UNI-AST: A SQL-Aligned Universal AST Framework for Robust Cross-Dialect Text-to-SQL

Anonymous ACL submission

Abstract

Building robust Text-to-SQL systems for real-world databases requires handling both semantic parsing and the variation introduced by heterogeneous SQL dialects. Directly generating target-dialect SQL entangles query semantics with dialect-specific surface forms, while ORM-style intermediate representations improve cross-dialect transfer but weaken structural locality, making execution feedback difficult to localize and repair. We present UNI-AST, a cross-dialect Text-to-SQL framework built around a SQL-aligned universal JSON AST. Unlike generic intermediate forms, the SQLGlot-style AST represents columns, predicates, joins, aggregations, ordering clauses, and subqueries as explicit typed nodes, enabling node-level validation, recursive subtree repair, and deterministic dialect-aware rendering. UNI-AST first fine-tunes Qwen2.5-Coder 7B and 14B backbones to generate dialect-agnostic ASTs from questions and database schemas. It then uses target-dialect execution probes to validate AST candidates, applying deterministic white-box correction for diagnosable errors and constrained model-based subtree revision when local rules are insufficient. Finally, the repaired AST is compiled into executable target-dialect SQL through dialect-aware rendering and execution stabilization. Experiments on BIRD, SPIDER, and five SQL dialects show that UNI-AST consistently improves execution accuracy over DIRECT-SQL and the closest dialect-aware baseline, DIALECT-SQL. These results demonstrate that a SQL-aligned universal AST provides an effective generation–repair–rendering interface for robust multi-dialect Text-to-SQL. Code is available at https://anonymous.4open.science/r/UNI-AST.

1 Introduction

Text-to-SQL aims to translate natural language questions into executable SQL queries over relational databases, enabling non-expert users to access structured data through natural language interfaces. Despite strong progress on benchmarks such as SPIDER (Yu et al., 2018b) and BIRD (Li et al., 2023c), practical systems must generalize to unseen schemas, satisfy execution constraints, and operate across heterogeneous database engines. Robustness therefore depends not only on recovering query semantics, but also on realizing them correctly under the target SQL dialect.

A straightforward approach is DIRECT-SQL generation, where the model directly produces SQL in the target dialect. However, this formulation entangles query semantics with dialect-specific surface forms. Semantically equivalent queries may require different identifier quoting, pagination syntax, date/time functions, implicit casts, and engine-specific constraints across database engines. As a result, errors from schema grounding, query structure, and dialect realization are mixed in the final SQL string, making cross-dialect transfer and error recovery brittle.

Intermediate representations offer a natural way to reduce this coupling. Recent cross-dialect work, especially DIALECT-SQL (Shi et al., 2025), uses ORM-style intermediate code to reduce the model's dependence on dialect-specific SQL syntax before producing executable SQL. This is an important step beyond DIRECT-SQL generation because it separates part of semantic construction from target-dialect realization. However, ORM-style code also weakens structural locality: SQL operations are often expressed through object-level APIs or method chains, so projections, predicates, joins, aggregations, ordering clauses, and limits may be hidden behind abstraction layers. This makes it difficult to map a database diagnostic message to a precise SQL component and to repair only the affected part.

We argue that the intermediate representation for cross-dialect Text-to-SQL should not only reduce dialect-specific syntax, but also remain aligned

with the structural units that SQL engines parse, validate, and execute. To this end, we propose UNI-AST, a framework that uses a SQLGlot-style JSON AST (Mao, 2024) as a shared generation–repair–rendering interface. Compared with ORM-style code or generic IRs, the AST keeps columns, predicates, joins, aggregations, ordering clauses, limits, and subqueries as explicit typed nodes with named arguments. This SQL-aligned structure brings three advantages: it enables node-level validation by linking execution feedback to concrete AST components; it supports recursive subtree repair by revising the smallest suspicious subtree before expanding to larger parent scopes; and it provides a natural source for deterministic dialect-aware rendering.

UNI-AST follows this AST-centered design in three stages. First, we supervised fine-tune Qwen2.5-Coder 7B and 14B backbones (Hui et al., 2024) to generate dialect-agnostic SQLGlot-style JSON ASTs from the question and database schema. Second, the system performs recursive execution-validated AST repair: it obtains target-dialect feedback through execution probes, applies deterministic white-box correction for well-defined schema or type errors, and invokes constrained model-based subtree revision only when local rules are insufficient. Third, the repaired AST is compiled into executable target-dialect SQL through dialect-aware rendering and execution stabilization.

The key design principle is *weak binding*. The generated AST is grounded enough for schema-aware validation and localized subtree repair, while dialect-specific realization decisions such as quoting, pagination, functions, and casts are delayed until the rendering stage. This separates the responsibilities of the three stages: Phase 1 learns SQL-compositional structure, Phase 2 corrects detectable AST errors using target-dialect execution feedback, and Phase 3 realizes the corrected structure under concrete database-engine conventions.

We evaluate UNI-AST on BIRD, SPIDER, and five SQL dialects. Results show that UNI-AST consistently outperforms both DIRECT-SQL and the closest dialect-aware baseline, DIALECT-SQL, under 7B and 14B backbones. These results suggest that the gains come not merely from avoiding direct SQL generation, but from using a SQL-aligned universal AST as a unified generation–repair–rendering interface.

Our contributions are threefold:

- We propose UNI-AST, a cross-dialect Text-to-SQL framework that fine-tunes Qwen2.5-Coder backbones to generate SQLGlot-style JSON ASTs and uses the AST as a shared generation–repair–rendering interface.
- We introduce recursive execution-validated AST repair, which exploits the tree structure of the AST to support node-level validation, white-box correction, and constrained model-based subtree revision under target-dialect execution feedback.
- We design a dialect-aware rendering and execution-stabilization stage that compiles the repaired AST into target-dialect SQL, achieving consistent gains over DIRECT-SQL and DIALECT-SQL on BIRD, SPIDER, and five SQL dialects.

2 Related Work

Text-to-SQL and execution feedback. Text-to-SQL has been widely studied for mapping natural language questions to executable SQL over relational databases, including early neural parsers, schema-linking models, constrained decoding methods, and LLM-based self-correction approaches (Zhong et al., 2017; Xu et al., 2017; Yu et al., 2018a; Bogin et al., 2019; Wang et al., 2020; Scholak et al., 2021; Pourreza and Rafiei, 2023; Gao et al., 2024). Benchmarks such as SPIDER (Yu et al., 2018b) and BIRD (Li et al., 2023c) have pushed the field toward unseen-schema generalization and execution-based evaluation, while robustness-oriented benchmarks further expose sensitivity to question and schema perturbations (Gan et al., 2021a; Chang et al., 2023). Execution feedback and constrained decoding have also been used to reduce invalid or non-executable SQL generations (Wang et al., 2018; Scholak et al., 2021; Pourreza and Rafiei, 2023).

Intermediate representations. Intermediate representations reduce the burden of directly generating complex SQL strings by separating semantic parsing from surface realization. IRNet (Guo et al., 2019), NatSQL (Gan et al., 2021b), and related structure-aware parsers show that intermediate or grammar-guided representations can improve cross-domain semantic parsing by making query intent easier to infer (Rubin and Berant, 2021; Scholak et al., 2021; Li et al., 2023b). Code- or

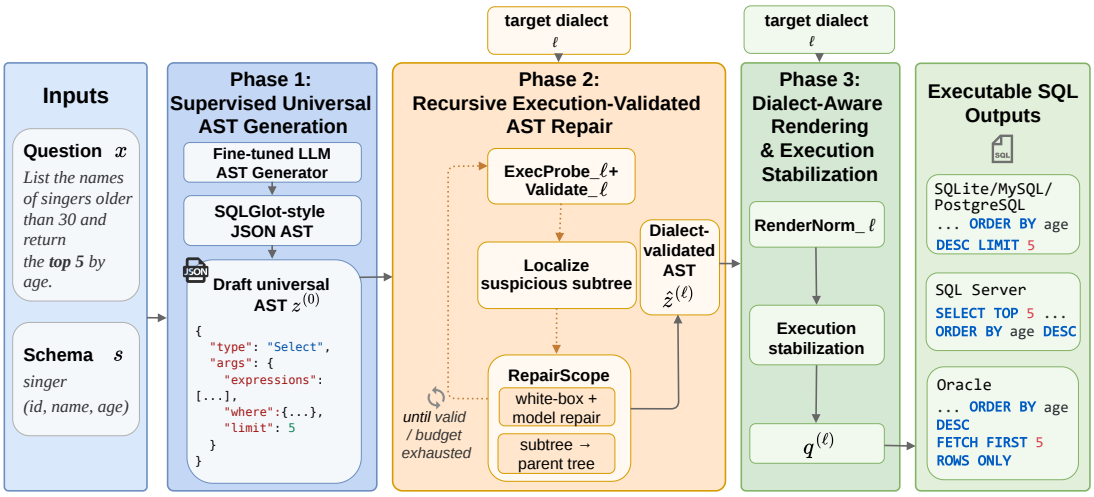

Figure 1: Overview of UNI-AST. The framework first generates a dialect-agnostic SQLGlot-style JSON AST from the question and schema, then performs recursive execution-validated AST repair under the target dialect, and finally renders the repaired AST into executable target-dialect SQL.

API-style forms have also been used to guide Text-to-SQL reasoning and reduce hallucination (Qu et al., 2024, 2025). Yet the utility of an intermediate representation depends on how well it aligns with SQL execution and downstream validation. Generic semantic forms may abstract away execution-relevant details, while representations not directly aligned with relational operators make it harder to map database diagnostics to specific columns, predicates, joins, aggregations, or subqueries. ASTs have also been explored for retrieval-augmented Text-to-SQL, such as AST-based ranking and schema pruning (Shen et al., 2024). In contrast, UNI-AST uses a SQLGlot-style AST as the central generation–repair–rendering interface, where SQL operations are represented as typed nodes with named arguments and can therefore serve as both a generation target and a repairable structure.

Cross-dialect Text-to-SQL. Recent work has studied the dialect gap, where semantically equivalent queries require different surface realizations across database engines. SQL-GEN (Pourreza et al., 2024) improves dialect coverage through synthetic multi-dialect data, while EXESQL (Zhang et al., 2025) emphasizes execution-driven bootstrapping and validation. Beyond Text-to-SQL, CRACKSQL further shows the value of structured SQL processing for dialect translation (Zhou et al., 2025). DIALECT-SQL (Shi et al., 2025) is closest to our setting, using ORM-style intermediate code to reduce dependence on dialect-specific SQL syntax. UNI-AST follows this direction but differs in both the choice and use of the intermediate representation: it predicts a dialect-agnostic SQLGlot-style JSON AST, repairs detectable structural and execution errors at the AST level, and renders the repaired structure into target-dialect SQL through deterministic normalization.

3 Method

We propose UNI-AST, a SQL-aligned universal AST framework for robust cross-dialect Text-to-SQL. Given a natural language question x, a database schema s, and a target SQL dialect ℓ, the goal is to produce an executable SQL query $q^{(\ell)}$ for the database engine of dialect ℓ.

Instead of directly generating target-dialect SQL, UNI-AST factorizes the task into three stages:

$$(x, s) \rightarrow z^{(0)}, \qquad (z^{(0)}, \ell) \rightarrow \hat{z}^{(\ell)} \rightarrow q^{(\ell)}. \quad (1)$$

Here, $z^{(0)}$ is the initial dialect-agnostic AST generated in Phase 1, $\hat{z}^{(\ell)}$ is the AST repaired under dialect-ℓ execution feedback in Phase 2, and $q^{(\ell)}$ is the final target-dialect SQL produced in Phase 3. The target dialect ℓ is not used during initial AST generation, but is used in Phase 2 and Phase 3, where AST candidates must be validated and finally realized under a concrete database engine.

This design follows from the SQL-aligned nature of the representation. Since SQL engines parse, validate, and execute queries through structured components such as columns, predicates, joins, aggregations, ordering clauses, and limits, UNI-AST keeps these components explicit in the AST. This alignment supports three operations that are difficult to perform over raw SQL strings or ORM-style

code: node-level validation, recursive subtree repair, and deterministic dialect-aware rendering.

3.1 SQL-Aligned Universal AST

UNI-AST uses a normalized SQLGlot-style JSON AST (Mao, 2024) as the intermediate representation. Each AST node is represented as

$$n = \{\texttt{"type"} : \tau,\ \texttt{"args"} : \mathcal{A}\} \tag{2}$$

where τ is the SQLGlot expression class name, such as `Select`, `Column`, `Table`, `Where`, or `EQ`, and $\mathcal{A}$ is a dictionary of named arguments. Argument values can be child nodes, lists of nodes, strings, numbers, Boolean values, or `null`. This recursive structure represents projections, predicates, joins, aggregations, grouping, ordering, subqueries, and set operations in a uniform format. The full schema is provided in Appendix A.

The representation is SQL-aligned in the sense that common relational operations are explicit typed nodes rather than implicit spans in a SQL string or hidden calls in an ORM-style abstraction. This locality makes columns, predicates, joins, aggregations, and ordering clauses directly available as validation and repair units. We refer to this property as *weak binding*: the AST is structured enough for schema-aware validation and localized repair, while dialect-specific surface decisions such as quoting, pagination, date/time functions, and type casts are delayed until final rendering.

3.2 Supervised Universal AST Generation

Phase 1 learns to generate the SQL-aligned AST rather than a dialect-specific SQL string. Given the question x and schema s, the fine-tuned generator predicts the initial AST:

$$z^{(0)} = \arg\max_{z} p_\theta(z \mid x, s), \tag{3}$$

where p_θ is parameterized by the supervised fine-tuned Qwen2.5-Coder backbone. Since the target is a dialect-agnostic AST, the dialect ℓ is not provided to the generator.

The supervision target is constructed from gold SQL annotations by parsing each query into a SQLGlot expression tree and serializing it into the normalized JSON AST format. Thus, the model learns compositional SQL structure rather than dialect-specific surface forms. During inference, the generated AST is parsed, sanitized, and passed to Phase 2 as $z^{(0)}$. Details of AST construction, pruning, validation, and fine-tuning are provided in Appendix B.

3.3 Recursive Execution-Validated AST Repair

Phase 2 uses the SQL-aligned AST to perform constrained repair over explicit subtrees. Given $z^{(0)}$ and target dialect ℓ, the goal is to find a nearby AST $\hat{z}^{(\ell)}$ that passes target-dialect validation:

$$\begin{aligned} \hat{z}^{(\ell)} \in \arg \min_{z \in \mathcal{N}(z^{(0)})} & \operatorname{dist}(z, z^{(0)}) \\ \text{s.t.} \quad & \operatorname{Valid}_\ell(z, x, s) = 1. \end{aligned} \tag{4}$$

Here, $\mathcal{N}(z^{(0)})$ denotes the set of ASTs reachable from $z^{(0)}$ through bounded subtree edits, and $\operatorname{dist}(z, z^{(0)})$ denotes the edit scope, approximated by the size or depth of the modified subtree. In practice, UNI-AST approximates this objective by starting from the smallest suspicious subtree and expanding to parent subtrees only when local repair fails.

At repair round t, the current AST $z^{(t)}$ is converted into a temporary probe query for the target engine:

$$\tilde{q}^{(\ell,t)} = \operatorname{ExecProbe}_\ell(z^{(t)}), \tag{5}$$

where $\tilde{q}^{(\ell,t)}$ is used only to obtain execution feedback and is not the final SQL output. The target database returns a validation record:

$$\begin{aligned} r^{(t)} =& \operatorname{Validate}_\ell(\tilde{q}^{(\ell,t)}, x, s) \\ =& \big(b^{(t)}_{\text{parse}}, b^{(t)}_{\text{exec}}, b^{(t)}_{\text{timeout}}, b^{(t)}_{\text{guard}}, m^{(t)}\big). \end{aligned} \tag{6}$$

The Boolean fields indicate parseability, executability, timeout status, and lightweight guardrail status, while $m^{(t)}$ is the database diagnostic message. Here, $b^{(t)}_{\text{guard}}$ covers lightweight checks for schema references, aliases, non-trivial query forms, and question-implied operators such as aggregation or top-k when detectable. The stopping predicate is defined as

$$\operatorname{Stop}(r^{(t)}) = b^{(t)}_{\text{parse}} \wedge b^{(t)}_{\text{exec}} \wedge \neg b^{(t)}_{\text{timeout}} \wedge b^{(t)}_{\text{guard}}. \tag{7}$$

For compactness, we define

$$\begin{aligned} &\operatorname{Valid}_\ell(z, x, s) = \\ &\qquad \operatorname{Stop}\big(\operatorname{Validate}_\ell(\operatorname{ExecProbe}_\ell(z), x, s)\big). \end{aligned} \tag{8}$$

If $\operatorname{Stop}(r^{(t)})$ is true, the current AST is accepted as $\hat{z}^{(\ell)} = z^{(t)}$.

When validation fails, the SQL-aligned structure allows the system to localize the failure to an AST component. The validation record $r^{(t)}$ is mapped

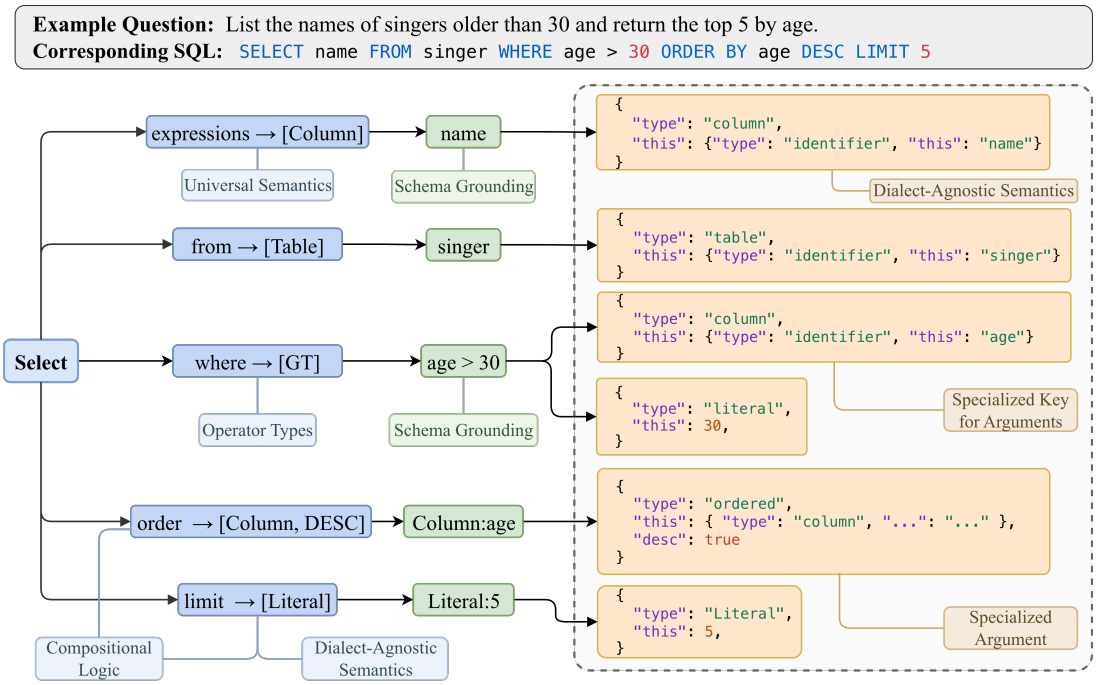

Figure 2: Detailed universal AST in the normalized SQLGlot-style JSON format. Each node contains a `type` field and an `args` dictionary, while nested SQL structures are represented recursively as child nodes.

to a repair category $c^{(t)}$ and the smallest suspicious subtree $u_0^{(t)}$:

$$(c^{(t)}, u_0^{(t)}) = \mathrm{LocalizeSubtree}(r^{(t)}, z^{(t)}, x, s). \tag{9}$$

The repair scope then follows a subtree-to-parent path:

$$u_d^{(t)} = \mathrm{Parent}^d(u_0^{(t)}), \qquad d = 0, \ldots, D, \tag{10}$$

where D is the maximum recursion depth.

At each scope, repair is performed in two ordered steps. First, white-box repair applies deterministic corrections for errors that can be reliably diagnosed from the schema, AST structure, and database message, such as invalid columns, missing table qualifiers, alias conflicts, malformed predicates, and type-incompatible comparisons. If white-box repair cannot produce a dialect-valid AST, UNI-AST invokes constrained model-based subtree repair using the corresponding off-the-shelf Qwen2.5-Coder backbone without the Phase 1 AST-generation SFT. The 7B and 14B settings therefore use non-SFT Qwen2.5-Coder-7B and non-SFT Qwen2.5-Coder-14B repair agents, respectively; no external proprietary LLM is used. The agent receives the question, schema, target dialect, probe query, database diagnostic message, and serialized AST subtree, and is instructed to revise only that subtree. The candidate is accepted only if it passes dialect-ℓ validation, keeping model-based repair constrained by the AST scope, schema, and execution feedback.

Figure 3 shows a representative repair case. The database diagnostic message identifies an invalid reference, the system localizes the corresponding AST subtree, and the recursive repair module revises the subtree before producing executable SQL for the target dialect. The complete pseudocode of the recursive repair loop is provided in Appendix C.3.

3.4 Dialect-Aware Rendering and Execution Stabilization

Phase 3 uses the same SQL-aligned AST as the source for final dialect realization. It serves as a compilation layer from the dialect-validated AST $\hat{z}^{(\ell)}$ to executable target-dialect SQL. Its role differs from Phase 2: Phase 2 changes the AST when the predicted structure is invalid, whereas Phase 3 preserves the repaired structure and realizes it under the conventions of a concrete database engine.

Formally, Phase 3 produces

$$q^{(\ell)} = \mathrm{Finalize}_\ell(\hat{z}^{(\ell)}, s), \tag{11}$$

where $\mathrm{Finalize}_\ell(\cdot)$ consists of structure-preserving rendering and lightweight execution stabilization. The core renderer is

$$q_0^{(\ell)} = \mathrm{RenderNorm}_\ell(\hat{z}^{(\ell)}, s). \tag{12}$$

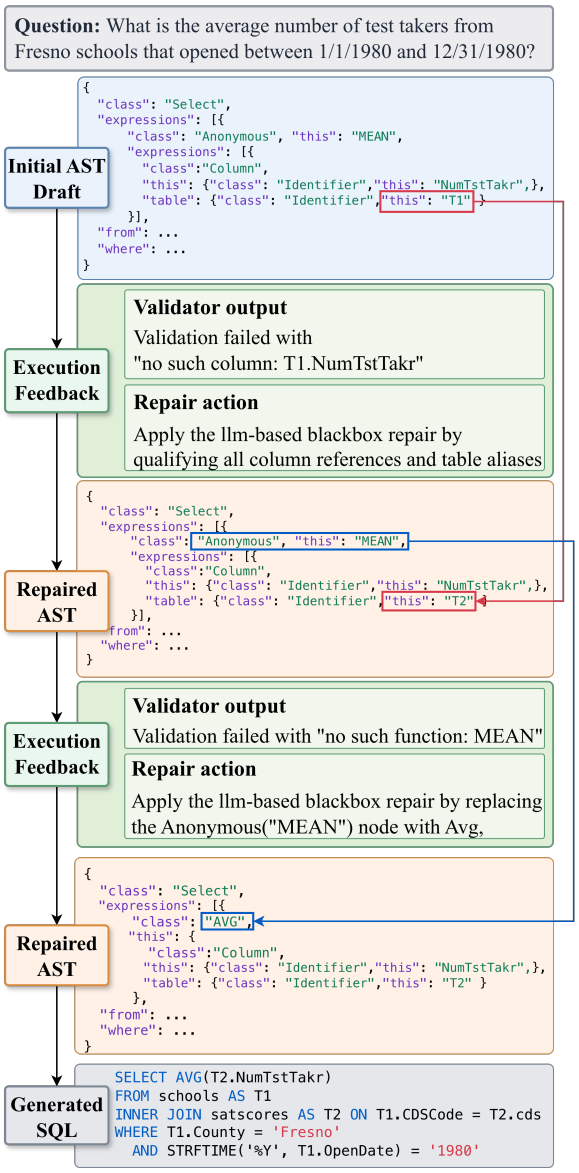

Figure 3: A UNI-AST repair case. Database diagnostic messages expose an invalid attribute reference, after which the recursive repair module revises the suspicious AST subtree and produces executable SQL for different target dialects.

Because rendering is applied at the AST-node level, dialect-specific choices such as pagination, identifier quoting, function names, boolean expressions, date/time operations, and casts are handled as compilation decisions rather than semantic repairs.

After rendering, UNI-AST applies conservative local normalization for values and types and validates the final SQL on the target engine. These operations do not regenerate the query or change unrelated clauses. Overall, Phase 2 improves structural validity through recursive AST repair, while Phase 3 preserves the validated structure and reduces dialect-induced surface-form and runtime failures.

4 Experiments

Our experiments are designed to test three claims. First, direct target-dialect SQL generation suffers from cross-dialect entanglement. Second, ORM-style intermediate representations improve transfer but still lack the structural locality needed for fine-grained validation and repair. Third, the SQL-aligned AST used by UNI-AST supports complementary gains from node-level repair and dialect-aware rendering.

4.1 Experimental Setup

We evaluate on BIRD (Li et al., 2023c) and SPIDER (Yu et al., 2018b). BIRD is our primary benchmark because it emphasizes execution correctness under realistic database conditions, while SPIDER evaluates cross-database generalization over unseen schemas. For cross-dialect evaluation, we follow the multi-dialect BIRD setting of DIALECT-SQL (Shi et al., 2025), which adapts BIRD dev from SQLite to MySQL, SQL Server, Oracle, and PostgreSQL, resulting in five target dialects. We report Execution Accuracy (EX) as the main metric and VES for repair-budget analysis following the official BIRD protocol. On SPIDER, we report EX in the main text and leave Exact Match (EM) to the Appendix G.

We instantiate Phase 1 with Qwen2.5-Coder 7B and 14B backbones (Hui et al., 2024), supervised fine-tuned to generate serialized SQLGlot-style JSON ASTs rather than SQL strings. Phase 2 and Phase 3 are applied at inference time for recursive execution-validated AST repair and dialect-aware rendering with execution stabilization. We use `max_tries=8` by default; full fine-tuning hyperparameters are provided in Appendix B.7.

For Phase 2 model-based repair, we use the corresponding off-the-shelf Qwen2.5-Coder backbone without our AST-generation SFT. Specifically, the 7B and 14B settings use non-SFT Qwen2.5-Coder-7B and Qwen2.5-Coder-14B repair agents, respectively, while Phase 1 uses the fine-tuned AST generators. No external proprietary LLM is used for repair.

We compare with DIRECT-SQL and DIALECT-SQL reproduced from the released DIALECT-SQL code under the same BIRD dev split, target engines, and execution protocol. Agent-style Text-to-SQL frameworks mainly improve decomposition,

Method	Model	SQLite	MySQL	SQL Server	Oracle	PostgreSQL	Avg.
DIRECT-SQL (code repro)	7B	51.04	46.28	39.44	43.94	43.81	44.90
DIALECT-SQL (code repro)	7B	50.59	48.50	49.54	49.22	50.39	49.65
UNI-AST	7B	**59.65**	**57.56**	**56.45**	**54.11**	**54.82**	**56.52**
DIRECT-SQL (code repro)	14B	56.71	53.39	43.48	47.72	49.54	50.17
DIALECT-SQL (code repro)	14B	56.71	56.78	56.71	53.72	57.82	56.35
UNI-AST	14B	**63.89**	**62.32**	**60.43**	**60.37**	**60.82**	**61.57**

Table 1: Main results on multi-dialect BIRD dev following the DIALECT-SQL evaluation setting (Shi et al., 2025). DIRECT-SQL and DIALECT-SQL are reproduced using the released DIALECT-SQL code. UNI-AST uses supervised universal AST generation, recursive execution-validated AST repair, and dialect-aware rendering with execution stabilization. Avg. denotes the mean over five SQL dialects. Bold numbers indicate the best result within the same backbone scale.

retrieval, or self-correction (Wang et al., 2025), and are complementary to our AST interface; we therefore focus on baselines reproducible under the same multi-dialect BIRD protocol.

4.2 Does UNI-AST Reduce Cross-Dialect Entanglement?

Table 1 follows the progression from direct SQL generation to ORM-style intermediate code and then to the proposed SQL-aligned AST interface. DIRECT-SQL obtains the lowest average EX at both 7B and 14B scales, confirming that directly generating target-dialect SQL suffers from the entanglement between query semantics and dialect-specific surface forms. DIALECT-SQL improves over DIRECT-SQL by using ORM-style intermediate code, showing that intermediate representations help reduce dialect dependence.

UNI-AST achieves the best EX on all five dialects under both backbone scales. At the 7B scale, UNI-AST improves the average score from 44.90 to 56.52 over DIRECT-SQL and from 49.65 to 56.52 over DIALECT-SQL. At the 14B scale, UNI-AST reaches 61.57 average EX, outperforming DIRECT-SQL by 11.40 points and DIALECT-SQL by 5.22 points. These gains indicate that the advantage of UNI-AST does not merely come from using an intermediate representation, but from using a SQL-aligned AST whose structure supports validation, repair, and dialect-aware rendering.

4.3 Do SQL-Aligned Repair and Rendering Matter?

Table 2 studies whether the SQL-aligned AST is useful beyond being a generation target. The lower EX of the AST-only setting shows that generating a structured representation alone is insufficient for robust execution. Adding recursive AST repair improves EX from 48.89 to 54.50, supporting the claim that SQL alignment helps localize execution feedback to repairable subtrees. Keeping dialect-aware rendering and execution stabilization without recursive repair reaches 56.13 EX, showing that the same AST also provides a useful source for dialect-specific compilation. The full model achieves 59.65 EX, demonstrating that subtree-level repair and dialect-aware rendering are complementary uses of the same SQL-aligned representation.

Setting	Simple	Mod.	Chall.	Total
AST only	58.92	34.91	29.66	48.89
+ Repair	64.00	40.95	37.24	54.50
+ Rendering	65.62	42.03	40.69	56.13
Full	**68.43**	**47.41**	**42.76**	**59.65**

Table 2: Ablation study on BIRD dev under SQLite. "AST only" removes both recursive AST repair and dialect-aware execution stabilization. "+ Repair" keeps recursive AST repair without execution stabilization, while "+ Rendering" keeps dialect-aware rendering and execution stabilization without recursive repair.

Setting	EX	ExecErr	Mismatch	Avg. rounds
No repair	56.13	105	567	1.00
White-box only	56.71	89	574	1.12
Model-only	56.98	82	577	1.34
White + model	58.21	60	580	1.45
Full method	**59.65**	**45**	573	1.58

Table 3: Ablation of recursive execution-validated AST repair components on BIRD dev under SQLite. ExecErr and Mismatch denote execution-error and executable result-mismatch cases.

Table 3 further analyzes Phase 2. White-box repair reduces execution-error cases by applying deterministic corrections to diagnosable schema, alias, column, and type errors. Model-based repair adds flexibility for cases beyond fixed rules, while remaining constrained to the selected AST scope. Combining both gives the best EX and reduces

execution errors from 105 to 45, while requiring only 1.58 repair rounds on average. This indicates that the gain does not come from aggressive retrying, but from localizing errors to repairable AST scopes and applying bounded subtree-level corrections. Detailed error statistics are provided in Appendix F.

4.4 How Much Repair Is Needed?

Table 4 studies the cost–accuracy trade-off of recursive execution-validated repair. Increasing `max_tries` from 3 to 16 raises the average number of repair rounds from 1.17 to 2.22, but changes EX only from 59.52 to 59.78. We therefore use `max_tries=8` by default, which captures most gains with moderate repair cost.

`max_tries`	EX	ΔEX	VES	ΔVES	Avg. rounds
3	59.52	–	60.92	–	1.17
8	59.65	+0.13	60.96	+0.04	1.58
12	59.45	-0.07	60.67	-0.25	1.96
16	59.78	+0.26	61.06	+0.14	2.22

Table 4: Effect of the recursive AST repair budget on BIRD dev under SQLite. ΔEX and ΔVES are computed relative to `max_tries` $= 3$. Larger repair budgets yield only marginal gains, showing clear diminishing returns beyond a moderate budget.

4.5 Cross-Dialect Robustness across Repair Budgets

Figure 4 separates dialect-aware AST compilation from recursive execution repair. Phase 1+3 already improves over DIRECT-SQL, while adding Phase 2 further improves all dialects, showing that rendering and localized repair provide complementary cross-dialect robustness.

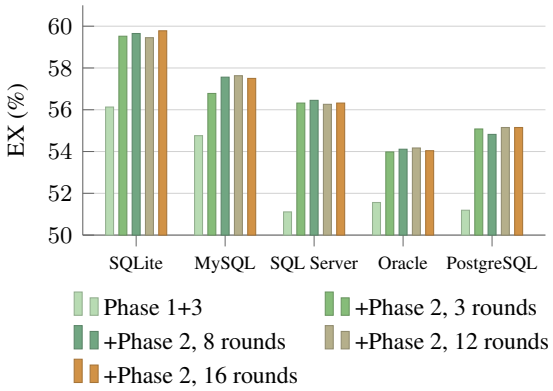

Figure 4: Cross-dialect EX on BIRD dev under different UNI-AST variants and repair-budget settings. Each bar group corresponds to one SQL dialect.

4.6 Generalization on Spider

Table 5 reports EX on SPIDER dev. Although SPIDER is not a multi-dialect benchmark, it tests cross-domain compositional generalization over unseen schemas. UNI-AST outperforms both DIRECT-SQL and DIALECT-SQL at 7B and 14B scales, indicating that the SQL-aligned AST interface is not only useful for dialect adaptation, but also improves structured query prediction on unseen databases.

At the 7B scale, UNI-AST achieves 83.8 total EX, outperforming DIRECT-SQL by 8.8 points and DIALECT-SQL by 5.5 points. At the 14B scale, UNI-AST reaches 85.7 total EX, outperforming DIRECT-SQL and DIALECT-SQL by 6.8 and 3.4 points, respectively. The gains are especially clear on hard and extra-hard examples, showing that the benefits of the SQL-aligned AST interface persist with stronger backbones and more complex query structures.

Method	Easy	Med.	Hard	Extra	Total
DIRECT-SQL 7B	88.7	82.5	69.5	40.4	75.0
DIALECT-SQL 7B	89.9	85.7	63.2	57.2	78.3
UNI-AST 7B	**94.0**	**91.5**	74.1	57.8	83.8
DIRECT-SQL 14B	87.5	87.0	70.7	53.0	78.9
DIALECT-SQL 14B	90.7	89.0	75.3	59.0	82.3
UNI-AST 14B	**95.2**	**91.1**	**79.2**	**64.1**	**85.7**

Table 5: Execution accuracy on SPIDER dev under 7B and 14B backbones.

5 Conclusions

This paper presented UNI-AST, a SQL-aligned universal AST framework for robust cross-dialect Text-to-SQL. UNI-AST addresses the semantic–dialect entanglement of DIRECT-SQL generation and the weak structural locality of ORM-style intermediate representations by using a SQLGlot-style JSON AST as a shared generation–repair–rendering interface.

By aligning the representation with SQL components that database engines parse, validate, and execute, UNI-AST supports node-level validation, recursive subtree repair, and deterministic dialect-aware rendering. Experiments on BIRD, SPIDER, and five SQL dialects show consistent gains over DIRECT-SQL and DIALECT-SQL, demonstrating the value of SQL-aligned ASTs for robust multi-dialect Text-to-SQL. These results suggest that SQL-aligned intermediate structures offer a practical basis for execution-aware and dialect-portable semantic parsing.

Limitations

UNI-AST has several limitations. First, although the SQL-aligned AST provides better structural locality than SQL strings or ORM-style IRs, the framework still depends on the quality of the initial AST; globally misaligned predictions may be hard to recover through local subtree repair. Second, node-level validation is limited by the specificity of database diagnostic messages, and execution-valid queries can still be semantically wrong. Third, dialect-aware rendering depends on the coverage of normalization rules, so rare engine-specific behaviors may require additional handling. Finally, recursive execution-validated repair introduces extra inference cost because each repair round may involve execution probing, validation, and subtree revision.

References

Ben Bogin, Jonathan Berant, and Matt Gardner. 2019. Representing schema structure with graph neural networks for text-to-SQL parsing. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4560–4565, Florence, Italy. Association for Computational Linguistics.

Shuaichen Chang, Jun Wang, Mingwen Dong, Lin Pan, Henghui Zhu, Alexander Hanbo Li, Wuwei Lan, Sheng Zhang, Jiarong Jiang, Joseph Lilien, Steve Ash, William Yang Wang, Zhiguo Wang, Vittorio Castelli, Patrick Ng, and Bing Xiang. 2023. Dr.Spider: A diagnostic evaluation benchmark towards text-to-SQL robustness. In *Proceedings of the International Conference on Learning Representations*.

Yujian Gan, Xinyun Chen, Qiuping Huang, Matthew Purver, John R. Woodward, Jinxia Xie, and Pengsheng Huang. 2021a. Towards robustness of text-to-SQL models against synonym substitution. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing*, pages 2505–2515. Association for Computational Linguistics.

Yujian Gan, Xinyun Chen, Jinxia Xie, Matthew Purver, John R. Woodward, John Drake, and Qiaofu Zhang. 2021b. Natural sql: Making sql easier to infer from natural language specifications. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 2030–2042. Association for Computational Linguistics.

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

Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. 2019. Towards complex text-to-sql in cross-domain database with intermediate representation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4524–4535. Association for Computational Linguistics.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, and 5 others. 2024. Qwen2.5-Coder technical report. *Preprint*, arXiv:2409.12186.

Haoyang Li, Shang Wu, Xiaokang Zhang, Xinmei Huang, Jing Zhang, Fuxin Jiang, Shuai Wang, Tieying Zhang, Jianjun Chen, Rui Shi, Hong Chen, and Cuiping Li. 2025. OmniSQL: Synthesizing high-quality text-to-sql data at scale. *Proc. VLDB Endow.*, 18(11):4695–4709.

Haoyang Li, Jing Zhang, Cuiping Li, and Hong Chen. 2023a. RESDSQL: Decoupling schema linking and skeleton parsing for text-to-SQL. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 13067–13075.

Haoyang Li, Jing Zhang, Hanbing Liu, Ju Fan, Xiaokang Zhang, Jun Zhu, Renjie Wei, Hongyan Pan, Cuiping Li, and Hong Chen. 2024. CodeS: Towards building open-source language models for text-to-sql. *Proceedings of the ACM on Management of Data*, 2(3):1–28.

Jinyang Li, Binyuan Hui, Reynold Cheng, Bowen Qin, Chenhao Ma, Nan Huo, Fei Huang, Wenyu Du, Luo Si, and Yongbin Li. 2023b. Graphix-T5: Mixing pre-trained transformers with graph-aware layers for text-to-SQL parsing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 13076–13084.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Ma Chenhao, Guoliang Li, Kevin Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023c. 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*, volume 36, pages 42330–42357. Curran Associates, Inc.

Xi Victoria Lin, Richard Socher, and Caiming Xiong. 2020. Bridging textual and tabular data for cross-domain text-to-SQL semantic parsing. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 4870–4888. Association for Computational Linguistics.

Toby Mao. 2024. SQLGlot: Python sql parser and transpiler. `https://github.com/tobymao/sqlglot`. Accessed: 2026-05-24.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, and 2 others. 2019. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, volume 32.

Mohammadreza Pourreza and Davood Rafiei. 2023. DIN-SQL: Decomposed in-context learning of text-to-sql with self-correction. In *Advances in Neural Information Processing Systems*, volume 36, pages 36339–36348.

Mohammadreza Pourreza, Ruoxi Sun, Hailong Li, Lesly Miculicich, Tomas Pfister, and Sercan O. Arik. 2024. SQL-GEN: Bridging the dialect gap for text-to-sql via synthetic data and model merging. *Preprint*, arXiv:2408.12733.

Ge Qu, Jinyang Li, Bowen Li, Bowen Qin, Nan Huo, Chenhao Ma, and Reynold Cheng. 2024. Before generation, align it! a novel and effective strategy for mitigating hallucinations in text-to-SQL generation. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 5456–5471, Bangkok, Thailand. Association for Computational Linguistics.

Ge Qu, Jinyang Li, Bowen Qin, Xiaolong Li, Nan Huo, Chenhao Ma, and Reynold Cheng. 2025. SHARE: An SLM-based hierarchical action CorREction assistant for text-to-SQL. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 11268–11292, Vienna, Austria. Association for Computational Linguistics.

Ohad Rubin and Jonathan Berant. 2021. SmBoP: Semi-autoregressive bottom-up semantic parsing. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 311–324. Association for Computational Linguistics.

Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. PICARD: Parsing incrementally for constrained auto-regressive decoding from language models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 9895–9901. Association for Computational Linguistics.

Zhili Shen, Pavlos Vougiouklis, Chenxin Diao, Kaustubh Vyas, Yuanyi Ji, and Jeff Z. Pan. 2024. Improving retrieval-augmented text-to-SQL with AST-based ranking and schema pruning. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 7865–7879, Miami, Florida, USA. Association for Computational Linguistics.

Jie Shi, Xi Cao, Bo Xu, Jiaqing Liang, Yanghua Xiao, Jia Chen, Peng Wang, and Wei Wang. 2025. Dialect-SQL: An adaptive framework for bridging the dialect gap in text-to-SQL. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 3604–3619. Association for Computational Linguistics.

Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2020. RAT-SQL: Relation-aware schema encoding and linking for text-to-sql parsers. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7567–7578. Association for Computational Linguistics.

Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Linzheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, and Zhoujun Li. 2025. MAC-SQL: A multi-agent collaborative framework for text-to-SQL. In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 540–557. Association for Computational Linguistics.

Chenglong Wang, Kedar Tatwawadi, Marc Brockschmidt, Po-Sen Huang, Yi Mao, Oleksandr Polozov, and Rishabh Singh. 2018. Robust text-to-sql generation with execution-guided decoding. *Preprint*, arXiv:1807.03100.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, and 3 others. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45. Association for Computational Linguistics.

Xiaojun Xu, Chang Liu, and Dawn Song. 2017. SQLNet: Generating structured queries from natural language without reinforcement learning. *Preprint*, arXiv:1711.04436.

Jiaxi Yang, Binyuan Hui, Min Yang, Jian Yang, Junyang Lin, and Chang Zhou. 2024. Synthesizing text-to-SQL data from weak and strong LLMs. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, pages 7864–7875. Association for Computational Linguistics.

Tao Yu, Michihiro Yasunaga, Kai Yang, Rui Zhang, Dongxu Wang, Zifan Li, and Dragomir Radev. 2018a. SyntaxSQLNet: Syntax tree networks for complex and cross-domain text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1653–1663. Association for Computational Linguistics.

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. 2018b. 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*, pages 3911–3921. Association for Computational Linguistics.

Jipeng Zhang, Haolin Yang, Kehao Miao, Ruiyuan Zhang, Renjie Pi, Jiahui Gao, and Xiaofang Zhou. 2025. ExeSQL: Self-taught text-to-SQL models with execution-driven bootstrapping for SQL dialects. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 24305–24326. Association for Computational Linguistics.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, and Zheyan Luo. 2024. LlamaFactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pages 400–410, Bangkok, Thailand. Association for Computational Linguistics.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2SQL: Generating structured queries from natural language using reinforcement learning. *Preprint*, arXiv:1709.00103.

Wei Zhou, Yuyang Gao, Xuanhe Zhou, and Guoliang Li. 2025. Cracksql: A hybrid sql dialect translation system powered by large language models. *Preprint*, arXiv:2504.00882.

A AST Serialization Schema

UNI-AST serializes SQLGlot-style expression trees into a normalized JSON AST that serves as the shared interface between AST generation, execution-validated repair, and dialect-aware rendering. The goal of this serialization is not to introduce a new SQL grammar, but to expose the SQLGlot expression tree in a compact, machine-readable format that can be generated by a language model and consumed by deterministic post-processing modules.

Each AST node is represented as a JSON object with two fields: `type` and `args`. The `type` field records the SQLGlot expression class of the node, such as `Select`, `Column`, `Table`, `Where`, `EQ`, `Join`, `Order`, or `Limit`. The `args` field stores the named arguments of this node. These arguments recursively encode the children and attributes of the expression. For example, a `Select` node may contain projected expressions, a `from_` clause, a `where` clause, an `order` clause, and a `limit` clause, while a `Column` node may contain an `Identifier` child that specifies the column name.

Formally, each node follows the schema

$$n = \{\texttt{"type"} : \tau,\ \texttt{"args"} : A\},$$

Field	Type	Description
`type`	string	SQLGlot expression class name. Typical examples include `Select`, `Column`, `Table`, `Where`, `EQ`, `Join`, `Order`, and `Limit`.
`args`	object	A dictionary of named arguments for the current expression node. These arguments encode the node's children and attributes.
Node value	object	A nested AST node with its own `type` and `args` fields, used to represent recursive SQL structures.
List value	list	An ordered list of child nodes, commonly used for projected columns, join clauses, grouping keys, or order expressions.
Scalar value	string / number / Boolean	A primitive value used for identifiers, literal values, flags, or other node attributes.
`null`	null	An empty or optional argument value. During pruning, non-informative `null` fields are removed when they are not needed for reconstruction.

Table 6: Core fields and value types of the normalized SQLGlot-style JSON AST used by UNI-AST.

where τ is the SQLGlot expression type and A is a dictionary from argument names to argument values. An argument value can be another AST node, a list of AST nodes, a scalar value, or `null`. This recursive design makes nested SQL structures such as predicates, joins, aggregations, subqueries, set operations, ordering clauses, and limits explicit in the serialized representation.

Table 6 summarizes the core fields of the normalized AST. The representation preserves SQL-relevant structure while avoiding dialect-specific surface forms such as quoting conventions, pagination syntax, and engine-specific function names. These surface decisions are deferred to the dialect-aware rendering stage.

A simplified example is shown below. The query structure is represented by typed nodes rather than by a dialect-specific SQL string. The projected column, source table, filter predicate, ordering clause, and limit are all encoded as explicit substructures:

```
{
  "type": "Select",
  "args": {
    "expressions": [ {"type": "Column", "args": {...}} ],
    "from_": {"type": "From", "args": {...}},
    "where": {"type": "Where", "args": {...}},
    "order": {"type": "Order", "args": {...}},
    "limit": {"type": "Limit", "args": {...}}
  }
}
```

This format has three practical advantages. First, it gives the generator a stable JSON target instead of requiring it to directly emit dialect-specific SQL. Second, it provides explicit repair units: when execution feedback points to an invalid column, malformed predicate, missing table reference, or incompatible expression, the corresponding subtree can be localized and revised. Third, it gives the final renderer a structured input from which dialect-specific SQL can be produced by normalizing identifiers, functions, casts, and pagination syntax for the target engine.

B Fine-tuning Details

B.1 Fine-tuning Objective

The goal of fine-tuning is to train the backbone model to generate a serialized SQLGlot-style JSON AST rather than a SQL string. Given a question x and a database schema s, the model is trained to predict the AST target z:

$$z = \arg\max_{z} p_\theta(z \mid x, s).$$

The target sequence is constrained to be machine-readable JSON content following the schema in Appendix A, without SQL strings or natural language explanations. This follows the recent trend of improving smaller Text-to-SQL models through task-specific supervised data construction and training (Li et al., 2024; Yang et al., 2024; Li et al., 2025), but our supervision target is a SQLGlot-style JSON AST rather than a SQL string.

B.2 Construction of AST Supervision

We construct AST supervision from gold SQL annotations in the training split. In our preprocessing pipeline, each raw training example contains an original SQL annotation stored in the `SQL` field. The preprocessing script reads this field and converts the gold SQL into a normalized SQLGlot-style JSON AST:

```
converter = SQLASTConverter(dialect="sqlite")

gold_sql = item["SQL"]
gold_ast = converter.sql_to_ast(gold_sql)

item["gold_ast"] = gold_ast
```

The conversion from gold SQL to AST is implemented in two stages. First, the SQL string is parsed by SQLGlot under the SQLite dialect:

```
expression = sqlglot.parse_one(sql_str, read=self
    .dialect)
```

where `self.dialect` is set to `"sqlite"` for the original BIRD training annotations. This step converts the gold SQL string into a SQLGlot expression tree. Second, the SQLGlot expression tree is recursively serialized into the normalized `type/args` JSON format used by UNI-AST:

```
{
  "type": type(expression).__name__,
  "args": args_json
}
```

Thus, the overall supervision construction process is:

gold SQL string
→ `sqlglot.parse_one`
→ SQLGlot Expression AST
→ `_expr_to_json`
→ JSON AST target.

The resulting AST is not stored as a native SQLGlot object. Instead, it is saved as a plain JSON object. For example, a parsed `Select` expression is serialized as:

```
{
  "type": "Select",
  "args": {
    "expressions": [...],
    "from_": {...},
    "where": {...}
  }
}
```

After conversion, the processed training example keeps the original fields and adds the new `gold_ast` field:

```
{
  "db_id": "...",
  "question": "...",
  "SQL": "...",
  "gold_ast": {
    "type": "Select",
    "args": {...}
  }
}
```

Finally, the supervised fine-tuning data are built by reading the processed examples with `gold_ast`.

The assistant-side target is the JSON serialization of `gold_ast`, rather than the original SQL string:

```
target_ast = item_ast.get("gold_ast")
assistant_output = json.dumps(target_ast,
     ensure_ascii=False)
```

Therefore, UNI-AST is trained to generate the normalized SQLGlot-style JSON AST from the question and schema. The original SQL annotation is used only to construct AST supervision in the training split.

B.3 AST Pruning and Validation

After converting gold SQL into the normalized SQLGlot-style JSON AST, we further apply pruning rules to make the supervision target compact, stable, and suitable for supervised fine-tuning.

Initial pruning during AST serialization. During the recursive conversion from SQLGlot expressions to JSON AST nodes, we remove empty or non-informative argument values. In particular, when serializing a SQLGlot expression into the `type/args` format, we skip argument values that are `None`, empty lists, or empty dictionaries:

```
if v is None or (isinstance(v, (list, dict)) and
     len(v) == 0):
    continue
```

This step removes obvious redundant fields while preserving the recursive structure of the SQLGlot expression tree.

Advanced AST pruning. We then apply a second pruning step over the whole serialized AST. This step recursively removes non-essential fields, including `None` values, empty arrays, empty objects, comments, and redundant default quotation flags:

- remove `None`;
- remove empty lists `[]`;
- remove empty objects `{}`;
- remove `comments`;
- remove `quoted:false`.

Importantly, the pruning step does not treat all false-like values as removable. Values such as `0`, `false`, `quoted:true`, and `is_string:false` are preserved when they carry semantic information. For example, `is_string:false` in a SQLGlot literal can indicate a numeric literal rather than a string literal, and removing it may affect SQL reconstruction.

The pruning rule can be summarized as follows:

```
if k == "quoted" and v is False:
    continue

if k == "comments":
    continue

cleaned_v = prune_ast_advanced(v)

if cleaned_v is not None:
    if isinstance(cleaned_v, (list, dict)) and
     len(cleaned_v) == 0:
        continue
    new_dict[k] = cleaned_v
```

Round-trip validation after pruning. The pruned AST is not saved blindly. We validate that pruning does not break SQL reconstruction. Specifically, the pruned AST is first reconstructed into SQL, and the reconstructed SQL is compared with the original gold SQL after normalization:

$$\text{pruned AST} \rightarrow \text{reconstructed SQL} \rightarrow \text{normalized SQL comparison.}$$

The SQL normalization is performed with SQLGlot by transpiling both the original SQL and reconstructed SQL into the SQLite dialect. If the normalized strings are not identical, we further compare the parsed SQLGlot expression trees. Only examples that pass this round-trip validation are kept in the final pruned AST training file.

Thus, AST pruning is not a blind compression step. It follows a prune–reconstruct–validate workflow:

$$\text{gold SQL} \rightarrow \text{JSON AST} \rightarrow \text{pruned AST} \rightarrow \text{reconstructed SQL} \rightarrow \text{validation.}$$

This ensures that redundant fields are removed while preserving the information needed for SQL reconstruction.

Overall, the final supervision construction pipeline is:

$$\text{train SQL} \rightarrow \text{SQLGlot AST} \rightarrow \text{JSON AST} \rightarrow \text{AST pruning} \rightarrow \text{round-trip validation} \rightarrow \text{JSON AST target.}$$

The AST pruning step removes redundant SQLGlot fields while preserving semantic values, so that the supervised target remains compact, stable, and faithful to the original SQL annotation.

B.4 AST Pruning Statistics

Table 7 reports the pruning statistics on the BIRD training set. The pruning procedure is applied to all 9,428 training examples. After pruning, each AST is reconstructed into SQL and validated against the original gold SQL through SQLGlot-based normalization or expression-tree comparison. All examples pass the round-trip validation, indicating that the pruning rules remove redundant fields without breaking SQL reconstruction.

Item	Value
Total samples	9,428
Valid after pruning	9,428
Broken after pruning	0
Original AST size	18,813,138 chars
Pruned AST size	16,615,517 chars
Compression rate	11.68%

Table 7: AST pruning statistics on the BIRD training set. The compression rate is computed at the character level. All pruned ASTs pass round-trip validation.

B.5 Instruction Template

All fine-tuning and inference examples use a unified instruction template, shown in Figure 5. The template specifies the input fields, the required SQLGlot-style JSON AST format, and strict output constraints. By requiring a single parseable JSON object and explicitly disallowing SQL strings or natural language explanations, the template makes the model output directly consumable by the downstream AST parser, repair loop, and dialect-aware renderer.

During supervised fine-tuning, the assistant-side target is the serialized `gold_ast` JSON object. During inference, the same template is used to prompt the model, and the generated content is parsed as a JSON AST before entering the execution-validated AST repair and dialect-aware rendering stages. This shared template keeps training and inference aligned and reduces format drift between the supervised target and the deployment-time output.

B.6 Decoding and Output Sanitization

During inference, the model is prompted with the same instruction format used in fine-tuning. We extract the generated JSON-style AST from the model output and check whether it can be parsed into the expected `type`/`args` structure. The extracted AST is then sanitized by applying the same pruning and normalization rules used during target construction. If the output cannot be parsed as a valid AST, the example is passed to subsequent fallback or repair logic. Our supervised fine-tuning is conducted with LLaMA-Factory (Zheng et al., 2024), and our implementation uses PyTorch and the HuggingFace Transformers library for model fine-tuning and inference (Paszke et al., 2019; Wolf et al., 2020).

```
Task:
You are given a database schema and a natural
     language question. Generate the SQL abstract
     syntax tree (AST)
in JSON using the SQLGlot-style format.

Output Rules:
- Return ONLY a single valid JSON object.
- Do NOT output SQL.
- Do NOT output any explanation.
- The JSON must be parseable by a standard JSON
     parser.

AST Format:
Every node must be a JSON object:
  {"type": "<SQLGlotExpressionName>", "args": {...
     }}

"type" is the SQLGlot Expression class name, e.g.,
Select, Column, Table, Where, EQ, Order, Limit,
     Literal,
Identifier, Join, Subquery, Union, Case, Window.

"args" is a dictionary of node arguments. Values
     can be:
  - another node object,
  - a list of node objects,
  - a string, number, boolean, or null.

Constraints:
- Only use tables and columns in the provided
     schema.

Database Schema:
{schema_text}

Question:
{question_text}

Return:
AST JSON:
```

Figure 5: Instruction template for generating SQLGlot-style universal ASTs.

B.7 Fine-tuning Configuration

Table 8 summarizes the fine-tuning configuration used for training the universal AST generators. We use the same training target and hyperparameter setting for both the 7B and 14B backbones so that the comparison between model scales mainly reflects the effect of backbone capacity rather than differences in optimization or data construction.

Item	Value
Backbone	Qwen2.5-Coder-7B / 14B
Training target	SQLGlot-style JSON AST
GPUs	8 × NVIDIA A100 40GB
Learning rate	5×10^{-6}
Epochs	2.0
Max sequence length	4096
Per-device batch size	1
Gradient accumulation	16

Table 8: Fine-tuning configuration for UNI-AST. The 7B and 14B AST generators use the same training target and hyperparameter settings unless otherwise specified.

Signal	Role in repair
Parseability	Detect malformed probe queries
Runtime executability	Detect database execution failures
Timeout detection	Detect inefficient or non-terminating queries
DB diagnostics	Provide messages for subtree repair
Schema/question guardrails	Check lightweight structural constraints

Table 9: Inference-time validation signals used by the recursive execution-validated AST repair loop.

The model is trained to generate SQLGlot-style JSON ASTs instead of SQL strings, which keeps the supervised target aligned with the generation–repair–rendering interface used by UNI-AST.

The maximum sequence length is set to 4096 to accommodate database schemas, natural-language questions, and serialized AST targets within a unified instruction template. We use a small per-device batch size due to the memory cost of long-context fine-tuning and increase the effective batch size through gradient accumulation. The same fine-tuned AST generator is then used as the Phase 1 model during inference, while Phase 2 model-based repair uses the corresponding non-SFT Qwen2.5-Coder backbone, as described in Appendix C.2.

C Recursive Repair Details

C.1 Validation Signals Used in Repair

Table 9 summarizes the inference-time validation signals used by the recursive execution-validated AST repair loop. These signals are used to detect parse-time, runtime, timeout, schema, and lightweight structural failures during repair. Execution-based validation follows the common practice in Text-to-SQL evaluation and self-correction, where executability and result consistency provide feedback signals for query refinement (Wang et al., 2018; Pourreza and Rafiei, 2023; Gao et al., 2024).

C.2 Constrained Model-Based Repair Prompt

When deterministic white-box repair is insufficient, UNI-AST invokes a constrained model-based repair module. The repair model is the corresponding non-SFT Qwen2.5-Coder backbone, rather than the Phase 1 SFT AST generator.

The repair model is prompted as a compiler-style agent to revise either a local AST fragment or, as a fallback, the whole AST. The prompt provides the database schema, user question, target dialect, execution or validation error, schema-aware negative feedback, the current probe query produced by $\mathrm{ExecProbe}_\ell$, and the AST fragment to be repaired. This encourages localized, schema-grounded correction rather than unconstrained SQL regeneration.

Figure 6 shows the prompt template.

The dynamic fields are instantiated from the current repair state. The `schema_prompt`, `dialect`, and `question` fields provide the schema, target SQL dialect, and original question. If the question cannot be extracted, it is set to `(unknown)`. When available, a bounded reasoning hint is inserted as an optional CoT block:

```
Previous Reasoning (CoT Hint):
{cot_hint}
```

The `error_msg` field contains the execution or validation error, and `probe_query` is the temporary query produced by $\mathrm{ExecProbe}_\ell$. The `fragment_str` field is the AST fragment serialized in the `type/args` JSON format. The `negative_feedback` field is generated by schema-aware validation and contains the authoritative schema, diagnostic warnings, and repair constraints:

```
Valid Schema (authoritative)
{schema_hint}

Rules:
- Do NOT invent tables/columns not in schema.
- Use Identifier.args.quoted=true for reserved
    names
  or names containing spaces/symbols.
- Keep changes minimal; preserve overall query
    intent.
```

```
Task Overview:
You are a compiler/repair agent.
Given a database schema, a user question,
and an incorrect SQLGlot JSON AST, repair the AST.

Target Dialect:
{dialect}

Database Schema:
{schema_prompt}

User Question:
{question}

{optional_cot_block}

Execution or Validation Error:
{error_msg}

Negative Feedback & Valid Schema:
{negative_feedback}

Current Probe Query:
{probe_query}

Incorrect AST Fragment:
```json
{fragment_str}
```

Repair Instructions:
{task_desc}

Think step by step briefly.
Then output one ```json code block containing
the repaired JSON AST.

Output Format:
Thought:
[short reasoning]

```json
{... repaired JSON AST ...}
```
```

Figure 6: Prompt template for constrained model-based repair of invalid SQLGlot-style AST fragments.

When the database reports invalid or ambiguous references, such as `no such column`, `no such table`, or `ambiguous column name`, the negative feedback block additionally inserts a warning to guide the model toward schema-valid corrections.

The repair instruction `task_desc` depends on the repair scope. For local fragment repair, the model must preserve the top-level node type:

```
Fix ONLY the given AST fragment while keeping
SQLGlot JSON AST type/args format.
The fragment top-level 'type' MUST stay the same.
```

For whole-AST repair, the model may rewrite the full tree but must preserve the question intent and use only schema-valid tables and columns:

```
Rewrite the WHOLE AST to match the user question
    intent.
The result must be executable in the target
    dialect.
Use only existing tables/columns from schema.
```

Although the prompt allows a short reasoning field, the system extracts only the repaired JSON AST from the final `json` block and validates it using the inference-time validation signals described in Appendix C.1.

C.3 Recursive Repair Algorithm

Algorithm 1 Recursive execution-validated AST repair.

Require: Initial AST $z^{(0)}$, question x, schema s, target dialect ℓ, repair-round budget K, recursion depth D

Ensure: Repaired AST $\hat{z}^{(\ell)}$, dialect-validated if validation succeeds

for $t = 0, \ldots, K-1$ **do**
  $\tilde{q}^{(\ell,t)} \leftarrow \mathrm{ExecProbe}_\ell(z^{(t)})$
  $r^{(t)} \leftarrow \mathrm{Validate}_\ell(\tilde{q}^{(\ell,t)}, x, s)$
  if $\mathrm{Stop}(r^{(t)})$ **then**
    return $\hat{z}^{(\ell)} \leftarrow z^{(t)}$
  end if
  $(c^{(t)}, u_0^{(t)}) \leftarrow \mathrm{LocalizeSubtree}(r^{(t)}, z^{(t)}, x, s)$
  $z_{\mathrm{cand}} \leftarrow \varnothing$
  for $d = 0, \ldots, D$ **do**
    $u_d^{(t)} \leftarrow \mathrm{Parent}^d(u_0^{(t)})$
    $z_{\mathrm{cand}} \leftarrow \mathrm{WhiteBoxRepair}(z^{(t)}, u_d^{(t)}, c^{(t)}, r^{(t)}, s, \ell)$
    if $z_{\mathrm{cand}} \neq \varnothing$ **then**
      break
    end if
    $z_{\mathrm{cand}} \leftarrow \mathrm{ModelRepair}(z^{(t)}, u_d^{(t)}, c^{(t)}, r^{(t)}, x, s, \ell)$
    if $z_{\mathrm{cand}} \neq \varnothing$ **then**
      break
    end if
  end for
  if $z_{\mathrm{cand}} \neq \varnothing$ **then**
    $z^{(t+1)} \leftarrow z_{\mathrm{cand}}$
  else
    $z^{(t+1)} \leftarrow \mathrm{WholeASTRepair}(z^{(t)}, r^{(t)}, x, s, \ell)$
  end if
end for
return $\hat{z}^{(\ell)} \leftarrow z^{(K)}$ ▷ Return the last repaired AST if no valid AST is found within budget.

The returned AST is guaranteed to be dialect-validated only when the algorithm terminates through the stopping condition. If the repair budget is exhausted, the algorithm returns the last repaired AST candidate, which is then handled by the subsequent rendering, validation, or fallback logic. This bounded design prevents uncontrolled retrying while still allowing errors to be corrected at increasingly larger AST scopes.

D Baseline Reproduction Details

We reproduce DIRECT-SQL and DIALECT-SQL using the publicly released implementation of DIALECT-SQL (Shi et al., 2025). Both baselines are evaluated under the same multi-dialect BIRD setting used in the main experiments, covering SQLite, MySQL, SQL Server, Oracle, and PostgreSQL.

DIRECT-SQL directly generates target-dialect SQL, while DIALECT-SQL first generates ORM-style intermediate code and then converts it into dialect-specific SQL. For multi-dialect BIRD evaluation, we follow the DIALECT-SQL setting: predictions on non-SQLite engines are executed on the corresponding target database, and the results are compared with the execution results of the original SQLite gold SQL.

All reproduced baselines are evaluated on the same development split and with the same evaluation protocol as UNI-AST. This setup ensures that the reported improvements are attributed to the proposed SQLGlot-style AST interface, execution-validated AST repair, and dialect-aware rendering rather than to differences in evaluation data or metrics.

E Repair-Budget Details

This appendix provides the full difficulty-wise EX Table 10 and BIRD-VES statistics Table 11 under different execution-validated AST repair budgets. These results complement Table 4 in the main text.

Setting	Simple	Moderate	Chall.	EX	VES
`max_tries=3`	68.43	47.20	42.07	59.52	60.92
`max_tries=8`	68.43	**47.41**	**42.76**	59.65	60.96
`max_tries=12`	68.32	46.98	**42.76**	59.45	60.67
`max_tries=16`	**68.76**	47.20	**42.76**	**59.78**	**61.06**
DIALECT-SQL	58.59	40.73	31.03	50.59	51.25

Table 10: Difficulty-wise EX and overall EX/VES results on BIRD dev under SQLite with different values of `max_tries`. Larger recursive AST repair budgets yield only marginal gains.

F Additional Error Analysis

Table 12 summarizes the remaining errors of UNI-AST on BIRD dev under SQLite with `max_tries=8`. We separate failures into execution errors and executable result mismatches to characterize the effect of execution-validated AST repair.

Setting	Simple	Moderate	Chall.	Total
`max_tries=3`	68.79	**50.90**	42.76	60.92
`max_tries=8`	68.80	50.84	43.31	60.96
`max_tries=12`	68.76	49.83	**43.69**	60.67
`max_tries=16`	**69.12**	50.44	43.60	**61.06**

Table 11: Difficulty-wise BIRD-VES results of UNI-AST on BIRD dev under SQLite with different values of `max_tries`. Larger recursive AST repair budgets yield only marginal gains.

Outcome	7B	14B
Total examples	1534	1534
Correct EX	915	980
EX accuracy	59.65	63.89
Result mismatch	573	536
Execution error	45	16
Gold unreliable	1	2

Table 12: Error bucket analysis on BIRD dev under SQLite with `max_tries=8`.

Table 13 further breaks down executable result mismatches. Most remaining mismatches are content differences, indicating that the dominant residual errors are executable but semantically different from the gold query.

Mismatch subtype	7B	14B
Content difference	294	299
Row-count difference	124	110
Column-count difference	49	55
Column + row-count difference	106	72

Table 13: Breakdown of executable result mismatches on BIRD dev under SQLite.

Table 14 reports the execution-error categories. Invalid column references are the main execution error, and they are substantially reduced when moving from 7B to 14B.

Overall, scaling the backbone reduces detectable runtime failures, especially invalid schema references. The remaining errors are mainly executable result mismatches, suggesting that further gains require stronger semantic disambiguation and schema grounding beyond execution-validity signals (Lin et al., 2020; Li et al., 2023a).

G Additional SPIDER Exact-Match Results

Table 15 reports exact-match (EM) results on SPIDER dev. We include EM as an auxiliary structural metric because it is stricter than execution accu-

Execution-error category	7B	14B
No such column	39	12
No such table	2	0
Misuse of aggregate	4	0
Ambiguous column	0	1
Syntax error	0	1
Incomplete input	0	1
Other execution error	0	1

Table 14: Execution-error categories after the full UNI-AST pipeline on BIRD dev under SQLite.

racy (EX) and is sensitive to SQL surface form, canonicalization choices, and evaluation-script assumptions. Therefore, the main paper uses EX as the primary metric, while EM is reported here as supplementary evidence for structural regularity.

Overall, UNI-AST achieves substantially higher EM than both direct SQL generation and DIALECT-SQL. Compared with DIALECT-SQL, UNI-AST improves the total EM from 45.2 to 74.9. The gains are especially large on Medium, Hard, and Extra-Hard queries, where compositional structures such as nested queries, aggregation, set operations, and complex filtering are more frequent. This suggests that the AST-centered formulation helps produce more canonical and structurally stable SQL, rather than only improving executability. In contrast, DIALECT-SQL improves over DIRECT-SQL but still directly generates SQL strings, leaving the query structure more exposed to surface-form variation and structural mismatches.

These results complement the SPIDER EX results in the main paper. While EX measures whether the predicted SQL executes to the correct answer, EM provides an additional view of whether the generated query matches the gold SQL structure under the official canonicalization procedure. The strong EM performance of UNI-AST therefore supports our claim that generating a unified structured AST before SQL rendering improves not only execution correctness but also structural consistency.

Method	Easy	Med.	Hard	Extra	Total
DIRECT-SQL	75.4	43.9	24.7	7.2	42.4
DIALECT-SQL	78.2	48.1	26.2	7.8	45.2
UNI-AST	**89.1**	**84.3**	**58.6**	**45.2**	**74.9**

Table 15: Exact-match (EM) results on SPIDER dev. EM is reported as an auxiliary structural metric and is not used as the primary basis for comparison in the main paper.