

# Course Project: Question Answering System

CS480: Database Systems

Fall 2025

## 1 Overview

In this project, you will design and implement a **document question-answering system** that combines two major parts:

1. **Relational Database Part:** You will design entities, relationships, and schemas for managing different entities of the system, like users, roles, documents, and logs. You will create an ER diagram, translate it into a relational schema (SQL), and implement CRUD operations on that.
2. **Vector Database Part:** You will implement a pipeline similar to **Retrieval-Augmented Generation (RAG)**.<sup>1</sup> Specifically, you will process unstructured documents by chunking them, generating embeddings, building vector indices, and running top- $k$  retrieval queries. A language model will then generate answers with citations drawn from the retrieved passages.

### 1.1 Project Goals

- Learn ER diagram design and schema creation.
- Implement CRUD over a relational DB.
- Gain hands-on experience with document embedding vectors.
- Using popular vector indices (like IVF or HNSW).
- Generate answers with citations from retrieved passages.

### Questions?

If you have any questions, feel free to contact the TA (Mohsen: `mdehgh2@uic.edu`) or visit during office hours on Wednesdays from 3:30–5:30 pm in CDRLC 2402.

## 2 Timeline & Phases

The project will be completed in several phases. Each phase has a deliverable that must be submitted on the due date. Late submissions will follow the course late policy.

---

<sup>1</sup>What is RAG? See this blog post.

## Phase 0: Teams (Deadline: Monday, September 15)

- Select a dataset (containing approximately 10-30 documents.<sup>2</sup>)
- Submit the following:
  1. Description of the dataset and its source.
  2. List of team members (at most 3).
  3. Chosen team name.
  4. GitHub usernames of all members.

## Phase 1: ER Model (Deadline: October 5th)

- Create an ER diagram for the system.<sup>3</sup>
- The diagram should clearly include all required entities and their relationships.
- Submit screenshots or image files of the ER diagram.

## Phase 2: Relational Schema (Deadline: October 22nd)

- Make sure to follow the requirements specified in Section 3 for your ER model.
  - You can still change/finalize your ER model in this phase.
- Translate the ER model (submitted in the previous phase of the project) into a relational schema, implemented as an SQL script using PostgreSQL's dialect.
- Submit just a `.sql` file that generates the relational schema.

## Phase 3: Vector Pipeline (Deadline: November 17th)

1. Chunk the textual data from your dataset ( $text \rightarrow chunks$ ).
2. Embed the chunks into vector representations ( $chunks \rightarrow vectors$ ).
3. Store the vectors into a vector database ( $vectors \rightarrow vectorDB$ )
4. Answer a query by retrieving the nearest-neighbors in the stored vectors ( $query$ ).
5. Commit all the code to the team's GitHub repository.
  - The *last commit* before November 17th (11:59 p.m.) is considered for grading this phase.
  - Look at *Section 4* for the details of deliverables.

## Phase 4: Full System Integration (Deadline: December 4th)

TBD!

---

<sup>2</sup>If you choose a larger dataset, you may work with a smaller sample.

<sup>3</sup>See the following section: *Entities and Requirements*.

## 3 Relational Database

### 3.1 Entities

Design an **ER diagram** and the corresponding relational schema, including the following entities:

- **Users** – Represents all system users, storing attributes such as `id`, `name`, and `email`. Each user is uniquely identified by the `id`. Users fall into three roles:
  - **EndUser** – submits queries to the system. For each EndUser, also store the timestamp of their most recent activity.
  - **Admin** – responsible for managing and overseeing user accounts.
  - **Curator** – responsible for adding, updating, and deleting documents.
- **Document** – stores metadata for each document, including `id`, `title`, `type`, `source`, `added_by`, and `timestamp`. Also, for each document, record whether it has been processed by the Vector Pipeline.
- **QueryLog** – records details of user queries, such as the query text, the issuing user, timestamp, and the IDs of the retrieved documents.

### 3.2 Requirements

The system should support the following functionalities for different user roles.

#### 3.2.1 General

All users must be able to log into the system using a username and password (credentials are created when the user account is generated).

#### 3.2.2 Admin

Admins can perform full CRUD operations on users: create new users, view a list of all users, update user information, and delete user accounts.

#### 3.2.3 Curator

Curators can perform CRUD operations on documents. However, a curator may only update or delete documents they originally created (not those of other curators).

#### 3.2.4 EndUser

EndUsers can submit queries (i.e., ask questions to the system). The system should use the Vector Pipeline to retrieve relevant documents and return the top- $k$  most relevant results. Each query must also be recorded as a **QueryLog** entity.

## 4 Vector Database (100 pts)

### 4.1 Step 1: Text → Chunks (25 pts)

In this step, you will transform your selected textual dataset into chunks of **fixed size**. Refer to the in-class example available in the project documents repository (see the `demo` directory for reference).

Implementation should be done in a **stand-alone project**—not within a notebook. You may use Python or any programming language of your choice.

You can try different chunk (window) sizes to identify which configuration yields the best results in the final **query** stage.

**What to Deliver?** In your GitHub repository’s `README.md`, you must clearly indicate where in your code the chunking logic is implemented. Refer to the exact source file (e.g., the specific `.py` file or a function in your code) responsible for this functionality.

Additionally, your `README.md` should include an example that shows how to execute this part of your code (i.e., how to perform the chunking). For example, the specific commands to run.

### 4.2 Step 2: Chunks → Vectors (25 pts)

In this step you convert the previously formed chunks into **vector embeddings**. Each chunk is represented as a vector in a high-dimensional space. You may use a simple embedder or a more advanced model (see the `demo` or Section 6 for reference). An embedder is simply a black-box function that maps text → vector. Try a quick search to find an embedder and integrate it into your pipeline.

**What to Deliver?** In your `README.md`, specify the exact location in your code where the embedder is used or called. Clearly point to the relevant file (for example, the specific `.py` file) and describe it.

### 4.3 Step 3: Vector → VectorDB (25 pts)

In this step, you will store the generated vectors in a **vector database (Vector DB)**. You may choose any vector database of your preference or refer to the suggestions in Section 6. Typically, you can use a Python package that handles the storage internally—using a separate vector database server (similar to what you did for PostgreSQL) is optional and not required.

For simplicity, you may use an in-memory vector database, where vectors are stored temporarily in memory without persisting them to disk. This phase is often referred to as **indexing** in some vector database packages.

**What to Deliver?** In your `README.md`, clearly indicate where in your code the vector database is used. Point to the specific location (e.g., the function or `.py` file) where you pass the vectors to the database for storage or indexing.

### 4.4 Step 4: Query Answering (25 pts)

In this step, given a user’s textual query, you will retrieve the most relevant data (i.e., the vectors corresponding to similar text) from the **vector database**. This process is known as **nearest neighbor search**. Different vector databases provide multiple algorithms for performing such searches—you should experiment with *at least two of them* to retrieve the most relevant results.

As output, return the original text associated with the retrieved vectors. To enable this, you must maintain a mapping between text chunks and their corresponding vectors (e.g., stored in memory).

You should implement a function that takes the user's query (text or question) as input and returns the text chunks most relevant to that query.

**What to Deliver?** In your `README.md`, include the relevant code snippet demonstrating this functionality and explain how to run it. Specify the exact function or file where this retrieval logic is implemented.

#### 4.5 Step 5: Bonus Part (20 pts)

As an optional extension, you may use a **Large Language Model (LLM)** to generate a well-written response based on the retrieved chunks (or original documents). This step is open-ended and aims to produce a complete, natural-language answer to the user's query.

If implemented, the final output should be a coherent and informative response to the user's question, grounded in the factual content of your dataset.

## 5 Constraints & Tips

- Keep dataset modest to run locally.
- Use only public or course-provided content.
- Ensure reproducibility of your code (in GitHub Repository).
- If using an API, ensure there's a free tier or fallback template-only generator.

## 6 Suggested Tools

- **Relational Database:** PostgreSQL
- **ER Diagram Design:** draw.io, Lucidchart, dbdiagram.io
- **Document Parsing:** pdfminer.six, PyPDF2, textract, readability-lxml
- **Embeddings:** SentenceTransformers (e.g., `all-MiniLM-L6-v2`, `bge-small-en`), Hugging Face API models
- **Vector Index / Vector Database:** FAISS, pgvector (Postgres extension), Qdrant, Weaviate, Milvus
- **Backend / Application:** Python (CLI with argparse or click; minimal API with Flask or FastAPI)
- **LLM for Generation:**
  - OpenAI API (`gpt-4o-mini`, `gpt-4o`, or similar free-tier model)
  - Hugging Face Inference API (e.g., `mistralai/Mistral-7B-Instruct-v0.2`, `tiuae/falcon-7b-instruct`)
  - Local open-source models via `llama.cpp` or `Ollama` (if no API is used)