**Knowledge Navigator - Technical Documentation**

1. Project Overview

The **Knowledge Navigator** is a *Retrieval-Augmented Generation (RAG)* system designed to process books and provide interactive query capabilities. It integrates multiple components for efficient data processing, storage, and natural language interaction. The key features include:

* **Backend**: FastAPI for API and data processing
* **Frontend**: Streamlit for user interaction
* **Database**: Neo4j graph database for knowledge storage
* **AI Integration**: OpenAI GPT models for question answering

**Database:**

As the problem statement resolved around ontologies, which are basically entities and their relationships and also vectorization thus a graph database like Neo4j was good fit. Had it been just vector, the postgres sql with vectorization or any other vector database would have been fine.

**AI Integration**: For critical and more reasoning oriented tasks like ontology creation and final response generation we are using gpt-4o while for simpler tasks and for low token rate scenarios like in case of summary creation we are using gpt-3.5-turbo model. For embedding we are using text-embedding-3-large model.

2. System Architecture

**2.1 Components**

The project is organized as follows:

knowledge\_navigator/

├── project/

│ ├── main.py # FastAPI application

│ ├── streamlit\_app.py # Streamlit UI

│ ├── config/ # Configuration management

│ ├── models/ # Data models and database interactions

│ ├── pipelines/ # RAG processing pipelines

│ ├── services/ # Business logic services

│ └── utils/ # Helper functions

**2.2 Technology Stack**

* **Backend**: FastAPI (Python)
* **Frontend**: Streamlit (Python)
* **Database**: Neo4j graph database
* **AI Models**: OpenAI GPT models
* **Dependencies**: LangChain, Pydantic, Requests

3. Key Features

**3.1 Book Processing**

* Accepts book URLs (e.g., from Project Gutenberg).
* Chunks and processes text content.
* Stores structured knowledge in a Neo4j graph database.

**3.2 Query Interface**

* Provides natural language question answering.
* Delivers context-aware responses.
* Maintains conversation history for better user experience.

4. Installation & Setup

**4.1 Requirements**

Install dependencies from requirements.txt:

bash

openai>=1.0.0

langchain>=0.1.14

neo4j>=5.13.0

fastapi>=0.109.0

streamlit>=1.36.0

**4.2 Configuration**

Set up environment variables in a .env file:

OPENAI\_API\_KEY=your\_openai\_api\_key

NEO4J\_URI=bolt://localhost:7687

NEO4J\_USER=neo4j

NEO4J\_PASSWORD=password

5. API Documentation

**5.1 Endpoints**

* POST /process-book: Processes a book URL and stores its content in the database.
* POST /query: Queries the processed knowledge base to return answers.

6. Development Guide

**6.1 Running the System**

Run the backend and frontend applications:

bash

*# Start FastAPI backend*

uvicorn knowledge\_navigator.project.main:app –reload

You can also directly run the main.py file.

*# Start Streamlit frontend*

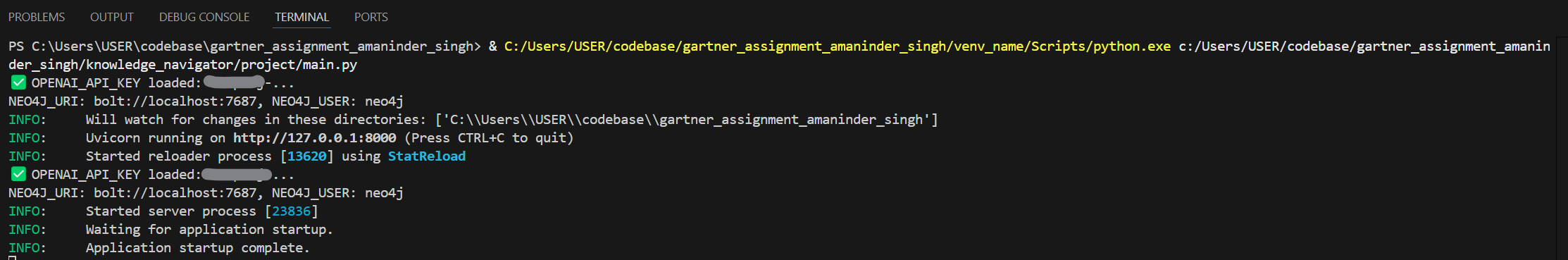
streamlit run knowledge\_navigator/project/streamlit\_app.py

**6.2 Testing**

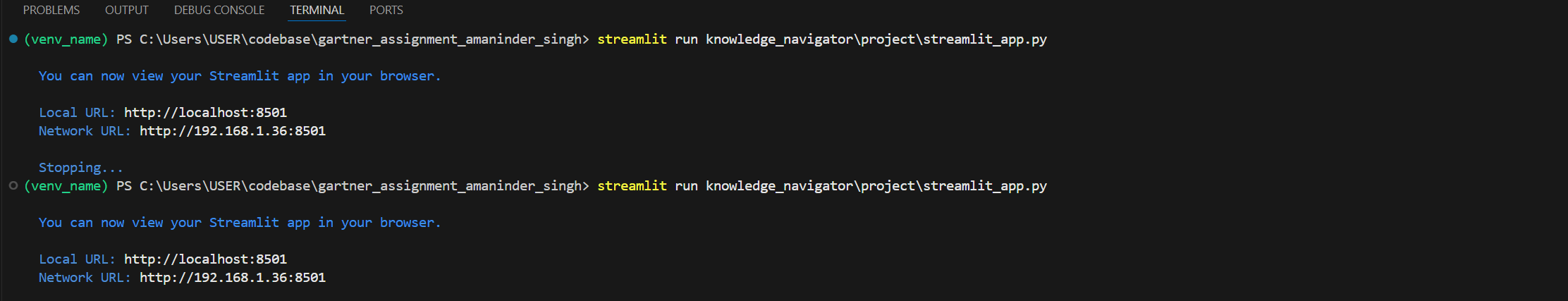
Tested the solution on various book and following are the results.

Book 1: Link <https://www.gutenberg.org/cache/epub/1342/pg1342.txt>

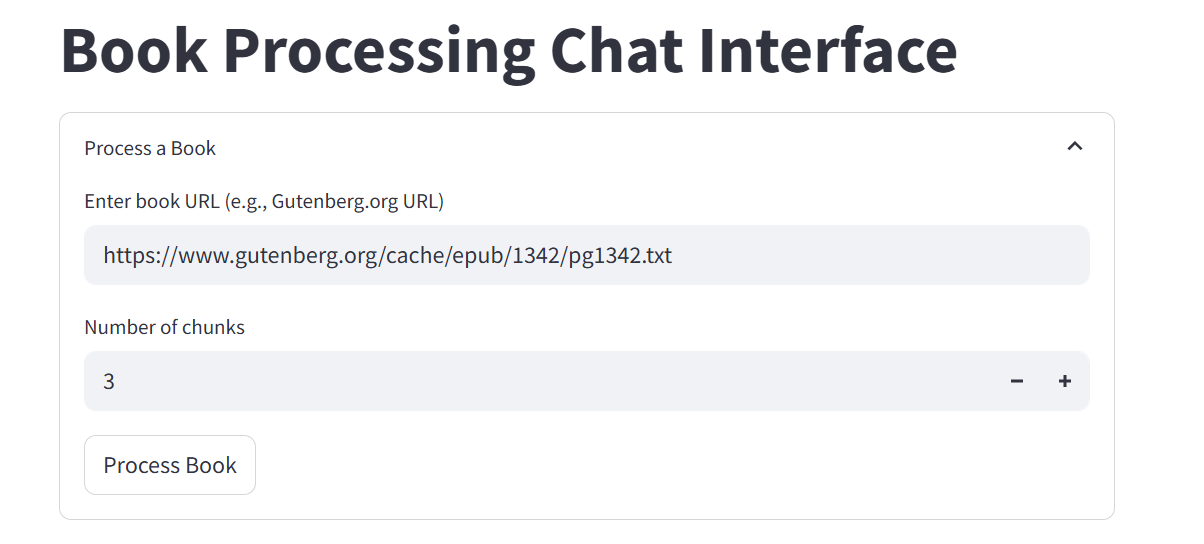
Starting the application:



Starting the stream lit UI:



Stream Lit UI:



This is the default url: <https://www.gutenberg.org/cache/epub/1342/pg1342.txt>

Any link can be added in its place.

The number of chunks to process could configured. This is so testing purpose. The chunks can be increased, but the processing will take time.

We have used LLM prompt for summarizing the chunks and adding together and using that for generating the ontologies and passing the entire book is not possible. Thus, this strategy was opted.

Following is the summarization prompt:

        prompt = f"""

        Summarize the following text in less than 300 tokens:

        {text}

        """

Following is the ontology generation prompt (Joined summary text is provided as input):

        Analyze the following text and extract information in the following structured format:

        {{

          "entities": {{

            "People": [...],

            "Places": [...],

            "Concepts": [...]

          }},

          "relationships": [

            {{"source": ..., "relation": ..., "target": ...}}

          ],

          "themes": [...]

        }}

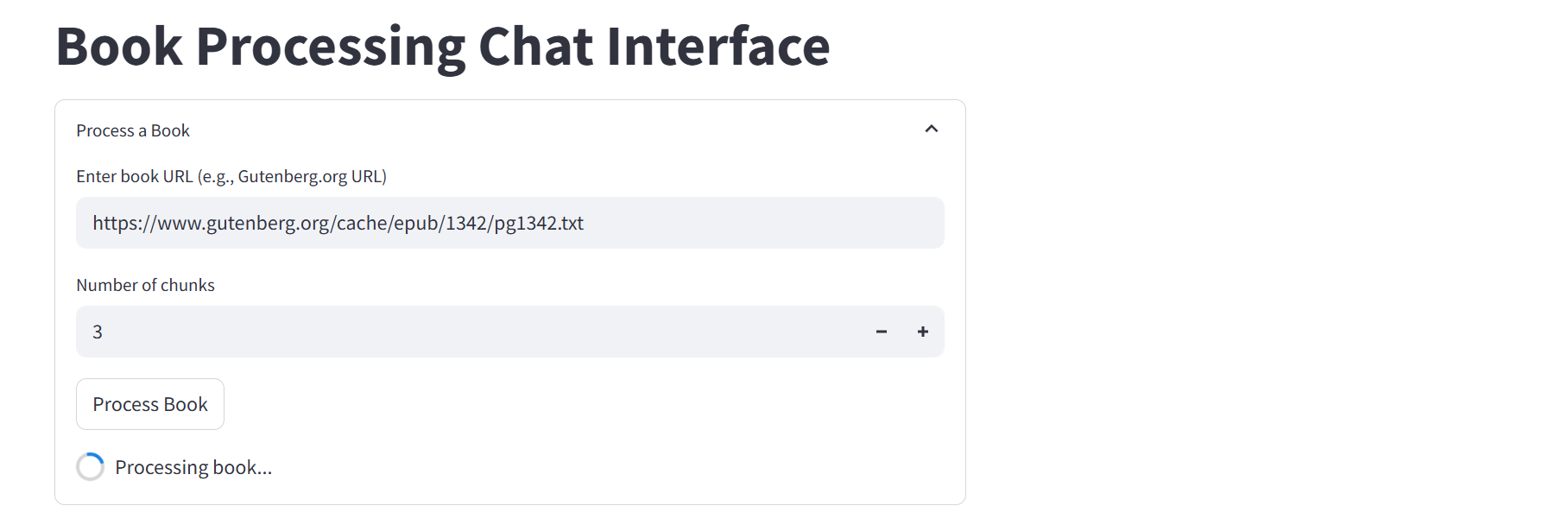
        Ensure output is valid JSON.

        Text:

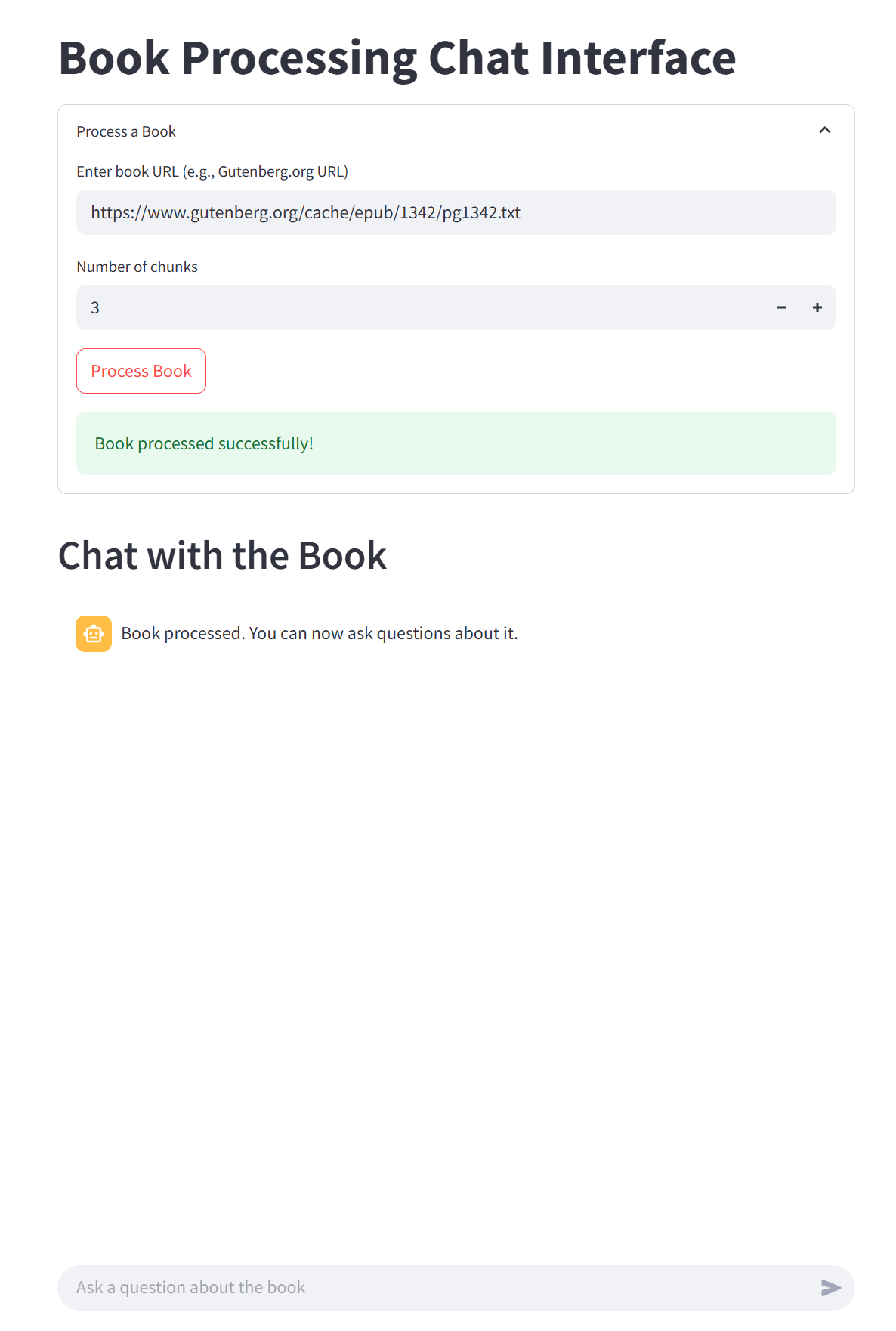
        {input}

        """)

Click on process link:



Once done, it will say processing and the chat window will open:



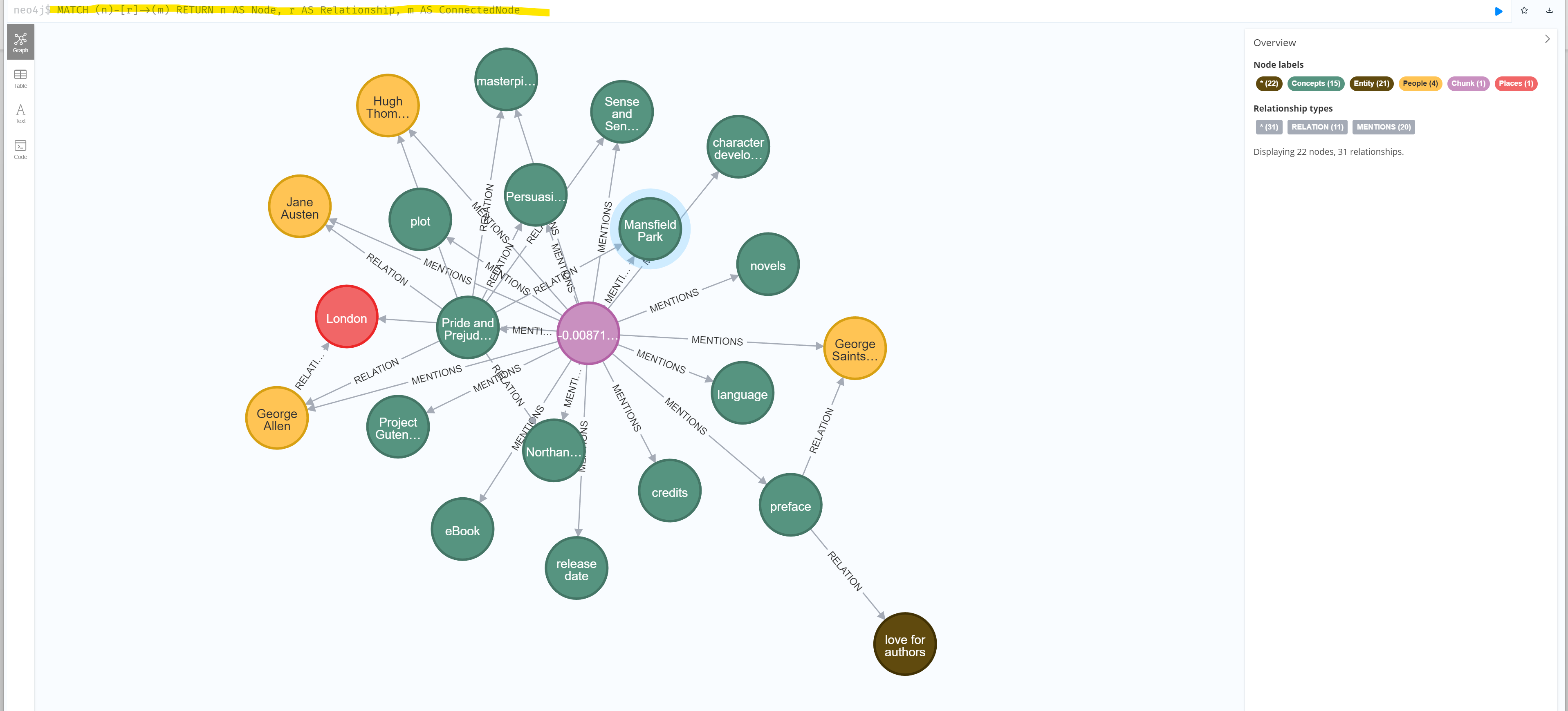
In the neo4j desktop you will be able to the graph has been created with Ontology based relations and also the document chunk has been vectorized and stored.

Use the cypher query to get the full graph:

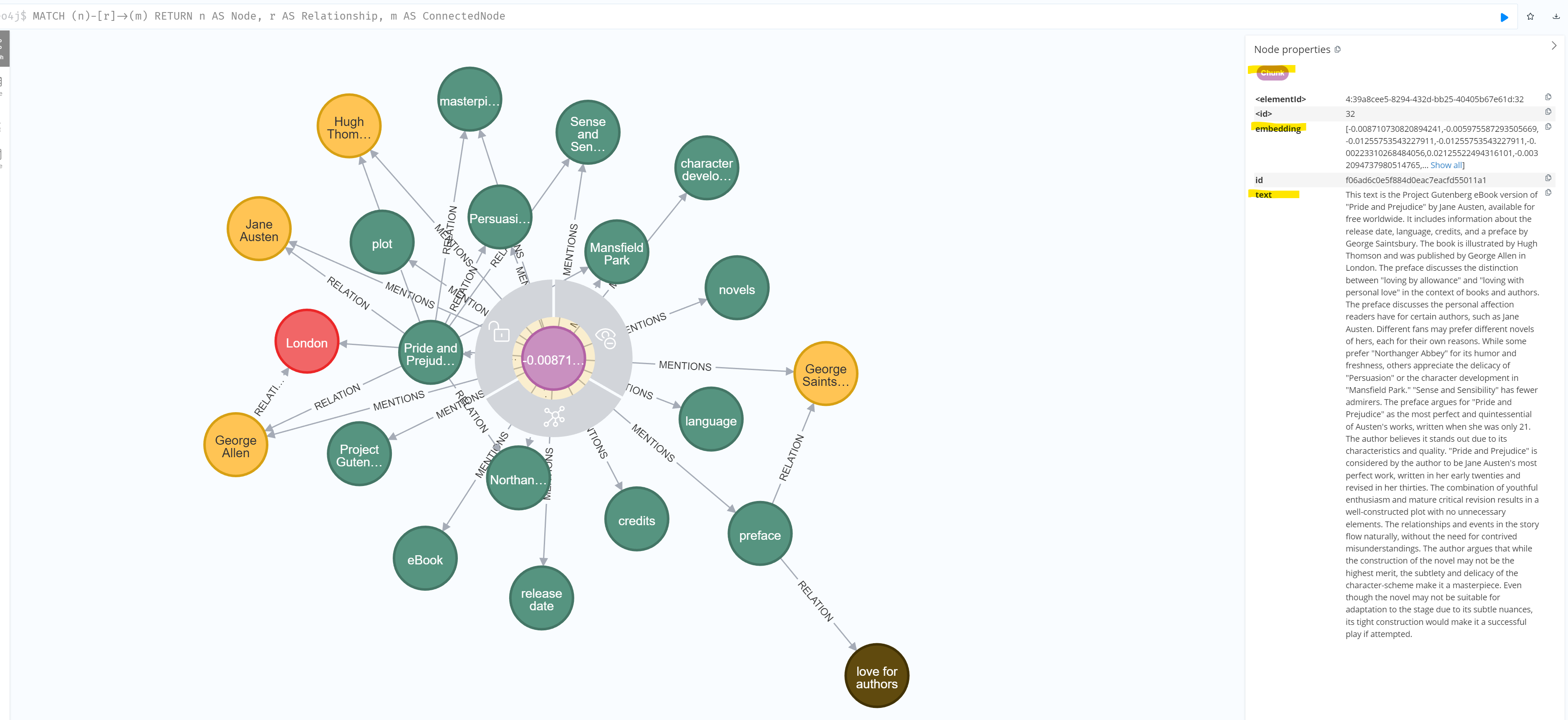
MATCH (n)-[r]->(m)

RETURN n AS Node, r AS Relationship, m AS ConnectedNode

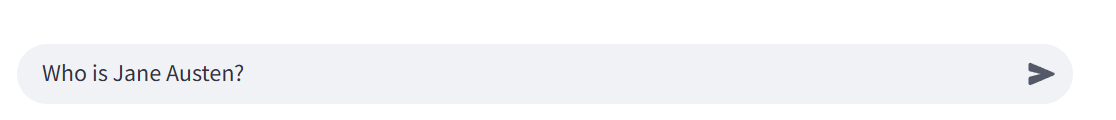
You see a highly connected graph like:



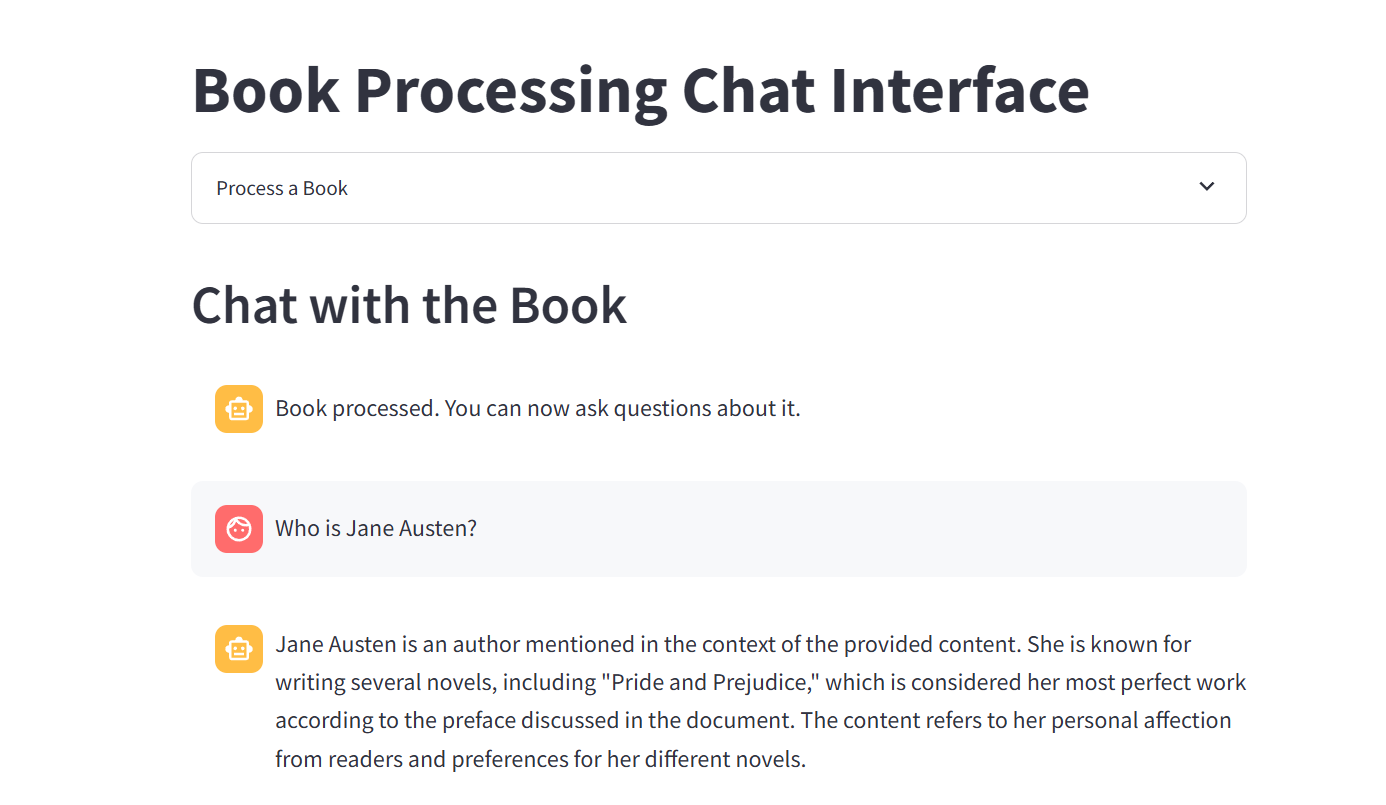
You can also see the chunk, text and embedding.



Next, we ask a question:

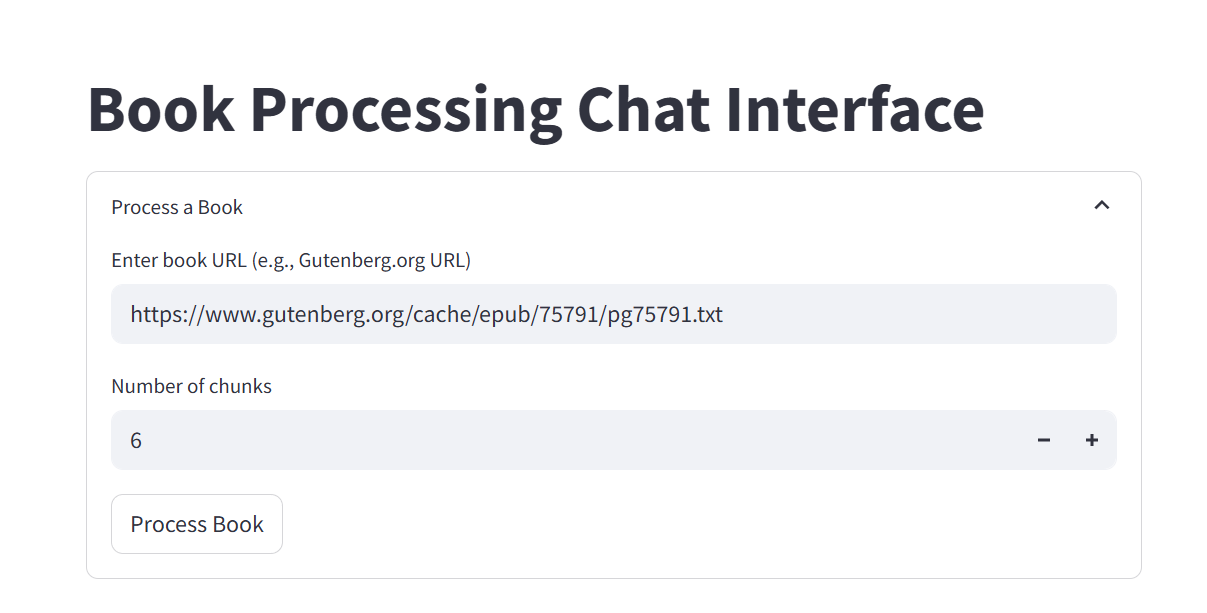


Following is the response we get:

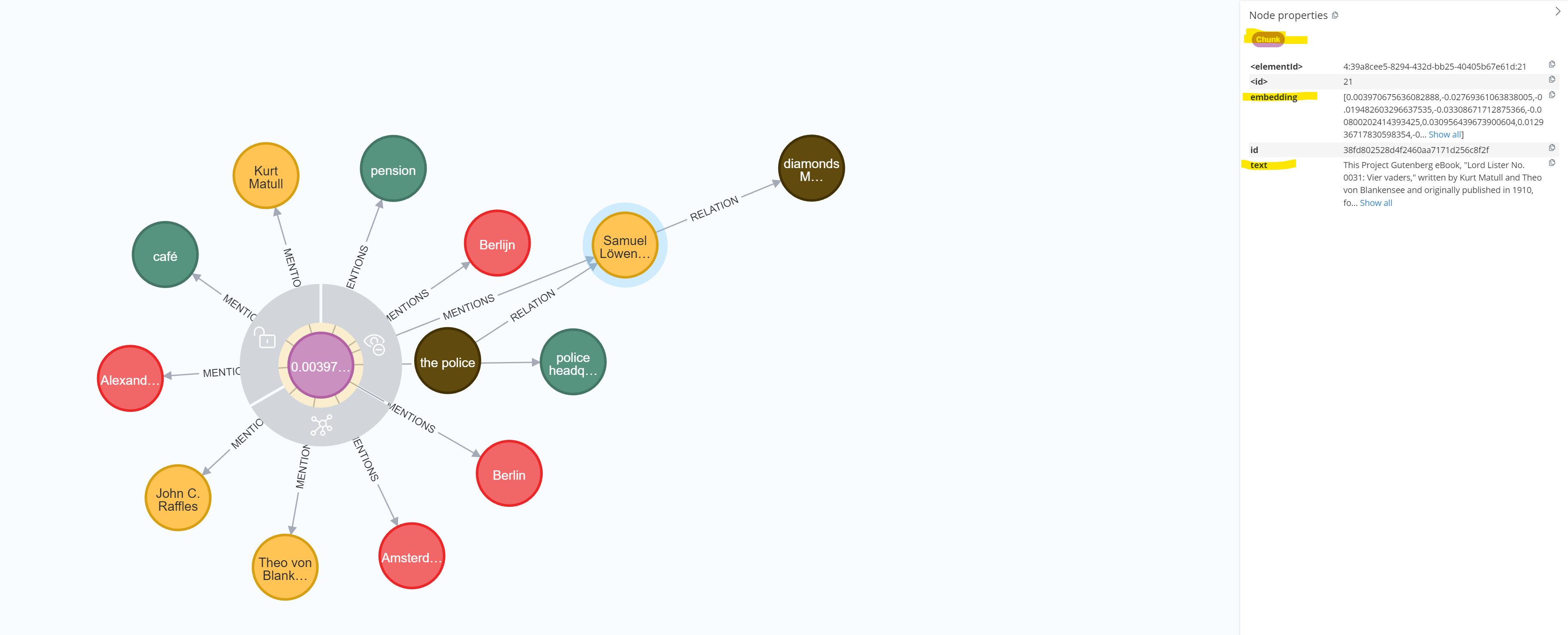


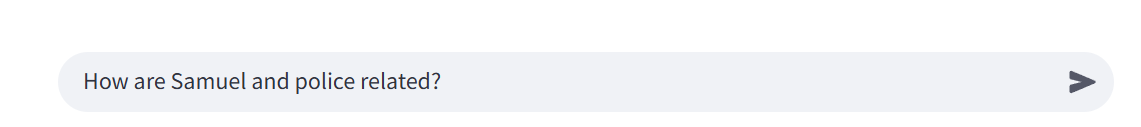
Then we try processing another book link:

<https://www.gutenberg.org/cache/epub/75791/pg75791.txt>

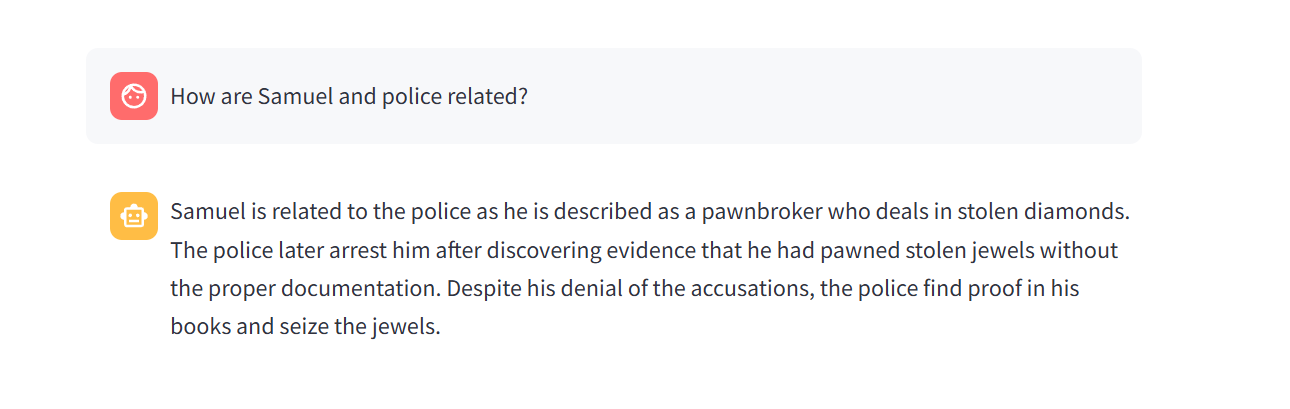


The graph with ontologies, chunk and text is created.





Response:



7. Future Enhancements

Potential improvements include:

* Support for additional document types (PDFs, Word files).
* Advanced text chunking strategies for better processing.