[COMPLETE] RAG and Vector

What You Will Learn About Building a RAG System

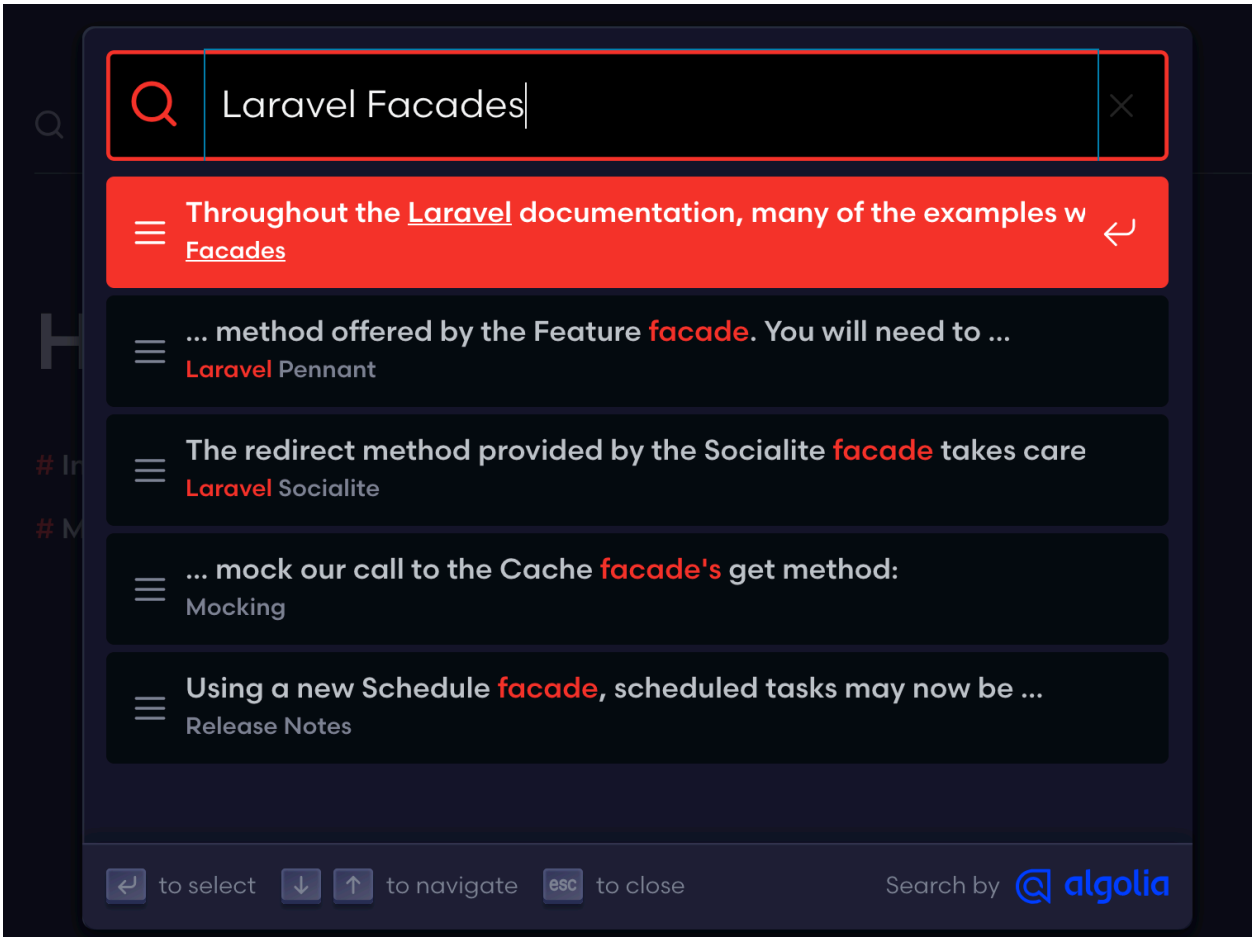
In this chapter, you will learn the steps involved in building a Retrieval-Augmented Generation (RAG) system using a vector database.

- The chapter explains how to set up a **vector database** using Postgres and the pgvector-php library. The sources use the example of a database about the sport of pickleball.
- You will learn how to ingest data into the database, including how to **chunk** it into smaller pieces for better search results.
- The sources describe the process of **embedding** the data using an LLM API, such as Ollama or OpenAI. Embedding converts text into a numerical representation that can be processed by a machine learning model.
- The chapter provides a step-by-step guide on how to query the vector database using an embedded question. You will learn about the different distance types used in vector searches, such as cosine distance.
- The sources also cover how to incorporate images and audio into a RAG system. They explain how to extract text from images using OpenAI's API and how to transcribe audio using OpenAI's Whisper model.
- Finally, the chapter discusses the importance of structuring data in a RAG system to improve the accuracy and relevance of search results.

> Retrieval-Augmented Generation (RAG) is a technique that enhances AI language models by combining their built-in knowledge with relevant information retrieved from an external database. This technique allows the AI to provide more accurate, up-to-date, and context-specific responses by drawing on both its general knowledge and specific, retrieved facts. —Claude

Why RAG?

This is a good question, and one I sometimes skip over because I am like, "Why not?" Most applications use search, like Algolia, and it does an amazing job finding content - searching "Laravel Facades" will give me a list of document pages with that information.



In this case, I want my user to be able to ask questions about the data (that might be private or public), however I want the LLM to confine its results to just that data.

For example:

"I want to make a facade from my LLMDriver class. How can I do that?" I want the response to include not only documents that have the word "Facade", but also more about the context of the question, allowing the system to return a greater depth of results that I then pass to the LLM to answer the question.

Remember, this is not just about chatting with your data; you can also automate tasks around it. You could use your Vector database to find content related to other content that otherwise might be hard to do on keyword search alone. For example, your system gets a support question about how to refund a customer; your RAG system does a vector query, finds several answers, and then creates a very nicely formatted and worded email to help the customer step-by-step.

One thing I want to emphasize is that I think RAG is great, but I believe it is not an end in itself, but rather a way to store your data and use it for other tasks like automating tags, finding related articles, user shopping preferences, customer support, email automation, and more.

What'll we do with RAG?

I recently wrote a Medium article Laravel RAG System in 4 Steps![1]. This chapter will dig into the details of each step mentioned in the article, but also go further in a few ways. First, we will go into processing images, then how to make tables as HTML, and lastly, hybrid search so we get a bit more out of our search steps.

Here's a quick overview:

1. Set up the vector database so we can do searches. Unlike keyword searches, vector searches can understand context and meaning, finding related items even if they don't share exact words.
2. Pass those results to the LLM using a prompt that keeps it from drifting or hallucinating.
3. Allow the user to have a threaded discussion.

So how do we get there?

1. We use Postgres as our vector database.
2. We install `pgvector-php` to give us the ability to do vector searches.
3. We import the required data, chunking it so that when we get results, it is small sets of data and not large full pages. This way, we know which section of the page matches.
4. We take the user's question and embed it using an LLM API, and then use it to search.
5. We then take the results and ask the LLM to stay within them to answer the original question.

Later, we will add a step to maximize images and tables, so that we can show them differently in the UI and utilize them better in our responses.

Getting Your Application Database Set up

We are using Postgres as our main database here. (You can use it as a second database if you already use MySQL.) Laravel has an easy way to set up Postgres, and it will be our central database for now. Once Postgres is installed, the library will do much of the heavy lifting regarding vector. When the database is ready, we'll get data into the system. We'll ingest a bunch of markdown files from a folder. Later, we'll try more complex PDFs.

The easiest ways to run Postgres locally for this is either Laravel HERD[2], the paid version, or DBngin[3] for free.

To deploy, you can easily use the Forge[4] Postgres option on your existing server or deploy it as a standalone server. Then, if you follow the installation instructions of https://github.com/pgvector/pgvector-php you will be all set. Keep in mind, the migration that it adds to your system needs permissions to run:

[1] https://medium.com/@alnutile/laravel-rag-in-4-steps-6264b4df173a
[2] https://herd.laravel.com/
[3] https://docs.tableplus.com/utilities/dbngin
[4] https://forge.laravel.com/

Figure 149. Migration

```
DB::statement('CREATE EXTENSION IF NOT EXISTS vector');
```

So as long as you are using the default user and permissions from Herd, DBngin and Forge, you should be fine.

For example, my local `.env` looks like this:

Figure 150. Local .env

```
DB_CONNECTION=pgsql
DB_HOST=127.0.0.1
DB_PORT=5432
DB_DATABASE=laravel_llms
DB_USERNAME=postgres
DB_PASSWORD=
```

> NOTE: When using Postgres, Laravel does not seem to be able to create the schema for you, as with MySQL. Using a tool like TablePlus[5], you can easily add the new schema.

Now that the database is ready, let's get some data and ingest it into the database.

Data and Ingestion

We're going to use the sport Pickleball as our example source data. We are going to start with a small collection of publicly-available files I have about Pickleball in markdown format; you can see them in `storage/data/pickleball`. As with most of this data, we save it in two tables. One will be called `documents` and the other `chunks`. Then, we `embed` the data using the Ollama API (the same as OpenAI) and store those embeddings. Once done, we will have one document per file and multiple chunks per file size.

Document Table

Let us start with the document model. The model is a straightforward table - you can see it all here:

[5] https://tableplus.com/

Figure 151. database/migrations/2024_09_15_211042_create_documents_table.php

```php
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration
{
    /**
    * Run the migrations.
    */
    public function up(): void
    {
        Schema::create('documents', function (Blueprint $table) {
            $table->id();
            $table->string('title')->nullable();
            $table->longText('summary')->nullable();
            $table->longText('content')->nullable();
            $table->timestamps();
        });
    }

    /**
    * Reverse the migrations.
    */
    public function down(): void
    {
        Schema::dropIfExists('documents');
    }
};
```

Once that is in place, we can now make the more complex model `chunks`.

Chunk Table

A lot of this will be new, so let us dig in one file at a time.

Chunks Table Migration

Figure 152. database/migrations/2024_09_15_211349_create_chunks_table.php

```
Schema::create('chunks', function (Blueprint $table) {
    $table->id();
    $table->string('guid');
    $table->string('sort_order')->default(1);
    $table->longText('content')->nullable();
    $table->longText('summary')->nullable();
    $table->vector('embedding_768', 768)->nullable();
    $table->vector('embedding_1536', 1536)->nullable();
    $table->vector('embedding_2048', 2048)->nullable();
    $table->vector('embedding_3072', 3072)->nullable();
    $table->vector('embedding_1024', 1024)->nullable();
    $table->vector('embedding_4096', 4096)->nullable();
    $table->integer('page_number')->nullable();
    $table->foreignIdFor(\App\Models\Document::class, 'document_id');
    $table->timestamps();
});
```

The unique field here is `vector` and the vector types. Here, I allow the system to have more than one vector type, depending on our model. In this initial system example, we use the one embedding type. But later on in other chapers, we will use other models for cost (local vs. OpenAI) or speed. When we save our data to the model, I will explain more about which LLM model we use and the embedding size.

The Factory Database/Factories/ChunkFactory.php

Figure 153. database/factories/ChunkFactory.php

```
    public function definition(): array
    {
        $embeddings = get_fixture('embedding_response.json');

        return [
            'guid' => fake()->uuid(),
            'content' => fake()->sentence(10),
            'page_number' => fake()->numberBetween(1, 100),
            'section_number' => fake()->numberBetween(1, 100),
            'original_content' => fake()->sentence(10),
            'summary' => fake()->sentence(5),
            'embedding_3072' => data_get($embeddings, 'data.0.embedding'),
            'embedding_1536' => null,
            'embedding_2048' => null,
```

```
            'embedding_4096' => null,
            "document_id" => Document::factory(),
        ];
    }
```

Here, we can use the fixture file to generate a vector for the factory. A segment of that file looks like this:

Figure 154. tests/fixtures/embedding_response.json

```
{
    "object": "list",
    "data": [
        {
            "object": "embedding",
            "index": 0,
            "embedding": [
                -0.02932045,
                -0.019364825,
                // and on and on and on...
```

The Model App/Models/Chunk.php

And, of course, the model file:

Figure 155. app/Models/Chunk.php

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;
use Pgvector\Laravel\HasNeighbors;
use Pgvector\Laravel\Vector;

class Chunk extends Model
{
    use HasFactory;
    use HasNeighbors;

    protected $guarded = [];

```

```
    protected $casts = [
        'embedding_768'  => Vector::class,
        'embedding_3072' => Vector::class,
        'embedding_1536' => Vector::class,
        'embedding_2048' => Vector::class,
        'embedding_1024' => Vector::class,
        'embedding_4096' => Vector::class
    ];

    public function document(): \Illuminate\Database\Eloquent\Relations\BelongsTo
    {
        return $this->belongsTo(Document::class);
    }
}
```

For the `trait` and the `casts`, we follow these docs: here[6].

And we have many casts! But now we are ready to make a class to fill up this data.

Ingestion and Embedding

So, how you get the data in here is up to you: file uploads in the UI, web scraping, or a folder, etc. What I want to cover, though, is how to **chunk** the data and then how to **embed it**.

Chunking

We will keep this simple: you will see in Larallama.io[7], and this will lead to this ProcessFileJob[8].

Here is the goal:

Figure 156. app/Domains/Documents/ChunkContent.php

```
<?php

namespace App\Domains\Documents;

class ChunkContent
{
    public function handle(string $content, string $title = null): string
    {
        // 1. Make a document model
```

[6] https://github.com/pgvector/pgvector-php?tab=readme-ov-file#laravel
[7] https://github.com/LlmLaraHub/larallama/blob/main/app/Http/Controllers/CollectionController.php#L116
[8] https://github.com/LlmLaraHub/larallama/blob/main/app/Jobs/ProcessFileJob.php#L111

```
        // 2. Break the content into chunks
        // 3. Save the chunks to the database
        // 4. Embed the chunks
        // 5. 🎉 Done
    }
}
```

Break the Content into Chunks

We add this to the file:

Figure 157. app/Domains/Documents/ChunkContent.php

```
<?php

namespace App\Domains\Documents;

use App\Domains\Chunking\TextChunker;
use App\Models\Document;

class ChunkContent
{
    public function handle(string $content, string $title = null): string
    {
        // Make a document model
        $document = new Document();
        $document->title = $title;
        $document->content = $content;
        $document->save();

        // Break the content into chunks
        $chunks = (new TextChunker)->handle($content);
        // Save the chunks to the database
        // Embed the chunks
        // Done
    }
}
```

Making the **Document** model is easy, and breaking the text into `chunks` is easy as well. Let's look into that class.

Figure 158. app/Domains/Documents/Chunking/TextChunker.php

```php
<?php

namespace App\Domains\Chunking;

class TextChunker
{
    public static function handle(string $text, int $chunkSize = 600, int $overlapSi\
ze = 100): array
    {
        $chunks = [];
        $textLength = strlen($text);

        // Calculate where the first chunk starts and the subsequent chunks.
        for ($start = 0; $start < $textLength; $start += ($chunkSize - $overlapSize)\
) {
            if ($start + $chunkSize > $textLength) {
                // Get the remaining text if it's shorter than the chunk size.
                $chunks[] = substr($text, $start);
                break;
            }

            // Get the chunk from the text.
            $chunks[] = substr($text, $start, $chunkSize);
        }

        return $chunks;
    }
}
```

We are taking that text and considering a chunk size of `600` and an overlap of `100`, and then making an array of these chunks.

We want to break the data into small enough chunks that when we embed it and search for it, we get just enough data back and not too much. For example, if I make the entire page a chunk, but the user is searching for something that is really in the middle of the page, then we are going to return the entire page, giving the LLM too much context to really answer the users's promopt (question) well.

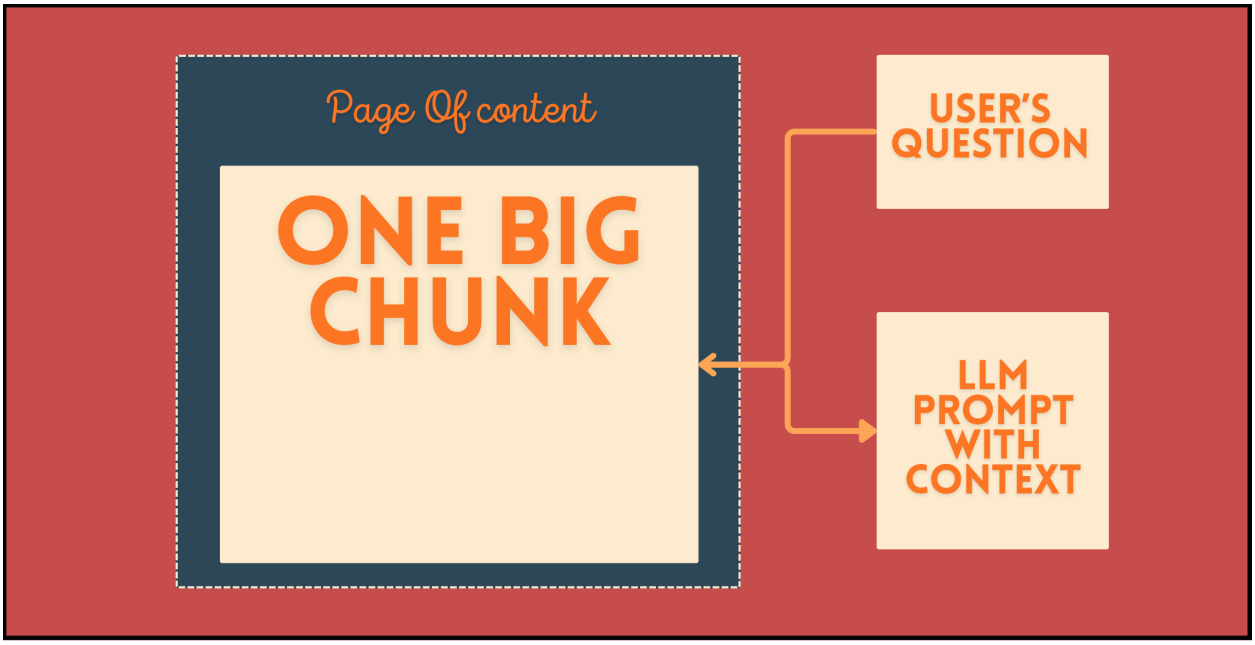


Figure 159. Too much Context

Versus smaller chunks with overlap:

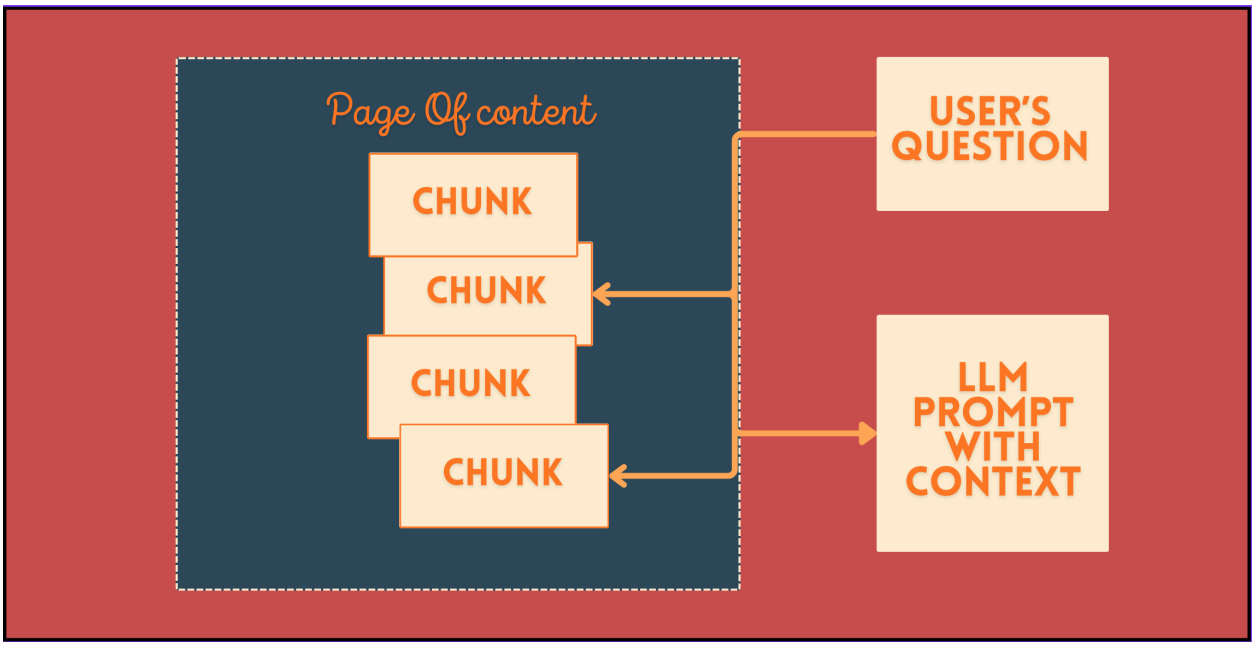


Figure 160. Just the right Chunk

We are starting to give the LLM better context so it can focus on the answer to the user's prompt. Now, let's save the chunks.

Figure 161. app/Domains/Documents/ChunkContent.php

```php
<?php

namespace App\Domains\Documents;

use App\Domains\Chunking\TextChunker;
use App\Models\Chunk;
use App\Models\Document;

class ChunkContent
{
    public function handle(string $content, string $title = null): string
    {
        $document = new Document();
        $document->title = $title;
        $document->content = $content;
        $document->save();

        $chunks = (new TextChunker)->handle($content);

        foreach ($chunks as $chunkSection => $chunk) {
            $page_number = 1;
            Chunk::create(
                [
                    'section_number' => $chunkSection,
                    'content' => $chunk,
                    'document_id' => $document->id,
                    'sort_order' => $page_number,
                ]
            );
        }
    }
}
```

So, we simply iterate through the chunk, saving the chunk's order and content.

And we can see in our test that there are many chunks for one file!

Figure 162. tests/Feature/ChunkContentTest.php

```php
<?php

test('can chunk content', function () {
    $data = get_fixture("example_markdown.txt", false);

    $results = (new \App\Domains\Documents\ChunkContent)->handle($data);

    expect($results->content)->not->toBeNull();
    expect($results->chunks->count())->toBe(23);

});
```

That is one file in the docs about Artisan!

But we still need to embed the chunk - let's move on to the most important part.

Embedding

Now, we will embed the chunk of content. This can take time if you are using your machine and Ollama; it depends on your machine's speed. If you are not multi-threading the work, as with a batch system, then it will be limited to one thread at a time. For this example, I will use Ollama to start, and then OpenAI.

Embedding with Ollama

First, we need to pull down a model that enables us to embed. If we visit https://ollama.com/search?c=embedding, we see the embedding models. Typically, I use **mxbai-embed-large**, but for this one, I want to try **nomic-embed-text**.

We need to fist **pull** down the model from Ollama:

Figure 163. ollama

```
ollama pull nomic-embed-text
```

We are ready to use the model in our app once we have it.

And in our code, per the `ollama` docs https://github.com/ollama/ollama/blob/main/docs/api.md#generate-embeddings, we generate embeddings using the abstracted driver system we built in the previous chapter. But first, for a moment, let us just use it directly in this code so we can see a very simple use case:

Figure 164. app/Domains/Documents/ChunkContent.php

```php
<?php

namespace App\Domains\Documents;

use App\Domains\Documents\Chunking\TextChunker;
use App\Models\Chunk;
use App\Models\Document;
use Illuminate\Support\Facades\Http;

class ChunkContent
{
    public function handle(string $content, string $title = null): Document
    {
        $document = new Document();
        $document->title = $title;
        $document->content = $content;
        $document->save();

        $chunks = (new TextChunker)->handle($content);

        foreach ($chunks as $chunkSection => $chunk) {
            $embedding = Http::withHeaders([
                'Content-Type' => 'application/json',
                'Authorization' => 'Bearer ollama',
            ])->post('http://localhost:11434/api/embed', [
                'model' => 'nomic-embed-text',
                'input' => $chunk,
            ])->json();

            Chunk::create(
                [
                    'content' => $chunk,
                    'document_id' => $document->id,
                    'sort_order' => $chunkSection,
                    'embedding_768' => $embedding['embeddings'][0],
                ]
            );
        }

        return $document;
    }
```

```
}
```

We see above that we are just an API call away from embedding our data! We just send it to the API we use (we will abstract this out shortly) and then save it to the correct column.

"How did I know the correct column?" - this is a good question. I could read the docs, or try it and get an error like this:

> SQLSTATE[22000]: Data exception: 7 ERROR: expected 2048 dimensions, not 768 (Connection: pgsql, SQL: insert into "chunks" ("content", "document_id", "sort_order", "embedding_2048", "updated_at", "created_at")

Then I add or use that correct column.

Before we refactor, let's ask our first question.

Our First RAG Question

We are going to build this "vector search" as a Tool shortly, but for now we will use TinkerWell[9] to show how this works so we can keep it simple and all in one place.

Figure 165. Tinkerwell

```
use App\Models\Chunk;
use App\Services\LlmServices\LlmDriverFacade;
use App\Services\LlmServices\Requests\MessageInDto;
use Illuminate\Support\Facades\Http;
use Pgvector\Laravel\Distance;
use Pgvector\Vector;

/**
 * 1) The user's question
 */
$question = "What are the top two tips to being a good teammate";

/**
 * 2) We embed the question so we can use it in the query
 */
$embedding = Http::withHeaders([
  "Content-Type" => "application/json",
  "Authorization" => "Bearer ollama"
])
```

[9] https://tinkerwell.app/

```
  ->post("http://localhost:11434/api/embed", [
    "model" => "nomic-embed-text",
    "input" => $question
  ])
  ->json();

/**
 * 3) Using the Vector library we instantiate the class
 *    with the question vectorized 🔫
 */
$embedding = new Vector($embedding["embeddings"][0]);

/**
 * 4) We use the Vector library to then search
 *   More on this shortly
 */
$searchresults = Chunk::query()
  ->select("chunks.content", "chunks.sort_order", "documents.title")
  ->join("documents", "chunks.document_id", "=", "documents.id")
  ->orderBy("chunks.sort_order")
  ->nearestNeighbors("embedding_768", $embedding, Distance::Cosine)
  ->limit(3)
  ->get()
  ->map(function ($chunk) {
    return "File Name: " .
      $chunk->title .
      "\n" .
      "Sort Order: " .
      $chunk->sort_order .
      "\n" .
      $chunk->content;
  })
  ->implode("\n\n");


/**
 * 5) Then we take this context and
 *    give it to the LLM with the original text version of the question
 *    to give a full answer to the question
 */
$messages = [
  MessageInDto::from([
    "role" => "system",
```

```
    "content" =>
      "You are an assistant with our collection of data about Pickleball. Do not ans\
wer questions outside the content given. If you do not know the answer due to a lack
 of information, ask the user to rephrase their question. The results will include F
ile Name and Sort Order. Please reference the File Name in your results, make a list
 of Files, and sort them in order as citations in Markdown at the bottom of the resu
lts."
  ]),
  MessageInDto::from([
    "role" => "user",
    "content" => $question
  ])
];

$payload = [
  "model" => "llama3",
  "messages" => $messages,
  "stream" => false,
  "options" => [
    "temperature" => 0
  ]
];

//HTTP HERE WITH OPTIONS
$results = Http::withHeaders([
  "content-type" => "application/json"
])
  ->timeout(120)
  ->baseUrl("http://localhost:11434/api/")
  ->post("/chat", $payload);

$results->json();
```

> NOTE: We set the temperature to 0 - this is key! We are saying, "Do not be creative, just stick to the facts". You could set it higher to experiment, say 0.2, and so on. You can read more about this in the OpenAi Docs[10]

Results:

[10] https://platform.openai.com/docs/api-reference/chat/create#chat-create-temperature

Figure 166. Results from question

```
[
    "model" => "llama3",
    "created_at" => "2024-09-21T10:56:53.398165Z",
    "message" => [
      "role" => "assistant",
      "content" => """
 Based on our collection of data about Pickleball, here are the top two tips to bein\
g a good teammate:\n
        \n
 1. **Communicate Effectively**: Good communication is key to success in Pickleball.\
 As a teammate, you should be able to convey your thoughts and intentions to your pa
rtner. This includes calling out shots, signaling where you want the ball to go, and
 providing encouragement or constructive feedback.\n
 2. **Support Each Other's Strengths**: A good Pickleball team complements each othe\
r's strengths and weaknesses. As a teammate, you should be aware of your partner's s
kills and play styles and work together to create opportunities for them to shine. T
his might mean setting up your partner for an easy shot or covering their weaknesses
.\n
        \n
 Files:\n
        \n
 * [Pickleball_Teamwork.pdf](#) (Sort Order: 1)\n
 * [Effective_Communication_in_Pickleball.md](#) (Sort Order: 2)\n
        \n
 Citations:\n
 [Markdown]\n
 [Pickleball_Teamwork.pdf](#) - "The Importance of Teamwork in Pickleball"\n
 [Effective_Communication_in_Pickleball.md](#) - "Why Communication is Key to Succes\
s in Pickleball"
        """,
    ],
    "done_reason" => "stop",
    "done" => true,
    "total_duration" => 13753143833,
    "load_duration" => 20333792,
    "prompt_eval_count" => 109,
    "prompt_eval_duration" => 2630338000,
    "eval_count" => 250,
    "eval_duration" => 11100467000,
  ]
```

Let's review each one of these steps in detail.

Step 1: The User's Question

Not much more info is needed here, so let us look at the next step.

Step 2: Embedding the Question

This is where things get interesting. We take the user's question and embed it using the API. It's just a normal HTTP request so we can see how this all works.

Step 3: Vectorize the Embeddings

Then we take that JSON result, which becomes an array of integers in PHP, and pass it into the `Vector` class to use in our query step.

Step 4: Query the Database

Then we query the database with this. Let's dig in here:

Figure 167. Vector Query

```
$searchresults = Chunk::query()
  ->select("chunks.content", "chunks.sort_order", "documents.title")
  ->join("documents", "chunks.document_id", "=", "documents.id")
  ->orderBy("chunks.sort_order")
  ->nearestNeighbors("embedding_768", $embedding, Distance::Cosine)
  ->limit(3)
  ->get()
  ->map(function ($chunk) {
    return "File Name: " .
      $chunk->title .
      "\n" .
      "Sort Order: " .
      $chunk->sort_order .
      "\n" .
      $chunk->content;
  })
  ->implode("\n\n");
```

Distance Types

Since I am not an AI or data expert, I am going to ask Claude for help explaining Cosine and the other options:

Figure 168. Vector Distances

```
L2 (Euclidean) Distance:

Description: Measures the straight-line distance between two points in Euclidean spa\
ce.
Use case: Best for when the magnitude of the vectors is important, and you want to f\
ind the closest vectors in absolute terms.
When to use: Ideal for applications where the scale of features matters, such as in \
geographical coordinates or physical measurements.


Cosine Distance:

Description: Measures the cosine of the angle between two vectors, ignoring magnitud\
e and focusing on direction.
Use case: Best when you want to find vectors pointing in a similar direction, regard\
less of their magnitude.
When to use: Ideal for text similarity, document classification, and recommendation \
systems where the relative importance of features matters more than their absolute v
alues.


Inner Product Distance:

Description: Measures the dot product of two vectors, considering both direction and\
 magnitude.
Use case: Useful when both the direction and magnitude of vectors are important, but\
 you want to emphasize similarity in larger magnitudes.
When to use: Often used in machine learning models, especially in neural networks, w\
here the inner product represents a similarity score.



In your RAG (Retrieval-Augmented Generation) system lookup, you used Cosine distance\
. This is a good choice for text-based applications because it focuses on the simila
rity of direction between vectors, which often represents semantic similarity in tex
t embeddings, regardless of the length of the text or the specific magnitudes of the
 embedding values.
```

So we will stick to Cosine for now but I was curious when the heck I would use the other ones!

Figure 169. When to use other types

```
There might be specific cases where you'd consider alternatives:

If the absolute magnitude of features is crucial (e.g., if the price range is a crit\
ical factor in product similarity), you might consider L2 distance.
If you want to emphasize both direction and magnitude (e.g., in a very specialized p\
roduct recommendation system where both similarity and popularity matter), you might
 use Inner Product distance.

However, for general-purpose related product recommendations and document similarity\
 in blog or documentation systems, Cosine distance is usually the most reliable and
widely-used choice.
```

Step 5: Use the Results for the Final Prompt

> Context is key!

Now that we have some results, we need to use those results as our **context**. In this example, I take the user's question and append the context to it. And then we add a `system` level prompt to make sure the LLM understands the goal and limits of its role:

Figure 170. Message Array as a DataObject

```
$prompt = [
    MessageInDto::from([
        "role" => "system",
        "content" => "You are an assistant with our collection of data about Pickleb\
all. Do not answer questions outside the content given. If you do not know the answe
r due to a lack of information ask the user to rephrase their question. Please inclu
de the document name and page in your response"
    ]),
    MessageInDto::from([
        "role" => "assistant",
        "content" => "Here is relevant information from our database:\n\n" . $search\
results
    ]),
    MessageInDto::from([
        "role" => "user",
        "content" => $question
    ])
];
```

Adding `assistant` array objects to the thread is fine. You might need to sort them based on the LLM you are using. (The Abstration chapter covers this topic).

Pull in all the Files and Computer Considerations

Getting data in can be done in many ways: file upload, webhooks, API, etc. Below, I just wrote a quick command to get the files into the database.

Figure 171. app/Console/Commands/ChunkFolder.php

```php
<?php

namespace App\Console\Commands;

use App\Domains\Documents\ChunkContent;
use Illuminate\Console\Command;
use Illuminate\Support\Facades\File;

class ChunkFolder extends Command
{
    /**
 * The name and signature of the console command.
 *
 * @var string
 */
    protected $signature = 'app:chunk-folder {absolute-path}';

    /**
 * The console command description.
 *
 * @var string
 */
    protected $description = 'Chunk all files in the folder';

    /**
 * Execute the console command.
 */
    public function handle()
    {
        $this->withProgressBar(File::allFiles($this->argument('absolute-path')), fun\
ction ($file) {
            try {
                $fileName = $file->getFilenameWithoutExtension();
                $content = File::get($file);
                $this->info('Chunking '.$file);
                app(ChunkContent::class)->handle($content, $fileName);
```

```
            } catch (\Throwable $e) {
                $this->error('Error chunking '.$file);
            }
        });
    }
}
```

This will take some time on my **M3**, so keep that in mind. I could use the OpenAI API, but this is a great example of how we can play with ideas without spending extra money. OK, that laptop is extra money, but I would have bought it anyway. These are just some thoughts on that while I wait.

Figure 172. TOPs

```
Tensor Processing Units (TOPs, or Tera Operations Per Second) and how this impacts y\
our computers ability to run a local LLM.

Here's a comparison of the TOPs (Tensor Operations Per Second) for the M1, M2, and M\
3 series Mac chips:

1. M1 Series:
 - M1: Up to 11 TOPs
 - M1 Pro: Up to 22 TOPs
 - M1 Max: Up to 44 TOPs
 - M1 Ultra: Up to 88 TOPs

2. M2 Series:
 - M2: Up to 15.8 TOPs
 - M2 Pro: Up to 31.6 TOPs
 - M2 Max: Up to 63.2 TOPs
 - M2 Ultra: Up to 126.4 TOPs

3. M3 Series:
 - M3: Up to 18 TOPs
 - M3 Pro: Up to 36 TOPs
 - M3 Max: Up to 72 TOPs
 - M3 Ultra: predicting 144 TOPs
```

Adding Images

Now for images! If you look in the folder `storage/data/pickleball/images`, you will see many images that are charts and data related to the documents. This can also be data we get out of

PowerPoints, PDFs, and more. All those seem to have decent tools we can use to not only strip the text out of them but also get the images and tables.

First let us refactor the `app/Console/Commands/ChunkFolder.php` class so when we come to an image we take a moment to call the OpenAI API (was not impressed with Lava 🤔).

Figure 173. app/Console/Commands/ChunkFolder.php

```
<?php

namespace App\Console\Commands;

use App\Domains\Documents\ChunkContent;
use App\Services\LlmServices\LlmDriverFacade;
use Illuminate\Console\Command;
use Illuminate\Support\Facades\File;
use Illuminate\Support\Facades\Http;
use Illuminate\Support\Facades\Log;

class ChunkFolder extends Command
{
    /**
 * The name and signature of the console command.
 *
 * @var string
 */
    protected $signature = 'app:chunk-folder {absolute-path}';

    /**
 * The console command description.
 *
 * @var string
 */
    protected $description = 'Chunk all files in the folder';

    /**
 * Execute the console command.
 */
    public function handle()
    {
        $this->withProgressBar(File::allFiles($this->argument('absolute-path')), fun\
ction ($file) {
            try {
                $fileName = $file->getFilenameWithoutExtension();

```

```
                $content = File::get($file);

                if(str(File::mimeType($file))->contains('image')){
                    $content = $this->getImageContent($file);
                }

                $this->info('Chunking '.$file);
                app(ChunkContent::class)->handle($content, $fileName);
            } catch (\Throwable $e) {
                $this->error('Error chunking '.$file);
                Log::error($e);
            }
        });
    }

    protected function getImageContent($file) : string
    {
        $prompt = <<<PROMPT

This is an image with chart data about Pickleball. Please describe the data shown in\
 the image and render it as a markdown table.
Do not add any flavor text.

PROMPT;

        $results = LlmDriverFacade::driver("openai")->image(
            prompt: $prompt,
            base64Image: base64_encode(File::get($file))
        );

        return $results->content;
    }
}
```

And now we end with more data like:

```
| Year | Total Players (millions) |
|------|--------------------------|
| 2017 | 3.1                      |
| 2018 | 3.3                      |
| 2019 | 3.5                      |
| 2020 | 4.2                      |
| 2021 | 4.8                      |
| 2022 | 8.9                      |
```

Then when we ask a question like "Total Players in 2021 compared to 2017", our system can easily give us an answer.

```
According to our database, the total number of Pickleball players in 2017 was 3.1 mi\
llion.\n
\n
To find the total number of players in 2021, we can look at the "player_growth" file\
:\n
\n
| Year | Total Players (millions) |\n
|------|--------------------------|\n
| 2017 | 3.1                      |\n
| 2018 | 3.3                      |\n
| 2019 | 3.5                      |\n
| 2020 | 4.2                      |\n
| 2021 | 4.8                      |\n
\n
According to this information, the total number of Pickleball players in 2021 was 4.\
8 million.\n
\n
To find the difference between the number of players in 2021 and 2017, we can subtra\
ct:\n
\n
4.8 (2021) - 3.1 (2017) = 1.7\n
\n
So, there were approximately 1.7 million more Pickleball players in 2021 compared to\
 2017.
```

Pretty nice!

Adding Voice

Let's do an audio example to show how easy this really is and how it is not far off from what we did above. In this case, I downloaded a short audio clip and then updated the `ChunkFolder` command as

below.

Figure 174. app/Console/Commands/ChunkFolder.php

```
<?php

namespace App\Console\Commands;

use App\Domains\Documents\ChunkContent;
use App\Services\LlmServices\LlmDriverFacade;
use Illuminate\Console\Command;
use Illuminate\Support\Facades\File;
use Illuminate\Support\Facades\Log;

class ChunkFolder extends Command
{
    /**
 * The name and signature of the console command.
 *
 * @var string
 */
    protected $signature = 'app:chunk-folder {absolute-path}';

    /**
 * The console command description.
 *
 * @var string
 */
    protected $description = 'Chunk all files in the folder';

    /**
 * Execute the console command.
 */
    public function handle()
    {
        $this->withProgressBar(File::allFiles($this->argument('absolute-path')), fun\
ction ($file) {
            try {
                $fileName = $file->getFilenameWithoutExtension();

                if (str(File::mimeType($file))->contains('image')) {
                    $content = $this->getImageContent($file);
                } elseif (str(File::mimeType($file))->contains('audio')) {
                    $content = $this->getAudioContent($file);
```

```
                } else {
                    $content = File::get($file);
                }

                $this->info('Chunking '.$file);
                app(ChunkContent::class)->handle($content, $fileName);
            } catch (\Throwable $e) {
                $this->error('Error chunking '.$file);
                Log::error($e);
            }
        });
    }

    protected function getAudioContent($file): string {

        $results = LlmDriverFacade::driver("openai")->audio(filePath: $file);

        return $results->content;
    }


    protected function getImageContent($file): string
    {
        $prompt = <<<'PROMPT'

This is an image with chart data about Pickleball. Please describe the data shown im\
age and render as a markdown table.
Do not add any flavor text.

PROMPT;

        $results = LlmDriverFacade::driver('openai')->image(
            prompt: $prompt,
            base64Image: base64_encode(File::get($file))
        );

        return $results->content;
    }
}
```

And added to the driver (see the chapter on this):

Figure 175. app/Services/LlmServices/OpenAiClient.php

```
    public function audio(
        string $filePath,
        string $model = "whisper-1"): CompletionResponse
        {
            $token = $this->getConfig('openai')['api_key'];

            if (is_null($token)) {
                throw new \Exception('Missing open ai api key');
            }

            $file = new UploadedFile($filePath, basename($filePath), null, null, tru\
e);

            $response = Http::withToken($token)
                ->baseUrl('https://api.openai.com/v1')
                ->timeout(240)
                ->attach('file', $file->getContent(), $file->getClientOriginalName())
                ->post('/audio/transcriptions', ['model' => $model]);

            if ($response->failed()) {
                Log::error('OpenAi API Error ', [
                    'error' => $response->body(),
                ]);

                throw new \Exception('OpenAi API Error Chat');
            }

            return CompletionResponse::from([
                'content' => data_get($response->json(), 'text', ''),
                'stop_reason' => 'stop',
            ]);
        }
```

I used OpenAI for this, making it so easy to send the file and get results.

> NOTE: I use HTTP here to show how easy it is to talk to these services. The PHP library for OpenAI[11] is great, but I want to enable you to talk to any API and see how it works here.

Now, when we pull in the files, it will take care of this one, adding a chunk that has nothing to do with pickleball!

[11] https://github.com/openai-php/laravel

Figure 176. Test Audio File Output

```
This is a test audio file to prove that this is working.
```

And that's it. Our RAG system now has audio and data from images.

Unstructured vs Structured

> ...almost like the use cases between NoSQL and SQL, I have to consider what can the LLM do for me and what do I have to do for it.

As this data comes in, we are treating it as unstructured. You can read about a very mature library[12] that helps to add some structure to this data. Sure, we did pull out some data from images in the example above, but we did not consider the importance of simple Markdown or HTML elements like H1, H2, H3, Bold and so on. We also did not mark the chunks as `type` of content for example "H1", "Image", "Table" and so forth. If you are working on a project where this level of detail is important, then putting together the PHP code to do it is of course possible.

Let us look at some existing code on this to see how it could be done. If you visit https://github.com/LlmLaraHub/larallama/blob/main/app/Jobs/ProcessFileJob.php#L17 you start to see that this job considers the different types of files. What this will lead to for a PPTX file is this: https://github.com/LlmLaraHub/larallama/blob/main/app/Domains/Documents/Transformers/ProcessPpt.php; let's take a moment to see the data type:

Figure 177. app/Domains/Documents/Transformers/ProcessPpt.php

```
if ($shape instanceof RichText) {
    $pageContent = $shape->getPlainText();
    $guid = $shape->getHashCode();

    $content = $this->output(
        type: StructuredTypeEnum::Narrative,
        content: $pageContent,
        page_number: $page_number,
        guid: $guid,
        element_depth: $shapeCount,
        is_continuation: $shapeCount > 0,
    );

    yield $content;
} elseif ($shape instanceof Table) {
    $table = $shape->getRows();
```

[12] https://unstructured.io/

```
    $this->title = 'Table';
    $this->subject = 'Table';

    $content = $this->output(
        type: StructuredTypeEnum::Table,
        content: 'table data',
        page_number: $page_number,
        guid: $shape->getHashCode(),
        element_depth: 0,
        is_continuation: false,
    );
    yield $content;
```

We could take time to do a few different integrations based on the type of data. If a table, we could use the LLM to turn it into a Markdown table. Or if it is an image we could do as we did before to pull the information out of it. Then at the same time we could add a field to the `chunks` table called `type` to make it clear what type of data this is. As we see earlier in the search method, we then can surface this data to help the LLM consider it in its results:

Figure 178. Example Search

```
$searchresults = Chunk::query()
  ->select("chunks.content", "chunks.sort_order", "chunks.type", "documents.title")
  ->join("documents", "chunks.document_id", "=", "documents.id")
  ->orderBy("chunks.sort_order")
  ->nearestNeighbors("embedding_768", $embedding, Distance::Cosine)
  ->limit(3)
  ->get()
  ->map(function ($chunk) {
    return "File Name: " .
      $chunk->title .
      "\n" .
      "Sort Order: " .
      $chunk->sort_order .
      "\n" .
      "Type: " .
      $chunk->type .
      "\n" .
      $chunk->content;
  })
  ->implode("\n\n");
```

By tagging different parts of your content (like headers, tables, or images), you're essentially giving your AI a roadmap to navigate the information.

Let the LLM do the work for us as we then fix our prompt to ask it to consider types and what to do with that information. Maybe something like this:

Figure 179. Update Prompt to consider Types

```
You are an assistant with our collection of data about Pickleball. Do not answer que\
stions outside the content given. If you do not know the answer due to a lack of inf
ormation ask the user to rephrase their question. The results will include File Name
 and Sort Order. Please reference the File Name in your results and make a list of F
iles and sort order as citations in Markdown at the bottom of the results. There is
a Type attribute as well to help you give weight to the type of data for example, Ta
ble, H1, H2, H3, and so on.
```

While chunking files and throwing them at your LLM can work, taking the time to add a bit of structure to your data can seriously level up your game. It's not just about making your data look pretty – it's about giving your LLM the context it needs to return more concise results. By tagging different parts of your content (like headers, tables, or images), you're essentially giving your AI a roadmap to navigate the information. This means more accurate responses, better handling of complex documents, and the ability to squeeze more value out of those precious tokens. It is a funny balance, though, since as a developer, I really really want to structure my data, but almost like the use cases between NoSQL and SQL, I have to consider what the LLM can do for me and what I have to do for it.

Beyond RAG

I want to end this chapter by discussing going beyond RAG. First, there are so many use cases for vector searching that it would be a waste to associate it only with RAG. For example, you can share with a user "similar searches," or you can use it to find "similar content," "recommendation system," "anomaly detection in time series data," and more.

Lastly, people and businesses will want to do more than just chat with their data. Though RAG makes a great foundation for so many possibilities for users and their data, any RAG system can be a hub for numerous other use cases, or what I call a Content Retrieval System or CRS. Creating a system that can vectorize incoming data is excellent and makes a foundation for many other possible use cases. On that note, I was using NotbookLM[13] by Google, which is essentially a RAG system. But then they took it so much further, "generate a podcast", "create a study guide", and more. You can see how easy it is to set up this foundation for any data store you or your client might have.

What You Learned

The Basics of RAG Systems: Retrieval-Augmented Generation combines the power of large language models (LLMs) with the precision of information retrieval from external databases, helping

[13] https://notebooklm.google/

to overcome the limitations of LLMs by grounding their responses in accurate and up-to-date information.

The Importance of Vector Databases: Vector databases are crucial for RAG systems because they enable semantic search, allowing users to find related information even when the search terms don't exactly match the document content. Traditional keyword-based search engines often fall short in this area.

Chunking and Embedding: These are essential steps for preparing your data for a RAG system. Chunking divides large documents into smaller, more manageable units, ensuring that the LLM receives the most relevant context for a given query. Embedding transforms text into numerical vectors that capture the meaning and relationships between words, making it possible to measure semantic similarity.

Using LLMs for Embedding and Response Generation: LLMs are used in two key ways within a RAG system. First, they generate the embeddings used to represent text in the vector database. Second, they process retrieved information and user queries to generate coherent and informative responses.

Going Beyond Simple Text: The source shows how to incorporate images and audio data into a RAG system. Techniques such as extracting text from images using optical character recognition (OCR) and transcribing audio using speech-to-text models extend the capabilities of the RAG system beyond traditional text-based information retrieval.

The Power of Structured Data: Structuring your data improves the RAG system's accuracy and efficiency by adding semantic tags or metadata, guiding the LLM to better understand the content and provide more relevant results.