TSSDA: An Autonomous Agent for Constructing a Time-Series Text-to-SQL Benchmark

Zhihan Zheng
Tsinghua University, China
zhengzh26@mails.tsinghua.edu.cn

Shaoxu Song
Tsinghua University, China
sxsong@tsinghua.edu.cn

Haitao Yuan
NTU, Singapore
haitao.yuan@ntu.edu.sg

Minxiao Chen
BUPT, China
chenminxiao@bupt.edu.cn

arXiv:2611.02132v1 [cs.LG] 1 Nov 2026

ABSTRACT

As agents increasingly undertake data querying, Text-to-SQL serves as a foundational capability for database access. However, for text-to-SQL in time-series databases, high-quality benchmarks and large-scale training corpora remain scarce, limiting the systematic evaluation and domain adaptation of existing models and agents. Existing automated SQL generation approaches lack sufficient autonomy, leaving key challenges unresolved. First, numerous combinations of temporal syntax and functions complicate comprehensive coverage of valid queries. Second, as the corpus grows, generating valuable and diverse new data becomes increasingly challenging. Third, schema and data heterogeneity hinder reuse of temporal query structures, limiting generation efficiency. To address these challenges, we propose **TSSDA**, a **T**ime-**S**eries **S**QL (TSSQL) Generation **D**ata **A**gent framework, and construct **TERN**, a comprehensive **T**ime-s**ER**ies text-to-SQL be**N**chmark. First, to achieve broad coverage of valid queries, we systematically model TSSQL syntax and semantics to construct a data generation space. Next, to explore the space and improve diversity, we develop a Data Agent that autonomously orchestrates data generation. The agent comprises a planning and decision-making module and a query generation module, with online adaptation for continuous self-improvement. Finally, we introduce a temporal-aware memory that enables adaptive reuse of experience across heterogeneous data instances, improving generation efficiency. Comprehensive evaluations on TERN reveal the limitations of existing methods in performing Text-to-TSSQL tasks. Furthermore, extensive experiments demonstrate that TSSDA substantially improves coverage, diversity, and complexity.

PVLDB Reference Format:
Zhihan Zheng, Shaoxu Song, Haitao Yuan, and Minxiao Chen. TSSDA: An Autonomous Agent for Constructing a Time-Series Text-to-SQL Benchmark. PVLDB, 20(1): XXX-XXX, 2027.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/chris265100/TSSDA.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 20, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

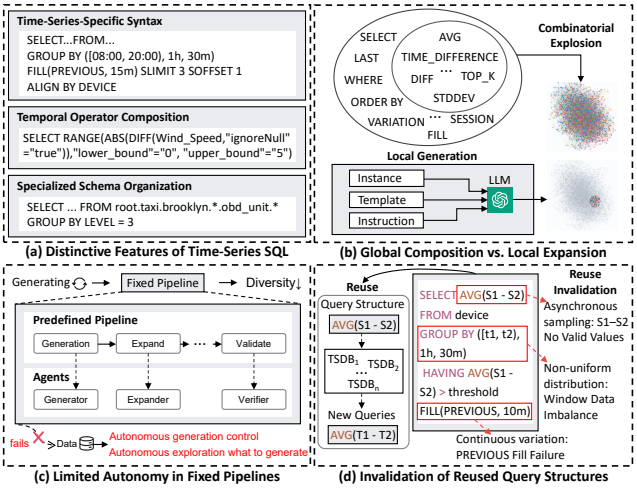


Figure 1: Challenges of Text-to-SQL for TSDBs

1 INTRODUCTION

With advances in large language models (LLMs), agents increasingly undertake data querying [3, 78] and processing [10, 22] tasks. Text-to-SQL, which translates natural language requests into SQL queries, is becoming a foundational capability for these agents [24, 39]. Numerous approaches [31, 33, 71] have emerged, alongside increasingly challenging benchmarks [7, 25, 28]. However, existing studies primarily focus on relational databases, leaving Text-to-SQL for time-series databases (TSDBs) largely underexplored.

SQL in TSDBs differs fundamentally from conventional SQL in both query semantics and language constructs. As shown in Fig. 1(a), we highlight several distinctive features of time-series SQL (TSSQL). First, TSDBs support diverse time-series-specific syntax, including temporal windows, missing-value filling, and result alignment, as illustrated in the example. Second, TSSQL often composes multiple functions and operators for complex temporal analytics. The example detects absolute wind-speed changes exceeding 5 between consecutive valid readings, while real-world queries may be more complex. Third, TSDBs exhibit diverse schema organizations and are tightly coupled with time-series syntax. For example, we illustrate IoTDB's schema [66] with a query aggregating matching OBD sensor readings by vehicle in Brooklyn. Beyond SQL structure, time-series queries also embody rich and distinctive temporal semantics. Consequently, existing Text-to-SQL methods designed for relational databases are inapplicable to TSDBs.

To comprehensively evaluate Text-to-SQL for TSDBs and provide a domain-specific training corpus, there is a pressing need for a high-quality, large-scale benchmark. Although recent efforts [9, 40] have made initial attempts, they often rely on manual curation or direct prompting, limiting data coverage, diversity, and scalability. Driven by recent developments in agentic AI, data agents [79] have emerged as a promising paradigm for autonomously handling complex data tasks. However, existing data agents primarily focus on managing [37], preparing [10, 11], and analyzing [22, 59] existing data, with limited attention to the autonomous generation of domain-specific data, such as NL–TSSQL pairs. Unlike general-purpose data generation, TSSQL data generation is subject to strict syntactic and semantic constraints, demanding higher data quality. Meanwhile, existing LLM- and agent-based Text-to-SQL data generation approaches [7, 25, 33, 36] lack sufficient autonomy for generating NL–TSSQL pairs, leaving several key challenges unresolved:

Challenge 1: Modeling Valid Compositions in TSSQL. TSSQL introduces a variety of specialized temporal syntax and numerous time-series functions. As shown in Fig. 1(b), directly taking the Cartesian product leads to a combinatorial explosion, while only a small subset is syntactically valid, semantically meaningful, and compatible with the schema instance. Existing manually constructed datasets [28, 35, 40] require experts or annotators, which are costly and difficult to scale, while automated approaches primarily rely on instance transformation [25, 52], template instantiation [7, 75], or instruction-guided synthesis [9, 33, 36]. Instance- and template-based methods constrain generation to predefined structural neighborhoods, whereas instruction-guided methods are highly sensitive to the quality of manually specified task requirements and constraints, leaving query validity largely dependent on LLM reasoning and post-hoc verification. Consequently, existing approaches lack systematic modeling of valid SQL compositions, limiting both coverage and validity.

Challenge 2: Autonomous Orchestration for TSSQL Diversity. Data generation is inherently dynamic: as the corpus evolves, the utility of new data changes with the knowledge and characteristics represented in the generated data. A key challenge is therefore to determine what is most valuable to explore among numerous valid candidates at each generation step to enhance data diversity. As shown in Fig. 1(c), existing agent-based approaches [15, 36, 58] introduce multi-agent collaboration and environmental interaction, yet their agents are typically embedded in predefined generation pipelines, with autonomy largely confined to executing or coarsely selecting among predefined procedures such as generation, expansion, and validation. Such coarse-grained autonomy lacks fine-grained control over the characteristics of generated data and cannot dynamically assess the marginal utility of potential data additions based on the evolving corpus. Consequently, these methods may repeatedly revisit well-covered regions while under-exploring more informative combinations, limiting the diversity and complexity of generated data.

Challenge 3: Efficiently Reusing Temporal Semantics. TSSQL generation instantiates recurring local temporal semantics across diverse database entities, creating a pronounced structure–instance asymmetry: similar semantic structures must be repeatedly reconstructed across heterogeneous data instances. However, the applicability of TSSQL structures depends not only on schema compatibility but also on data characteristics, such as sampling granularity, temporal distribution, and synchronization. As illustrated in Fig. 1(d), asynchronous sampling invalidates point-wise operations, non-uniform distribution undermines time-window aggregation, and continuous variation requires linear rather than previous filling. Existing approaches [18, 56] typically achieve reuse by extracting query-level structural SQL templates. For TSSQL, however, structural compatibility does not imply temporal applicability, while coarse SQL-level reuse fails to capture local temporal structures.

To comprehensively address the aforementioned challenges, we propose a novel data generation framework, **TSSDA**, and construct **TERN**, a large-scale Text-to-TSSQL benchmark. **To address the first challenge**, we systematically model TSSQL syntax and semantics to construct a data generation space. We decompose TSSQL into core syntactic modules with explicit semantic decision variables, and model their cross-module dependencies. The resulting space integrates schema information, dependency propagation, and evolving semantic states to provide valid decision paths for composing diverse TSSQL queries. **To address the second challenge**, we develop a *Data Agent* that autonomously navigates, explores, and orchestrates the data space. Specifically, we design dozens of Skills that provide the Agent with comprehensive autonomous capabilities. The Data Agent consists of two collaborative agents: a *Semantic Decision Agent* that dynamically plans Skill execution and makes semantic decisions, and a *Query Generation Agent* that incrementally materializes the temporal query. An online adaptation mechanism further enables the Agent to adapt to the evolving data distribution and accumulated experience, continuously improving generation quality and diversity. **To address the third challenge**, we propose a temporal-aware hierarchical memory for efficient reuse of generation experience. We abstract data characteristics and generation states into temporal profiles as memory indices. A hierarchical memory then retrieves historical semantic decisions and operator expressions through temporal profile matching and reliability-aware selection. An adaptive management mechanism further dynamically updates the accumulated experience.

In summary, we make the following contributions:

- We formalize the problem of automated Text-to-SQL data generation for TSDBs and propose a highly autonomous Data Agent framework for constructing high-quality data. (Sec. 2 and 3).
- We extract core syntax modules from TSSQL and model their semantic dependencies to construct an executable and explorable data generation space. (Sec. 4).
- We propose a Data Agent with carefully designed skills, exploration algorithms, and online adaptation mechanisms to autonomously explore and orchestrate the data space. (Sec. 5)
- We develop a temporal-aware memory with temporal pattern profiling and adaptive memory management to efficiently reuse local temporal semantics. (Sec. 6).
- We construct **TERN**, a large-scale benchmark built on 32 real-world industrial datasets and comprising 13,182 unique NL–TSSQL pairs. We conduct extensive evaluations on **TERN** and comprehensively compare **TSSDA** against baseline methods. (Sec. 7)

2 PRELIMINARIES

2.1 Problem Formulation

Basic Definitions. We first introduce the basic concepts related to time-series databases (TSDBs).

- *Time Series*: A time series is an ordered sequence of observations, each consisting of a timestamp and its corresponding value, where time serves as the primary dimension in TSDBs.
- *Schema*: A TSDB schema $\mathcal{S}$ describes the organization of time-series data under a specific data model, such as tree-based [66] or tag-based [1] models, including the associated entities, devices, measurements, and their structural relationships.
- *Temporal Operators*:Temporal operators O_T are query operators or functions applied to time series for various temporal analysis tasks. They mainly include time-series generating functions (TSGFs) for sequence transformation and aggregate functions (AGGs) for sequence aggregation.
- *TSSQL*: TSSQL refers to SQL-like query languages provided by TSDBs for querying and analyzing time-series data. A TSSQL query q specifies target time series, temporal semantics, and optionally temporal operators from O_T. Its execution under schema $\mathcal{S}$ is denoted as $\text{Exec}(q, \mathcal{S}) \rightarrow Y$, where Y is the result.

NL2TSSQL. Given a natural language question u and a time-series schema $\mathcal{S}$, the NL2TSSQL task aims to translate u into a TSSQL query q that correctly expresses the intended query semantics. Formally, the task can be expressed as $f_{\text{NL2TSSQL}} : (u, \mathcal{S}, O_T, \Gamma_T) \rightarrow q$, where Γ_T denotes the temporal grammar and syntactic constraints imposed by the target TSSQL dialect. The generated query q should conform to schema $\mathcal{S}$, the supported temporal operator space O_T, and the temporal grammar constraints Γ_T, while faithfully preserving the semantics specified by the natural language question u.

Data Generation. Given the schema $\mathcal{S}$ of a target TSDB instance, the data generation task aims to generate a set of NL2TSSQL instances $\mathcal{Z} = \{(u_i, q_i)\}_{i=1}^{N}$, where u_i is a natural language question and q_i is its corresponding TSSQL query. Formally, the task is expressed as $f_{\text{DG}} : (\mathcal{S}, O_T, \Gamma_T) \rightarrow \mathcal{Z}$. Each generated pair (u_i, q_i) should be semantically consistent, while q_i should conform to $\mathcal{S}$, O_T, and Γ_T, and be executable on the target TSDB.

2.2 Benchmark Definition

TERN Benchmark. TERN is an NL2TSSQL benchmark defined as $\mathcal{B} = (\mathbb{E}, \mathcal{Z}, \mathcal{R}, \Pi)$, consisting of the following components:

- *TSDB Environments*: $\mathbb{E} = \{(\mathcal{S}_j, O_{T,j}, \Gamma_{T,j})\}_{j=1}^{N}$, representing the TSDB environments, where N is the number of TSDB instances.
- *NL2TSSQL Instances*: $\mathcal{Z} = \bigcup_{j=1}^{N} \mathcal{Z}_j$, where $\mathcal{Z}_j = \{(u_{j,i}, q_{j,i})\}_{i=1}^{n_j}$ denotes the set of NL–TSSQL pairs associated with environment j, and each pair $(u_{j,i}, q_{j,i})$ satisfies the validity requirements.
- *Generation Specification*: $\mathcal{R}$ is an optional set of requirements that specifies the desired benchmark construction targets and enables data generation under customized objectives.
- *Evaluation Protocol*: Π specifies the data partitioning strategy and evaluation settings of the benchmark.

Beyond instance-level validity, a benchmark should additionally satisfy dataset-level requirements on *coverage*, *diversity*, and *quality* to enable comprehensive and reliable evaluation of NL2TSSQL systems.

3 FRAMEWORK OVERVIEW

In this section, we present the design of our framework, **TSSDA**. Unlike existing automated NL2SQL data generation approaches, which primarily rely on instance transformation [25, 52], template instantiation [7, 75], or predefined workflows [33, 36, 58], our method constructs an interactive data generation space by explicitly modeling syntax and semantic dependencies, and develops a Data Agent system that autonomously explores, orchestrates, and adapts this data infrastructure. As illustrated in Figure 2, we collect diverse industrial time-series datasets as TSDB instances. **TSSDA** comprises five key components, detailed below.

I. Semantic Dependency Modeling. Automated TSSQL generation should jointly ensure broad compositional coverage and strong validity. Existing approaches [7, 36, 58] largely rely on prompt-specified rules and post-hoc augmentation, leaving the definition of valid query compositions implicit and requiring costly filtering to remove invalid queries. To address this issue, we model the TSSQL generation process at a fine granularity from both syntactic and semantic perspectives. Specifically, we first extract nine core syntactic modules from the base TSSQL grammar. We then analyze temporal dependencies and constraints from temporal semantics, official documentation, and validity checks in the DBMS Analyzer to construct a dependency graph.

II. Data Generation Space. Towards true autonomy in data generation, we construct a structured generation space that enables the Data Agent to explore autonomously. Based on the dependency graph, we topologically order the nine core modules to progressively construct NL–SQL pairs. Then, a unified time-series meta-schema with queryable details is introduced to abstract heterogeneous schema models. To efficiently propagate dependency information across phases, we introduce lightweight Structured Phase Outputs (SPOs) and Flags as *Dependency Signals*. A serialized *Query Frame* is further used to maintain the intermediate query representation throughout generation. Finally, we define a *Global Context* and combine it with the above structures to form the space state, whose transitions are driven by Agent interactions.

III. Autonomous Agent Exploration. To enable autonomous fine-grained control over the generation process and exploration of the data space, we develop a Data Agent with orchestration, decision-making, and generation capabilities. At each exploration step, we first construct the generation state from the data space and the dependency information. Next, we design a Skill Repository comprising dozens of Skills that support diverse autonomous capabilities. A Semantic Decision Agent $\mathcal{M}_D$ then selects a Skill κ and leverages it to determine an action a. The action a drives exploration through semantic decisions and tool execution, while dynamically updating the set of eligible subsequent Skills. Based on the action, the Query Generation Agent $\mathcal{M}_G$ is activated as needed to generate a $\Delta Query$ for the corresponding query component under the current semantics. The Agent's operations constitute the Exploration Outcome, which is then used to update the space environment accordingly. The Data Agent continues exploration until all modules are completed, generating a batch of queries. In addition, we design a memory mechanism which is detailed later.

IV. Online Adaptation. Online adaptation enables the Agent to continuously learn and improve, as illustrated by the red arrows

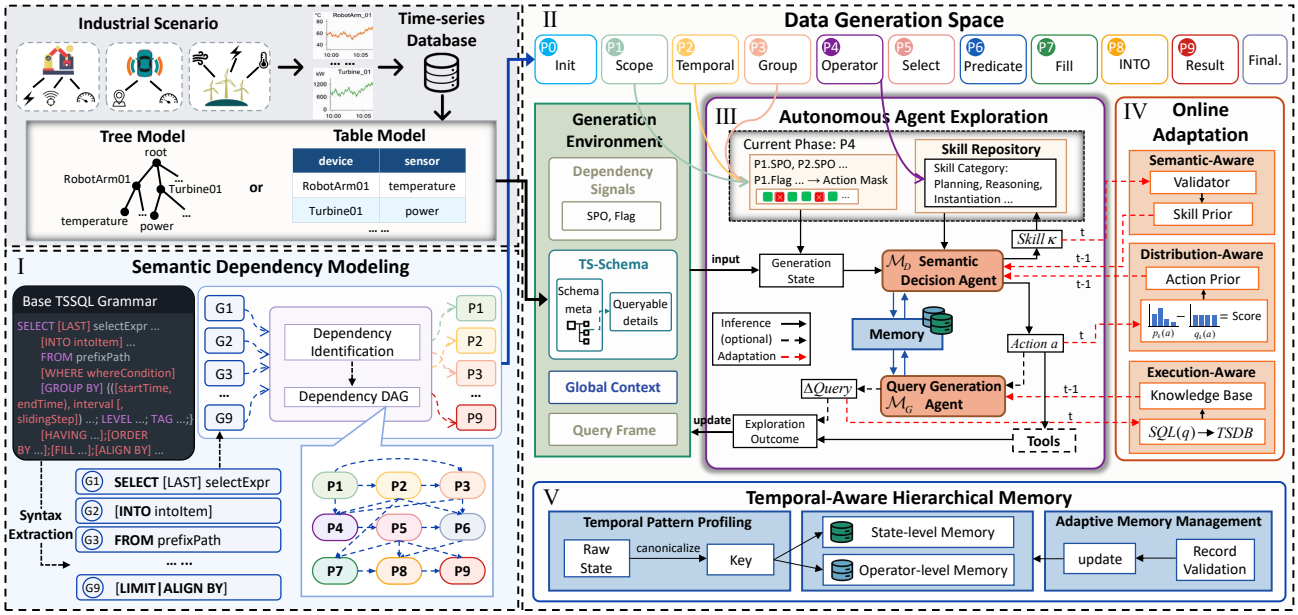


Figure 2: Architecture of the TSSDA Framework

in Figure 2. To construct a high-quality benchmark, we guide the Agent from multiple perspectives, including semantic validity, coverage and diversity, and executability. Therefore, we propose three training-free online adaptation mechanisms. The *Semantic-Aware Adaptation* uses rule-based and model-based validators to assess semantic consistency and guide Skill selection. The *Distribution-Aware Adaptation* provides decision guidance through action priors derived from the distribution of validated data. The *Execution-Aware Adaptation* leverages TSDB execution feedback to refine query generation. At step t, the Data Agent leverages adaptation knowledge accumulated up to step $t-1$, while the knowledge generated at step t is used to guide subsequent exploration.

V. Temporal-Aware Hierarchical Memory. TSSQL data inherently exhibit high structural repetition but low instance-level overlap. To address this efficiency challenge, we extract temporal patterns from the raw state for efficient storage and retrieval. A hierarchical memory is then introduced, with two levels tailored to the decision process and fine-grained nested operator-chain generation, respectively. Before Agent activation, the system retrieves memory using a matching algorithm. A successful hit bypasses the computationally intensive Agent invocation and directly returns the result. Finally, we design an Adaptive Memory Management mechanism to dynamically update the memory. This memory substantially improves system efficiency while achieving a favorable trade-off between efficiency and generation performance.

4 DEPENDENCY-AWARE GENERATION SPACE

4.1 Dependency Modeling

Given an industrial TSDB schema $\mathcal{S}$, comprehensive TSSQL generation requires systematically exploring a vast query space spanning diverse temporal intents. Existing LLM-based approaches [7, 9, 33] typically rely on seed examples, templates, or manually crafted instructions to guide generation. However, both paradigms remain anchored to human-specified starting points and rely on loosely constrained generation for expansion. Consequently, they lack an explicit data-centric modeling of the query space, limiting data coverage. Therefore, we systematically extract the syntactic and semantic constraints of TSSQL and construct a dependency directed acyclic graph (DAG) to explicitly model the implicit relationships among query components.

First, we model the TSSQL syntax space to enable valid generation and comprehensive exploration. An intuitive approach is to construct the syntax based on the Abstract Syntax Tree (AST). However, a complete AST is overly complex, introducing substantial overhead and limiting practical feasibility. Prior work [17, 69] explored SQL generation with lightweight customized ASTs, but these methods largely predate the LLM era, as tree-structured ASTs are poorly suited to serialized generation. To this end, we analyze the formal TSSQL grammar by traversing its production rules from the query-statement production and grouping related non-terminals according to the query semantics they govern. Productions describing the same query component are consolidated, while auxiliary lexical rules are omitted. This process yields nine core syntactic modules $V = \{\mu_{\text{select}}, \mu_{\text{from}}, ..., \mu_{\text{result}}\}$, which capture the major structural dimensions of TSSQL queries.

With these syntactic modules, we can comprehensively model all components of TSSQL. However, organizing them remains challenging, as syntactic order does not necessarily align with semantic reasoning. For example, although *SELECT* appears first syntactically in a SQL statement, semantically its valid functions and target time series can only be determined after the query scope and grouping structure are established. We therefore first identify the semantic task of each syntactic module, including its query decision and corresponding choices. Specifically, for each module $\mu_i \in V$, we introduce a set of semantic decision variables $X_i = \{x_{i,1}, x_{i,2}, \ldots, x_{i,m_i}\}$,

where each variable $x_{i,k}$ is associated with a candidate action space $\mathcal{A}_{i,k}$. We then define a dependency constraint $r(x_{i,k}, x_{j,l})$ to exist whenever the resolved decision of $x_{i,k}$ in module μ_i affects the decision of $x_{j,l}$ in another module μ_j. If $r(x_{i,k}, x_{j,l})$ explicitly prohibits an action a of $x_{j,l}$, we formulate a binary validity function $\phi_r(x_{i,k}, a) \in \{0, 1\}$ and include $x_{i,k}$ in $\mathbf{x}^r_{j,l}$. The feasible action space of $x_{j,l}$ is then defined as:

$$\mathcal{A}^r_{j,l}(\mathbf{x}^r_{j,l}) = \left\{ a \in \mathcal{A}_{j,l} \mid \phi_r(\mathbf{x}^r_{j,l}, a) = 1 \right\}, \tag{1}$$

where $\mathbf{x}^r_{j,l}$ denotes the preceding decision variables that restrict the action space of $x_{j,l}$, and $\phi_r = 1$ indicates that the action is valid. Based on these dependency constraints, we define a dependency from module μ_i to μ_j whenever $r(x_{i,k}, x_{j,l})$ exists, denoted by:

$$E_{ij} = \left\{ r \,\middle|\, \exists\, x_{i,k} \in \mathbf{x}^r_{j,l},\ x_{j,l} \in X_j \right\}. \tag{2}$$

Accordingly, we construct a module-level dependency DAG $\mathcal{G} = (V, E)$ by treating the syntactic modules in V as vertices and adding a directed edge (μ_i, μ_j) whenever $E_{ij} \neq \emptyset$.

The constraints r are derived from three complementary sources. We first analyze the temporal semantics of TSSQL to identify ordering and compatibility constraints among temporal decisions. We then extract explicit cross-module restrictions from the official documentation, and further inspect the semantic validity checks implemented in the DBMS Analyzer to recover constraints enforced by the DBMS. The extracted rules are normalized into the predicate form defined above and collected as $R_c = R_T \cup R_D \cup R_A$. We further validate and consolidate the candidate constraints R_c, yielding the final set R used to construct the module-level dependencies. The resulting dependency graph is shown in Figure 2.

4.2 Generation Space Construction

Based on the dependency graph, we further model the generation process as a data space, providing the Data Agent with a perceivable and explorable environment. Specifically, given a TSDB schema $\mathcal{S}$, we construct a structured generation space, denoted by $\Omega(\mathcal{S}) = \left(\mathcal{P}, \widehat{\mathcal{S}}, \Psi, \mathcal{F}, \Theta, \mathcal{Q}, \delta\right)$. Next, we describe each component in detail.

Generation Phases $\mathcal{P}$: Given the dependency DAG $\mathcal{G} = (V, E)$, we topologically order the syntactic modules as $\pi = \text{TopoSort}(\mathcal{G})$. With initialization and finalization phases, we define the space backbone as $\mathcal{P} = (P_0, P_{\pi_1}, P_{\pi_2}, \ldots, P_{\pi_9}, P_f)$, where each P_{π_i} corresponds to the syntactic module μ_{π_i}. Each phase P_i contains its decision variables X_i and action spaces $\{\mathcal{A}_{i,k}\}_{k=1}^{m_i}$ for agent interaction.

Unified TS-Schema $\widehat{\mathcal{S}}$: To accommodate large-scale and heterogeneous TSDB schemas, we normalize the raw schema into a unified representation $\widehat{\mathcal{S}} = (\mathcal{S}^M, \mathcal{S}^D)$, where $\mathcal{S}^M$ provides a concise schema-level overview, while $\mathcal{S}^D$ maintains fine-grained schema details that can be efficiently retrieved on demand.

Structured Phase Output (SPO) Ψ: SPOs provide lightweight propagation of cross-phase dependencies. For an edge $(\mu_i, \mu_j) \in E$, we extract the associated dependencies $r(x_{i,k}, x_{j,l})$. If the decision result of $x_{i,k}$ semantically affects $x_{j,l}$, we record this result in the SPO Ψ. For each phase P_j, its input SPO is aggregated from the predecessor phases: $\Psi^{\text{in}}_j = \bigoplus_{P_i \in \text{Pred}(P_j)} \Pi_{ij}(\Psi_i)$, Π_{ij} projects the dependency-relevant information from Ψ_i for P_j, and $\oplus$ denotes aggregation over multiple predecessors.

Dependency Flags $\mathcal{F}$: Flags encode explicit cross-phase restrictions. Specifically, if $r(x_{i,k}, x_{j,l})$ induces a validity function $\phi_r(x_{i,k}, a)$, the corresponding restriction is recorded in $\mathcal{F}$.

Global Context Θ: Θ maintains global context and shared system states, such as temporal information and runtime settings.

Query Frame $\mathcal{Q}$: The Query Frame maintains a serialized intermediate representation of the partially constructed query, denoted as $Q = \text{Serialize}\left(\{X_i\}_{i=1}^{9}, \mathcal{O}_T\right)$, which records the generated results of semantic decision variables X_i together with the intermediate expressions constructed from the operator space $\mathcal{O}_T$.

Transition δ: At step t, given a resolved action $a_t \in \mathcal{A}_{i,k}$ for decision variable $x_{i,k}$, we define the transition as $\Omega_{t+1} = \delta(\Omega_t, a_t)$, where the relevant structures in Ω_t are updated accordingly.

5 DATA AGENT

5.1 Skill-Orchestrated Agent Architecture

The generation space defines the boundary and structure of data construction, while raising a further challenge: how to effectively explore this space? Existing approaches [7, 33, 36] typically use predefined workflows that assign LLMs or agents fixed roles, such as generation, augmentation, or verification. The generation trajectory is determined by the framework, limiting autonomous decision-making and exploration. To address this challenge and endow the Agent with data autonomy beyond monolithic prompting, we abstract dozens of fine-grained capabilities into Skills organized in phase-specific repositories. Specifically, we define two primary types of Skills: decision Skills and generation Skills. A decision Skill is represented as $\kappa^D = \left\langle x_{i,k}, \mathcal{A}_{i,k}, \rho_D \right\rangle$, where $x_{i,k}$ is the target semantic decision variable, $\mathcal{A}_{i,k}$ is action space, and ρ_D denotes the Skill execution policy. Given the current generation state Ω, the decision Skill κ^D produces an action $a = \rho_D\left(\Omega, x_{i,k}, \mathcal{A}^r_{i,k}\right), a \in \mathcal{A}^r_{i,k}$. Based on a resolved action a, the generation Skill is denoted as $\kappa^G = \langle a, \rho_G \rangle$, where ρ_G produces an incremental intermediate representation $\Delta Q = \rho_G(\Omega, a)$. These Skills provide complementary capabilities, such as planning, reasoning, and instantiation, to support autonomous generation. Finally, each phase P_i is associated with a dedicated Skill repository $\mathcal{K}_i = \left(\kappa^D_1, \ldots, \kappa^D_m, \kappa^G_1, \ldots, \kappa^G_n\right)$, containing Skills specialized for that phase.

Next, we decouple data exploration into two specialized agents: the *Semantic Decision Agent* $\mathcal{M}_D$ and the *SQL Generation Agent* $\mathcal{M}_G$. The former autonomously navigates the data generation space through semantic reasoning, while the latter incrementally constructs queries based on these decisions. Specifically, $\mathcal{M}_D$ makes semantic decisions such as selecting temporal patterns and grouping paradigms, guided by semantic context and decision Skill specifications. In turn, $\mathcal{M}_G$ constructs and updates the intermediate query representation by incrementally instantiating concrete query elements according to generation Skill specifications. The collaboration between the two agents prevents syntactic generation from directly determining high-level query semantics and enables semantic decisions to be explicitly maintained and constrained.

5.2 Autonomous Agent Exploration

In this section, we detail how the Data Agent autonomously analyzes and explores the data space. Starting from the initialization

Algorithm 1: Compositional Operator-Chain Construction

Input : state Ω'_t; raw time series TS; Agents $\mathcal{M}_D$, $\mathcal{M}_G$; Skills: {SeriesExpr Extension κ_1^D; AggExpr Extension κ_2^D; AGG Selection κ_3^D; TSGF Selection κ_4^D; Operator Selection κ_5^D; Expr Instantiation κ_1^G}

Output : SeriesExpr set C_S; AggExpr set C_A.

1 $C_S \leftarrow TS, \quad C_A \leftarrow \varnothing$;
2 $r_t \leftarrow \mathcal{M}_D(\Omega'_t, \{\kappa_1^D, \kappa_2^D, \kappa_1^D \rightarrow \kappa_2^D\})$;
3 **if** $r_t \in \{\kappa_1^D, \kappa_1^D \rightarrow \kappa_2^D\}$ **then**
4 **while** *SeriesExpr Construction is not stopped* **do**
5 $a_t \leftarrow \mathcal{M}_D(\Omega'_t, \kappa_1^D)$;
6 $\kappa^D \leftarrow \kappa_4^D$ if TSGF, otherwise κ_5^D according to a_t;
7 $(\Omega'_t, C_S) \leftarrow Exploration(\Omega'_t, \mathcal{M}_D, \mathcal{M}_G, \kappa^D, \kappa_1^G, C_S)$;
8 **if** $r_t \in \{\kappa_2^D, \kappa_1^D \rightarrow \kappa_2^D\}$ **then**
9 $(\Omega'_t, C_A) \leftarrow Exploration(\Omega'_t, \mathcal{M}_D, \mathcal{M}_G, \kappa_3^D, \kappa_1^G, C_A)$;
10 **while** *AggExpr Construction is not stopped* **do**
11 $a_t \leftarrow \mathcal{M}_D(\Omega'_t, \kappa_2^D)$;
12 $\kappa^D \leftarrow \kappa_4^D$ if TSGF, otherwise κ_5^D according to a_t;
13 $(\Omega'_t, C_A) \leftarrow Exploration(\Omega'_t, \mathcal{M}_D, \mathcal{M}_G, \kappa^D, \kappa_1^G, C_A)$;
14 **return** C_S, C_A;

Function: Exploration

Input : $\Omega'_t, \mathcal{M}_D, \mathcal{M}_G, \kappa^D, \kappa^G, C$

1 $f_t \leftarrow \mathcal{M}_D(\Omega'_t, \kappa^D), \Omega'_t \leftarrow \text{Update}(\Omega'_t, f_t)$;
2 $\Delta Q_t \leftarrow \mathcal{M}_G(\Omega'_t, \kappa^G)$;
3 $C \leftarrow C \cup \{\Delta Q_t\}$;
4 $\Omega'_t \leftarrow \text{WriteBack}(\Omega'_t, \Delta Q_t)$;
5 **return** (Ω'_t, C);

phase P_0, the Agent iteratively explores each phase until all phases are completed. We next describe the exploration process.

State Preparation: Given the current space state Ω_t, we construct the input state Ω'_t by extracting the phase-relevant Θ, $\mathcal{Q}$, and $\widehat{\mathcal{S}}$. Next, according to the dependency DAG $\mathcal{G} = (V, E)$, we retrieve the SPOs Ψ and Flags $\mathcal{F}$. For example, phase P_4 obtains dependency from its related phases P_1, P_2, and P_3, as illustrated in Figure 2. The relevant SPOs Ψ are incorporated into Ω'_t to provide the semantic dependency context, while the Flags $\mathcal{F}$ impose action masks on the action space of the available Skills.

Dynamic Skill Routing: After preparing the input state, we derive the set of currently eligible candidate Skills $\mathcal{K}'_{i,t}$ from the phase-specific Skill repository. Rather than following a predetermined Skill sequence, the *Semantic Decision Agent* $\mathcal{M}_D$ dynamically selects a Skill from the candidate set based on the current semantics and the input state Ω'_t: $\kappa_t^D = \mathcal{M}_D\left(\Omega'_t, \mathcal{K}'_{i,t}\right)$. The Skill execution result further updates the set of eligible subsequent Skills.

Agent Exploration: Our agent progressively explores the data space using Skills. Specifically, each decision Skill defines the currently admissible exploration actions $\mathcal{A}^r_{i,k}$. The *Semantic Decision Agent* $\mathcal{M}_D$ then selects an action $a_t = \mathcal{M}_D(\Omega'_t, \kappa_t^D)$ according to the current input state and the Skill policy ρ_D. Each action advances the exploration and refines the query semantics. Execution actions invoke the corresponding tools, while generation actions activate the *SQL Generation Agent* $\mathcal{M}_G$ to instantiate the current semantics.

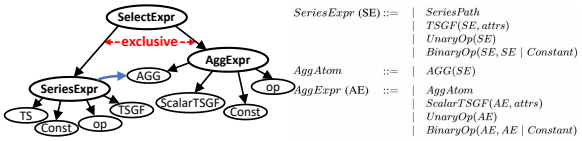


Figure 3: Recursive Operator Expression Structure

When invoked, the action a_t provides $\mathcal{M}_G$ with the generation Skill κ_t^G required for the current step. The agent then produces the corresponding incremental query: $\Delta Q_t = \mathcal{M}_G(\Omega'_t, \kappa_t^G)$. Additionally, before either agent is activated, a memory mechanism is accessed to directly obtain the required result when available. Further details are provided in Sec. 6.

Outcome Update: The decision a_t, its execution result, and the generated ΔQ_t are recorded in the *Exploration Outcome* and subsequently used to update the corresponding global context, query frame, and dependency-related variables in the data space.

Operator-Chain Exploration Algorithm: TSDBs support operator chains over hundreds of temporal functions. To explore their semantics and combinations, we develop an iterative chain-construction algorithm based on the general exploration procedure.

Expressions in TSDBs include time-series generating functions (*TSGFs*) and aggregate functions (*AGGs*). The former produce timestamp-indexed data points, while the latter collapse points within a temporal segment to a single value. Due to their distinct computation granularity and cardinality, *SelectExpr* has two mutually exclusive forms: *SeriesExpr* and *AggExpr*. Additionally, the two expression types can be nested when the semantic and syntactic constraints are satisfied. Therefore, to automatically construct valid operator chains, we formulate a recursive expression grammar, as illustrated in Figure 3. A *SeriesExpr* is initialized from raw time series *TS* and recursively extended with *TSGFs*, constants (*Const*), and operators (*op*). We then define *AggAtom*, which applies an *AGG* to an arbitrary *SeriesExpr*. An *AggExpr* is built from *AggAtom* and can be recursively extended with *Const*, *op*, and scalar-compatible *TSGFs*, excluding those that accept only sequence inputs.

As shown in Algorithm 1, we use operator-chain construction as an example to illustrate the exploration algorithm. We first initialize C_S with raw time series and C_A as an empty set. Next, $\mathcal{M}_D$ performs Skill routing to determine the generation trajectory. If $\mathcal{M}_D$ selects κ_1^D, we iteratively construct the *SeriesExpr* (lines 3-7 of Algorithm 1). Specifically, $\mathcal{M}_D$ iteratively invokes κ_1^D to select the next *SeriesExpr* extension action according to the grammar, then invokes κ_4^D or κ_5^D based on the selected action. Following this, we execute the **Exploration** procedure, where $\mathcal{M}_D$ and $\mathcal{M}_G$ jointly perform one instantiation step for C_S. Then, if κ_2^D is selected, we first construct an *AggAtom* using κ_3^D and subsequently extend the *AggExpr* iteratively in a manner analogous to *SeriesExpr* (lines 8-13 of Algorithm 1).

Exploration: within each exploration step, $\mathcal{M}_D$ first invokes the input decision Skill κ^D to select a function or operator, including decisions on the function source, functional family, and concrete function. Subsequently, after incorporating the decision result into Ω'_t, $\mathcal{M}_G$ invokes κ^G to instantiate an expression as the incremental query ΔQ_t. The resulting expression is then added to the corresponding set C, and the relevant state in Ω'_t is updated accordingly.

5.3 Online Adaptation

Although the Data Agent can autonomously explore the data space, it still lacks the capacity for autonomous learning. High-quality TSSQL generation requires broad coverage, diversity, validity, and semantic plausibility. To enable efficient self-improvement of the Data Agent across these dimensions, we develop a training-free online adaptation mechanism that leverages exploration feedback through three complementary strategies, as detailed below.

Distribution-aware adaptation: Coverage and diversity are key dimensions of data generation quality. Existing approaches [2, 7, 36] typically improve diversity through offline augmentation by transforming query skeletons, templates, or existing queries. In contrast, we introduce online distribution-aware adaptation guided by the evolving data, enabling the Agent to autonomously improve its decision-making and explore underrepresented regions. For each semantic decision Skill κ, we track $N_{\kappa,t}(a)$, the number of valid instances generated by each action $a \in \mathcal{A}_\kappa$. The empirical action distribution at step t is:

$$p_{\kappa,t}(a) = \frac{N_{\kappa,t}(a) + \epsilon}{\sum_{a' \in \mathcal{A}_\kappa} \left(N_{\kappa,t}(a') + \epsilon\right)}, \tag{3}$$

where ϵ is a smoothing factor. Given a target distribution $p'_\kappa(a)$, which may be derived from real-world scenarios or user preferences, we quantify the distribution deficit of each action as $D_{\kappa,t}(a) = \frac{[p'_\kappa(a) - p_{\kappa,t}(a)]_+}{p'_\kappa(a)}$, where $[x]_+ = \max(x, 0)$. Larger deficits indicate greater underrepresentation in the generated data. We derive an action prior over the current feasible action set $\mathcal{A}^r_{\kappa,t}$ from these deficits and provide it to $\mathcal{M}_D$ as online distributional knowledge:

$$\varrho_{\kappa,t}(a) = \frac{\exp(D_{\kappa,t}(a)/\tau)}{\sum_{a' \in \mathcal{A}^r_{\kappa,t}} \exp(D_{\kappa,t}(a')/\tau)}, \tag{4}$$

where τ controls the strength of distribution-aware guidance. The action prior guides the Agent toward underexplored decisions while preserving semantic validity.

Execution-aware adaptation: Despite our fine-grained modeling of TSSQL syntax and semantics, the inherent uncertainty of LLMs may still cause generated queries to violate DBMS-specific constraints. Therefore, we design an execution-aware adaptation mechanism that continuously distills DBMS execution feedback into reusable generation knowledge, enabling $\mathcal{M}_G$ to progressively improve generation quality. Given the complete query q_t generated by the Agent, we execute it on the database $\text{Exec}(q_t, \mathcal{S}) \to (y_t, e_t)$, where y_t denotes the execution result and e_t captures the DBMS feedback, such as the status code and error message. $\mathcal{M}_G$ then repairs the query using execution feedback and historical experience, $q'_t = \mathcal{M}_G(q_t, e_t, \mathcal{H}_t)$, where $\mathcal{H}_t$ denotes experience in the execution knowledge base. After repair, we distill the repair experience into a reusable rule $h_t = \text{Distill}(e_t, q_t, q't)$ and add it to the knowledge base: $\mathcal{H}t + 1 = \mathcal{H}_t \cup h_t$. For subsequent generation, we incorporate relevant knowledge retrieved from $\mathcal{H}_t$ based on the current query structure and operator signatures into the generation context. If execution fails, we retrieve additional experience based on the error category to guide repair.

Semantic-aware adaptation: Although the dependency mechanism improves decision-making, it offers limited guidance for Skill routing. We therefore introduce semantic-aware adaptation to learn the state-dependent Skill suitability from online semantic feedback. A rule-based validator checks local semantic consistency at each routing step, while an LLM-based validator assesses overall Skill route consistency after generation. Let $v^{\text{r}}_t, v^{\text{LLM}}_t \in \{0, 1\}$ denote the two validation results, we define the semantic reward as $r_t = v^{\text{r}}_t \cdot v^{\text{LLM}}_t$. We then compress Ω'_t into a lightweight semantic context c_t and track the selection count $N_t(c_t, \kappa)$ and cumulative semantic reward $R_t(c_t, \kappa) = \sum r_t$. A routing score is computed as:

$$S_t(\kappa) = \frac{R_t(c_t, \kappa) + 1}{N_t(c_t, \kappa) + 2} + \lambda \sqrt{\frac{\log(N_t(c_t) + 2)}{N_t(c_t, \kappa) + 1}}, \tag{5}$$

where $N_t(c_t) = \sum_\kappa N_t(c_t, \kappa)$ and λ controls the exploration strength. The resulting scores are incorporated into the Agent context as online routing knowledge to guide subsequent Skill selection.

6 TEMPORAL-AWARE MEMORY

Temporal queries essentially instantiate recurring temporal semantics over heterogeneous schema entities. The same local temporal semantic structure may appear across industrial scenarios, while their schemas and temporal characteristics, such as sampling granularity, can differ substantially. This gives rise to a structure-instance asymmetry between reusable temporal structures and heterogeneous data instances. However, the reuse of these structures depends not only on schema compatibility but also on the temporal characteristics and semantics of the underlying data. To address this challenge, we propose a temporal-aware hierarchical memory that efficiently models temporal characteristics and enables the reuse of generation experience across instances. As illustrated in Figure 4, we next detail the memory mechanism.

Temporal Pattern Profiling: To identify reusable experience, we need to extract temporal characteristics from the data space. The raw state Ω_t is unsuitable for direct matching due to substantial redundancy. We therefore extract a temporal profile to capture the current data characteristics and temporal semantics of the data space. Specifically, we first canonicalize Ω_t as $\mathcal{I}_C = \text{Canon}(\Omega_t)$, where Canon prunes the dependency information Ψ, $\mathcal{F}$, and query representation $\boldsymbol{Q}$, and maps their state values into matchable ordered sequences, sets, and atomic values. Next, we design temporal profiling to extract a low-dimensional, instance-agnostic representation of the temporal context. For the m time series across n devices in the current space state, let $I_i = [l_i, r_i]$ denote the valid interval of the i-th series, $D = [L, R]$ the overall data interval with $L = \min_i l_i$ and $R = \max_i r_i$, $Q = [a, b]$ the query range, and (α, β) the window parameters. We compute:

$$\mathcal{I}_T = \left\langle \sigma_1(n, m, \{I_i\}), \sigma_2(R - L), \sigma_3\left(\frac{|Q|}{|D|}\right), \ldots, \sigma_k(\alpha, \beta) \right\rangle, \tag{6}$$

which characterize temporal patterns, data span, query coverage, window configuration, and other temporal properties. Each $\sigma_i(\cdot)$ discretizes a temporal characteristic into a compact representation, abstracting instance-specific values while retaining temporal information relevant to experience reuse. For expression-related states, we additionally extract an expression profile $\mathcal{I}_E$, which captures structural properties of the current expression, such as its operator composition and input/output characteristics. Finally, we combine the above features into a unified temporal profile $\mathcal{I} = \langle \mathcal{I}_C, \mathcal{I}_T, \mathcal{I}_E \rangle$, which serves as the key for subsequent memory retrieval.

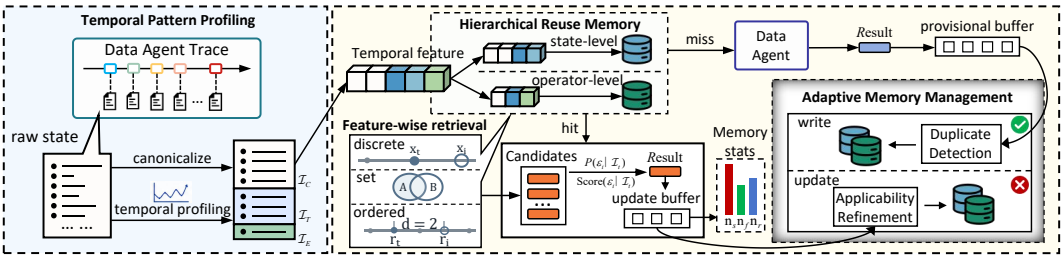


Figure 4: Overview of the Temporal-Aware Hierarchical Memory Mechanism

Hierarchical Reuse Memory: Reusable decision and generation experiences are associated with distinct temporal characteristics. We therefore design a hierarchical memory architecture consisting of *state-level memory* and *operator-level memory*, which store semantic decision results for $\mathcal{M}_D$ and operator expressions for $\mathcal{M}_G$, respectively. The current temporal profile $\mathcal{I}_t$ is used to retrieve the memory. A valid match enables direct result reuse without Agent invocation; otherwise, the Agent is invoked, and its experience may be stored after validation. Memory matching is based on a similarity function $\mathrm{Sim}(\mathcal{I}_t, \mathcal{I}_i)$ that compares the current temporal profile with each stored memory entry. Categorical features use exact or neighboring-category matching, set-valued features use set similarity, and ordered features use distance-based soft matching. Entries above the similarity threshold form the retrieved candidate set $\mathcal{H}_t$. For each memory entry ε_i, we maintain its successful reuse count N_i^s, failed reuse count N_i^f, and total reuse count N_i^r. Then, its confidence is estimated as $\mathrm{Conf}(\varepsilon_i) = \frac{N_i^s+1}{N_i^s+N_i^f+2}$. When multiple entries are matched, we score them by effectiveness and frequency:

$$\mathrm{Score}(\varepsilon_i \mid \mathcal{I}_t) = \frac{\mathrm{Sim}(\mathcal{I}_t, \mathcal{I}_i) \cdot \mathrm{Conf}(\varepsilon_i)}{1 + \lambda N_i^r}, \tag{7}$$

where λ controls the penalty on repeatedly reused entries. Based on the retrieval scores, the system retains the top-K entries from the retrieved candidate set $\mathcal{H}_t$ and constructs a normalized selection distribution:

$$P(\varepsilon_i \mid \mathcal{I}_t) = \frac{\mathrm{Score}(\varepsilon_i \mid \mathcal{I}_t)}{\sum_{\varepsilon_j \in \mathrm{TopK}(\mathcal{H}_t)} \mathrm{Score}(\varepsilon_j \mid \mathcal{I}_t)}, \tag{8}$$

The system then samples one memory entry from this distribution as the retrieval result.

Adaptive Memory Management: To maintain the reliability and applicability of the reuse memory, we dynamically manage memory entries through admission, update, refinement, and eviction. New experiences produced by the Data Agent are first stored in a provisional buffer and admitted into memory only after validation and redundancy checking. When a memory entry is retrieved, its reuse outcome is temporarily recorded in an update buffer; after validation, the corresponding success, failure, and reuse statistics are updated. If reuse fails due to temporal incompatibility, we refine the entry's applicability as $\mathcal{I}_i \leftarrow (\mathcal{I}_i, \mathcal{I}_f)$, where $\mathcal{I}_f$ denotes the failed reuse profile. Finally, an entry ε_i is removed when its failure ratio exceeds a threshold $\frac{N_i^f}{N_i^s+N_i^f} > \gamma$. This process allows the memory to continuously absorb validated experience, refine reuse conditions, and discard unreliable entries during generation.

7 EXPERIMENTS

7.1 Experimental Setup

Datasets. For *TSDB* data, we collect 32 large-scale time-series datasets from real-world industrial scenarios and construct 32 *TSDB* instances. The databases average 92.19 devices and 2,313.34 measurements, with respective maxima of 963 and 16,920. On average, each database contains 4.5 million data points spanning 797.73 days.

Baseline methods. We evaluate the TERN benchmark and our TSSDA method with two categories of baselines.

(1) **NL2SQL Baselines.** We extensively evaluate TERN with fine-tuned LLMs, frontier LLMs, and mainstream NL2SQL methods:

- **Fine-tuned LLMs**: To assess transferability to TSSQL and ensure comparability, we perform supervised fine-tuning (SFT) on representative open-source LLMs, including Meta-Llama-3.1-8B [60], GLM-4-9B [62], the Qwen3 family [61] (Qwen3-8B, Qwen3-30B-A3B, and Qwen3.8-27B), and code model Qwen2.5-Coder-7B [21].
- **Frontier LLMs**: We evaluate representative proprietary LLM families on Text-to-TSSQL, including GPT (GPT-5.6-sol [44] and GPT-6.1-sol [43]), Qwen [61] (Qwen3.7-Max, Qwen3.8-Max, and Qwen3-Coder-480B), and DeepSeek-V4.1-Flash [6].
- **NL2SQL Methods**: We evaluate mainstream NL2SQL methods, including fine-tuned approaches such as CodeS [34], OmniSQL [33], and XiYan-SQL [38], as well as proprietary LLM-based methods such as DAIL-SQL [14], OpenSearch-SQL [71], DeepEye-SQL [31], and NL2InfluxDB [9].

(2) **NL-SQL Generation Baselines.** We compare TSSDA with:

- **DeepSeek** [6]: We use DeepSeek-V4.1-Flash as a reference baseline, augmented with RAG [30] over a TSSQL knowledge base.
- **OmniSQL** [33]: This method proposes a four-stage LLM-driven data synthesis pipeline and employs semantic consistency selection and majority voting to select high-quality data.
- **SQL-Factory** [36]: This method proposes a multi-agent SQL generation framework with three collaborative teams for generating initial queries, expanding seed queries, and validation.
- **SemanticAgent** [15]: This method constructs a structured semantic knowledge base to guide data synthesis and employs knowledge-grounded diagnosis and iterative refinement.
- **TACO** [7]: This work proposes a three-stage data synthesis pipeline that employs grammar-based skeleton generation, schema-guided query instantiation, and CoT-based NL generation.
- **NL2InfluxDB** [9]: This work proposes a documentation-driven data synthesis pipeline for InfluxDB that integrates retrieval-augmented generation and knowledge-graph-based verification.

Table 1: Statistics of different NL2SQL datasets.

Method	# Example	# DB	# Table / DB	# Col / DB	# Tok. / SQL	# Unique Func.	Build Mode	TSDB -native	Time-series Analytics
WikiSQL [76]	80,654	26,521	1.00	6.34	12.2	5	Hybrid	✗	✗
Spider 1.0 [73]	10,181	200	5.11	26.82	18.5	5	Manual	✗	✗
ScienceBenchmark [75]	5,031	3	16.67	86.67	15.6	5	Hybrid	✗	✗
BIRD [35]	12,751	95	7.64	54.56	30.9	24	Manual	✗	✗
Spider 2.0-Snow [28]	547	152	52.63	812.1	161.8	106	Manual	✗	✗
BIRD-History [77]	1,393	11	6.8	72.5	80.9	22	Automatic	✗	✗
TACO-Beijing [7]	6,705	24	92.1	1458	14.6	9	Automatic	✗	✗
NL2InfluxDB [9]	1,106	1	-	-	18	-	Automatic	✓	✓
TERN	13,182	32	92.19	2313.34	38.08	127	Automatic	✓	✓

Benchmark Configuration. To enable more fine-grained evaluation on TERN, we further organize the benchmark into three versions. TERN-Foundation covers basic TSSQL knowledge with relatively short, simple queries. TERN-Standard, the largest subset, targets moderately difficult NL2TSSQL tasks. TERN-Analytical focuses on time-series analytical queries with more complex temporal operator chains. Each version is split into training, validation, and test sets at an 8:1:1 ratio. All collected benchmark instances are validated through LLM-based and manual verification.

Evaluation Metrics. For TERN, we adopt the widely used [7, 15, 35] *Execution Accuracy* (EX) and *Successful Execution Rate* (SER). EX measures the proportion of predictions matching ground-truth execution results, while SER measures the proportion that execute successfully. For TSSDA, we evaluate the coverage, diversity, and complexity. Following SQL-Factory [36], we compute *Hybrid Similarity* and the *Vendi Scores* to evaluate sample similarity. The *TSSQL Coverage* metric builds on SQL coverage criteria from prior work [57], covering temporal, SQL-structural, and schema dimensions. The *Hill diversity* metric further measures TSSQL diversity, defined as $\mathrm{HD} = \exp\left(-\sum_{i=1}^{K} p_i \ln p_i\right)$, where p_i is the frequency of the i-th TSSQL feature category. Finally, following the SQL adaptation [63] of Halstead measures [19], we define *Halstead Complexity* as the geometric mean of the Halstead measure and expression depth. The Halstead measure is computed as $\mathrm{H}(q) = \frac{n_1}{2} \cdot \frac{N_2}{n_2} \cdot \log_2(n_1 + n_2)$, where n_1, n_2, and N_2 denote the numbers of distinct operators, distinct operands, and total operand occurrences.

Experimental Settings. All experiments are implemented with PyTorch 2.6 and Python 3.11 on Ubuntu 26.04. Methods requiring training are trained using 4 NVIDIA A100 GPUs.

7.2 Benchmark Statistics

Table 1 compares TERN with representative NL2SQL benchmarks, including established benchmarks such as WikiSQL [76], Spider [73], ScienceBenchmark [75], and BIRD [35], as well as recent challenging benchmarks, including Spider 2.0-Snow [28], BIRD-History [77], and TACO-Beijing [7]. We also include NL2InfluxDB [9], a recent attempt at NL2SQL for time-series databases. However, its benchmark data are not publicly available, preventing complete statistics.

Our benchmark TERN contains 13,182 examples collected from 32 real-world industrial TSDBs. The schema scale is comparable to recent complex benchmarks such as Spider 2.0-Snow and TACO-Beijing. In particular, the industrial TSDBs contain numerous measurements, averaging 2,313.34 columns per database and up to 16,920. The SQL length is computed following Spider 2.0 [28]. Since TERN is not constructed to contain only highly complex queries, its SQL length is lower than that of benchmarks specifically targeting complex SQL, such as Spider 2.0-Snow and BIRD-History. Nevertheless, rich temporal syntax and complex expressions make its queries longer than those in other benchmarks. More importantly, TERN covers 127 unique time-series and aggregate functions, exceeding even Spider 2.0-Snow, which contains 106 unique functions.

In addition, TERN differs from existing benchmarks in its focus on TSDB-native queries and time-series analytics. Although NL2InfluxDB explores this direction with 1,106 automatically synthesized examples, it uses a single database with limited query and schema complexity and lacks publicly available data. To address this gap, we introduce TERN, providing large-scale coverage of TSDB-native analytical queries.

7.3 Evaluation on Benchmark

We comprehensively evaluate existing methods on **TERN** across three categories: Fine-tuned LLMs, Frontier LLMs, and NL2SQL Methods. As shown in Table 2, we draw the following observations:

(1) **Fine-tuned LLMs.** Without fine-tuning, these models perform poorly, with even Qwen3.8-27B achieving only 5.33% overall EX. We therefore apply SFT, which substantially improves their EX and SER, supporting the effectiveness of our training corpus. Qwen3.8-27B achieves the highest EX across all three subsets, suggesting that larger models yield better performance. Qwen2.5-Coder-7B shows no clear advantage over similarly sized models, indicating limited benefits of code-oriented pretraining for TSSQL.

(2) **Frontier LLMs.** Frontier LLMs exhibit substantial performance variation. GPT-6.1 achieves the best EX and SER, outperforming GPT-5.6 by 11.65 pp in overall EX. In contrast, Qwen3.8 underperforms Qwen3.7, potentially due to excessive reasoning. These differences suggest varying degrees of transferability to TSSQL, making task-specific validation essential for model selection.

(3) **NL2SQL Methods.** SFT-based NL2SQL models show no clear advantage over general-purpose models, potentially due to the limited benefits of standard SQL pretraining for TSSQL. ICL-based methods perform relatively poorly, likely because their techniques designed for relational databases inadequately address TSSQL challenges.

(4) **Cross-paradigm Comparison.** Comparing the strongest models from each category, Qwen3.8-27B achieves the highest EX on TERN-F, while GPT-6.1 performs better on TERN-S and TERN-A. This suggests that SFT is particularly effective for basic TSSQL

Table 2: Performance comparison on the TERN benchmark.

Method	TERN-Foundation		TERN-Standard		TERN-Analytical		Overall	
	EX	SER	EX	SER	EX	SER	EX	SER
Fine-tuned LLMs (≤32B)								
Qwen3.8-27B(ICL)	9.17%	15.83%	4.34%	12.10%	2.45%	8.87%	5.33%	12.36%
GLM-4-9B(SFT)	58.61%	73.33%	29.22%	73.06%	21.71%	43.73%	36.44%	64.62%
Meta-Llama-3.1-8B-Instruct(SFT)	65.56%	75.83%	42.01%	74.89%	27.83%	51.07%	45.42%	68.27%
Qwen2.5-Coder-7B-Instruct(SFT)	55.83%	75.55%	36.07%	**83.79%**	27.52%	58.72%	39.91%	73.87%
Qwen3-30B-A3B-Instruct(SFT)	69.17%	78.89%	40.64%	81.96%	29.05%	52.29%	46.40%	72.36%
Qwen3.8-27B(SFT)	**74.44%**	82.78%	45.89%	78.31%	29.97%	56.88%	**50.40%**	73.51%
Frontier LLMs								
Qwen3.7-max	16.94%	45.28%	7.08%	27.40%	11.31%	25.38%	11.46%	32.53%
Qwen3.8-max	18.06%	46.39%	6.39%	21.23%	8.87%	31.80%	10.84%	32.36%
Qwen3-Coder-480B	29.17%	55.00%	7.53%	29.45%	8.26%	27.52%	14.67%	37.07%
DeepSeek-V4.1-Flash	54.72%	81.11%	31.05%	52.05%	22.94%	40.98%	36.27%	58.13%
GPT-5.6-sol	49.17%	71.94%	35.84%	63.01%	26.30%	39.45%	37.33%	59.02%
GPT-6.1-sol	68.61%	**92.22%**	**46.35%**	79.68%	**30.89%**	**59.63%**	48.98%	**77.87%**
NL2SQL Methods								
CodeS-15B (SFT)	31.39%	51.94%	10.50%	28.31%	7.95%	18.04%	16.44%	32.89%
OmniSQL-32B (SFT)	63.06%	76.67%	35.16%	78.31%	28.75%	56.27%	42.22%	71.38%
XiYan-SQL-32B (SFT)	64.72%	78.61%	38.58%	80.82%	27.83%	53.52%	43.82%	72.18%
DAIL-SQL	11.39%	23.89%	6.62%	18.95%	5.81%	15.90%	7.91%	19.64%
OpenSearch-SQL	30.83%	42.50%	8.22%	18.95%	3.06%	8.56%	13.96%	23.47%
DeepEye-SQL	42.50%	69.17%	11.64%	47.26%	8.87%	25.38%	20.71%	47.91%

queries, whereas GPT-6.1's stronger reasoning capabilities better support complex TSSQL queries and temporal analytics.

(5) **Task-specific Performance Analysis.** Across all baselines, EX generally declines from TERN-F to TERN-S and TERN-A. Even the best model achieves less than one-third EX on TERN-A, highlighting the challenges of predicting analytical time-series queries.

(6) **Executability vs. Correctness.** Even the best methods achieve only around 75% overall SER and 50% EX, indicating that limited ability to generate executable TSSQL constrains EX accuracy.

7.4 Error Analysis

To characterize failures, we evaluate the strongest baseline in each category on 300 randomly sampled examples from TERN. We analyze incorrect predictions and assign each a primary error category: *Validity* (syntax or execution violations), *Schema* (schema grounding or organization), *Temporal* (temporal structure or operation), *Expression* (function or computation), or *Output* (output organization). Figure 5 shows that Qwen3.8-27B has higher temporal and expression error shares than GPT-6.1-sol, suggesting weaker reasoning capabilities for complex temporal structures and expressions. GPT-6.1-sol has the lowest validity share but the highest schema and output shares, indicating potential benefits from dedicated schema linking and output organization mechanisms. DeepEye-SQL has the largest validity share, suggesting limited transferability to TSSQL.

Figure 6 presents concrete TSSQL examples and errors made by GPT-6.1-sol. In the first, the model omits the unit conversion implied by schema context and selects an inappropriate function. In addition, its `COUNT_IF` calls also lack a required argument. In the second, it enumerates series by device rather than by measurement, changing their expansion order. Consequently, `SLIMIT 3 SOFFSET 3`, which skips the first three series and returns the next three, selects an incorrect subset (highlighted in red) instead of the three requested `high_price` series, illustrating the challenges posed by specialized time-series schemas and output organization.

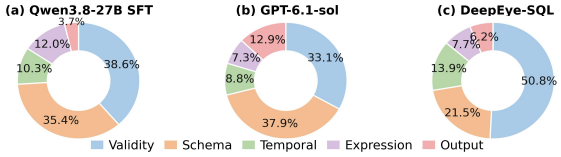


Figure 5: Statistics of Errors

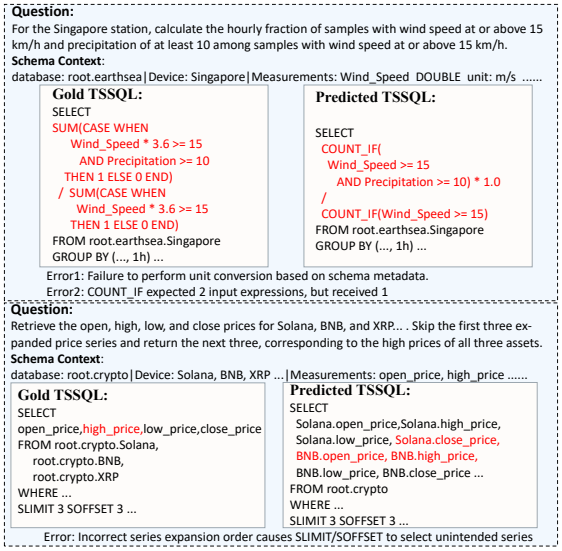


Figure 6: Examples of Prediction Errors

7.5 Effectiveness Comparison

To evaluate generation effectiveness, we generate 500 executable samples using each baseline on the same TSDB. For a fair comparison, all baselines use DeepSeek-V4.1-Flash as the backbone model.

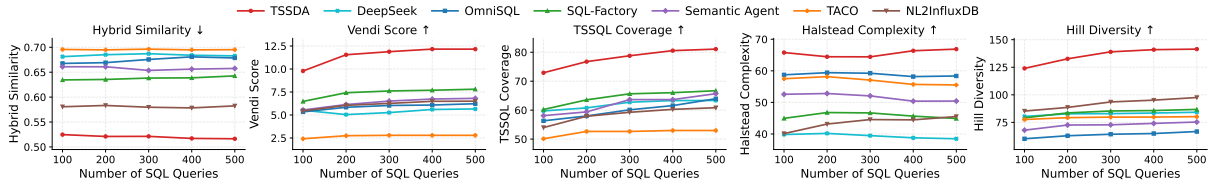


Figure 7: Scalability Comparison of Different Methods

In addition to the baselines, we evaluate three ablated variants of TSSDA, removing semantic dependencies, online adaptation, and memory, respectively, to assess each component's contribution. Table 3 reports the results and we have the following observations:
(1) **Overall Effectiveness.** TSSDA achieves the best performance across all metrics. It improves Vendi Score, TSSQL Coverage, and Hill Diversity by 55.80%, 21.56%, and 45.00%, respectively, over their strongest baselines. These results demonstrate its effectiveness in generating diverse TSSQL queries with broad feature coverage.
(2) **Generation Diversity.** Hybrid Similarity and Vendi Score assess sample similarity, with TSSDA achieving the best results on both. TACO performs worst, largely due to repeated instantiation of a limited skeleton set. TSSDA also achieves the highest Hill Diversity, whereas OmniSQL records the lowest score, indicating concentrated feature reuse despite its complexity-oriented generation. Among the baselines, SQL-Factory achieves the highest Vendi Score, potentially benefiting from its diversity-oriented query rewriting. In contrast, NL2InfluxDB performs best on Hybrid Similarity and Hill Diversity, likely due to its balanced query-family sampling.
(3) **Query Complexity.** Among baselines, OmniSQL and TACO produce the most complex queries, consistent with their complexity-oriented designs, while DeepSeek produces the simplest. SQL-Factory and NL2InfluxDB also show relatively low complexity. This trend contrasts with the diversity results, potentially because complex queries share more common SQL structures. In contrast, TSSDA achieves the best results in both complexity and diversity.
(4) **TSSQL Coverage.** TSSDA achieves the highest coverage, benefiting from its broad generation space and highly autonomous Agent, which facilitate exploration of previously unexplored query features. In contrast, TACO performs worst, indicating that its limited skeleton set constrains generation beyond predefined patterns.
(5) **Ablation Analysis.** Removing adaptation causes the largest degradation on most metrics, highlighting its central role in effectiveness. Dependency removal also degrades performance, supporting the value of semantic constraints and dependency context. Removing memory yields only minor changes, including slight gains on some metrics, suggesting that memory, designed to improve efficiency through experience reuse, largely preserves effectiveness.

7.6 Scalability Comparison

To assess how generation effectiveness scales with dataset size, we increase the number of generated SQL queries from 100 to 500 in increments of 100 and track all five metrics for TSSDA and the baselines. Observations derived from Figure 7 are as follows:
(1) As the dataset grows, TSSDA's Hybrid Similarity decreases overall, whereas most baselines remain stable or increase. Its Vendi Score, TSSQL Coverage, and Hill Diversity remain the highest and generally increase with dataset size, while most baselines show smaller gains. After an initial decline, TSSDA's complexity recovers and increases overall, while most baselines remain stable or decline. These trends suggest that TSSDA maintains superior effectiveness as generation scales.
(2) From 100 to 500 queries, TSSDA achieves the largest absolute gains in Vendi Score, TSSQL Coverage, and Hill Diversity, increasing by 2.37, 8.16, and 17.48, respectively. The corresponding largest baseline gains are 1.33 (SQL-Factory), 7.62 (OmniSQL), and 12.28 (NL2InfluxDB). These gains are achieved despite TSSDA starting from substantially higher scores.
(3) Across all evaluated dataset sizes, TSSDA consistently achieves the best results on all five metrics, demonstrating sustained effectiveness as generation scales.

Table 3: Effectiveness comparison of different methods.

Method	Hybrid Similarity	Vendi Score	TSSQL Coverage	Halstead Complexity	Hill Diversity
DeepSeek-V4.1	0.6827	5.6563	63.3424	38.4834	84.2069
OmniSQL	0.6786	6.2234	63.9392	58.3691	66.7064
SQL-Factory	0.6426	7.8040	66.7029	44.9125	86.7652
SemanticAgent	0.6575	6.8108	65.6844	50.4266	75.3282
TACO	0.6953	2.7882	52.9698	55.5263	80.2348
NL2InfluxDB	0.5824	6.5023	60.8359	45.5195	97.5521
TSSDA	**0.5164**	12.1590	**81.0817**	66.8761	**141.4524**
w/o dependency	0.5203	12.0060	79.6381	64.7948	136.7946
w/o adaptation	0.5580	10.1385	74.9015	65.5940	125.8635
w/o memory	0.5178	**12.2656**	80.9504	**66.9497**	141.2948

7.7 Efficiency Comparison

To evaluate generation efficiency, we compare the TSSQL Coverage and Hill Diversity under varying resource budgets. We separately vary the token budget from 1 to 31 million tokens and the generation time budget from 0.5 to 5 hours. Figure 8 presents the results:
(1) As token and time budgets increase, TSSDA consistently outperforms all baselines on both metrics from 6M tokens and 2 hours onward, indicating more effective use of generation resources to achieve broader feature coverage and greater dataset diversity.
(2) TSSDA achieves comparable effectiveness with substantially fewer resources. At 6M tokens or 2 hours, it surpasses all baselines at their respective budgets of 31M tokens or 5 hours.
(3) All methods generally improve with increasing token and time budgets, but most baselines, such as TACO and NL2InfluxDB, plateau early. In contrast, TSSDA continues to improve as resources increase, widening its lead over these methods and indicating more efficient scaling of data generation.

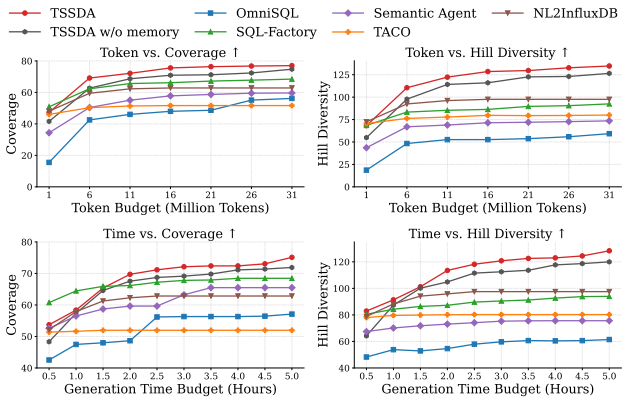


Figure 8: Efficiency Comparison of Different Methods

(4) The *w/o memory* variant shows lower TSSQL Coverage and Hill Diversity under matched budgets, supporting the efficiency benefits of the memory mechanism. Nevertheless, it still outperforms all baselines on both metrics at larger budgets.

8 RELATED WORK

8.1 Data Generation for NL2SQL

Data generation for NL2SQL can be broadly divided into two categories: NL2SQL benchmark construction and SQL generation. We discuss these two categories in detail below.

NL2SQL benchmark construction: Early NL2SQL benchmarks [27, 35, 73, 74] relied mainly on manual collection and annotation. However, manual construction requires substantial human effort and becomes particularly inefficient in the era of LLMs. To address this limitation, prior work has explored automated benchmark generation through predefined templates or data augmentation. DBPal [67] generates NL–SQL pairs by instantiating templates with schema elements, while HIER [69] synthesizes SQL from abstract syntax trees and generates questions using a neural model. With the rise of LLMs, recent work [23, 25, 41, 51, 68] has increasingly leveraged LLMs to assist benchmark construction. CLARITY [52] uses a schema-constrained LLM to generate ambiguous queries and multi-turn conversations from Spider and BIRD seeds. SQL-Exchange [4] employs template-guided query transformations to generate samples through cross-database transfer. However, these approaches still rely on seed data or templates, with LLMs primarily serving as auxiliary tools. Moreover, none of the above approaches target TSDBs. Accordingly, a few studies have begun to investigate generating time-series queries from natural language for TSDBs. TEFD [70] synthesizes Flux [1] queries from manually predefined query templates. However, Flux is a pipeline-oriented query language rather than an SQL dialect. NL2InfluxDB [9] provides InfluxDB [1] documentation to an LLM and directly generates queries. However, it produces only a small dataset through direct LLM synthesis, limiting systematic evaluation.

SQL generation: Beyond NL2SQL benchmark construction, SQL generation also involves automated query synthesis, but primarily aims to generate large-scale SQL workloads. Early SQL generation methods construct workloads by instantiating predefined templates or grammars [42, 45, 55]. TPC-H [46] samples parameters for expert-designed query templates over a fixed schema. JOB [29] grounds workloads in real-world data using 33 query structures with predicate variants, while DSB [8] extends TPC-DS [42] with configurable workload distributions to guide parameter instantiation. Despite increasing workload realism and control, these methods remain constrained by predefined query structures. Recent methods [5, 26, 53, 72] use LLMs to enable more flexible and scalable query synthesis. OmniSQL [33] builds databases from web tables and synthesizes 2.54M pretraining samples through LLM-based CoT. SQL-Factory [36] introduces a multi-agent framework for data generation, expansion, rewriting, and validation. However, although these approaches introduce LLM-based agents, their autonomy remains largely confined to predefined pipelines and procedures. Consequently, they cannot autonomously control the generation process according to the evolving data environment, limiting data coverage and diversity. In contrast, we model the entire generation process as an interactive and explorable data space, transforming the agent from a passive data consumer into a Data Agent capable of exploring, orchestrating, and managing data generation.

8.2 NL2SQL Methods

Early NL2SQL methods [32, 50] mapped natural language to SQL using hand-crafted semantic representations and domain knowledge. With the advancement of neural networks, subsequent studies [13, 54, 64, 65] began to predict SQL using architectures such as LSTMs and attention mechanisms. However, limited model capabilities and generalization constrained their practical applicability. The emergence of LLMs substantially accelerated the development of NL2SQL [12, 14]. DIN-SQL [48] employs in-context learning to decompose Text-to-SQL into multiple modules, such as schema linking and SQL generation. FINER-SQL [20] enhances SLMs through reasoning distillation and reinforcement learning with fine-grained execution feedback. More recent studies [16, 47, 49] have increasingly focused on multi-stage reasoning. OpenSearch-SQL [71] introduces a four-stage workflow with an alignment mechanism to maintain pipeline consistency. DeepEye-SQL [31] follows the software development lifecycle, integrating predefined reasoning, deterministic verification, and execution-aware selection. However, these methods focus on schema grounding and reasoning in relational databases, leaving TSSQL syntax and semantics, temporal operators, and specialized TSDB schemas insufficiently addressed.

9 CONCLUSION

We present **TSSDA**, a Data Agent framework for Text-to-TSSQL data generation, and construct **TERN**, a large-scale benchmark. First, we systematically model TSSQL syntax and semantics to construct a data generation space, ensuring broad coverage and validity. Next, we develop a Data Agent that autonomously explores and orchestrates data generation through continuous decision-making, query generation, and adaptation, improving data diversity and complexity. Finally, we design a temporal-aware memory mechanism that improves generation efficiency by reusing historical experience. Extensive experiments on real-world industrial datasets demonstrate that Text-to-TSSQL remains challenging, while **TSSDA** shows the best performance among the baseline methods.

REFERENCES

[1] 2026. InfluxDB. https://www.influxdata.com/products/influxdb-overview

[2] Qifeng Cai, Hao Liang, Chang Xu, Tao Xie, Wentao Zhang, and Bin Cui. 2026. Text2sql-Flow: a Robust Sql-Aware Data Augmentation Framework for Text-To-Sql. In *ICDE*. IEEE, 3308–3321.

[3] Bowen Cao, Weibin Liao, Yushi Sun, Dong Fang, Haitao Li, and Wai Lam. 2026. APEX-SQL: Talking to the data via Agentic Exploration for Text-to-SQL. In *Proceedings of the 32nd ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2, KDD 2026, Jeju Island, Korea, August 9-13, 2026*, Won Kim, Jae-Gil Lee, Kyuseok Shim, James Caverlee, David F. Gleich, Xiangliang Zhang, Qi He, Faisal Farooq, Rómer Rosales, Sayan Ranu, Haixun Wang, Yan Liu, Carl Yang, Leman Akoglu, Nitesh V. Chawla, and Vagelis Papalexakis (Eds.). ACM, 234–244.

[4] Mohammadreza Daviran, Brian Lin, and Davood Rafiei. 2026. SQL-Exchange: Transforming SQL Queries Across Domains. *Proc. VLDB Endow.* 19, 6 (2026), 1291–1304.

[5] Angjela Davitkova and Sebastian Michel. 2026. Persona-Conditioned Query Generation for Selectivity-Controlled Workloads. *Proc. VLDB Endow.* 19, 12 (2026), 4714–4717.

[6] DeepSeek-AI. 2026. DeepSeek-V4: Towards Highly Efficient Million-Token Context Intelligence. *CoRR* abs/2606.19348 (2026).

[7] Chao Deng, Ju Fan, Yuyu Luo, Qinliang Xue, Meihao Fan, Yuxin Zhang, Min Zhang, Xiaofeng Jia, Jing Zhang, and Xiaoyong Du. 2026. TACO: A Benchmark for Open-Domain Text-to-SQL with Ambiguous and Cross-Database Queries. *Proc. VLDB Endow.* 19, 6 (2026), 1386–1399.

[8] Bailu Ding, Surajit Chaudhuri, Johannes Gehrke, and Vivek R. Narasayya. 2021. DSB: A Decision Support Benchmark for Workload-Driven and Traditional Database Systems. *Proc. VLDB Endow.* 14, 13 (2021), 3376–3388.

[9] Lacramioara Dranca, Pablo Donate, Julio A. Sanguesa, Piedad Garrido, Vicente Torres-Sanz, and Francisco J. Martinez. 2026. LLMs for Industrial Databases: An Agro-Food Production Plant Use Case. *Frontiers in Artificial Intelligence* 9 (2026), 1764367. https://doi.org/10.3389/frai.2026.1764367

[10] Meihao Fan, Ju Fan, Nan Tang, Lei Cao, Guoliang Li, and Xiaoyong Du. 2025. AutoPrep: Natural Language Question-Aware Data Preparation with a Multi-Agent Framework. *Proc. VLDB Endow.* 18, 10 (2025), 3504–3517.

[11] Meihao Fan, Ju Fan, Yuxin Zhang, Shaolei Zhang, Xiaoyong Du, Jie Song, Peng Li, Fuxin Jiang, Tieying Zhang, and Jianjun Chen. 2026. DeepPrep: An LLM-Powered Agentic System for Autonomous Data Preparation. *Proc. VLDB Endow.* 19, 11 (2026), 3371–3384.

[12] Yuankai Fan, Zhenying He, Tonghui Ren, Can Huang, Yinan Jing, Kai Zhang, and X. Sean Wang. 2024. Metasql: A Generate-Then-Rank Framework for Natural Language to SQL Translation. In *ICDE*. IEEE, 1765–1778.

[13] Han Fu, Chang Liu, Bin Wu, Feifei Li, Jian Tan, and Jianling Sun. 2023. CatSQL: Towards Real World Natural Language to SQL Applications. *Proc. VLDB Endow.* 16, 6 (2023), 1534–1547.

[14] 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. *Proc. VLDB Endow.* 17, 5 (2024), 1132–1145.

[15] Qiang Gao, Zhenping Li, Anqi Zhuo, Yingxiao Zhao, Weibo Geng, and Xiaosong Li. 2026. SemanticAgent: A Semantics-Aware Framework for Text-to-SQL Data Synthesis. *CoRR* abs/2604.21414 (2026).

[16] Yue Gong, Chuan Lei, Xiao Qin, Kapil Vaidya, Balakrishnan Narayanaswamy, and Tim Kraska. 2025. SQLens: An End-to-End Framework for Error Detection and Correction in Text-to-SQL. In *NIPS*, Danielle Belgrave, Cheng Zhang, Laura N. Montoya, Hsuan-Tien Lin, Razvan Pascanu, Piotr Koniusz, Marzyeh Ghassemi, Nancy Chen, Iván Vladimir Meza Ruíz, and Arturo Loaiza-Bonilla (Eds.).

[17] Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. 2019. Towards Complex Text-to-SQL in Cross-Domain Database with Intermediate Representation. In *ACL*, Anna Korhonen, David R. Traum, and Lluís Màrquez (Eds.). Association for Computational Linguistics, 4524–4535.

[18] Yu Guo, Dong Jin, Shenghao Ye, Shuangwu Chen, Jian Yang, and Xiaobin Tan. 2025. SQLForge: Synthesizing Reliable and Diverse Data to Enhance Text-to-SQL Reasoning in LLMs. In *ACL (Findings of ACL)*, Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (Eds.), Vol. ACL 2025. Association for Computational Linguistics, 8441–8452.

[19] Maurice H. Halstead. 1977. *Elements of Software Science*. Elsevier Science Inc.

[20] Thanh Dat Hoang, Thanh Trung Huynh, Matthias Weidlich, Thanh Tam Nguyen, Tong Chen, Hongzhi Yin, and Quoc Viet Hung Nguyen. 2026. Boosting Small Language Models for Text-to-SQL with Fine-Grained Execution Feedback and Cost-Efficient Rewards. In *ICDE*. IEEE, 1–14.

[21] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, An Yang, Rui Men, Fei Huang, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-Coder Technical Report. *CoRR* abs/2409.12186 (2024).

[22] Jie Jiang, Haining Xie, Siqi Shen, Yu Shen, Zihan Zhang, Meng Lei, Yifeng Zheng, Yang Li, Chunyou Li, Danqing Huang, Yinjun Wu, Wentao Zhang, Xiaofeng Yang, Bin Cui, and Peng Chen. 2025. SiriusBI: A Comprehensive LLM-powered Solution for Data Analytics in Business Intelligence. *Proc. VLDB Endow.* 18, 12 (2025), 4860–4873.

[23] Tengjun Jin, Yoojin Choi, Yuxuan Zhu, and Daniel Kang. 2026. Pervasive Annotation Errors Break Text-to-SQL Benchmarks and Leaderboards. *Proc. VLDB Endow.* 19, 5 (2026), 931–944.

[24] George Katsogiannis-Meimarakis and Georgia Koutrika. 2023. A survey on deep learning approaches for text-to-SQL. *VLDB J.* 32, 4 (2023), 905–936.

[25] Apostolos Koukouvinis, George Katsogiannis-Meimarakis, and Georgia Koutrika. 2026. Comparison and Analysis of Value Linking in Text-to-SQL systems: [Experiments & Analysis]. *Proc. ACM Manag. Data* 4, 3 (2026), 152:1–152:26.

[26] Jiale Lao and Immanuel Trummer. 2026. SQLBarber: A System Leveraging Large Language Models to Generate Customized and Realistic SQL Workloads. *Proc. ACM Manag. Data* 4, 1 (2026), 85:1–85:27.

[27] Chia-Hsuan Lee, Oleksandr Polozov, and Matthew Richardson. 2021. KaggleDBQA: Realistic Evaluation of Text-to-SQL Parsers. In *ACL*, Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli (Eds.). Association for Computational Linguistics, 2261–2273.

[28] Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, Victor Zhong, Caiming Xiong, Ruoxi Sun, Qian Liu, Sida Wang, and Tao Yu. 2025. Spider 2.0: Evaluating Language Models on Real-World Enterprise Text-to-SQL Workflows. In *ICLR*. OpenReview.net.

[29] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2018. Query optimization through the looking glass, and what we found running the Join Order Benchmark. *VLDB J.* 27, 5 (2018), 643–668.

[30] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *NIPS*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.).

[31] Boyan Li, Chong Chen, Zhujun Xue, Yinan Mei, and Yuyu Luo. 2026. DeepEye-SQL: A Software-Engineering-Inspired Text-to-SQL Framework. *Proc. ACM Manag. Data* 4, 3 (2026), 158:1–158:28.

[32] Fei Li and H. V. Jagadish. 2014. Constructing an Interactive Natural Language Interface for Relational Databases. *Proc. VLDB Endow.* 8, 1 (2014), 73–84.

[33] Haoyang Li, Shang Wu, Xiaokang Zhang, Xinmei Huang, Jing Zhang, Fuxin Jiang, Shuai Wang, Tieying Zhang, Jianjun Chen, Rui Shi, Hong Chen, and Cuiping Li. 2025. OmniSQL: Synthesizing High-quality Text-to-SQL Data at Scale. *Proc. VLDB Endow.* 18, 11 (2025), 4695–4709.

[34] 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. *Proc. ACM Manag. Data* 2, 3 (2024), 127.

[35] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin Chen-Chuan Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023. Can LLM Already Serve as A Database Interface? A BIg Bench for Large-Scale Database Grounded Text-to-SQLs. In *NIPS*, Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (Eds.).

[36] Jiahui Li, Tongwang Wu, Yuren Mao, Yunjun Gao, Yajie Feng, and Huaizhong Liu. 2025. SQL-Factory: A Multi-Agent Framework for High-Quality and Large-Scale SQL Generation. *Proc. VLDB Endow.* 19, 3 (2025), 292–305.

[37] Yiyan Li, Haoyang Li, Jing Zhang, Renata Borovica-Gajic, Shuai Wang, Tieying Zhang, Jianjun Chen, Rui Shi, Cuiping Li, and Hong Chen. 2025. AgentTune: An Agent-Based Large Language Model Framework for Database Knob Tuning. *Proc. ACM Manag. Data* 3, 6 (2025), 293:1–293:29.

[38] Yifu Liu, Yin Zhu, Yingqi Gao, Zhiling Luo, Xiaoxia Li, Xiaorong Shi, Yuntao Hong, Jinyang Gao, Yu Li, Bolin Ding, and Jingren Zhou. 2026. XiYan-SQL: A Novel Multi-Generator Framework for Text-to-SQL. *IEEE Trans. Knowl. Data Eng.* 38, 4 (2026), 2474–2487.

[39] Yuyu Luo, Guoliang Li, Ju Fan, Chengliang Chai, and Nan Tang. 2025. Natural Language to SQL: State of the Art and Open Problems. *Proc. VLDB Endow.* 18, 12 (2025), 5466–5471. https://doi.org/10.14778/3750601.3750696

[40] Fei Lyu, Zhiyi Peng, Jiaming Liu, Yixuan Yang, Changjian Chen, Zhuo Tang, Jiapeng Zhang, and Kenli Li. 2026. TQTS-Bench: A Multi-Syntax Benchmark for Text-to-Query over Time-Series Databases. *arXiv preprint arXiv:2609.34783* (2026).

[41] Mohammadhossein Malekpour, Mohamed Riahi, Maxime Lamothe, and Amine Mhedhbi. 2026. SQLMorph: Query Mutation and Fine-Grained Metrics for Text-to-SQL Evaluation. In *ICDE*. IEEE, 2628–2640.

[42] Raghunath Othayoth Nambiar and Meikel Poess. 2006. The Making of TPC-DS. In *VLDB*, Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim (Eds.). ACM, 1049–1058.

[43] OpenAI. 2026. *Addendum to GPT-6 Astra System Card: GPT-6.1 Sol.* Technical Report. OpenAI. https://deploymentsafety.openai.com/gpt-6-1-sol

[44] OpenAI. 2026. *GPT-5.6 System Card.* Technical Report. OpenAI. https://deploymentsafety.openai.com/gpt-5-6

[45] Meikel Poess and John M. Stephens. 2004. Generating Thousand Benchmark Queries in Seconds. In *VLDB*, Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer (Eds.). Morgan Kaufmann, 1045–1053.

[46] Meikel Pöss and Chris Floyd. 2000. New TPC Benchmarks for Decision Support and Web Commerce. *SIGMOD Rec.* 29, 4 (2000), 64–71.

[47] Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan Ö. Arik. 2025. CHASE-SQL: Multi-Path Reasoning and Preference Optimized Candidate Selection in Text-to-SQL. In *ICLR*. OpenReview.net.

[48] Mohammadreza Pourreza and Davood Rafiei. 2023. DIN-SQL: Decomposed In-Context Learning of Text-to-SQL with Self-Correction. In *NIPS*, Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (Eds.).

[49] Anik Pramanik, Murat Kantarcioglu, Vincent Oria, and Shantanu Sharma. 2026. DexterSQL: Deep Schema Exploration and Rule-based Correction for Text-to-SQL Generation. *CoRR* abs/2608.11889 (2026).

[50] Diptikalyan Saha, Avrilia Floratou, Karthik Sankaranarayanan, Umar Farooq Minhas, Ashish R. Mittal, and Fatma Özcan. 2016. ATHENA: An Ontology-Driven System for Natural Language Querying over Relational Data Stores. *Proc. VLDB Endow.* 9, 12 (2016), 1209–1220.

[51] Irina Saparina and Mirella Lapata. 2024. AMBROSIA: A Benchmark for Parsing Ambiguous Questions into Database Queries. In *NIPS*, Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (Eds.).

[52] Tabinda Sarwar, Farhad Moghimifar, Cong Duy Vu Hoang, Xiaoxiao Ma, Shawn Chang Xu, Fahimeh Saleh, Poorya Zaremoodi, Avirup Sil, and Katrin Kirchhoff. 2026. CLARITY: A Framework and Benchmark for Conversational Language Ambiguity and Unanswerability in Interactive NL2SQL Systems. In *ACL*, Yunyao Li, Georg Rehm, and Mei Tu (Eds.). Association for Computational Linguistics, 1218–1250.

[53] Tobias Schmidt, Viktor Leis, Peter Boncz, and Thomas Neumann. 2025. SQLStorm: Taking Database Benchmarking into the LLM Era. *Proc. VLDB Endow.* 18, 11 (2025), 4144–4157.

[54] Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. PICARD: Parsing Incrementally for Constrained Auto-Regressive Decoding from Language Models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, 9895–9901.

[55] Donald R. Slutz. 1998. Massive Stochastic Testing of SQL. In *VLDB*, Ashish Gupta, Oded Shmueli, and Jennifer Widom (Eds.). Morgan Kaufmann, 618–622.

[56] Marko Sterbentz, Kevin Cushing, Cameron Barrie, and Kristian J. Hammond. 2026. RingSQL: Generating Synthetic Data with Schema-Independent Templates for Text-to-SQL Reasoning Models. *CoRR* abs/2601.05451 (2026).

[57] María José Suárez-Cabal and Javier Tuya. 2009. Structural Coverage Criteria for Testing SQL Queries. *J. Univers. Comput. Sci.* 15, 3 (2009), 584–619.

[58] Giovanni Sullutrone, Luca Sala, Sania Aftar, Georgia Koutrika, and Sonia Bergamaschi. 2026. ABISS: Evaluating Text-to-SQL Systems Through Agent Interaction. *CoRR* abs/2607.23340 (2026).

[59] Zirui Tang, Boyu Niu, Xuanhe Zhou, Boxiu Li, Wei Zhou, Jiannan Wang, Guoliang Li, Xinyi Zhang, and Fan Wu. 2025. ST-Raptor: LLM-Powered Semi-Structured Table Question Answering. *Proc. ACM Manag. Data* 3, 6 (2025), 364:1–364:27.

[60] Llama Team. 2024. The Llama 3 Herd of Models. *CoRR* abs/2407.21783 (2024).

[61] Qwen Team. 2025. Qwen3 Technical Report. *CoRR* abs/2505.09388 (2025).

[62] Team GLM, Aohan Zeng, Bin Xu, Bowen Wang, et al. 2024. ChatGLM: A Family of Large Language Models from GLM-130B to GLM-4 All Tools. *CoRR* abs/2406.12793 (2024).

[63] Aditya Vashistha and Shrainik Jain. 2016. Measuring Query Complexity in SQLShare Workload. Technical report, University of Washington.

[64] Enzo Veltri, Gilbert Badaro, Mohammed Saeed, and Paolo Papotti. 2023. Data Ambiguity Profiling for the Generation of Training Examples. In *ICDE*. IEEE, 450–463.

[65] Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2020. RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers. In *ACL*, Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel R. Tetreault (Eds.). Association for Computational Linguistics, 7567–7578.

[66] Chen Wang, Jialin Qiao, Xiangdong Huang, Shaoxu Song, Haonan Hou, Tian Jiang, Lei Rui, Jianmin Wang, and Jiaguang Sun. 2023. Apache IoTDB: A Time Series Database for IoT Applications. *Proc. ACM Manag. Data* 1, 2 (2023), 195:1–195:27.

[67] Nathaniel Weir, Prasetya Ajie Utama, Alex Galakatos, Andrew Crotty, Amir Ilkhechi, Shekar Ramaswamy, Rohin Bhushan, Nadja Geisler, Benjamin Hättasch, Steffen Eger, Ugur Çetintemel, and Carsten Binnig. 2020. DBPal: A Fully Pluggable NL2SQL Training Pipeline. In *SIGMOD*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 2347–2361.

[68] Fabian Wenz, Omar Bouattour, Devin Yang, Justin Choi, Cecil Gregg, Nesime Tatbul, and Çagatay Demiralp. 2026. BenchPress: A Human-in-the-Loop Annotation System for Rapid Text-to-SQL Benchmark Curation. In *CIDR*. www.cidrdb.org.

[69] Kun Wu, Lijie Wang, Zhenghua Li, Ao Zhang, Xinyan Xiao, Hua Wu, Min Zhang, and Haifeng Wang. 2021. Data Augmentation with Hierarchical SQL-to-Question Generation for Cross-domain Text-to-SQL Parsing. In *EMNLP*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, 8974–8983.

[70] Xuefeng Wu, Yuanfeng Song, Jiawei Wen, Dandan Huang, Raymond Chi-Wing Wong, and Haodi Zhang. 2026. TEFD: A Benchmark for Natural Language to Flux Query Generation in Time-Series Databases. In *SIGKDD*, Won Kim, Jae-Gil Lee, Kyuseok Shim, James Caverlee, David F. Gleich, Xiangliang Zhang, Qi He, Faisal Farooq, Rómer Rosales, Sayan Ranu, Haixun Wang, Yan Liu, Carl Yang, Leman Akoglu, Nitesh V. Chawla, and Vagelis Papalexakis (Eds.). ACM, 10020–10031.

[71] Xiangjin Xie, Guangwei Xu, Lingyan Zhao, and Ruijie Guo. 2025. OpenSearch-SQL: Enhancing Text-to-SQL with Dynamic Few-shot and Consistency Alignment. *Proc. ACM Manag. Data* 3, 3 (2025), 194:1–194:24.

[72] Jiaxi Yang, Binyuan Hui, Min Yang, Jian Yang, Junyang Lin, and Chang Zhou. 2024. Synthesizing Text-to-SQL Data from Weak and Strong LLMs. In *ACL*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, 7864–7875.

[73] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir R. Radev. 2018. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. In *EMNLP*, Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii (Eds.). Association for Computational Linguistics, 3911–3921.

[74] Tao Yu, Rui Zhang, Michihiro Yasunaga, Yi Chern Tan, Xi Victoria Lin, Suyi Li, Heyang Er, Irene Li, Bo Pang, Tao Chen, Emily Ji, Shreya Dixit, David Proctor, Sungrok Shim, Jonathan Kraft, Vincent Zhang, Caiming Xiong, Richard Socher, and Dragomir R. Radev. 2019. SParC: Cross-Domain Semantic Parsing in Context. In *ACL*, Anna Korhonen, David R. Traum, and Lluís Màrquez (Eds.). Association for Computational Linguistics, 4511–4523.

[75] Yi Zhang, Jan Deriu, George Katsogiannis-Meimarakis, Catherine Kosten, Georgia Koutrika, and Kurt Stockinger. 2023. ScienceBenchmark: A Complex Real-World Benchmark for Evaluating Natural Language to SQL Systems. *Proc. VLDB Endow.* 17, 4 (2023), 685–698.

[76] Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning. *CoRR* abs/1709.00103 (2017).

[77] Yunfan Zhou, Qiming Shi, Yizhou Yang, Di Weng, and Yingcai Wu. 2026. BIRD-History: A Benchmark for History-Driven Text-to-SQL with Fine-Grained Knowledge Annotations. *CoRR* abs/2608.29345 (2026).

[78] Jun-Peng Zhu, Boyan Niu, Peng Cai, Zheming Ni, Jianwei Wan, Kai Xu, Jiajun Huang, Shengbo Ma, Bing Wang, Xuan Zhou, Guanglei Bao, Donghui Zhang, Liu Tang, and Qi Liu. 2025. Towards Automated Cross-domain Exploratory Data Analysis through Large Language Models. *Proc. VLDB Endow.* 18, 12 (2025), 5086–5099.

[79] Yizhang Zhu, Liangwei Wang, Chenyu Yang, Xiaotian Lin, Boyan Li, Wei Zhou, Xinyu Liu, Zhangyang Peng, Tianqi Luo, Yu Li, Chengliang Chai, Chong Chen, Shimin Di, Ju Fan, Ji Sun, Nan Tang, Fugee Tsung, Jiannan Wang, Chenglin Wu, Yanwei Xu, Shaolei Zhang, Yong Zhang, Xuanhe Zhou, Guoliang Li, and Yuyu Luo. 2025. A Survey of Data Agents: Emerging Paradigm or Overstated Hype? *CoRR* abs/2510.23587 (2025).