Policy-Conditioned Constrained Decoding for Column-Level Access Control in Text-to-SQL

Ryoto Miyamoto **Xin Fan** **Hayato Yamana**
Waseda University
r-miyamoto@toki.waseda.jp

Abstract

Text-to-SQL is increasingly deployed across trust boundaries between data providers and users. Such deployment must balance three competing requirements: policy compliance, answer coverage, and bounded cost. Existing approaches typically decide refusal based on which columns a query mentions and enforce it stochastically. Whether a query is compliant, however, depends not only on which columns appear but on how they are used, and stochastic enforcement cannot deterministically rule out violations. We formalize this requirement as a column-use policy over semantic use: output, filter condition, and aggregation argument. We integrate the policy by aligning each role with grammar productions tracked by the decoder. The resulting system, PCC-SQL, applies a per-token logits mask that deterministically eliminates single-query column-use violations on the supported SQL fragment in a single decoding pass. Across three benchmarks and three open-source models, PCC-SQL achieves 0% Leakage Rate and Coverage up to 88.7% on Spider-CU, while staying within +10% tokens of direct prompting. We additionally assess semantic alignment with execution accuracy.

1 Introduction

Querying databases in natural language has been a goal of NLP for decades (Androutsopoulos et al., 1995). Recent LLMs have brought this within reach: text-to-SQL has moved from research benchmark to practical deployment (Yu et al., 2018; Li et al., 2023; Hong et al., 2025). At the same time, LLMs increasingly serve as natural language interfaces to external systems (Yao et al., 2023; Schick et al., 2023; Patil et al., 2024). Together, these trends bring text-to-SQL into settings where the data provider and the user issuing queries sit on opposite sides of the trust boundary, including SaaS operational analytics, enterprise data portals, and analytical support for healthcare, financial, and government data. In such settings, the generated SQL must not only respond to the user's request but also satisfy the disclosure policy specified by the data provider.

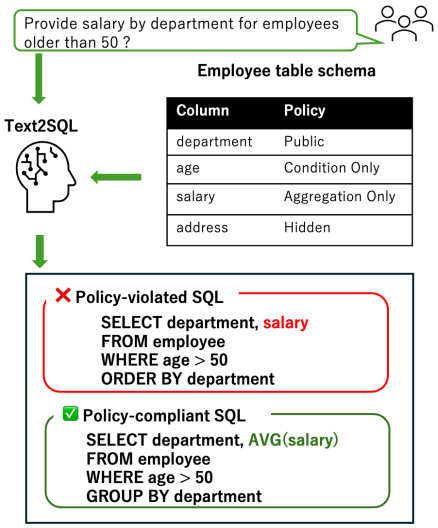


Figure 1: When `salary` is AGGREGATEONLY, placing it directly in `SELECT` (top) violates the policy, while using it only as the argument of `AVG()` (bottom) is compliant. The same NL request thus has both a violating and a compliant SQL form; only the column's role distinguishes them.

To deploy text-to-SQL in this setting, three requirements must be met simultaneously. First, the generated SQL must not violate the policy. Second, whenever the user's request admits a policy-compliant SQL, the system should produce one rather than refuse, so that coverage of admissible requests stays high. Third, the response latency and computational cost must remain low enough to support sustained online serving. These three requirements conflict with one another: stricter refusal lowers coverage, while broadening the set of admitted responses requires additional violation checking or regeneration steps and inflates computational cost. Post-hoc remediation is locked into this trade-off because each verification call or retry

Proceedings of the 15th International Joint Conference on Natural Language Processing and the 5th Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics, pages 5377–5391
November 6-10, 2026 ©2026 Association for Computational Linguistics

pays in tokens for violation reduction; meeting all three therefore calls for a design that prevents violations at generation time, rather than handling them afterwards.

Policy-compliant text-to-SQL is a critical concern for trustworthy deployment, with multiple methods proposed (Klisura et al., 2026; Abedini et al., 2025; Liu et al., 2026). While these methods represent important progress, two gaps remain on the way to satisfying all three requirements: how a policy is expressed, and how compliance is achieved. First, the policy vocabulary operates at the level of column mention (whether a column appears in the query) and does not yet distinguish the role a column plays within the SQL. As a result, when the same column mention can yield either a violating or a compliant form depending on its role (Figure 1), the policy is forced into a binary choice between refusing the column (lowering coverage) and admitting it (allowing leakage). Second, existing methods pursue compliance through training-time alignment or input-side preprocessing, which cannot deterministically rule out violations on unseen inputs. Driving the residual leakage to zero requires post-hoc verification with regeneration, which inflates token cost and stands in tension with the low-cost requirement. Closing this trade-off therefore calls for jointly extending both the policy vocabulary and the compliance mechanism.

We address both gaps with two complementary contributions. On the policy side, we introduce a role-based column-use policy. On the decoding side, we propose PCC-SQL (Policy-Conditioned Constrained SQL Generation), a constrained decoder that applies the policy in a single decoding pass. The column-use policy assigns each column one of four roles: PUBLIC (may appear as an output column), HIDDEN (may not appear in any SQL context), CONDITIONONLY (only in condition clauses), and AGGREGATEONLY (only as an aggregation argument). This extends column-level access controls used in commercial databases, such as Snowflake's projection policies (Snowflake Inc., n.d.), to a granularity that depends on a column's role within the query. The resulting per-role granularity preserves paths to compliant forms that mention-based policies cannot capture, maintaining coverage. PCC-SQL leaves the token-exclusion mechanism of grammar-constrained decoding (Scholak et al., 2021; Ugare et al., 2025a) unchanged and instead contributes the constraint it enforces, in which the columns to exclude are determined by the policy and the role at the current position. At each decoding step, it tracks the role context implied by the current SQL grammar position and uses a logits mask to exclude columns disallowed under that role from the next-token candidates. The mechanism completes in a single decoding pass without post-hoc verification or regeneration, deterministically eliminating within-query violations while keeping token cost low. Together, the two contributions meet all three requirements in a single decoding pass.

Contributions.

- **Column-use policy** formalized along four permission types and four roles, with annotated Spider-CU and BIRD-CU.
- **PCC-SQL**, a constrained decoder that deterministically eliminates single-query column-use policy violations on the SQL fragment we consider, in a single decoding pass.
- **Evaluation** on three open-source models across Spider-CU, BIRD-CU, and Spider-ACL against four baselines spanning prompting, regeneration, post-hoc verification (Klisura et al., 2026), and step-level rollback: 0% Leakage, Coverage up to 88.7% on Spider-CU, within +10% tokens of direct prompting.[1]

2 Related Work

2.1 Text-to-SQL safety and policy compliance

Attack and leakage detection. Research on text-to-SQL safety has expanded rapidly in recent years (Hong et al., 2025), with research focused on identifying attack surfaces, including schema inference from output SQL (Klisura and Rios, 2025), detection of malicious prompts and SQL (Song et al., 2024b), and backdoor attacks via training data poisoning (Lin et al., 2025). SecureSQL (Song et al., 2024a) provides a benchmark that systematically evaluates sensitive data leakage in LLM-based natural language interfaces to databases (NLIDBs).

These studies characterize, detect, or benchmark policy-violating SQL after it has been generated. Detection alone does not prevent the violating SQL from being produced, leaving deployment to filter or regenerate downstream at the cost of additional LLM calls.

[1] Code and benchmarks are available at https://github.com/yamanalab/pcc-sql.

Privacy-preserving generation. Rather than detecting policy violations after the fact, several methods target compliant SQL at generation time: Refuse under role-based access control policies (Klisura et al., 2026), abstraction of sensitive tokens before sending them to external LLMs (Abedini et al., 2025), and safety alignment that combines security-aware data synthesis with iterative preference optimization (Liu et al., 2026). Of these, Klisura et al. (2026) is closest, addressing column-level access on single queries (used as our external benchmark Spider-ACL); SafeNLIDB targets multi-query inference attacks via the ShieldSQL benchmark. Their policy vocabulary is limited to per-column binary visibility or row filtering, and cannot distinguish whether the same column appears as an output, a condition, or an aggregation argument; compliance also relies on stochastic training-time alignment or post-generation abstraction.

2.2 Database access control

Database research has long addressed context-dependent access control and controlled release of aggregate results. Examples include purpose-based fine-grained access control (Xue et al., 2020), dynamic information flow control in database systems (Guarnieri et al., 2019), and release of aggregate results under differential privacy (Johnson et al., 2018).

These designs operate primarily at the database execution layer or middleware layer. To enforce equivalent control at the generation stage, where the LLM composes SQL from natural language, both the policy vocabulary and the mechanism that prevents violations must be brought to the model side.

2.3 Constrained / structured generation

Constrained decoding intervenes in the per-step token distribution so that the output satisfies a given constraint. Many prior studies focus on ensuring syntactic or schema validity by excluding constraint-violating tokens from the next-token candidates, including SQL grammar constraints (Scholak et al., 2021), structured output formats such as JSON or Python (Dong et al., 2025; Ugare et al., 2025b), and language-level query syntax (Zheng et al., 2024). In contrast, IterGen (Ugare et al., 2025a) alternates generation and verification at the symbol level and rolls back on failure, while RAIN (Li et al., 2024) performs decoding-time alignment via self-evaluation and rollback. Both rely on post-generation verification.

These methods are primarily designed for syntactic validity or general-purpose alignment, and applying them to security constraints such as policy violations at decoding time has been reported only to a limited extent. It has also been observed that hard constraints can have side effects on the generation distribution (output-quality skew correlated with the constraint) (Banerjee et al., 2025).

3 Preliminaries

This section formalizes column-use policy compliance as a decision based on a column's policy π and its usage role ρ within SQL.

The four usage roles ρ correspond to syntactic positions. SEL marks projections in `SELECT/GROUP BY/ORDER BY`, JOIN marks predicates inside `JOIN …ON`, WHERE marks predicates inside `WHERE/HAVING` outside aggregation, and AGG marks arguments of the set-aggregation functions `COUNT`, `SUM`, and `AVG`. The four column policies π are PUBLIC, CONDITIONONLY, AGGREGATEONLY, and HIDDEN, and the permission relation $\text{perm}(\pi, \rho)$ is defined in Table 1.

	Usage role ρ			
Policy π	SEL	JOIN	WHERE	AGG
PUBLIC	✓	✓	✓	✓
CONDITIONONLY	—	✓	✓	—
AGGREGATEONLY	—	—	—	✓
HIDDEN	—	—	—	—

Table 1: Permission relation $\text{perm}(\pi, \rho)$. Rows are column policies π, columns are usage roles ρ. ✓ permitted, — prohibited.

For a policy assignment $P : \text{Col} \to \text{Policy}$, the safety of a query q is $\text{safe}(q, P) \Leftrightarrow \forall (c, \rho) \in \text{cols}(q),\ \text{perm}(P(c), \rho) = \top$, where $\text{cols}(q)$ collects the role-tagged column occurrences in q. On the SQL fragment considered in this paper (specified in §8), the role is uniquely determined by the syntactic position, reducing role assignment to tracking the SQL prefix during generation.

4 Method: PCC-SQL

PCC-SQL (Policy-Conditioned Constrained SQL Generation) is a constrained decoder that outputs policy-compliant SQL in a single decoding pass. The method builds on a key property of SQL: at any column-name position, the role is determined

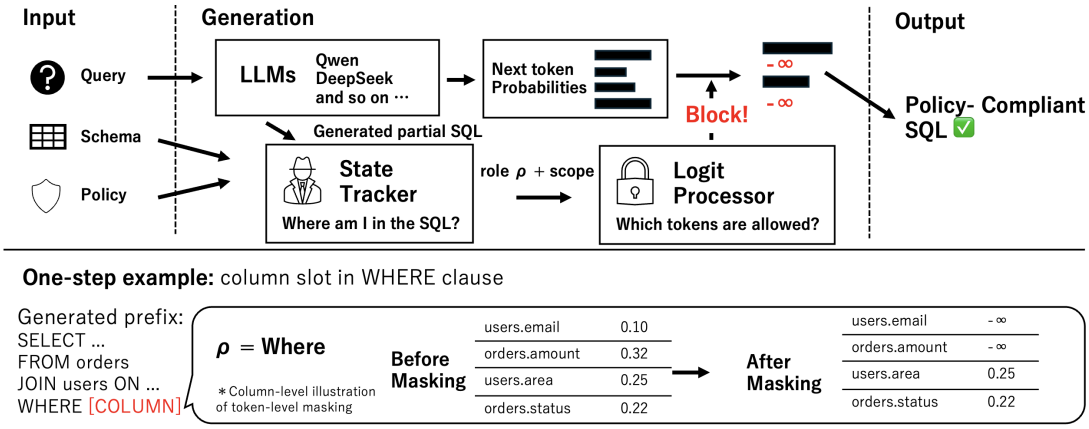


Figure 2: Architecture and a token-level masking step. Top: the State Tracker derives the role ρ from the partial SQL, and the Logit Processor masks the logits of disallowed columns to $-\infty$. Bottom: at a column slot inside `WHERE` ($\rho = \textsc{Where}$), policy-violating columns (`users.email`, `orders.amount`) are masked, so the next token is sampled only from permitted columns (`users.area`, `orders.status`).

solely from the prefix. For instance, a column inside the `SELECT` clause has role SEL, and a column inside `JOIN …ON` has JOIN. PCC-SQL consists of two components. (1) The state tracker maintains the current role from the syntactic position. (2) The logits processor compares each column's policy against the role at column-name positions and keeps only the permitted columns among the next-token candidates. The decoder generates only tokens permitted by the logits processor. Figure 2 illustrates the architecture and a single-step example.

4.1 Correspondence between roles and grammar

The state tracker maintains a small set of SQL grammar state during generation, such as the current clause and the aggregation nesting state. At each column-name position, it uses this state to assign one of SEL / JOIN / WHERE / AGG, which then drives the masking described in §4.2.

4.2 Mask construction and the Refuse path

At each column-name position, given the scope (the in-scope tables at the current subquery level) produced by the state tracker and the current role ρ, we compute the allowed column set

$$\text{Allowed} = \{c \in \text{Scope} \mid \text{perm}(P(c), \rho) = \top\} \quad (1)$$

The logits processor sets the logits of column-name tokens outside Allowed to $-\infty$, leaving the decoder to generate next tokens only from permitted columns. When Allowed is empty, no permitted column is available at the current column-name position, and PCC-SQL terminates with Refuse instead of completing the SQL. At each column-name position, the choice is therefore binary: either output a permitted column, or, if no permitted alternative exists, exit through Refuse instead of emitting a violation.

4.3 Deterministic guarantee

Combining the prefix-determined role assignment (§3) with the masking rule of Eq. 1, every token emitted at a column-name position belongs to Allowed. PCC-SQL therefore satisfies the safe predicate of §3 deterministically in a single decoding pass on the SQL fragment we consider. Full implementation details are in Appendix B.

5 Experimental Setup

This section presents the benchmarks, models, baselines, decoding settings, and evaluation metrics.

5.1 Benchmarks

We evaluate on three benchmarks. Two extend existing text-to-SQL benchmarks with our column-use policy, and one is an external benchmark for column-access tasks:

- **Spider-CU** ($N = 1{,}034$, four-permission): the Spider (Yu et al., 2018) dev split extended with our column-use policy.
- **BIRD-CU** ($N = 1{,}534$, four-permission): the BIRD (Li et al., 2023) dev split extended with our column-use policy.

- **Spider-ACL** (Klisura et al., 2026) ($N = 19{,}624$, binary-permission): a role-based policy extension of Spider whose datasets we translate into PUBLIC/HIDDEN labels.

We use *four-permission* for policies that draw on all of PUBLIC / CONDITIONONLY / AGGREGATEONLY / HIDDEN, and *binary-permission* for policies restricted to PUBLIC and HIDDEN. The latter precludes verification of role-dependent permissions and therefore restricts such evaluation to Spider-CU and BIRD-CU.

Spider-CU and BIRD-CU are controlled four-permission stress tests for role-sensitive enforcement, while Spider-ACL tests transfer to an independently released binary column-access benchmark from the closest prior work.

Spider-CU and BIRD-CU Construction. We synthesize column-use policies by combining column-name regex (flagging PII-like attributes such as email, phone, ssn) with SEL / JOIN / WHERE / AGG occurrence counts in train/dev gold SQL. Names signal semantic sensitivity, usage counts reveal consumption patterns, the two signals a human policy author would rely on. The full assignment procedure is in Appendix C.

Distribution perturbations. To measure robustness to shifts in the policy distribution, the assignment admits two probability parameters that perturb columns toward HIDDEN or PUBLIC (definitions in Appendix C). The main results use the base assignment with no perturbation. Sensitivity to non-zero values is examined in §6.4.

We additionally evaluate on Spider-ACL, an external benchmark whose annotations are also rule-based. Our policy annotations are not verified against human-labeled gold. Full schema, dataset, and policy-distribution details, including the effect of the four-permission split, are in Appendix D.

5.2 Models

We evaluate three open-source models (Qwen 2.5 Coder 7B / 32B Instruct (Hui et al., 2024), DeepSeek Coder 6.7B Instruct (Guo et al., 2024)) for both PCC-SQL and the prompting-based baselines, plus two API-accessed models (Claude Haiku4.5 (Anthropic, 2025a) / Opus4.5 (Anthropic, 2025b)) as reference points for the baselines on large commercial models. PCC-SQL requires logits intervention and is therefore restricted to the open-source models. Decoding uses temperature $= 0$. All methods share the same prompt format (schema + policy + question).

5.3 Baselines

We compare PCC-SQL (§4) against four baselines:

- **direct prompting**: single output from the generation prompt, with no defense.
- **retry** ($N = 3$): regenerate up to three times when a violation is detected. Sensitivity of N from 1 to 10 is reported in §6.4.
- **2-step verifier** (Klisura et al., 2026): re-verify the generated SQL with the same model and fall back to Refuse on violation (prompts in Appendix A).
- **IterGen + role check**: at each SQL identifier boundary, run our role check and policy comparison and roll back on failure. Like PCC-SQL, this requires logits intervention and is evaluated only on the open-source models.

5.4 Metrics

Each output is classified as one of four values. **SafeSQL** parses, satisfies the policy, and executes without error. **UnsafeSQL** parses but violates the policy. **Refuse** is an explicit refusal output. **Failure** covers parse failures, runtime errors, empty outputs, and provider exceptions.

The gold label is binary: **SQL-gold** marks queries whose natural SQL interpretation is admissible, and **Refuse-gold** marks queries whose interpretation would violate the policy. For Refuse-gold queries, a policy-satisfying *safe alternative* (e.g., `AVG(salary)` in place of individual `salary`) may still be constructible on the schema. Returning SafeSQL counts as success: a correct answer on SQL-gold queries, or a safe alternative on Refuse-gold queries.

We report:

- **Leakage Rate** (↓): fraction of UnsafeSQL outputs over all queries.
- **Coverage** (El-Yaniv and Wiener, 2010) (↑): fraction of SafeSQL outputs over all queries. We additionally split this by gold label into **Recall** on SQL-gold queries and **Recovery Rate** on Refuse-gold queries, so that Coverage is a weighted combination of the two by gold-label counts. The upper bound on Recovery Rate is set by the fraction of Refuse-gold queries with a constructible safe alternative.

- **Failure Rate** (↓): fraction of Failure outputs over all queries.

Cost metrics (↓): **Calls/query** (LLM calls per query), **Tokens/query** (tokens processed per query), **Tokens / Safe SQL Return** (total tokens divided by SafeSQL count).

6 Results

We compare PCC-SQL with the four baselines on three benchmarks (Spider-CU, BIRD-CU, Spider-ACL) and report two ablations covering the retry count and the benchmark probability parameters.

6.1 Results on Spider-CU and BIRD-CU

PCC-SQL is the only method reaching 0% Leakage Rate on every open-source model on both benchmarks (Tables 2, 3). The single baseline cell that also touches 0%, retry ($N = 3$) on Qwen 2.5 Coder 32B on Spider-CU, does so by Refuse-conversion. Its Recovery Rate stays at 1.14%, 74.00 points below PCC-SQL on the same model. On BIRD-CU every baseline leaves residual leakage.

PCC-SQL also leads on Coverage of recoverable refusals. Against Recovery Rate upper bounds of 99.4% (Spider-CU) and 100% (BIRD-CU), the two Qwen models reach 72–88% across both benchmarks, while the single 0%-reaching baseline (retry, Qwen 32B on Spider-CU) attains only 1.14%. DeepSeek Coder 6.7B is an outlier at 58.29 / 36.99% (Spider-CU / BIRD-CU), reflecting model-dependent variation that §7 analyzes.

6.2 Results on Spider-ACL

Spider-ACL (Table 10, Appendix F) is an external benchmark whose policy annotations are derived independently of ours: we translate the GRANT statements released by Klisura et al. (2026) into a binary PUBLIC / HIDDEN scheme over 19,624 queries. The Safety picture matches Spider-CU and BIRD-CU. PCC-SQL is again the only method reaching 0% Leakage Rate on all three open-source models, while every baseline leaks: direct prompting at 1.69–40.90%, retry ($N = 3$) at 0.19–0.32%, the 2-step verifier of Klisura et al. (2026) at 0.48–28.76%, and IterGen at 3.51–7.34%.

Coverage follows the same pattern. Against an upper bound of 97.1%, PCC-SQL on Qwen 2.5 Coder 7B reaches Answer Recall 95.03% and Recovery Rate 80.64%, over 51 points above the strongest baseline Recovery Rate (IterGen on Qwen 2.5 Coder 32B, 29.28%). On the same Qwen 7B cell, the 2-step verifier of Klisura et al. (2026), designed specifically for this binary scheme, remains at Leakage Rate 13.53% and Recovery Rate 2.95%, illustrating that detection-and-refusal alone cannot recover the safe alternatives that PCC-SQL constructs at decoding time.

Because Spider-ACL uses binary PUBLIC / HIDDEN, role-dependent permission cases are not exercised. The result therefore corroborates that the constrained-decoding framework extends beyond our annotation rules, but does not independently validate the role-dependent claim itself.

6.3 Semantic correctness check

SafeSQL captures policy compliance and successful execution, not whether the output matches the question's intent. To check that PCC-SQL's Coverage is not inflated by safely-executable but off-target SQL, we additionally compute execution accuracy on SQL-gold queries (predictions that parse, execute, and return the gold SQL's execution result). On Qwen 2.5 Coder 32B, PCC-SQL reaches 48.2% on Spider-CU and 18.2% on BIRD-CU; the BIRD-CU figure is the best result among open-source methods on Qwen 32B, and on Spider-CU PCC-SQL outperforms every Qwen 32B baseline except IterGen. On Qwen 2.5 Coder 7B and DeepSeek Coder 6.7B, PCC-SQL trails the prompting baselines by 5–18 points but stays in the same range. PCC-SQL therefore trades a moderate amount of execution match on weaker models for deterministic compliance, rather than falling back to off-target SQL. The full per-method breakdown is in Table 11 (Appendix G).

6.4 Ablation

We report two ablations: how retry's residual Leakage Rate and token budget evolve as N grows, and how PCC-SQL behaves when the probability parameters of our benchmark construction are perturbed away from the worst-case assignment.

Sensitivity of retry to the retry count. We sweep $N \in \{1, 2, 3, 5, 10\}$ on Spider-CU and BIRD-CU (Table 4). Qwen 2.5 Coder 7B and DeepSeek Coder 6.7B plateau above 0% even at $N = 10$, while Qwen 2.5 Coder 32B reaches 0% only by Refuse-conversion, with Recovery Rate held at 1.14% / 1.69% (Tables 2, 3). Token cost grows with N when violations are frequent, whereas PCC-SQL eliminates leakage in a single pass.

		Safety	Answer			Failure	Cost		
Method	**Model**	**Leak↓**	**Coverage↑**	**Recall↑**	**Recovery↑**	**Fail↓**	**Calls↓**	**Tok/q↓**	**Tok/Safe↓**
direct prompting	qwen7b	30.66	50.10	74.27	2.86	1.26	1.00	525	1,048
direct prompting	deepseek67b	42.36	50.77	73.39	6.57	6.58	1.00	719	1,416
direct prompting	qwen32b	4.84	25.82	38.30	1.43	**0.00**	1.00	**499**	1,934
direct prompting	claude-haiku	22.82	49.03	70.91	6.29	0.58	1.00	629	1,284
direct prompting	claude-opus	20.41	58.22	84.50	6.86	**0.00**	1.00	627	1,076
retry (N=3)	qwen7b	0.39	50.19	74.27	3.14	10.06	1.32	740	1,474
retry (N=3)	deepseek67b	0.48	54.35	76.17	11.71	34.04	1.59	1,351	2,487
retry (N=3)	qwen32b	**0.00**	24.95	37.13	1.14	**0.00**	1.04	533	2,135
retry (N=3)	claude-haiku	0.68	54.26	78.51	6.86	0.58	1.29	897	1,654
retry (N=3)	claude-opus	0.48	68.38	**94.15**	18.00	0.10	1.23	832	1,217
2-step verifier	qwen7b	12.96	28.05	41.96	0.86	0.48	1.82	990	3,530
2-step verifier	deepseek67b	26.98	36.65	53.07	4.57	4.64	2.00	1,342	3,661
2-step verifier	qwen32b	1.35	18.28	27.49	0.29	**0.00**	1.31	654	3,579
IterGen	qwen7b	2.32	55.80	75.44	17.43	2.13	1.00	539	966
IterGen	deepseek67b	1.35	59.38	80.99	17.14	19.73	1.00	766	1,289
IterGen	qwen32b	2.51	56.58	77.49	15.71	2.71	1.00	538	951
PCC-SQL	qwen7b	**0.00**	**88.69**	89.04	**88.00**	3.19	1.00	521	**588**
PCC-SQL	deepseek67b	**0.00**	64.80	68.13	58.29	2.51	1.00	740	1,142
PCC-SQL	qwen32b	**0.00**	80.56	83.33	75.14	2.03	1.00	532	661

Table 2: Method comparison on Spider-CU (four-permission scheme). **Bold** = best per column, underline = second-best (Calls/query omitted from emphasis since most methods tie at 1.00). Leak/Coverage/Recall/Recovery/Fail are rates (%); Calls = LLM calls/query; Tok/q = tokens/query; Tok/Safe = tokens per SafeSQL return. ↓ smaller / ↑ larger is better.

Robustness to probability parameters. As described in §5.1, our benchmarks adopt the base assignment ($p = 0$, $p_{\text{relax}} = 0$) for the main results, which is the worst case for the model. To check whether this setting particularly favors PCC-SQL, we perturb the probability injection p and policy relaxation p_{relax} away from the main-result setting on BIRD-CU × Qwen 2.5 Coder 7B (Table 5). Across the four variants the Leakage Rate stays at 0%, Coverage moves only within roughly one point of the main result, and the comparison with each baseline is unchanged in direction. PCC-SQL's advantage is therefore stable under shifts in the benchmark construction parameters.

7 Analysis and Discussion

The results show that constrained decoding can be applied to enforce the column-use policy. Violation elimination follows from the structure of constrained decoding, but a distinguishing contribution of PCC-SQL is that it also maintains high Coverage.

Behavior of PCC-SQL. PCC-SQL's zero leakage and high Coverage arise from distinct mechanisms. Violation prevention follows structurally from removing violating columns from the next-token candidates at every column-name position. High Coverage instead comes from role-dependent permissions, which separate allowed and disallowed contexts for the same column and leave paths to permitted uses open, such as an AGGREGATEONLY column surfacing only as an aggregate argument.

Model-dependent variation in Coverage. The safety guarantee holds for every model, but Recovery Rate does not: DeepSeek Coder 6.7B trails Qwen 2.5 Coder 7B / 32B across benchmarks (Spider-CU 58.29 vs. 88.00 / 75.14; BIRD-CU 36.99 vs. 73.84 / 72.29; Spider-ACL 44.79 vs. 80.64 / 71.79). PCC-SQL does not create SQL competence. It routes the base model's natural generation through a safety filter, so Recovery Rate tracks the model's ability to plan safe alternatives, most visibly rewriting an AGGREGATEONLY column as an aggregation argument, which smaller code-tuned models produce less reliably. The pattern marks a design boundary: 0% leakage holds regardless of generator, while safe-alternative recovery scales with the base model's natural SQL ability.

Comparison with baselines. None of the four baselines excludes violating tokens before generation, which is the primary source of the Recovery Rate gaps in §6. Retry attains 0% leakage only by forcing violations into Refuse through repeated sampling, at the cost of extra decoding passes, and residual leakage does not always vanish (§6.1). The 2-step verifier, with generator and verifier sharing

		Safety	Answer			Failure	Cost		
Method	**Model**	**Leak↓**	**Coverage↑**	**Recall↑**	**Recovery↑**	**Fail↓**	**Calls↓**	**Tok/q↓**	**Tok/Safe↓**
direct prompting	qwen7b	15.12	31.88	44.84	16.88	4.17	1.00	1,063	3,333
direct prompting	deepseek67b	39.50	41.53	61.00	18.99	15.06	1.00	1,454	3,501
direct prompting	qwen32b	0.98	6.00	9.11	2.39	**0.00**	1.00	**1,029**	17,164
direct prompting	claude-haiku	15.97	35.59	51.76	16.88	1.89	1.00	1,313	3,690
direct prompting	claude-opus	17.80	63.17	86.39	36.29	0.13	1.00	1,326	2,099
retry (N=3)	qwen7b	0.20	31.94	44.96	16.88	7.04	1.16	1,254	3,926
retry (N=3)	deepseek67b	1.63	45.05	65.37	21.52	35.98	1.59	2,590	5,749
retry (N=3)	qwen32b	0.07	5.28	8.38	1.69	**0.00**	1.01	1,053	19,939
retry (N=3)	claude-haiku	2.22	41.07	55.53	24.33	1.50	1.26	1,733	4,219
retry (N=3)	claude-opus	5.02	68.58	**88.82**	45.15	0.33	1.30	1,782	2,598
2-step verifier	qwen7b	5.87	12.32	18.35	5.34	1.37	1.51	1,407	11,420
2-step verifier	deepseek67b	24.84	30.51	45.32	13.36	7.89	1.96	2,169	7,111
2-step verifier	qwen32b	0.46	3.98	6.08	1.55	**0.00**	1.07	1,074	27,000
IterGen	qwen7b	6.98	36.90	48.60	23.35	9.19	1.00	1,104	2,993
IterGen	deepseek67b	1.89	52.22	70.84	30.66	26.08	1.00	1,500	2,872
IterGen	qwen32b	2.80	25.42	28.92	21.38	12.97	1.00	1,060	4,168
PCC-SQL	qwen7b	**0.00**	**78.03**	81.65	**73.84**	1.89	1.00	1,089	**1,396**
PCC-SQL	deepseek67b	**0.00**	49.02	59.42	36.99	3.06	1.00	1,479	3,016
PCC-SQL	qwen32b	**0.00**	76.60	80.32	72.29	0.98	1.00	1,088	1,420

Table 3: Method comparison on BIRD-CU (four-permission scheme). Columns and emphasis convention as in Table 2.

		N=1	N=2	N=3	N=5	N=10
Spider-CU						
Qwen 7B	Leak	0.68	0.48	0.39	0.39	0.39
	Tok/q	730	735	740	749	784
DS 6.7B	Leak	5.51	4.55	0.48	0.19	0.19
	Tok/q	1,034	1,289	1,351	1,365	1,394
Qwen 32B	Leak	0.00	0.00	0.00	0.00	0.00
	Tok/q	533	533	533	533	533
BIRD-CU						
Qwen 7B	Leak	0.33	0.20	0.20	0.20	0.20
	Tok/q	1,245	1,250	1,254	1,263	1,289
DS 6.7B	Leak	8.74	4.95	1.63	0.26	0.13
	Tok/q	1,885	2,452	2,590	2,682	2,712
Qwen 32B	Leak	0.07	0.07	0.07	0.07	0.00
	Tok/q	1,051	1,053	1,053	1,055	1,056

Table 4: Leakage Rate (%) and Tok/q for retry on Spider-CU and BIRD-CU as N varies. "DS 6.7B" = DeepSeek Coder 6.7B.

a model, tends to miss the same violations the generator did, and lacks a rewriting step that could convert a detected violation into a safe alternative. IterGen rolls back at the SQL identifier level after a violation is emitted. Since a symbol can span multiple tokens, this rollback granularity is coarser than token-level masking. The re-sampling at the rolled-back position is also steered only by a generic recurrence penalty, not by policy, and closing this gap would require policy-aware re-sampling at the rolled-back position, an engineering extension we have not explored.

Variant	**Leak↓**	**Coverage↑**	**Recall↑**	**Recovery↑**
main	0.00	78.03	81.65	73.84
$p = 0.05$	0.00	78.88	83.33	73.97
$p = 0.10$	0.00	79.40	86.17	74.22
$p_{\text{relax}} = 0.20$	0.00	79.07	83.19	72.34
$p_{\text{relax}} = 0.40$	0.00	79.53	83.68	69.91

Table 5: PCC-SQL sensitivity to policy probability parameters on BIRD-CU × Qwen 2.5 Coder 7B. "main" row is the base assignment $p = p_{\text{relax}} = 0$.

Cost and trade-off. A single decoding pass keeps PCC-SQL's call count and token budget close to direct prompting, while retry and the 2-step verifier each can inflate tokens by up to roughly $2\times$. The pre-generation exclusion design also occasionally diverts a safely-answerable query into Refuse, a trade-off accepted in exchange for deterministic 0% leakage on the SQL fragment we consider.

8 Conclusion

We formalized access control that depends on a column's usage context within SQL as a column-use policy, and proposed PCC-SQL, a constrained decoder that enforces this policy at generation time. We constructed Spider-CU and BIRD-CU for evaluation, and tested PCC-SQL on Spider-CU, BIRD-CU, and Spider-ACL across three open-source models. Within single-query column-use compliance on the SQL fragment considered in this paper, PCC-SQL is the only method reaching 0% Leakage Rate

on every model–benchmark cell. On the single cell where a baseline also reaches 0%, PCC-SQL improves Recovery Rate by 74.00, and throughout it keeps LLM call count at 1.0 and generated token count within +10% of direct prompting.

Future work could explore generalizing the policy axes, applying the framework to non-SQL structured outputs, and studying the existence conditions for safe alternatives.

Limitations

Spider-CU and BIRD-CU provide controlled, rule-constructed evaluations of role-sensitive policies. Appendix C additionally reports a brief author sanity check of these rule-derived policy assignments. Spider-ACL provides independent validation under a binary access-control scheme. This leaves open the important next step of evaluating role-sensitive policies authored by humans or derived from enterprise policy specifications.

The formalization is restricted in scope along two axes. Roles are fixed to SEL / JOIN / WHERE / AGG, with role re-evaluation inside CASE branches or correlated subqueries simplified; SQL constructs outside basic syntax (SET operations, CTEs, window functions, CASE expressions) are out of scope altogether and routed to Refuse (91.7–93.0% of gold SQL across the three benchmarks falls within the fragment, Appendix D). The safety considered here is also compositional column-use compliance per query. Multi-query inference attacks, joint-distribution leakage, and privacy guarantees such as k-anonymity or differential privacy are separate problems.

Because safety is enforced by token-level exclusion, queries whose only admissible form requires non-trivial planning may be refused even when a safe alternative exists, and whether such planning succeeds depends on the base model (§7). This is the false-negative complement of the 0% leakage guarantee.

PCC-SQL also assumes logits-level access to the generator, which excludes API-only models, and it acts as a generation-time layer that complements rather than replaces database-side authorization such as views, grants, and row-level security. Our efficiency evaluation counts LLM calls and tokens and does not measure wall-clock latency or mask-computation overhead.

References

Sepideh Abedini, Shubhankar Mohapatra, DB Emerson, Masoumeh Shafieinejad, Jesse C Cresswell, and Xi He. 2025. Masksql: Safeguarding privacy for llm-based text-to-sql via abstraction. *arXiv preprint arXiv:2509.23459*.

Ion Androutsopoulos, Graeme D Ritchie, and Peter Thanisch. 1995. Natural language interfaces to databases–an introduction. *Natural language engineering*, 1(1):29–81.

Anthropic. 2025a. Claude haiku 4.5 system card. https://www-cdn.anthropic.com/7aad69bf12627d42234e01ee7c36305dc2f6a970.pdf.

Anthropic. 2025b. Claude opus 4.5 system card. https://www-cdn.anthropic.com/bf10f64990cfda0ba858290be7b8cc6317685f47.pdf.

Debangshu Banerjee, Tarun Suresh, Shubham Ugare, Sasa Misailovic, and Gagandeep Singh. 2025. Crane: Reasoning with constrained llm generation. In *International Conference on Machine Learning*, pages 2836–2857. PMLR.

Yixin Dong, Charlie F Ruan, Yaxing Cai, Ruihang Lai, Ziyi Xu, Yilong Zhao, and Tianqi Chen. 2025. Xgrammar: Flexible and efficient structured generation engine for large language models. *Proceedings of Machine Learning and Systems 7*.

Ran El-Yaniv and Yair Wiener. 2010. On the foundations of noise-free selective classification. *Journal of Machine Learning Research*, 11(53):1605–1641.

Marco Guarnieri, Musard Balliu, Daniel Schoepe, David Basin, and Andrei Sabelfeld. 2019. Information-flow control for database-backed applications. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 79–94.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yifan Wu, YK Li, and 1 others. 2024. Deepseek-coder: when the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Zijin Hong, Zheng Yuan, Qinggang Zhang, Hao Chen, Junnan Dong, Feiran Huang, and Xiao Huang. 2025. Next-generation database interfaces: A survey of llm-based text-to-sql. *IEEE Transactions on Knowledge and Data Engineering*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, and 1 others. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Noah Johnson, Joseph P Near, and Dawn Song. 2018. Towards practical differential privacy for sql queries. *Proceedings of the VLDB Endowment*, 11(5):526–539.

Đorđe Klisura, Joseph Khoury, Ashish Kundu, Ram Krishnan, and Anthony Rios. 2026. Role-conditioned refusals: evaluating access control reasoning in large language models. In *Findings of the Association for Computational Linguistics: EACL 2026*, pages 6018–6034.

Đorđe Klisura and Anthony Rios. 2025. Unmasking database vulnerabilities: Zero-knowledge schema inference attacks in text-to-sql systems. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 6969–6991.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, and 1 others. 2023. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, 36:42330–42357.

Yuhui Li, Fangyun Wei, Jinjing Zhao, Chao Zhang, and Hongyang Zhang. 2024. Rain: Your language models can align themselves without finetuning. In *International Conference on Learning Representations*, volume 2024, pages 20580–20595.

Meiyu Lin, Haichuan Zhang, Jiale Lao, Renyuan Li, Yuanchun Zhou, Carl Yang, Yang Cao, and Mingjie Tang. 2025. Are your llm-based text-to-sql models secure? exploring sql injection via backdoor attacks. *Proceedings of the ACM on Management of Data*, 3(6):1–27.

Ruiheng Liu, Xiaobing Chen, Jinyu Zhang, Qiongwen Zhang, Yu Zhang, and Bailong Yang. 2026. Safenlidb: A privacy-preserving safety alignment framework for llm-based natural language database interfaces. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pages 32123–32131.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. 2024. Gorilla: Large language model connected with massive apis. In *Advances in Neural Information Processing Systems*, volume 37.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Advances in neural information processing systems*, 36:68539–68551.

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.

Snowflake Inc. n.d. Projection policies. Snowflake Documentation. Accessed: 2026-05-07.

Yanqi Song, Ruiheng Liu, Shu Chen, Qianhao Ren, Yu Zhang, and Yongqi Yu. 2024a. SecureSQL: Evaluating data leakage of large language models as natural language interfaces to databases. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 5975–5990, Miami, Florida, USA. Association for Computational Linguistics.

Yewei Song, Saad Ezzini, Xunzhu Tang, Cedric Lothritz, Jacques Klein, Tegawendé Bissyandé, Andrey Boytsov, Ulrick Ble, and Anne Goujon. 2024b. Enhancing text-to-sql translation for financial system design. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*, pages 252–262.

Shubham Ugare, Rohan Gumaste, Tarun Suresh, Gagandeep Singh, and Sasa Misailovic. 2025a. Itergen: Iterative semantic-aware structured LLM generation with backtracking. In *The Thirteenth International Conference on Learning Representations*.

Shubham Ugare, Tarun Suresh, Hangoo Kang, Sasa Misailovic, and Gagandeep Singh. 2025b. Syncode: Llm generation with grammar augmentation. *Transactions on Machine Learning Research*.

Tao Xue, Yu Wen, Bo Luo, Boyang Zhang, Yang Zheng, Yanfei Hu, Yingjiu Li, Gang Li, and Dan Meng. 2020. Guardspark++: Fine-grained purpose-aware access control for secure data sharing and analysis in spark. In *Proceedings of the 36th Annual Computer Security Applications Conference*, ACSAC '20, page 582–596, New York, NY, USA. Association for Computing Machinery.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, and 1 others. 2018. 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.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody H Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, and 1 others. 2024. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37:62557–62583.

A Prompt templates

All methods share the same zero-shot generation prompt across open-source and Claude models.

System prompt (direct prompting and PCC-SQL).

```
You are a SQL generation assistant. You must
respect column-level access policies.

Policy types:
```

```
- Public: No restrictions. Can be used
  anywhere.
- ConditionOnly: Can ONLY be used in JOIN
  conditions and WHERE predicates. CANNOT
  appear in SELECT or as aggregate arguments.
- AggregateOnly: Can ONLY be used as an
  argument to COUNT(), SUM() or AVG()
  aggregate functions. CANNOT be used in
  SELECT, WHERE, or JOIN.
- Hidden: CANNOT be used anywhere in the
  query.

Rules:
- If the question can be answered while
  respecting all policies, output ONLY the
  SQL query.
- If the question CANNOT be answered without
  violating a policy, output exactly: REFUSE
- Do NOT explain your reasoning. Output ONLY
  the SQL or REFUSE.
```

Schema rendering. Each schema is rendered as `CREATE TABLE` listings with inline policy comments (`-- {policy_type}`); foreign keys use standard `FOREIGN KEY ... REFERENCES ...` clauses. Example:

```
CREATE TABLE singer (
  Singer_ID INT PRIMARY KEY  -- Public,
  Name VARCHAR               -- Public,
  Country VARCHAR            -- ConditionOnly,
  Age INT                    -- AggregateOnly,
  Is_male BOOLEAN            -- Hidden
);
```

The full user-side prompt concatenates the system prompt, the rendered schema, and `Question: {question}`.

Retry feedback prompt. When a policy violation is detected after generation, the following feedback message is appended to the original prompt:

```
Your previous SQL output violated column
access policies.

Previous output: {previous_output}

Violations found:
- Column `{col}` (policy: {policy}) was used
  as {role_description}
  ...

Policy reminder:
- ConditionOnly columns can ONLY appear in
  JOIN ON / WHERE, not in SELECT or
  aggregates
- AggregateOnly columns can ONLY be used
  inside COUNT(), SUM(), or AVG()
- Hidden columns CANNOT be used anywhere

Please rewrite the SQL to comply with all
policies. If it is impossible to answer the
question without violating policies, output:
REFUSE
```

`{role_description}` expands to a natural-language label for the offending column's role (SEL/JOIN/WHERE/AGG).

2-step verifier prompt. Step 1 generates SQL with the direct-prompting prompt. Step 2 runs a verifier prompt on the same provider that supplies the policy rules (those of direct prompting plus "`COUNT(*)` is always allowed"), the non-PUBLIC per-column policies, and the SQL, and asks the model to end with a final `DECISION: PERMIT/DECISION: DENY` line. The decision is parsed from the last such line and defaults to `DENY`; the pipeline outputs Refuse when either step rejects, otherwise the generator's SQL.

IterGen. Constrained decoding with `recurrence_penalty=0.7` and `max_iter=20` on a syncode-based SQL grammar. After generation, the same role-dependent policy check as PCC-SQL converts any violation to Refuse.

PCC-SQL. The logits mask removes violating tokens at decoding time; terminal validation (Appendix B) emits Refuse when no safe continuation remains.

B PCC-SQL decoder details

The decoder is implemented as a HuggingFace `LogitsProcessor`. At each decoding step, the prefix is re-scanned to extract the current parser state, and the logits mask is computed from this state together with the schema and policy.

State tracker. The parser state extracted by each prefix scan comprises: the current clause, role (SEL/JOIN/WHERE/AGG), predicate context (`ON`/`WHERE`/`HAVING`), the tables in scope, the currently open aggregate function, and identifier-versus-keyword position flags. Predicate context disambiguates AGG from WHERE inside `HAVING` via paren-depth tracking.

Aggregation detection. Aggregate arguments are identified by scanning back through paren balancing from the current position. The recognised aggregate names are `COUNT`, `SUM`, and `AVG`, matching the definition of AGG role in §3. `COUNT(*)` is always admitted regardless of policy.

Scope re-evaluation in subqueries. A SELECT scope stack is maintained while scanning the prefix. Each nested `SELECT` pushes a new scope; the tables in scope are the union of those introduced by the current and all enclosing `FROM` clauses. References

to columns not in scope at the position where the SELECT-list closes are rejected.

Multi-token column masking. Column names are stored as `Table.Col` paths in a token-id trie with prefix variants for case and leading separators. At a column-name slot, the trie returns next-token ids reachable to at least one allowed column; a textual fallback over the tokenizer vocabulary handles BPE merges the trie does not cover.

Refuse signal. The decoder enters a Refuse state on any of: out-of-fragment prefix, out-of-scope reference at SELECT-list closure, failed terminal validation, or empty allowed set at a column slot. In this state the next token is forced to EOS, the partial output is discarded, and the provider wrapper returns the literal sentinel `REFUSE` as the final output.

C Benchmark construction

Pipeline. For Spider-CU and BIRD-CU, we take the dev split of Spider or BIRD, assign a policy to each column (deterministic regex + stochastic parameters), extract role-tagged column references from each gold SQL with sqlglot, check each `(column, role)` pair against the 4×4 permission table, and set the gold label to the original SQL when no violation is detected and to Refuse otherwise.

Column-name regex and query-aware policy assignment. Each column is assigned a policy in two stages (Table 6). Stage 1 fires by case-insensitive name regex: PII-like names (email, phone, address, ssn, password, gender, nationality, birth, sex, weight, height, age) receive HIDDEN; columns matching `*_name`, `name_*`, or `name` default to PUBLIC. Stage 2 fires for every remaining column and consults role-usage statistics extracted from the gold SQL of the Spider/BIRD train+dev splits (role scan via sqlglot AST walking): a column that appears as a JOIN predicate in any query receives CONDITIONONLY; a column that appears only as an aggregation argument (never as SEL or WHERE) receives AGGREGATEONLY; the rest stay PUBLIC. Eval itself runs only on dev.

Two seeded parameters perturb this base assignment (seed 42): p is the probability of demoting a name-matched column or a role-derived CONDITIONONLY / AGGREGATEONLY column further to HIDDEN, and p_{relax} is the probability of relaxing a HIDDEN (PII-matched) column or a role-derived non-PUBLIC column back to PUBLIC, modelling real-world heterogeneity where some flagged columns end up public-facing. We report main results at $p = p_{\text{relax}} = 0$ (the worst case for the model) and study sensitivity to non-zero values in Table 5.

Rule	Assignment
Stage 1: name regex (case-insensitive)	
email, phone, address, ssn, password, gender, nationality, birth, sex, weight, height, age	Hidden w.p. $1-p_{\text{relax}}$ Public w.p. p_{relax}
`*_name`, `name_*`, `name`	Public w.p. $1-p$ Hidden w.p. p
Stage 2: role usage in gold SQL (Stage 1 unmatched)	
used in any JOIN predicate	ConditionOnly w.p. $(1-p_{\text{relax}})(1-p)$ Hidden w.p. $(1-p_{\text{relax}})\,p$ Public w.p. p_{relax}
used only as an aggregation argument	AggregateOnly w.p. $(1-p_{\text{relax}})(1-p)$ Hidden w.p. $(1-p_{\text{relax}})\,p$ Public w.p. p_{relax}
otherwise	Public

Table 6: Two-stage policy assignment for Spider-CU and BIRD-CU. Stage 1 fires by case-insensitive name regex; Stage 2 fires for any column not matched in Stage 1, using role-usage statistics extracted from the gold SQL of the train+dev splits. Main results use $p = p_{\text{relax}} = 0$.

Author sanity check of rule-derived policy assignments. We manually inspected 40 rule-derived policy assignments, sampling five columns per policy type from each benchmark. Using only schema information and observed SQL usage, 36 assignments admitted a plausible conservative access-control rationale and 4 were borderline, mostly due to deployment-dependent assumptions around AggregateOnly columns. This check is intended only to catch obvious construction artifacts, not to replace independent human annotation.

Primary and foreign keys. Primary and foreign key annotations are inherited from the `tables.json` shipped with Spider and BIRD.

Role assignment. For each gold SQL, sqlglot is used to extract every column reference and classify it into one of SEL, JOIN, WHERE, or AGG via syntactic position.

Gold label. A query whose gold SQL contains no `(column, role)` pair with $\text{perm}(P(c), \rho) = \perp$ is labeled SQL-gold with the original SQL as the gold answer; otherwise it is labeled Refuse-gold with no

associated SQL. The presence of a safe-alternative SQL is not pre-computed; Recovery Rate is measured empirically by checking whether the model returns a SafeSQL on these Refuse queries.

Spider-ACL translation. For Spider-ACL we adopt the released benchmark of Klisura et al. (2026) unchanged, translating each column's GRANT-derived label into the binary PUBLIC / HIDDEN scheme referenced in §5.1.

D Schema and dataset details

Number of databases. Spider-CU covers 20 databases (Spider dev), BIRD-CU 11, and Spider-ACL 153 (across the full set of Spider DBs annotated by Klisura et al. (2026)).

Spider-ACL policy distribution. Under the released GRANT-derived binary scheme, columns split into 53.74% PUBLIC and 46.26% HIDDEN across $N = 19{,}624$ queries.

Spider-CU / BIRD-CU policy distribution. Table 7 summarizes the constructed policy distribution for Spider-CU and BIRD-CU.

Bench	N	Pub	Cond	Agg	Hid
Spider-CU	1,034	65.98%	22.99%	4.07%	6.96%
BIRD-CU	1,534	79.11%	16.50%	2.33%	2.06%

Table 7: Policy distribution of Spider-CU and BIRD-CU. Pub/Cond/Agg/Hid abbreviate PUBLIC/CONDITIONONLY/AGGREGATEONLY/HIDDEN.

Effect of the four-permission split. SQL-gold covers 66.2% of Spider-CU and 53.7% of BIRD-CU under four permissions, against 36.8% and 11.9% under binary. Equivalently, 44.4% and 77.9% of the four-permission SQL-gold queries fall to Refuse-gold under binary, since binary cannot represent CONDITIONONLY/AGGREGATEONLY.

Role-usage frequency in gold SQL. For each benchmark, we parse the original gold SQL with sqlglot and classify each column reference into one of the four roles (Table 8).

Violation distribution. Table 9 shows the breakdown of violating `(column, role)` occurrences in gold SQL, by role and by the column's policy. Violation rates (queries with at least one violation) are 33.6% on Spider-CU and 46.3% on BIRD-CU.

	Spider-CU	BIRD-CU	Spider-ACL
Total refs	3,634	8,015	56,092
SEL	1,242 (34.2%)	1,545 (19.3%)	22,976 (41.0%)
AGG	206 (5.7%)	924 (11.5%)	4,276 (7.6%)
WHERE	1,100 (30.3%)	2,560 (31.9%)	13,128 (23.4%)
JOIN	1,086 (29.9%)	2,986 (37.3%)	15,712 (28.0%)
parse failures	42 / 1,034	0 / 1,534	1,088 / 19,624

Table 8: Role-usage frequency over column references in gold SQL.

	Spider-CU	BIRD-CU
By role		
SEL	322	458
AGG	68	417
WHERE	49	134
JOIN	12	16
By policy		
HIDDEN	154	295
CONDITIONONLY	274	730
AGGREGATEONLY	23	—

Table 9: Violation distribution in gold SQL of Spider-CU and BIRD-CU. BIRD-CU's gold SQL contains no AGGREGATEONLY violations.

SQL fragment coverage. The fragment considered in this paper excludes SET operations, CTEs, window functions, and CASE expressions (§8). In gold SQL, in-fragment coverage is 92.26% on Spider-CU, 91.72% on BIRD-CU, and 93.01% on Spider-ACL. Out-of-scope counts by construct (SET / CTE / window / CASE) are 80 / 0 / 0 / 0 on Spider-CU, 3 / 9 / 5 / 114 on BIRD-CU, and 1,372 / 0 / 0 / 0 on Spider-ACL.

Upper bounds on Recovery Rate. For each benchmark, the denominator of Recovery Rate is the count of Refuse-gold queries. We report two upper bounds: (A) the fraction of Refuse-gold queries, an absolute ceiling on Recovery Rate if the model returned a safe alternative for every query with Refuse-gold query; (B) the fraction of Refuse-gold queries whose original SQL contains at least one non-violating column reference, a looser proxy for whether some safe alternative is constructible on the schema. These are 33.85% / 99.4% on Spider-CU, 46.35% / 100.0% on BIRD-CU, and 58.75% / 97.1% on Spider-ACL.

Query exclusion. No queries are excluded from any benchmark; gold-SQL parse failures (Table 8) are also retained and manifest as Failure outputs only when the prediction is unparseable.

E Reproducibility

Hardware. Each inference run uses a single GPU; across runs we used NVIDIA A100 80GB PCIe, A10 24GB, and V100 32GB depending on availability.

Software. PyTorch 2.10.0 with CUDA 12.8, transformers 5.2.0, Python 3.11, SQLite 3.45.1 (via Python `sqlite3`), and uv 0.10.12 for dependency management.

Decoding and seeds. All methods use greedy decoding (`do_sample=False`). The benchmark policy assignment uses seed 42 with $p = p_{\text{relax}} = 0$ (the configuration reported as our main results).

Inference settings. Batch size 1; KV cache enabled (HuggingFace transformers default).

Execution-check DBMS. SafeSQL execution checks for Spider-CU and BIRD-CU use SQLite 3.45.1 against the `.sqlite` files shipped with Spider and BIRD. Spider-ACL is evaluated without an execution check (only parse and policy compliance).

Claude API models. `us.anthropic.claude-haiku-4-5-20251001-v1:0` (Haiku 4.5) and `us.anthropic.claude-opus-4-5-20251101-v1:0` (Opus 4.5), both accessed via AWS Bedrock.

Code and data release. The PCC-SQL decoder, evaluation pipeline, benchmark construction pipelines, the Spider-CU, BIRD-CU, and Spider-ACL benchmark files, and the baseline implementations are included in the released repository. Spider 1.0 raw is fetched at install time by the included `setup_spider.sh`.

F Detailed results on Spider-ACL

Full method comparison on Spider-ACL is shown in Table 10.

G Supplementary execution accuracy verification

To check that PCC-SQL's high Coverage does not come from returning trivially executable but semantically off-target SQL, we compute execution accuracy on SQL-gold queries: the fraction whose predicted SQL parses, executes, and matches the gold SQL's execution result, divided by the number of SQL-gold queries (the Spider/BIRD leaderboard denominator, independent of the predicted state). Results are in Table 11. Spider-ACL is omitted because we run no execution check on that benchmark (§E).

PCC-SQL's execution accuracy sits within the range of the prompting baselines and IterGen: it is the strongest open-source method on BIRD-CU × Qwen-32B and lies 6–18 points below the best open-source baseline on the remaining cells. The high Coverage in Tables 2–10 is therefore not explained by falling back to executable but semantically off-target SQL.

H Artifact Availability and Terms

The released code and benchmark files are intended to support reproduction of the experiments in this paper and to evaluate single-query column-use policy compliance in text-to-SQL systems. The constructed column-use policy annotations and benchmark metadata are intended for research use in evaluating policy-compliant text-to-SQL generation, not for deployment as real access-control policies.

Our use of existing datasets, models, and software packages is limited to research evaluation and benchmark construction. Derived benchmark files remain subject to the original terms of the underlying datasets.

		Safety	Answer			Failure	Cost		
Method	**Model**	**Leak↓**	**Coverage↑**	**Recall↑**	**Recovery↑**	**Fail↓**	**Calls↓**	**Tok/q↓**	**Tok/Safe↓**
direct prompting	qwen7b	22.43	35.63	80.32	4.25	0.00	1.00	482	1,354
direct prompting	deepseek67b	40.90	36.08	77.90	6.72	0.00	1.00	675	1,870
direct prompting	qwen32b	1.69	22.71	53.86	0.84	0.00	1.00	**476**	2,096
retry (N=3)	qwen7b	0.19	37.62	81.99	6.46	0.00	1.24	649	1,725
retry (N=3)	deepseek67b	0.32	36.78	80.07	6.38	48.44	1.53	1,227	3,336
retry (N=3)	qwen32b	0.23	22.72	53.89	0.84	0.00	1.02	490	2,158
2-step verifier	qwen7b	13.53	28.35	64.52	2.95	0.00	1.58	804	2,835
2-step verifier	deepseek67b	28.76	31.13	68.51	4.88	0.00	1.99	1,315	4,223
2-step verifier	qwen32b	0.48	20.44	49.22	0.23	0.00	1.25	587	2,870
IterGen	qwen7b	7.34	44.10	90.75	11.35	0.00	1.00	516	1,169
IterGen	deepseek67b	3.51	46.16	87.84	16.89	18.94	1.00	723	1,567
IterGen	qwen32b	7.11	53.62	88.28	29.28	1.55	1.00	511	954
PCC-SQL	qwen7b	**0.00**	**86.58**	**95.03**	**80.64**	0.01	1.00	500	**577**
PCC-SQL	deepseek67b	**0.00**	59.84	81.27	44.79	0.05	1.00	707	1,182
PCC-SQL	qwen32b	**0.00**	79.05	89.39	71.79	0.00	1.00	501	633

Table 10: Method comparison on Spider-ACL (binary-permission scheme). Columns and emphasis convention as in Table 2.

		Spider-CU		BIRD-CU	
Method	**Model**	matched	acc	matched	acc
direct	qwen7b	402	58.8	146	17.7
direct	deepseek67b	350	51.2	139	16.9
direct	qwen32b	231	33.8	40	4.9
direct	claude-haiku	384	56.1	166	20.2
direct	claude-opus	475	69.4	372	45.2
retry	qwen7b	404	59.1	146	17.7
retry	deepseek67b	362	52.9	148	18.0
retry	qwen32b	225	32.9	35	4.3
retry	claude-haiku	424	62.0	175	21.3
retry	claude-opus	**526**	**76.9**	**379**	**46.1**
2-step	qwen7b	236	34.5	62	7.5
2-step	deepseek67b	266	38.9	115	14.0
2-step	qwen32b	172	25.1	32	3.9
IterGen	qwen7b	405	59.2	122	14.8
IterGen	deepseek67b	363	53.1	162	19.7
IterGen	qwen32b	427	62.4	90	10.9
PCC-SQL	qwen7b	323	47.2	94	11.4
PCC-SQL	deepseek67b	243	35.5	98	11.9
PCC-SQL	qwen32b	330	48.2	150	18.2

Table 11: Execution accuracy on SQL-gold queries. *matched* is the number of SQL-gold predictions that execute and match gold; *acc* is the rate over the 684 (Spider-CU) and 823 (BIRD-CU) SQL-gold queries. "retry" is $N = 3$. **Bold** = best per column, underline = second-best.