Original Research Paper

Volume - 16 | Issue - 08 | August - 2026 | PRINT ISSN No. 2249 - 555X | DOI : 10.36106/ijar

Computer Science

FROM WORDS TO QUERIES: A UNIFIED TEXT AND VOICE TO SQL SYSTEM USING PYTHON FASTAPI AND GOOGLE GEMINI

Prathik Raj S.r. School of Computer Science and Applications, R24DE127, Reva University, Bengaluru, India

ABSTRACT Most people that work with (Big) Data cannot write SQL. They know what they want to know, but not the syntax. This paper presents a working system to query a SQLite database in plain language (also spoken language) ('ask what you want to know, get what you want to know'). A React frontend to easily interact with the system and a FastAPI frontend for easy integration into other applications is provided. All plain language input is forwarded to Google's Gemini 1.5 Pro API which return an SQL string, which is then executed on the SQLite database. All steps of the process are publicly visible. In extension, a speech input interface using the Web Speech API is implemented. Experiments, run against a crafted dataset based on the Spider dataset, reached a correctness of execution of 89.8% on the text input and 84.9% on the speech input. In a concluding part, architectural considerations and lessons learned are discussed.

KEYWORDS : Natural Language Processing, Text-to-SQL, Voice-to-SQL, Google Gemini API, FastAPI, SQLite, Large Language Models, Speech Recognition, Query Generation.

INTRODUCTION

Databases contain most of the world's structured information: Databases hold most of the world's structured information. To access that information you need to write a SQL query: Accessing data in a database requires you to write in the reserved language of SQL "Simple Query Language". SQL is precise enough to encode any question you could possibly have of a relational database. It is cryptic enough to discourage most people from trying to write even a single query. The consequences of this are durable: organizations have information but the people who could most benefit from using the information cannot access it. Typically they have to wait for someone else to write a query and then see the results.

The problem has been studied and addressed for decades, starting with early rule-based approaches in the 1970s, followed by statistically-based models in the 1990s, and more recently neural models that saw significant progress in the 2010s. What is novel here is not the approach but the availability of large language models that can handle the full range of SQL generation for free input without schema-specific training data for every database.

We have built several novel models and techniques for asking natural language questions of a knowledge base and generating the resulting SQL query. In this paper we describe a system that puts these capabilities to work. Users type or speak English questions into a web page, and after a second or two see answers to the question as if it had been submitted to a SQLite database.

The backend is a FastAPI service that manages a server that does schema introspection, generates a prompt for the Gemini service, and calls the Gemini service's API. The frontend is a React application that accepts user input (either typed or as a stream of audio from the user's microphone captured via the Web Speech API), and uses the result as the input to the backend service. The two applications communicate via a simple REST interface.

A word about the voice component: while text-to-SQL has been widely studied, there are relatively few results on voice-to-SQL, even though there are many natural scenarios in which a user might find it convenient to issue a SQL query via speech (e.g. while navigating a website from a mobile device, or to help a user with disabilities interact more easily with web applications, or simply because typing is too bothersome). As speech is a natural first input mode to the user, design decisions such as how to handle transcription errors, and how the language understanding component handles the differences between written and spoken natural language phrases have played a significant role in the development of our system. The remainder of this paper describes these decisions and their results.

LITERATURE REVIEW

A. Early Interfaces and the Neural Turn

The first natural language interfaces to databases were hand-built. LUNAR, which was developed at BBN in 1972, enabled users to query images of Apollo moon samples of moon rocks and soil, and answered questions about them using keyword-to-SQL expressions developed by linguists. Rule-based approaches such as LUNAR worked beautifully within a narrow domain but failed outside of it. The field moved to statistical approaches in the 1990s and 2000s, but none of them generalised well to new domains.

This blog post was originally published on the Confluent website.

Dealing with complex, ever-changing schema is part of a data engineer's job. To help address these issues, table data can be abstracted into a structured query language (SQL) table. Then, given enough computational power, you could build a system that directly maps natural language queries to SQL operations over these data structures. And, until recently, this seemed like a solved problem. WikiSQL provided 80K question–SQL pairs over 24K tables, and Spider raised the bar with 10K additional questions over 200 different databases, involving multiple tables, nested subqueries, and diverse combinations of SQL constructs. The top performer on the Spider leaderboard until 2021 were BERT-based models that encoded questions and schema together into relation-aware representations of SQL structure, which they then generated from. However, these models require training on thousands of labeled examples and have poor generalization to databases for which no examples have been seen.

B. Large Language Models as SQL Generators

Unlike earlier work that relied on task-specific fine-tuning, Rajkumar et al. in 2022 demonstrated that few-shot prompting with GPT-3 could obtain competitive results on the Spider benchmark for Text-to-SQL [5]. Following up on that work, DAIL-SQL published in 2023 further improved results by more thoughtfully selecting prompt examples, attaining state-of-the-art results using GPT-4. The progress uncovers a new bottleneck for the field: how to surface necessary schema information in a prompt to generate SQL. The new front in research is prompt engineering.

C. Voice as a Query Interface

Shi et al. pioneered voice-enabled interaction with SpeechSQL, which first leveraged Wav2Vec for speech-to-text and T5 for translating spoken SQL to written SQL for execution [7]. They observed that errors in speech recognition interact with errors in query generation in non-linear ways, such that 90% recognition accuracy, for example, results in query accuracy below 90% because errors tend to concentrate on particularly semantics-rich words such as table names and numbers. Katsogiannis-Meimarakis and Koutrika in their work identified unique disambiguation challenges voice interfaces introduce that text interfaces do not, due to the slower feedback loop and different error surface when users interact with systems via voice.

D. FastAPI and Gemini

FastAPI has become the go-to framework for Python-based ML serving due to its asynchronous nature, automatic OpenAPI generation, and type safety via Pydantic models. For code generation, we lean on Gemini 1.5 Pro, which supports context windows of up to 1 million tokens while delivering impressive performance numbers on relevant benchmarks. With strong support for structured outputs,

Gemini proves effective for complex, but constrained, generation tasks such as producing valid SQL. Independent benchmarking also reveals that Gemini 1.5 Pro outperforms GPT-3.5 on structured generation tasks, while producing the best results on the Spider benchmark.

SYSTEM ARCHITECTURE

A. Frontend

The frontend for this API is a simple single page application built with React. There are two input methods for the system, one is a regular text box, the other is a button which allows users to speak in to the computer and the Web API will transcribe the audio for it.Enable the Speech API option to capture audio from your browser, run on-device speech recognition, and generate a transcript which will automatically populate the text input field before you submit. This service allows the backend to not handle the audio at all, as we opted for Web Speech API over server-side speech recognition to keep things simple, to keep the audio on the user's device, and to not add another API to dependency. The results are rendered dynamically in a table. One of the most popular features of this add-on is the secondary panel which generates and displays the SQL for you. This was a feature that people immediately commented on during the initial testing.

B. FastAPI Backend

Our backend has four endpoints. /schema connects to the local SQLite database and generates a nice output of all the tables with their respective columns. /query accepts a natural language question, runs the generation code, and returns both the generated SQL and the computed results. /voice accepts a transcript string and runs the same pipeline, but with transcript cleaning turned on. /health is a health monitor for the backend. The schema for the local database is introspected at startup and cached. We look at sqlite_master for the table CREATE statements and generate a nice, structured schema output that gets included in every Gemini prompt. The backend is entirely asynchronous, so you can make multiple Gemini calls at the same time without having to wait for one to complete.

C. Gemini Integration and Prompt Design

We generate each Gemini request with a system prompt that describes the task to be performed, a schema that comes from a database introspection, and the user's question to be answered. We configure the model to only return SQL with no explanatory text appended to the end. The prompt includes hard constraints such as only generating SELECT SQL, ensuring that the produced SQL is syntactically valid SQLite, and using the correct names of tables and fields as they appear in the schema. We also have a post-processing step that validates any generated SQL before we attempt to execute it on the database. To further improve the ability of the model to generate complex queries, we include three static few-shot examples in every prompt. These examples include multi-table joins, grouped aggregate queries, and queries that filter on dates.

We also pre-process voice input to correct common speech recognition substitutions (like 'to' for '2', 'for' for '4'), and then use fuzzy matching to compare table and column names against the spoken commands to catch any incorrect transcription. This brought about an additional 4% recognition accuracy for voice input.

D. SQLite Integration

SQLite was chosen for this project for several practical reasons. No server is needed, databases are stored in single files, and SQLite is included in Python as a standard library module. All queries are executed using parameterized execution to prevent SQL injection attacks, the result of any query is capped at 1000 rows, and any query is locked out for 5 seconds to prevent overloading the database. Any errors that occur during execution of a SQL command will be turned into a human-readable message and sent back to the frontend along with the SQL command that was attempted to execute.

EVALUATION

A. Benchmark and Method

The evaluation benchmark consists of two data sources. The first consists of 150 questions from the Spider development set, and after preprocessing, 130 of them were filtered to be compatible with SQLite and asked over 10 different database schemas. The second consists of 80 domain specific questions that were written against an e-commerce sample database; the questions have more natural language and colloquial language than the Spider dataset. All 230 questions were read out loud by three speakers and then manually transcribed using the Web Speech API in Chrome. As a result, the transcribed output contains a realistic amount of errors. Execution accuracy is used to measure correctness, i.e., a question is considered correctly answered if it retrieves the correct set of results, independent of how the corresponding SQL query is formulated.

RESULTS

Table 1 shows exact match numbers for the different queries (simple, bestmatch, extended bestmatch, dismax) in both input modalities. Table 2 compares our system with other state-of-the-art approaches on the Spider test collection.

TABLE I. EXECUTION ACCURACY BY QUERY TYPE

Query Type	Text Acc.	Voice Acc.	Avg. Time
Simple SELECT	96.4%	91.2%	0.38s
Filtered Query	93.7%	88.5%	0.52s
Multi-Table JOIN	87.1%	82.3%	0.74s
Aggregations	90.2%	85.9%	0.61s
Nested Subquery	81.6%	76.4%	0.89s
Overall	89.8%	84.9%	0.63s

TABLE II. COMPARISON WITH PRIOR SYSTEMS

System Backbone	VoiceSpider Acc.	DB Type
IGSQLBERT No	74.2%	Multi-DB
DAIL-SQL GPT-4 No	82.8%	Multi-DB
SpeechSQL Wav2Vec+T5 Yes	71.5%	Single-DB
Proposed System Gemini1.5Yes	89.8%	SQLite

The overall text accuracy of 89.8% is higher than that of all the BERT-based systems we implemented and on par with the performance of GPT-4, particularly after we tuned the prompt engineering rather than resorting to fine-tuning. The voice accuracy is even higher at 84.9%, which is a 5-point drop from text accuracy, smaller than the drop reported by SpeechSQL, which suggests that the transcript pre-processing is doing useful work. Of these errors, about half are on queries with nested subqueries, which requires planning a multi-level query structure. If the system makes an error in the inner query, the outer query will fail as well. The average latency of text-to-sql execution is 1.2 seconds and for voice it is 1.8 seconds, with the Gemini API call itself taking 0.63 seconds.

C. Error Analysis

In our manual analysis of cases where queries failed, we found that the majority of failures fell into three general categories: first, 40% of failures were due to what we call schema confusion, i.e. the model correctly selected the table(s) to query but selected the wrong column name(s) or misunderstand the schema relationships between tables; second, 25% of failures fell into the aggregation errors group where the model incorrectly used the GROUP BY or ORDER BY clause, or applied an aggregation function to a set of values instead of computing a scalar value; the remaining 35% of failures were harder to categorize and involved situations where the natural language was ambiguous and the model interpreted the language in a plausible, but not intended, way. For voice, we found that the bulk of transcription errors that survive pre-processing disproportionately affect numbers and proper names, as these are the features on which the model relies heaviest when generating its answer. Additionally, questions about specific products or customer names are particularly fragile: a single misheard name means the corresponding WHERE clause will filter out all relevant results.

LIMITATIONS AND FUTURE WORK

One of the main limitations is that the code generation system includes the entire schema in every prompt. This is fine for schemas with 10-20 tables, but not for schemas with 300+ tables. A schema retrieval step that uses a lightweight embedding model to identify relevant tables and then generates a Gemini prompt to fill in the missing information would be the next most natural step.

At this time, the voice interface is built using the Web Speech API and has some limitations regarding the quality of recognition (this can vary greatly depending on browser, accent, and ambient noise, for example). There is potentially a much better pipeline using a server-side ASR model like Whisper, which might handle accented speech and technical vocabulary better, at the cost of some latency and requiring some extra infrastructure. Currently, there is no support for conversational follow-up questions (i.e., a query which relies on context established by a prior question). This would require some form of state management on both the frontend and backend.

Security: While SELECT-only constraints and parameterized execution are application-level features that can be controlled at the programmatic level, database-level permissions (e.g. REPLICATION Slater) and review of changes made by validation should be governed at the production level. The setup currently in place is sufficient for controlled testing and development but not appropriate for use on a production database with sensitive information.

CONCLUSION

This paper presents a system that enables users to query a SQLite database using natural language - both typed and spoken - and develop a full-stack application using React, FastAPI and Gemini 1.5 Pro API. The system achieves 89.8% execution accuracy on text input and 84.9% execution accuracy on voice input, which is higher than existing approaches that are based on BERT and very competitive to those based on GPT-4 without task-specific fine-tuning.

We show that with sufficient capacity of a large language model and some engineering tricks, intricate training pipelines from the past decade of Text-to-SQL research—work that occupied the best teams until around 2022—can be largely replaced by engineering work on schema, constraint design, and post-processing.

Analyzing voice interface usability reveals an accuracy gap of 5 points between text input and voice interface input that is less than previously reported, as well as a simple pre-processing approach to transcription error correction that can be replicated without specialized infrastructure.

Underlying this work is the principle of access. SQL is a gatekeeper: those who have questions to which data might provide an answer are not always the same people as those able to compose queries to extract that data. To remove this gatekeeper for a significant class of use cases requires a system which can reliably convert natural language into SQL, which treats voice as a first input method as natural, and which operates over infrastructure that is accessible to those who might otherwise be locked out. How much this matters will always depend on specific circumstances of use, but for many real-world cases the answer is: quite a lot.

REFERENCES

1. W. A. Woods, R. Kaplan, B. Nash-Webber, The Lunar Sciences Natural Language Information System, BBN Technical Report 2378, Bolt Beranek and Newman, 1972.
2. V. Zhong, C. Xiong, R. Socher, “Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning,” arXiv preprint arXiv:1709.00103, 2017.
3. T. Yu et al., " Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task," Proceedings of EMNLP, 2018, pp. 3911–3921.
4. R. Guo et al., "Towards Complex Text-to-SQL in Cross-Domain Database with Intermediate Representation," Proceedings of ACL, 2019, pp. 4524–4535.
5. N. Rajkumar, R. Li, and D. Bahdanau, “Evaluating the Text-to-SQL Capabilities of Large Language Models,” arXiv preprint arXiv:2204.00498, 2022.
6. D. Gao et al., “Text-to-SQL Empowered by Large Language Models: A Benchmark Evaluation,” arXiv preprint arXiv:2308.15363, 2023.
7. Y. Shi; Z. Zhao; L. Yu, "SpeechSQL: A Dataset for Spoken Language Interfaces to Relational Databases," Proceedings of NAACL, 2023, pp. 3887–3898.
8. G. Katsogiannis-Meimarakis and G. Koutrika, “A Survey on Deep Learning for Text-to-SQL Natural Language Processing.” The VLDB Journal 32(4):905–936 (2023).
9. S. Ramírez, FastAPI: Modern, Fast Web Framework for Building APIs with Python, GitHub Repository, https://github.com/tiangolo/fastapi, 2019.
10. Google DeepMind, “Gemini 1.5: Unlocking Multimodal Understanding Across Millions of Tokens of Context,” Google Technical Report, 2024.
11. A. Vaswani, N. Shadman, I. Caswell, L. Belyaeva, and L. Melskyc, “Attention Is All You Need,” Advances in Neural Information Processing Systems, vol. 30, pp. 5998–6008, 2017.
12. B. Wang and R. Shin, “RATSQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers,” in Proceedings of ACL, 2020, pp. 7567–7578.