**Project Showcase: Interactive SQL Query App with Gen AI**

Author: Lingam Sravan

**1. Project Overview 📝**

The Interactive SQL Query App is a full-stack web application designed to make data analysis simple and accessible. It empowers users to upload their own datasets (in CSV, Excel, or Parquet format) and gain insights by asking questions in either direct SQL or plain English.

The core innovation lies in its integration with a generative AI model (Google's Gemini), which translates natural language questions into executable SQL queries. This removes the barrier for non-technical users, allowing them to perform complex data analysis without writing a single line of code. The application provides a seamless workflow from data upload to insightful visualization, featuring a clean user interface, dynamic results tables, and interactive charts if necessary.

**2. Key Features ✨**

* **File Upload:** Supports common data formats including CSV, XLSX (Excel), and Parquet.
* **Dual Query Modes:** Users can switch between:
  + **SQL Mode:** For technical users to write and execute direct SQL queries.
  + **Natural Language Mode:** For all users to ask questions in plain English (e.g., "what were the total sales by region?").
* **AI-Powered Translation:** Leverages the Google Gemini API to accurately convert natural language prompts into valid DuckDB SQL queries.
* **Dynamic Data Tables:** Displays query results in a clean, well-aligned, and horizontally scrollable table.
* **Interactive Charts:** Automatically generates visualizations based on the query results using Chart.js.
* **Responsive UI:** A modern and intuitive interface built with React that provides clear user feedback, including loading states and error messages.

**3. Technology Stack 💻**

The project is built with a modern, robust stack, separating backend and frontend concerns.

* **Backend:**
  + **Language:** Python 3.11+
  + **Framework:** FastAPI
  + **In-Memory Database:** DuckDB
  + **Data Manipulation:** Pandas
  + **Generative AI:** Google Gemini API (gemini-1.5-flash)
  + **Server:** Uvicorn
* **Frontend:**
  + **Framework:** React.js 18+
  + **Build Tool:** Vite
  + **Data Visualization:** Chart.js
  + **Styling:** Custom CSS
* **Development & Deployment:**
  + **Environment:** Python Virtual Environment (venv), Node.js (npm)
  + **Automation:** PowerShell scripts for streamlined setup and startup.

**4. System Architecture ⚙️**

The application follows a classic client-server architecture. The frontend (React) is decoupled from the backend (FastAPI), communicating via a RESTful API.

**Workflow Diagram:**

[User] -> [React Frontend] -> [FastAPI Backend] -> [DuckDB]

^ | | |

| | v |

+--------------+ [Google Gemini API] <----+

(Display Results)

**Step-by-Step Flow:**

1. **File Upload:** The user uploads a CSV/Excel file via the React UI.
2. **Backend Processing:** The FastAPI backend saves the file and uses DuckDB to load its contents into an in-memory database table.
3. **Query Request:** The user types a query (SQL or Natural Language) and submits it.
4. **AI Translation (if NL):** If the query is in natural language, the backend constructs a detailed prompt (including table schema) and sends it to the Google Gemini API. Gemini returns a valid SQL query.
5. **Database Execution:** The backend executes the SQL query (either user-provided or AI-generated) against the in-memory DuckDB table.
6. **Data Serialization:** The query result (a Pandas DataFrame) is processed—converting booleans to strings for compatibility—and serialized into a JSON format.
7. **Frontend Rendering:** The React frontend receives the JSON data and dynamically renders the results in a table and a Chart.js-powered chart.

**5. Key Code 🔍**

**Backend: AI Prompt Engineering (main.py)**

This small code shows how the prompt is carefully constructed to guide the AI, ensuring it returns only a valid SQL query.

Python

# From the /query endpoint in main.py

sample\_df = conn.execute(f"SELECT \* FROM {table\_name} LIMIT 5").fetchdf()

cols\_info = "\n".join([f"- '{col}' ({dtype})" for col, dtype in sample\_df.dtypes.items()])

prompt = f"""

You are an expert DuckDB SQL query generator. Your task is to convert a user's natural language question into a valid SQL query for a table named '{table\_name}'.

Table Schema:

{cols\_info}

Instructions:

1. Generate a single, valid SQL SELECT query.

2. \*\*DO NOT\*\* include any explanations, comments, or markdown formatting like ```sql.

3. Return \*\*ONLY\*\* the SQL query.

User Question: "{user\_query}"

SQL Query:

"""

model = genai.GenerativeModel("gemini-1.5-flash")

response = model.generate\_content(prompt)

sql\_to\_run = sanitize\_sql(response.text)

**Frontend: Conditional Styling (SQLChat.jsx)**

The below code demonstrates how the active query mode button is highlighted using conditional CSS classes in React.

JavaScript

// From the return statement in SQLChat.jsx

<div className="chips">

<button

className={`chip ${queryMode === 'nl' ? 'active' : ''}`}

onClick={() => setQueryMode('nl') >

Natural Language

</button>

<button

className={`chip ${queryMode === 'sql' ? 'active' : ''}`}

onClick={() => setQueryMode('sql')}>

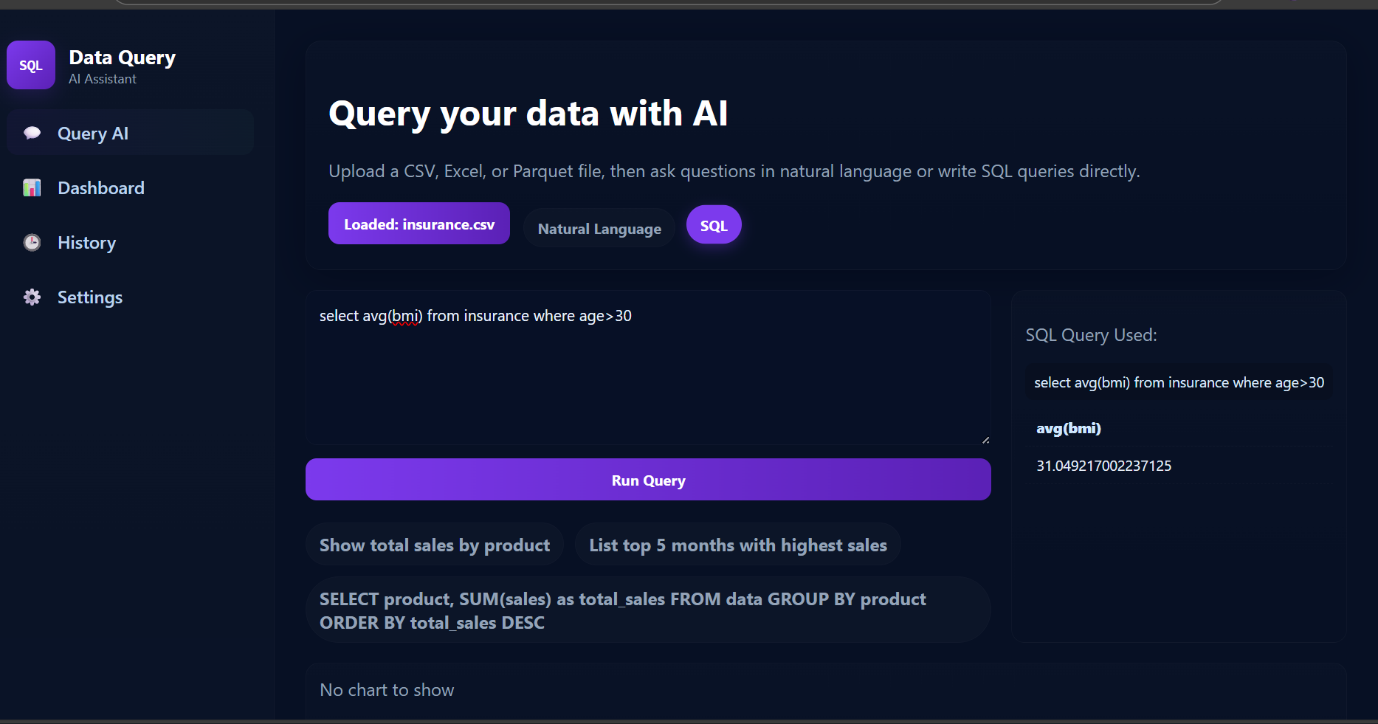
SQL

</button>

</div>

MYProject Snapshots:

SQLMODE:



NATURAL\_LANGUAGE\_MODE:

