Thoughts on Text-to-SQL

Textual interface for talking to Tabular data

Introduction

Text-to-SQL systems enable non-expert users to query relational databases using natural language, greatly democratizing data access. By bridging the gap between human language and precise SQL syntax, they allow business users to extract insights without writing code. However, translating a free-form question into a correct SQL query is challenging: natural language is ambiguous and context-dependent, whereas SQL requires exact schema knowledge and structure. Historically, models needed large labeled datasets of (question, SQL) pairs and careful normalization of both inputs and outputs. The rise of large language models (LLMs) has shifted the landscape: modern LLMs can leverage vast knowledge to generate SQL with far less manual engineering, often without extensive fine-tuning. Still, domain-specific schema (table/column names, synonyms) must be provided as context, and automated SQL execution introduces security considerations (e.g. running only in read-only mode).

Key Challenges

- **Language vs. Schema Ambiguity:** Natural queries may use vague terms (e.g. “sales last quarter”), require complex reasoning, or omit details. Mapping these to a specific schema and SQL structure is hard.

- **Schema Complexity:** Real-world databases often have many tables and columns. The system must know which tables to use and how they join, which is non-trivial across domains.

- **Domain Adaptation:** A model trained on one schema doesn’t generalize easily to another. Even synonyms (e.g. “revenue” vs “sales”) must be handled via retrieval or metadata.

- **Data Ingestion:** Extracting tabular data from documents (PDFs, reports) is difficult. Specialized tools (Camelot, Tabula, PDFPlumber) are often required to detect and parse tables correctly.

- **Security and Correctness:** Generated SQL may be erroneous or unsafe. Frameworks must sandbox execution, use restricted DB roles, and include feedback/correction loops.

Modern Approaches

1. **RAG-based Generation:** Store domain knowledge (schema descriptions, synonyms, sample (NL, SQL) pairs) in a vector database. At query time, retrieve relevant context and feed it to an LLM to synthesize SQL. This “retrieve-then-generate” paradigm (RAG) provides the model with tailored clues, improving accuracy. Open-source frameworks like **Vanna** use exactly this strategy: you first “train” a RAG model on your data (essentially indexing Q–SQL examples and metadata), then ask questions to generate SQL. Advanced variants like *Function RAG* even convert frequent question–SQL pairs into callable templates, making generation deterministic and secure.

2. **LLamaIndex / Structured Indexing:** Use tools like **LlamaIndex**’s SQL interfaces. For example, LlamaIndex lets you wrap an SQLite/SQLAlchemy engine with a `SQLDatabase` object and then use a built-in *NL2SQL query engine* that automatically retrieves relevant tables/columns and prompts an LLM for SQL. LlamaIndex also supports *table and row retrieval*: it can index each table’s schema and sample rows so that, at query time, it only supplies the pertinent table schemas or example rows to the LLM. These modules save much manual prompt engineering.

3. **Other Open-Source Systems:** In addition to LlamaIndex and Vanna, other options include LangChain-based SQL agents, GPT4All, or Bloomberg’s open-source NL2SQL models. Frameworks like LangChain can combine multiple tools (LLM, retriever, SQL engine) under an **agent**. For example, a *router* agent can inspect the query and decide whether to invoke the RAG module, a direct SQL-generation LLM, or a vector-search fall-back. LlamaIndex even provides a `RouterQueryEngine` that learns to pick the best sub-engine for each question.

Proposed Architecture

Figure: Example RAG-based Text-to-SQL architecture (adapted from AWS). We propose a hybrid pipeline combining RAG, LlamaIndex, and open-source tools. Incoming documents with tables and CSVs are first ingested into a **SQLite** database (or another open-source DB) and indexed. A text processing layer uses libraries like Camelot/Tabula to extract tables from PDFs into DataFrames, then writes them into SQLite. All tables’ schemas and optionally sample rows are collected. A vector store (e.g. FAISS, Chroma, Weaviate) holds embeddings of schema descriptions, column names, synonyms, and example Q–SQL pairs (from training data).

When a user submits an English query, a *router agent* first classifies the intent or tries multiple methods in parallel. For queries involving aggregates or known patterns, the agent can route directly to a text-to-SQL engine (LlamaIndex or a specialized LLM prompt), which uses the database schema context to generate SQL. Simultaneously, the query is embedded and passed to the RAG retriever: relevant schema and example queries are pulled and appended to the

prompt. The chosen LLM (open-source like LLaMA-2, Mistral, etc., or commercial via API) then generates a SQL statement. This SQL is executed on the SQLite DB, and the results are fetched. Optionally, a final LLM pass can convert the raw results into a polished natural-language answer. This modular design (ingest → retrieve → generate → execute) ensures high accuracy and maintains context.

Data Ingestion and Storage

- **Table Extraction:** Use Python libraries to extract tables from documents. *Camelot* and *Tabula* excel at simple table layouts, outputting CSV/JSON. *PDFPlumber* is useful for more complex or irregular tables. An emerging tool, *LLMWhisperer*, can also convert PDFs to layout-preserving text, which LLMs can interpret directly.
- **Loading into SQLite:** Parsed tables and CSV files are loaded into SQLite. For example, using Pandas:

```
import pandas as pd, sqlite3
df = pd.read_csv("data.csv")
conn = sqlite3.connect("mydb.db")
df.to_sql("table_name", conn, if_exists="replace", index=False)
```

- This centralizes all data in one relational store. As demonstrated by LlamaIndex examples, one can iterate through CSV files, create a SQLite database, and then build an index of each table's schema for retrieval.

- **Indexing Metadata:** Create vector embeddings of each table's schema description, column lists, and any human-readable doc text. Also embed example Q–SQL pairs (from domain training data) into the vector store. Tools like Vanna support many stores (FAISS, Chroma, Qdrant, etc.). The SQLite database itself is wrapped by an indexing tool (`LlamaIndex.SQLDatabase`, `Vanna`'s DB connector) so that it can be queried by the system. Ingested data should include up to ~100 tables or more (per assumption) to cover all sources.

Text-to-SQL Modules and Techniques

- **RAG with Example Queries:** We index our corpus of (English question, SQL query) pairs. At runtime, the user's query is embedded and similar training examples are retrieved. These examples form in-context shots or function templates for the LLM. Vanna's approach is illustrative: it **"trains" a RAG model on your data** by storing this metadata and then answers new questions by returning SQL. In practice, the system might retrieve table schema info *and* one or two example Q–SQL pairs that match the query intent. The prompt to the LLM then includes the natural question, the schemas of relevant tables, and the example pair(s). The LLM outputs a SQL query consistent with

these templates. This method often yields more accurate and consistent SQL than few-shot alone, since it effectively "remembers" past solutions. For stricter control, **Function RAG** can be used: it converts frequent Q–SQL patterns into callable templates (with parameters) so the LLM only fills slots rather than freely generating text.

- **LlamaIndex SQLQueryEngine:** We use LlamaIndex's built-in SQL tools to handle straightforward queries. The `SQLDatabase` class (built on SQLAlchemy) connects to our SQLite instance, automatically reflecting table schemas. We then use its query engine (e.g. `SQLTableRetrieverQueryEngine`) which takes a natural language prompt and internally retrieves relevant table names/rows before calling the LLM. For example, one can do:

```
from llama_index.core import SQLDatabase

from llama_index.llms.openai import OpenAI

engine = create_engine("sqlite:///mydb.db")
sql_db = SQLDatabase(engine, include_tables=["sales",
"customers"])
llm = OpenAI(temperature=0)
query_engine = sql_db.as_query_engine(llm)
answer = query_engine.query("Total sales in 2024?")
```

- This engine handles the "create table" schema description and few-shot prompting under the hood. We can further enhance it by using query-time table and row retrievers to add specific schema snippets or example rows as context.

- **Hybrid Routing Agent:** A *RouterQueryEngine* or LangChain agent sits in front. It can analyze the query to choose between: (a) running a direct SQL generation engine (if schema is straightforward), (b) engaging the RAG pipeline (if query is complex or ambiguous), or (c) falling back to a general QA over the indexed documents. LlamaIndex's router modules allow defining multiple "tools" and selecting one per query. In our design, for example, routine count/sum queries on known tables might use the SQLTableQueryEngine, while unusual or multi-table queries trigger RAG retrieval first. This ensures robustness and efficient use of context.

Technical Stack (Open Source)

- **Database:** SQLite (embedded, zero-config) as the main store. Supports SQL operations and easily integrates with Python. Can scale to hundreds of tables and is scriptable via SQLAlchemy.

- **Data Extraction:** Python libraries: *Camelot*, *Tabula*, *pdfplumber* for PDFs; *Pandas* for CSV/Excel; *LLMWhisperer* for layout-preserving text if needed.

- **Vector Store:** FAISS, Chroma, Qdrant, etc., for indexing embeddings. Vanna and LlamaIndex support FAISS/Chroma out of the box. We embed table metadata and example Q&A.

- **LLM:** Open-source or cloud LLM (e.g. Meta LLaMA2, Mistral, GPT-4o, or similar). The system should be LLM-agnostic. We can interface via HuggingFace or local inference libraries. Vanna supports many backends (OpenAI, Anthropic, HuggingFace, Ollama, etc.).

- **Frameworks:**

    - **LlamaIndex:** For connecting LLMs to the SQL database and doing retrieval-based query processing.

    - **Vanna.ai:** For easy RAG-based SQL generation.

    - **LangChain / Agents:** If we build a custom multi-tool agent to orchestrate querying.

    - **Streamlit/Flask:** For a simple front-end, if desired (AWS example used Streamlit).

- **Language/Tools:** Python 3, `sqlalchemy`, `sqlite3`, `pandas`, `llama-index` (GPT Index), `faiss`/`chromadb`, and any desired LLM inference library.

Example Workflow

1. **Ingestion:** PDF report ⇒ Camelot ⇒ CSV ⇒ `df.to_sql()` into SQLite (using Pandas). Repeat for all tables.

2. **Indexing:** Run LlamaIndex's table parser to extract schema text and insert into a vector index, alongside embeddings of (table name, column names) and sample rows. In parallel, take the QA training set of (question, SQL) and index those examples in the same vector store (each example can include the SQL and natural language).

3. **Query-Time:** User asks "How many users signed up last month?" The router embeds this query and finds the top-matching context: perhaps the "users" table schema and a similar past question. The prompt to the LLM then includes: the query, the Users table schema (columns), and one example Q–SQL pair. The LLM outputs: `SELECT`

`COUNT(*) FROM users WHERE signup_date >= ...`. The system executes it on SQLite and returns the count. Optionally the LLM is asked to explain it in plain language.

Conclusion

The proposed Text-to-SQL solution integrates the strengths of RAG-based retrieval, LlamaIndex's structured SQL query capabilities, and open-source tools like Vanna into a unified, agent-driven architecture. By centralizing extracted tabular data from diverse document formats and CSVs into SQLite, enriching it with schema-aware metadata and example Q–SQL pairs, and intelligently routing queries to the most suitable processing path, the system delivers accurate, contextually relevant answers in plain English. This hybrid approach not only addresses the core challenges of schema complexity, domain adaptation, and ambiguity in natural language but also ensures flexibility, scalability, and maintainability through modular design. Leveraging mature open-source technologies and LLM frameworks reduces implementation cost and promotes transparency, while the routing agent enables extensibility for future models and retrieval techniques. In sum, this architecture positions the organization to unlock the full value of its tabular data assets by making them accessible to anyone without requiring SQL expertise, while maintaining control, security, and performance.