DATA MODEL REFINEMENT FOR LLM-BASED SQL CODE GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

The current development of Text-to-SQL systems are mostly based on the improvement of the underlying Large Language Models (LLMs) or the surrounding harness. While this direction is proved to be highly effective for the frontier models, smaller locally deployable solutions usually lack the ability to write suitable complex SQL queries for realistic, complicated databases. This motivated our research to search an alternative direction, where we keep the model intact and optimize the database schema it has to read. The schema becomes the optimization variable, and we search for a data model which is optimal for a small local model to write SQL queries. An evolutionary loop proposes restructured schemas as code, every candidate must preserve the original data, and the fitness of a candidate is the measured execution accuracy of the fixed translator on a training split of the workload and the quality components of the data model such as integrity, implementability, simplicity. On 13 real databases and 101 questions from the Spider 2.0-Lite benchmark, 20 generations of schema evolution raise the mean execution accuracy of three small local translators from 27.1% to 50.6%, at a mutator cost of about $3.50 per database. The two stronger evolved translators were on par with Claude Opus 4.6 with thinking mode reading the original schemas (56.4% and 57.4% against 54.5%). The best schemas are purely additive: they drop nothing, grow a documented layer of derived tables that removes most joins from the translator's job, and differ per model. The whole loop can also run on local hardware: a 27B mutator with thinking mode recovers most of the gains of a frontier mutator at zero API cost.

1 INTRODUCTION

Text-to-SQL systems write queries for databases that already exist, and their promise is to speed up the writing of SQL for these databases. It is not clear whether the available methods are reliable enough for this. Spider 2.0 (Lei et al., 2025), a benchmark of real analytical databases and the natural language requests analysts run against them, gives the answer: even frontier models get only a fraction of these requests right, and small open-weight models do far worse. This lack of reliability holds back adoption. Their failures are usually specific to one database, yet the usual response is to improve the model: make it larger, fine-tune it, give it more attempts, wrap it in an agent. This paper takes the opposite direction: we keep the model fixed and change the structure of the data model so that it fits the abilities of the model that writes the SQL.

Small models are the ones many organizations must use, because running a model on their own hardware is often the only acceptable option for privacy and cost, a constraint studied well before the current model generation (Yang & Váradi, 2023). The three local models we study start between 8.9% and 41.9% execution accuracy on the local subset of the benchmark. Their failures follow a pattern: real schemas carry cryptic column names, wide tables, implicit conventions, and facts scattered across many tables, because they were designed for application code and administrators, long before anyone expected a language model to read them.

Most of the problem however are addressable through renamed columns, documented tables, and pre-joined or pre-aggregated views. Because the model's capabilities show high variance (Sclar et al., 2024), different optimal solutions may exist for each model. We therefore treat the schema as an optimization variable and search for the presentation on which a fixed model is most accurate.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

The search is evolutionary: a mutator model proposes restructurings as code, every candidate must preserve the original data exactly, static checks reject fabrication, and the fitness of a candidate is the measured execution accuracy of the fixed translator on a training split of the workload. Questions held out for evaluation never reach the mutator in any form.

A database administrator could redesign a schema by hand. The difficulty is knowing what to change: which of the many possible presentations helps a given model is not obvious in advance, and our results show that it differs from model to model. The search replaces this guesswork with measurement. Each proposed schema is judged by the accuracy the model achieves on it, and each failed question returns to the mutator as concrete feedback, so the design decisions are grounded in observed failures rather than intuition about what a language model prefers.

We evaluate this idea on 13 real databases of Spider 2.0-Lite with three small local translators, compare the evolved schemas against a frontier model reading the original schemas, repeat the experiment with locally served mutators, and analyze what the best schemas contain and how they change the generated SQL.

This paper makes three contributions. First, a formulation of schema optimization for a fixed text-to-SQL model, with data preservation as a hard constraint and measured benchmark accuracy as the fitness. Second, evidence on a hard public benchmark that this optimization roughly doubles the accuracy of small local translators, together with a comparison of four mutator models. Third, a qualitative analysis of what the best schemas contain, showing that they form additive, documented, model-specific view layers that cut join counts by more than half.

2 Related work

Text-to-SQL has a long benchmark tradition. Spider (Yu et al., 2018) established cross-domain evaluation by execution accuracy, BIRD (Li et al., 2023) added large and dirty databases with external knowledge, and Spider 2.0 (Lei et al., 2025) moved to real enterprise data, where the accuracy of every published system drops sharply. Progress on these benchmarks has come from the model side: decomposed prompting with self-correction (Pourreza & Rafiei, 2023), systematic prompt engineering (Gao et al., 2024), and agentic systems that iterate against the database (Lei et al., 2025). All of this work treats the database as immutable. Our work is complementary: any of these systems could run over an evolved schema.

Within that literature, it is well documented that how the schema appears in the prompt matters. Schema linking selects the relevant tables and columns, and the format of the schema text alone affects accuracy (Pourreza & Rafiei, 2023; Gao et al., 2024). These techniques choose what part of a fixed schema to show. We change the schema itself, and we let measured accuracy decide the presentation instead of heuristics.

Our search procedure belongs to the evolutionary tradition, which has a long record on hard combinatorial problems where exact methods become infeasible (Almahasneh et al., 2025). Its recent extension replaces hand-designed mutation operators with language models that edit programs against an automatic evaluator. FunSearch (Romera-Paredes et al., 2024) discovered new mathematical constructions this way, AlphaEvolve (Novikov et al., 2025) improved production algorithms, and ShinkaEvolve (Lange et al., 2026) is the sample-efficient open-source engine we build on. We bring this recipe to a different artifact, a database schema under a data-preservation constraint, and to a different evaluator: the measured behavior of a second, fixed language model rather than a hand-written objective.

The idea of adapting a database to its workload is much older than language models. Physical database design has done it for decades, most prominently through automatic index selection (Chaudhuri & Narasayya, 1997) and materialized view selection (Agrawal et al., 2000). Our method extends this tradition to a new kind of workload: the queries are natural-language questions, and the cost model being optimized is not disk access but a language model's ability to read the schema. Prompt optimization (Yang et al., 2024) shares our spirit of optimizing the input rather than the weights, but it tunes a per-task prompt, while we tune a persistent piece of infrastructure that serves every future question.

3 METHOD

3.1 PROBLEM SETTING

We consider a relational database together with a workload of natural language questions about its contents, each paired with a verified answer. A translator is a language model that maps the schema text and one question to a single SQL query. The translator stays fixed throughout this work: it is not fine-tuned, receives no error feedback, and runs no agent loop. It reads the schema, emits one query, and the executed result either matches the verified answer or it does not.

We treat the schema as the optimization variable. The objective is execution accuracy: the fraction of workload questions for which the translator's query, executed against the restructured database, returns the verified answer under the benchmark's official comparison rules. We constrain the evolution to schemas that represent the original contents without loss, so only the presentation of the data may change, never the data itself and never the model. Execution accuracy, measured on a training split of the workload, is the fitness that drives the evolution described below.

The evolution proceeds by generating and evaluating candidates. A candidate is one proposed restructuring of the database, and it has two parts. The first part is the schema definition (DDL). The second part is a migration script that fills the new schema from the original database, which stays attached in read-only mode. Evaluating a candidate means building the proposed database and measuring the translator's accuracy on it. The migration may only copy and transform rows that exist in the source. Statements that invent rows or silently drop them are rejected before evaluation, and a candidate whose migrated tables hold fewer total rows than the largest source table scores zero. This count check is a floor, not an equality: a faithful migration must carry every source row somewhere, so its total can never fall below the largest source table, while splitting or pre-joining tables only grows the total and never trips the check. Under these rules, every candidate is the same data behind a different surface: renamed columns, pre-joined tables, summary tables, views, indexes, and comments that explain what each table stores.

The first candidate is the identity: the original schema with a plain copy of every table. Its score is the zero-shot baseline of the translator, measured by the same procedure as every later candidate. Any improvement over this baseline is therefore attributable to the schema alone.

In our experiments, the databases, the questions, and the verified answers all come from Spider 2.0-Lite (Lei et al., 2025), a benchmark built from real analytical databases, so the workload reflects questions that are actually asked of such data. Section 4 describes the benchmark subset and the evaluation protocol in detail.

3.2 THE EVOLUTIONARY LOOP

We run an evolutionary search over candidates, built on an existing open-source engine for LLM-driven program evolution (Lange et al., 2026). Each generation proposes and evaluates one new candidate, and a run lasts 20 generations. The engine keeps an archive of the 20 best evaluated candidates, split into two separate islands, and samples a parent from it with probability weighted by score, with one draw in five reserved for the top 30% of the archive. A mutator model then receives the task instructions, the parent's code, two further high-scoring candidates as inspiration, and the feedback from the parent's evaluation, and proposes a change: a small patch in 80% of the proposals and a full rewrite in the remaining 20%. We cycle the mutator's sampling temperature across the values 0, 0.5, and 1, because a purely greedy mutator tends to repeat the same patch; a malformed patch is retried up to three times against the same parent, and the parent is redrawn up to five times before a generation is given up. We disable crossover: merging the schema of one parent with the migration of another usually produces a pair that no longer fits together. Every fourth generation, a summarizer model reads all evaluated candidates and distills up to three standing recommendations, which are added to later mutation prompts. This gives the evolution a simple long-term memory.

Before any candidate is built, a fast static check runs without any model call. It rejects candidates that delete required parts, declare foreign keys against columns that are not unique, or fabricate or truncate data in the migration. The rejection message states the problem in one imperative sentence and is returned to the mutator as feedback, so cheap mistakes cost almost nothing.

3.3 THE EVOLUTIONARY FITNESS

Data model optimization is inherently a multi-objective problem, and as such a different optimal solution exists for each objective. In this work, for simplicity, we collapse the objectives into a single scalar fitness: a gated linear combination of several components with user-defined weights. The components used in this study are execution accuracy, integrity, implementability, simplicity, and a data-loss guard; they are defined in Table 1.

Let c be a candidate, that is, a DDL together with its migration script, and let Q be the training questions of the database, $N = |Q|$. We write $A(c) \in [0, 1]$ for the execution accuracy of the fixed translator on c; $S_1(c), S_2(c), S_3(c) \in [0, 1]$ for the integrity, implementability and simplicity scores; and $g(c) \in \{0, 1\}$ for the data-loss guard, which is 0 exactly when the migration fails the row-count check of Section 3.1 and 1 otherwise. The fitness of a candidate is

$$F(c) \;=\; g(c)\Big(A(c) \;+\; \varepsilon \sum_{i=1}^{3} w_i S_i(c)\Big), \qquad \sum_{i=1}^{3} w_i = 1, \tag{1}$$

with structural weights $(w_1, w_2, w_3) = (0.5, 0.3, 0.2)$ and tie-breaker scale $\varepsilon = 0.01$. Two properties of Eq. 1 are deliberate. The guard is multiplicative, so a candidate that loses data scores zero no matter how good its structure is.

Table 1: Components of the candidate score. Execution accuracy is the primary term; the three structural measures act only as tie-breakers between candidates with equal accuracy. Structural weights are scaled by $\epsilon = 0.01$; see Eq. 1

Component	**How it is computed**
Execution accuracy	Per question, the fraction of three translation rounds whose executed result matches the verified answer; averaged over the training questions.
Integrity	Share of tables with a primary key (0.5), foreign keys per table, capped at one (0.3), and share of non-null columns (0.2).
Implementability	Data load time, schema creation time, and the median latency of passing queries, each rescaled to a value between zero and one so that faster is better.
Simplicity	Mean syntax-tree complexity of the generated queries (node count plus five times tree depth), rescaled to a value between zero and one so that simpler is better.
Data-loss guard	Total rows after migration must not fall below the row count of the largest source table; growth is allowed.

The score is only one part of what an evaluation returns; the most important signal in the loop is the feedback attached to each evaluated parent. For every failed question, the mutator sees the SQL the translator actually generated, together with either the database error or a mismatch report: the expected columns, sample rows from both the expected and the returned result, and the closest matching returned column. Questions that pass in some rounds but not all are marked as almost solved. The feedback also lists the slowest and the most complex passing queries. In this form, a wrong answer stops being a number and becomes a concrete design instruction, for example that the model keeps joining through the wrong key, or that it aggregates at the wrong level. Each quoted query and mismatch report is truncated to a fixed length, so the feedback for a whole database stays within the mutator's context window even when many questions fail.

Because the score rewards benchmark accuracy, the evolution will exploit any shortcut it can find, so the scoring procedure also closes the shortcuts we know of in three ways. First, only a fixed 70% training split of the questions is visible during the evolution. The split is deterministic: within each database, every third question in the benchmark's order is held out, so every run, translator, and mutator uses the same 72 training and 29 held-out questions. The held-out questions and their answers never appear in any mutator input, and we verify this mechanically over the recorded prompts. Second, schema comments that refer to question identifiers are removed from every candidate before evaluation, so the schema must describe the data rather than index the test. Third, the data preservation rules of Section 3.1 prevent the evolution from writing answers into the database. What remains available to the evolution is what we want it to use: structure, naming, and documentation.

Table 2: Translator models (top) and mutator models (bottom). All open-weight models run locally as quantized builds; the frontier mutator is a cloud API.

Model	Parameters	Architecture	Reasoning	Quantization
Translators				
Qwen3.6-35B-A3B	35B (3B active)	Mixture of experts	thinking on	UD-Q8_K_XL
Gemma-4-31B-it	31B	Dense	thinking on	UD-Q8_K_XL
Devstral-Small-2-24B	24B	Dense, code model	none	UD-Q8_K_XL
Mutators				
Claude Sonnet 4.6	undisclosed	cloud API	default	none
Qwen3.6-27B	27B	Dense hybrid	thinking on	UD-Q8_K_XL
Qwen3-Coder-Next	80B (3B active)	Mixture of experts, code	none	UD-Q6_K_XL
Devstral-2-123B	123B	Dense, code model	none	UD-Q4_K_XL

4 Experimental setup

4.1 Benchmark and databases

We evaluate on the local SQLite subset of Spider 2.0-Lite (Lei et al., 2025). Unlike earlier text-to-SQL benchmarks, Spider 2.0 is built from real analytical databases and asks the kind of questions analysts run against them, which makes it substantially harder: leading models solve most of the classic Spider benchmark but only a small fraction of Spider 2.0. We ingested all 30 local databases with their 135 questions and ran the experiments on the 13 databases that have at least five questions each, giving 101 questions in total. The databases cover sports, e-commerce, finance, media, public administration, and system logs, and range up to 38 tables and about two million rows. Each question keeps its official external-knowledge document, and correctness is judged against the official gold result tables under the official comparison rules. We verified our judging code against the benchmark's own evaluation script and obtained identical verdicts.

Every reported evaluation uses the three-round procedure of Section 3.3; a question counts as correct when at least two of the three rounds return the verified answer. Aggregate numbers are means over rounds and are therefore fractional by design.

4.2 Models

We use three translator models, chosen to span size, architecture, and specialization; Table 2 lists them. All three run locally on the same inference server build, with 8-bit quantization and the sampling settings recommended by each vendor. The translator receives the schema text, the value digest described in Section 3.3, the question, and its external-knowledge document, and returns one SQL query.

For reference, we also measure two frontier translators on the unchanged schemas: Claude Sonnet 4.6 and Claude Opus 4.6, both with thinking mode and the same thinking budget as the local models. These reference runs go through the official API.

The mutator in the main experiment is Claude Sonnet 4.6, a frontier model accessed through an API. In a separate experiment, we replace it with three locally served open-weight mutators, listed in the lower half of Table 2: a 27B general model with thinking mode, an 80B mixture-of-experts code model, and a 123B dense code model. This lets us ask whether the schema evolution itself requires a frontier model, or whether the whole loop can run on local hardware. All experiments ran on one machine with two NVIDIA H100 GPUs: one serves the translator, the other serves the local mutator when one is used.

4.3 Protocol

For each of the 39 runs (three translators on 13 databases), the protocol has three stages. First, the unchanged original schema is evaluated on all questions of the database; this is the zero-shot baseline. Second, the evolution runs for 20 generations, seeing only the fixed 70% training split,

which is 72 of the 101 questions; the remaining 29 are held out. Third, the best schema found by the evolution is re-evaluated on all questions with the same procedure as the baseline, so baseline and evolved numbers are directly comparable, including on the held-out 30%. Every gain we report is this pair: the same translator on the same questions, with only the schema changed. The frontier translators of Section 4.2 read the original schema and take no part in the evolution; they serve only as an external reference point for the evolved local numbers. Across the main experiment, the evolution evaluated 791 candidate schemas in total. With Claude Sonnet 4.6 as the mutator, the total API cost was $138, about $3.50 per database run. The runs with local mutators have no API cost at all, so the fully local configuration costs nothing beyond GPU time.

5 Results

5.1 Main result

Table 3 shows the effect of schema evolution for every combination of translator and mutator. We describe the main experiment first, the rows with Claude Sonnet 4.6 as the mutator, and turn to the local mutators afterwards. Averaged over all 39 runs of the main experiment, accuracy rises from 27.1% to 50.6%, a factor of 1.87, at a total mutator cost of $138. Of the 39 runs, 36 improved, 3 regressed, and none stayed flat. At the level of individual questions, each of the three translators is scored on all 101 questions, giving 303 question outcomes per schema; 75 of these flip from wrong to right, while only 9 flip from right to wrong. Queries that fail to execute at all drop from 92 to 43, so a large part of the gain comes from the translator no longer producing invalid SQL against the restructured database.

The frontier reference translators put these numbers in scale, as Figure 1 shows. On the unchanged schemas, Claude Sonnet 4.6 with thinking mode answers 46.5% of the questions and Claude Opus 4.6 with thinking mode 54.5%. After evolution, the 35B and the 31B local translators reach 56.4% and 57.4% and overtake both, and the mean over all three translators, pulled down by the weak code model, still passes Sonnet. In other words, a schema evolved once for the workload lets a local model a fraction of the size answer more of that workload than a frontier model reading the original schema. On the held-out questions the frontier models keep their strength (48.3% and 44.8%) while the evolved locals do not, which fits the same picture: evolution buys workload-specific accuracy, not general model quality.

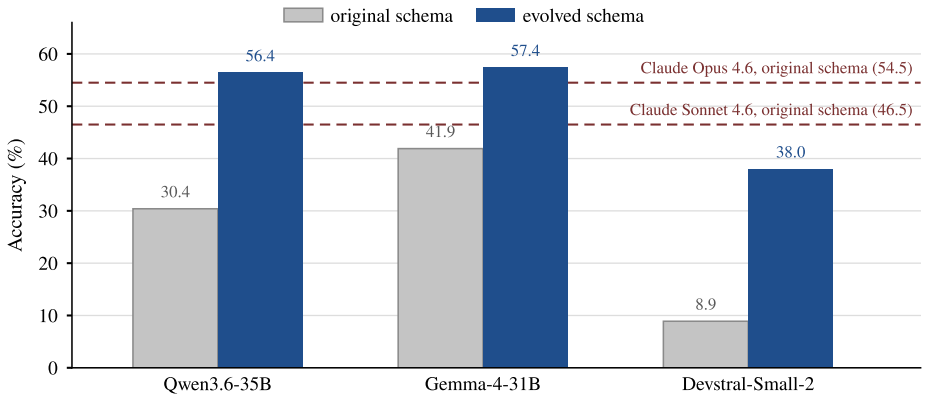


Figure 1: Accuracy of the three local translators on 101 questions from 13 Spider 2.0-Lite databases, before and after schema evolution. The dashed lines mark two frontier models reading the original schemas.

Two patterns stand out. First, the weaker the translator starts, the larger the gain from the schema: the code-specialized translator that begins below 9% more than quadruples its accuracy, and it did not regress on a single database. Second, the gains concentrate on the questions the evolution could observe. On the training split, accuracy rises from 28.2% to 59.7%; on the held-out split it moves from 24.1% to 28.0%. The evolved schemas are therefore best understood as an optimization of a

database for its known workload, not as a general repair of the translator. The 20-generation budget is a binding constraint rather than a saturated one: the best schema appeared in generation 12.4 on average, so improvement was still arriving late in the run, and a larger budget may yield further gains.

The results above were obtained with a frontier model as the mutator. To test whether this capability level is necessary, the last three rows of Table 3 repeat the whole experiment with the three locally served mutators. The 27B thinking model is a near substitute for the frontier mutator: it reaches 53.5% against Sonnet's 56.4% on the Qwen translator and exceeds it on the Gemma translator (58.4% against 57.4%). The frontier mutator's advantage concentrates on the hardest case, the weak code translator, where Sonnet reaches 38.0% and the 27B model 25.7%. On the held-out questions, no mutator moves far from the baseline; the small differences in that column are within the sampling variability expected for 29 held-out questions.

Table 3: Execution accuracy per translator and mutator on 13 databases. Left: all 101 questions of a translator. Right: the 29 held-out questions of a translator that the evolution never saw. The All columns pool the three translators (Qwen3.6-35B, Gemma-4-31B, Devstral-Small-2). Bold marks the best value in each column.

	All questions				**Held-out questions**			
Mutator	Qwen	Gemma	Devstral	**All**	Qwen	Gemma	Devstral	**All**
none (baseline)	30.4%	41.9%	8.9%	27.1%	33.3%	35.6%	3.4%	24.1%
Claude Sonnet 4.6	**56.4%**	57.4%	**38.0%**	**50.6%**	**36.8%**	35.6%	**11.5%**	**28.0%**
Qwen3.6-27B (thinking)	53.5%	**58.4%**	25.7%	45.9%	27.6%	34.5%	5.7%	22.6%
Qwen3-Coder-Next 80B	36.6%	45.5%	16.8%	33.0%	25.3%	**35.7%**	9.2%	23.4%
Devstral-2-123B	39.3%	45.2%	19.5%	34.6%	34.5%	35.6%	8.0%	26.0%

The comparison between the local mutators carries a second message: on the mutator side, thinking matters more than parameter count or code specialization. The 27B thinking model reaches higher accuracy than the 80B and the 123B code models on every translator, even though both larger models are trained for code and the mutation task is writing SQL and Python. The non-thinking mutators were also less reliable: each failed one database outright and regressed more runs. The two failed runs enter Table 3 at their baseline accuracy.

There is a simple reason why a small mutator can compete. The mutator never has to answer a benchmark question itself; it has to read the failure feedback and propose a reasonable restructuring, and the loop measures every proposal before accepting it. Selection, not the mutator, is the quality control. A weaker mutator wastes more candidates on ideas that do not survive evaluation, but the ideas that do survive are just as good. The same view explains why thinking helps more than size or code training: the useful work in a mutation step is diagnosis, reading a set of failed queries and working out what confused the translator, and that is reasoning rather than coding.

5.2 Structure of the evolved schemas

The evolution was free to restructure the databases in any data-preserving way. This section describes what is chose to do and quantifies the changes across the 39 best-of-run schemas of the main experiment and their 791 evaluated candidates.

Table 4 summarizes the structural changes per translator. Across the three translators, the evolution added 168 tables, 110 views, and 242 indexes, and it dropped nothing. Not a single original table was removed or renamed, and only six columns were renamed, all of them cleanups of characters that break SQL, such as `discount(%)` becoming `discount_pct`. The result is a layer of derived tables grown around the untouched original schema. For a production database this means that every application that works against the original schema keeps working.

The names of the added tables show what the layer is made of. The most frequent name tokens are `stats`, `season`, `monthly`, `counts`, and `summary`: more than half of the added tables are precomputed aggregates at the exact grain the workload asks about. The row counts confirm this. The median database grows by only 2 to 7 percent, which rules out wholesale denormalized copies; the

Table 4: Structural changes in the best schemas, per translator, over the 13 databases. The original schemas contain 2 comment lines in total.

	Qwen3.6-35B	Gemma-4-31B	Devstral-Small-2
Tables added	54	55	59
Tables dropped or renamed	0	0	0
Columns renamed	3	0	3
Views created	36	31	43
Indexes created	71	100	71
Comment lines in the schemas	687	541	732
Median schema text growth	2.3×	1.8×	2.3×
Median row count growth	1.07×	1.02×	1.03×

layer consists of small summary tables. The one large exception is a 2.8-million-row materialized co-appearance table that one run built for a film database, inflating it 19-fold.

The second largest change is documentation. The 13 original schemas contain 2 comment lines in total; the best schemas contain 541 to 732 per translator. The schema text is the only channel through which the evolution can pass information to the translator, and the best schemas use it to state what one row means, which values a column actually stores, date formats, units, and the correct join keys.

The added layer also changes the SQL that the translators write. Figure 2 shows how the translators' SQL changes when the same questions are asked over the evolved schemas. Joins drop the most: per query, from 2.25 to 0.91 for Qwen, from 3.49 to 0.87 for Gemma, and from 2.75 to 0.85 for Devstral. The reductions are significant under a two-sided sign test over paired questions: joins decreased for 55 questions and increased for 8 with Qwen ($p < 10^{-9}$), decreased for 59 and increased for 5 with Gemma ($p < 10^{-12}$), and decreased for 62 and increased for 4 with Devstral ($p < 10^{-13}$). On the questions that flipped from wrong to right, the evolved queries average between 0.25 and 0.39 joins, down from about 2.2 to 2.4. The mean syntax-tree complexity of the generated queries falls as well: from 206 to 126 for Qwen, from 254 to 155 for Gemma, and from 230 to 179 for Devstral. Small translators fail disproportionately at composing multi-table joins with the correct keys and grain; the evolution moves that work out of query time and into the schema, where it is done once and verified by the fitness signal.

The figure also reveals that the three translators write structurally different SQL to begin with, and that the evolution shrinks whichever construct each model leans on. Qwen decomposes problems into chains of common table expressions (2.65 per query, by far the highest) and almost never nests subqueries; the evolution halves its chains. Gemma does the opposite: it nests 3.06 subqueries per query, and the evolution cuts them to 1.17. Devstral relies on CASE expressions (1.61 per query), which the evolution halves. For each model, the evolution reduces the construct that model uses most, which is a first sign that the schemas are tailored to the model rather than generically simplified.

Finally, the best schemas differ from translator to translator. If the evolution simply discovered one good presentation of each database, the best schemas of different translators would converge. They do not. For every database, each translator's best schema stays closer to the original schema (mean similarity 0.79 to 0.85) than any two best schemas of different translators are to each other (0.73 to 0.77). On the Formula 1 database, the three translators arrived at three different patches. Together with the construct shifts of Figure 2, this shows the fitness signal tailoring the presentation to the failure modes of one specific model.

The schema diff is therefore diagnostic: reading what the evolution had to add for a given model is a qualitative map of that model's weaknesses, and can inform fine-tuning or the choice of model size. A model that already performs well on the original schema gains little: for one Gemma run the evolution never found anything better than the original schema, and its final answer was the unchanged seed. The method changes the schema where the model and the schema mismatch, and leaves it as it is where they already fit.

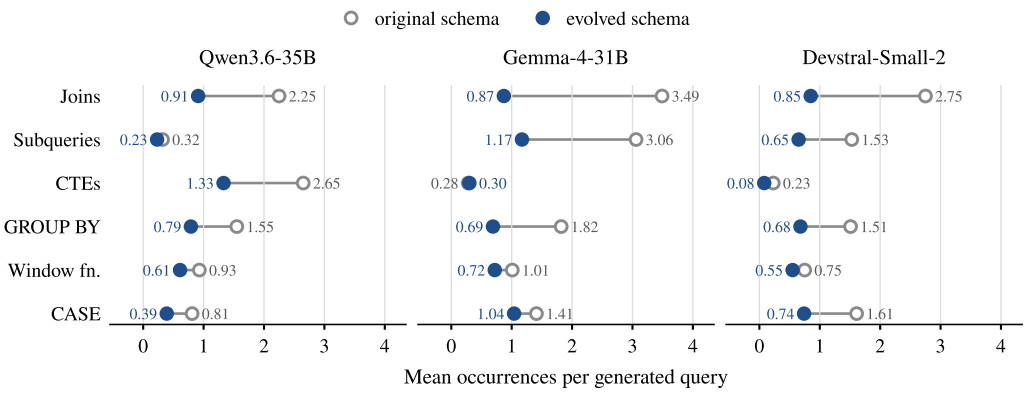


Figure 2: Mean occurrences of SQL constructs per generated query over the original and the evolved schemas, computed on the questions where both queries parse (95, 90, and 96 question pairs).

6 Conclusion

This research is tailored toward the question: how much can we improve the text-to-SQL accuracy by changing the data-model instead of the code generator LLM. With the translator fixed, evolving the schema under a strict data-preservation constraint roughly doubles the mean execution accuracy of three small local models, from 27.1% to 50.6%, for a few dollars per database, or for GPU time alone with a local mutator. With the evolved schemas the gap to frontier models closes: once the schema is evolved for a workload, small local models write better SQL for that workload than frontier models reading the original schemas, with the evolved 31B and 35B translators overtaking Claude Opus 4.6 while running entirely on local hardware. In the best schemas: every original table stays in place under a documented layer of derived tables. This layer takes the joins out of the model's task, so the generated queries become shorter and simpler, and it differs from model to model: each translator receives a schema shaped around its own way of writing SQL, so the diff between the original and the evolved schema is a fingerprint of that model's weaknesses. The gains concentrate on the workload the evolution can observe, which makes the method a form of workload-driven database tuning for the language-model era. Accuracy is the only objective we optimized here; in a deployment, other properties of the data model matter as much, above all keeping the derived layer consistent with the original tables as the data changes, and such requirements enter the loop simply as further terms of the fitness. And since the evolved schema is an ordinary set of tables and views, better prompting, agents, or fine-tuning can all run on top of it.

The use of large language models

Language models are the object of study in this paper: they serve as the fixed translators being measured and as the mutators driving the evolution, exactly as described in the method and setup sections. In addition, the authors used an AI assistant to help draft manuscript text and analysis scripts. All experimental results, claims, and citations were produced or verified by the authors.

References

Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB)*, pp. 496–505, 2000.

Ruba Almahasneh, Boldizsar Tuu-Szabo, and Laszlo T. Koczy. The extension of the triple fuzzy time dependent travelling salesman problem model, with a discrete bacterial memetic optimization algorithm. *Acta Polytechnica Hungarica*, 22(5), 2025.

Surajit Chaudhuri and Vivek Narasayya. An efficient, cost-driven index selection tool for Microsoft SQL Server. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB)*, 1997.

Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. Text-to-SQL empowered by large language models: A benchmark evaluation. *Proceedings of the VLDB Endowment*, 17(5):1132–1145, 2024.

Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. Shinkaevolve: Towards open-ended and sample-efficient program evolution. In *International Conference on Learning Representations (ICLR)*, 2026.

Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, Victor Zhong, Caiming Xiong, Ruoxi Sun, Qian Liu, Sida Wang, and Tao Yu. Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows. In *International Conference on Learning Representations (ICLR)*, 2025.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin C.C. Chang, Fei Huang, Reynold Cheng, and Yongbin Li. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, et al. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

Mohammadreza Pourreza and Davood Rafiei. DIN-SQL: Decomposed in-context learning of text-to-sql with self-correction. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625:468–475, 2024.

Melanie Sclar, Yejin Choi, Yulia Tsvetkov, and Alane Suhr. Quantifying language models' sensitivity to spurious features in prompt design. In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2310.11324.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *International Conference on Learning Representations (ICLR)*, 2024.

Zijian Győző Yang and Tamás Váradi. Training experimental language models with low resources, for the Hungarian language. *Acta Polytechnica Hungarica*, 20(5), 2023.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2018.

A Training and held-out accuracy over the generations

Figure 3 shows how the accuracy of the best candidate developed over the 20 generations of every run, in the units of Table 3. For each run we take, at each generation, the candidate with the highest fitness F of Eq. 1 evaluated so far, the schema the evolution would deliver if it stopped there. The top row is its accuracy on the training questions as the loop measured it; the bottom row is its accuracy on the held-out questions, which the loop never saw. The held-out values were measured after the runs by evaluating every candidate that became the running best of its run on the held-out split of its database, with the same translator, procedure, and scoring as inside the loop. Within a

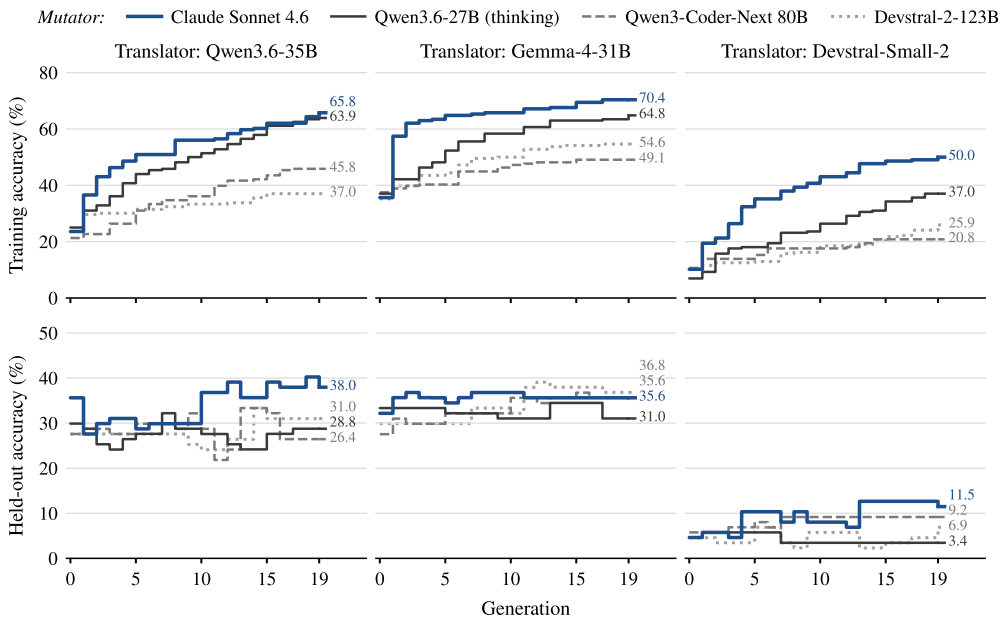


Figure 3: Accuracy of the best candidate over the generations. Each column is one translator; each line is one mutator; the best candidate of a run is the one with the highest training fitness F evaluated.

translator, accuracy is pooled over the questions of the 13 databases, exactly as in Table 3, so the last generation is directly comparable to the table: for the frontier mutator, the held-out accuracy pooled over the three translators goes from 24.1% at generation 0 to 28.4% at generation 19, against 24.1% and 28.0% in the table's pooled held-out column. Generation 0 is the original schema as evaluated inside the loop; that evaluation failed in 29 of the 156 runs, and those runs count zero correct training answers at generation 0, so the training curves start below the baselines of Table 3. The training values are the loop's own measurements of the winning candidates and run slightly above a fresh re-evaluation: 62.0% in generation 19 for the frontier mutator, pooled over the three translators, against the re-evaluated 59.7% of Section 5.1.

On the training questions, most of the gain arrives in the first five generations and the curves keep rising until the end: with the frontier mutator, accuracy moves from 23.6% to 65.8% for Qwen3.6-35B, from 35.6% to 70.4% for Gemma-4-31B, and from 10.2% to 50.0% for Devstral-Small-2, and still rise in generation 19 for two of the three translators. The 27B thinking mutator follows the same shape with a delay of a few generations and ends within two points of the frontier mutator on Qwen3.6-35B (63.9% against 65.8%), six points behind on Gemma-4-31B (64.8% against 70.4%), and thirteen points behind on Devstral-Small-2 (37.0% against 50.0%). The two other local mutators improve more slowly and flatten after generation 15.

On the held-out questions the curves move by a few points and not always upward: when the loop adopts a new training best, the held-out accuracy can fall, because the selection sees only the training questions. With the frontier mutator, the held-out accuracy moves from 35.6% to 38.0% for Qwen3.6-35B, from 32.2% to 35.6% for Gemma-4-31B, and from 4.6% to 11.5% for Devstral-Small-2, while the training accuracy of the same candidates at least doubles. The 27B thinking mutator, whose training curves end close to the frontier mutator's, ends at or below its starting point on every translator (28.8%, 31.0%, and 3.4%), and the two other local mutators gain between one and eight points.