PLForge: Enhancing Language Models for Natural Language to Procedural Extensions of SQL

HANG ZHANG, Tsinghua University, China
CHAOKUN WANG*, Tsinghua University, China
HONGWEI LI, Tsinghua University, China
CHENG WU, Tsinghua University, China
SONGYAO WANG, Tsinghua University, China
YABIN LIU, Tsinghua University, China
GENGYUAN SHI, Tsinghua University, China
ZIYANG LIU, Tsinghua University, China

Procedural Language extensions of SQL (abbr. PL/SQL) enhance database programming by integrating procedural constructs with SQL's declarative syntax, thereby improving the reusability, modularity, and maintainability of SQL. Besides, PL/SQL in database systems presents significant challenges in real-world development, primarily due to the inherent complexity of programming. To reduce the development difficulty of PL/SQL, this paper studies the novel task of translating natural language (NL) to PL/SQL (i.e., NL-to-PL/SQL), aimed at simplifying PL/SQL development. Recent advancements in language models have shown promise in translating natural language questions into SQL queries (i.e., Text-to-SQL). However, the state-of-the-art Text-to-SQL methods focus only on single SQL queries, neglecting the procedural extensions of SQL, which limits their effectiveness for the NL-to-PL/SQL task.

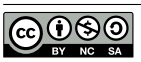
In this paper, we propose PLForge, a suite of pre-trained language models with parameter configurations of 3B, 7B, and 15B, tailored for NL-to-PL/SQL tasks. To enhance the PL/SQL generation capabilities of PLForge, we leverage a curated PL/SQL-centric data corpus and employ an incremental pre-training approach. Furthermore, to fully exploit the potential of PLForge, we propose a comprehensive prompt construction strategy tailored specifically for PL/SQL. Given the scarcity of NL-to-PL/SQL datasets, we develop a template-based method for generating NL-to-PL/SQL data. We conduct a series of experiments on PLForge and several baseline models. Based on execution match and exact match metrics that are designed specifically for the NL-to-PL/SQL task, the experimental results demonstrate that PLForge outperforms existing models in both in-context learning and supervised fine-tuning settings.

CCS Concepts: • **Software and its engineering** → *Domain specific languages*; • **Information systems** → **Query languages**; **Language models**.

Additional Key Words and Phrases: NL-to-PL/SQL, Language Model, Natural Language Interface for Databases

*Corresponding author (chaokun@tsinghua.edu.cn).

Authors' Contact Information: Hang Zhang, Tsinghua University, Beijing, China, zhanghang24@mails.tsinghua.edu.cn; Chaokun Wang, Tsinghua University, Beijing, China, chaokun@tsinghua.edu.cn; Hongwei Li, Tsinghua University, Beijing, China, lhw23@mails.tsinghua.edu.cn; Cheng Wu, Tsinghua University, Beijing, China, wuc22@mails.tsinghua.edu.cn; Songyao Wang, Tsinghua University, Beijing, China, wangsong23@mails.tsinghua.edu.cn; Yabin Liu, Tsinghua University, Beijing, China, liu-yb23@mails.tsinghua.edu.cn; Gengyuan Shi, Tsinghua University, Beijing, China, shigy19@mails.tsinghua.edu.cn; Ziyang Liu, Tsinghua University, Beijing, China, liu-zy21@mails.tsinghua.edu.cn.



ACM 2836-6573/2025/12-ART348
https://doi.org/10.1145/3769813

ACM Reference Format:
Hang Zhang, Chaokun Wang, Hongwei Li, Cheng Wu, Songyao Wang, Yabin Liu, Gengyuan Shi, and Ziyang Liu. 2025. PLForge: Enhancing Language Models for Natural Language to Procedural Extensions of SQL. *Proc. ACM Manag. Data* 3, 6 (SIGMOD), Article 348 (December 2025), 28 pages. https://doi.org/10.1145/3769813

1 INTRODUCTION

The fusion of procedural constructs with SQL in PL/SQL[1] enhances database programming by improving reusability, modularity, maintainability, and performance [20, 28]. This integration allows for direct execution of data manipulations, control structures, and error handling within the database. Widely supported by relational database management systems, PL/SQL extends SQL with stored procedures, user-defined functions, and triggers [6, 16, 20]. Through the combination of procedural constructs with declarative SQL, these studies improve both code reuse and system modularity. Furthermore, PL/SQL reduces communication overhead by enabling business logic execution within database systems, thus enhancing the performance [31].

Although PL/SQL is widely used in real-world production environments, its development process presents significant complexity and challenges. PL/SQL plays a crucial role in various types of operational environments [28]. According to statistics, there are more than two billion defined PL/SQL procedures in Microsoft Azure SQL Database Services, resulting in billions of daily invocations [28]. Despite its importance, the development of PL/SQL procedures is nontrivial [31, 48]. Developers must carefully craft declarative statements for data access and modification, while the inherent complexity of procedural programming obscures the control logic, making it difficult to comprehend and implement effectively [11].

In this study, to address the inherent complexity of PL/SQL development, we propose a novel task of translating natural language (NL) to PL/SQL (i.e., NL-to-PL/SQL). This initiative provides developers with a method to generate PL/SQL procedures directly from NL descriptions, thus significantly reducing the complexity of PL/SQL development.

Example 1. *We implement a database-side version of the Apache OFBiz* `createOrder` *API [2] as a stored procedure, specifically in PostgreSQL using PL/pgSQL. Figure 1 compares the workflows for implementing this procedure through manual development and the NL-to-PL/SQL approach. Manual development of stored procedure is relatively time-consuming, necessitating a thorough understanding of the database and business logic, followed by the meticulous task of coding line by line. In contrast, developing PL/SQL through NL is significantly more streamlined. Users simply need to provide a description of the PL/SQL and the corresponding database, which are then used to generate PL/SQL procedure through the NL-to-PL/SQL method.*

However, to the best of our knowledge, existing studies face difficulties in the NL-to-PL/SQL task. First, Text-to-SQL approaches, whether rule-based [5, 34, 56, 58], deep learning-based [7, 21, 42], or LLM-based [23, 50, 54], predominantly focus on generating single SQL queries rather than PL/SQL, and their evaluation metrics are inapplicable to NL-to-PL/SQL. Second, general code generation language models [39, 46, 47] are primarily trained on a mixture of programming languages (e.g., C, Python, and Java). These models excel in multi-language generation but struggle with domain-specific languages like PL/SQL. Third, existing PL/SQL generation approaches, such as WeBridge [31], are confined to translating ORM-generated database access code into PL/SQL and do not support NL inputs.

In this paper, we use language models to address the NL-to-PL/SQL task. We outline two main technical challenges in addressing this task. **C1: How to pre-train language models to perform**

[1]PL/SQL was originally developed by Oracle. In this paper, we use PL/SQL as a generic term for procedural extensions of SQL, including Oracle PL/SQL, PL/pgSQL, and other dialects.

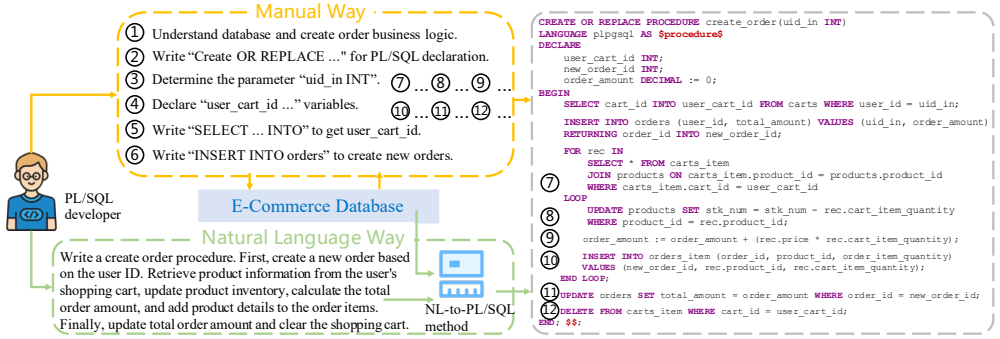


Fig. 1. A comparison of scenarios between manually developing PL/SQL and developing PL/SQL through natural language.

complex NL-to-PL/SQL reasoning? Existing pre-trained large language models (LLMs), such as StarCoder2 and CodeLlama, face significant challenges when applied to the NL-to-PL/SQL task. The challenges mainly stem from the fact that the training datasets of these models contain only a small proportion of SQL and PL/SQL code. Since the pre-training phase aims to fit the model to the overall distribution of the data, this data bias results in poor performance for domain-specific language tasks like NL-to-PL/SQL. **C2: How can the characteristics of PL/SQL be better exploited to generate more accurate PL/SQL?** Several related studies [17, 54] explore methods for capturing the skeleton of SQL. Furthermore, previous studies [50, 63] focus on the structural decomposition of SQL. However, these approaches are inadequate for effectively handling and capturing the modular and procedural nature of the PL/SQL code.

To tackle the above challenges, this paper proposes PLForge specifically designed for the NL-to-PL/SQL task. For challenge C1, we bring forward an incremental pre-training approach utilizing a vast curated NL-to-PL/SQL dataset with 22.2GB data in total, including PL/SQL, SQL, NL-related, and NL-to-code data. We incrementally pre-train StarCoder2 [46] on this corpus to obtain PLForge models with 3B, 7B, and 15B parameters, each exhibiting strong PL/SQL generation capabilities. To address challenge C2, we develop a specialized prompt construction tailored for PL/SQL, incorporating schema pruning, database metadata, PL/SQL chain of thought, and PL/SQL predicted skeleton. Consequently, our PLForge demonstrates superior performance in the NL-to-PL/SQL task in both the in-context learning (ICL) and supervised fine-tuning (SFT) settings.

We conduct a comprehensive evaluation of PLForge and baseline models. Our experiments primarily focus on ICL and SFT settings. A detailed analysis of our approach is then undertaken, including adjustments in hyperparameters, sensitivity to PL/SQL keywords, and ablation studies. This analysis substantiates the superiority of our method in the NL-to-PL/SQL task.

The main contributions of this paper are summarized as follows:

(1) We formally define the NL-to-PL/SQL task, aimed at simplifying the complexity inherent in PL/SQL development tasks (Section 2).
(2) We curate a comprehensive corpus for incremental pre-training of language models dedicated to the NL-to-PL/SQL task and release a template-based dataset for NL-to-PL/SQL, facilitating further research on the NL-to-PL/SQL task (Section 3).
(3) We propose PLForge, a series of language models ranging from 3B to 15B parameters, designed for PL/SQL generation. We enhance PLForge's performance using a comprehensive prompt construction strategy and PL/SQL error-aware reasoning correction (Section 4).

(4) We define execution match and exact match metrics for the NL-to-PL/SQL task. Based on these metrics, we conduct a series of evaluations on our NL-to-PL/SQL dataset and demonstrate that PLForge surpasses other notable language models in PL/SQL code generation (Section 5).

2 PRELIMINARIES

This section provides fundamental concepts in this paper, including NL-to-PL/SQL, SQL skeleton, and pre-trained language models.

NL-to-PL/SQL Task. For an NL-to-PL/SQL task, the input comprises an NL description x and a database $db = \langle db_s, db_m \rangle$, where db_s represents the schema (e.g., tables and columns), and db_m denotes metadata (e.g., column types and sample values). The objective is to generate a functionally equivalent PL/SQL program y that accurately performs the operations described in x on the given db. This task is defined as $\hat{y} = M(x, db)$, where M aims to interpret x with respect to db and generate a candidate translation $\hat{y}$. The output $\hat{y}$ is considered accurate if it is functionally equivalent to the target y.

SQL Skeleton. The SQL skeleton is designed to emphasize the logical operator structure within SQL queries while omitting specific database details. By using placeholders in the form of underscores ("_"), the skeleton is abstracted from particular elements such as tables, columns, and constant values. Figure 2(a) is an example of SQL and its corresponding skeleton.

Pre-trained Language Models. Pre-trained language models are sophisticated models primarily based on the Transformer [61] architecture, trained extensively on large text corpora. They can be categorized into two main paradigms: autoregressive language modeling, which predicts the next token in a sequence (e.g., GPT [51], LLaMA [60]), and masked language modeling, which reconstructs masked tokens from their surrounding context (e.g., BERT [1], T5 [52]). These models excel at generalizing across different NL-related tasks, offering superior capabilities in language understanding and generation.

3 PL/SQL-centric Datasets Building

We first propose a novel template-based method for the NL-to-PL/SQL dataset generation, along with two constructed datasets to support the NL-to-PL/SQL task. Then we meticulously collect a new data corpus from publicly available, real-world datasets including 11GB of PL/SQL data, 2.4GB of SQL data, 3.6GB NL-related data, and 5.2GB of NL-to-code data for incremental pre-training.

3.1 NL-to-PL/SQL Dataset Generation

Research in the NL-to-PL/SQL methodology development and evaluation requires specialized datasets, but the publicly available corpora that contain PL/SQL are scarce, with none specifically designed for the NL-to-PL/SQL studies. To bridge this gap, we propose a systematic dataset generation method involving three phases: initialization, template augmentation, and instantiation.

In the *initialization* phase, we establish a core set of 20 representative NL-to-PL/SQL templates, each of which consists of paired NL template and PL/SQL template that express equivalent query requirements, as illustrated in Figure 2(b). The NL-to-PL/SQL templates are initially built using detailed and semantically explicit NL paragraphs to promote the alignment of NL and PL/SQL procedures. Specifically, we first curate 20 functionally diverse and structurally distinct PL/SQL scripts from real-world platforms including Stack Overflow and GitHub. For each script, we manually construct a corresponding NL description, forming 20 NL-PL/SQL pairs grounded in practical applications. These pairs are subsequently abstracted into template forms to create the core NL-to-PL/SQL template set.

SQL

```
UPDATE employee SET base_salary = base_salary * (1 + com_pct)
WHERE department_id = (
    SELECT department_id FROM department
    WHERE department_name = "Sales") AND com_pct > 0.05;
```

Skeleton

```
UPDATE _ SET _ = _ * (_ + _)
WHERE _ = (SELECT _ FROM _ WHERE _ = _) AND _ > _;
```

(a) An example of SQL skeleton.

NL template

```
Create a stored procedure in PL/pgSQL that opens a cursor to select all
records from the {TABLE}. For each record in the cursor, update the {COLUMN}
to {PARAMETER}, if the {COLUMN2} equals {PARAMETER2}. After processing all
records, delete rows from the {2TABLE} where the {2COLUMN} is {2PARAMETER} or
{2COLUMN2} is less than {2PARAMETER2}. Finally, update the {3COLUMN} in the
{3TABLE} to {3PARAMETER} where the {3COLUMN2} equals {3PARAMETER2}.
```

PL/SQL template

```
CREATE OR REPLACE PROCEDURE sp({PARAMETER} {TYPE}, {PARAMETER2} {TYPE2},
{2PARAMETER} {2TYPE}, {2PARAMETER2} {2TYPE2}, {3PARAMETER} {3TYPE},
{3PARAMETER2} {3TYPE2})
LANGUAGE plpgsql AS $$
DECLARE
    ref_cursor CURSOR FOR SELECT * FROM {TABLE};
    rec RECORD;
BEGIN
    OPEN ref_cursor;
    LOOP
        FETCH ref_cursor INTO rec;
        EXIT WHEN NOT FOUND;
        IF rec.{COLUMN2} = {PARAMETER2} THEN
            UPDATE {TABLE} SET {COLUMN} = {PARAMETER}
            WHERE CURRENT OF ref_cursor;
        END IF;
    END LOOP;
    DELETE FROM {2TABLE}
    WHERE {2COLUMN} = {2PARAMETER} OR {2COLUMN2} < {2PARAMETER2};
    UPDATE {3TABLE}
    SET {3COLUMN} = {3PARAMETER}
    WHERE {3COLUMN2} = {3PARAMETER2};
    CLOSE ref_cursor;
END; $$;
```

(b) An example of NL-to-PL/SQL template.

Fig. 2. Examples of SQL skeleton and NL-to-PL/SQL template.

The *template augmentation* phase employs an iterative "generate-verify" approach to expand the core set of NL-to-PL/SQL templates, overcoming the limitation of scarce real-world NL-to-PL/SQL pairs. We maintain a growing pool of verified NL-to-PL/SQL templates. In each iteration, we randomly select eight existing NL-to-PL/SQL templates from the pool and use them as prompts for GPT-4o to generate a new candidate NL-to-PL/SQL template. The correctness of the PL/SQL template of the generated candidate NL-to-PL/SQL template is verified in a random selected database. Specifically, we instantiate the PL/SQL template multiple times with varying database schemas and parameter settings, and execute each resulting PL/SQL script within the selected database. A PL/SQL template is deemed valid if at least one execution succeeds, with its corresponding candidate NL-to-PL/SQL template being added to the pool. We initially apply the augmentation method to expand the growing pool to 102 templates. To further enhance the diversity of the templates and better reflect real-world scenarios, we subsequently employ eight domain experts, each possessing at least five years of experience in database systems. These experts construct an additional 100 NL-to-PL/SQL template sets, which are incorporated into the growing pool used during the template

Table 1. Taxonomy of 320 templates

Category	Description	Key PL/SQL Construct	Amount
Data Retrieval	DQL for retrieving relational data	`SELECT`	158
Database Operation	DML for modifying relational data	`INSERT`, `UPDATE`, `DELETE`	320
Control Flow	Branching and decision-making	`IF-ELSE`, `CASE WHEN`	78
Batch Processing	Looping over data or logic units	`FOR`, `LOOP`, `CURSOR`	108

augmentation phase. Building upon this enriched pool, we then continue generating templates through the iterative template augmentation process. As a result, we ultimately construct a total of 320 NL-to-PL/SQL templates. It is important to note that this total does not include the core set of 20 NL-to-PL/SQL templates.

The final *instantiation* phase produces the comprehensive NL-to-PL/SQL dataset. For each NL-to-PL/SQL template, we create multiple instances by applying diverse database schemas and parameter settings. Each NL-to-PL/SQL instance comprises an NL description paired with its corresponding executable PL/SQL script. To further enhance linguistic diversity while maintaining semantic fidelity, we leverage GPT-3.5 to generate paraphrased versions of the NL descriptions in the NL-to-PL/SQL instances, thereby enriching the robustness of the dataset without compromising the functional correctness of the PL/SQL codes.

To better reflect real-world user expressions and improve the model's semantic alignment with executable PL/SQL logic, we propose a two-stage NL construction strategy for each NL-to-PL/SQL instance. First, we utilize GPT-3.5 to generate a concise NL description that faithfully expresses the user intent while omitting low-level implementation details such as database details, variable declaration, cursor in loop mechanics, and so on. This concise NL description reflects how users typically issue high-level requests in practice. Next, we employ GPT-3.5 to transform the concise NL description and corresponding database schema into an expanded, structured NL paragraph that includes precise database and code-level semantics. This detailed NL description aims to provide stronger supervision for code generation models by explicitly aligning with procedural logic.

Based on the proposed method, we construct two NL-to-PL/SQL datasets. Specifically, we classify the 320 augmented NL-to-PL/SQL templates according to two difficulty levels (simple and hard). The taxonomy of 320 templates is shown in Table 1. For each difficulty category, we further split the corresponding templates into two mutually exclusive training and testing partitions, guaranteeing that the testing NL-to-PL/SQL templates remain completely unseen during the model training. This partitioning scheme ensures rigorous assessment of generalization capabilities. The statistics of the constructed datasets, namely Simple and Hard, are analyzed in Section 5.1.1.

3.2 Incremental Pre-training Data Corpus

To improve the NL-to-PL/SQL model's ability to generate accurate PL/SQL code and comprehend NL, we construct four datasets from different perspectives. The first and second datasets comprise PL/SQL and SQL data, respectively, while the third focuses on NL data. Because PL/SQL extends SQL with procedural programming capabilities, the fourth dataset consists of NL-to-code data.

PL/SQL data [11GB]. To augment the PL/SQL code generation proficiency of language models, we incorporate the PL/SQL segment derived from the StarCoder2's pre-training corpus [46], predominantly comprising PL/SQL and PL/pgSQL with a minor inclusion of T-SQL and SQL PL corpora. We refine the dataset by cleansing it based on syntactic keywords of PL/SQL and excluding excessively lengthy texts to enhance overall data quality.

SQL data [2.4GB]. To enhance the model's ability to generate SQL statements, we collect a subset of SQL corpora from the StarCoder's pre-training corpus [39]. The incorporation of SQL

corpora is predicated on the foundational role of SQL in PL/SQL, with PL/SQL representing an extension of SQL in procedural programming.

NL-related data [3.6GB]. To improve NL understanding capability of the model, we sample 3.6GB data from CODES [36], which is high-quality dialog data collected from Alpaca-cleaned[2], Unnatural-instructions [30] and UltraChat [14].

NL-to-code data [5.2GB]. To bridge the gap between NL descriptions and PL/SQL codes, we incorporate four types of NL-to-code datasets into our pre-training corpus: **(1) NL-to-PL/SQL** is a new dataset with 108K items derived from the PL/SQL data mentioned above. To improve the training efficiency, we exclude code segments over 1.5K characters, preventing overly long inputs from diluting the model's focus and enhancing its ability to identify and encode key information. Subsequently, we utilize GPT-3.5 to generate corresponding NL descriptions for each PL/SQL code. Additionally, based on the training templates, we employ the generation method described in Section 3.1 to generate a high-quality dataset of 32K entries. **(2) NL-to-SQL**. Given that SQL serves as the foundation for PL/SQL, we gather the NL-SQL-458K dataset from CODES, along with NL-to-SQL pairs sourced from both the CoNaLa [65] and StaQC [64] datasets. **(3) NL-to-C/C++**. We create this new dataset to enhance procedural language understanding for the PL/SQL generation. Since PL/SQL extends SQL with procedural features, training on C/C++ snippets helps models better capture these features. First, we collect NL-to-C/C++ snippets from XLCoST [70], a text-to-code dataset for seven programming languages. Second, we extract C/C++ snippets from GitHub Code[3] with regular pattern matching and generate corresponding NL descriptions via GPT-3.5. Third, we curate (problem, solution) pairs from description2code[4] which is built upon Codeforces submissions. To eliminate extraneous narrative elements from the problem descriptions, we use GPT-3.5 to extract concise summaries that capture the core algorithmic intent of each problem. **(4) Other types**. We also incorporate NL-to-Python and Jupyter-structured-clean-dedup compiled from CODES. For all the above data processing tasks using GPT-3.5, we respectively design three tailored examples for each, ensuring the examples align with the specific task objectives.

4 METHODOLOGIES

In this section, we first provide an overview of the proposed method, and then detail the training process of PLForge models, the prompt design tailored to the NL-to-PL/SQL task, and the application of PLForge in both ICL and SFT settings.

4.1 Method Overview

As depicted in Figure 3, our method for the NL-to-PL/SQL task encompasses four principal steps: incremental pre-training, prompt construction, PLForge model application, and PL/SQL correction.

4.1.1 Incremental Pre-training. We adopt an incremental pre-training approach on existing code language models to augment the proficiency of NL-to-PL/SQL in both code generation and NL comprehension. Specifically, we perform the mixed-task incremental pre-training using PL/SQL-centric datasets on StarCoder2 to obtain our pre-trained language model, denoted as PLForge-(3B, 7B, 15B).

4.1.2 Prompt Construction. We propose a prompt construction strategy tailored to the NL-to-PL/SQL task, consisting of four key components: (1) schema pruning that eliminates irrelevant tables and columns in the schema context, (2) enriched database metadata that improves the

[2]https://huggingface.co/datasets/yahma/alpaca-cleaned
[3]https://huggingface.co/datasets/codeparrot/github-code
[4]https://github.com/ethancaballero/description2code

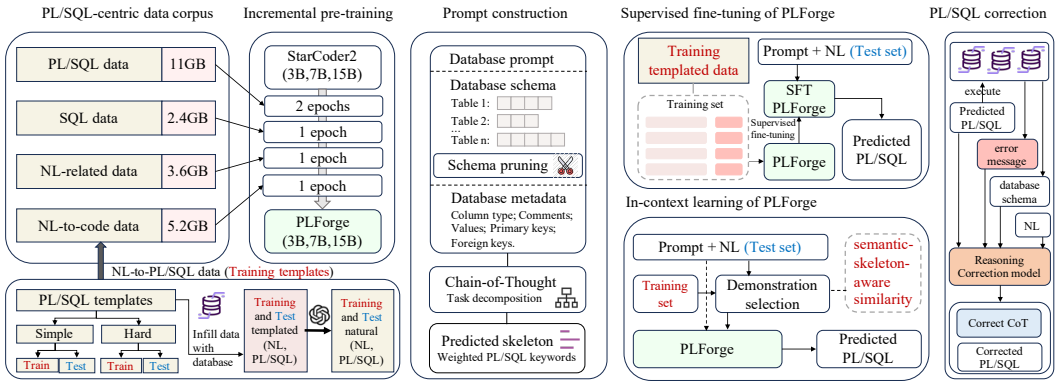


Fig. 3. Method overview.

database understanding, (3) PL/SQL chain-of-thought to guide step-wise code decomposition, and (4) process-oriented PL/SQL skeleton prediction as an intermediate representation to support the PL/SQL code generation.

4.1.3 PLForge Model Application. We detail the application of the PLForge model for generating PL/SQL code from NL description, employing both SFT and ICL approaches. SFT, ideal for scenarios with ample training data, enables the PLForge model to refine its predictive accuracy for PL/SQL code by learning from an annotated dataset. In contrast, ICL is employed under conditions of limited data annotation or resource constraints, where a small set of exemplar NL-to-PL/SQL demonstrations [54] guides the model's generation without the necessity for fine-tuning. To enhance the effectiveness of ICL, we propose a novel demonstration selection strategy that identifies examples based on semantic and skeleton-level similarity to the input. This method significantly improves the generation accuracy for PL/SQL code.

4.1.4 PL/SQL Error-aware Reasoning Correction. The PL/SQL Error-aware Reasoning Correction module is built upon a reasoning-driven model that automatically corrects errors in generated PL/SQL code. The model leverages four contextual inputs, namely the database schema, the NL description, the generated PL/SQL code containing errors, and the corresponding database error messages. It generates two complementary outputs: a correction reasoning chain that articulates the diagnostic and correction process, and the corrected PL/SQL code. This design enhances both the interpretability and reliability of the NL-to-PL/SQL translation.

4.2 Incremental Pre-training

PLForge-(3B, 7B, 15B) is incrementally pre-trained based on StarCoder2 models [46], a family of advanced open-source code language models. To effectively adapt these models to the NL-to-PL/SQL task, we propose two key innovations in the pre-training process: a comment-enriched PL/SQL corpus and a differentiated mixed-task training schedule.

Comment-Enriched PL/SQL Corpus. We augment the collected real-world PL/SQL and PL/pgSQL codebase with NL comments heuristically generated using GPT-3.5. These comments include both inline annotations ("--") and block-level summaries ("/∗ · · · ∗/"), inserted at semantically meaningful locations throughout the code.

By embedding comments directly into procedural scripts, the model is exposed to aligned code and NL content during the pre-training. These comments serve as lightweight semantic signals,

providing the model with both high-level intent and fine-grained procedural cues. This structure-aware semantic enrichment enables the model to better capture the correspondence between NL and PL/SQL code, thereby enhancing its reasoning capability and transferability for the NL-to-PL/SQL task.

PL/SQL-centric Differentiated Mixed-Task Training. To balance domain specialization in PL/SQL with generalization across NL and programming code, we adopt a differentiated mixed-task pre-training strategy. Specifically, we incrementally pre-train StarCoder2 for two epochs on PL/SQL data, versus one epoch on SQL, NL-related, and NL-to-code data. This asymmetric schedule assigns higher frequency to PL/SQL data, thereby prioritizing the target procedural semantics. This differentiated mixed training strategy enhances PLForge's specialization in PL/SQL understanding while maintaining broad competence across NL and code tasks.

The training procedure for PLForge adheres to the auto-regressive decoding approach, in which the model generates tokens sequentially, with each token's generation conditioned on all previously generated tokens. This approach employs a scoring function $p(e \mid l)$ that factorizes the probability of generating the output sequence by progressively conditioning from left to right,

$$p(e \mid l) = \prod_{j=1}^{m} p\left(e_j \mid e_{<j}, l\right), \tag{1}$$

where $l = (l_1, \cdots, l_n)$ denotes the input sequence and $e = (e_1, \cdots, e_m)$ denotes the output sequence. The training objective is to maximize the likelihood of the output sequence e given the input sequence l.

To optimize the training procedure for the incremental pre-training of PLForge under the differentiated mixed training strategy, we adopt a set of training configurations and optimization hyperparameters. We show more details about the architectural details of PLForge models in Appendix Table 1 [3].

Optimizer and Hyperparameters. We utilize the AdamW optimizer [45] with $\beta_1 = 0.9$, $\beta_2 = 0.95$, and $\epsilon = 10^{-8}$. The learning rate is initialized at 5×10^{-5}, coupled with a weight decay of 0.1. A cosine decay schedule is employed to progressively reduce the learning rate to 5×10^{-6} without any warm-up steps. Due to GPU memory limitations, the batch size is set to 128 for PLForge-(3B, 7B) and 64 for PLForge-15B.

Memory Optimization. We adopt DeepSpeed ZeRO-2 framework [53] with BF16 mixed precision and FlashAttention-2 [12] to improve GPU memory efficiency during incremental pre-training.

Training Resources and Duration. The incremental pre-training is conducted on eight NVIDIA A800 (80GB) GPUs, completing in approximately 3.5, 8.5, and 18 days for the 3B, 7B, and 15B variants, respectively.

PLForge-(3B,7B,15B) are designed for different computational and deployment needs. The PLForge-3B model offers fast, resource-efficient inference with decent NL-to-PL/SQL performance but shows weaker generalization. The PLForge-7B model strikes a good balance between performance and efficiency, improving over the 3B model. The PLForge-15B model delivers the best accuracy and generalization but demands substantially more hardware resources.

4.3 Prompt Construction

In addition to constructing an advanced model tailored for PL/SQL through incremental pre-training, the prompts inputted into the model are also crucial. Specifically, high-quality prompts enrich the model with valuable information, thereby more precisely directing correct PL/SQL code. To this end, we carefully craft a set of prompts designed to guide the generation of PL/SQL code,

encompassing database schema pruning, integration of database metadata, guidance with chain-of thought, and the incorporation of a predictive skeleton for PL/SQL code.

4.3.1 Schema Pruning. Schema pruning eliminates irrelevant tables and columns for the target PL/SQL code, offering two key advantages: (1) reducing the input length to accommodate more demonstrations within token limits, and (2) simplifying the inference by focusing on a relevant schema subset. Following [35], we employ a schema pruning classifier that takes the serialized sequence of the NL description x and the database schema $db_s = \langle t, c \rangle$ as input, where $t = \left\{t_1, \cdots, t_{|T|}\right\}$ represents the set of tables, with $|T|$ indicating the total number of tables, and $c = \bigcup_{i=1}^{|T|} c_i$ represents the set of all columns in the database, with $c_i = \left\{c_i^1, \cdots, c_i^{|C_i|}\right\}$ being the set of columns in table t_i, where $|C_i|$ indicating the total number of columns in t_i. The classifier outputs the relevance probability of each t_i and c_i with respect to x. We train the classifier using the focal loss [41]. In the inference stage, we select several tables and columns with the highest relevance probabilities, composing the database schema part in the prompt.

4.3.2 Database Metadata. We incorporate valuable database metadata to further improve the accuracy of PL/SQL. First, we consider column data types as the governing factors that determine validation rules, permissible operations, and variable declarations. This specification ensures accurate type matching for parameters and variables by imposing unique computational rules and helps imply potential type conversions for certain operations such as the `CAST` function. Second, we supplement table and column names with descriptive comments to enhance schema comprehension, as database identifiers are often abbreviated and ambiguous. This semantic context reduces schema linking errors and enables more logically consistent code generation. Third, we include two distinct non-null representative values sampled for per column to help the model infer data formats, validation rules, and necessary conversions, thereby improving the precision of generated PL/SQL code. Finally, we incorporate primary key-foreign key relationships to help the model identify inter-table associations and infer proper `JOIN` clauses by revealing the database's semantic structure.

4.3.3 PL/SQL Chain-of-Thought. Unlike declarative SQL generation, PL/SQL involves multistep procedural logic, including procedure declaration, parameter determination, variable declarations, code framing, and control structures. Directly generating such code from NL often results in syntactic errors or incomplete logic due to its hierarchical and imperative structure.

To handle the complexity and hierarchical nature of PL/SQL code generation, we propose a PL/SQL step-wise chain-of-thought (CoT) prompting strategy specifically tailored to the procedural characteristics of PL/SQL. This strategy guides the model to generate code in a structured and reasoning-driven manner by integrating syntactic scoping and control flow awareness into each step of the generation process.

Five-Stage CoT Decomposition. We define a five-stage CoT decomposition that mirrors the canonical structure of PL/SQL procedures:

(1) **Procedure Declaration**: Determine the appropriate top-level construct, such as `PROCEDURE`, `FUNCTION`, or `TRIGGER`, based on the task's semantics (e.g., `CREATE OR REPLACE PROCEDURE`).
(2) **Parameter Determination**: Define the formal parameter list, specifying data types (e.g., `INTEGER`, `TEXT`) and parameter modes (e.g., `IN`, `OUT`).
(3) **Variable Declaration**: Declare local variables used within the procedure body, including appropriate types and optional default values.
(4) **Code Framing**: Introduce the main execution block using `BEGIN` and `END` constructs, ensuring proper scoping and nesting.

(5) **Internal Logic Design**: Incorporate control structures such as `IF`, `LOOP`, and `CURSOR`, using the predicted skeleton to guide coherent and semantically grounded procedure construction.

This step-wise CoT strategy explicitly encodes the hierarchical dependencies of procedural code, enabling the model to incrementally reason about declaration structure, control logic, and execution flow. In contrast to prior CoT approaches in Text-to-SQL, which typically focus on clause selection or filtering logic, our method captures the imperative and block-structured nature of PL/SQL. This design leads to significant improvements in both syntactic correctness and structural completeness.

Dialect-Agnostic Applicability. This CoT prompting strategy is not tied to any dialect-specific syntax. Instead, it abstracts over the shared structure of procedural SQL languages. Core constructs, such as procedure declaration, parameter determination, variable block, and code logic design, are consistently present across PL/SQL dialects including Oracle PL/SQL, PL/pgSQL, and so on. Thus, this approach generalizes well with minimal adaptation required for other dialects.

4.3.4 PL/SQL skeleton prediction. Existing studies in Text-to-SQL often leverage SQL skeletons as intermediate structures to constrain the decoding space and improve syntactic correctness [17, 54]. However, applying this concept directly to PL/SQL is insufficient, as PL/SQL code involves not only declarative SQL clauses but also procedural constructs such as parameter determination, variable declaration, and control flow statements.

To address this challenge, we propose a process-oriented PL/SQL skeleton that captures not only declarative SQL elements but also the procedural structure of executable logic. The skeleton is designed to represent the hierarchical and sequential nature of PL/SQL programs, making it more suitable for guiding code generation in procedural contexts.

We implement a skeleton predictor based on a pre-trained T5 model. Given an NL input x, the model predicts a structured PL/SQL skeleton s token by token. To enhance the predictor's focus on critical structural components, we propose a weighted training loss. This loss function increases the importance of correctly predicting PL/SQL-specific keywords such as control flow constructs (e.g., `IF`, `ELSE`, `LOOP`), declarations (e.g., `DECLARE`), and procedural boundaries (e.g., `BEGIN`, `END`).

Let V be the token vocabulary used in the model. Let K be the set of keywords in PL/SQL. Note that K is a subset of V. For each PL/SQL keyword $u \in K$, we denote by v_u its assigned weight. We define the weight assignment function $D : V \to \mathbb{R}_{>0}$ by

$$D(u) = \begin{cases} v_u, & u \in K, \\ 1, & u \notin K, \end{cases} \qquad \{v_u\}_{u \in K} \subset \mathbb{R}_{>0}. \tag{2}$$

For each training example with NL input x and target skeleton sequence $s = (s_1, \ldots, s_{|s|})$, we define the weighted loss as:

$$L = -\sum_{i=1}^{|s|} w_i \log P\big(s_i \mid s_{<i}, x\big), \quad w_i := D(s_i). \tag{3}$$

By assigning greater weights to key PL/SQL tokens, the model is steered to prioritize structural accuracy during training, thereby enhancing the reliability of the generated skeleton. This predicted skeleton is then integrated into the prompt used for PL/SQL code generation, providing structural guidance that helps the model better organize procedural logic and preserve syntactic correctness. To the best of our knowledge, this work is the first to propose a weighted skeleton prediction mechanism specifically designed for the procedural characteristics of PL/SQL. This approach facilitates a more effective mapping between NL description and PL/SQL code, allowing the model to more precisely capture the syntactic and control flow features unique to PL/SQL.

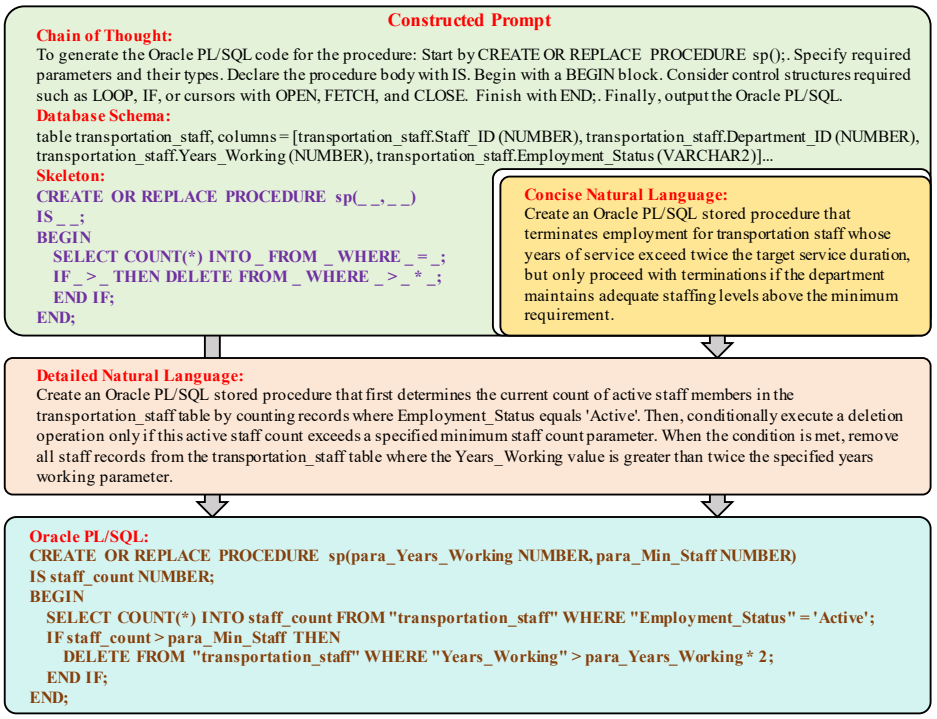


Fig. 4. Constructed prompt, concise NL, detailed NL and Oracle PL/SQL for simple case.

As shown in Figures 4 and 5, we present Oracle simple and PostgreSQL hard data examples produced with our prompt construction strategy designed for the NL-to-PL/SQL task.

4.4 Model Application

In this subsection, we present the application of the PLForge model in both SFT and ICL scenarios. In addition, we propose a novel semantic-skeleton-aware similarity metric, which jointly considers semantic and skeleton similarities to more effectively identify valuable demonstrations for the NL-to-PL/SQL task.

We assume that the training set $D = \{d_1, d_2, \cdots, d_n\}$ with n NL-to-PL/SQL items, where each item d_i consists of a triplet $(d_i^{db}, d_i^x, d_i^{pl})$, representing the database, the PL/SQL description, and the corresponding PL/SQL code, respectively. The test set $T = \{t_1, t_2, \cdots, t_m\}$, where each $t_i = (t_i^{db}, t_i^x)$ consists of the database t_i^{db} and the PL/SQL NL description t_i^x. The model's objective is to generate the corresponding PL/SQL code t_i^{pl}. Prior to applying PLForge, we construct prompts for both training and test items as described in Section 4.3. Formally, each training item d_i is represented as a triplet $(d_i^{dbp}, d_i^x, d_i^{pl})$, while each test item t_i is represented as a pair (t_i^{dbp}, t_i^x), where d_i^{dbp} and t_i^{dbp} denote the constructed prompts.

4.4.1 Supervised Fine-Tuning. For effective adaptation to a specific dataset, SFT is a highly efficient approach, leveraging collections of training data. In this process, each input sequence is constructed by concatenating the prompt with its corresponding NL description. Subsequently, the PLForge model is fine-tuned to generate the corresponding PL/SQL code based on the input sequence. The

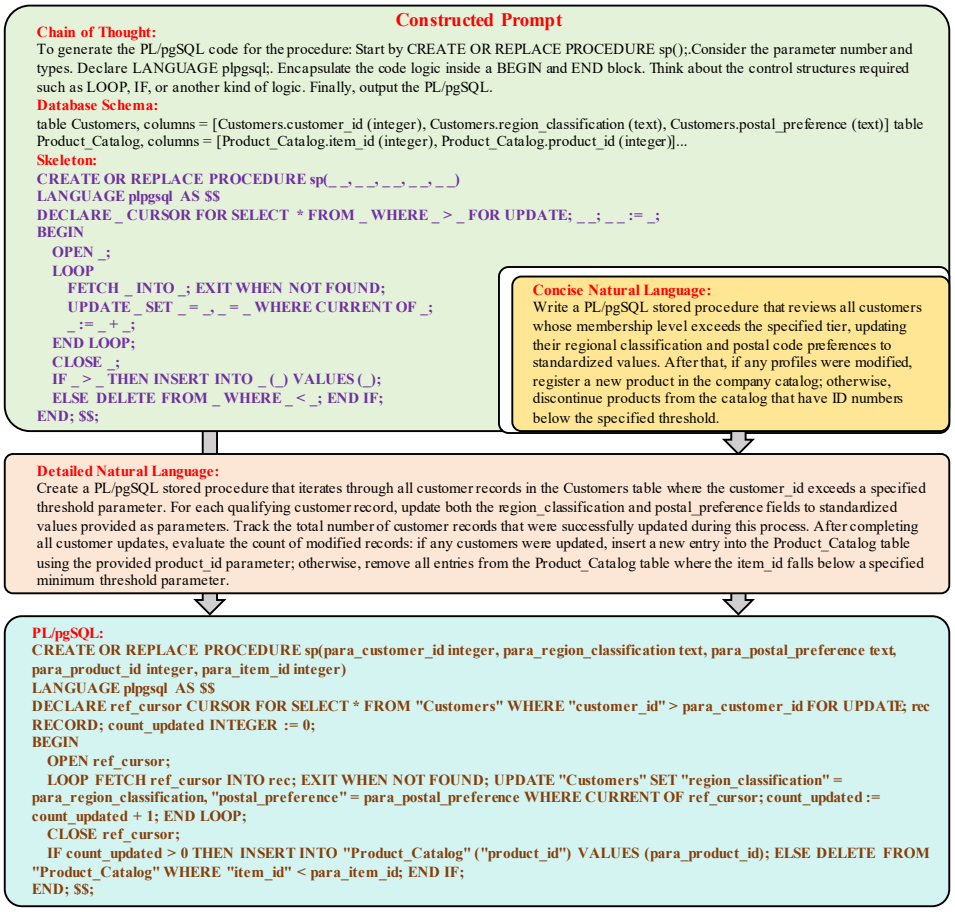


Fig. 5. Constructed prompt, concise NL, detailed NL and PL/pgSQL for hard case.

goal of SFT is to improve the model's ability to accurately predict PL/SQL code based on the given input. The objective of SFT PLForge can be formalized as:

$$L_{sft} = \frac{1}{n}\sum_{i=1}^{n} p(d_i^{pl} | d_i^{dbp}, d_i^{x}). \tag{4}$$

During the inference, given a test item, the SFT PLForge can generate the corresponding PL/SQL code based on the input sequence composed of t^{dbp} and t^{x}.

4.4.2 In-Context Learning. When the fine-tuning is impractical, ICL serves as an effective alternative by leveraging the model's built-in NL-to-PL/SQL capabilities through task-specific examples. However, the performance of ICL is highly dependent on the relevance of the chosen examples [62, 67]. To ensure the high relevance, we select top-k most similar samples from dataset D based on a predefined similarity metric.

CODES [36] introduces a similarity metric termed "question-pattern-aware similarity" (abbr. Sim_{qp}), which avoids overemphasizing the named entities by stripping them from the NL description. This design allows the metric to focus on the core structural features of the question. Formally,

Sim_{qp} between an NL description t^x and a description d_i^x within the training set can be defined as:

$$Sim_{sem}(t^x, d_i^x) = SimCSE(t^{xs}, d_i^{xs}), \tag{5}$$

$$Sim_{qp}(t^x, d_i^x) = \max(SimCSE(t^x, d_i^x), Sim_{sem}(t^x, d_i^x)), \tag{6}$$

where t^{xs} and d_i^{xs} denote the extracted core structures of t^x and d_i^x, respectively. We then employ SimCSE [24], a sentence similarity model, to compute the semantic similarity between t^x and d_i^x.

However, the question-pattern-aware similarity focuses only on the semantic similarity of abstracted NL patterns, thus overlooking the intrinsic syntactic and semantic relationships within the code itself. These relationships are essential for guiding models to generate accurate outputs. To address this limitation, we propose a novel similarity metric, termed "semantic-skeleton-aware similarity", which jointly accounts for semantic similarity in NL and skeleton similarity in code, thereby providing a more comprehensive and effective demonstration selection method.

Skeleton similarity. The skeleton similarity is designed to capture the relevance between the predicted PL/SQL skeleton from the test NL description and PL/SQL skeletons within the training set. By incorporating skeleton similarity, it becomes feasible to identify training examples that share structural patterns of PL/SQL skeletons with the predicted PL/SQL skeleton corresponding to the test NL description. Selecting such structurally similar examples can enhance the model's understanding of correct and relevant PL/SQL structures, thereby guiding it to generate more accurate PL/SQL code.

For each item d_i in the training set, we extract the skeleton d_i^{skt} from its corresponding PL/SQL code d_i^{pl}. For a given test item t, we denote the PL/SQL skeleton predicted by the skeleton predictor as t^{skt}. We formalize the skeleton similarity between t^{skt} and d_i^{skt} in the training set as:

$$Sim_{skt}(t^{skt}, d_i^{skt}) = TreeDist(AST(t^{skt}), AST(d_i^{skt})), \tag{7}$$

where $AST(\cdot)$ denotes the abstract syntax tree (AST) of PL/SQL skeleton and the $TreeDist(\cdot, \cdot)$ represents the tree edit distance between AST of t^{skt} and AST of d_i^{skt}.

Semantic-skeleton-aware similarity. Upon the semantic and skeleton similarity, we construct a formula to quantify the similarity metric between the test item t and the training item d_i as:

$$Sim_{ss}(t, d_i) = w \cdot \hat{Sim}_{sem}(t^x, d_i^x) + (1 - w) \cdot \hat{Sim}_{skt}(t^{skt}, d_i^{skt})$$

where $w \in [0, 1]$ serves as a weighting factor that balances the influence between semantic similarity and skeleton similarity. Prior to the aggregation, both similarity scores are independently normalized to the same scale.

Finally, the top-k most relevant demonstrations are selected based on the semantic-skeleton-aware similarity. These demonstrations are then merged with the test item to construct a consolidated prompt, which is subsequently fed into the model to generate the corresponding PL/SQL code. The overall demonstration selection algorithm is presented in Appendix Algorithm 1 [3].

Although similar samples may not always be available at the full procedure level in real-world scenarios, our similarity metric operates on abstracted NL semantics and PL/SQL structural skeletons, enabling effective retrieval of relevant examples even when overall procedures differ. Combined with our proposed NL-to-PL/SQL dataset generation method, which produces a diverse set of procedural data covering various logic types (Section 3.1), our approach robustly supports few-shot ICL without relying on the existence of procedure-level similar samples in real-world codebases.

4.5 PL/SQL Error-aware Reasoning Correction

Although PLForge performs well in translating NL into PL/SQL, there can still be occasional syntax errors during generation. In such cases, the issues can be identified by executing the PL/SQL on the database. These errors can typically be surfaced by executing the generated code against a

database, but immediate and intelligent PL/SQL correction remains a challenge. To address this gap, we propose PL/SQL error-aware reasoning correction, a novel correction module that not only fixes erroneous code but also provides a transparent reasoning chain that explains the correction process. It enhances both the interpretability and reliability of the NL-to-PL/SQL translation.

The core of this module is a reasoning-driven correction model. It takes as input a rich error context, including (1) database schema, (2) NL description, (3) generated (but erroneous) PL/SQL code, and (4) the exact database error messages indicating the type and location of the error. The output of the model consists of two parts: (1) a step-by-step correction reasoning chain, which performs error localization, cause diagnosis, and repair strategy explanation; (2) the corrected PL/SQL code, refined through the reasoning process.

To train this correction module, we first construct a dedicated NL-to-PL/SQL reasoning error correction dataset. We use the PLForge-3B model to generate initial outputs on the training set of both simple and hard questions. All samples that result in translation errors are collected and restructured using the input-output format mentioned above. Then, we leverage GPT-4o to annotate each erroneous instance with a structured correction reasoning chain, ensuring consistency in error analysis and resolution logic. Finally, we fine-tune the StarCoder2 model via SFT on this dataset, resulting in a PL/SQL correction model with enhanced error-aware reasoning capabilities.

5 EXPERIMENTS

This section presents the experimental setup, followed by evaluations of PLForge in ICL and SFT, the analysis of key hyperparameters, and ablation studies to validate effectiveness. The relevant artifacts are available on GitHub[5].

5.1 Experimental Setup

5.1.1 Datasets. In this experiment, we use four datasets: Simple, Hard, ProcBench, and WeBridge. The statistics of these datasets are shown in Table 2. Below is an introduction to each dataset.

Simple and **Hard**. These two datasets are generated from templates. We apply the template dataset generation method described in Section 3.1, generating a total of 220 templates. Additionally, we engage eight database experts to manually create 100 templates. The full set consists of 320 templates, divided into 150 simple templates and 170 hard templates based on the complexity of the corresponding PL/SQL code. We define "simple" and "hard" templates based on the number of statements in the PL/SQL code. Templates with fewer than three statements are considered simple, while those with three or more statements are classified as hard. The simple templates involve operations on a single table, while the hard templates involve operations across one to three tables. To populate these template sets and the corresponding databases, we select 166 databases from the Spider dataset [66], which covers over 100 unique domains. Finally, we construct two versions of datasets for both PostgreSQL and Oracle, denoted as $\text{Simple}_{\text{pg}}$, $\text{Simple}_{\text{oc}}$, Hard_{pg}, and Hard_{oc}.

For each Simple dataset, we randomly split the 150 simple templates into 100 for the training set and 50 for the test set, ensuring that the training and test templates are mutually exclusive. Using the templates from the training set and the 166 databases, we generate 1,500 samples. Similarly, using the templates from the test set, we generate 300 samples. For each Hard dataset, we randomly divide the 170 hard templates into 114 for the training set and 56 for the test set, maintaining mutual exclusivity between the training and test templates. We then generate 1,500 samples for the training set and 300 samples for the test set, following the same method.

ProcBench. This dataset is derived from ProcBench [28], a PL/SQL benchmark suite meticulously designed through in-depth analysis of real-world workloads. To facilitate the evaluation,

[5]https://github.com/ZhanGHanG9991/plforge

Table 2. Details of Simple, Hard, ProcBench, WeBridge and WeBridgeL datasets.

	PL/pgSQL	Oracle PL/SQL	Train (Demo) / Test set size	Avg. length of detailed NL description	Avg. length of PL/SQL code	Avg. number of statements	Avg. lines of code	Avg. number of parameter	Data retrieval (%)	Database operation (%)	Control flow (%)	Batch processing (%)
Simple	$\text{Simple}_{\text{pg}}$	$\text{Simple}_{\text{oc}}$	1500 / 300	191.6	261.7	3.8	9.3	2.1	27.3	100.0	14.7	0.0
Hard	Hard_{pg}	Hard_{oc}	1500 / 300	498.4	579.6	11.7	14.5	3.9	94.7	100.0	30.7	73.7
ProcBench	$\text{ProcBench}_{\text{pg}}$	$\text{ProcBench}_{\text{oc}}$	1500 / 300	217.7	304.8	4.3	11.3	2.5	55.0	97.3	15.3	13.0
WeBridge	$\text{WeBridge}_{\text{pg}}$	-	1500 / 6400	640.6	776.8	13.3	22.8	3.4	49.2	97.8	25.4	25.3
WeBridgeL	$\text{WeBridgeL}_{\text{pg}}$	-	1500 / 209	1580.5	1675.5	30.6	44.7	4.6	45.9	100.0	62.2	37.3

we construct two parallel sets of 35 templates each: one implemented in PL/pgSQL, denoted as $\text{ProcBench}_{\text{pg}}$, and the other in Oracle PL/SQL, denoted as $\text{ProcBench}_{\text{oc}}$. Using the same database configuration and randomization procedure as in the original benchmark, we generate 300 samples for the test set in each variant. Furthermore, from the training set which contains 1,500 samples for each complexity level (simple and hard), we randomly select 750 samples from each dataset to compose the demonstration set used in the few-shot experiments.

WeBridge. The WeBridge dataset [31] consists of 6,400 PL/pgSQL procedures that are automatically generated from six real-world Java Web applications. Based on these procedures, we construct two NL-to-PL/SQL datasets. We refer to the complete dataset as $\text{WeBridge}_{\text{pg}}$. To evaluate the model's ability to generate very long procedures, which are characterized by a large number of sequential statements, we extract a subset of 209 particularly lengthy procedures from $\text{WeBridge}_{\text{pg}}$, denoted as $\text{WeBridgeL}_{\text{pg}}$. On average, each procedure in this subset contains 30.55 statements, making it appropriate for assessing the model's capacity to handle complex and extended code generation tasks. Furthermore, we use the training set of our constructed Hard_{pg} dataset as the demonstration pool for few-shot experiments conducted on both WeBridge datasets.

5.1.2 Evaluation Metrics. We propose two evaluation metrics designed specifically for the NL-to-PL/SQL task to assess the performance of different approaches: PL/SQL execution match (EX) accuracy and PL/SQL exact match (EM) accuracy.

EX accuracy measures the equivalence between the generated and ground-truth PL/SQL scripts. The evaluation involves executing each PL/SQL script k times with different parameter sets in the same database state and checking for the consistency in table information via `SELECT` statements. The consistency check includes all data tables in the database, as well as eight key system tables such as indexes, constraints, views, and rules, in order to comprehensively assess the potential side effects of the PL/SQL code. If all k executions yield identical outputs, the scripts are considered equivalent. The parameter sets are crafted based on the ground-truth PL/SQL and its associated database data. EX is formally defined as:

$$\text{EX} = \frac{1}{m}\sum_{j=1}^{m}\mathbb{I}\left(\bigwedge_{i=1}^{k} S^{j}_{\text{pd}}(p_i) = S^{j}_{\text{gt}}(p_i)\right), \tag{8}$$

where m is the size of the test set, $S^{j}_{\text{pd}}(p_i)$ is the state of the database after executing the j-th generated PL/SQL script with the i-th parameter set, and $S^{j}_{\text{gt}}(p_i)$ is the corresponding ground-truth state. We set $k = 5$ by default.

EM accuracy evaluates the match between predicted and ground-truth PL/SQL scripts by comparing their structural equivalence through AST representations. It accounts for minor syntactic variations, such as differences in variable names, ensuring these do not influence the comparison. A prediction is considered correct if the AST of the predicted script is structurally equivalent to the

AST of the ground-truth script. EM accuracy is given by:

$$\text{EM} = \frac{1}{m}\sum_{j=1}^{m}\mathbb{I}\left(\text{AST}(t_j^{pdc}) = \text{AST}(t_j^{gtc})\right), \tag{9}$$

where $\text{AST}(t_j^{pdc})$ and $\text{AST}(t_j^{gtc})$ are the AST representations of the predicted and ground-truth PL/SQL scripts, respectively.

5.1.3 Baselines. Since the NL-to-PL/SQL task proposed in this paper is novel and lacks existing methods for the direct comparison, we select 12 state-of-the-art language models from three representative domains: Text-to-SQL, NL-to-Code, and general-purpose LLMs such as the GPT series, all of which demonstrate the leading performance within their respective fields.

Text-to-SQL language models. CODES is a fully open-source language model meticulously designed for the Text-to-SQL domain. This model demonstrates the enhanced performance on established Text-to-SQL benchmarks, such as Spider [66] and Bird [38].

Code language models. We incorporate five widely recognized open-source code language models, such as StarCoder2-(3B,7B,15B) [46], StarCoder [39], and CodeLlama [55], all of which represent the current state of the art in code generation. These models have been pre-trained on large-scale multilingual code datasets, which encompass a substantial amount of SQL data, thereby endowing them with the capability to generate PL/SQL.

General language models. To ensure a comprehensive evaluation, we include four general-purpose language models, including the recently released Llama-3.1-8B and the current strong reasoning models o3-mini, GPT-4.1 and Gemini 2.5 Pro. These models excel in processing NL and generating text with human-like quality, suitable for the NL-to-PL/SQL task.

5.1.4 Implementation Details. We migrate all 166 databases from the Spider dataset to both PostgreSQL and Oracle, which serve as the target database environments for our experiments. For the schema pruning module, we follow the training and evaluation procedures outlined in RESDSQL [35], configuring the system to retain 6 tables and 10 columns relevant to the NL descriptions. The weighting factor w utilized in semantic-skeleton-aware similarity is set to 0.5, with further analysis in Section 5.5. We fine-tune a T5-base model as a skeleton predictor. For the relatively simple Simple dataset, the weight is set to 2, while for the more complex Hard and ProcBench datasets, it is set to 4. For the PLForge fine-tuning process, we adhere to the optimization strategy described in Section 4.2. We fine-tune PLForge and baseline models for 2 epochs using a batch size of 128, setting the maximum context length to 8,192. Due to GPU memory limitations, we use a batch size of 64 for all models with 15B parameters. For the PL/SQL correction task, we fine-tune the StarCoder2-3B model using a batch size of 128 for 4 epochs.

5.1.5 Environments. The experiments are performed utilizing PyTorch 1.13.1 on a server equipped with an Intel(R) Xeon(R) Platinum 8352Y CPU @ 2.20GHz (32 cores), eight NVIDIA A800 (80GB) GPUs, 256GB of RAM, and Ubuntu 20.04.5 LTS operating system.

5.2 Evaluation on In-Context Learning

To ensure a fair comparison across the board, all models utilize the prompt construction strategy proposed in Section 4.3, implying that they use the same demonstrations, database schema, metadata, chain-of-thought, and predicted skeletons. The PLForge models are built upon StarCoder2 via incremental pre-training, and notably, they are not subjected to any SFT. We first assess the EX and EM of each model under zero-shot conditions. The evaluation is conducted on all datasets under PostgreSQL and Oracle. As shown in Tables 3 and 4, PLForge exhibits a clear and consistent advantage over all baseline models across all datasets. Among all variants, PLForge-15B achieves the

Table 3. In-context learning performance on datasets under PostgreSQL using zero-shot setting. The best performance is highlighted in bold, while the runner-up is underlined. All values are in percentage (%).

Language models	$\text{Simple}_{\text{pg}}$		Hard_{pg}		$\text{ProcBench}_{\text{pg}}$		$\text{WeBridge}_{\text{pg}}$		$\text{WeBridgeL}_{\text{pg}}$	
	EX	EM	EX	EM	EX	EM	EX	EM	EX	EM
CODES-3B	51.00	29.33	8.67	3.00	43.00	35.33	25.02	6.77	4.78	1.44
CODES-7B	54.00	29.00	13.67	1.33	36.00	33.00	27.84	4.61	20.57	3.83
CODES-15B	42.33	22.00	20.00	2.67	34.00	28.67	34.34	10.91	24.88	10.05
StarCoder2-3B	31.33	17.33	2.67	0.67	21.33	15.33	15.05	3.67	0.00	0.00
StarCoder2-7B	35.67	18.67	9.33	2.33	26.67	17.67	20.64	6.63	0.48	0.00
StarCoder2-15B	25.67	12.00	8.67	1.67	22.00	14.33	12.39	3.27	0.00	0.00
Llama-3.1-8B	14.33	4.00	4.00	1.00	5.00	0.67	4.58	2.38	5.94	2.19
CodeLlama-13B	34.00	21.67	18.67	2.00	18.00	13.00	18.06	3.31	17.81	3.13
StarCoder-15B	9.67	4.67	2.33	0.00	5.33	7.33	6.22	1.05	5.94	0.94
o3-mini	70.67	44.00	23.67	4.67	66.67	41.67	37.13	15.81	9.09	2.39
GPT-4.1	69.33	44.00	44.00	16.67	68.67	43.33	47.02	16.69	26.32	11.48
Gemini 2.5 Pro	66.67	45.00	49.33	19.00	60.00	41.67	54.44	22.30	41.15	21.53
PLForge-3B	83.00	61.00	59.00	43.00	66.00	43.33	61.05	26.17	44.98	16.27
PLForge-7B	84.67	61.00	54.00	39.00	72.33	50.33	62.58	27.06	45.93	21.53
PLForge-15B	**88.00**	**66.33**	**75.67**	**54.67**	**79.00**	**54.33**	**64.61**	**31.38**	**53.11**	**26.32**

Table 4. In-context learning performance on datasets under Oracle using zero-shot setting.

Language models	$\text{Simple}_{\text{oc}}$		Hard_{oc}		$\text{ProcBench}_{\text{oc}}$	
	EX	EM	EX	EM	EX	EM
CODES-3B	37.33	15.00	12.33	6.33	28.00	22.67
CODES-7B	39.00	18.33	18.00	7.67	30.33	19.33
CODES-15B	39.00	23.67	22.67	11.67	38.00	30.67
StarCoder2-3B	40.00	9.67	6.33	1.00	31.33	12.67
StarCoder2-7B	35.33	12.67	13.67	5.33	27.67	9.33
StarCoder2-15B	18.67	11.00	18.33	7.00	21.00	6.33
Llama-3.1-8B	16.67	6.67	7.33	1.33	28.33	12.67
CodeLlama-13B	40.00	19.67	13.67	3.67	25.67	19.33
StarCoder-15B	15.33	12.00	5.67	0.67	17.00	9.00
o3-mini	50.67	12.67	18.67	4.67	31.67	15.00
GPT-4.1	74.67	17.33	42.67	12.67	46.33	23.33
Gemini 2.5 Pro	71.00	15.33	57.67	16.33	51.67	32.67
PLForge-3B	86.67	65.00	63.00	53.33	60.33	54.00
PLForge-7B	83.00	62.67	61.00	50.67	61.33	55.33
PLForge-15B	**90.00**	**72.00**	**79.33**	**59.00**	**66.33**	**56.33**

best overall performance, attaining an average EX accuracy of 74.51% and EM accuracy of 52.54%. Notably, PLForge-7B ranks as runner-up in the majority of scenarios. Furthermore, when tested on the more realistic ProcBench and WeBridge datasets, PLForge continues to deliver the strong and stable performance. Note that as the model size increases, both the EX and EM accuracies improve, indicating the enhanced generalization with larger parameter scales.

We analyze the zero-shot setting across Simple, Hard, and ProcBench datasets, categorizing errors into: (1) parameter errors (45.96%), such as incorrect input types or counts; (2) syntax/semantic errors (19.19%), including invalid syntax or misuse of PL/SQL constructs; (3) logical errors (32.83%), where code is valid but yields incorrect results; (4) constraint violations (2.02%), where generated DML statements breach database constraints.

Tables 5 and 6 display the results of (1,3,5)-shot ICL evaluations for PostgreSQL. The results for Oracle are available in Appendix Table 2 [3]. A comparative analysis between the incrementally trained PLForge and the original model StarCoder2, reveals that all PLForge-(3B, 7B, 15B) consistently outperform StarCoder2-(3B, 7B, 15B). This consistent improvement underscores the effectiveness of the incremental pre-training on PL/SQL-centric corpus. It is noteworthy that the StarCoder2-15B model also performs robustly in this task, attributable to its pre-training corpus

Table 5. In-context learning performance on $\text{Simple}_{\text{pg}}$ and Hard_{pg} using (1,3,5)-shot settings.

Language models	$\text{Simple}_{\text{pg}}$						Hard_{pg}					
	1-shot		3-shot		5-shot		1-shot		3-shot		5-shot	
	EX	EM	EX	EM	EX	EM	EX	EM	EX	EM	EX	EM
CODES-3B	75.33	52.00	77.00	52.67	78.33	52.67	37.67	25.00	34.67	23.67	33.67	23.33
CODES-7B	76.33	55.00	76.00	52.33	77.67	54.00	39.00	24.00	38.67	24.67	40.67	26.00
CODES-15B	76.33	51.33	75.67	48.00	79.33	50.67	45.00	26.67	47.67	28.33	49.00	29.67
StarCoder2-3B	76.00	55.67	75.00	49.00	75.00	46.00	40.00	23.67	38.33	23.67	40.67	22.00
StarCoder2-7B	81.33	56.33	79.00	53.33	77.67	51.33	49.67	26.33	48.00	24.33	44.67	23.33
StarCoder2-15B	81.33	58.67	78.33	52.67	79.00	53.67	59.00	32.67	59.67	28.67	58.00	27.00
Llama-3.1-8B	81.33	59.00	80.67	52.00	80.00	51.00	48.00	28.67	46.00	26.33	50.67	28.33
CodeLlama-13B	78.67	57.67	79.33	54.67	78.67	51.67	46.67	27.33	46.67	23.33	48.00	25.67
StarCoder-15B	72.67	50.67	80.33	52.33	78.67	53.33	44.67	26.33	41.67	22.67	45.67	24.67
o3-mini	75.33	50.67	77.33	54.67	79.33	55.00	35.33	17.33	45.00	26.33	47.33	28.00
GPT-4.1	75.33	52.33	80.67	55.67	83.00	57.33	49.33	30.00	56.33	32.00	56.67	32.33
Gemini 2.5 Pro	78.33	53.67	82.00	60.00	84.67	58.67	61.67	30.67	68.67	34.00	68.33	33.00
PLForge-3B	83.33	60.00	83.00	59.00	82.00	58.67	62.33	44.33	62.67	45.33	63.33	46.67
PLForge-7B	86.67	62.33	87.00	62.33	86.67	62.00	57.67	44.00	56.00	42.67	56.33	43.00
PLForge-15B	**87.00**	**66.00**	**87.67**	**65.33**	**87.67**	**65.67**	**75.33**	**56.33**	**77.00**	**57.00**	**77.33**	**57.00**

Table 6. In-context learning performance on $\text{ProcBench}_{\text{pg}}$, $\text{WeBridge}_{\text{pg}}$ and $\text{WeBridgeL}_{\text{pg}}$ (1,3,5)-shot.

Language models	$\text{ProcBench}_{\text{pg}}$						$\text{WeBridge}_{\text{pg}}$		$\text{WeBridgeL}_{\text{pg}}$	
	1-shot		3-shot		5-shot		1-shot		1-shot	
	EX	EM	EX	EM	EX	EM	EX	EM	EX	EM
CODES-3B	59.67	43.00	65.67	45.00	65.00	43.00	41.67	11.48	14.35	3.35
CODES-7B	60.00	43.67	60.33	43.67	60.67	43.33	45.28	8.81	22.01	7.80
CODES-15B	58.67	42.67	60.67	42.67	61.33	43.33	52.92	14.50	30.14	10.05
StarCoder2-3B	49.00	39.67	54.00	39.00	54.33	39.00	36.02	8.28	2.87	0.00
StarCoder2-7B	62.67	41.33	62.67	42.67	65.33	45.00	48.84	9.80	14.83	3.83
StarCoder2-15B	66.00	42.00	65.00	41.67	65.33	41.67	57.61	11.08	38.28	8.61
Llama-3.1-8B	62.67	42.33	66.67	43.67	68.33	44.00	49.81	11.30	28.78	7.80
CodeLlama-13B	62.33	42.00	62.67	42.33	63.33	43.00	42.33	10.19	11.96	2.87
StarCoder-15B	55.67	41.67	60.67	42.33	61.33	42.33	43.25	8.72	17.56	0.49
o3-mini	68.33	44.00	73.00	47.00	75.00	47.67	13.33	6.92	9.09	2.39
GPT-4.1	69.33	47.33	74.67	48.33	74.67	49.33	48.20	23.58	25.84	11.48
Gemini 2.5 Pro	71.67	47.67	72.67	47.67	74.67	50.00	60.81	22.86	43.54	19.62
PLForge-3B	68.33	45.00	69.67	45.00	71.00	44.67	62.84	26.36	46.89	16.75
PLForge-7B	75.33	51.33	76.67	51.00	76.33	49.67	65.11	26.84	50.24	22.01
PLForge-15B	**80.33**	**54.33**	**79.67**	**54.00**	**80.00**	**53.67**	**67.02**	**30.47**	**53.59**	**26.79**

that covers PL/SQL-related data. Ultimately, the PLForge-15B model demonstrates exceptional NL-to-PL/SQL capabilities on all datasets under PostgreSQL and Oracle, significantly surpassing the performance of other models. PLForge-3B model slightly outperforms the PLForge-7B model on the Hard dataset; however, on more realistic benchmarks such as ProcBench and WeBridge, the PLForge-7B model exhibits stronger generalization ability. In most cases, PLForge-7B ranks as runner-up. Moreover, general LLMs also demonstrate competitive performances. In particular, Gemini 2.5 Pro surpasses all open-source baseline models across all datasets, further highlighting the rapid advancement of proprietary LLMs in code generation and structured query understanding tasks. Furthermore, we observe that the performance under the 3-shot and 5-shot settings is sometimes lower than that under the 1-shot setting. We have also encountered similar phenomena in previous studies [36, 40]. This issue is likely due to schema noise and logic misalignment in the few-shot examples, which mislead the model and negatively impact its performance.

We conduct an analysis of the total token usage and inference budget across all zero-shot and few-shot experiments for the three reasoning models. Specifically, o3-mini consumes 48.66M tokens at a cost of $75.95, GPT-4.1 requires 50.08M tokens with an associated cost of $149.46, and Gemini 2.5 Pro utilizes 45.19M tokens incurring an inference cost of $97.52.

Table 7. Evaluation of SFT PLForge and SFT baseline models.

Language models	$\text{Simple}_{\text{pg}}$		Hard_{pg}		$\text{ProcBench}_{\text{pg}}$	
	EX	EM	EX	EM	EX	EM
SFT-CODES-3B	66.00	44.67	33.67	17.67	52.67	48.67
SFT-CODES-7B	71.67	46.00	34.33	19.33	44.33	42.00
SFT-CODES-15B	69.33	48.33	40.33	21.67	50.00	43.00
SFT-StarCoder2-3B	65.33	41.00	29.67	5.33	41.67	37.00
SFT-StarCoder2-7B	70.00	46.00	32.67	16.33	50.33	40.67
SFT-StarCoder2-15B	79.00	53.33	42.00	21.00	58.67	43.33
SFT-PLForge-3B	87.00	62.33	73.00	45.67	73.67	47.00
SFT-PLForge-7B	89.00	64.33	75.00	49.67	74.67	53.67
SFT-PLForge-15B	**89.33**	**67.67**	**77.33**	**56.00**	**80.33**	**55.00**

Table 8. In-context learning performance of PLForge on three PostgreSQL datasets under (0,1,3,5)-shot settings with correction. Upward arrows indicate metric improvements over the model without correction.

PLForge Models	0-shot		1-shot		3-shot		5-shot	
	EX	EM	EX	EM	EX	EM	EX	EM
				$\text{Simple}_{\text{pg}}$				
PLForge-3B	$84.33_{\uparrow 1.33}$	$61.33_{\uparrow 0.33}$	$84.00_{\uparrow 0.67}$	$60.33_{\uparrow 0.33}$	$83.67_{\uparrow 0.67}$	$59.33_{\uparrow 0.33}$	$83.67_{\uparrow 1.67}$	$59.33_{\uparrow 0.67}$
PLForge-7B	$85.67_{\uparrow 1.00}$	$61.67_{\uparrow 0.67}$	$88.00_{\uparrow 1.33}$	$62.67_{\uparrow 0.33}$	$88.33_{\uparrow 1.33}$	$62.67_{\uparrow 0.33}$	$87.67_{\uparrow 1.00}$	$62.67_{\uparrow 0.67}$
PLForge-15B	$89.33_{\uparrow 1.33}$	$66.33_{\uparrow 0.00}$	$87.67_{\uparrow 0.67}$	$66.00_{\uparrow 0.00}$	$88.00_{\uparrow 0.33}$	$65.33_{\uparrow 0.00}$	$88.33_{\uparrow 0.67}$	$65.67_{\uparrow 0.00}$
				Hard_{pg}				
PLForge-3B	$59.67_{\uparrow 0.67}$	$43.67_{\uparrow 0.67}$	$63.33_{\uparrow 1.00}$	$44.67_{\uparrow 0.33}$	$64.67_{\uparrow 2.00}$	$46.33_{\uparrow 1.00}$	$65.67_{\uparrow 2.33}$	$48.33_{\uparrow 1.67}$
PLForge-7B	$58.67_{\uparrow 4.67}$	$40.33_{\uparrow 1.33}$	$61.33_{\uparrow 3.67}$	$45.33_{\uparrow 1.33}$	$59.33_{\uparrow 3.33}$	$43.67_{\uparrow 1.00}$	$59.00_{\uparrow 2.67}$	$44.00_{\uparrow 1.00}$
PLForge-15B	$77.00_{\uparrow 1.33}$	$55.00_{\uparrow 0.33}$	$77.00_{\uparrow 1.67}$	$57.00_{\uparrow 0.67}$	$78.33_{\uparrow 1.33}$	$57.33_{\uparrow 0.33}$	$78.67_{\uparrow 1.33}$	$57.67_{\uparrow 0.67}$
				$\text{ProcBench}_{\text{pg}}$				
PLForge-3B	$69.67_{\uparrow 3.67}$	$44.33_{\uparrow 1.00}$	$71.67_{\uparrow 3.33}$	$45.67_{\uparrow 0.67}$	$72.33_{\uparrow 2.67}$	$45.00_{\uparrow 0.00}$	$74.33_{\uparrow 3.33}$	$45.67_{\uparrow 1.00}$
PLForge-7B	$76.33_{\uparrow 4.00}$	$51.00_{\uparrow 0.67}$	$77.67_{\uparrow 2.33}$	$51.67_{\uparrow 0.33}$	$78.33_{\uparrow 1.67}$	$51.67_{\uparrow 0.67}$	$78.00_{\uparrow 1.67}$	$50.33_{\uparrow 0.67}$
PLForge-15B	$81.67_{\uparrow 2.67}$	$55.00_{\uparrow 0.67}$	$82.00_{\uparrow 1.67}$	$55.00_{\uparrow 0.67}$	$81.33_{\uparrow 1.67}$	$54.67_{\uparrow 0.67}$	$82.33_{\uparrow 2.33}$	$54.33_{\uparrow 0.67}$

5.3 Evaluation on Supervised Fine-Tuning

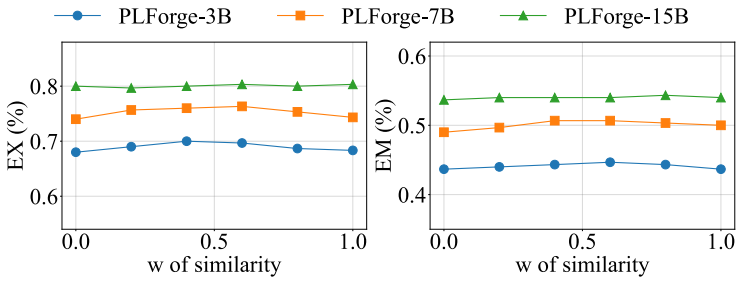
In the experiment, all SFT baseline models and SFT-PLForge models utilize the prompt construction strategy proposed in Section 4.3, enabling a fair comparison of SFT model performance. Table 7 delineates the performances of the PLForge, StarCoder2 and CODES models across the 3B, 7B, and 15B variants on $\text{Simple}_{\text{pg}}$, Hard_{pg} and $\text{ProcBench}_{\text{pg}}$ datasets subsequent to SFT under zero-shot setting. The 15B and 7B of PLForge consistently achieve the best and runner-up results, respectively, across all experimental results. Furthermore, the results suggest that SFT-PLForge demonstrates notable improvements in nearly all evaluated scenarios. After SFT, the three models within the PLForge suite exhibit improvements on the experimental datasets, with the EX accuracy increasing by an average of 6.41% and the EM accuracy by an average of 3.15%. The results show that SFT effectively enhances the model's performance in the NL-to-PL/SQL task by helping it better capture PL/SQL-specific patterns and improve the generalization in zero-shot evaluation.

5.4 Study of PL/SQL Correction

Table 8 presents the effectiveness of the correction model in enhancing PL/SQL correction across PLForge-(3B,7B,15B) models under the (0,1,3,5)-shot settings on $\text{Simple}_{\text{pg}}$, Hard_{pg} and $\text{ProcBench}_{\text{pg}}$ datasets. Specifically, the correction model yields an average improvement of 1.95% in EX and 0.67% in EM for PLForge-3B, 2.39% and 0.75% for PLForge-7B, and 1.42% and 0.45% for PLForge-15B, respectively. The consistent gains across model sizes and shot settings demonstrate the correction model's robustness in handling syntactic and semantic errors. Furthermore, integrating

Table 9. Comparison of the semantic-skeleton-aware and question-pattern-aware similarity.

Methods		Simple_{oc}						Hard_{oc}					
		3B		7B		15B		3B		7B		15B	
		EX	EM	EX	EM	EX	EM	EX	EM	EX	EM	EX	EM
1-shot	Sim_{ss}	**86.67**	**65.67**	**87.33**	**68.67**	**90.33**	**72.33**	**68.00**	**54.33**	**63.33**	**52.33**	**81.67**	**60.67**
	Sim_{qp}	86.33	65.33	81.67	64.00	**90.33**	71.67	67.33	53.33	60.67	48.00	79.67	59.67
3-shot	Sim_{ss}	84.33	**63.67**	**83.00**	**64.00**	**90.00**	**71.33**	65.33	53.67	**62.00**	**52.33**	**81.00**	**59.67**
	Sim_{qp}	**85.33**	62.00	80.67	63.67	88.67	70.67	**66.67**	**54.67**	59.33	47.67	79.67	59.33
5-shot	Sim_{ss}	**82.33**	**63.00**	**83.00**	**64.00**	**89.67**	**72.00**	**67.00**	**55.00**	**61.33**	**51.67**	**81.00**	**59.33**
	Sim_{qp}	81.67	61.67	82.00	63.67	88.33	70.00	64.67	53.67	60.67	48.00	**81.00**	59.00



(a) EX (b) EM

Fig. 6. Comparison of EX / EM scores varying similarity weighting factor from 0 to 1 on ProcBench_{pg} (3-shot).

the correction model with the PLForge series consistently enhances their NL-to-PL/SQL translation capabilities, thereby validating the effectiveness of our error correction framework.

5.5 Study of Similarity

This subsection compares our semantic-skeleton-aware similarity (Sim_{ss}) with the question-pattern-aware similarity (Sim_{qp}) from [36]. As shown in Table 9, the experimental results on Simple_{oc} and Hard_{oc} under (1,3,5)-shot settings (with $w = 0.5$ for Sim_{ss}) show that Sim_{ss} outperforms Sim_{qp} across PLForge-(3B, 7B, 15B) in the vast majority of cases. The results demonstrate its effectiveness in leveraging semantic and structural cues to enhance the model accuracy and generalization.

In Figure 6, we illustrate the impact of adjusting the weighting factor w from 0 to 1 on the semantic-skeleton-aware similarity in the ProcBench_{pg} dataset within a 3-shot setting. For PLForge-3B and PLForge-7B, both EX and EM metrics exhibit a clear trend of first increasing and then decreasing, suggesting that a balanced integration of semantic and skeleton similarities facilitates more effective few-shot example selection. In contrast, the performance of PLForge-15B remains relatively stable across different values of w, indicating that this larger model already possesses strong capabilities in understanding both NL semantics and PL/SQL skeleton structures. This robustness implies that PLForge-15B is less sensitive to the similarity weighting strategy due to its inherently stronger generalization ability.

5.6 Study of Skeleton

Figure 7 compares the skeleton prediction accuracy between the weighted skeleton prediction and the base skeleton prediction models across the Simple, Hard, and ProcBench datasets in both PostgreSQL and Oracle environments. The edit distance metric used here is normalized and converted to a percentage scale, where higher values indicate greater similarity and thus better

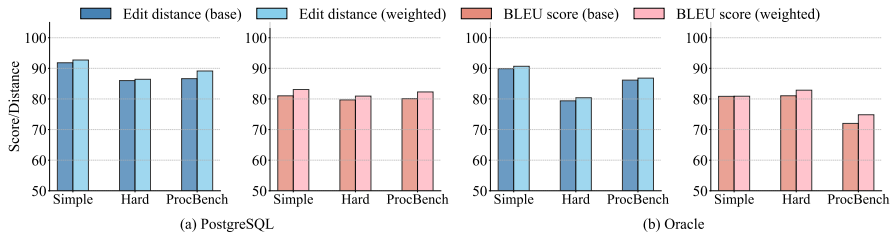


Fig. 7. Comparison of skeleton prediction accuracy on Simple, Hard and ProcBench under zero-shot setting.

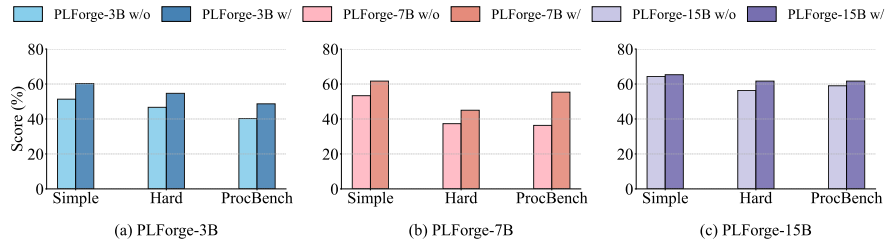


Fig. 8. Comparison of skeleton following scores on $\text{Simple}_{\text{pg}}$, Hard_{pg} and $\text{ProcBench}_{\text{pg}}$ (zero-shot).

accuracy. Our proposed weighted skeleton prediction method outperforms the base method with an average improvement of 1.18% in the edit distance score and 1.567% in the BLEU score. These improvements demonstrate that assigning higher weights to PL/SQL keywords in the loss function significantly enhances the generation quality of PL/SQL skeletons.

Figure 8 presents the skeleton-following capabilities of the PLForge model family under the zero-shot setting across the $\text{Simple}_{\text{pg}}$, Hard_{pg}, and $\text{ProcBench}_{\text{pg}}$ datasets. We evaluate the models by comparing the EM accuracy of the predicted skeletons with and without providing the ground-truth skeletons as the input. The results indicate that supplying the predicted skeletons leads to an average increase in the EM accuracy of 6.11%, 7.00%, and 8.00% on the $\text{Simple}_{\text{pg}}$, Hard_{pg}, and $\text{ProcBench}_{\text{pg}}$ datasets, respectively. The results suggest that the PLForge models are capable of leveraging the given skeletons to guide and constrain their PL/SQL generation, thus demonstrating a notable degree of the skeleton adherence.

5.7 Study of Keyword Sensitivity

As shown in Figure 9, we present the keywords sensitivity experiments with four baseline models and PLForge on Hard_{pg} and $\text{ProcBench}_{\text{pg}}$ in a zero-shot setting across various operations and control structures. Here, EX accuracy is measured as the proportion of correctly generated PL/SQL procedures that contain a given keyword out of all generated procedures including that keyword. The results indicate that PLForge-15B consistently achieves the highest accuracy, benefiting from the pre-training on PL/SQL-centric corpus. All models perform relatively poorly on the `LOOP` keyword, likely due to its structural complexity, including variable dependencies and multi-step logic, which pose greater challenges.

5.8 Ablation Studies

We conduct ablation studies on Simple, Hard and ProcBench datasets under PostgreSQL and Oracle in 3-shot setting. The specific results are presented in Table 10.

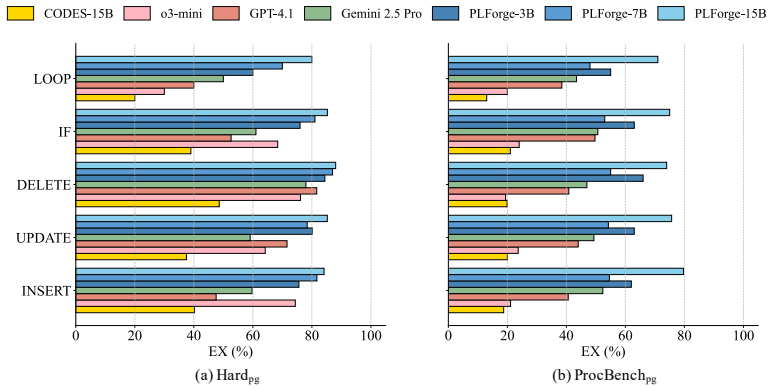


Fig. 9. Comparison of EX scores of keywords sensitivity on Hard$_{pg}$ and ProcBench$_{pg}$ under zero-shot setting.

Table 10. Ablation studies with PLForge on datasets under PostgreSQL and Oracle in 3-shot setting.

Methods	Simple						Hard						ProcBench					
	PLForge-3B		PLForge-7B		PLForge-15B		PLForge-3B		PLForge-7B		PLForge-15B		PLForge-3B		PLForge-7B		PLForge-15B	
	EX	EM	EX	EM	EX	EM	EX	EM	EX	EM	EX	EM	EX	EM	EX	EM	EX	EM
PostgreSQL Full	**83.00**	**59.00**	**87.00**	**62.33**	**87.67**	**65.33**	**62.67**	**45.33**	**56.00**	**42.67**	**77.00**	**57.00**	**69.67**	**45.00**	**76.67**	**51.00**	79.67	**54.00**
-w/o pre-training	75.00	49.00	79.00	53.33	78.33	52.67	38.33	23.67	48.00	24.33	59.67	28.67	54.00	39.00	62.67	42.67	65.00	41.67
-w/o database metadata	78.33	58.67	80.67	59.00	87.00	**65.33**	51.67	36.00	46.00	36.00	74.67	54.67	67.00	43.67	69.33	48.33	78.67	53.67
-w/o CoT	82.33	**59.00**	85.67	61.67	87.00	**65.33**	61.67	**45.33**	54.33	41.33	**77.00**	56.33	69.33	44.67	76.00	48.00	**80.33**	**54.00**
-w/o skeleton	74.00	56.33	68.33	50.67	87.33	63.00	56.67	40.67	41.67	32.00	71.00	52.67	68.33	**45.00**	56.00	32.00	**80.33**	47.33
Oracle Full	**84.33**	**63.67**	**83.00**	64.00	90.00	**71.33**	**65.33**	**53.67**	**62.00**	**52.33**	81.00	**59.67**	61.67	**57.00**	**64.33**	**57.33**	69.67	**58.00**
-w/o pre-training	73.00	61.67	71.33	59.33	79.33	65.00	45.00	36.00	49.00	40.00	58.67	43.00	42.00	38.00	50.33	42.67	60.33	48.33
-w/o database metadata	81.33	60.67	80.67	60.00	89.67	69.67	61.67	48.00	51.00	44.00	**81.67**	53.00	59.67	51.33	57.33	49.00	67.33	53.33
-w/o CoT	84.00	62.67	**83.00**	**64.33**	90.00	**71.33**	**65.33**	53.33	60.33	52.00	80.67	59.33	**62.00**	56.67	63.00	56.67	69.33	57.67
-w/o skeleton	78.33	58.67	71.33	55.00	**90.33**	65.00	60.33	47.33	42.00	37.33	76.33	57.00	56.00	55.00	51.67	46.67	**70.00**	57.67

Incremental Pre-training. Removing the pre-training stage leads to a notable performance drop in the PLForge models. The average EX and EM accuracies decrease by 13.26% and 14.07% on the PostgreSQL dataset, and by 14.70% and 11.44% on the Oracle dataset, respectively. The results show that incremental training on a PL/SQL-centric corpus helps the model better capture PL/SQL-specific syntax and semantics, enhancing its performance across different settings.

Database Metadata. When the database metadata is excluded from the prompt, there is a significant decline in both the EX and EM accuracies. Across the Simple$_{oc}$, Hard$_{oc}$ and ProcBench$_{oc}$ datasets, the EX accuracy decreases by an average of 4.28%, while the EM accuracy diminishes by an average of 4.13%. The results highlight the critical role of the database metadata, which provides essential auxiliary information that enables the model to generate PL/SQL codes accurately.

Chain-of-Thought. The experimental results on the Simple$_{oc}$, Hard$_{oc}$ and ProcBench$_{oc}$ datasets show that removing the CoT module leads to an average decrease of 0.45% in the EX accuracy and 0.50% in the EM accuracy. CoT decomposes the generation process into interpretable steps, such as declaration, parameter configuration, and logic design. This structured reasoning approach reduces generation errors and enhances syntactic alignment.

Predicted Skeleton. We further examine the role of the predicted skeleton in PL/SQL generation. Its removal results in an average decline of 7.80% in the EX accuracy and 6.52% in the EM accuracy, highlighting its importance in conveying structural cues. The predicted skeleton functions as a logical scaffold, guiding the model in generating control flow and procedural constructs that align with the intended semantics.

6 RELATED WORK

To the best of our knowledge, there is not existing method dedicated to the NL-to-PL/SQL task. Therefore, this section reviews three closely related research areas.

Supervised Fine-Tuning-Based Text-to-SQL. Supervised fine-tuning of the encoder-decoder neural network model is one of the popular approaches to tackling the Text-to-SQL task. Many existing studies focus on optimizing the encoding and the decoding processes. GlobalGNN [8], LGESQL [9], S^2SQL [32], Graphix-T5 [37], and GRL-SQL [25] leverage graph-based representations that incorporate both the query and the database schema. Similarly, recent studies in recommender systems have explored graph contrastive learning to mitigate exposure bias or activate interaction potentials [43, 68, 69], demonstrating the general effectiveness of graph-based contrastive paradigms across tasks. IRNet [27] and NatSQL [22] simplify the decoder's complexity by utilizing intermediate representations of SQL. CatSQL [21] and PICARD [57] introduce SQL syntax to constrain the decoder output space to ensure the generation of syntactically correct SQL queries. Additionally, ranking models [18, 19, 35, 44], such as RESDSQL [35], GAR [18], and MetaSQL [19], further enhance the precision of generated SQL queries. Finally, leveraging the capabilities of pre-trained language models [33, 52], some studies [13, 29] have fine-tuned these models for Text-to-SQL tasks.

In-Context Learning Based Text-to-SQL. Recent advancements in LLMs have significantly impacted Text-to-SQL tasks. LLM-based approaches are broadly categorized into zero-shot and few-shot methodologies. Zero-shot methods [10, 15, 50] concentrate on designing effective prompts without prior examples. DIN-SQL [50] employs a chain-of-thought approach that decomposes the Text-to-SQL task for effective guidance. Few-shot methodologies, on the other hand, involve examples to guide LLMs [4, 23, 26, 36, 49, 54, 59]. DAIL-SQL [23] prefers SQL-aligned examples, while CODES [36] prioritizes semantically similar demonstrations. In contrast, PURPLE [54] focuses on extracting logical operator composition knowledge from demonstrations. However, existing Text-to-SQL methods are solely designed for SQL and do not extend to PL/SQL. Conversely, PLForge specializes in the Text-to-PL/SQL task, with optimizations specifically tailored around PL/SQL.

Code Generation with Language Models. Code language models [39, 46, 47, 55] are typically trained on hybrid corpora consisting of multiple programming languages. However, broad-spectrum training data result in the suboptimal performance of code language models on specific programming languages like PL/SQL. We address this limitation by optimizing an open-source code language model through incremental training with a PL/SQL-centric corpus.

7 CONCLUSION

In this study, we proposed PLForge, a suite of pre-trained language models for the NL-to-PL/SQL task. We proposed a template-based method for dataset generation and built a PL/SQL-centric corpus for incremental pre-training. We designed effective prompt construction strategies. Experiments show that PLForge significantly outperforms existing baselines. This study primarily focuses on generating complete PL/SQL procedures from natural language. A typical real-world application scenario is interactive code completion, which we identify as a key direction for future work.

Acknowledgments

The authors would like to thank the anonymous reviewers for their insightful comments and suggestions on this work. The authors also gratefully acknowledge Guanchen Ge for his insightful suggestions and generous assistance during the revision process. The authors further gratefully acknowledge Zhaoguo Wang and Gansen Hu for generously providing the WeBridge dataset. This work is supported in part by the National Natural Science Foundation of China (No. 62372264 and No. 92467203). Chaokun Wang (chaokun@tsinghua.edu.cn) serves as the corresponding author.

References

[1] 2021. BERT: a review of applications in natural language processing and understanding. *arXiv preprint arXiv:2103.11943* (2021).

[2] 2025. Apache OFBiz. https://github.com/apache/ofbiz-framework

[3] 2025. Extra Materials for PLForge. https://github.com/ZhanGHanG9991/plforge

[4] Aseem Arora, Shabbirhussain Bhaisaheb, Harshit Nigam, Manasi Patwardhan, Lovekesh Vig, and Gautam Shroff. 2023. Adapt and Decompose: Efficient Generalization of Text-to-SQL via Domain Adapted Least-To-Most Prompting. In *Proceedings of the 1st GenBench Workshop on (Benchmarking) Generalisation in NLP*. 25–47.

[5] Christopher Baik, Hosagrahar V Jagadish, and Yunyao Li. 2019. Bridging the semantic gap with SQL query logs in natural language interfaces to databases. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 374–385.

[6] Itzik Ben-Gan, Dejan Sarka, Roger Wolter, Greg Low, Ed Katibah, and Isaac Kunen. 2009. *Inside Microsoft SQL Server 2008 T-SQL Programming*. Microsoft Press.

[7] Ben Bogin, Jonathan Berant, and Matt Gardner. 2019. Representing Schema Structure with Graph Neural Networks for Text-to-SQL Parsing. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. 4560–4565.

[8] Ben Bogin, Matt Gardner, and Jonathan Berant. 2019. Global Reasoning over Database Structures for Text-to-SQL Parsing. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. 3659–3664.

[9] Ruisheng Cao, Lu Chen, Zhi Chen, Yanbin Zhao, Su Zhu, and Kai Yu. 2021. LGESQL: Line Graph Enhanced Text-to-SQL Model with Mixed Local and Non-Local Relations. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. 2541–2555.

[10] Shuaichen Chang and Eric Fosler-Lussier. [n. d.]. How to Prompt LLMs for Text-to-SQL: A Study in Zero-shot, Single-domain, and Cross-domain Settings. In *NeurIPS 2023 Second Table Representation Learning Workshop*.

[11] Alvin Cheung, Owen Arden, Samuel Madden, and Andrew C. Myers. 2012. Automatic Partitioning of Database Applications. *Proc. VLDB Endow.* 5 (2012), 1471–1482.

[12] Tri Dao. 2024. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. In *12th International Conference on Learning Representations, ICLR 2024*.

[13] Xiang Deng, Ahmed Hassan, Christopher Meek, Oleksandr Polozov, Huan Sun, and Matthew Richardson. 2021. Structure-Grounded Pretraining for Text-to-SQL. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 1337–1350.

[14] Ning Ding, Yulin Chen, Bokai Xu, Yujia Qin, Shengding Hu, Zhiyuan Liu, Maosong Sun, and Bowen Zhou. 2023. Enhancing Chat Language Models by Scaling High-quality Instructional Conversations. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. 3029–3051.

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

[16] Korry Douglas and Susan Douglas. 2003. *PostgreSQL: a comprehensive guide to building, programming, and administering PostgresSQL databases*. SAMS publishing.

[17] Ju Fan, Zihui Gu, Songyue Zhang, Yuxin Zhang, Zui Chen, Lei Cao, Guoliang Li, Samuel Madden, Xiaoyong Du, and Nan Tang. 2024. Combining small language models and large language models for zero-shot NL2SQL. *Proceedings of the VLDB Endowment* 17, 11 (2024), 2750–2763.

[18] Yuankai Fan, Zhenying He, Tonghui Ren, Dianjun Guo, Lin Chen, Ruisi Zhu, Guanduo Chen, Yinan Jing, Kai Zhang, and X Sean Wang. 2023. Gar: A generate-and-rank approach for natural language to sql translation. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 110–122.

[19] Yuankai Fan, Zhenying He, Tonghui Ren, Can Huang, Yinan Jing, Kai Zhang, and Xiaoyang Sean Wang. 2024. Metasql: A Generate-Then-Rank Framework for Natural Language to SQL Translation. *2024 IEEE 40th International Conference on Data Engineering (ICDE)* (2024), 1765–1778.

[20] Steven Feuerstein and Bill Pribyl. 2005. *Oracle pl/sql Programming*. " O'Reilly Media, Inc.".

[21] Han Fu, Chang Liu, Bin Wu, Feifei Li, Jian Tan, and Jianling Sun. 2023. Catsql: Towards real world natural language to sql applications. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1534–1547.

[22] Yujian Gan, Xinyun Chen, Jinxia Xie, Matthew Purver, John R Woodward, John Drake, and Qiaofu Zhang. 2021. Natural SQL: Making SQL Easier to Infer from Natural Language Specifications. In *Findings of the Association for Computational Linguistics: EMNLP 2021*. 2030–2042.

[23] Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2024. Text-to-SQL Empowered by Large Language Models: A Benchmark Evaluation. *Proceedings of the VLDB Endowment* 17, 5 (2024), 1132–1145.

[24] Tianyu Gao, Xingcheng Yao, and Danqi Chen. 2021. SimCSE: Simple Contrastive Learning of Sentence Embeddings. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. 6894–6910.
[25] Zheng Gong and Ying Sun. 2024. Graph reasoning enhanced language models for text-to-sql. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 2447–2451.
[26] Zihui Gu, Ju Fan, Nan Tang, Lei Cao, Bowen Jia, Sam Madden, and Xiaoyong Du. 2023. Few-shot text-to-sql translation using structure and content prompt learning. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–28.
[27] Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. 2019. Towards Complex Text-to-SQL in Cross-Domain Database with Intermediate Representation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. 4524–4535.
[28] Surabhi Gupta and Karthik Ramachandra. 2021. Procedural Extensions of SQL: Understanding their usage in the wild. *Proceedings of the VLDB Endowment* 14, 8 (2021), 1378–1391.
[29] Jonathan Herzig, Pawel Krzysztof Nowak, Thomas Mueller, Francesco Piccinno, and Julian Eisenschlos. 2020. TaPas: Weakly Supervised Table Parsing via Pre-training. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 4320–4333.
[30] Or Honovich, Thomas Scialom, Omer Levy, and Timo Schick. 2023. Unnatural Instructions: Tuning Language Models with (Almost) No Human Labor. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 14409–14428.
[31] Gansen Hu, Zhaoguo Wang, Chuzhe Tang, Jiahuan Shen, Zhiyuan Dong, Sheng Yao, and Haibo Chen. 2024. WeBridge: Synthesizing Stored Procedures for Large-Scale Real-World Web Applications. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–29.
[32] Binyuan Hui, Ruiying Geng, Lihan Wang, Bowen Qin, Yanyang Li, Bowen Li, Jian Sun, and Yongbin Li. 2022. S2SQL: Injecting Syntax to Question-Schema Interaction Graph Encoder for Text-to-SQL Parsers. In *Findings of the Association for Computational Linguistics: ACL 2022*. 1254–1262.
[33] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdel rahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2019. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 7871–7880.
[34] Fei Li and Hosagrahar V Jagadish. 2014. Constructing an interactive natural language interface for relational databases. *Proceedings of the VLDB Endowment* 8, 1 (2014), 73–84.
[35] Haoyang Li, Jing Zhang, Cuiping Li, and Hong Chen. 2023. Resdsql: Decoupling schema linking and skeleton parsing for text-to-sql. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 13067–13075.
[36] Haoyang Li, Jing Zhang, Hanbing Liu, Ju Fan, Xiaokang Zhang, Jun Zhu, Renjie Wei, Hongyan Pan, Cuiping Li, and Hong Chen. 2024. Codes: Towards building open-source language models for text-to-sql. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–28.
[37] Jinyang Li, Binyuan Hui, Reynold Cheng, Bowen Qin, Chenhao Ma, Nan Huo, Fei Huang, Wenyu Du, Luo Si, and Yongbin Li. 2023. Graphix-t5: Mixing pre-trained transformers with graph-aware layers for text-to-sql parsing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 13076–13084.
[38] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. 2024. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems* 36 (2024).
[39] R Li, LB Allal, Y Zi, N Muennighoff, D Kocetkov, C Mou, M Marone, C Akiki, J Li, J Chim, et al. 2023. StarCoder: May the Source be With You! *Transactions on machine learning research* (2023).
[40] Zhaodonghui Li, Haitao Yuan, Huiming Wang, Gao Cong, and Lidong Bing. 2024. LLM-R2: A Large Language Model Enhanced Rule-based Rewrite System for Boosting Query Efficiency. *Proc. VLDB Endow.* 18 (2024), 53–65.
[41] Tsung-Yi Lin, Priya Goyal, Ross B. Girshick, Kaiming He, and Piotr Dollár. 2017. Focal Loss for Dense Object Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42 (2017), 318–327.
[42] Xi Victoria Lin, Richard Socher, and Caiming Xiong. 2020. Bridging Textual and Tabular Data for Cross-Domain Text-to-SQL Semantic Parsing. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. 4870–4888.
[43] Ziyang Liu and Chaokun Wang. 2025. TeRDy: Temporal Relation Dynamics through Frequency Decomposition for Temporal Knowledge Graph Completion. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 9611–9622.
[44] Ziyang Liu, Chaokun Wang, Hao Feng, Lingfei Wu, and Liqun Yang. 2022. Knowledge Distillation based Contextual Relevance Matching for E-commerce Product Search. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing: Industry Track*. 63–76.
[45] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. In *7th International Conference on Learning Representations, ICLR 2019*.

[46] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173* (2024).

[47] Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023. Codegen2: Lessons for training llms on programming and natural languages. *arXiv preprint arXiv:2305.02309* (2023).

[48] Andrew Pavlo. 2017. What are we doing with our lives? Nobody cares about our concurrency control research. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 3–3.

[49] Gabriel Poesia, Oleksandr Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable code generation from pre-trained language models. (2022).

[50] Mohammadreza Pourreza and Davood Rafiei. 2024. Din-sql: Decomposed in-context learning of text-to-sql with self-correction. *Advances in Neural Information Processing Systems* 36 (2024).

[51] Alec Radford. 2018. Improving language understanding by generative pre-training. (2018).

[52] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research* 21, 140 (2020), 1–67.

[53] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–16.

[54] Tonghui Ren, Yuankai Fan, Zhenying He, Ren Huang, Jiaqi Dai, Can Huang, Yinan Jing, Kai Zhang, Yifan Yang, and Xiaoyang Sean Wang. 2024. PURPLE: Making a Large Language Model a Better SQL Writer. *2024 IEEE 40th International Conference on Data Engineering (ICDE)* (2024), 15–28.

[55] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[56] 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. *Proceedings of the VLDB Endowment* 9, 12 (2016), 1209–1220.

[57] 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*. 9895–9901.

[58] Alkis Simitsis, Georgia Koutrika, and Yannis Ioannidis. 2008. Précis: from unstructured keywords as queries to structured databases as answers. *The VLDB Journal* 17 (2008), 117–149.

[59] Ruoxi Sun, Sercan Ö. Arik, Hootan Nakhost, Hanjun Dai, Rajarishi Sinha, Pengcheng Yin, and Tomas Pfister. 2024. SQL-PaLM: Improved Large Language Model Adaptation for Text-to-SQL. *Transactions on machine learning research* 2024 (2024).

[60] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[61] A Vaswani. 2017. Attention is all you need. *Advances in Neural Information Processing Systems* (2017).

[62] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[63] Tomer Wolfson, Daniel Deutch, and Jonathan Berant. 2022. Weakly Supervised Text-to-SQL Parsing through Question Decomposition. In *Findings of the Association for Computational Linguistics: NAACL 2022*. 2528–2542.

[64] Ziyu Yao, Daniel S Weld, Wei-Peng Chen, and Huan Sun. 2018. Staqc: A systematically mined question-code dataset from stack overflow. In *Proceedings of the 2018 World Wide Web Conference*. 1693–1703.

[65] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *Proceedings of the 15th international conference on mining software repositories*. 476–486.

[66] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 2018. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. 3911–3921.

[67] Yiming Zhang, Shi Feng, and Chenhao Tan. 2022. Active Example Selection for In-Context Learning. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*. 9134–9148.

[68] Leqi Zheng, Chaokun Wang, Canzhi Chen, Jiajun Zhang, Cheng Wu, Zixin Song, Shannan Yan, Ziyang Liu, and Hongwei Li. 2025. LAGCL4Rec: When LLMs Activate Interactions Potential in Graph Contrastive Learning for

Recommendation. In *The 2025 Conference on Empirical Methods in Natural Language Processing*.
[69] Leqi Zheng, Chaokun Wang, Ziyang Liu, Canzhi Chen, Cheng Wu, and Hongwei Li. 2025. Balancing Self-Presentation and Self-Hiding for Exposure-Aware Recommendation Based on Graph Contrastive Learning. In *Proceedings of the 48th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 2027–2037.
[70] Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K Reddy. 2022. Xlcost: A benchmark dataset for cross-lingual code intelligence. *arXiv preprint arXiv:2206.08474* (2022).

Received April 2025; revised July 2025; accepted August 2025