APEX-SQL: Talking to the data via Agentic Exploration for Text-to-SQL

Bowen Cao*†
bwcao@link.cuhk.edu.hk
The Chinese University of Hong Kong
Hong Kong, China

Weibin Liao*†
liaoweibin@stu.pku.edu.cn
Peking University
Beijing, China

Yushi Sun‡
ysunbp@connect.ust.hk
LIGHTSPEED
Shenzhen, China

Dong Fang‡
df572@outlook.com
LIGHTSPEED
Shenzhen, China

Haitao Li
729156675@qq.com
LIGHTSPEED
Shenzhen, China

Wai Lam
wlam@se.cuhk.edu.hk
The Chinese University of Hong Kong
Hong Kong, China

Abstract

Text-to-SQL systems powered by Large Language Models have excelled on academic benchmarks but struggle in complex enterprise environments. The primary limitation lies in their reliance on static schema representations, which fails to resolve semantic ambiguity and scale effectively to large, complex databases. To address this, we propose APEX-SQL, an Agentic Text-to-SQL Framework that shifts the paradigm from passive translation to agentic exploration. Our framework employs a hypothesis-verification loop to ground model reasoning in real data. In the schema linking phase, we use logical planning to verbalize hypotheses, dual-pathway pruning to reduce the search space, and parallel data profiling to validate column roles against real data, followed by global synthesis to ensure topological connectivity. For SQL generation, we introduce a deterministic mechanism to retrieve exploration directives, allowing the agent to effectively explore data distributions, refine hypotheses, and generate semantically accurate SQLs. Experiments on BIRD (70.65% execution accuracy) and Spider 2.0-Snow (51.01% execution accuracy) demonstrate that APEX-SQL outperforms competitive baselines with reduced token consumption. Further analysis reveals that agentic exploration acts as a performance multiplier, unlocking the latent reasoning potential of foundation models in enterprise settings. Ablation studies confirm the critical contributions of each component in ensuring robust and accurate data analysis.

Keywords

Agentic Text-to-SQL, Schema Linking, SQL Generation

*This work was done during an internship at Tencent.
†Both authors contributed equally to this research.
‡Correspondence to: Yushi Sun and Dong Fang.

Conference acronym 'XX, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Bowen Cao, Weibin Liao, Yushi Sun, Dong Fang, Haitao Li, and Wai Lam. 2018. APEX-SQL: Talking to the data via Agentic Exploration for Text-to-SQL. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 21 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Text-to-SQL [18, 22] aims to make data analysis accessible by translating natural language questions into executable SQL queries. Recent advancements in Large Language Models (LLMs) have significantly improved execution accuracy on academic benchmarks like Spider 1.0 [30] and BIRD [15]. However, this performance drops substantially in real-world enterprise environments [10].

A fundamental limitation of current approaches lies in their reliance on passive perception. Standard methods depend on static schema definitions (Figure 1a), but enterprise databases often feature opaque column names, hiding true semantics within data values rather than metadata. Without access to the data, models are forced to guess. To address this, existing works attempt to inject pre-processed data profiles (Figure 1b). However, this approach has limitations. On one hand, static summaries remain detached from specific question contexts, failing to provide the exact evidence needed. On the other hand, covering the vast scale of databases introduces noise and incurs heavy token overhead. Consequently, these passive paradigms disconnect reasoning from reality, leading to queries that are syntactically correct but semantically flawed.

To resolve this, we propose a paradigm shift from passive perception to agentic exploration (Figure 1c). Central to our approach is a Hypothesis-Verification loop where the agent formulates logical assumptions based on the user question and validates them by executing exploratory SQLs against actual data. This process transforms Text-to-SQL from a static translation task into an interactive reasoning task grounded in database reality.

We introduce APEX-SQL, a unified framework applying agentic exploration to both Schema Linking and SQL Generation. First, given massive industrial databases, Schema Linking is crucial to isolate the relevant schema elements before processing. We employ Logical Planning to verbalize question requirements into schema-agnostic hypotheses, and introduce Dual-Pathway Pruning to compress the search space, making the subsequent validation feasible

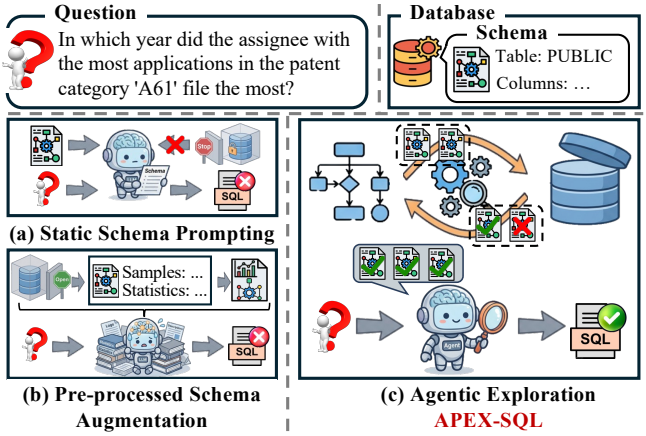

Figure 1: A comparison of Text-to-SQL paradigms. (a) Static Schema Prompting relies solely on the schema, causing hallucinations when metadata is ambiguous. (b) Pre-processed Schema Augmentation attempts to enrich context with preliminary data profiling, yet introduces irrelevant noise and token overhead. (c) APEX-SQL introduces Agentic Exploration, employing a hypothesis-verification loop to actively interrogate the database and ground logical reasoning.

within large databases. The agent then conducts Parallel Data profiling to validate these hypotheses against real data, followed by Global Synthesis to ensure topological connectivity across the verified elements. For SQL Generation, we bridge the gap between abstract intent and data-level constraints through Deterministic Guidance Retrieval, which maps logical steps into exploration directives. Equipped with this guidance, the agent performs agentic exploration by navigating a flexible action space to profile data distributions, consolidate findings, and synthesize candidate queries. The process concludes with a final confirmation step to verify semantic fidelity against the accumulated evidence, ensuring the generated SQL is syntactically and semantically correct.

Extensive experiments on BIRD and Spider 2.0 validate this paradigm shift. On **BIRD-Dev**, APEX-SQL achieves an execution accuracy of 70.7%, surpassing competitive baselines such as OpenSearch-SQL (69.3%) [28] and RSL-SQL (67.2%) [3]. On the enterprise-grade **Spider 2.0-Snow**, APEX-SQL attains 51.0% execution accuracy, outperforming existing agentic systems like DSR-SQL (35.3%) [8]. Further analysis reveals that agentic exploration acts as a performance multiplier, yielding up to an 18.33% absolute gain for high-capacity models like DeepSeek-V3.2 [17]. Test-time compute scaling evaluations show that our framework effectively unlocks the latent reasoning potential of foundation models. Moreover, ablation studies demonstrate that key components such as logical planning and exploration are crucial for improving schema linking performance; deterministic guidance in SQL generation enhances exploration effectiveness and efficiency, allowing the agent to resolve implementation uncertainties in fewer interaction rounds.

Our contributions can be summarized as follows:

- **Agentic Exploration for Schema Linking.** We propose a schema linking approach utilizing logical planning to verbalize hypotheses, dual-pathway pruning to reduce the massive search space, and parallel data profiling followed by global synthesis to ground column selection in empirical evidence.
- **Agentic Exploration for SQL Generation.** We introduce an agentic SQL generation process where deterministic retrieval provides exploration directives for the agent to verify hypotheses and consolidate data findings, autonomously resolving implementation ambiguities before query synthesis.
- **Empirical Validation & Insights.** We demonstrate superior performance on BIRD and Spider 2.0 benchmarks, outperforming competitive baselines. Our analysis confirms that agentic exploration serves as a performance multiplier and is a highly effective mechanism for robust data analysis in enterprise environments.

2 Related Work

Static Schema Prompting. To empower LLMs for Text-to-SQL tasks, standard approaches typically linearize database schemas and supply them alongside queries as prompts [5, 16, 20]. While methods such as DIN-SQL [20] and DAIL-SQL [6] optimize prompt structures to enhance reasoning capabilities, a fundamental challenge remains: LLMs are inherently "blindness" to underlying data distribution. Real-world databases often contain ambiguous column names, opaque abbreviations, or domain-specific codes that are semantically indecipherable without access to actual data [15]. Consequently, standard prompting confines models to surface-level schema representations, often inducing hallucinations when correct SQL logic depends on concrete data content rather than metadata alone. To mitigate the uncertainty arising from this "blindness", several works [9, 19, 24] generate multiple candidate SQL queries and employ consensus or ranking-based strategies to select the most plausible output. *However, these passive perception approaches still compel models to infer implementation details from ambiguous schemas. This severance of user intent from data reality frequently results in syntactically valid yet semantically flawed queries, particularly when the correct logic hinges on concealed data distributions.*

Pre-processed Schema Augmentation. To bridge the gap between abstract schema definitions and concrete data values, recent research has introduced pre-processing pipelines to enrich schema representations. Approaches such as AskData [23] leverage data profiling and query log analysis to statically augment schemas with value distributions and historical join paths prior to inference. Similarly, TA-SQL [21] employs LLMs to generate detailed natural-language descriptions for each database field. While these methods provide richer context, they rely on "pre-computed" knowledge that is static and often misaligned with the specific reasoning trajectory of a novel query. *Despite offering additional context, these static augmentation strategies incur substantial token overhead and introduce irrelevant noise. By decoupling context retrieval from the evolving reasoning demands of complex queries, they fail to provide the targeted, dynamic verification necessary for accurate problem-solving.*

Agentic Exploration. Contemporaneous literature have begun to adopt agentic workflows for database interaction. For instance, DSR-SQL [8] utilizes a dual-state mechanism to manage context updates, while AutoLink [26] focuses on iterative schema exploration to resolve linking issues in massive databases. Furthermore, ReFoRCE [4] introduces a column exploration mechanism to verify

data values before generation. Although these methods mark a paradigm shift towards active problem-solving, they often treat schema linking and value verification as isolated sub-tasks or rely on unguided exploration, leading to computational inefficiency. *Crucially, existing agentic frameworks lack a unified mechanism to discipline the exploration process. Consequently, they often suffer from inefficient, random probing trajectories that fail to systematically bridge the gap between abstract logical planning and concrete data-level constraints through a coherent hypothesis-verification loop.*

3 Methodology

This section details how APEX-SQL applies agentic exploration to schema linking and SQL generation through a hypothesis-verification loop. Figure 2 illustrates the framework of APEX-SQL.

3.1 Problem Formulation

Let q denote a natural language question. The target database is structured as a set of tables $\mathcal{D} = \{T_1, \ldots, T_m\}$[1], where each table $T_j = \{c_1, \ldots, c_k\}$ consists of columns along with their respective metadata (e.g., types and descriptions). APEX-SQL addresses two sequential objectives. The first is *schema linking*, which aims to identify a sufficient column set $\mathcal{D}^* \subset \mathcal{D}$ that maximizes the recall of columns relevant to the question while minimizing the number of columns. The second is *SQL generation*, which focuses on synthesizing a correct query requested by q against D^*.

Hypothesis-Verification (H-V). Ambiguous naming in enterprise databases often obscures true semantics, necessitating a direct and data-driven interpretation. To enable effective agentic exploration over large-scale schemas, we introduce a unified *hypothesis–verification* loop, in which the system actively interacts with the database to iteratively refine its understanding. Importantly, this agentic exploration is explicitly instantiated at two critical stages of the Text-to-SQL pipeline: *schema linking* and *SQL generation*. We detail the design of agentic exploration for schema linking and SQL generation separately in the following subsections.

3.2 Agentic Exploration for Schema Linking

To navigate large-scale enterprise databases with ambiguous naming conventions and complex structures, APEX-SQL instantiates *agentic exploration within the schema linking stage* through the H-V loop. Specifically, APEX-SQL generates schema-agnostic hypotheses in the form of logical plans, which are subsequently grounded and validated against the database. To further streamline this verification process, APEX-SQL introduces a Dual-Pathway Pruning mechanism prior to verification. The following sections provide a detailed breakdown of each component (Figure 2 (Top)).

3.2.1 Hypothesis Generation via Logical Planning. Directly mapping a query onto a massive schema forces the LLM to conflate retrieval and reasoning, often resulting in suboptimal outcomes. This coupling obscures implicit intermediate steps and exacerbates *schema bias*, wherein LLMs hallucinate spurious connections based on superficial string-level similarities. To address this issue, APEX-SQL introduces a dedicated **Logical Planning** stage. Specifically, APEX-SQL first generates a schema-agnostic plan that hypothesizes the required computational steps and latent constraints, deliberately isolating high-level reasoning from the noise introduced by concrete table and column names.

To ensure the derived plan covers all potential logical branches required to answer the query, APEX-SQL employs a consensus-based synthesis strategy where multiple reasoning paths are aggregated into a unified master plan. Formally, APEX-SQL generates N independent candidates $\{P^{(i)}\}_{i=1}^{N}$ where each candidate is sampled from $p_\theta(\cdot \mid q, F^{sl}_{plan})$. These candidates are then integrated into a master plan P^* through a secondary inference process defined by

$$P^* = p_\theta\left(\cdot \mid q, \{P^{(1)}, \ldots, P^{(N)}\}, F^{sl}_{agg}\right), \tag{1}$$

with p_θ the LLM and F^{sl}_{plan}, F^{sl}_{agg} instruction templates.

3.2.2 Search Space Reduction via Dual-Pathway Pruning. Exhaustive data exploration is computationally prohibitive, which motivates APEX-SQL to introduce a preliminary filtration step. However, standard relevance filtering is prone to false negatives, often discarding foreign keys that lack both lexical and semantic alignment with the query terms. To address this issue, APEX-SQL employs a **Dual-Pathway Pruning** strategy. Specifically, APEX-SQL simultaneously executes a negative pass to remove obvious noise and a positive pass to identify relevant items, and subsequently fuses the results from both passes. Let $\{B_1, \ldots, B_K\}$ represent K manageable batches partitioned from the schema $\mathcal{D}$, where each batch consists of a subset of columns. For each batch B_j, APEX-SQL identifies a deletion set of columns $C_{del,j} \subseteq B_j$ and a preservation set of columns $C_{keep,j} \subseteq B_j$. The pruned schema $\mathcal{D}_{pruned}$ is then formulated as:

$$\begin{aligned} C_{del,j} &= p_\theta(\cdot \mid q, P^*, B_j, F^{sl}_{del}) \\ C_{keep,j} &= p_\theta(\cdot \mid q, P^*, B_j, F^{sl}_{sel}) \\ \mathcal{D}_{pruned} &= \bigcup_{j=1}^{K} \left((B_j \setminus C_{del,j}) \cup C_{keep,j}\right) \end{aligned} \tag{2}$$

where the union logic ensures that any ambiguous element is preserved unless it is both confidently rejected as noise and fails to be explicitly recognized as task-critical.

3.2.3 Hypothesis Verification via Agentic Exploration. To verify hypotheses in the logical plan over $\mathcal{D}_{pruned}$, APEX-SQL employs **Agentic Exploration**, starting with **Semantic Linking**. In this step, APEX-SQL hypothesizes the specific role of each table and column, such as serving as the primary filter. This role mapping establishes the direction for the subsequent **Parallel Data Profiling**, in which APEX-SQL assigns independent agents to each table, restricted to the subset of columns in $\mathcal{D}_{pruned}$. Unlike static profiling, these agents dynamically generate SQL queries grounded in the hypothesized roles to verify data patterns. Following this local exploration, the **Global Synthesis** phase enables APEX-SQL to integrate the entire $\mathcal{D}_{pruned}$ along with their respective empirical observations, and to make the final decision on the sufficient subgraph $\mathcal{D}^*$. This process can be formulated as follows:

$$\begin{aligned} \mathcal{R} &= p_\theta(\cdot \mid q, \mathcal{D}_{pruned}, P^*, F^{sl}_{semantics}) \\ \mathcal{E}_i &= \text{Exec}(p_\theta(\text{sql} \mid q, T_i, \mathcal{R}_i, F^{sl}_{exp}), \mathcal{D}_{pruned}) \\ \mathcal{D}^* &= p_\theta(\cdot \mid q, \mathcal{D}_{pruned}, \{\mathcal{E}_i\}_{i=1}^{m}, F^{sl}_{final}) \end{aligned} \tag{3}$$

where $\mathcal{R}$ denotes the functional role analysis, and $\mathcal{E}_i$ represents the empirical observations for each candidate table T_i.

[1]To simplify notation, $\mathcal{D}$ includes both schema and data; real data is accessed only during agentic exploration.

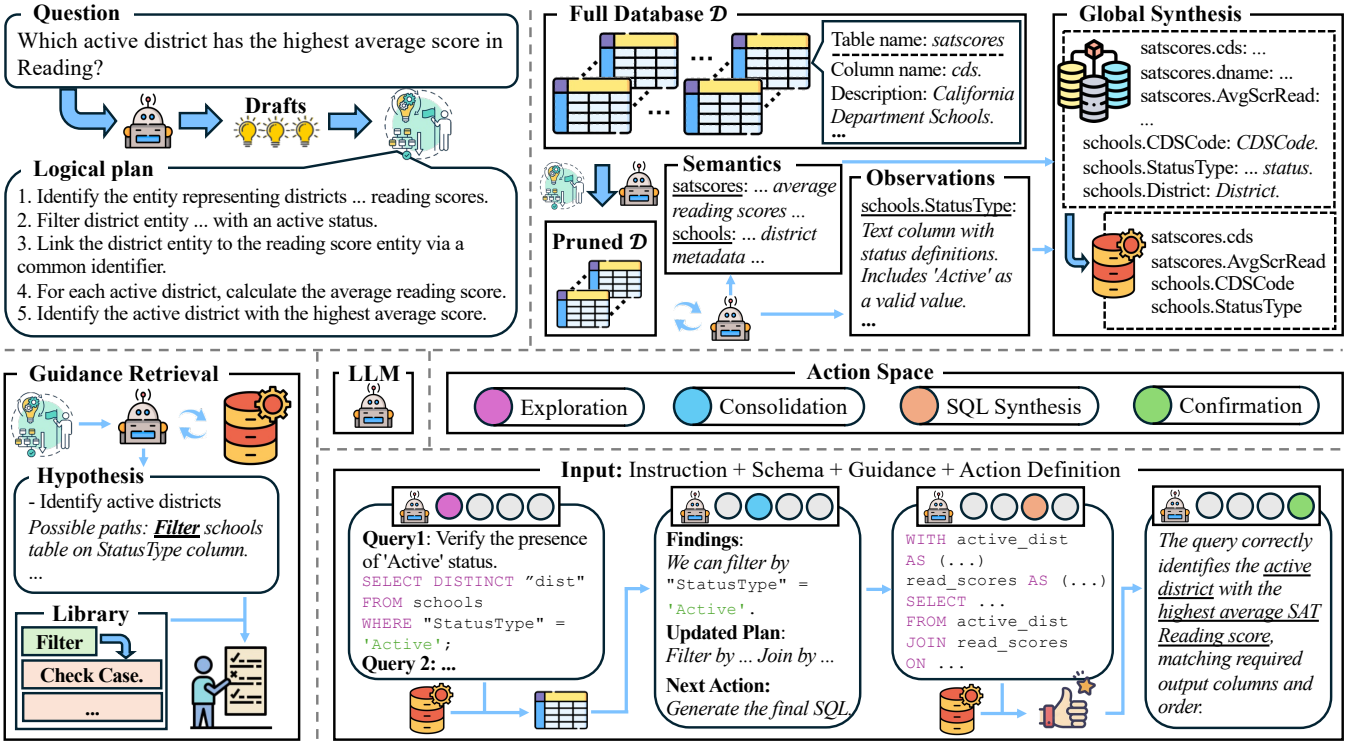

Figure 2: The proposed APEX-SQL framework. Schema Linking (Top): To navigate massive databases, APEX-SQL first verbalizes a logical plan, serving as a grounded reference for subsequent pruning and verification. It actively verifies the semantic roles of candidate tables through parallel profiling and summarizes key observations for each column. Finally, APEX-SQL refines the schema subgraph by ensuring topological connectivity and recovering missing dependencies via global synthesis. SQL Generation (Bottom): APEX-SQL employs a deterministic retrieval mechanism to map logical operations to specific exploration guidance. Directed by these constraints, it navigates a flexible action space to autonomously profile data distributions, consolidate exploration findings, synthesize candidate queries, or perform a final confirmation check to validate the executable SQL.

3.3 Agentic Exploration for SQL Generation

To translate abstract user intent into executable SQL under rigid schema constraints, APEX-SQL instantiates *agentic exploration within the SQL generation stage* through the H-V loop. Specifically, APEX-SQL interleaves reasoning and exploration to actively synthesize hypotheses and validate them through evidence-driven query execution (Figure 2 (Bottom)). To ensure meaningful exploration, APEX-SQL first retrieves question-relevant guidance that constrains the search space and prevents superficial or degenerate probing.

3.3.1 Guidance via Deterministic Retrieval. Exploration is only effective when the agent knows *what* to verify. Without explicit guidance, models are prone to overlooking critical constraints such as integer division truncation or case-sensitivity. To address this limitation, APEX-SQL introduces a **Deterministic Retrieval** mechanism that leverages the logical plan (Section 3.2.1) to retrieve query-dependent exploration guidance. Specifically, APEX-SQL infers potential SQL realization paths for each logical step, such as implementing "ranking" using the `RANK()` function over the `score` column. Based on these realizations, APEX-SQL applies a rule engine to extract and match operational keywords against a predefined library, such that detecting a `RANK` keyword activates specific directives, including inspecting `NULL` values in the corresponding column. This process is formulated as:

$$\begin{aligned} \mathcal{W} &= \text{ExtractKeywords}\Big(p_\theta(\cdot \mid q, P^*, \mathcal{D}^*, F_{kw}^{sql})\Big) \\ \mathcal{G} &= \{m \in \mathcal{M} \mid \text{Match}(\mathcal{W}, m) = \text{True}\} \end{aligned} \tag{4}$$

where $\mathcal{W}$ denotes the set of operational keywords, F_{kw}^{sql} is the instruction template, and $\mathcal{M}$ represents a library of SQL best practices. Since SQL operations constitute a finite and closed set, APEX-SQL adopts deterministic keyword mapping to achieve higher reliability than dense retrieval, which often introduces irrelevant noise. As a result, APEX-SQL transforms the logical plan from a passive blueprint into an executable verification guide, ensuring that exploration remains focused on relevant technical risks.

3.3.2 Grounding via Agentic Exploration. Guided by $\mathcal{G}$, APEX-SQL initiates an agentic process to resolve implementation ambiguities within $\mathcal{D}^*$ by transitioning over the action space $\mathcal{A}$ through a sequence of states H_t, where each new state is determined based on the prior history $H_{<t}$ and the accumulated observations $O_{<t}$:

$$H_t = p_\theta(\cdot \mid \mathcal{A}, \mathcal{G}, H_{<t}, O_{<t}). \tag{5}$$

Profiling. To resolve uncertainties regarding value distributions and data formats, APEX-SQL autonomously generates exploratory

Table 1: Schema linking performance evaluating Strict Recall Rate (SRR), Non-Strict Recall (NSR), Non-Strict Precision (NSP) and Non-Strict F1-Score (NSF). Evaluation settings include BIRD Subset (DeepSeek-V3.2), BIRD Full Set (Qwen3-32B), and Spider 2.0-Snow Subset (GPT-4.1). All baseline results in this table are locally reproduced.

Method	BIRD (N = 147)				BIRD (N = 1534)				Spi.-Snow (N = 120)			
	SRR	NSR	NSP	NSF	**SRR**	NSR	NSP	NSF	**SRR**	NSR	NSP	NSF
Training-based Methods												
CodeS	59.86	86.53	31.41	44.84	67.08	88.06	30.64	43.90	8.30	34.15	20.16	25.35
RESDSQL	46.24	74.59	27.70	39.23	49.61	76.24	27.48	38.82	23.30	55.64	27.42	36.74
Training-free Methods												
	Backbone: DeepSeek-V3.2				*Backbone: Qwen3-32B*				*Backbone: GPT-4.1*			
TA-SQL	82.99	93.47	**80.19**	**84.96**	75.55	90.71	**81.71**	**84.35**	67.08	88.06	30.64	45.46
RSL-SQL	91.84	97.74	37.83	51.30	85.92	95.75	47.90	60.43	80.83	88.19	25.43	35.24
ReFoRCE	34.01	67.50	76.07	67.81	19.30	51.29	68.05	55.02	35.00	69.58	53.73	54.33
DSR-SQL	89.12	96.78	25.48	38.41	86.70	**96.23**	23.65	35.77	53.33	82.02	**57.05**	**64.10**
AutoLink	93.88	96.91	10.29	17.50	39.50	74.39	16.27	24.86	85.83	94.55	8.11	14.18
APEX-SQL	**97.28**	**99.35**	39.02	52.78	**87.68**	96.15	51.95	64.10	**88.33**	**97.08**	36.52	46.96

queries $Q_{exp,t}$ at iteration t. To prevent large result sets from flooding the context window, APEX-SQL automatically compresses extensive outputs into a statistical summary $O_t = \sigma(\text{Exec}(Q_{exp,t}, \mathcal{D}^*))$, providing a compact yet comprehensive view of the data landscape.

Consolidation. As multi-step exploration progresses, APEX-SQL performs periodic state compression to prevent context saturation and maintain logical coherence as evidence accumulates. Once sufficient observations are collected, APEX-SQL synthesizes a consolidated status update that aligns observations with the overall logical plan and refines its understanding of schema semantics.

SQL Synthesis. After all implementation details are grounded in empirical evidence, APEX-SQL synthesizes a candidate SQL query S that maps the refined reasoning state onto the physical database syntax. The synthesized query is immediately validated against the database environment to ensure execution feasibility. If the execution $\text{Exec}(S, \mathcal{D}^*)$ returns an error, APEX-SQL leverages the resulting feedback to drive self-correction in subsequent iterations.

Confirmation. Following a successful execution, APEX-SQL enters a terminal state to perform secondary verification of semantic fidelity. Specifically, APEX-SQL evaluates whether the SQL logic aligns with the user requirements, the accumulated observations, and the latest logical plan. The output is finalized only after APEX-SQL confirms these conditions, thereby preventing logical drift during the multi-step generation process.

4 Experiments

Our evaluation focuses on the effectiveness of schema linking in reducing the search space while preserving relevant columns, and the downstream execution accuracy of the generated SQL queries.

4.1 Datasets

BIRD-Dev. The development set of BIRD [15], which is a large-scale cross-domain dataset consists of 1,534 queries across 11 databases. It requires the model to resolve ambiguities in fine-grained schema details, such as similarly named columns or dirty database values.

Spider 2.0-Snow. Spider 2.0 [10] represents the frontier of enterprise-level Text-to-SQL tasks. Its primary challenge lies in its massive schema size, featuring an average of over 800 columns per database and highly complex SQL structures. We focus on the Spider 2.0-Snow sub-task, which requires generating SQL queries in the Snowflake dialect, a standard for modern cloud data warehouses.

4.2 Experiment 1: Schema Linking Performance

4.2.1 Metrics. We evaluate schema linking as a retrieval task, aiming to identify the minimal sufficient subgraph required for the query. We follow prior work [3] to report the following metrics:

- **Strict Recall Rate (SRR)** measures the percentage of queries where the retrieved schema fully covers the ground truth. We consider SRR the most important metric, as it directly determines whether downstream SQL generation is possible to succeed.
- Non-Strict Recall (NSR) computes the proportion of required columns retrieved, even if the full subgraph is incomplete.
- Non-Strict Precision (NSP) measures the ratio of relevant columns in the retrieved set, reflecting the model's ability to filter noise.
- Non-Strict F1-Score (NSF) is the harmonic mean of NSR and NSP, offering a holistic view of retrieval quality.

4.2.2 Experimental Setup. To ensure a rigorous evaluation, we define specific testing splits and model backbones for each dataset. For BIRD-Dev, we use a widely adopted subset of 147 cases [12, 24] to evaluate the performance of DeepSeek-V3.2 [17] as the backbone, allowing us to test the model's capability within a controlled range and replicate more baselines. For full-set evaluation, we use Qwen3-32B [29]. For Spider 2.0-Snow, we utilize GPT-4.1 [1] on the 120-case subset with official golden SQLs[2], selected for its extended context window, which ensures both baseline and our method are not limited by context window size when processing Spider 2.0's larger schema. Ground truth columns are obtained by using both GPT-4.1 and DeepSeek-V3.2 to extract referenced columns from the golden SQLs, followed by manual verification. The instruction

[2]Full execution ground truth is available, but golden SQLs cover only 120 cases.

Table 2: Execution accuracy on BIRD-Dev ($N = 1534$) across difficulty levels. Results marked with * are locally reproduced, while others are collected from existing literature. Token consumption statistics are provided in Table 7.

Method	LLM	Sim. (925)	Mod. (464)	Chall. (145)	Total (1534)
C3-SQL	GPT-4	58.9	38.5	31.9	50.2
DAIL-SQL	GPT-4	62.5	43.2	37.5	54.3
TA-SQL	GPT-4	63.1	48.6	36.1	56.2
MAG-SQL	GPT-4	65.9	46.2	41.0	57.6
SuperSQL	GPT-4	66.9	46.5	43.8	58.5
MAC-SQL	GPT-4	65.7	52.7	40.3	59.4
MCS-SQL	GPT-4	70.4	53.1	51.4	63.4
Contextual-SQL*	GPT-4o	72.9	60.3	52.4	67.1
RSL-SQL	GPT-4o	74.4	57.1	53.8	67.2
CHESS	Gemini-1.5-Pro	-	-	-	68.3
DSR-SQL	DeepSeek-V3.1	72.7	61.2	**63.5**	68.3
AutoLink	Gemini-1.5-Pro	-	-	-	68.7
OpenSearch-SQL	GPT-4o	-	-	-	69.3
APEX-SQL	GPT-4o	**75.9**	**64.4**	57.2	**70.7**

templates F^{sl}_{plan} and F^{sl}_{agg} in Equ. 1, F^{sl}_{del} and F^{sl}_{sel} in Equ. 2, $F^{sl}_{semantics}$, F^{sl}_{exp}, and F^{sl}_{final} in Equ. 3 are provided in Appendix A.

4.2.3 Baselines. We compare our approach against comprehensive baselines, including CodeS [14] and RESDSQL [13], which are training-base methods, and TA-SQL [21], RSL-SQL [3], ReFoRCE [4], DSR-SQL [8], and AutoLink [26], which are training-free methods.

4.2.4 Results on Schema Linking. Table 1 presents the schema linking performance across two benchmarks and three experimental settings. APEX-SQL achieves the highest SRR across all settings, recording 97.28% on the BIRD subset, 87.68% on the BIRD full set, and 88.33% on Spider 2.0-Snow. We observe that existing training-based methods perform poorly on Spider 2.0, with significant drops in SRR. Among training-free methods, although TA-SQL, ReFoRCE, and DSR-SQL (on Spider 2.0) achieve higher NSP, their SRR is much lower than ours (e.g., ReFoRCE achieves 35.00% on Spider 2.0), rendering them unsuitable for downstream SQL generation, where missing columns can lead to critical errors. While AutoLink achieves an SRR close to ours on the BIRD subset and Spider 2.0, its NSP is noticeably lower, highlighting the superiority of APEX-SQL in maintaining precision while maximizing recall. Furthermore, APEX-SQL better adapts to weaker models, whereas AutoLink shows a significant drop in SRR on the BIRD full set (with Qwen3-32B).

4.2.5 Schema Linking Details. Logical Planning: We sample $N = 2$ paths at temperature 0.8 using only the query q as input, then aggregate them at temperature 0.2. Dual-Pathway Pruning: Columns are batched by table to maintain semantic integrity, with each batch constrained to 8–12k tokens. To minimize API overhead, we implement *Schema Merging*: tables with identical columns are consolidated into a single prompt entry that displays one representative schema alongside a list of all matching table names.

Table 3: Main results on Spider 2.0-Snow ($N = 547$), evaluating execution accuracy (EX), solution coverage (Pass@8), and average token consumption. v denotes number of votes. Baseline results are as reported in the literature.

Method	LLM	EX	Pass@8	Avg Token
Spider-Agent	DeepSeek-R1	10.79	-	-
Spider-Agent	o1-preview	23.77	-	-
ReFORCE	DeepSeek-R1	29.25	-	-
ReFORCE	o4-mini	29.80	31.99	$23k \cdot v$
ReFORCE	o3	35.83	39.85	$23k \cdot v$
DSR-SQL	DeepSeek-R1	35.28	-	-
APEX-SQL	DeepSeek-R1	**51.01**	**70.20**	**18k** $\cdot v$

4.3 Experiment 2: SQL Generation Performance

4.3.1 Metrics. We use **Execution Accuracy (EX)** as the evaluation metric, measuring the percentage of generated queries that return the exact result set as the ground truth[3]. In addition, following prior work [4], we also report **Pass@8** on Spider 2.0-Snow, where a question is considered solved if at least one correct SQL appears among eight candidates. Unless specified otherwise, we use the results reported in the official papers of the respective baselines.

4.3.2 Baselines. We compare APEX-SQL against open-source methods or those that have been reproduced by other works. For **BIRD-Dev**, we benchmark against C3-SQL [5], DAIL-SQL [7], TA-SQL [21], MAG-SQL [27], SuperSQL [11], MAC-SQL [25], Contextual-SQL [2], MCS-SQL [9], RSL-SQL [3], CHESS [24], DSR-SQL [8], AutoLink [26], and OpenSearch-SQL [28]. For **Spider 2.0-Snow**, we compare against Spider-Agent [10], ReFoRCE [4], and DSR-SQL [8].

4.3.3 Result Analysis. As presented in Tables 2 and 3, APEX-SQL consistently outperforms mainstream approaches. On **BIRD-Dev**, APEX-SQL achieves **70.7%** accuracy, surpassing retrieval-augmented methods like DAIL-SQL (54.3%$_{\downarrow 16.4}$) and RSL-SQL (67.2%$_{\downarrow 3.5}$). Despite slightly trailing DSR-SQL in the *Challenging* subset, APEX-SQL dominates the *Simple* and *Moderate* categories, demonstrating superior robustness against semantic ambiguity. This advantage extends to **Spider 2.0-Snow**, where APEX-SQL achieves **51.01%** with DeepSeek-R1, exceeding DSR-SQL (35.28%$_{\downarrow 15.73\%}$) and ReFORCE with o3 (35.83%$_{\downarrow 15.18\%}$). Notably, APEX-SQL attains a Pass@8 of **70.20%**, nearly doubling ReFoRCE-o3 (39.85%), indicating that our guided exploration effectively maximizes solution coverage. Additionally, APEX-SQL optimizes token consumption, using fewer tokens ($18k \cdot v$) than ReFoRCE ($23k \cdot v$) on Spider 2.0, and only 3.7k tokens per query with superior accuracy on BIRD (Appendix B). The comparison with DSR-SQL further highlights APEX-SQL's superiority: while DSR-SQL incorporates data profiling in SQL generation process, it lacks agentic exploration in the schema linking phase and guidance for exploration, achieving only 35.28% EX.

4.3.4 SQL Generation Details. We implement all agentic components in a training-free manner, with detailed instructions provided

[3]Compared to BIRD, Spider 2.0 adopts a relaxed evaluation protocol due to output format ambiguity in complex queries, tolerating irrelevant columns in the result set.

Table 4: Cross-Model Comparison on the Spider 2.0-Snow subset ($N = 120$) with oracle schema under three settings: w/ Exploration (ours), w/o Exploration (pure SQL generation with execution-guided refinement), and Oracle Exploration (implementation details derived from Ground Truth SQL), evaluated using EX, EX@8 (average EX score across 8 results), and Pass@8 metrics.

Base Model	w/ Exploration (Ours)			w/o Exploration			Oracle Exploration		
	EX	EX@8	Pass@8	EX	EX@8	Pass@8	EX	EX@8	Pass@8
GPT-4o	41.67	29.79	55.83	36.67$_{\downarrow 5.00}$	24.17$_{\downarrow 5.62}$	51.67$_{\downarrow 4.16}$	45.83$_{\uparrow 4.16}$	35.73$_{\uparrow 5.94}$	62.50$_{\uparrow 6.67}$
Kimi-k2-instruct	48.33	41.46	70.00	38.33$_{\downarrow 10.00}$	32.50$_{\downarrow 8.96}$	52.50$_{\downarrow 17.50}$	57.50$_{\uparrow 9.17}$	46.98$_{\uparrow 5.52}$	65.83$_{\downarrow 4.17}$
GPT-5	55.00	47.92	73.33	40.83$_{\downarrow 14.17}$	31.77$_{\downarrow 16.15}$	55.00$_{\downarrow 18.33}$	56.67$_{\uparrow 1.67}$	52.19$_{\uparrow 4.27}$	69.17$_{\downarrow 4.16}$
DeepSeek-R1	57.50	48.75	79.17	51.67$_{\downarrow 5.83}$	37.50$_{\downarrow 11.25}$	69.17$_{\downarrow 10.00}$	58.33$_{\uparrow 0.83}$	55.83$_{\uparrow 7.08}$	78.33$_{\downarrow 0.84}$
DeepSeek-V3.2	57.50	52.71	79.17	39.17$_{\downarrow 18.33}$	32.92$_{\downarrow 19.79}$	52.50$_{\downarrow 26.67}$	55.83$_{\downarrow 1.67}$	48.13$_{\downarrow 4.58}$	70.83$_{\downarrow 8.34}$
GPT-4.1	62.50	47.19	74.17	44.17$_{\downarrow 18.33}$	35.21$_{\downarrow 11.98}$	59.17$_{\downarrow 15.00}$	63.33$_{\uparrow 0.83}$	52.19$_{\uparrow 5.00}$	75.00$_{\uparrow 0.83}$
Gemini-3-Pro	66.67	**63.33**	78.33	57.50$_{\downarrow 9.17}$	41.46$_{\downarrow 21.87}$	62.50$_{\downarrow 15.83}$	70.00$_{\uparrow 3.33}$	60.94$_{\downarrow 2.39}$	78.33$_{\downarrow 0.00}$
Claude-4.5-Sonnet	**67.50**	61.67	**80.83**	50.83$_{\downarrow 16.67}$	48.44$_{\downarrow 13.23}$	64.17$_{\downarrow 16.66}$	69.17$_{\uparrow 1.67}$	67.92$_{\uparrow 6.25}$	79.17$_{\downarrow 1.66}$

in Appendix A. Guidance Retrieval: We map inferred natural language paths to the library $\mathcal{M}$ (provided in Appendix C) using keyword matching, which achieved > 95% recall in pilot tests on a BIRD-train subset. Data Profiling: The profiling function σ compresses result sets exceeding 30 rows into the top 10 rows plus statistics (row count, cardinality, types, and NULL ratios). For each question, we enforce a limit of 40 actions and 56k tokens; a mandatory SQL synthesis action is triggered at 38 actions or 52k tokens. For Spider 2.0-Snow cases with external knowledge (up to 30k tokens), we pre-filter the documentation for query-relevant snippets using the tested model (Appx. A). Context Management: During consolidation, we prune the interaction history to retain only exploratory queries, execution results, and the latest consolidated plan/understanding. Inference: We sample 8 candidates per task. For BIRD, we select the final answer using a reward model following [2]. For Spider 2.0-Snow, we use result-based majority voting with model selection (Appx. A) as a tie-breaker. The instruction F_{kw}^{sql} for generating SQL realization paths, and the definition of the Action space $\mathcal{A}$ are also provided in Appendix A.

4.4 Analysis

In this section, we present a series of experiments analyzing the impact of autonomous exploration, the scalability of test-time compute, and the contributions of core components in both the schema linking and SQL generation stages. We also present a case study in Appendix D. All experiments are conducted on the Spider 2.0-Snow subset, which includes official golden SQLs. We prioritize this benchmark due to its higher complexity, which provides more discriminative metrics, effectively highlighting the performance boundaries of different strategies. For SQL generation-related investigations (Sections 4.4.1 and 4.4.2), we consistently use the *oracle schema* setting, where models are provided with ground truth schema information. This approach isolates the impact of schema linking errors, ensuring that performance differences are attributed solely to SQL generation. We also introduce the **EX@8** metric, which calculates the average EX score across 8 results.

4.4.1 Effect of Exploration. To quantify the contribution of agentic exploration, we compare three settings: **w/ Exploration** (ours), **w/o Exploration** (pure SQL generation with execution-guided refinement), and **Oracle Exploration**[4]. Table 4 presents the results, revealing three critical insights:

Exploration as a Performance Multiplier. Exploration significantly improves performance across all metrics. For instance, DeepSeek-V3.2's EX score increases from 39.17% to 57.50% (↑18.33%). This shows that reasoning alone cannot fully resolve the gap between abstract logic and physical data states—empirical verification is essential for generating accurate enterprise-level SQL queries.

The "Rich Get Richer" Effect. We observe that exploration gains correlate with base model capability. Weaker models like GPT-4o show modest improvements (↑5.00% in EX), while stronger models such as DeepSeek-V3.2 and Gemini-3-Pro experience much larger gains (e.g., DeepSeek-V3.2 ↑18.33% in EX). This suggests that stronger models possess superior cognitive agility, allowing them to better formulate hypotheses, interpret exploration feedback, and synthesizing this information into more accurate SQL logic.

Autonomy vs. Oracle Guidance. Comparing autonomous exploration (w/ Exploration) with Oracle Exploration reveals a capability-dependent trade-off. For weaker models like GPT-4o, Oracle guidance consistently improves all metrics (e.g., Pass@8 ↑6.67%), suggesting that models with limited reasoning capacity benefit more from shortcuts. However, for stronger models, while Oracle improves EX, it often hurts Pass@8 (e.g., DeepSeek-V3.2 ↓8.34%). This indicates that while Oracle narrows the search space, autonomous exploration provides a broader understanding of the data. For DeepSeek-V3.2 and Gemini-3-Pro, self-exploration even leads to higher EX@8 scores than Oracle, indicating that stronger models more consistently benefit from exploration.

4.4.2 Test-Time Compute Scaling. To investigate the relationship between inference budget and performance scaling, we conduct a scalability analysis by performing $N = 8$ independent generation runs using the oracle schema. Figure 3 reveals three key insights:

Expanding Latent Potential. The growth of Pass@k confirms that scaling test-time compute effectively expands the solution search space, uncovering the models' latent reasoning potential. Notably, with a budget of $k = 8$, the coverage of GPT-4o ($Pass@8 = 55.8\%$) surpasses the average single-pass performance of DeepSeek-V3.2

[4] **Oracle Exploration** injects implementation details derived from Ground Truth SQL via GPT-5, including parsing strategies and implicit data handling steps that are not explicitly mentioned in the query or schema, such as how to join multiple tables.

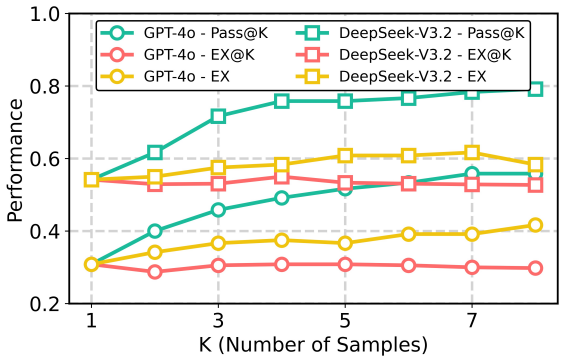

Figure 3: Performance scaling of GPT-4o and DeepSeek-V3.2 on Spider 2.0-Snow subset ($N = 120$) using the oracle schema.

($EX@k = 52.7\%$), suggesting that test-time compute scaling can effectively compensate for intrinsic model capabilities.

Performance Consistency & The Selection Gap. While Pass@k rises significantly, EX@k remains stable, suggesting high fidelity per trace rather than relying on brute-force sampling to rescue low-quality attempts. However, a substantial gap persists between the potential maximum (Pass@k) and the voting result (EX). For instance, DeepSeek-V3.2 achieves 79.17 Pass@8, yet voting captures only 57.50. This highlights that while the generative capability is sufficient, *answer selection* remains a bottleneck. In this work, we employ multiple sampling to mitigate generation variance rather than as a primary source of performance improvement, leaving the optimization of answer selection to future research.

4.4.3 Ablation Study on Schema Linking Components. We perform an ablation study to evaluate the contributions of key components in schema linking. Results are shown in Table 5.

The Recall-Precision Trade-off. We observe that our full method achieves lower precision (NSP) compared to baselines. We argue this behavior is accepted, as enterprise queries often allow for multiple valid solution paths. For instance, returning either `id` or `name` is acceptable for entity identification requirements. While ground truth SQL may select one, a robust schema linker must preserve all candidates to avoid irreversible pruning errors. As a result, we trade reduced precision for higher SRR, as missing a single critical column renders correct SQL generation impossible.

Logical Planning. Removing *Logical Planning* minimally impacts pruning stage SRR (↓ 0.3%) but notably affects final SRR (↓ 5.0%). This indicates that verbalizing query intent and constraints provides necessary context for the agent to correctly verify column roles during exploration, preventing false rejections of critical elements.

Dual-Pathway Pruning. In the pruning stage, relying on *Only Selection* or *Only Deletion* reduces SRR by 4.2% and 16.7%, respectively. Standard selection misses subtle columns, while aggressive deletion discards necessary structural keys. Combining both pathways, our method achieves the highest SRR of 97.5%. We note that the average of 383 retained columns corresponds to about 7k tokens, which fits well within the context window for verification.

Agentic Verification. The bottom section confirms that static methods are insufficient. Removing *Agentic Verification* entirely

Table 5: Ablation study on schema linking components. The evaluation is stratified into two stages: the Pruning Stage (top), measuring the quality of the intermediate candidate columns before verification, and the Verification Stage (bottom), assessing the final output. $\bar{C}$ denotes the average number of columns retained per query (3,430 before pruning).

Method	Spider 2.0-Snow ($N = 120$) SRR	NSR	NSP	NSF	$\bar{C}$
Workflow: Planning & Pruning (w/o Verification)					
APEX-SQL	**97.5**	**99.5**	11.6	19.3	383
Logical Planning					
w/o Planning	97.2$_{\downarrow 0.3}$	99.2$_{\downarrow 0.3}$	11.7$_{\uparrow 0.1}$	19.5$_{\uparrow 0.2}$	375
Dual-Pathway Pruning					
Only Deletion	80.8$_{\downarrow 16.7}$	95.4$_{\downarrow 4.1}$	**29.0**$_{\uparrow 17.4}$	**39.4**$_{\uparrow 20.1}$	217
Only Selection	93.3$_{\downarrow 4.2}$	98.5$_{\downarrow 1.0}$	27.6$_{\uparrow 16.0}$	37.6$_{\uparrow 18.3}$	305
Workflow: Planning & Pruning & Verification					
APEX-SQL	**77.5**	**94.9**	31.7	43.0	57
Logical Planning					
w/o Planning	72.5$_{\downarrow 5.0}$	94.6$_{\downarrow 0.3}$	37.2$_{\uparrow 5.5}$	49.5$_{\uparrow 6.5}$	33
Agentic Verification					
w/o Semantic Linking	63.3$_{\downarrow 14.2}$	91.3$_{\downarrow 3.6}$	37.2$_{\uparrow 5.5}$	48.2$_{\uparrow 5.2}$	30
w/o Data Profiling	68.3$_{\downarrow 9.2}$	90.8$_{\downarrow 4.1}$	41.4$_{\uparrow 9.7}$	53.4$_{\uparrow 10.4}$	25
w/o Global Synthesis	73.3$_{\downarrow 4.2}$	94.4$_{\downarrow 0.5}$	38.7$_{\uparrow 7.0}$	51.0$_{\uparrow 8.0}$	45
w/o All	55.8$_{\downarrow 21.7}$	82.7$_{\downarrow 12.2}$	**63.5**$_{\uparrow 31.8}$	**69.8**$_{\uparrow 26.8}$	12

Table 6: Ablation study on SQL generation. The evaluation compares performance with and without the guidance module, reporting EX@8 for quality, and average exploration rounds ($\bar{\mathcal{R}}$) and query count ($\bar{\mathcal{Q}}$) per example for efficiency.

LLM	Method	Spider 2.0-Snow ($N = 120$) EX@8 (%)↑	$\bar{\mathcal{R}}$ ↓	$\bar{\mathcal{Q}}$
GPT-4o	w/ Guid.	29.79	3.52	10.38
	w/o Guid.	26.04$_{\downarrow 3.75}$	3.96$_{\uparrow 0.44}$	9.99
GPT-5	w/ Guid.	47.92	1.23	6.25
	w/o Guid.	43.65$_{\downarrow 4.27}$	1.34$_{\uparrow 0.11}$	6.72
DeepSeek-R1	w/ Guid.	48.75	1.32	5.04
	w/o Guid.	47.50$_{\downarrow 1.25}$	1.69$_{\uparrow 0.37}$	6.97
GPT-4.1	w/ Guid.	47.19	1.62	10.57
	w/o Guid.	46.35$_{\downarrow 0.74}$	1.89$_{\uparrow 0.27}$	10.59

(w/o All) leads to a 21.7% drop in SRR. Specifically, *Semantic Linking* proves most critical (↓14.2%), as the agent must first hypothesize column roles (e.g., foreign keys) to guide exploration. *Table Exploration* is also vital (↓9.2%) for resolving ambiguities not addressed by metadata. Finally, *Global Synthesis* contributes to robustness (↓4.2%) by ensuring the selected columns form a connected graph.

4.4.4 Ablation Study on SQL Generation Components. While Section 4.4.1 demonstrates the impact of exploration, we now further isolate the contribution of the deterministic guidance module by

comparing it against a baseline without directives. We report EX@8 to remove the influence of answer selection. To quantify exploration efficiency, we define the average exploration rounds $\bar{\mathcal{R}}$ and the average query count $\bar{\mathcal{Q}}$ across m questions and $n = 8$ samples as:

$$\bar{\mathcal{R}} = \frac{1}{nm}\sum_{i=1}^{m}\sum_{j=1}^{n}\mathcal{R}_{i,j}, \quad \bar{\mathcal{Q}} = \frac{1}{nm}\sum_{i=1}^{m}\sum_{j=1}^{n}\mathcal{Q}_{i,j} \tag{6}$$

where $\mathcal{R}_{i,j}$ and $\mathcal{Q}_{i,j}$ denote the number of exploration rounds and total exploratory SQL queries generated for the i-th sample in the j-th run. As shown in Table 6, incorporating guidance consistently improves execution accuracy across all models (e.g., GPT-4o ↑ 3.75%). This confirms that guidance helps the agent start with a stronger foundation, leading to more focused and efficient exploration.

Beyond performance gains, the guidance module streamlines the interaction process. Across all models, $\bar{R}$ decreases when guidance is provided, indicating that fewer steps are needed to resolve uncertainties. Notably, while the number of rounds decreases, the total query count $\bar{Q}$ remains comparable or increases, suggesting that the agent performs more meaningful probes per interaction.

5 Conclusion

In this work, we present APEX-SQL, an agentic framework shifting the Text-to-SQL paradigm from passive schema perception to agentic exploration. For schema linking, we combine logical planning with dual-pathway pruning to navigate massive search spaces, employing agentic exploration to ground column selection in empirical evidence. For SQL generation, we utilize deterministic retrieval to guide data profiling, allowing the agent to resolve implementation details and value ambiguities. Experiments on BIRD and Spider 2.0-Snow show that APEX-SQL outperforms competitive baselines. Analysis reveals that agentic exploration acts as a performance multiplier, particularly for stronger models, while deterministic guidance enhances effectiveness and efficiency. We conclude that equipping models to verify hypotheses against physical data is a highly effective mechanism for robust enterprise data analysis.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[2] Sheshansh Agrawal and Thien Nguyen. 2025. Open-Sourcing the Best Local Text-to-SQL System. https://contextual.ai/blog/open-sourcing-the-best-local-text-to-sql-system/

[3] Zhenbiao Cao, Yuanlei Zheng, Zhihao Fan, Xiaojin Zhang, Wei Chen, and Xiang Bai. 2024. Rsl-sql: Robust schema linking in text-to-sql generation. *arXiv preprint arXiv:2411.00073* (2024).

[4] Minghang Deng, Ashwin Ramachandran, Canwen Xu, Lanxiang Hu, Zhewei Yao, Anupam Datta, and Hao Zhang. [n. d.]. Reforce: A Text-to-SQL agent with self-refinement, format restriction, and column exploration. In *ICLR 2025 Workshop: VerifAI: AI Verification in the Wild.*

[5] Xuemei Dong, Chao Zhang, Yuhang Ge, Yuren Mao, Yunjun Gao, Jinshu Lin, Dongfang Lou, et al. 2023. C3: Zero-shot text-to-sql with chatgpt. *arXiv preprint arXiv:2307.07306* (2023).

[6] Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2023. Text-to-sql empowered by large language models: A benchmark evaluation. *arXiv preprint arXiv:2308.15363* (2023).

[7] 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 (2024), 1132–1145.

[8] Zhifeng Hao, Qibin Song, Ruichu Cai, and Boyan Xu. 2025. Text-to-SQL as Dual-State Reasoning: Integrating Adaptive Context and Progressive Generation. *arXiv preprint arXiv:2511.21402* (2025).

[9] Dongjun Lee, Choongwon Park, Jaehyuk Kim, and Heesoo Park. 2025. Mcs-sql: Leveraging multiple prompts and multiple-choice selection for text-to-sql generation. In *Proceedings of the 31st International Conference on Computational Linguistics*. 337–353.

[10] Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, et al. 2024. Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows. *arXiv preprint arXiv:2411.07763* (2024).

[11] Boyan Li, Yuyu Luo, Chengliang Chai, Guoliang Li, and Nan Tang. 2024. The Dawn of Natural Language to SQL: Are We Fully Ready? *Proceedings of the VLDB Endowment* 17, 11 (2024), 3318–3331.

[12] Boyan Li, Jiayi Zhang, Ju Fan, Yanwei Xu, Chong Chen, Nan Tang, and Yuyu Luo. [n. d.]. Alpha-SQL: Zero-Shot Text-to-SQL using Monte Carlo Tree Search. In *Forty-second International Conference on Machine Learning*.

[13] Haoyang Li, Jing Zhang, Cuiping Li, and Hong Chen. 2023. Resdsql: Decoupling schema linking and skeleton parsing for text-to-sql. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 13067–13075.

[14] 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 (2024), 1–28.

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

[16] Weibin Liao, Xin Gao, Tianyu Jia, Rihong Qiu, Yifan Zhu, Yang Lin, Xu Chu, Junfeng Zhao, and Yasha Wang. 2025. LearNAT: Learning NL2SQL with AST-guided Task Decomposition for Large Language Models. *arXiv preprint arXiv:2504.02327* (2025).

[17] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437* (2024).

[18] Xinyu Liu, Shuyu Shen, Boyan Li, Peixian Ma, Runzhi Jiang, Yuxin Zhang, Ju Fan, Guoliang Li, Nan Tang, and Yuyu Luo. 2025. A Survey of Text-to-SQL in the Era of LLMs: Where are we, and where are we going? *IEEE Transactions on Knowledge and Data Engineering* (2025).

[19] Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan O Arik. 2024. Chase-sql: Multi-path reasoning and preference optimized candidate selection in text-to-sql. *arXiv preprint arXiv:2410.01943* (2024).

[20] Mohammadreza Pourreza and Davood Rafiei. 2023. Din-sql: Decomposed in-context learning of text-to-sql with self-correction. *Advances in Neural Information Processing Systems* 36 (2023), 36339–36348.

[21] 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*. 5456–5471.

[22] Liang Shi, Zhengju Tang, Nan Zhang, Xiaotong Zhang, and Zhi Yang. 2025. A survey on employing large language models for text-to-sql tasks. *Comput. Surveys* 58, 2 (2025), 1–37.

[23] Vladislav Shkapenyuk, Divesh Srivastava, Theodore Johnson, and Parisa Ghane. 2025. Automatic Metadata Extraction for Text-to-SQL. *arXiv preprint arXiv:2505.19988* (2025).

[24] Shayan Talaei, Mohammadreza Pourreza, Yu-Chen Chang, Azalia Mirhoseini, and Amin Saberi. 2024. Chess: Contextual harnessing for efficient sql synthesis. *arXiv preprint arXiv:2405.16755* (2024).

[25] Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Linzheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, et al. 2025. Mac-sql: A multi-agent collaborative framework for text-to-sql. In *Proceedings of the 31st International Conference on Computational Linguistics*. 540–557.

[26] Ziyang Wang, Yuanlei Zheng, Zhenbiao Cao, Xiaojin Zhang, Zhongyu Wei, Pei Fu, Zhenbo Luo, Wei Chen, and Xiang Bai. 2025. AutoLink: Autonomous Schema Exploration and Expansion for Scalable Schema Linking in Text-to-SQL at Scale. *arXiv preprint arXiv:2511.17190* (2025).

[27] Wenxuan Xie, Gaochen Wu, and Bowen Zhou. 2024. Mag-sql: Multi-agent generative approach with soft schema linking and iterative sub-sql refinement for text-to-sql. *arXiv preprint arXiv:2408.07930* (2024).

[28] Xiangjin Xie, Guangwei Xu, Lingyan Zhao, and Ruijie Guo. 2025. OpenSearch-SQL: Enhancing Text-to-SQL with Dynamic Few-shot and Consistency Alignment. arXiv:2502.14913 [cs.CL] https://arxiv.org/abs/2502.14913

[29] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388* (2025).

[30] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *arXiv preprint arXiv:1809.08887* (2018).

A Instruction Templates

This section provides the instruction templates referenced in the main text. Each prompt corresponds to specific components of our framework, as detailed below:

- F^{sl}_{plan} and F^{sl}_{agg}: For Logical Planning in Section 3.2.1.
- F^{sl}_{del} and F^{sl}_{sel}: For Dual-Pathway Pruning in Section 3.2.2.
- $F^{sl}_{semantics}$, F^{sl}_{exp}, and F^{sl}_{final}: For Agentic Exploration in Section 3.2.3.
- F^{sql}_{kw}: For SQL realization path generation in Section 3.3.1.
- $\mathcal{A}$: Defining Action Space in Section 3.3.2.
- The instructions for Evidence linking and model-based Answer Selection (as a tie-breaker) are discussed in Section 4.3.4.

Logical Planning (F^{sl}_{plan})

*** TASK CONTEXT ***
You are a Lead Data Architect. Your task is to break down the User Question into abstract logical steps needed to answer it.

IMPORTANT: Do NOT reference specific table or column names yet. Focus purely on the logic (e.g., filter, join, count, aggregate).

*** USER QUESTION ***
{question}

*** OUTPUT FORMAT ***
```
{
  "logical_steps": [
    "1. Identify [Entity]...",
    "2. Filter where [Condition]...",
    "3. Link [Entity A] to [Entity B]...",
    "4. Calculate [Aggregation]..."
  ]
}
```

Aggregating Plan Candidates (F^{sl}_{agg})

*** TASK CONTEXT ***
We have collected {len(candidates)} draft logical plans. Synthesize them into a single, comprehensive Master Logical Plan. Ensure the steps cover all conditions, filters, joins, and aggregations required.

*** USER QUESTION ***
{question}

*** DRAFT PLANS ***
{plans_text}

Output just the steps as a numbered list.

Identifying Deletion Set (F^{sl}_{del})

*** TASK CONTEXT ***
You are a Lead Data Architect. You have a Logical Plan to answer a query.
Your task: **Negative Pruning**. Identify database tables or columns that are **100% IRRELEVANT** to the plan.

*** USER QUESTION ***
{question}

*** MASTER LOGICAL PLAN ***
{logical_plan}

*** FULL DATABASE SCHEMA ***
{schema}

*** EVIDENCE ***
{evidence}

*** STRICT GUIDELINES ***
1. **High Recall (Safety)**:
   - If the column name is related to the query (even 1% chance), you should keep it. If not, check the desciption to see if it is related to the query. Sometimes the description is not clear, then you should pay close attention to the sample rows of the table. If the sample values of some columns are related to the query, you should keep these columns. If all of these information are not clear enough, remove it.
2. **Definition of Relevance**: Relevance includes both **Lexical Matching** and **Semantic Relatedness** over column name and description.
   - **Lexical**: If a word from the query appears in the name (e.g., query mentions "school" -> keep ‘school_code‘, ‘school_type‘, etc.), it MUST be retained.
   - **Semantic**: Keep tables/columns conceptually related to the topic. For example, if the query asks about "patents that were granted in ...", then the column ‘grant_date‘ should be kept.
   - **CRITICAL**: Do NOT remove discriminator columns such as ‘xxx_id‘, ‘xxx_name‘, ‘xxx_code‘, or ‘xxx_type‘ if the table itself is kept.
3. **Output Removal List**:
   - **Tables**: If a whole table is irrelevant, list it in ‘obviously_irrelevant_tables‘. Then all columns of that table will be kept. You do NOT need to list its columns separately.
   - **Columns**: If specific columns of a table are noise, list them in ‘obviously_irrelevant_columns‘.
4. **Grouped Tables**: If multiple tables are presented as sharing the same columns, you MUST list the removal instructions for **EACH** table explicitly. Pay close attention to name differences within the group (e.g., xx_2017 vs xx_2026), as these reflect specific data dimensions (like time) that determine relevance to the query.

*** OUTPUT FORMAT ***
```json
{
  "obviously_irrelevant_tables": ["table_unused_1", "table_unused_2"],
  "obviously_irrelevant_columns": [
    {
      "table": "t1",
      "columns": ["col_unused_1", "col_unused_2"]
    }
  ]
}
```

Identifying Preservation Set (F^{sl}_{sel})

*** TASK CONTEXT ***
You are a Lead Data Architect. You have a Logical Plan to answer a query.
Your task: **Positive Selection**. Identify database tables or columns that are **RELEVANT** or **NECESSARY** to the plan.

*** USER QUESTION ***
question

*** MASTER LOGICAL PLAN
{logical_plan}

*** FULL DATABASE SCHEMA ***
{schema}

*** EVIDENCE ***
{evidence}

*** STRICT GUIDELINES ***
1. **High Recall (Safety)**: Select ALL columns that might be useful for joining, filtering, grouping, or returning results. If you are not sure about the relevance of a column, e.g., the name and the description are ambiguous, **PICK IT**.
2. **Definition of Relevance**: Relevance includes both **Lexical Matching** and **Semantic Relatedness** over column name and description.
   - **Lexical**: If a word from the query appears in the table or column name (e.g., query mentions "school" -> keep 'school_code', 'school_type', etc.), it MUST be selected.
   - **Semantic**: Identify tables/columns conceptually related to the topic. For example, if the query asks about "patents that were granted in ...", then the column 'grant_date' should be kept.
   - **Discriminators**: ALWAYS select primary keys and common identifiers ('xxx_id', 'xxx_code', 'xxx_name') for relevant tables, as they are needed for joins.
3. **Output Selection List**:
   - **Tables**: If a whole table is relevant, list it in 'relevant_tables'.
   - **Columns**: List specific useful columns in 'relevant_columns'. If a table is already listed in 'relevant_tables', the columns can be omitted.
4. **Grouped Tables**: If multiple tables are presented as sharing the same columns, you MUST list the selection instructions for **EACH** table explicitly. Pay close attention to name differences within the group (e.g., xx_2017 vs xx_2026), as these reflect specific data dimensions (like time) that determine relevance to the query.

*** OUTPUT FORMAT ***
```json
{
  "relevant_tables": ["table_useful_1"],
  "relevant_columns": [
    {
      "table": "t1",
      "columns": ["col_useful_1", "col_pk_id"]
    }
  ]
}
```

Semantic Linking ($F^{sl}_{semantics}$)

*** TASK CONTEXT ***
You are a Senior Data Architect. You have full visibility of the database schema and a user question.
Your goal is to perform **Semantic Linking**: Analyze the database structure and how it grounds the user's intent.
{CRITICAL_RULES}

*** USER QUESTION ***
{question}

*** Logical Plan ***
{logical_plan}

*** Evidence ***
{evidence}

*** DATABASE SCHEMA ***
{schema}

*** YOUR TASKS ***
1. **Database Structure Overview**: Describe the database structure in detail (e.g., 'A banking system with customers and transactions...').
2. **Query-Specific Content Analysis**: Analyze the query against the available columns. Identify which tables/columns are likely targets, filters, or join keys.

3. **Table Functional Analysis**: For EVERY potentially relevant table, describe its specific function regarding this query.
- Is it a **Target Table**? (Contains the answer columns)
- Is it a **Bridge Table**? (Doesn't have semantic data but is needed to join Table A and Table B via Foreign Keys)
- Is it a **Filtering Table**? (Contains columns for WHERE clauses)
- **CRITICAL**: A table may have multiple roles. If a table is needed as a BRIDGE, you MUST explicitly state that it connects Entity X and Entity Y, even if it looks empty of content.

*** OUTPUT FORMAT ***
{
"database_structure": "Database structure overview...",
"query_specific_content_analysis": "Detailed mapping of query terms to DB tables/columns/logic...",
"table_functions": {
"table_name_1": "Acts as a bridge table connecting Students and Classes via student_id and class_id.",
"table_name_2": "Contains the 'score' column needed for calculation and 'exam_date' for filtering."
}
}
Perform the semantic linking analysis:

Parallel Data Profiling (F^{sl}_{exp})

*** TASK CONTEXT ***
You are an agent exploring a database table to verify its relevance to a user question.
You must not explore randomly. You must verify if this table fits its anticipated role.
{CRITICAL_RULES}

*** TARGET TABLE: {table_name} ***
Columns:
{column_name} ({column_type}): {column_description}
...

*** USER QUESTION ***
{question}

*** ANTICIPATED ROLE ***
This table was identified as: {semantic_role}. Use this to guide your exploration.

*** Evidence ***
{evidence}

*** YOUR MISSION ***
Generate 3-8 SQLite queries to investigate. **Focus on understanding the table's semantics and utility.**

Motivation for Exploration:
1. **Semantic Alignment**: Check distinct values to understand what the column *means* versus what the query *needs*. (e.g., If column is 'type', does it contain the specific categories? If 'status', does it contain values like 'Active' or code '1'?)
2. **Granularity & Scope**: Verify the table's grain (e.g., is it one row per Order or per Item?). This determines if it supports the required aggregations.
3. **Bridge/Connectivity**: If this looks like a linking table, verify the Foreign Keys are populated (not all NULL) to ensure it can actually serve as a bridge.
4. **Data Quality**: Are critical columns (targets for filters or answers) usable, or are they mostly NULL?

*** OUTPUT FORMAT ***
Provide SQL queries in a single 'sql' block with comments explaining the *motivation*.
```sql
– Motivation: Checking distinct values in 'status' to see if it aligns with the query's filter requirement
SELECT DISTINCT status FROM table_name LIMIT 10;
```

Generate your exploration queries:

[*After Profiling ...*]

*** TASK ***
Based on the exploration history and results above, determine if table '{table_name}' is RELEVANT to the User Question.

*** EXPLORATION EVIDENCE ***
{observations}

*** USER QUESTION ***
{question}

*** EVIDENCE ***
{evidence}

*** DECISION GUIDELINES ***
- **Direct Match**: Contains the specific answer data.
- **Bridge Table**: Contains IDs needed to join other relevant tables (CRITICAL: Keep even if no other useful data).
- **Filter Source**: Contains columns needed to restrict the result.
- **Calculation Support**: Contains numerical columns needed for aggregation (e.g., 'score' for AVG, 'price' for SUM).

*** OUTPUT GUIDELINES ***
- 'relevance_reason': Explain the LOGICAL role (e.g., 'Provides the Join Key for X and Y', 'Contains the target column Z').
- 'observations': Summarize FACTUAL findings from exploration (e.g., 'Column A contains integer codes 1-5', 'Table is empty').
- 'table_summary': A concise summary of what this table represents in the context of the query.

*** OUTPUT FORMAT ***
```json
{
  "relevant": true/false,
  "relevant_columns": [
  {
    "column_name": "name",
    "relevance_reason": "...",
    "observations": "..."
  }
  ],
  "table_summary": "..."
}
```
Provide your analysis:

Global Synthesis (F^{sl}_{final})

*** TASK CONTEXT ***
You are the Lead Data Architect. We are synthesizing initial exploration findings.
Review the [MARKED RELEVANT] and [MARKED IRRELEVANT] tables. Fix blind spots.

*** USER QUESTION ***
{question}

*** EVIDENCE ***
{question}

*** SEMANTIC ANALYSIS ***
{db_summary}

*** SCHEMA STATUS ***
Table: {t_name} [MARKED RELEVANT/MARKED IRRELEVANT]
Columns:
{column_name} ({column_type}): {column_description}
Observations: {observations}
Reason: {reason}

*** YOUR MISSION ***
Determine the final list of columns required to write the SQL query.

You must ensure the selected columns form a connected graph (tables can be joined) and cover all functional requirements of the query.

*** SELECTION CRITERIA (FUNCTIONALITY) ***
Keep a column if it serves one of the following purposes:
1. **Identification**: Unique identifiers (IDs, Codes) needed to count or distinguish entities (Primary keys).
2. **Linking**: Columns needed to join two tables together (Foreign Keys).
3. **Filtering**: Columns involved in conditions (e.g., status='Active', date > 2023).
4. **Aggregation**: Numerical columns for calculations (Sum, Avg, Max, Min).
5. **Grouping & Sorting**: Columns used for 'GROUP BY' or 'ORDER BY'.
6. **Direct Result**: Columns explicitly requested in the output.
Note on Multi-Path: If multiple columns might serve the same purpose, KEEP ALL OF THEM. Alternative columns might help to construct another solution paths.
Note on Type of Entity: DO NOT guess the type of an unspecified entity even you have some prior knowledge, e.g., if the query contains location entity like 'Riverside', then ALL columns related to location (e.g., County, District, etc.) should be kept. Another example is 'Fresno County Office of Education' which is actually a full name of a district.

*** REJECTION REQUIREMENTS ***
If a column was marked as **[MARKED RELEVANT]** in the Schema Status but you decide to **REJECT** it, you MUST include it in the 'rejected_candidates' list with a 'reject_reason' explaining why it is unnecessary. You can NOT reject a column for the reason that it is only a potentially useful column.

*** INTERACTIVE PROCESS ***
You can perform up to MAX_REFINE_ROUNDS rounds of verification.
- To EXPLORE: Output 'exploration_queries' in JSON to test joins or content.
- To FINISH: Output '[CONFIRM]' in the JSON (or just output the final refined_schema without queries).

*** OUTPUT FORMAT ***
You MUST explicitly list rejected candidates to prove you considered them.
IMPORTANT: In 'rejected_candidates', ONLY list columns that were previously marked RELEVANT but you decided to reject, OR columns that look ambiguous. Do NOT list obviously irrelevant columns to save space.

```json
{
  "refined_schema": {
```

```
    "table_name": {
      "relevant_columns": [
        {
          "column_name": "...",
          "relevance_reason": "Functional reason (e.g.,
Needed for Filtering)"
        }
      ]
    }
  },
  "rejected_candidates": [
    {
      "table": "t1",
      "column": "c1",
      "reject_reason": "Originally marked relevant, but
rejected because..."
    }
  ],
  "exploration_queries": ["SELECT 1 FROM t1 JOIN t2 ON
t1.id=t2.id LIMIT 1"],
  "status": "EXPLORING" or "[CONFIRM]"
}
```

Begin refinement:

Inferring potential SQL realization paths (F_{kw}^{sql})

You are refining a logical plan. For each step, think about:
1. What information is needed
2. Different ways to obtain it (direct access, join, calculation, etc.)
3. Keywords that describe the operation

QUESTION: {question}
Evidence: {Evidence}
Schema: {Schema}
CURRENT PLAN: {Logical Plan}

YOUR TASK:
Refine the plan by analyzing each step. For each step, provide:
Step N: [Brief description]
- Info need: [What information is required]
- Possible paths: [List 2-3 ways to get this info, e.g., 'direct column X', 'join tables A-B', 'calculate using formula']
- Keywords: [table names, column names, operations like filter/join/aggregate, concepts]
- Evidence: [exact evidence text if applicable]

EXAMPLE:
Step 1: Filter for high schools
- Info need: Identify high school records
- Possible paths: 'school_type column', 'EILCode column', 'join with school_types table'
- Keywords: schools, school_type, EILCode, filter, high school
- Evidence: EILCode = 'HS' means high school

Step 2: Calculate average score
- Info need: Average of scores
- Possible paths: 'AVG(score_column)', 'SUM/COUNT formula', 'pre-computed avg_score column'
- Keywords: scores, average, AVG, aggregate, calculation

IMPORTANT:
- Focus ONLY on the logical steps needed to answer the question
- Do NOT specify output columns in this plan
- Evidence: preserve EXACTLY (formulas, column names, values)
- Paths: list alternatives naturally (don't force if only one way makes sense)
- Keywords: comprehensive but relevant
- Keep plan abstract (avoid specific table/column names unless from evidence)

Now refine the plan:

Action Space Definition ($\mathcal{A}$)

You are an expert SQL query generator. Your task is to convert natural language questions into SQL queries.

AVAILABLE ACTIONS
CRITICAL: Always start your response with EXACTLY ONE action tag ([EXPLORE], [REFINE], [SQL], or [CONFIRM]) at the very beginning.

[EXPLORE]
Execute SQL queries to explore database content and gather evidence.
Use this when you need to:
- Discover possible values in a column (e.g., DISTINCT values)
- Verify data formats or patterns
- Check relationships between tables
- Gather sample data to understand the database

Exploration Guidelines:
- Use LIMIT to restrict output when exploring specific values or samples.
- If you need to understand data distribution (e.g., range, distinct values), you may omit LIMIT. For large results (>30 rows), we will report: max value, min value, data format, and distinct values.

Format: Start with [EXPLORE] tag, then write SQL queries with comments:
```
[EXPLORE]
– Purpose: Check available product categories
SELECT DISTINCT category FROM products LIMIT 10;
– Purpose: Verify date format
SELECT date_column FROM orders LIMIT 5;
```
Important: After exploration, please use [REFINE] to analyze the results before generating SQL.

[REFINE]
Analyze exploration results, update your understanding, and plan the next steps.
Use this to:
- Summarize what you learned from exploration and the remaining problems
- Update your logical plan
- Plan the SQL query structure (JOINs, filters, aggregations, etc.)
- Decide if more exploration is needed or if you're ready to generate SQL

Format: Start with [REFINE] tag, then provide structured reasoning:
```
[REFINE]
### Findings from Exploration:
- [Summarize key discoveries]
### Updated Understanding:
- [How this changes your approach]
### Query Plan:
- [Step-by-step plan for the SQL query]
### Next Action:
- [EXPLORE more] OR [Generate SQL]
```

[SQL]
Generate the final SQL query.
Use this when you are confident about the query logic.

Format: Start with [SQL] tag, then provide the query: [SQL] ```sql <Your SQL query> ```

[CONFIRM]
Confirm the logic of the generated SQLs and the final result after SQL execution.
Use this ONLY after [SQL] execution returns a satisfactory result.
Format: [CONFIRM] <Brief description of what the query does>

Evidence Linking

You are an expert Data Analyst Assistant supporting a Text-to-SQL system.
We have a User Query and an External Knowledge Document (Markdown format) that contains business rules, calculation logic, or data dictionary definitions.
Your task is to **extract** every piece of information from the document that is relevant to the User Query.

Input Information
- **User Query**: {query}
- **Knowledge File Name**: {knowledge_file_name}
- **Original Knowledge Content**:
```markdown
{knowledge_content}
```

Extraction Instructions (CRITICAL)
1. **Goal: High Recall (Better Safe Than Sorry).**
- If any section, paragraph, definition, description, entity code, formula, or table row is **potentially** related to the entities, metrics, conditions, constraints, or logic in the query (even slightly), **KEEP IT**.
- Do NOT try to be concise. We prefer extra context over missing information.
- Only remove content that is obviously and strictly irrelevant (e.g., legacy codes not mentioned, definitions of completely unrelated departments).
2. **Maintain Context & Integrity.**
- Do NOT pick out single words or fragmented sentences.
- Keep entire paragraphs, list items, or table rows to ensure the context remains readable and authentic.
- If a calculation rule depends on previous lines (like a variable definition), include those lines too.
3. **Do Not Rewrite.**
- Do NOT summarize, paraphrase, or change the original text. **Copy and paste** the relevant sections exactly as they appear in the source.

Output
Output ONLY the extracted markdown content below, without any introductory or concluding text.

Answer Selection

You are a Senior Data Architect acting as a Judge. You are provided with a User Question, the Database Schema, and several Candidate Solutions generated by an AI agent.
Each candidate consists of:
1. **The Execution Strategy**: The logic derived after exploring the database (identifying specific tables, columns, and values).

2. **The Final SQL**: The query implementation (We have varified that the SQL is executable).
YOUR GOAL: Identify the SINGLE best candidate that is most likely to execute correctly and return the accurate answer.

*** DATABASE SCHEMA ***
{schema}

*** USER QUESTION ***
{question}

*** CANDIDATES ***
{candidates}

*** EVALUATION CRITERIA (Prioritize in this order) ***
1. **Specificity of Evidence (The "Verified" Test)**:
- **Favor** candidates where the Strategy explicitly lists *verified values* found during exploration.
- **Reject** candidates with vague strategies (e.g., "Filter by population metric" without stating *which* metric ID).
2. **Entity Isolation (The "Explosion" Test)**:
- Look at the Schema. If a table contains mixed data types (e.g., 'MetricID', 'EventType', 'Year'), the SQL **MUST** filter for a specific value.
- **Reject** candidates that aggregate a Fact/Event table without a 'WHERE' clause filtering for the specific metric/type (this leads to wrong sums).
3. **Logic Robustness (The "Safety" Test)**:
- **Ratios**: The SQL should handle zero denominators (e.g., 'WHERE denom > 0' or 'NULLIF').
- **Joins**: If the task involves multiple independent event tables (e.g., Sends, Opens), **Favor** candidates using 'UNION ALL' or 'FULL JOIN' strategies over simple 'INNER/LEFT JOIN' which might lose data.
4. **Consistency**:
- The SQL must strictly follow the Strategy. If the Strategy says "Filter X" but SQL does not, reject it.

*** OUTPUT INSTRUCTION ***
1. Analyze each candidate one by one based on the criteria above.
2. Compare the candidates (both the strategy and the SQL) to point out if some of them miss necessary filters (Entity Isolation) or lacks specific verified details.
3. Select the best candidate.
4. Output the chosen file name in this format:
```plaintext
xxx.sql
```

B Token Consumption Analysis

BIRD-Dev. As a supplement to Table 2, we collect token consumption statistics for baselines from publicly available reports, as summarized in Table 7. The results show that while some high-performing systems (e.g., CHESS-SQL) consume over 300k tokens per query, APEX-SQL achieves superior accuracy with only 3.7k tokens. Compared to lightweight methods like DAIL-SQL, APEX-SQL provides a substantial gain in execution accuracy with only a marginal increase in token usage. These findings indicate that agentic exploration is more cost-effective than passive strategies, enabling robust performance with significantly lower inference overhead.

Table 7: Execution accuracy versus token consumption on BIRD-Dev. Due to limited reporting in existing literature, token statistics are provided only for a subset of baselines. Specific sources and their respective table references are listed in the Source and Note columns. APEX-SQL demonstrates a competitive accuracy with a notable reduction in token overhead.

Method	EX (%)	Token (k) / query	Source	Note
RSL-SQL	67.2	12.4		
CHESS-SQL	68.3	351.8		-
MAC-SQL	59.4	6.9		
TA-SQL	56.2	7.3		Table V
C3-SQL	50.2	5.9		
DAIL-SQL	54.3	1.6		
SuperSQL	58.5	**1.4**		Table 5
APEX-SQL	**70.7**	3.7	-	-

Spider 2.0-Snow. We analyze token consumption of different methods on Spider 2.0-Snow in Table 3. Our approach effectively improves performance while reducing token usage, demonstrating the efficiency of agentic exploration.

C Tip Library Details

This document presents the complete tip library $\mathcal{M}$ used for SQL generation guidance (Section 3.3.1). The library contains 49 tips organized into 14 categories. We use DeepSeek-V3.2 to analyze a randomly selected 10% of BIRD-train cases to extract these tips. Each tip is formatted as it appears in the actual case prompts: [TIP_ID] Title followed by Description.

C.1 Categories Overview

- evidence_enforcement: Tips for correctly implementing evidence
- string_matching: Tips for choosing = vs LIKE
- output_columns: Tips for selecting correct output columns
- table_selection: Tips for choosing tables and joins
- column_interpretation: Tips for understanding column names
- schema_grounding: Tips for mapping logical concepts to schema
- join_strategy: Tips for implementing joins
- filter_implementation: Tips for implementing WHERE conditions
- aggregation: Tips for aggregation and calculations
- sorting_limiting: Tips for ORDER BY and LIMIT
- multi_step_logic: Tips for subqueries and CTEs

- sql_syntax: Tips for SQL syntax correctness
- common_pitfalls: Tips for avoiding common errors
- advanced_patterns: Tips for advanced SQL patterns

C.2 Complete Tip Listing

C.2.1 Category: Evidence Enforcement.

[TIP001] Evidence-Specified Column Must Be Used Exactly

When evidence specifies a column name, explore using that exact column. Do NOT substitute with similar columns.

[TIP002] Evidence-Specified Formula Must Be Implemented Exactly

When evidence provides a calculation formula, explore implementing it EXACTLY as specified. Do NOT use built-in functions that seem equivalent.

[TIP003] Evidence-Specified Value Must Be Used Exactly

When evidence provides value mappings, explore using the EXACT database value, not the English translation or interpretation.

[TIP004] Evidence Clarifies Ambiguous Terms

When evidence clarifies which column or table an ambiguous term refers to, explore following the evidence interpretation.

[TIP044] Evidence Formula Preservation - No Rephrasing Allowed

When evidence provides a formula or calculation method, explore preserving it EXACTLY. Do NOT rephrase, simplify, or use 'equivalent' built-in functions.

[TIP045] WHERE Clause from Evidence - Extract Exact Conditions

When evidence specifies how to filter data (column names, values, or conditions), explore extracting and implementing these EXACTLY.

C.2.2 Category: String Matching.

[TIP005] Use Exact Match (=) When Value Is Known

When question or evidence provides a specific value, explore using exact match (=). Only use LIKE for partial matching.

[TIP006] Use LIKE Only for Partial Matching

Explore using LIKE only when question explicitly asks for partial matching (contains, starts with, ends with).

[TIP007] After Exploration, Use Exact Values Not Patterns

After exploring to find possible values, use exact match (=) with the found value to explore, not LIKE pattern.

[TIP025] Choose String Matching Strategy Based on Question Intent

Explore choosing between exact match (=), prefix match (LIKE 'X%'), suffix match (LIKE '%X'), or contains (LIKE '%X%') based on question.

C.2.3 Category: Output Columns.

[TIP008] Exclude ORDER BY Columns from SELECT Unless Requested

Explore excluding columns used only for sorting (ORDER BY) from SELECT unless explicitly requested in question.

[TIP009] Return Columns in Exact Order Mentioned in Question

Explore returning output columns in the EXACT order they appear in the question. First mentioned → first in SELECT.

[TIP010] Exclude Filter Columns from SELECT Unless Requested

Explore excluding columns used only for filtering (WHERE) from SELECT unless explicitly requested.

[TIP011] For 'How Many' Questions, Return COUNT Only

When question asks 'how many', explore returning only COUNT, not individual records or other columns.

[TIP043] SELECT Clause Precision - Return ONLY Requested Columns

Explore ensuring the SELECT clause returns ONLY the columns explicitly requested in the question, in the exact order mentioned.

C.2.4 Category: Table Selection.

[TIP012] Check If Single Table Contains All Required Information

Before exploring JOIN, verify if all required information exists in a single table. Prefer exploring single-table solutions.

[TIP013] JOIN Only When Information Spans Multiple Tables

Explore using JOIN only when question explicitly needs information from multiple entities that exist in different tables.

[TIP014] Avoid Over-Engineering with Unnecessary Complexity

Explore avoiding JOINs, subqueries, or conditions that aren't needed. Use the simplest solution that answers the question.

C.2.5 Category: Column Interpretation.

[TIP015] Parenthetical Text in Column Names Is NOT a Filter Condition

When exploring, note that text in parentheses within column names (e.g., '(K-12)', '(Ages 5-17)') is part of the column name, NOT a filter condition.

[TIP016] Distinguish Column Name Context from Filter Conditions

When exploring, note that parenthetical text in column names provides context about the data, not additional filter conditions to apply.

C.2.6 Category: Schema Grounding.

[TIP017] Use Direct Name Matching for Column Identification

When exploring, if logical plan mentions a concept, look for columns with similar names first (case-insensitive, with variations).

[TIP018] Use Semantic Matching When Direct Names Don't Match

When exploring, if direct name matching fails, think about what the concept means and look for semantically related columns.

[TIP019] Always Verify Column Existence Before Using

When exploring, never assume a column exists. Always verify through exploration before using.

C.2.7 Category: Join Strategy.

[TIP020] Identify Join Keys from Schema Relationships

When exploring joins, look for foreign key relationships in schema. Common patterns: ID columns, Code columns, Name columns.

[TIP021] Verify Join Keys Have Matching Values

When exploring joins, verify that join keys have matching values in both tables and check for NULLs.

[TIP022] Choose Appropriate Join Type

When exploring joins, select join type based on whether you need all records from one/both tables or only matching records.

[TIP042] Prefix Columns with Table Alias When Joining

When exploring joins, prefix all columns with table alias to avoid ambiguous column errors.

C.2.8 Category: Filter Implementation.

[TIP023] Use Exact Value Filters When Value Is Specified

When question explicitly mentions a value, explore using exact match (=) for filtering.

[TIP024] Use Correct Numeric Comparison Operators

When exploring filters, translate question's numeric comparisons to correct operators. Pay attention to inclusive vs exclusive. BETWEEN is inclusive on both ends.

C.2.9 Category: Aggregation.

[TIP026] Choose Correct Aggregation Function

When exploring aggregations, select appropriate function based on what question asks: COUNT, SUM, AVG, MAX, MIN.

[TIP027] Protect Division Operations Against Zero and Use CAST

When exploring calculations with ratios or percentages, protect against division by zero and ensure float division with CAST.

[TIP028] Use GROUP BY When Aggregating Per Category

When exploring aggregations and question asks 'per category', 'by group', 'for each', use GROUP BY with the category column.

[TIP029] Use HAVING to Filter After Aggregation

When exploring aggregations, use HAVING clause to filter groups after aggregation. Use WHERE to filter rows before aggregation.

[TIP046] Precision Preservation - No Arbitrary Rounding

When exploring calculations, do NOT apply ROUND() unless the question explicitly specifies precision requirements.

[TIP047] Ratio Calculation - Count All Occurrences

When exploring ratio or percentage calculations, count ALL occurrences (including duplicates) unless the question explicitly asks for DISTINCT counts.

[TIP048] Percentage Calculation - CAST to REAL and Multiply by 100

When exploring percentage calculations, always CAST the numerator to REAL before division, then multiply by 100.

[TIP049] DISTINCT Usage - List vs Count

When exploring, use DISTINCT when listing entities (names, IDs, items) to avoid duplicates. Do NOT use DISTINCT when counting occurrences for statistical purposes unless explicitly asked for unique counts.

C.2.10 Category: Sorting and Limiting.

[TIP030] Implement ORDER BY Based on Question Keywords

When exploring sorting, translate question's keywords to correct ORDER BY direction: ASC for lowest/first, DESC for highest/last.

[TIP031] Use LIMIT for Top N or Single Extreme Value

When exploring and question asks for 'top N', 'first N', or single extreme value, use LIMIT with ORDER BY.

[TIP032] Filter Out NULLs When Finding Extreme Values

When exploring extreme values with ORDER BY, add WHERE column IS NOT NULL to avoid NULL results.

C.2.11 Category: Multi-Step Logic.

[TIP033] Use Subquery for Dependent Steps

When exploring and logical plan has dependent steps (step 2 needs result from step 1), explore using subquery or CTE.

[TIP034] Use CTE for Complex Multi-Step Logic

When exploring complex queries with multiple dependent steps, explore using CTE (WITH clause) for clarity.

C.2.12 Category: SQL Syntax.

[TIP035] Use Double Quotes for Identifiers with Spaces or Special Characters

When exploring, note that column/table names with spaces, special characters, or reserved words must be quoted with double quotes.

[TIP036] Use Aliases for Calculated Columns

When exploring calculations, give meaningful aliases to calculated columns, aggregations, and complex expressions using AS.

[TIP037] Cannot Use Column Alias in WHERE Clause of Same Level

When exploring, note that column aliases defined in SELECT cannot be used in WHERE clause of the same query level. Use subquery or repeat expression.

C.2.13 Category: Common Pitfalls.

[TIP038] Never Hard-Code Values Discovered in Exploration

When exploring, do NOT hard-code values you discovered unless they are explicitly mentioned in the question.

[TIP039] Handle NULL Values Appropriately

When exploring, be aware of NULL values in aggregations, comparisons, and joins. Filter or handle them explicitly.

[TIP040] Avoid Integer Division - Use CAST for Float Results

When exploring calculations in SQLite, note that dividing two integers gives integer result. Use CAST(x AS REAL) to get float result.

[TIP041] Apply Entity Isolation for Mixed-Data Tables

When exploring tables with mixed entity types (e.g., different metrics in same table), filter by entity type before aggregating.

C.2.14 Category: Advanced Patterns.

[TIP043] Use Conditional Aggregation with CASE

When exploring aggregations of different subsets in one query, use CASE inside aggregate functions.

[TIP044] Use Window Functions for Ranking

When exploring ranking, row numbering, or running calculations, use window functions (RANK, ROW_NUMBER, etc.).

[TIP045] Find Top N Per Group with Window Functions

When exploring top N records per category, use ROW_NUMBER() with PARTITION BY, then filter.

C.3 Tip Retrieval Relationships

The tip retrieval system uses rule-based keyword matching to select relevant tips for each question.

C.3.1 Universal Tips (Always Selected).

The following tips are included in every query regardless of content: **TIP009**, **TIP019**, **TIP035**, and **TIP038**.

C.3.2 Evidence-Based Retrieval. Tips selected based on patterns detected in the evidence field:

- Evidence contains phrases like "can be represented as [column]=" or "[column]=[value]" (excluding "refers to") → **TIP001**
- Evidence contains formulas with arithmetic operators, or phrases like "sum of count" or "average =" → **TIP002, TIP027, TIP040**

- Evidence contains "stands for" or "means" (with quoted values), or patterns like "'value' stands for/means/is/are" → **TIP003**
- Evidence contains "refers to" → **TIP004**

C.3.3 Question-Based Retrieval. Tips selected based on keywords and patterns in the question text:

- Question contains "with", "where", or "whose" followed by a quoted value → **TIP005, TIP023**
- Question contains: highest, lowest, top, bottom, maximum, minimum, best, worst, most, least, order by, sort → **TIP008, TIP030, TIP031, TIP032**
- Question mentions multiple outputs connected by "and" (e.g., "name and age"), but does NOT start with "how many" → **TIP009**
- Question starts with "what", "list", "name", or "show" and contains "where", "with", "whose", or "that have" → **TIP010**
- Question starts with "how many" (without "what" or "which"), or contains "list" + "lowest" + "amount" → **TIP011**
- Question or evidence contains parentheses → **TIP015, TIP016**
- Question contains: more than, less than, greater than, between, at least, at most, above, below, over, under, exceed → **TIP024**
- Question contains: total, sum, average, avg, count, maximum, minimum, aggregate, group by → **TIP026, TIP028**
- Question contains aggregation keywords (average, total, sum, count) combined with comparison keywords (more than, less than, greater, between) → **TIP029**
- Question contains: calculate, ratio, percentage, average → **TIP036, TIP046**
- Question contains: ratio, percentage, percent, proportion, rate → **TIP047, TIP048**
- Question starts with: how many, what is, what are, which, list all, list the → **TIP049**
- Question contains: contain, include, start with, end with, like → **TIP006**
- Question contains: name, title, description → **TIP025**
- Question contains: null, missing, empty → **TIP039**

C.3.4 Logical Plan-Based Retrieval. Tips selected based on patterns in the logical plan structure:

- Plan contains "join", or mentions 4+ tables, or question contains patterns like "X and their/its Y" → **TIP012, TIP013, TIP020, TIP021, TIP022, TIP042**
- Plan contains both "subquery" and "nested", or has 4+ occurrences of "join" → **TIP014**
- Plan has 4+ steps, or contains both "subquery" and "nested", or uses CTE pattern (WITH...AS...SELECT) → **TIP033, TIP034**
- Plan contains "case when", or contains both "different" and "aggregate" → **TIP043**
- Plan contains: rank, ranking, row number, top n per, per group, partition → **TIP044, TIP045**

C.3.5 Schema-Based Retrieval. Tips selected based on database schema characteristics:

- Schema contains entity type columns (metricid, metric_id, event_type, entity_type, data_type) and question contains filtering keywords (specific, certain, particular, only, filter by type, where type) → **TIP041**

D Case Study

To demonstrate the effectiveness of agentic exploration in SQL generation, we present a representative case from the Spider 2.0-Snow benchmark. This case (**sf_bq028**) illustrates how agentic exploration enables the model to discover critical data quality issues and adapt its strategy accordingly, leading to successful query generation where the baseline method failed.

D.1 Case sf_bq028: Data Quality Discovery Through Exploration

D.1.1 Task Description. Considering only the latest release versions of NPM packages, which packages are the top 8 most popular based on the Github star number, as well as their versions?

D.1.2 Database Schema. The task involves three tables from the `DEPS_DEV_V1` database:

Table: PROJECTS.

- `Name` (TEXT): Project identifier (e.g., 'steven-tey/dub')
- `Type` (TEXT): Project type (e.g., 'GITHUB', 'GITLAB')
- `StarsCount` (NUMBER): Number of stars for the project
- `SnapshotAt` (NUMBER): Timestamp of data snapshot

Table: PACKAGEVERSIONS.

- `Name` (TEXT): Package name (e.g., '@dub/ui')
- `Version` (TEXT): Version string (e.g., '1.1.6')
- `System` (TEXT): Package system (e.g., 'NPM', 'PYPI')
- `VersionInfo` (VARIANT): JSON containing `IsRelease` (boolean) and `Ordinal` (number)
- `UpstreamPublishedAt` (NUMBER): Publication timestamp (often NULL)
- `SnapshotAt` (NUMBER): Timestamp of data snapshot

Table: PACKAGEVERSIONTOPROJECT.

- `System` (TEXT): Package system
- `Name` (TEXT): Package name
- `Version` (TEXT): Package version
- `ProjectType` (TEXT): Type of linked project (e.g., 'GITHUB')
- `ProjectName` (TEXT): Name of linked project
- `RelationType` (TEXT): Relationship type (e.g., 'SOURCE_REPO_TYPE')

D.1.3 Direct SQL Generation Result & Analysis. The baseline method directly assumed that `UpstreamPublishedAt` could be used to identify the latest release version of each NPM package.

Generated SQL strategy:

- Filter `PACKAGEVERSIONS` for `System = 'NPM'` and `VersionInfo:IsRelease = TRUE`
- Use `ROW_NUMBER() OVER (PARTITION BY "Name" ORDER BY "UpstreamPublishedAt" DESC)` to find latest release
- Filter `WHERE "UpstreamPublishedAt" IS NOT NULL`

Critical error: The baseline SQL included the condition:

```
WHERE "UpstreamPublishedAt" IS NOT NULL
```

Why this failed: This assumption was made without verification. The baseline had no way to know whether `UpstreamPublishedAt` was actually populated for NPM release versions. As it turns out,

this field is NULL for the majority of NPM releases, causing the query to miss most packages and return incorrect results.

Result. Failed (Score: 0)

D.1.4 APEX-SQL *Result & Analysis.*

Exploration Round 1: Timestamp Availability Check

Query 1: Check if UpstreamPublishedAt is populated for release versions

```
SELECT "UpstreamPublishedAt", "VersionInfo"
FROM "PACKAGEVERSIONS"
WHERE "System" = 'NPM' AND "VersionInfo":IsRelease = true
LIMIT 10;
```

Result: All 10 samples showed `UpstreamPublishedAt = None`

Query 2: Check if VersionInfo contains alternative timestamp

```
SELECT "VersionInfo"
FROM "PACKAGEVERSIONS"
WHERE "System" = 'NPM' AND "VersionInfo":IsRelease = true
LIMIT 10;
```

Result: VersionInfo contains `IsRelease` boolean and `Ordinal` number, but no timestamp

Query 3: Check SnapshotAt as fallback

```
SELECT "SnapshotAt", "VersionInfo", "UpstreamPublishedAt"
FROM "PACKAGEVERSIONS"
WHERE "System" = 'NPM' AND "VersionInfo":IsRelease = true
AND "UpstreamPublishedAt" IS NULL
LIMIT 10;
```

Result: SnapshotAt is consistent across records (system snapshot time, not release time)

Key Discovery: `UpstreamPublishedAt` is NULL for NPM releases, so it cannot be used!

Updated strategy:

Evidence: `UpstreamPublishedAt` is NULL for all NPM release versions, so cannot be used for timestamp. `VersionInfo` contains `IsRelease` boolean and `Ordinal` number but no timestamp.

Updated Plan: Use `VersionInfo:Ordinal` to find latest release per package via `ROW_NUMBER() OVER (PARTITION BY "Name" ORDER BY "VersionInfo":Ordinal DESC)`.

Exploration Round 2: Quantify the Impact

Query: Count NULL UpstreamPublishedAt in releases

```
SELECT COUNT(*) AS null_count
FROM "PACKAGEVERSIONS"
WHERE "System" = 'NPM' AND "VersionInfo":IsRelease = true
AND "UpstreamPublishedAt" IS NULL;
```

Result: `null_count = 3,935,243` (out of ~9M total NPM releases)

Critical Finding: 43.7% of NPM releases have NULL UpstreamPublishedAt! (The baseline's assumption would have excluded nearly half of all packages.)

Exploration Round 3: Verify Alternative Approach

Query 1: Verify VersionInfo structure

```
SELECT "VersionInfo"
FROM "PACKAGEVERSIONS"
WHERE "System" = 'NPM' AND "VersionInfo":IsRelease = TRUE
LIMIT 5;
```

Result: Confirmed `Ordinal` field exists and can be used for ordering

Query 2: Verify project name format for joins

```
SELECT DISTINCT "ProjectName"
FROM "PACKAGEVERSIONTOPROJECT"
WHERE "ProjectType" = 'GITHUB'
LIMIT 5;
```

Result: Confirmed format matches `PROJECTS.Name` (e.g., 'steventey/dub')

Final SQL Generation. Based on verified evidence, the model generated:

```
WITH LatestReleases AS (
    SELECT
        "Name",
        "Version",
        ROW_NUMBER() OVER (
            PARTITION BY "Name"
            ORDER BY "VersionInfo":Ordinal DESC
        ) AS version_rank
    FROM "PACKAGEVERSIONS"
    WHERE
        "System" = 'NPM'
        AND "VersionInfo":IsRelease = true
),
PackageProjects AS (
    SELECT DISTINCT
        lr."Name",
        lr."Version",
        pvp."ProjectName"
    FROM LatestReleases lr
    JOIN "PACKAGEVERSIONTOPROJECT" pvp
        ON lr."Name" = pvp."Name"
        AND lr."Version" = pvp."Version"
        AND pvp."ProjectType" = 'GITHUB'
    WHERE lr.version_rank = 1
)
SELECT
    pp."Name" AS PackageName,
    pp."Version",
    p."StarsCount"
FROM PackageProjects pp
JOIN "PROJECTS" p
    ON pp."ProjectName" = p."Name"
    AND p."Type" = 'GITHUB'
ORDER BY p."StarsCount" DESC
LIMIT 8;
```

Result. Success (Score: 1)

D.1.5 Key Insights. Why exploration was essential:

(1) Data Quality Discovery: Exploration revealed that `UpstreamPublishedAt` is NULL for 43.7% of NPM releases—a critical data quality issue that the baseline could not detect.

(2) Alternative Strategy: After discovering the NULL issue, exploration identified `VersionInfo:Ordinal` as a reliable alternative for version ordering.
(3) Verification Before Commitment: The model verified the alternative approach before generating the final SQL, ensuring correctness.

This case demonstrates that exploration is not guessing—it's systematic data quality verification that enables the model to adapt its strategy based on actual database characteristics.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009