An AI Agent-Based Approach for Automated SQL Query Generation and Optimization Using Python

A Multi-Agent Framework Combining Natural-Language-to-SQL Generation, Plan-Guided Query Rewriting and Verified Index Advice

Mr. Mahesh Prabhakar Lendave,
Mrs. Asmita Mohan Phatangare (Assistant Professor)

Department of Master of Computer Application,
JCEI's Jaihind Institute of Management and Research, Vadgaon Sahani
Email: maheshlendave1252@gmail.com,
asmitafatangare41@gmail.com

Abstract - Writing correct and efficient SQL queries requires the knowledge of the database schema and of the query engine's behavior. Therefore, direct data access is usually limited to the database experts. Most current natural language to SQL approaches focus on the result correctness of the generated SQL queries and neglect the efficiency of the queries. In this paper, we propose a Python-based multi-agent system which combines query generation and query optimization. Our system consists of five cooperating agents: a schema linking agent, a planning agent, a generation agent, a validation agent, and an optimization agent. These agents communicate with each other using explicit intermediate artifacts and use the database's feedback, such as the output of EXPLAIN statements and the measured execution time of the generated queries, in order to verify their work. The optimization agent uses sargability-restoring rewrite rules as well as an execution-validated index advisor. It only accepts a change if the result set of the query is unchanged and if, in case of index advice, the latency of the query is decreased. We evaluated our system on SQLite using a synthetic four-table e-commerce database with three scales (10,000 to 200,000 orders) and a benchmark of 56 in-scope questions with independently written gold SQL queries. For all 56 in-scope questions, the generated SQL queries were correct. For eight out-of-scope questions, the system refused six of them and silently mis-answered two. The optimized queries returned identical results to the unoptimized queries in all cases. For 50,000 orders, the full optimizer achieved a geometric mean speedup of 1.79x (95% bootstrap CI 1.44–2.26) and 1.89x for 200,000 orders. The mean latency of the largest scale was decreased from 184.6 ms to 153.6 ms. An ablation study shows that most of the gain is achieved by query rewriting (1.43x alone) and that index advice alone is unreliable (1.13x, with 18% of queries slowed down by more than 10%). Since the planner is rule-based and the data are synthetic, our results demonstrate the value of the generate-validate-optimize loop rather than the open-domain translation accuracy.

Key Words: text-to-SQL, AI agents, query optimization, index selection, query rewriting, SQLite, Python.

1. INTRODUCTION-Relational databases are still the main storage for operational and analytical data. Therefore, querying them still requires the knowledge of SQL as well as of the underlying database schema. Business analysts, domain scientists, and application developers often know what they want to ask but they do not know how to express it as a correct and efficient query. Natural language interfaces to databases aim to bridge this gap. The task of translating a question into SQL (text-to-SQL) has attracted much attention in recent years. This attention is supported by benchmarks such as WikiSQL-style corpora [20], Spider [19], and BIRD [10] as well as by recent approaches which decompose the problem into sub-tasks which are then solved by large language models [6, 13].

There are two limitations of current practice which motivate our work. First, most text-to-SQL research evaluates only whether the generated query returns the correct result. They do not ask whether the query is efficient. A semantically correct query can still be much slower than an equivalent formulation (about 16 times slower for one query type in our experiments). For example, if a predicate wraps an indexed column in a function, such as strftime('%Y', order_date) = '2024' or LOWER(city) = 'pune', then the query engine cannot use the index [15]. Since machine-generated SQL queries tend to favor such straightforward, correct-by-construction patterns, the cost of ignoring efficiency is systematic rather than incidental. Second, query optimization and physical design tuning are traditionally treated as separate concerns. They are either handled by the database's cost-based optimizer [3, 14] or by offline index advisors [1, 2]. They are rarely connected to the component which generates the query in the first place.

An agent-based design offers a natural way to connect these stages. Instead of a single monolithic translator, the task is split among cooperating components with narrow responsibilities (schema understanding, planning, generation, validation, and optimization). These components communicate with each other using explicit intermediate artifacts and can check each other's work. This approach has already been explored for the correctness of text-to-SQL queries [16, 18]. The feedback from the database, such as syntax errors, query plans, and measured run time, closes the loop. It allows the system to reject an unsafe transformation or to keep a physical design change only if it demonstrably helps.

This paper presents a Python implementation of such a multi-agent pipeline over SQLite and evaluates it empirically. A key design principle is that no optimization is accepted unless it is verified: every rewrite must return a result identical to that of the original query, and every candidate index must decrease measured latency by a minimum margin. Otherwise, it is rolled back.

1.1 Problem Statement

Even if a text-to-SQL translator returns the right answer, three gaps remain. (i) Efficiency is rarely measured. Therefore, slow but correct queries are not noticed. (ii) Rewriting and index tuning are disconnected from the component which generates the query. Therefore, patterns which a generator emits systematically are not corrected at the source. (iii) Automatic rewrites and indexes can be unsafe: a rewrite may change semantics, and an index which does not help still costs storage and write time. What is needed is a pipeline which generates, validates, and optimizes a query. It must check every change against the engine before adopting it.

1.2 Objectives

• Design a modular multi-agent architecture which turns a natural language question into validated, optimized SQL over a relational schema.

• Implement plan-guided rewrite rules and an index advisor which adopts a change only after result equivalence and latency checks.

• Evaluate generation accuracy and the latency effect of each optimization component through an ablation study at three data scales.

• Document failure cases, including questions outside the system's competence, in order to give an honest account of its limits.

1.3 Contributions

This paper makes four contributions: (1) an agent architecture which unifies natural-language-to-SQL generation with query optimization. It uses explicit hand-offs and verification between agents. (2) a verified optimization strategy which pairs sargability-restoring rewrite rules with a measurement-driven index advisor and an equivalence-checking safeguard. (3) an empirical study on a synthetic multi-table e-commerce workload at three scales. It uses an independently written gold-SQL benchmark and a four-way ablation study. It also analyzes failure cases. (4) a reproducible implementation in Python with fixed random seeds. Therefore, every reported figure can be regenerated.

We deliberately scope our claims. The prototype's planner is rule-based. Therefore, it is not a substitute for large-language-model translators on open-domain benchmarks. Its purpose is to isolate and measure the effect of the generate-validate-optimize loop on a controlled schema. The agent interfaces allow the planner and generator to be replaced by an LLM-backed component without changing the validation and optimization stages. The rest of the paper is organized as follows: Section 2 reviews related work. Section 3 describes the method and experimental setup. Section 4 reports results. Section 5 discusses results and their limits. Section 6 concludes.

2. LITERATURE REVIEW

Text-to-SQL. Neural approaches to translating questions into SQL were popularized by Seq2SQL [20]. The cross-domain Spider benchmark [19] made generalization to unseen schemas the central challenge. BIRD [10] moved to larger, noisier databases and also reports an efficiency-oriented metric. This signals growing interest in query cost. Surveys such as [7] organize this body of work around deep learning methods.

LLM and multi-agent approaches. DIN-SQL decomposes the task into sub-problems and applies self-correction [13]. DAIL-SQL systematically studies prompt construction and example selection [6]. MAC-SQL assigns sub-tasks such as schema selection, question decomposition, and refinement to collaborating agents [16]. The reasoning-and-acting paradigm [18] provides the general template of agents which interleave decisions with tool feedback. These systems are chiefly judged on whether the produced query returns the right answer.

Query optimization and rewriting. Cost-based plan selection dates to System R [14] and is surveyed in [3]. Rule-based query rewrite systems, such as the one in Starburst [12], transform a query into an equivalent but cheaper form before plan selection. Recent work discovers and verifies such rules automatically [17]. The behavior of the SQLite planner, including the conditions under which an index can serve a WHERE-clause term, is documented in [15].

Index selection and learned components. Cost-driven index advisors evaluate candidate indexes against a workload [1, 2]. An experimental comparison of index selection algorithms [8] shows that they differ in solution quality and in the time they need. Learning-based alternatives replace parts of the database, such as index structures [9] and the optimizer itself [11].

Gap. Text-to-SQL research focuses on correctness. Optimization research starts from an existing query. Our pipeline is a bridge between the two: it creates a query, validates it and then optimizes it, applying optimizations only after validation against the engine. We summarize approach families in Table 1.

Table -1: Comparison of approach families relevant to this work.

Approach	Focus	Strengths	Limitations
Neural / LLM text-to-SQL	Correct translation	Handles varied phrasing; standard benchmarks	Efficiency usually not evaluated
Multi-agent text-to-SQL	Decompose, refine, self-correct	Modular; iterative repair	Still centred on correctness
Cost-based optimizer	Plan choice for a given SQL	Mature and automatic	Limited by its rule set and statistics
Rule-based rewriting	Equivalent, cheaper SQL	Predictable; interpretable	Each rule must be authored or discovered and shown correct
Index advisors	Physical design	Workload-aware; cost-driven	Typically offline; separate from query generation
Proposed pipeline	Generate + validate + optimize	Verified rewrites and index advice in one loop	Rule-based planner; one engine; synthetic data

Approach Focus Strengths Limitations

Neural / LLM text-to-SQL Correct translation Varied phrasing; standard benchmarks Often not efficiency sevaluated

Multi-agent text-to-SQL Decompose, refine, self-correct Modular; iterative repair Still focus on correctness

Cost-based optimizer Plan choice given SQL Mature, automatic Limited by rule-set, statistics

Rule-based rewriting Equivalent, cheaper SQL Predictable, interpretable Rule authoring / discovery, shown correct

Index advisor Physical design Workload aware, cost-driven Often offline; separate from query generation

Proposed pipeline Generate + validate + optimize Verified rewrites & index advice in one loop Rule-based planner, one engine, synthetic data

3. METHODOLOGY

3.1 System Architecture

We have 5 agents working on a SQLite database, as depicted in Fig -1. A question is processed through schema-linking, planning, generation and validation agents to generate valid, but deliberately simple SQL. The optimization agent then rewrites it and optimizes physical design. The database provides metadata, attribute values, query plans and execution times. Every agent communicates with the next one via an explicit intermediate artifact (linked values, query frame, SQL string, validation verdict).

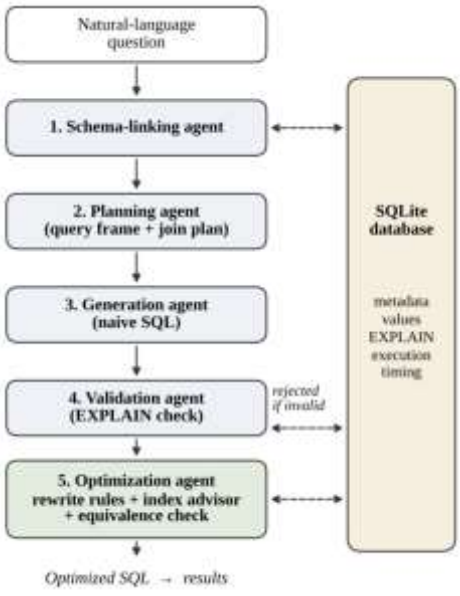


Fig -1: Architecture of the five-agent pipeline and its interaction with the database.

3.2 Agents

Schema-linking agent. It reads the distinct values of the categorical attributes (city, customer segment, product category, order status) and links words in the question to them, using case-insensitive whole-word matching, longest value first. It also maintains the foreign key graph of the schema.

Planning agent. It creates a query frame. It consists of: measure (revenue as sum of quantity times price, average price, units sold, or count of orders, customers, products, distinct customers); filters (linked values, calendar year 4-digit); grouping (added via words like per, by, each, implied by top-N request); top-N limit (digit, number word). If no measure is recognized, the agent throws an error. This is how the system rejects a question.

Generation agent. It translates the frame into SQL. Tables are joined via the shortest foreign key path found via breadth-first search. The agent deliberately generates idiomatic, but naive SQL. Filters are wrapped as LOWER(col) = 'value', strftime('%Y', col) = 'year'. Ties in top-N are broken by group key to obtain deterministic results.

Validation agent. It submits SQL to the engine with EXPLAIN. If engine reports error, it rejects SQL. This identifies syntactic and schema errors, but not semantic mistakes.

Optimization agent. See next subsection.

3.3 Optimization Procedure

For each valid query, the optimization agent does the following.

1. Run original query (3 times, timed) to obtain reference result, reference time.

2. Apply 2 rewrite rules. R1: strftime('%Y', c) = 'Y' -> c >= 'Y-01-01' AND c < 'Y+1-01-01'. R2: LOWER(c) = 'v' -> c = 'V'. (V is the stored spelling found via lookup of distinct values of column). Both restore use of index on column.

3. If rewritten query executes, and results (multiset of rows) are equal to reference (floats compared up to 4 decimals), then accept rewritten query. Otherwise keep original query, count rejection.

4. Index advisor: gather candidate columns (equality, range predicates, join keys (not primary keys)). For each candidate, create index. Time 3 runs. Keep index if latency is at least 5% lower than the best time so far. Otherwise drop index. Kept indexes persist. Thus, subsequent queries in workload benefit from them.

We use empirical equality of results on current database instance, not formal proof of equivalence. Formal verification of rewrite rules is subject of dedicated work [17].

3.4 Dataset and Benchmark

We generated a synthetic e-commerce database with 4 tables (customers, products, orders, order_items), joined via foreign keys. It was generated via a seeded random number generator. Orders are between 2022 and 2025. Orders have 1 to 5 items. We use 12 cities, 3 segments, 8 product categories, 4 order statuses. We created 3 scales: 10,000 orders (1,000 customers, 100 products, 30,079 items), 50,000 orders (5,000, 500, 150,229), 200,000 orders (20,000, 2,000, 599,195). Initially only primary key indexes exist. We collect statistics via ANALYZE.

The benchmark contains 56 in-scope questions, from 10 families. Each question has 1-2 paraphrases. We have: total revenue by year, orders by city, revenue per category, top-N customers by revenue, revenue by segment, year, distinct customers by status, year, average price by category, quantity by category, city, revenue by city, year, top-N products by quantity, year. We wrote separate gold SQL for each question,

using different structures (sub-queries, sargable predicates). We also have 8 out-of-scope questions (anti-joins, medians, set intersection, percentages, growth rates) to see how the system fails.

3.5 Evaluation Metrics

We score generation via execution accuracy. The result set must be equal to that of the gold SQL (order-insensitive, floats up to 4 decimals). For out-of-scope questions we score the system via count of refusals, count of silent errors. Optimizer safety is the percentage of queries where optimized result is equal to unoptimized result. Latency is the median of 5 runs, after optimizing the whole workload. We compute speedup per query: unoptimized time / optimized time. We aggregate speedups via geometric mean [5] with 95% percentile bootstrap interval from 2,000 resamples [4]. We also report the percentage of queries improved/slowed by more than 10%, mean latency, optimization overhead.

3.6 Experimental Setup

We ran experiments in Python 3.13, using SQLite 3.45.1 (via standard sqlite3 module). We ran them on a Linux machine with 2 virtual CPUs (Intel Xeon, 2.8 GHz), 7 GB memory. We compared 4 configurations, on a fresh copy of the database: no optimization, rewrite only, index advice only, full optimizer. We ran the ablation on 50,000 orders. We also ran the no optimization and full configurations on 10,000, 200,000 orders. We used fixed data, benchmark seeds (7, 11).

4. RESULTS

4.1 Generation Accuracy

The agent generated SQL whose result was equal to gold SQL for all 56 in-scope questions (56/56, 100%). This applies to all 10 families. For the 8 out-of-scope questions, 6 were rejected (planner didn't recognize any measure). 2 were answered silently, with wrong SQL. Request for monthly revenue growth in 2024: single revenue total returned. Request for cancelled order revenue as a percentage of total revenue: cancelled order revenue returned, but no percentage.

4.2 Optimizer Safety

For all 4 configurations, optimized queries returned exactly the same results as unoptimized queries (56/56). Still matched gold SQL. We never rejected a rewrite via the equivalence check (0 rejections). Thus, we can't tell if it would catch a faulty rule.

4.3 Ablation of the Optimizer

We report ablation at 50,000 orders in Table 2. Geometric mean speedup for full optimizer is 1.79 (95% CI: 1.44-2.26). Speedup for rewriting only is 1.43 (1.34-1.54). 75% queries improved by more than 10%. Speedup for index advice only is 1.13 (1.00-1.30). Median speedup: 0.98. 17.9% queries slowed by more than 10%. Full optimizer kept 7 indexes. It slowed 7.1% of queries. Incremental optimization time (over 56 queries): 41.8 s (full optimizer), 49.0 s (index advice only). (7.9 s for rewriting only). These include timing runs for verification. 7.1 s of each is baseline timing, common to all configurations.

Table -2: Ablation at 50,000 orders (56 queries). Speedup is relative to the unoptimized query; CI is a 95% bootstrap interval; Impr. and Regr. are the shares of queries improved or slowed by more than 10%; Idx is the number of indexes kept.

Setting	Geo. mean speedup (95% CI)	Impr. >10% (%)	Regr. >10% (%)	Mean (ms)	Idx
None	1.00	–	–	41.8	0
Rewrite only	1.43 (1.34–1.54)	75.0	1.8	33.8	0
Index only	1.13 (1.00–1.30)	12.5	17.9	44.0	2
Full optimizer	1.79 (1.44–2.26)	69.6	7.1	39.3	7

4.4 Effect of Data Scale

We compared unoptimized and full configurations on three data scales in Table 3. The geometric mean speedup did not change much, remaining between 1.79 and 2.07 for all three scales, but the improvement in the mean latency was very small, from 5.8% for 50k orders to 16.8% for 200k orders. The number of indexes kept was different on each scale (8, 7, 5), indicating that the advice of the advisor is also dependent on the timing noise in addition to the data.

Table -3: Unoptimized versus full optimizer at three data scales.

Orders	Unopt. (ms)	Full (ms)	Geo. mean speedup (95% CI)	Median speedup	Idx
10,000	6.78	6.23	2.07 (1.66–2.61)	1.76	8
50,000	41.77	39.33	1.79 (1.44–2.26)	1.25	7
200,000	184.62	153.56	1.89 (1.56–2.31)	1.52	5

4.5 Results by Query Family

Fig -2 shows the breakdown of speedup per query family on 50k orders. The highest improvement was on the count of orders per city (16.3 on full optimizer), with rewrite alone improving by 1.22, and index alone improving by 4.73. This is because they help each other, as the index on city can be used only after the LOWER() wrapper is removed. There was also clear improvement on average price per category (3.45), distinct customers by status and year (2.43). On the other hand, the full optimizer had slightly worse performance on revenue per city (0.92) and top-N customers (0.92). The revenue per category remained the same (1.01). On revenue per city and year, rewrite alone improved by 1.48, but with the additional indexes, this improvement was gone.

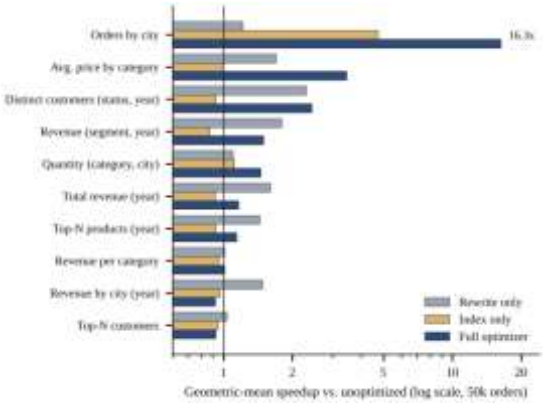


Fig -2: Geometric-mean speedup by query family at 50,000 orders (values below 1 are slowdowns).

5. DISCUSSION

The main idea that generation and optimization can be connected is validated. A generator that generates predictable non-sargable patterns can be coupled with a small set of validated rewrites that achieve most of the speedup with no risk of correctness. The most important component is rewriting. Index advice is mostly useful when combined with rewriting, as shown by the example with orders per city.

Index advice is not reliable. The median speedup was less than 1. Almost 20% of queries (nearly 1 in 5) were slowed down by more than 10%. Even in the full configuration, two query families were slightly worse. It is possible that the additional

indexes affected the join strategy that the engine used for large aggregations, but we did not explore this in detail. This indicates that per-query verification (as we do) is better than just taking an index because of a single measurement.

The absolute improvement was small. The dominant queries in the total latency are the full aggregations on order_items and orders tables (revenue per category, top-N customers). These were not improved by rewriting or single-column indexes. Next steps can be materialized aggregates or covering indexes. The mean latency improvement of 5.8% to 16.8% should be considered with the larger geometric mean speedups, which are due to many short queries.

The silent errors on out-of-scope questions (2 of them) show a limitation of our validation agent. It detects invalid SQL (with EXPLAIN), but not semantic correctness. In a deployed system, we would need to have an abstention policy, e.g., if a part of the question is not covered by the recognized frame.

Limitations and threats to validity. (i) Rule-based planner, benchmark written by the same author as the planner. The 100% accuracy indicates the pipeline is coherent, but not necessarily that it can generalize to open-domain questions. (ii) Synthetic data, from one schema and one engine (SQLite). (iii) 56 queries from 10 families, not independent. The bootstrap intervals (resample independently on queries) are likely too narrow. (iv) Two CPU VM for timing, index selection depends on scale, so timing noise is a factor. (v) Storage and write overhead of indexes is not measured. (vi) Result equality on one database instance is not a proof of equivalence.

6. CONCLUSION AND FUTURE WORK

In this paper, we presented a Python multi-agent pipeline to generate SQL from natural language questions and then optimize it with verified rewrites and measured index advice. We correctly answered all 56 in-scope questions on a synthetic e-commerce workload, with no change to query result, and achieved a geometric mean speedup of 1.79 to 2.07 (depending on scale). Most of the speedup was from rewrite, and index advice was mainly useful in combination with it. We also showed the limitations: modest absolute latency improvement, worse performance on some families, and silent errors on out-of-scope questions.

Future work includes the replacement of rule-based planner and generator with LLM-backed component and testing on public benchmarks with realistic databases. Extending the pipeline to other engines, e.g., PostgreSQL, MySQL. Using hypothetical indexes, cost estimates instead of repeated timing. Measurement of index storage and write overhead. Formal verification of rewrite rules. Abstention mechanism for detecting uncovered parts of question.

Data and code availability. Generator, benchmark, agent code, and raw results are available at [repository URL to be added by the authors].

REFERENCES

1. Agrawal, S., Chaudhuri, S., Kollár, L., Marathe, A., Narasayya, V., & Syamala, M. (2004). Database tuning advisor for Microsoft SQL Server 2005. Proceedings of the 30th International Conference on Very Large Data Bases (VLDB), 1110–1121.

2. Chaudhuri, S., & Narasayya, V. R. (1997). An efficient, cost-driven index selection tool for Microsoft SQL Server. Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB), 146–155.

3. Chaudhuri, S. (1998). An overview of query optimization in relational systems. Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS), 34–43.

4. Efron, B., & Tibshirani, R. J. (1993). An Introduction to the Bootstrap. Chapman & Hall.

5. Fleming, P. J., & Wallace, J. J. (1986). How not to lie with statistics: The correct way to summarize benchmark results. Communications of the ACM, 29(3), 218–221.

6. Gao, D., Wang, H., Li, Y., Sun, X., Qian, Y., Ding, B., & Zhou, J. (2024). Text-to-SQL empowered by large language models: A benchmark evaluation. Proceedings of the VLDB Endowment, 17(5), 1132–1145.

7. Katsogiannis-Meimarakis, G., & Koutrika, G. (2023). A survey on deep learning approaches for text-to-SQL. The VLDB Journal, 32. https://doi.org/10.1007/s00778-022-00776-8

8. Kossmann, J., Halfpap, S., Jankrift, M., & Plattner, H. (2020). Magic mirror in my hand, which is the best in the land? An experimental evaluation of index selection algorithms. Proceedings of the VLDB Endowment, 13(12), 2382–2395.

9. Kraska, T., Beutel, A., Chi, E. H., Dean, J., & Polyzotis, N. (2018). The case for learned index structures. Proceedings of the ACM SIGMOD International Conference on Management of Data, 489–504.

10. Li, J., Hui, B., Qu, G., et al. (2023). Can LLM already serve as a database interface? A big bench for large-scale database grounded text-to-SQLs. Advances in Neural Information Processing Systems (NeurIPS), 36, Datasets and Benchmarks Track.

11. Marcus, R., Negi, P., Mao, H., Zhang, C., Alizadeh, M., Kraska, T., Papaemmanouil, O., & Tatbul, N. (2019). Neo: A learned query optimizer. Proceedings of the VLDB Endowment, 12(11), 1705–1718.

12. Pirahesh, H., Hellerstein, J. M., & Hasan, W. (1992). Extensible/rule based query rewrite optimization in Starburst. Proceedings of the ACM SIGMOD International Conference on Management of Data, 39–48.

13. Pourreza, M., & Rafiei, D. (2023). DIN-SQL: Decomposed in-context learning of text-to-SQL with self-correction. Advances in Neural Information Processing Systems (NeurIPS), 36.

14. Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A., & Price, T. G. (1979). Access path selection in a relational database management system. Proceedings of the ACM SIGMOD International Conference on Management of Data, 23–34.

15. SQLite Consortium. The SQLite query optimizer overview. https://www.sqlite.org/optoverview.html (accessed October 2026).