

I'm <name> and I'm a <role> at MongoDB.

By now, you've probably been reading about or experimented with Retrieval Augmented Generation, or RAG applications.

If you haven't already, check out our content on RAG before continuing here.

As a quick refresher, RAG provides more context to a large language model or LLM. As a result, LLMs are able to provide better responses.

The user provides a query which is used to perform a vector search on the data in our MongoDB Atlas Cluster.

Chunks of relevant data are returned and combined with the original query to form a prompt, which is sent to a generative model.

Then the generative model uses the system prompt along with its own data to generate a response.

RAG applications supplement LLMs with additional information so the LLM can provide a better response to the user query. This helps us avoid hallucinations where the LLM could reply with outdated or inaccurate information.
In this process, how you break up your data into chunks matters. The strategy that you choose can determine how accurate and relevant the LLM's responses will be to the user's query.
In this Learning Byte, we're going to talk about Chunking strategies for your RAG application.

By the end, you'll understand some of the basic chunking strategies, how to fine tune a strategy, and how to choose the most appropriate strategy for your use-case.

Let's get started!
Chunking is the process of breaking up source text into smaller parts.

As mentioned earlier, those parts are stored in a database, like an Atlas Cluster.

We can then use Atlas Vector Search to find relevant chunks based on a user's query.

Chunks can be the exact data from the source document, or include some transformed version that provides more useful information to the LLM.

For example, imagine our source data is code for a mobile app. Some chunking strategies use LLMs to presummarize each chunk of code that is created when we break up that source data. Then vector embeddings are created for these text summaries and stored in a document alongside the corresponding chunk of raw code.

We could have a summary that tells us something like, the code in this chunk is an excerpt from a file that handles the mobile app's network connections, and so on.

Now, if we're searching for code related to network connections, it is more likely that relevant chunks will be retrieved.
Presummarization, like in this example, is a more advanced technique, so we won't cover it in this learning byte. But it's important to know what's possible when you're considering chunking strategies for your use-case.

And to that end - it's also important to keep in mind that modern AI models are increasingly multi-modal, meaning that they can understand more than just text. So, while this learning byte will cover simple code and text examples, you can chunk data that includes images, videos, and audio too!

We'll dive into some examples of chunking strategies in a moment.
But first, you may be wondering - why do we have to chunk at all?

Why don't we just feed the LLM the entire source document and let it determine what is relevant to the user's query?

Well, we chunk for two reasons:

First, LLMs have context windows and we chunk data so that it can fit into an LLM's context window.

The context window is the number of tokens, or words, partial-words, characters and punctuation, that the LLM can consider at a time, and this is limited. The size of the context window is determined by the model's token limit.
If a model reaches its token limit, it will either begin to forget earlier tokens to make room for new ones or stop generating and return its response immediately, even if it is mid-way through a word or sentence.

So, instead of inputting a single document with thousands or millions of tokens, which could exceed the LLM’s token limit and overload the context window, we break the source document into chunks that each contain fewer tokens.

This way, we only send relevant chunks to the generative model that fit in the context window.

For example, imagine our LLM has a token limit of 32 thousand tokens and our source has 50,000 tokens worth of data. Instead of feeding the LLM the source with 50,000 tokens and overloading its context window, we could break up the source into smaller chunks using a chunk size that makes sense for our data.

For example, we could end up with 50 chunks, with each chunk containing approximately 1,000 tokens.

Then, we'd send only the most relevant chunks, which will now easily fit into the context window along with the necessary metadata.

You might have heard that the amount of data that LLMs can process is constantly growing. For example, at the time of this video, several LLMs can fit well over 100 thousand tokens in their context window.
So will we still need to chunk data as LLMs improve?

The answer is yes, because chunking also improves the accuracy, relevancy, and precision of results.

Even with a larger context window, LLMs are slow to ingest tokens, they can still ‘forget’ tokens if they are ingesting a large dataset because it is not obvious which tokens are most important. And they require enormous resources.

Maybe you’ve heard of the ‘needle in the haystack test’ that is used to evaluate LLM RAG system performance?

Well, perhaps most importantlyIn addition, breaking a large source of data into small chunks can help yourallows the RAG application pass this test by to more efficiently searching through and retrievinge only the most relevant data.

Now that we know why we chunk, let’s look at some of the chunking strategies that you can use to break up your source data.

All chunking strategies have three key components that you can adjust to fine tune results:

Chunk size,
chunk overlap,
And splitting technique.
Chunk size is the maximum number of tokens contained in each chunk.

This value should always be less than the token limit for the LLM that you are using in order for your data to fit in the context window.

And in most cases, we will end up sending multiple chunks with metadata - so all of that data must fit comfortably in the context window.

So if an LLM has a context window of 2048 tokens, your chunks should probably be no bigger than two to three hundred tokens.

Just keep in mind that the smaller the chunk size, the more chunks you will have, which could require additional memory and storage.

Chunk overlap is the number of overlapping tokens between two adjacent chunks.

This overlap will create duplicate data across chunks.

Overlap can help preserve context between chunks and improve your results.

A larger chunk overlap will result in chunks sharing more common tokens, while a smaller chunk overlap will result in chunks sharing fewer common tokens.

While this duplication might appear unusual to the human eye, it significantly increases the chances of generating a more contextually rich prompt for the LLM. It also makes it less likely that we will send incomplete information to an LLM.

However, depending on your use-case, you might end up using a strategy where it doesn't make sense to have overlapping chunks.
And finally, the splitting technique determines where one chunk will end and the next will begin.

Splitting techniques can range from naive, like splitting a text by character or token, to incredibly complex, like using an LLM to semantically split your data for you.

To make our chunking strategy, we will combine these three components.

Now let's go deeper into different chunking strategies and how chunk size, chunk overlap, and splitting techniques are involved.
We'll look at recursive splitting, document specific splitting, and semantic splitting strategies.

We'll use splitting methods from langchain to demonstrate these strategies, but there are many other frameworks and tools available. You can even build your own!
Let's start with a recursive text splitter.

A recursive text splitter is a technique that breaks down large text files into smaller, more manageable segments.

Unlike a simplistic, non-recursive splitter which could end a chunk in the middle of a sentence, a recursive text splitter preserves semantic relationships in text by using separators like line breaks to recursively divide text into smaller and smaller chunks.

In theory, the resulting chunks are more meaningful and useful for further processing.

By default, a recursive splitter divides text into chunks by double newlines first. For example, in a research paper, you could think of this as separating chunks by big sections.

Then it will consider the chunk size that we set. If the chunks of text are bigger than our desired chunk size, it will then divide our chunks by single newlines.So again, in a research paper you could think of this as separating by paragraph.

This process will continue with spaces, and characters until the desired chunk size is achieved.

If our desired chunk size is large enough, it's possible that the recursive splitter will never actually make it to the space or character separator, for example.

With this approach, we can keep all paragraphs, sentences, and words together as long as possible. This way, semantically related pieces of text are more likely to be kept together in the same chunk, which should improve the relevancy of results.

Let's see how this works.

As we go through examples, keep in mind that we aren't showing complete code files. Check out the code recap if you're interested in seeing complete examples.

We're using the RecursiveCharacterTextSplitter method from langchain to chunk our example text. This method allows us to specify chunk size and chunk overlap as options. We'll start by setting chunk size to 50 and chunk overlap to 0.

This should create chunks that each contain up to 50 tokens with no overlap between chunks.

We're starting with a small chunk size to help us visualize how this strategy works, but you'll most likely want to start with a larger chunk size for your use case.

Let's run it and see what happens.

Ok, so here we can see that the first chunk, in blue, stops at the end of the title because the RecursiveCharacterTextSplitter method hit a double line break, so even though it is smaller than the chunk size that we specified, it created a chunk.

Next, we have a series of chunks in the main paragraph, and while we aren't cutting chunks off in the middle of words, we are breaking up a few sentences.

We've broken up these sentences because we started off with a small chunk size. If we want to try to capture whole sentences, paragraphs, or even pages, etc., we can adjust the chunk size. Let's adjust our chunk size to 200 to see if we can capture more full sentences.

Now we have fewer chunks and we're capturing more full sentences, but we're still breaking up sentences in some places.

This is where chunk overlap can be helpful to preserve the context between chunks.

Let's adjust the chunk overlap to 20.

Now, instead of the third chunk, which is orange, starting and stopping mid sentence, it overlaps with its neighboring chunks, as you can see in grey, and this overlap preserves the context of the entire sentence.
The recursive text splitter is a great place to start when you're experimenting with splitting techniques.

It allows you to capture sentences, paragraphs, sections, and more, which can help to capture semantic meaning in text.

But you'll need to look at your data and experiment with chunk size and overlap to make sure that your chunks are optimally capturing the semantic meaning in your text
If memory or storage are not a concern, this is a good default method for chunking a text document, like a PDF or Word document.
However, this approach can come with an increased number of vectors and vector memory requirements, depending on chunk size and overlap.
So if memory and storage are a concern or you are working with a large data set, this technique might not be a good fit.

That covers chunking text documents with a simple recursive splitter, but what if you're working with a different document type, like Python files? Wouldn't it make more sense to split by classes instead of double line breaks in this case?

This is where the document specific splitting strategy is useful. Document Specific splitters use separators that are unique to the document type to create chunks.

Let's take a look!

Take the Recursive Python splitter as an example. It uses this list of separators when creating chunks. So, it starts by splitting classes first, then recursively splits by functions, indented functions, double new lines, new lines, spaces, and finally, characters.

Like the simple recursive splitter, before proceeding to the next stage, it checks the chunk size that we specified to determine whether a chunk is small enough or if it needs to be split again.

Let's see this in action.
We'll use a simple example of a Person class to demonstrate.

We're using the RecursiveCharacterTextSplitter method from langchain again, but this time, we are chaining it with the from_language method and using the language option to specify that we want to use separators for Python.

We are starting with a chunk size of 100 and overlap of 0.

With these settings, we end up with 4 chunks. Notice that the Person class is broken into 3 separate chunks. We want to keep this Person class together since incomplete code won't help us.

To do this, we'll adjust the chunk size to 200 to see what happens.

Nice! Now the entire Person class is included in the first chunk. In this particular case, we don't really need any overlap to preserve the context between the Person class and the example usage so we leave the value of chunk overlap as 0.
Document specific splitting allows you to tailor your chunking strategy to the format of your data.

When using this strategy with documents that contain code, you can also use an LLM to create a summary for each chunk of code and store that summary alongside the chunk as metadata.

So, for our Python example, we might end up with a summary that says something like, "This is the function body of the Person.greet method. The full file defines the Person class, which has a name and can say hello."

This summary will help to increase the relevancy of results during retrieval.

Document Specific Splitting can be used for documents containing code but it's also great for multi-modal text splitting if, for example, your text documents also include images.

Remember that this technique is all about making sure that your strategy fits your data formats - so make sure that you understand your data and experiment.
We've covered Recursive Splitting Strategies and Document Specific Strategies, now let's talk about a slightly more complex strategy - Semantic Splitting.
Semantic splitting uses vector embeddings to split text based on semantic similarity.

For this example, we'll demo semantic splitting by using the Semantic Chunker method from langchain.

At a high level, this method works by splitting the source text into individual sentences based on punctuation, like periods or exclamation marks. Then it combines individual sentences into groups of three.

From there, it uses vector embeddings to compare those groups.

If two groups are similar enough, they are merged together. The method continues to compare and merge groups until the two groups being compared are considered different enough that they belong in separate chunks.

Let's look a little closer at how this is measured with a different example. Here we have two groups of sentences, one about cats and the other about horses.

By default, the semantic chunker method compares the cosine distance between vector embeddings for each group in sequential order.

Then it uses a percentile to determine where to split the text into chunks.

If the cosine distance between two groups falls below the default 95th percentile of distances, they are considered close enough or similar enough to be combined into a chunk.

But if they are above the 95th percentile, like in this example, then they will be considered far enough apart or different enough to remain separate chunks.

95 is the default percentile for the Semantic Chunker method but you can change this value. You can also choose to use standard deviation or interquartile distance to compare values instead of percentile.

Don't worry if this seems like a lot to take in - this is a complex strategy and the Semantic Chunker method is going to do the work for you.

We just want to give you an idea of how it works so that you know what your options are if you choose to experiment with it.
At this point, you might be wondering - if we are splitting by semantic similarity, does this mean that we can have chunks that are different sizes?

This answer is, yes! We could end up with a chunk containing three sentences and another chunk containing several paragraphs.

At the time of this recording, we cannot configure LangChain's Semantic Chunker method to limit chunk size.

If we want to specify chunk size as a part of our chunking strategy, we can always create a solution that breaks down the larger chunks after running the Semantic Chunker method, but that is out of scope for this video.

To get a better idea of how Semantic Splitting works with the Semantic Chunker method, let's try it out.
We're going to use an example text file that contains sentences from Practical Aggregations by Paul Done and sentences that we generated using AI about puppies.

Ideally, the semantic chunker method will be able to tell that each section has a different semantic meaning, even though they are lumped together in the same paragraph.

We're going to open the text example file and read its contents with the read() method.

Next we create the semantic text splitter with langchain's SemanticChunker method. This is also where we create the vector embeddings for each group of sentences.

Finally, we are going to split the text.

When we run this and print results to the terminal, we see that it did a good job of chunking by semantic meaning.

The blue chunk includes the passage from Practical Aggregations and the green chunk includes the sentences about puppies.

But this was an easy example. Let’s try something different.
This text file contains two passages from the Practical Aggregations book on two separate topics that have been combined into a single paragraph. The first passage generally defines MongoDB’s aggregation framework and the second passage covers explain plans.

When we run the semantic chunker method again, we see that it groups the sentences together according to topic.
This strategy may seem like a no-brainer for chunking text, like PDFs or word documents, since we ideally want to create chunks based on semantic meaning.

But at the time of this video, this strategy is still experimental. It’s important to test semantic splitting to confirm that it gives you the desired results.

Another factor to consider is cost. Since we are generating vector embeddings and using them to chunk by semantic meaning, this could be expensive and time-consuming depending on the size of your data set.
We’ve covered recursive splitting, document specific splitting, and semantic splitting, but this is by no means an exhaustive list - it’s really just the beginning!

As you experiment with chunking, you may want to try other methods, like LLM pre summarization, Agentic Splitting, or Parent Document Page retrieval.
Once you decide on a chunking strategy or strategies for your application, it’s important to continue to evaluate them.

Your data may change over time and additional strategies may emerge that will better suit your use-case.

Let’s recap what we covered in this learning byte:

Chunking is the process of breaking up source data into smaller chunks.

We chunk for two reasons:

To break data into smaller chunks that can easily fit into an LLM’s context window and

to improve the accuracy, relevancy, and precision of results

All chunking strategies have three key components that you can adjust to fine tune results:

Chunk size,
chunk overlap,
And splitting technique
We experimented with three chunking strategies in depth, including:

Recursive Splitting
Document Specific Splitting and
Semantic Splitting

Once you decide on a chunking strategy or strategies for your application, it’s important to continue to evaluate them.
To see the full code for the examples in this learning byte, check out the code recap section.

To learn more about building RAG applications with MongoDB, check out the MongoDB University course on RAG.

We hope you enjoyed this Learning Byte!