Pontifícia Universidade Católica
do Rio de Janeiro

Eduardo Roger Silva Nascimento

Querying Databases with Natural Language: The use of Large Language Models for Text-to-SQL tasks

Dissertação de Mestrado

Dissertation presented to the Programa de Pós–graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Informática.

Advisor: Prof. Marco Antonio Casanova

Rio de Janeiro
April 2024

Pontifícia Universidade Católica
do Rio de Janeiro

Eduardo Roger Silva Nascimento

Querying Databases with Natural Language: The use of Large Language Models for Text-to-SQL tasks

Dissertation presented to the Programa de Pós–graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Informática. Approved by the Examination Committee:

Prof. Marco Antonio Casanova
Advisor
Departamento de Informática – PUC-Rio

Profa. Melissa Lemos Cavaliére
Instituto Tecgraf – PUC-Rio

Prof. Luiz André Portes Paes Leme
UFF

Rio de Janeiro, April 4th, 2024

Eduardo Roger Silva Nascimento

Graduated in computer science by the Federal University of Maranhão.

Bibliographic data

Nascimento, Eduardo

Querying Databases with Natural Language: The use of Large Language Models for Text-to-SQL tasks / Eduardo Roger Silva Nascimento; advisor: Marco Antonio Casanova. – 2024.

106 f: il. color. ; 30 cm

Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2024.

Inclui bibliografia

1. Informática – Teses. 2. Texto-para-SQL. 3. Modelos de Linguagem Grandes. 4. LangChain. 5. GPT. I. Casanova, Marco. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

To my family, for their support
and encouragement.

Acknowledgments

First, I express my gratitude to God for His grace and strength, which have been sources of inspiration and empowerment at all times.

I thank my advisor, Marco A. Casanova, for his encouragement, partnership, patience and valuable insights throughout this process. Thank you for the fantastic person you are and the positive impact you have had on my life.

To my dear family, for their constant encouragement and support. Their words of encouragement have been my anchor in challenging times.

To my friends, especially Anderson, Nelia, and Mayara, and colleagues who have stood by me during this journey, my thanks. Your moral support, constructive discussions, and encouragement have been essential in maintaining my motivation and determination.

To PUC-Rio and the professors, for providing a stimulating academic environment and the opportunity to learn and grow as a researcher.

To Tecgraf Institute and my colleagues, especially Melissa, Grettel, Gustavo, Lucas, and Yenier, for their essential collaborations and support in the development of this work.

I thank Petrobras who made it possible to complete this study.

Finally, I would like to express my sincere gratitude to all individuals and entities who contributed to the completion of this research.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

Abstract

Nascimento, Eduardo; Casanova, Marco (Advisor). **Querying Databases with Natural Language: The use of Large Language Models for Text-to-SQL tasks**. Rio de Janeiro, 2024. 106p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

The *Text-to-SQL* task involves generating an SQL query based on a given relational database and a Natural Language (NL) question. While the leaderboards of well-known benchmarks indicate that Large Language Models (LLMs) excel in this task, they are evaluated on databases with simpler schemas. This dissertation first investigates the performance of LLM-based Text-to-SQL models on a complex and openly available database (Mondial) with a large schema and a set of 100 NL questions. Running under GPT-3.5 and GPT-4, the results of this first experiment show that the performance of LLM-based tools is significantly less than that reported in the benchmarks and that these tools struggle with schema linking and joins, suggesting that the relational schema may not be suitable for LLMs. This dissertation then proposes using LLM-friendly views and data descriptions for better accuracy in the Text-to-SQL task. In a second experiment, using the strategy with better performance, cost and benefit from the previous experiment and another set with 100 questions over a real-world database, the results show that the proposed approach is sufficient to considerably improve the accuracy of the prompt strategy. This work concludes with a discussion of the results obtained and suggests further approaches to simplify the Text-to-SQL task.

Keywords

Text-to-SQL; Large Language Models; LangChain; GPT.

Resumo

Nascimento, Eduardo; Casanova, Marco. **Consultando bancos de dados com linguagem natural: o uso de modelos de linguagem grandes para tarefas de texto-para-SQL**. Rio de Janeiro, 2024. 106p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

A tarefa chamada brevemente de *Texto-para-SQL* envolve a geração de uma consulta SQL com base em um banco de dados relacional e uma pergunta em linguagem natural. Embora os rankings de benchmarks conhecidos indiquem que Modelos de Linguagem Grandes (LLMs) se destacam nessa tarefa, eles são avaliados em bancos de dados com esquemas bastante simples. Esta dissertação investiga inicialmente o desempenho de modelos *Texto-para-SQL* baseados em LLMs em um banco de dados disponível ao público (Mondial) com um esquema conceitual complexo e um conjunto de 100 perguntas em Linguagem Natural (NL). Executando sob GPT-3.5 e GPT-4, os resultados deste primeiro experimento mostram que as ferramentas baseadas em LLM têm desempenho significativamente inferior ao relatado nesses benchmarks e enfrentam dificuldades com a vinculação de esquemas e joins, sugerindo que o esquema relacional pode não ser adequado para LLMs. Essa dissertação propõe então o uso de visões e descrições de dados amigáveis ao LLM para melhorar a precisão na tarefa *Texto-para-SQL*. Em um segundo experimento, usando a estratégia com melhor performance, custo e benefício do experimento anterior e outro conjunto com 100 perguntas sobre um banco de dados do mundo real, os resultados mostram que a abordagem proposta é suficiente para melhorar consideravelmente a precisão da estratégia de prompt. Esse trabalho conclui com uma discussão dos resultados obtidos e sugere abordagens adicionais para simplificar a tarefa de *Texto-para-SQL*.

Palavras-chave

Texto-para-SQL; Modelos de Linguagem Grandes; LangChain; GPT.

Table of contents

List of figures

List of tables

List of algorithms

List of Listing

List of Abreviations

API – Application Programming Interface

CoT – Chain-of-Thought

CM – Component Matching

EA – Execution Accuracy

ESM – Exact Set Matching

FFT – Full Fine-tuning

GPT – Generative Pre-trained Transformers

LaMDA — Language Model for Dialogue Applications

LLaMA -– Large Language Model Meta AI

LLM – Large Language Models

NL – Natural Language

NLP – Natural Language Processing

PaLM — Pathways Language Model

PEFT – Parameter Efficient Fine-Tuning

RAG – Retrieval Augmentation Generation

RLHF – Reinforcement Learning from Human Feedback

VES – Valid Efficiency Score

The limits of my language mean the limits of my world.

Ludwig Wittgenstein, *Tractatus Logico-Philosophicus.*

1
Introduction

Natural Language Interfaces for Databases (NLIDBs) have long been a crucial area of research. The purpose of these systems is to query databases and retrieve information using natural language queries, eliminating the need for formal query languages such as Structured Query Language (SQL) or SPARQL Protocol and RDF Query Language (SPARQL)(HENDRIX, 1982).

In NLIDB systems, individuals can effortlessly collect information from databases, reshaping our perception of database information. Traditionally, people are accustomed to working with forms, and their expectations heavily depend on the capabilities of these forms (NIHALANI; SILAKARI; MOTWANI, 2011). However, NLIDB systems face significant challenges, including ambiguities, implicit query context, linguistic variations, incomplete queries, and the query generation itself (AFFOLTER; STOCKINGER; BERNSTEIN, 2019).

Numerous systems and architectures attempted to address these challenges, including several systems that adopt a Text-to-SQL approach (XU; LIU; SONG, 2017; YAGHMAZADEH et al., 2017; YU et al., 2018a). Text-to-SQL refers to the task defined as "given a natural language question Q and a database schema S = ⟨T, C⟩, generate the corresponding SQL query P, assuming that the question Q is a sequence of words $(q_1, ..., q_n)$ and the database schema S consists of tables $T = (T_1, ..., T_m)$, and columns $C = (C_1, ..., C_k)$" (GUO et al., 2023a).

Large Language Models (LLMs) follow a deep neural network architecture composed of billions of parameters, such as the Transformer architecture (VASWANI; AL., 2017). Trained on enormous quantities of unlabeled text using self-supervised or semi-supervised learning, these models capture generalized semantic representations of words and texts, making them applicable to various natural language processing tasks.[1] The versatile nature of LLMs positions them at the forefront of advancing technologies in linguistic understanding and text generation. This expansive linguistic understanding positions LLMs as versatile tools for a broad spectrum of NLP applications such as question answering, writing, text translation, document summarization, and code generation in programming languages (SINGH, 2023). These model also play pivotal roles in chatbots, digital assistants, and various other applications necessitating text generation or comprehension capabilities. The rise in rele-

[1]https://openai.com/research/better-language-models

vance of LLMs is particularly evident with the introduction of ChatGPT, an OpenAI language model optimized for dialogue using Reinforcement Learning with Human Feedback (RLHF) (OPENAI, 2023b).

The use of *LLM-based Text-to-SQL strategies*, that is, Text-to-SQL strategies that use Large Language Models (LLMs), has gained popularity, showing relative success over well-known benchmarks, including the Spider benchmark (GAO et al., 2023; POURREZA; RAFIEI, 2023; DONG et al., 2023b). Despite promising, the results reported for these benchmarks are biased towards databases with small schemas, few columns, instances, and joins between tables. Also, the NL questions used in the benchmarks are written in the schema vocabulary.

By contrast, real-world databases often feature large complex schemas, with abbreviated and ambiguous attribute and table names. Furthermore, user's NL questions may be written in terms which are quite different from those of the database schema or the data values. In fact, the relational schema is often an inappropriate specification of the database from the point of view of an LLM.

Building upon these observations, this dissertation presents experiments to assess LLM-based Text-to-SQL strategies on two challenging scenarios characterized by: (1) an openly available database with a large complex schema, using terms close to those of the users' NL questions; (2) a real-world database with a large complex schema, using terms different from those of the users' NL questions.

To address the first scenario, the dissertation considers a benchmark, which will be referred to as the *Mondial benchmark*, consisting of an openly available database, Mondial, with a complex and large schema, and a set of 100 questions along with their corresponding SQL translations. This set contains a variety of queries, including aggregations, numerous filters, ambiguities, and requirements for multiple joins, which are similar to real-world user queries. The investigation explores various Text-to-SQL strategies involving prompt engineering, such as the Langchain tools for Text-to-SQL and successful strategies identified in the Spyder benchmark.

Using the GPT-3.5 and GPT-4 models from OpenAI, the results suggest that the performance of LLM-based Text-to-SQL tools over the Mondial benchmark is significantly less than that reported for the Spider benchmark. Despite excelling in handling aggregations, the tools exhibit considerable errors in schema linking and join operations. This discrepancy in performance can be attributed to the complexity of databases with larger schemas.

Consider now the second scenario for real-world databases. The dissertation first argues that the Text-to-SQL task can be greatly facilitated by providing a database specification based on the use of *LLM-friendly views*, that are close to the language of the users' questions and that eliminate frequently used joins, and *LLM-friendly data descriptions* of the database values. The dissertation proceeds with experiments that use a real-world relational database, with another set with 100 questions, and the Text-to-SQL tool with the best performance and cost in the experiment with the Mondial benchmark. The results show that the use of LLM-friendly views and data samples, albeit not too difficult to implement over a real-world relational database, are sufficient to considerably improve the accuracy of the prompt strategy.

Finally, this work aims at contributing to the broader dialogue around the potential and implementation of LLMs. By demonstrating its ability to excel in these specific tasks, the dissertation contributes to the growing body of knowledge about the practicality and usefulness of LLMs in real-world applications.

This work is organized as follows: Chapter 2 covers related work. Chapter 3 presents the required definitions. Chapter 4 describes the LLM-based Text-to-SQL strategies that were tested. Chapter 5 evaluates the effect of schema complexity on LLM-based Text-to-SQL strategies. Chapter 6 evaluates the effect of using views for this same task. Finally, Chapter 7 contains the conclusions and suggestions for future work.

2
Previous Work

This chapter provides an overview of related work focusing on benchmarks, evaluation metrics, and approaches to Text-to-SQL tasks. Section 2.1 discusses the benchmark datasets commonly employed in Text-to-SQL tasks, while Section 2.2 examines prevalent evaluation metrics. Finally, Section 2.3 discusses previous research utilizing LLMs for converting NL sentences to SQL queries.

2.1
Text-to-SQL Dataset Benchmarks

A high-quality dataset is essential for driving the development of various natural language processing tasks, including Text-to-SQL. There are single-domain Text-to-SQL datasets such as GeoQuery (ZELLE; MOONEY, 1996), ATIS (DAHL et al., 1994), Restaurant (IYER et al., 2017) and SQL-Eval (PING, 2023), which are designed for specific information retrieval tasks.

WikiSQL (ZHONG; XIONG; SOCHER, 2017) has 80,654 NL sentences and SQL annotations of 24,241 tables. Each query in WikiSQL is limited to the same table and does not contain complex operations such as sorting and grouping.

The Spider – Yale Semantic Parsing and Text-to-SQL Challenge (YU et al., 2018b) offers datasets for training and testing Text-to-SQL tools. Spider features nearly 200 databases covering 138 different domains from three resources: 70 complex databases from different college database courses, SQL tutorial websites, online CSV files, and textbook examples; 40 databases from DatabaseAnswers;[1] and 90 databases based on WikiSQL, with about 500 tables in about 90 different domains.

For each database, Spider lists 20-50 hand-written NL questions and their SQL translations. An NL question S, with an SQL translation Q, is classified as easy, medium, hard, and extra-hard, where the difficulty is based on the number of SQL constructs of Q – `GROUP BY`, `ORDER BY`, `INTERSECT`, nested sub-queries, column selections, and aggregators – so that an NL query whose translation Q contains more SQL constructs is considered harder.

Most databases in Spider have small schemas: the largest five databases have between 16 and 25 tables, and about half of the databases have schemas with five tables or fewer. Furthermore, all Spider NL questions are phrased in

[1] <http://www.databaseanswers.org/>

terms used in the database schemas. These two limitations considerably reduce the difficulty of the Text-to-SQL task. Therefore, the results reported in the Spider leaderboard are biased toward databases with small schemas and NL questions written in the schema vocabulary, which is not what one finds in real-world databases.

The Spider dataset has modified versions, such as Spider-Syn and Spider DK. In the Spider-Syn (GAN et al., 2021) dataset, words and phrases are replaced with synonyms and emphasizes the importance of maintaining accurate schema linking mechanisms. It comprises 5,672 questions and aims at leveraging common word substitutions related to schema items and values to enhance the accuracy of models in translating natural language questions into SQL queries.

The Spider DK dataset (GAN; CHEN; PURVER, 2021) addresses testing how well Text-to-SQL tools deal with domain knowledge. Contains 535 pairs of natural language (NL) questions and SQL queries. Of these, 270 pairs are identical to the original samples from the Spider development set, while the remaining 265 pairs have been modified to incorporate domain knowledge. The Spider DK dataset highlights the challenge of cross-domain knowledge, in which models need to generate correct and precise SQL queries, even when confronted with new domains or scenarios not encountered during training.

BIRD – BIg Bench for LaRge-scale Database Grounded Text-to-SQL Evaluation (LI et al., 2023) is a large-scale cross-domain Text-to-SQL benchmark in English. The dataset contains 12,751 Text-to-SQL data pairs and 95 databases with a total size of 33.4 GB across 37 domains. The BIRD dataset tries to bridge the gap between Text-to-SQL research and real-world applications by exploring three additional challenges: dealing with large and messy database values, external knowledge inference, and optimizing SQL execution efficiency. However, BIRD still does not have many databases with large schemas: of the 73 databases in the training dataset, only two have more than 25 tables. And one of them is the Mondial database, of which another and larger version is used in this work.

SQL-Eval is a framework that evaluates the correctness of Text-to-SQL strategies, created during the development of SQLCoder (this model is discussed in section 2.3).[2] It is a set of manually selected questions and queries, grouped by query category (PING, 2023).

Chase (GUO et al., 2021) is a dataset for the cross-database context-dependent Text-to-SQL problem in Chinese. It consists of 5,459 question sequences (17,940 questions) over 280 databases, in which only 35% of questions

[2] <https://github.com/defog-ai/sql-eval>

are context-independent, and 28% of SQL queries are easy. Each question in Chase has rich semantic annotations, including its SQL query, contextual dependency, and schema linking.

Finally, the `sql-create-context` dataset also addresses the Text-to-SQL task, and was built from WikiSQL and Spider. [3] It contains 78,577 examples of NL queries, SQL `CREATE TABLE` statements, and SQL Queries answering the questions. The `CREATE TABLE` statement provides context for the LLMs, without having to provide actual rows of data.

2.2 Text-to-SQL Evaluation Metrics

Evaluating the semantic accuracy of a Text-to-SQL model is a long-standing problem: how to know whether the predicted SQL query has the same denotation as the ground truth SQL query, for every possible database (ZHONG; YU; KLEIN, 2020). The main evaluation metrics in Text-to-SQL tasks include Component Matching (CM), Exact Set Matching (ESM), and Execution Accuracy (EA).

CM intricately analyzes model performance by measuring the average exact match across diverse SQL components. Each component, such as SELECT, WHERE, GROUP BY, ORDER BY, and KEYWORDS, undergoes decomposition into subcomponents, treated as unordered sets for precise matching (LAN et al., 2023). ESM measures the matched SQL keywords between the predicted SQL query and its corresponding ground truth SQL query (YU et al., 2018b). EA, on the other hand, compares the execution output of the predicted SQL query with that of the ground truth SQL query on some database instances (GAO et al., 2023).

There are metrics designed to assess the efficiency of valid SQLs generated by models. Valid Efficiency Score (VES), proposed by Li et al. (2023), aims at quantifying the efficiency of the SQL generated by LLM, that is, running time, throughput, memory cost, or merged metrics.

Most of the LLM-based Text-to-SQL tools (POURREZA; RAFIEI, 2023; DONG et al., 2023b; GAO et al., 2023; GUO et al., 2023b) and benchmarks (YU et al., 2018b; LI et al., 2023; LAN et al., 2023) have EA as a metric.

2.3 Text-to-SQL LLM Tools

Approaches to Text-to-SQL tasks using LLMs have recently garnered significant attention. Lu et al. (2023) use Large Language Models (LLMs) to

[3] <https://huggingface.co/datasets/b-mc2/sql-create-context>

create a natural language interface, enabling pharmacologists to query public or private pharmacology databases.

Liu et al. (2023) provide a comprehensive analysis of ChatGPT's zero-shot Text-to-SQL capability, assessing its performance on 12 benchmark datasets and comparing it to state-of-the-art models. The study highlights ChatGPT's robust Text-to-SQL abilities, surpassing current state-of-the-art models in specific scenarios. It underscores the significance of zero-shot code generation models, especially in Text-to-SQL tasks. The research suggests future directions, such as addressing non-executable SQL statements and incorporating more in-context examples into the prompt.

Nascimento et al. (2023) illustrated how ChatGPT and LangChain can contribute to the development of natural language interfaces for databases (NLIDBs) through two approaches, one involving tools to generate an SQL query from an NL question, and the other using the Language Model to extract keywords from an NL question and passing the keywords to a Keyword Search (KwS) tool to execute the query on the database. The most effective NLIDB approach was the second one, incorporating prompts with contextual information and examples to facilitate keyword extraction.

The Spider[4] and BIRD [5] web sites publish a leaderboard of the best performing Text-to-SQL tools, some of which are discussed below.

Dong et al. (2023b) introduced C3, a ChatGPT-based zero-shot Text-to-SQL method that proficiently generates SQL queries with fewer tokens and on LLM's cheapest model, the GPT-3.5-turbo. C3 incorporates Clear Prompting for schema linking, addressing irrelevant items, and reducing unnecessary length. Calibration with Hints mitigates biases in ChatGPT's output, guiding the model to select necessary columns and avoid keyword misusage. The Consistency Output component employs a self-consistency method, sampling multiple reasoning paths, executing queries, and selecting the most consistent answer for improved reliability. The evaluation focused on execution accuracy, and the implementation used the OpenAI ChatGPT API, achieving robust results in SQL query generation. C3 had an accuracy of 82.3% on the Spider dataset.

Gao et al. (2023) presented a comprehensive study on Text-to-SQL empowered by Large Language Models (LLMs). The research emphasizes three crucial components of LLM-based Text-to-SQL: question representation, in-context learning, and supervised fine-tuning. Through a systematic evaluation of existing prompt engineering methods, the authors introduce a novel inte-

[4]<https://yale-lily.github.io/spider>
[5]<https://bird-bench.github.io/>

grated solution, DAIL-SQL, that comprises three primary components: question representation, example selection, and example organization. Question representation involves using a technique that contains complete database information, including primary and foreign keys. Example selection is conducted through RAG based on the similarity between questions and SQL queries. The example organization is implemented to preserve the mapping between questions and SQL queries. DAIL has another extended model that uses the self-consistency method (DONG et al., 2023b). Both models demonstrated impressive execution accuracy on the Spider dataset and outperformed Text-to-SQL with over 86% accuracy. However, at BIRD, performance declined to approximately 57%.

LLMs may perform better through context learning (DONG et al., 2023a), i.e., learning by analogy, in which only a few examples are provided in the input prompts. DIN-SQL (POURREZA; RAFIEI, 2023) explores decomposing Text-to-SQL tasks to enhance Large Language Models' (LLMs) performance by providing some examples statically to the prompt. The strategy outlines key modules: schema linking, query classification and decomposition, SQL generation, and self-correction. Breaking down the Text-to-SQL task into these submodules and using context learning through a few examples passed in the prompt improved LLMs' performance. This strategy had an accuracy of 85.3% on Spider, but dropped to 55.9% on BIRD.

The examples provided at the DIN-SQL prompt are static. There are other Text-to-SQL models that use retrieval-augmentation generation (RAG) to provide examples dynamically. Guo et al. (2023b) presented an approach to enhance the generation of SQL queries from natural language questions, employing RAG in the prompts and a dynamic revision chain. The proposed method incorporates sample-aware demonstrations to refine SQL query generation and employs strategies to assist in retrieving questions with similar intents. Experimental results demonstrate that the proposed approach outperformed reference models across three Text-to-SQL benchmarks.

There are repositories that contain the best Text-to-SQL tools and how to leverage LLMs to achieve Text-to-SQL analysis. The Awesome Text2SQL (Eosphoros AI, 2024) lists the best-performing Text-to-SQL tools on WikiSQL, Spider (Exact Match and Exact Execution), and BIRD (Valid Efficiency Score and Execution Accuracy) benchmarks. The DB-GPT-Hub (Eosphoros AI, 2023) encompasses several stages, including data collection, data preprocessing, model selection and construction, and fine-tuning of model weights. The project aims at realizing automated question-answering capabilities based on databases, allowing users to execute complex database queries using natural language descriptions.

There are models trained specifically for the purpose of converting text to SQL. Defog's SQLCoder (Defog AI, 2023) is a specialized Text-to-SQL model, open-sourced under the Apache-2 license, trained on more than 20,000 human-curated questions. These questions were based on ten different schemas. The training dataset consisted of prompt-completion pairs, encompassing several schemas with varying difficulty levels, whereas the evaluation dataset featured questions from novel schemas. The fine-tuning process occurred in two stages: the base model was first refined using easy and medium questions, and then further fine-tuned on hard and extra-hard questions to yield SQLCoder. The latest model, sqlcoder-70b-alpha, features 70B parameters and achieved 93% accuracy over an evaluation framework, SQL-Eval, outperforming the gpt-4 and gpt-4-turbo models for SQL generation tasks from natural language queries.

3
Background and Definitions

This chapter provides an overview of the main concepts related to this dissertation. Section 3.1 approaches Natural Language Queries and Interfaces. Section 3.2 covers Large Language Models concepts. Section 3.3 describes the OpenAI family of GPT models. Finally, Section 3.4 addresses a generic LLM framework, LangChain.

3.1
Natural Language Queries and Interfaces

A Natural Language (NL) query is simply a sentence S in a natural language, such as "What is the character of Meryl Streep in the movie Out of Africa?". In this context, the use of NL in human-computer interaction presents itself as an appealing option.

Natural Language Interfaces (NLI) are a type of human-computer communication interface carried out through the use of sentences and phrases, similar to those used by humans in everyday conversations, acting as commands for computational systems.

At the core of NLI systems is their capability to comprehend and interpret a user's query articulated in Natural Language. These systems then generate a structured query, which is subsequently executed against a structured data source. However, the use of NL as a query language implies the emergence of some inherent problems, such as query ambiguity, linguistic variability, high processing costs, and the need for manual adaptation of the tool to the specific usage domain (KAUFMANN; BERNSTEIN, 2007).

Various NLI systems and architectures have been developed, each aiming to tackle the challenges mentioned above. These systems can be categorized according to their methods for interpretation and generation of structured queries. One of the predominant paradigms in NLI system architectures is Text-to-SQL (QUAMAR et al., 2022).

3.1.1
Text-to-SQL

As already mentioned in the introduction, Text-to-SQL refers to the task defined as: *"Given a database D and a natural language sentence S, generate an SQL query Q expressing S"*. Systems designed for the Text-to-SQL task use a trained model (machine learning or deep learning) that translates a natural language query, a sequence of words, to a structured query such as SQL as an output to be executed against a schema. The natural language input representation takes the natural language query and schema elements to generate feature vectors. The encoder takes the input feature vectors of the query text and schema elements as input and learns an intermediate representation of this combined input. Finally, this intermediate representation is decoded by the decoder to generate the SQL query.

3.2 Large Language Models

Large Language Models (LLMs) have emerged as a significant milestone in the field of Natural Language Processing (NLP). The fundamental architecture of LLMs is grounded in deep neural network structures characterized by an expansive parameter count. These models undergo training on vast corpora of unlabeled text, employing self-supervised or semi-supervised learning methodologies with the primary objective of predicting the subsequent word in a given context (ABDULLAH; MADAIN; JARARWEH, 2022).

The architectural inspiration for LLMs is derived from Transformers (VASWANI; AL., 2017). Unlike conventional linear sequence processing, Transformers use an attention mechanism to assess different segments based on their relevance to the ongoing task, thereby representing a position through a weighted combination of other input positions.

LLMs have many parameters that govern their learning and text generation processes. The number of parameters is indicative of the model size, with a larger number of parameters indicating a more complex model and enhanced data processing capabilities. However, they also entail higher computational costs for training and deployment. These parameters are used to learn the relationships between words and phrases in the training data (DESAI, 2024).

The training process involves the systematic masking of words in the text during training. The model is then tasked with predicting the missing words based on the outer context, an iterative process that contributes to the development of a comprehensive skill set and pattern recognition within the model. Model parameters are adjusted during training to minimize the error between predicted and actual output. These acquired abilities are leveraged at inference time, enabling the model to swiftly adapt to or recognize the desired task (BROWN et al., 2020).

However, an LLM reflects the data it was trained with (SANDHU, 2024). In particular, an LLM suffers from the “temporal generalization problem” - capturing facts that change over time - and the “factual grounding problem" - capturing specific facts. And it also produces untrue information, commonly called "hallucinations" or "fabulations" (ZIEGLER; BERRYMAN, 2023). These problems limit reasoning and can generate inappropriate content. To circumvent these limitations, the user may fine-tune the LLM, that is, re-train it with more examples, or he may adopt few-shot learning, that is, add a few examples in a dialog interaction so that the model can capture what the user is trying to do and generate a plausible completion.

Below are some well-known LLM families (PINHEIRO. et al., 2023):

GPT - Generative Pre-Trained Transformer: GPT 3.5 (announced on November 30th, 2022) and GTP 4 (announced on March 14th, 2023), respectively support ChatGPT and ChatGPT Plus (OPENAI, 2023b).

LLaMA - Large Language Model Meta AI: LLaMA (announced on February 23rd, 2023) was developed by Meta (META, 2024).

LaMDA - Language Model for Dialogue Applications and **PaLM** - Pathways Language Model: LaMDA (announced on May 18th, 2021) and PaLM (announced in March 2023 and upgraded in May 2023) support Google's Bard (GOOGLE, 2024).

The terms of use of GPT 3.5 prohibit developing models that compete with OpenAI; and LLaMA has a non-commercial use license.

3.2.1 LLM Parameters

Interactions with the LLM occurs either directly or through an API. The main parameters that can be configured to achieve distinct outcomes for the an input are:

temperature : a lower temperature results in more deterministic outcomes, as the next probable token with higher likelihood is consistently chosen. Conversely, increasing the temperature introduces more randomness, promoting diverse or creative outputs by essentially elevating the weights of other possible tokens. A lower temperature value may be preferable for fact-based quality control tasks, encouraging more factual and concise responses. Conversely, for creative tasks such as poem generation, it might be beneficial to raise the temperature value.

top_p : alters the way the model selects tokens for output. Tokens are chosen from the most probable (considering the top-K) to the least probable until the sum of probabilities equals the top-P value. For instance, if tokens A, B, and C have probabilities of 0.3, 0.2, and 0.1, respectively, and the top-P value is 0.5, the model will select either A or B as the next token using temperature and exclude C as a candidate. Specify a lower value for fewer random responses and a higher value for more random responses.

top_k : changes how the model selects tokens for output. A top-K of 1 means the next token selected is the most probable among all tokens in the model's vocabulary (also called greedy decoding), while a top-K of 3 means the next token is chosen from the top three most probable tokens using temperature. At each token selection step, the top-K tokens with the highest probabilities are sampled. Then, tokens are filtered based on the top-P value, with the final token selected through temperature sampling.

max_tokens : This is the maximum number of tokens that LLM generates in the response. A lower value is specified for shorter responses and a higher value for potentially longer responses.

OpenAI LLMs (see Section 3.3), through their API, offer a parameter that specifies the number of responses that the model should return in a single request, and the user is only charged for the extra tokens of the extra output cases.

3.2.2 Prompt

A *prompt* is an NL text describing the task that an LLM should perform. Prompts are used to interact and instruct models. An LLM works to predict the next best group of letters, called "tokens", from the prompt. LLMs generate outputs by leveraging the comprehensive content of all publicly available documents, striving to predict the next token within a given document and will only stop once it has reached a maximum threshold of tokens.

A prompt may encompass any of the following components (SARAVIA, 2022):

Instruction : a specific task or directive that is intended for the model to execute.

Context : may involve external information or additional context that can guide the model towards more optimal responses.

Input data : constitutes the input or question for which the user seek an answer.

Output indicator : denotes the type or format of the output.

Not all components are requisite for a prompt, and the format is contingent upon the task at hand.

LLMs can operate in two prompt modes: *chat* and *instruct*.[1] The Instruct mode is designed for natural language processing tasks in specific domains, where the LLM follows the user's instructions and produces the desired output. Figure 3.1 illustrates an example of an Instruct prompt (non-highlighted text) to an LLM asking it to translate a sentence from English to Turkish with the output of the LLM highlighted in green.

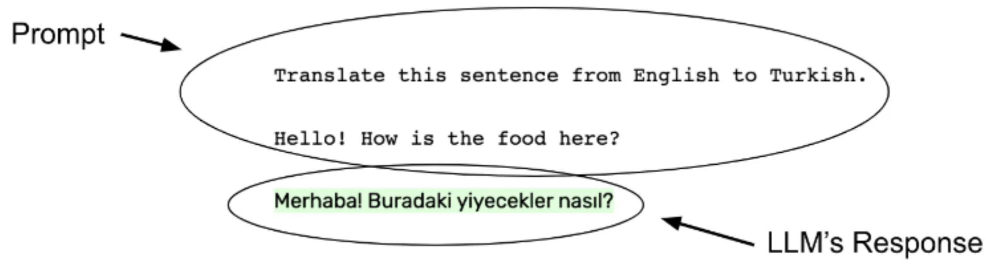

Figure 3.1: An instruct prompt to translate a sentence from English to Turkish. From (HABIB; OZDEMIR, 2022)

The Chat mode is designed for conversational contexts, where the LLM responds to the user's messages in a natural and engaging way. Figure 3.2 shows an example of an chat prompt to capture information about the Eiffel Tower.

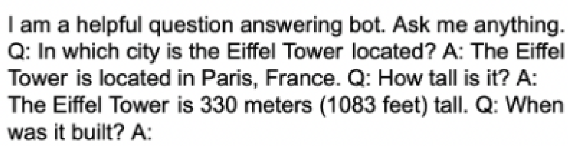
I am a helpful question answering bot. Ask me anything. Q: In which city is the Eiffel Tower located? A: The Eiffel Tower is located in Paris, France. Q: How tall is it? A: The Eiffel Tower is 330 meters (1083 feet) tall. Q: When was it built? A:

Figure 3.2: An example of an chat prompt. From (ANDERSON, 2023)

3.2.3 Prompt Engineering

Prompt engineering is the process of structuring text that can be interpreted and understood by a generative AI model . It offers a natural and intuitive interface for humans to interact with LLMs (ZHOU et al., 2023).

[1] <https://wowdata.science/chat-and-instruct-modes-in-llms/>

Prompt engineering is used to improve the security of LLMs and create new features, prevent hallucinations, and also augment models with domain knowledge and external tools. There are several prompt engineering techniques such as *Zero-shot learning*, *Few-shot learning*, *Chains-of-Thought*, *Retrieval-Augmented Generation* (RAG), *Self-Consistency* and *ReAct*, discussed in the sections that follow (SARAVIA, 2022).

3.2.3.1 Zero-shot learning

Zero-shot learning refers to the ability of a model to perform a task without any specific training on that task (LIU et al., 2023). This capability is particularly valuable as it allows models to adapt to new tasks or scenarios without extensive retraining, making them more versatile and efficient in handling a wide range of tasks.

Consider a sentiment analysis scenario, a traditional machine learning task. One might label paragraphs with sentiment classifications and train a model to predict classifications based on input paragraphs. However, such a model lacks adaptability; any modification, such as adding a new class or changing the task to summarization, necessitates retraining. Contrastingly, an LLM does not require retraining. By properly phrasing queries, the model can be instructed to classify or summarize without explicit task-specific training. While it may struggle to classify a paragraph into ambiguous categories like A or B, it can successfully classify into broader sentiments like “positive” or “negative” due to its understanding of these concepts acquired during training (TAM, 2023).

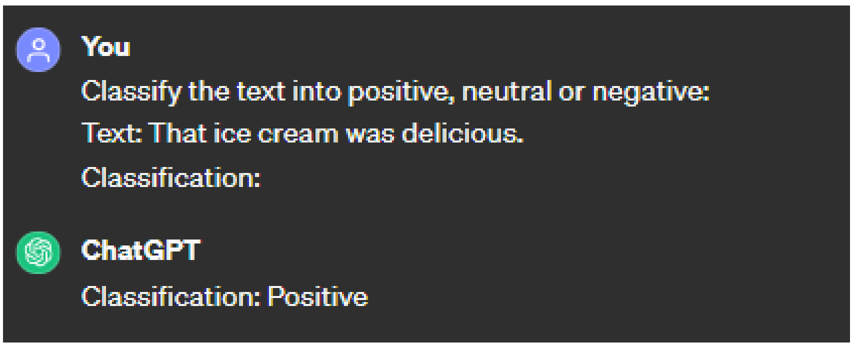

Figure 3.3: An Zero-Shot Prompting example

The prompt in Figure 3.3 lacks specific examples, showcasing the zero-shot capabilities in action – the model accurately provided a single-word answer, “positive,” showcasing its ability to understand the sentiment conveyed by the term “awesome.” This understanding is attributed to the initial instruction, “Classify the text into positive, neutral, or negative.” This example illustrates how the model can effectively respond based on its comprehension of given instructions.

3.2.3.2
Few-shot learning

While large language models showcase remarkable zero-shot capabilities, they still fall short on more complex tasks when relying solely on zero-shot configurations. The few-shot prompt technique can be employed to facilitate contextual learning, where examples provided in the prompt guide the model toward better performance. These demonstrations act as conditioning for subsequent examples, guiding the model to generate desired responses (SARAVIA, 2022).

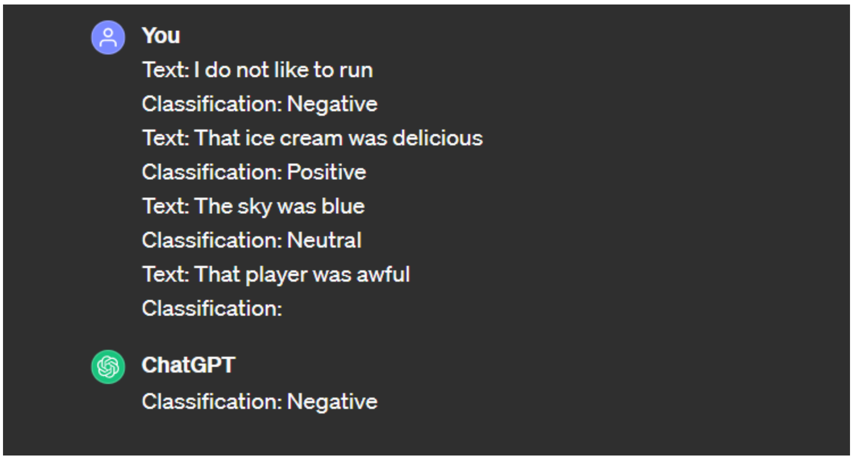

Figure 3.4: An Few-Shot Prompting example

In the Figure 3.4, no explicit instruction is given on what the model should do. However, with the inclusion of some examples, the model can infer how to respond. It is noteworthy that the model generates a response "Negative," aligning with the examples provided.

According to Tam (2023), the randomness inherent in the model may lead to variations in results, and attempting to reproduce the exact result may not be feasible. This way, different results can be observed with each model run.

3.2.3.3
Chains of Thought

Despite the power of Large Language Models (LLMs), even with the provision of some examples, these models struggle with more complex tasks. Notably, reasoning problems, such as arithmetic or commonsense reasoning, are recognized as challenging (WOLFE, 2023). In order to enhance the language models' ability to perform such tasks, Wei et al. (2023) introduced the Chain of Thought (CoT) technique. This approach involves inserting several example solutions into the LLM's prompt (as in Section 3.2.3.2) and breaking down the complex problem into intermediate steps. It provides a logical sequence of reasoning to guide models in generating more precise and coherent responses. An example prompt is shown in Figure 3.5. It is typical to decompose the problem into intermediate steps and solve each (highlighted in blue) before giving the final answer (highlighted in green).

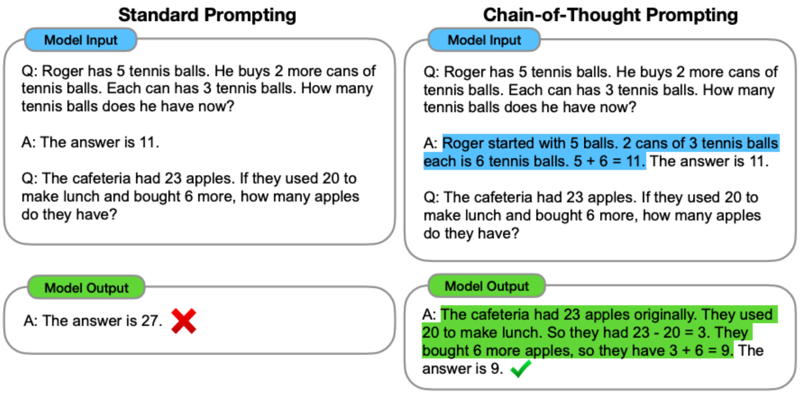

Figure 3.5: An example of Chains of Though prompting. From (WEI et al., 2023)

.

Another approach combines a zero-shot prompt with CoT (KOJIMA et al., 2023), essentially involving adding *Let's think step by step* to the original prompt. Figure 3.6 illustrates an example, particularly useful when it is not necessary to provide many examples in the prompt. Initially, a "reasoning" prompt is used to extract a full reasoning path from a language model. Then, a second "answer" prompt is employed to extract the answer in the correct format from the reasoning text.

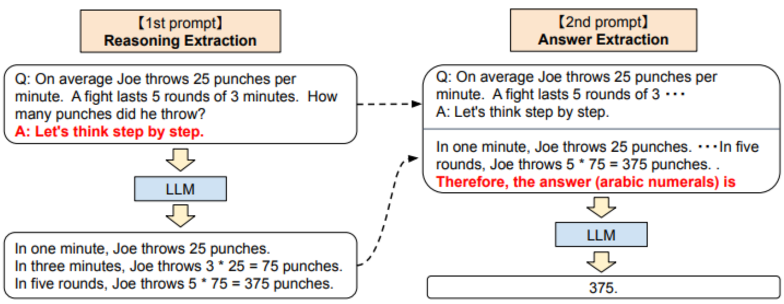

Figure 3.6: Full pipeline of Zero-shot-CoT. From (KOJIMA et al., 2023)

.

For sufficiently large models (models larger than 100 billion parameters), CoT approach significantly enhances the complex reasoning capabilities of LLMs in tasks involving arithmetic, commonsense, and symbolic reasoning (WEI et al., 2023).

3.2.3.4
Self-Consistency

The Self-Consistency method, proposed by Wang et al. (2023), aims at exploring a variety of reasoning paths to enhance answer accuracy in language models. This approach differs from the traditional method used in Chain-of-Thought, allowing for the exploration of multiple reasoning paths and resulting in more precise answers.

The concept involves sampling several reasoning paths through few-shot CoT and using these generations to select the most consistent answer (SARAVIA, 2022). The self-consistency method comprises three steps (WANG et al., 2023):

1. Prompt a language model using CoT prompting.
2. Replace the "greedy decode" in CoT prompting by sampling from the language model's decoder to generate a diverse set of reasoning paths.
3. Marginalize out the reasoning paths and aggregate by choosing the most consistent answer in the final answer set.

As shown in Figure 3.7, a model can generate multiple plausible responses to a mathematical question, some leading to the same correct answer (Outputs 1 and 3). Given that language models are not perfect reasoners, there is a possibility that the model might produce an incorrect reasoning path or make a mistake in one of the steps (e.g., as seen in Output 2). The idea is that correct reasoning processes, even if diverse, tend to exhibit greater agreement in their final answer compared to incorrect processes.

3.2.3.5
Retrieval-Augmented Generation (RAG)

RAG, introduced in (LEWIS et al., 2021), enhanced generative tasks. RAG involves an initial retrieval step where the LLMs query an external data source to obtain relevant information before proceeding to answer questions or generate text (GAO et al., 2024).

RAG operates in two main phases: retrieval and content generation. Figure 3.8 illustrates a RAG operation. During the retrieval phase, the process

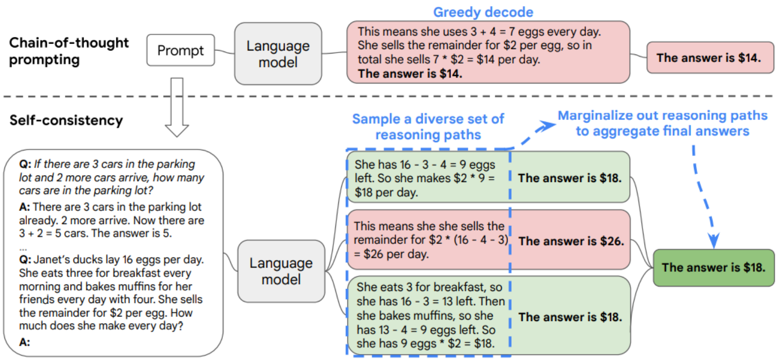

Figure 3.7: Example of Self-Consistency. From (WANG et al., 2023)

.

involves encoding documents and user input into vectors of real numbers, commonly referred to as embeddings. These document embeddings are subsequently stored in a vector database. Algorithms search for and retrieve relevant snippets of information based on the user's prompt or question in vector database. Finally, there is a concatenation of relevant documents with the user prompt, augmenting the prompt with external information. In the generative phase, the LLM uses the retrieved information and its internal representation of training data to synthesize a tailored answer for the user (SAFJAN, 2023).

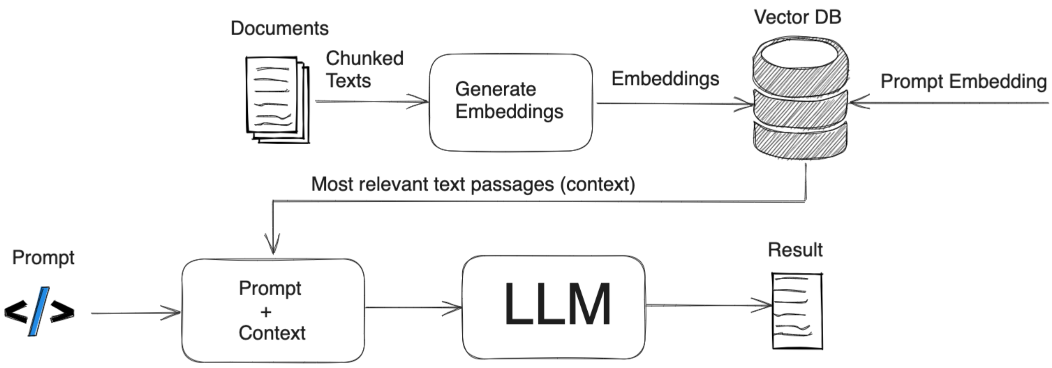

Figure 3.8: RAG operation - Augmenting prompt with external information. From (SAFJAN, 2023)

The dynamic retrieval of information from knowledge bases during the inference phase allows RAG to change what a pretrained language model knows, preventing the model from being retrained with new documents, and finally, accessing and extracting up-to-date information and then use them to produce the results (RIEDEL et al., 2020).

3.2.3.6 ReAct

Yao et al. (2023) introduced a framework named ReAct (Reason + Act), which combines reasoning and action generation in LLMs. ReAct aims at enhancing performance across various tasks by interleaving reasoning traces and task-specific actions.

ReAct prompts LLMs to generate verbal reasoning traces and actions for a given task, enabling dynamic reasoning to formulate, maintain, and adjust plans for action. Additionally, it assists the model in handling exceptions, facilitates interaction with external environments (e.g., Wikipedia) to incorporate additional information into the reasoning process (SARAVIA, 2022).

In Figure 3.9, an example of ReAct and the associated steps for performing question answering are illustrated. The LLM initiates reasoning and takes the action *Search[Apple Remote]*. This command interacts with an external tool that retrieves information in *Obs1*. Based on this observation, the LLM engages in further reasoning and formulates another action plan, *Search [Front Row]*. Since this is not found in the external tool, but similar terms exist, the model generates a new action plan accordingly. Finally, the process concludes with the model providing the answer.

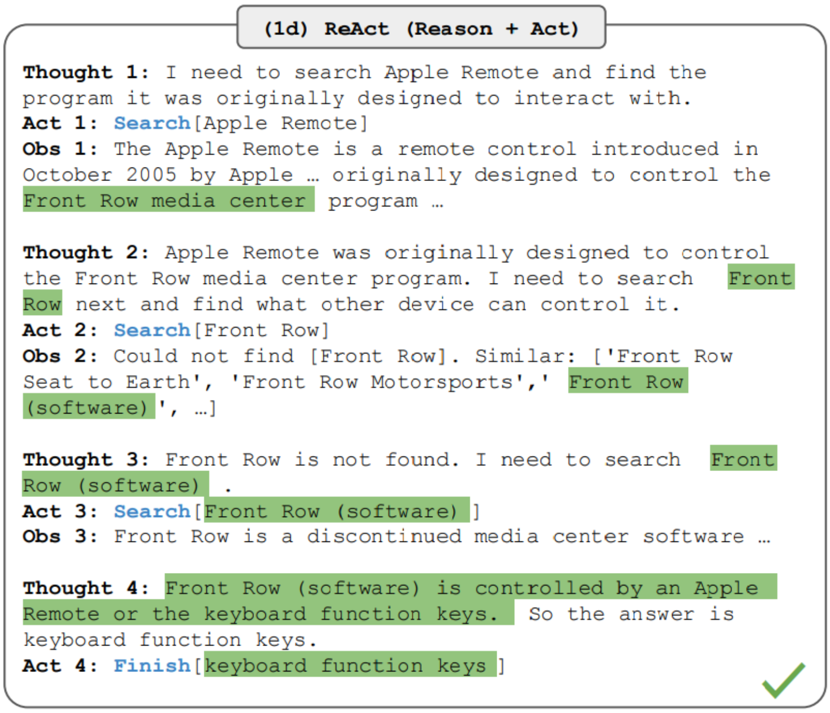

Figure 3.9: ReAct and the different steps involved to perform question answering. From (YAO et al., 2023)

3.2.4 Fine-Tuning

In many LLM use cases, prompt engineering is a sufficient and least resource-intensive approach. However, there are situations in which just using prompting techniques in a model will not solve the problem. In such situations, one of the options is to fine-tune the LLM. Fine-tuning LLM involves the additional training of a pre-existing model, which has previously acquired patterns and features from an extensive dataset, using a smaller, domain-specific dataset (MEHRA, 2023).

Figure 3.10 presents a simplified way of the application of the technique. Into the steps involved in LLM fine-tuning, initially, a pre-trained model is chosen, and a dataset relevant to the specific task is gathered, ensuring it is properly labeled or structured for the model to learn. The actual fine-tuning takes place, where the selected pre-trained model is adjusted based on the specific dataset, which may be related to a particular domain or application. This task-specific adaptation enables the model to specialize for the provided context while simultaneously retaining the general language knowledge acquired during pre-training (DAS, 2024).

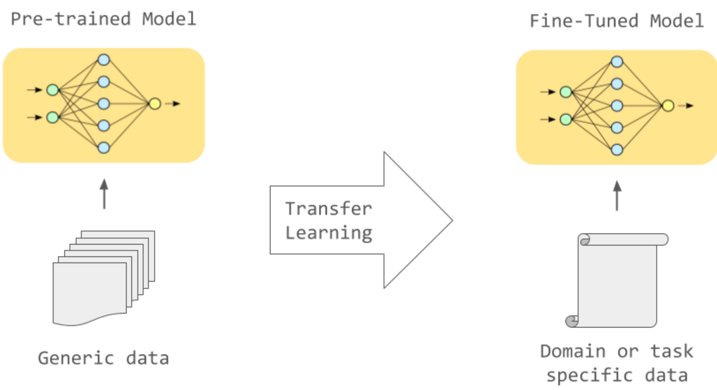

Figure 3.10: Fine-tuning tutorial to a general purpose. From (MEHRA, 2023)

There are two fundamental approaches to fine-tuning LLMs (DAS, 2024): Full Fine-tuning (FFT) and Parameter Efficient Fine-Tuning (PEFT). FFT involves reconfiguring all the parameters of the model during the training process for a specific task. This means updating all weights and layers of the model with the training data, creating a new version with improved capabilities. However, this process may require a significant amount of data and computational resources.

On the other hand, PEFT represents a more efficient form of instruction fine-tuning compared to FFT. This approach maintains the original LLM weights and updates only a subset of parameters, effectively "freezing" the rest. PEFT aims to optimize the performance and adaptability of language models across various applications (HU et al., 2023). Among the methods for achieving Parameter Efficient Fine-Tuning, there are approaches such as Low-Rank Adaptation (LoRA) (HU et al., 2021) and QLoRA (DETTMERS et al., 2023).

The models can be trained through unsupervised, supervised fine-tuning (SFT) or combining SFT with RLHF (OUYANG et al., 2022), utilizing human preferences as a reward signal to improve LLM's ability to follow instructions effectively.

3.3 The OpenAI and GPT family

This section discusses the GPT-based language models and the tools developed by OpenAI.

OpenAI is an AI research and deployment company, focusing on researching generative models, built using Deep Learning, a technology that uses large amounts of data to train AI systems to perform specific tasks. The text models developed by OpenAI are advanced language processing tools capable of generating, classifying, and summarizing text. The company's research on generative modeling extends to image and audio processing (OPENAI, 2023b).

3.3.1 GPT-based models

The OpenAI Research API offers several models of the GPT family, with different capabilities and prices.[2] OpenAI models are non-deterministic; that is, identical inputs can yield different outputs. Setting the *temperature* to zero will make the outputs mostly deterministic, but a small amount of variability may remain. Basically, the models are:

GPT-3.5 can understand and generate natural language or code. The model has 175 billion parameters.

GPT-4 is a large multimodal model that can solve difficult problems with greater accuracy, and is optimized for chat but works well for traditional completions tasks. The model has approximately 1.76 trillion parameters.

[2]The relationship between the models the OpenAI API offers and those mentioned in the dissertation is described at https://platform.openai.com/docs/models/overview

However, they also have their variations, with a larger context window, more updated training data, and new features. Table 3.1 summarizes some characteristics of the basic models, such as the date of the training data, the maximum number of tokens allowed in the context window, and prices for each one.

Model	**Description**	**Max tokens**	**Training data**	**Input Tokens Price (US$/1K tokens)**	**Output Tokens Price (US$/1K tokens)**
gpt-4	More capable than any GPT-3.5 model, able to do more complex tasks, and optimized for chat.	8,192 tokens	Up to Sep 2021	$0.03	$0.06
gpt-4-32k	Same capabilities as the base GPT-4 model, but with 4x the context length.	32,768 tokens	Up to Sep 2021	$0.06	$0.12
gpt-3.5-turbo	Most capable GPT-3.5 model and optimized for chat at 1/10th the cost of text-davinci-003.	4,096 tokens	Up to Sep 2021	$0.0015	$0.0020
gpt-3.5-turbo-16k	Same capabilities as the base gpt-3.5-turbo model, but with 4x the context length.	16,385 tokens	Up to Sep 2021	$0.0030	$0.0040

Table 3.1: Some OpenAI models available.

In January 2024, OpenAI launched the gpt-4-turbo model, trained with data until December 2023, larger context windows of 128k, and lower prices (OPENAI, 2024). This model completes tasks like code generation more thoroughly than the previous model and is intended to reduce cases of "laziness" where the model does not complete a task.

3.3.2 ChatGPT

ChatGPT from OpenAI, released in November 2022, is a sibling model to InstructGPT, which is trained to follow an instruction in a prompt and provide a detailed and optimized response.[3] According to OpenAI, "*ChatGPT provides articulated answers across several knowledge domains, but uneven factual accuracy has been identified as a significant drawback, and sometimes it writes plausible-sounding but incorrect or nonsensical answers*" (OPENAI, 2024) The name "ChatGPT" combines "Chat", denoting its chatbot capabilities, and "GPT", an acronym for Generative Pre-trained Transformer (LOCK, 2022).

The model was trained using Reinforcement Learning from Human Feedback (RLHF)(LI; YANG; WANG, 2023), following similar methods as InstructGPT but with slight variations in data collection and training data, include "Internet phenomena", software documentation, and code. Initial training involved supervised fine-tuning, where AI trainers played both user and assistant roles in conversations, aided by model-written suggestions.

[3] <https://openai.com/research/instruction-following>

This new dialogue dataset was combined with the InstructGPT dataset, transformed into a dialogue format (OPENAI, 2023b).

To create a reward model for reinforcement learning, comparison data was collected by having AI trainers rank two or more model responses for quality. Conversations with the chatbot provided the basis for selecting a model-written message, sampling alternative completions, and obtaining rankings. The model underwent fine-tuning using Proximal Policy Optimization through several iterations.

ChatGPT is fine-tuned from a GPT-3.5 series model, and, as the last column of Table 3.1 suggests, it has limited knowledge of events that occurred after September 2021. Both, ChatGPT and GPT-3.5, were trained on Azure AI supercomputing infrastructure.

In fact, OpenAI used outsourced Kenyan workers earning less than USD 2.00 per hour to label “toxic content” (PERRIGO, 2023).

Limitations include the occasional generation of plausible-sounding but incorrect or nonsensical answers. The model is sensitive to input phrasing variations and may be excessively verbose, with biases from training data contributing to these issues. Additionally, the model tends to guess user intent instead of asking clarifying questions for ambiguous queries.

Efforts have been made to address inappropriate requests, but the model may still respond to harmful instructions or exhibit biased behavior. The Moderation API is utilized to warn or block unsafe content, but false negatives and positives can occur. OpenAI seeks user feedback to enhance the system continually.

The reader is invited to test ChatGPT by running the following three generic knowledge queries, for which ChatGPT produces answers of varying quality: *(Correct answer)* “How Liz Taylor, Richard Burton, and John Hurt are related?”; *(Incorrect answer)* “Where did D. Pedro I of Brazil died?”; *(Failed answer due to lack of data)* “Tell me about the coronation of Charles III of England.”

Figure 3.11 shows the ChatGPT interface where the model is asked to generate a short email and it responds with an email template.

As for training a model, the user can adopt few- shot learning (see Section 3.2.3), passing examples in the prompts, or fine-tune the model, as mentioned in Section 3.2.4

ChatGPT Plus, the paid version of ChatGPT, was released on March 14th, 2023, uses GPT 4 (OPENAI, 2023a), at a cost of USD 20.00 per month (on May 2023). According to the OpenAI annoucement, “*GPT-4 is 82% less likely to respond to requests for disallowed content, is 40% more likely*

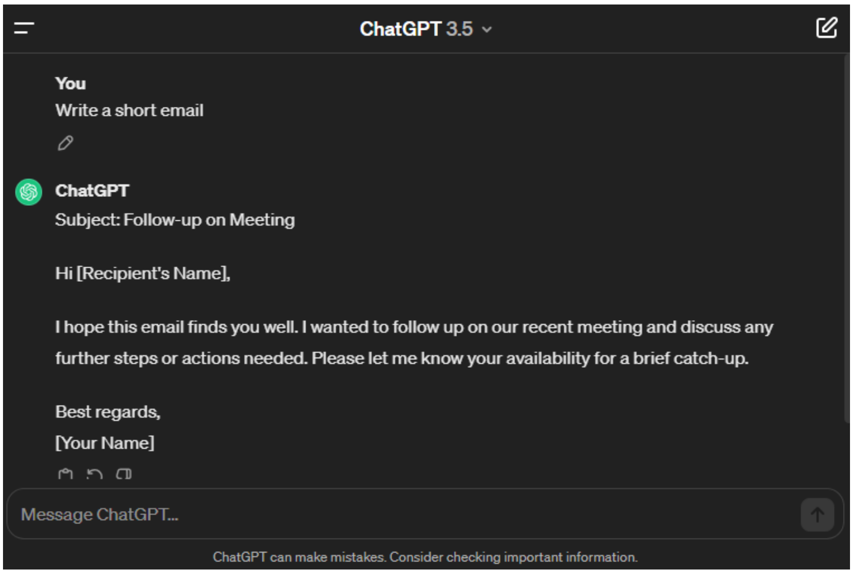

Figure 3.11: ChatGPT's Interface.

to produce factual responses than GPT-3.5, and can take images as input" (OPENAI, 2023a).

3.4 LangChain

LangChain is a framework for developing applications powered by language models. This framework facilitates the creation of applications characterized by their (LANGCHAIN, 2024):

1. **Context-Awareness:** Establishing a connection between a language model and various sources of context, such as prompt instructions, few-shot examples, or contextual content, enables the application to ground its responses effectively.

2. **Reasoning Capability:** Leveraging a language model for reasoning, including determining how to respond based on provided context and deciding on appropriate actions, empowers applications to navigate complex scenarios.

At the core of LangChain is the fundamental building block of invoking an LLM on a given input. For example, Listing 1 shows how to import the LLM wrapper, adjust parameters such as temperature for generating more diverse outputs, and finally, invoke the LLM (GPT-4 model from OpenAI) with a specific input.

Listing 1: Example of Using LangChain to Invoke an LLM

```
from langchain.llms import OpenAI
llm = OpenAI(model=‘‘gpt-4’’, temperature=0.9)
text = ‘‘Tell me a short joke about cakes.’’
print(llm(text))
```

While the example provided is rudimentary, in real-world applications, interactions with the LLM are not isolated but rather constitute a series of steps where intermediate results necessitate logical processing for initiating the subsequent step. This sequence of combined interactions is termed as *Chains*, which typically involves integrations with one or more model providers, data storage systems, and APIs, among others. These modules can be combined to build more intricate applications or used individually for simpler applications.

LangChain proves instrumental in designing applications such as personal assistants, chatbots, document-based question-answering (QA), summarizing lengthy documents, extracting structured information from unstructured text, querying tabular data, and performing Text-to-SQL tasks.

4
LLM-based Text-to-SQL strategies

This chapter addresses the strategies used to explore LLM features for the Text-to-SQL task. The strategies were implemented with the help of the LangChain framework. It outlines three “LangChain-based Text-to-SQL strategies for Text-to-SQL”, two models that performed well over the Spider benchmark, “C3 + ChatGPT+ Zero-Shot”(DONG et al., 2023b) and “DIN-SQL”(POURREZA; RAFIEI, 2023), and finally “C3+DIN”, a new strategy that combines components of C3 and DIN-SQL. A

4.1
LangChain-based Strategies for Text-to-SQL

LangChain offers some predefined chains[1] and agents for Text-to-SQL, that are compatible with any SQL dialect supported by SQLAlchemy (e.g., MySQL, PostgreSQL, Oracle SQL, Databricks, SQLite and other DBMSs).[2] The SQL chains and agent have predefined prompts, but can be customized. In simple terms, they operate through three main steps. First, the model converts the user’s NL question into a structured SQL query, making it understandable to the database. Next, the SQL query is executed, fetching the necessary data from the database. Finally, the model responds to the user’s input using the query results obtained.

```
CREATE TABLE "Album" (
    "AlbumId" INTEGER NOT NULL,
    "Title" NVARCHAR(160) NOT NULL,
    "ArtistId" INTEGER NOT NULL,
    PRIMARY KEY ("AlbumId"),
    FOREIGN KEY("ArtistId") REFERENCES "Artist" ("ArtistId")
)

/*
3 rows from Album table:
AlbumId Title   ArtistId
1   For Those About To Rock We Salute You   1
2   Balls to the Wall   2
3   Restless and Wild   2
*/
```

Figure 4.1: Table definitions and example rows in langchain’s prompt for Text-to-SQL task.

[1]<https://python.langchain.com/docs/use_cases/sql/prompting>
[2]<https://www.sqlalchemy.org/>

In addition to including the schema in the prompt, these SQL chains enable the provision of sample data, as illustrated in Figure 4.1. This data can assist an LLM in formulating accurate queries, especially when the data format is not evident. Sample rows are incorporated into the prompt following the column information for each respective table.

Listing 2 shows an example of generating SQL with LangChain. The Chains for SQL use a module named SQLDatabase that inspects the schema, tables, columns and foreign keys in the database. The DDL (Data Definition Language) of the schema is provided at the prompt as context for an LLM in an automated way. This representation of the schema, illustrated in Figure 4.1, is called *Code Representation* by Gao et al. (2023). Using such context in the prompt, the LLM constructs an SQL query for the original NL question, which is then executed and returned by LangChain.

Listing 2: An example of generating SQL with Langchian

```python
from langchain.utilities import SQLDatabase
from langchain_community.llms import OpenAI
from langchain_experimental.sql import SQLDatabaseChain

db = SQLDatabase.from_uri("sqlite:///Chinook.db")
llm = OpenAI(temperature=0, verbose=True)
db_chain = SQLDatabaseChain.from_llm(llm, db, verbose=True)

db_chain.run("How many employees are there?")
```

This dissertation proposes to test and evaluate these chains for LangChain's Text-to-SQL task.

SQLQueryChain and *SQLDatabaseChain* are chains that basically receive the database schema as a prompt and convert an NL question into an SQL statement. They are the simplest and easiest versions, as the schema is automatically extracted from the database, as illustrated in Figure 4.2. Listing 5, located in the Annex A, shows a prompt that begin describing the behavior of the LLM. Subsequently, within the LangChain framework, the prompt is customized with some tips. Then, the instruction establishes the desired format for the model's output. Lastly, the complete database schema and the natural language query are provided. It is possible to include a few sample database instances in the prompt.

SQLDatabaseSequentialChain is a more sophisticated version that employs a sequential chaining of prompts, as illustrated in Figure 4.3. The first prompt, referred to as the *decider chain prompt*, shown in Listing 6, identifies tables related to the user's query without providing the database schema. The second prompt is identical to the prompt in the SQLQueryChain (See Listing

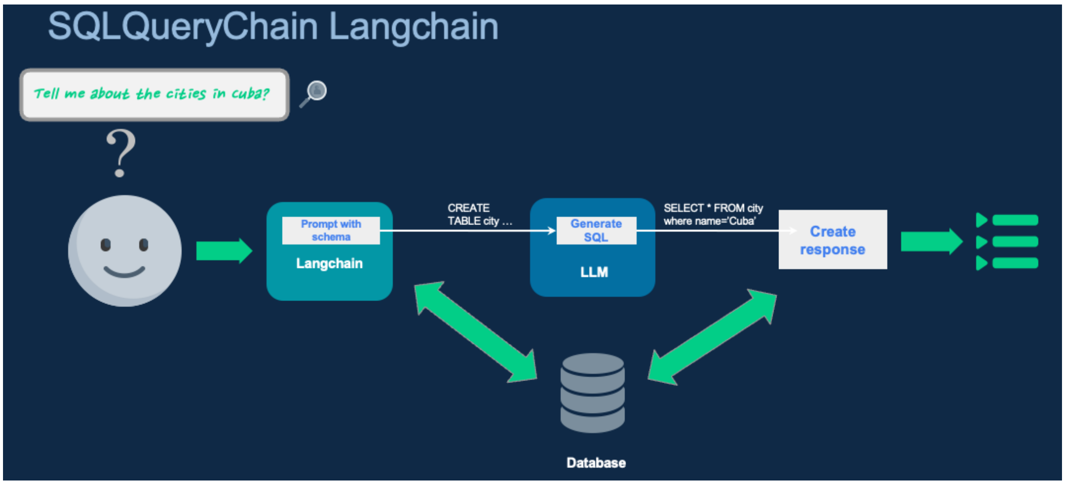

Figure 4.2: SQLQueryChain Overview.

5); however, only the involved tables are provided for the LLM to generate the SQL statement. This helps when the number of tables in the database is large. Similarly to the *SQLQueryChain*, it is possible to include samples of database instances.

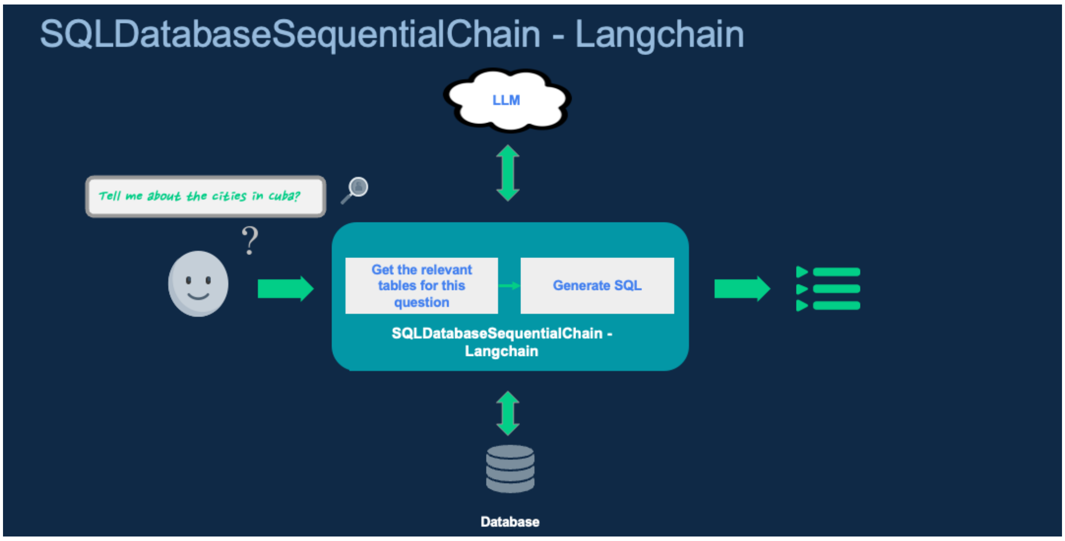

Figure 4.3: SQLDatabaseSequentialChain Overview.

Finally, *SQLAgent* provides a more flexible way of interacting with databases. Figure 4.4 shows an overview of the strategy, where the first two steps are similar to *SQLDatabaseSequentialChain*.

It uses the ReAct strategy (discussed in Section 3.2.3.6) to create SQL statements, that is, for each instance, it invokes a chain of "reasonings" that involve an "Action" (such as "I need to list the tables in this database"), an "Observation" (the tables), and a "Thought" ("I should access details only from table X"), as illustrated in Figure 4.5. This reasoning process iterates,

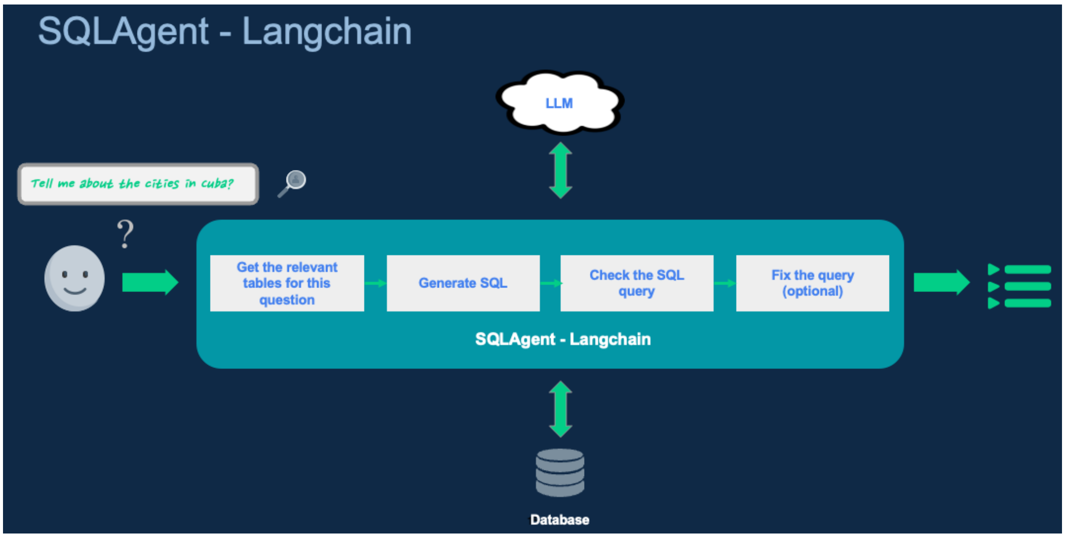

Figure 4.4: SQLAgent Overview.

recovering from errors by running a generated SQL query, capturing the trace, and regenerating it until the desired result is achieved.

Each "Action" that the Agent can use is called a "Tool" and is described in the prompt with the name, description and the function that should be called to take that action. In the context of the SQLAgent, as shown in Listing 7, there are four actions.

The first tool, named *sql_db_list_tables*, enumerates all tables in the database, allowing the LLM to identify those relevant to the query.

The second tool, *sql_db_schema*, takes as input this list of specific tables and returns their schema along with sample rows. It is specified in the prompt that *sql_db_schema* is always invoked prior to *sql_db_list_tables*.

The LLM generates the SQL query, which is then passed to the *sql_db_query_checker* tool, responsible for verifying the correctness of the SQL query before execution.

Finally, the *sql_db_query* tool is called to execute the SQL query. If an error is returned, it is captured, and the query is rewritten for another attempt. After a specified number of attempts, defined as a parameter in the function, the SQLAgent terminates. For the study in this dissertation, the last tool was modified to return the SQL query even if it was incorrect.

4.2 Strategies based on C3 + LangChain

"C3 + ChatGPT + Zero-Shot" (DONG et al., 2023b) (or briefly C3) is a prompt-based strategy, originally defined for ChatGPT. C3 has three key components: *Clear Prompting* (CP); *Calibration with Hints or bias* (CH); *Consistent Output* (CO). Figure 4.6 shows how C3 works.

```
> Entering new AgentExecutor chain...
Action: sql_db_list_tables
Action Input: ""
Observation: mondial_airport, mondial_borders, mondial_city, mondial_cityothername, mondial_citypops, mondial_continent, mondial_country
Thought:I should query the schema of the mondial_country table to see if it contains information about Thailand's area.
Action: sql_db_schema
Action Input: "mondial_country"
Observation:
CREATE TABLE mondial_country (
        name VARCHAR(50 CHAR),
        code VARCHAR(4 CHAR),
        capital VARCHAR(50 CHAR),
        province VARCHAR(50 CHAR),
        area NUMBER,
        population NUMBER,
        meta_repcol VARCHAR(4000 CHAR)
)
Thought:The mondial_country table does contain information about the area of countries, including Thailand.
Thought: I can now construct a query to retrieve the area of Thailand from the mondial_country table.
Action: custom_sql_db_query
Action Input: "SELECT area FROM mondial_country WHERE name = 'Thailand'"
Observation: SELECT area FROM mondial_country WHERE name = 'Thailand'
Thought:I now know the SQL QUERY generated.
```

Figure 4.5: ReAct strategy in SQLAgent.

Clear Prompting addresses two problems: (1) the size of the database schema may exceed the prompt limit; (2) a prompt with too many tables and columns may confuse ChatGPT. Clear prompting then recalls relevant tables and columns (to the NL question) and adds them to the prompt, along with the schema. This component employs Self-Consistency (seen in Section 3.2.3.4), where the LLM generates multiple responses for a given task, clusters them with similar ones, and selects the most consistent response through a voting mechanism.

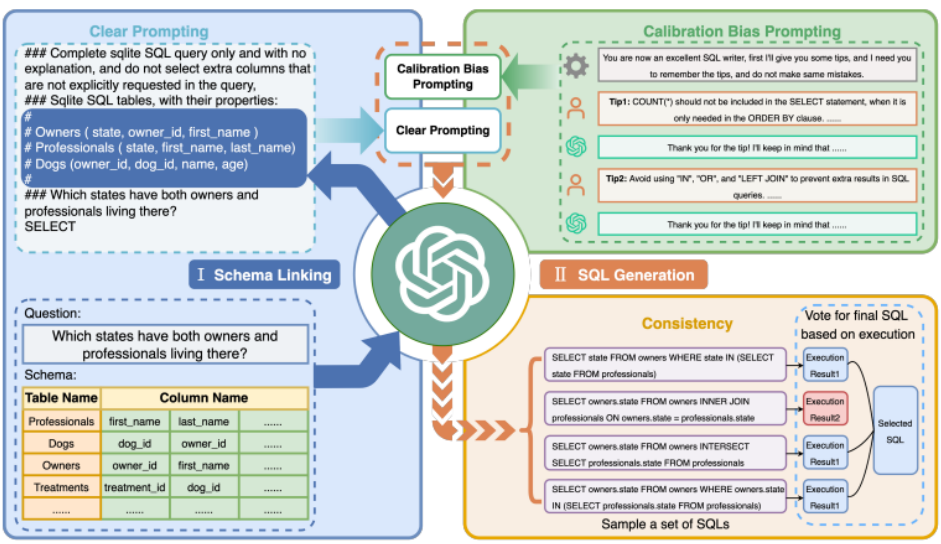

Figure 4.6: The framework of C3. From (DONG et al., 2023b).

Table recall is implemented by a zero-shot prompt that instructs ChatGPT to: (1) recall tables; (2) rank tables based on their relevance to the question; (3) check if all tables have been considered. Listing 8 shows the prompt. It consists of a description of the database schema (only tables and columns) and a query in Natural Language.

Column recall is also a zero-shot prompt that instructs the LLM to: (1) recall columns; (2) rank all columns within each candidate table based on their relevance to the question; (3) give priority to columns that better match NL question words and foreign keys. Listing 9 depicts the prompt. It consists of a description of the database schema (only tables and columns), the foreign keys and a query in natural language.

After running the clear prompting step of the C3 framework on an example with the Mondial database, Listing 3 shows the prompt generated, which is then submitted to the GPTs models to create the SQL equivalent to the user's NL query. In lines 4 and 5, only the tables and columns relevant to the user's NL query are provided, following the Table Recall and Column Recall steps. In line 6, foreign keys are specified, and in line 8 the user's query is presented. The prompt concludes with the "SELECT" statement to ask the LLM to generate the SQL code to complete the query.

Listing 3: Clear Prompting in the C3 approach

```
1 ### Complete oracle SQL query only and with no explanation, and do
      not select extra columns that are not explicitly requested in
      the query.
2 ### Oracle SQL tables, with their properties:
3 #
4 # country (name, code, capital, province)
5 # language (country, name, percentage)
6 # language.country=country.code
7 #
8 ### What are the languages spoken in Poland?
9 SELECT
```

Calibration with Hints or Calibration Bias avoids errors caused by certain biases inherent in ChatGPT: to select columns that are relevant to the question but not required; to use LEFT JOIN, OR, and IN incorrectly. Calibration with Hints then instructs ChatGPT to follow two "debias" hints: (1) select only the necessary column; (2) avoid misusing SQL constructs. Listing 10 shows all hints provided.

Consistent Output tries to avoid the problem that the output of ChatGPT is unstable due to the inherent randomness of LLMs. This step, as illustrated in Figure 4.7, again uses Self-Consistency, which generates several SQL queries, executes the SQL queries on the database, and collects the execution results. It uses a voting mechanism on the results to identify the most consistent SQL query.

At the time of writing, C3 was the sixth strategy listed in the Spider Leaderboard, achieving 82.3% in terms of execution accuracy on the test

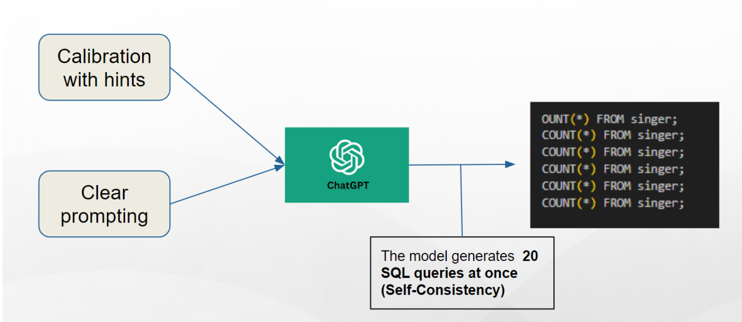

Figure 4.7: The consistency output.

set. It outperformed state-of-the-art fine-tuning-based approaches in execution accuracy on the test set, while using only approximately 1,000 tokens per query. The representation of the C3 strategy follows the "OpenAI Demonstration Prompt" format (GAO et al., 2023). It consists of instructions, table schemas, and questions, where all information is commented using the pound sign "#". To implement this strategy, LangChain was used in each of the steps, as illustrated in Figure 4.8. The SQLDatabase module of LangChain allows connecting to a database and inspecting the existing tables in the schema. This information is stored in metadata that can be manipulated via code. The schema of the Mondial database in the C3 strategy is shown in Listing 17. In fact, any database schema compatible with LangChain can be represented using this approach, as discussed in Section 4.1.

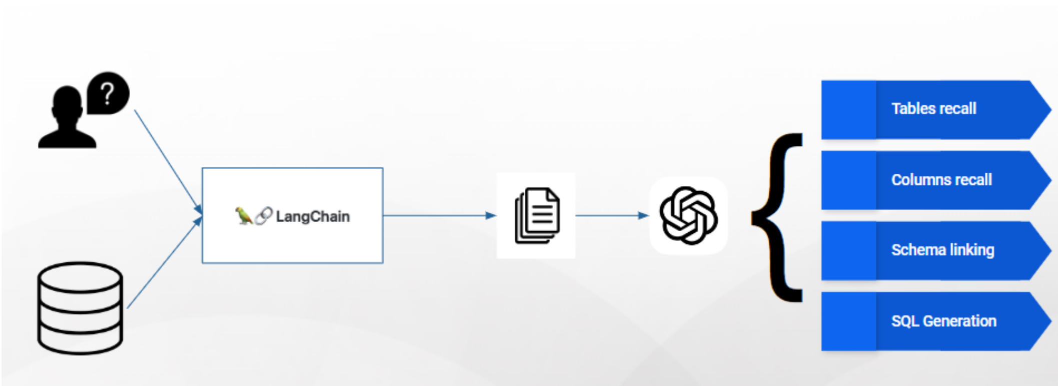

Figure 4.8: C3 strategy with langchain.

4.3 Strategies based on DIN + LangChain

"DIN-SQL" (POURREZA; RAFIEI, 2023) (or briefly DIN) uses only prompting techniques, such as few-shot and chains-of-thought.

It decomposes the Text-to-SQL task into 4 steps: *schema linking*; *query classification and decomposition*; *SQL generation*; and *self-correction*. Figure

4.9 depicts an overview of DIN. Each of the steps follows a specific prompt, but they share a pattern characterized by using a substantial number of tokens, resulting in considerable prompt length. These prompts consist of instructions for LLM behavior, a simplified representation incorporating table schemas, a limited set of examples for few-shot prompting, an NL question preceded by "Q:", and a response prefix "A: Let's think step by step" to initiate the LLM's Chains-of-Thought reasoning process. This prompt structure is referred to as the *Basic Prompt* (GAO et al., 2023).

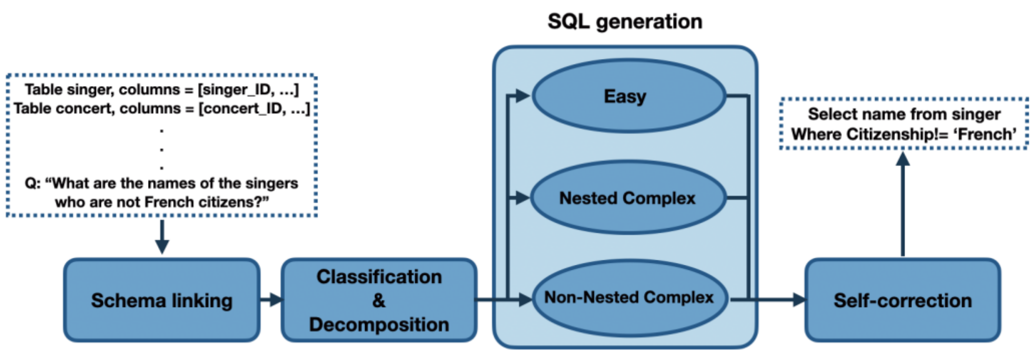

Figure 4.9: DIN-SQL overview. From (POURREZA; RAFIEI, 2023).

Schema Linking includes ten randomly selected samples from the training set of the Spider dataset and follows the chain-of-thought template. The 10 examples are static at the prompt, that is, they are not randomly generated at run time. For the column names mentioned in the question, the corresponding columns and their tables are selected from the schema; possible entities and cell values are also extracted from the question. Furthermore, the prompt tends to be very large, and its length may increase significantly depending on the size of the database. In some cases, it might exceed the token limit imposed by the LLM. The full prompt for this step is shown in Listing 11.

Classification and Decomposition classifies each query into: *easy* – single-table queries that can be answered without joins or nesting; *non-nested* – queries that require joins but no sub-queries; *nested* – queries that require joins, sub-queries, and set operations. This step, whose prompt is demonstrated in Listing 12 also detects the set of tables to be joined, for both non-nested and nested queries, any sub-queries of nested queries.

SQL Generation depends on the query classification. For easy queries, a simple few-shot prompting with no intermediate steps is adequate. For non-nested complex queries, it uses an intermediate representation, removes operators JOIN ON, FROM, GROUP BY, and set operators, and merges the HAVING and WHERE clauses. Briefly, for nested complex queries, it breaks down the problem into multiple steps; the prompt for this class is designed in a way that the LLM should first solve the sub-queries and then use them to generate the final answer. The prompts for the *easy*, *non-nested* and *nested* classes are shown in Listings 13, 14 and 15, respectively.

Finally, *Self-Correction* addresses the problem that the generated SQL queries can sometimes have missed or redundant keywords such as DESC, DISTINCT, and aggregation functions. To solve this problem, the self-correction step instructs the LLM to correct those minor mistakes through a zero-shot setting (see Listing 16), where only the buggy code is passed to the LLM, which is asked to fix the bugs.

When was released, "DIN-SQL + GPT-4" was the top-performing tool listed in the Spider Leaderboard, achieving 85.3% in terms of of execution accuracy.

As in C3, LangChain was used to capture the schema information in the database and be described in the DIN prompts. Hence, any database supported by LangChain can be represented using this approach. Listing 18 shows the schema representation of the Mondial database in the DIN strategy.

4.4 A new strategy: C3-DIN combination + LangChain

"C3+DIN" combines the mechanisms and modules of the C3 and DIN strategies, as depicted in Figure 4.10.

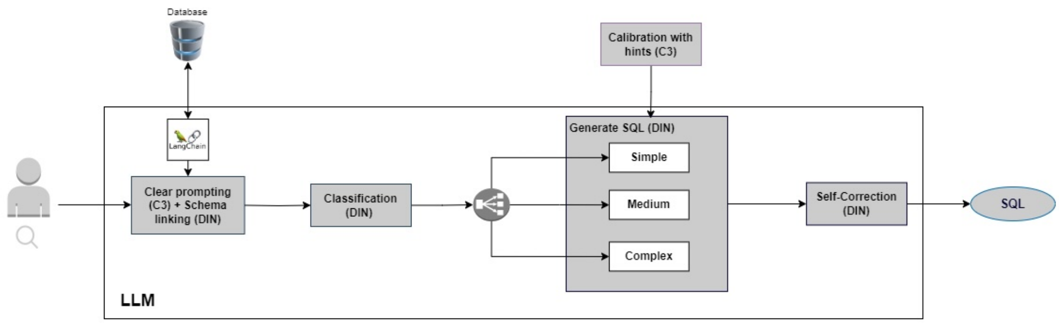

Figure 4.10: C3+DIN combination diagram.

Database schema description. "C3 + DIN" uses LangChain's SQLDatabase module to access database metadata. This metadata is manipulated to represent the schema in an automated way.

Clear Prompting (C3) + Schema Linking (DIN). C3 uses a prompt with much fewer tokens than DIN, that is, cheaper. Indeed, DIN considers all tables for schema linking and uses a chain-of-thought strategy (a series of intermediate reasoning steps) providing examples that assist the LLM during reasoning. "C3 + DIN" combines the two modules to take advantage of the chain of thought strategy used by DIN, but passing only the tables and columns relevant to the query, using the C3 clear prompting strategy. Hence, “C3 + DIN” uses fewer tokens to represent the schema in the prompt.

Classification and Decomposition (DIN). DIN classification is important for SQL generation, since it uses different prompts for each class. Additionally, the prompts help detect tables that should be joined and nested queries by detecting subqueries contained in the main query. “C3 + DIN” then adopts this strategy.

Calibration with hints (C3). C3 calibration incorporates prior knowledge of the LLM. Before the SQL generation step, “C3 + DIN” provides hints to help the model generate SQL queries that align more closely with the desired output.

SQL Generation Module (DIN). Generating a chain of thought (WEI et al., 2023), where some thought-chain demonstrations are provided as stimulus examples, significantly improves the ability of LLMs to perform complex reasoning. Furthermore, the SQL Generation Module from DIN applies different few-shot prompts for each query class through a chain of thought. Thus, “C3 + DIN” adopts this approach to generate SQL code.

Self-correction (DIN). LLMs have hallucination problems, that is, they can generate text that does not make sense (YU et al., 2024). In the Text-to-SQL task, an LLM can generate SQL code with incorrect syntax. Therefore, “C3 + DIN” incorporates DIN’s self-correction mechanism, which seeks to correct the SQL code in a simple way by passing some hints to the model along with the database schema.

5
Experiments to evaluate the effect of schema complexity

This chapter presents experiments to assess how LLM-based Text-to-SQL strategies perform on a challenging scenario characterized by a database with a large complex schema, assuming that the schema vocabulary is close to the vocabulary of the user NL questions. This scenario therefore isolates schema complexity from vocabulary mismatch.

Despite the availability of the benchmark datasets for the Text-to-SQL task described in Section 2.1, and inspired by them, this chapter first introduces the Mondial benchmark, based on a familiar open-sourced database with a complex schema, and a set of 100 NL questions and their translations to SQL queries. Then, it analyses the Text-to-SQL strategies discussed in Chapter 4 using this benchmark. The experiments and benchmark are in a Github repository[1].

5.1
The Mondial Benchmark

Mondial stores geographic data, with a total of 47,699 instances. It features a relational schema with 46 tables, a total of 184 columns, and 49 foreign keys (some of which are multi-column).[2] The Mondial schema is large and complex, but uses familiar terms, such as countries, cities, rivers, etc. Therefore, it meets the goal of this chapter. A version of Mondial, with 34 tables, is also part of the BIRD benchmark.

The benchmark contains a set of 100 NL questions, $L = \{L_1, ..., L_{100}\}$ and the ground truth SQL queries, $G = \{G_1, ..., G_{100}\}$, that were collected from the Internet or defined manually on the Mondial relational schema so that the execution of G_i returns the expected answer to the NL question L_i.

The questions are classified into *simple*, *medium*, and *complex*, that correspond to the easy, medium, and hard classes used in the Spider benchmark (extra-hard questions were not considered). As in the Spider benchmark, the difficulty is based on the number of SQL constructs, so that queries that contain more SQL constructs (GROUP BY, ORDER BY, INTERSECT, nested subqueries, column selections, and aggregators) are considered to be harder. The list of questions contains 33 simple, 33 medium, and 34 complex questions.

[1] <https://github.com/dudursn/text_to_sql_chatgpt_real_world/>
[2] <https://relational.fit.cvut.cz/dataset/Mondial>

Table 5.1 shows some samples of NL questions and their ground truth SQL translations. Column "**ID**" indicates the question identifiers; column "**NL Question**" indicates the test NL questions suggested by experts; column "**Gold SQL**" indicates the ground truth SQL query that represents the answer table for the NL question; and column "**Type**" represents the question type classification.

ID	NL Question	Gold SQL	Type
33	Show the Airports with elevation more than 3000	`SELECT name FROM mondial_airport WHERE elevation > 3000.0`	simple
59	What are the area, elevation and type of lakes in Italy?	`SELECT l.area, l.elevation, l.type FROM mondial_geo_lake gl , mondial_lake l, mondial_country c WHERE (gl.LAKE = l.NAME) AND (gl.COUNTRY = c.CODE) AND c.name = "Italy"`	complex
99	What type of government is Iran?	`SELECT p.government FROM mondial_country c INNER JOIN mondial_politics p ON p.country = c.code WHERE c.name = "Iran"`	medium

Table 5.1: A sample of the benchmark dataset.

5.2 Evaluation metrics

Execution accuracy will be used as the evaluation metrics. This metric provides a more precise estimate of the model's performance since there may be multiple valid SQL queries for a single given question (POURREZA; RAFIEI, 2023). Both the SQL query generated by LLM and the Ground Truth query were executed on an Oracle database, and their results were compared. The *accuracy* of a given text-to-SQL strategy over the benchmark is the number of correct predicted SQL queries divided by the total number of predicted queries, as usual.

5.2.1 Evaluation Procedure

Let $B = (D, \{(P_i, G_i)/i = 1, ..., n\})$ be a benchmark dataset. Let P_i be a predicted SQL query and G_i be the corresponding ground truth SQL query. Let PT_i and GT_i be the tables that P_i and G_i return when executed over D, called the *predicted* and the *ground truth* tables.

Intuitively, P_i is *correct* if PT_i and GT_i are similar. The notion of similarity adopted neither requires that PT_i and GT_i have exactly the same columns, nor that they have exactly the same rows. This allows for some mismatch between PT_i and GT_i. The similarity between two tables or columns was measured with Jaccard similarity. [3]

The following procedure captures this intuition:

1. Compute GT_i and PT_i over D.

2. For each column of GT_i, compute the most similar column of PT_i, respecting a minimum column similarity threshold of tc. This step induces a partial matching M from columns of GT_i to columns of PT_i.

3. If the fraction of the number of columns of GT_i that match some column of PT_i is below a given threshold tn, P_i is considered *incorrect*.

4. The *adjusted ground truth table* AGT_i is constructed by dropping all columns of GT_i that do not match any column of PT_i, and the *adjusted predicted table* APT_i is constructed by dropping all columns of PT_i that are not matched and permuting the remaining columns so that PC_k is the kth column of APT_i iff GC_k, the kth column of AGT_i, is such that $M(GC_k) = PC_k$.

5. Finally, AGT_i and APT_i are compared. If their similarity is above a given threshold tq, then P_i is *correct*; otherwise P_i is *incorrect*.

[3] <https://docs.snowflake.com/pt/user-guide/querying-approximate-similarity>

5.3 Experimental setup

Each strategy was executed with two models, *GPT-3.5-turbo* and *GPT-4*. However, since the Mondial schema is fairly large, samples of database instances were passed in some experiments, and since DIN-SQL has a large prompt, *GPT-3.5-turbo-16k*, which allows 16k tokens, had to be used in some cases.

In all experiments, LangChain played a pivotal role. Beyond the call to the LLM, it was instrumental in constructing prompts and automating the extraction of metadata from the Mondial schema directly from the database. Additionally, the framework offered valuable insights, including the count of input and output tokens, as well as the overall cost associated with each invocation of the OpenAI API, as shown in Figure 5.1.

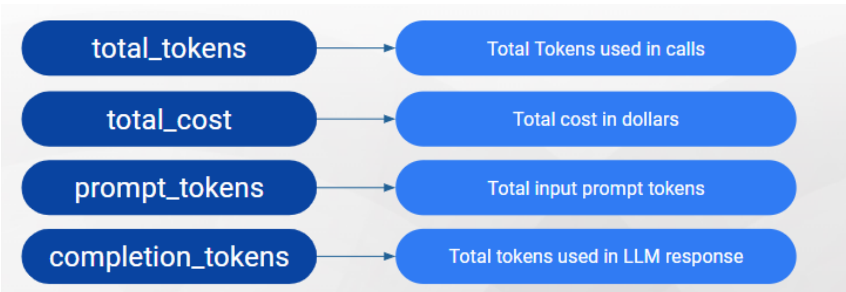

Figure 5.1: Data provided by LangChain about the tokens used.

LangChain-based strategies for Text-to-SQL were divided in two groups: passing the NL question and the schema; passing the NL question, the schema, and some sample rows from each table.

5.4 Results

Table 5.2 displays the results for the Mondial database. Columns "Simple", "Medium", and "Complex" denote the accuracy results for each type of question, while column "Overall" represents the total accuracy across the entire dataset. Additionally, columns "Input Tokens" and "Output Tokens" indicate the number of tokens passed as input and received as output from the model, respectively. Lastly, column "Estimated Cost" provides the estimated cost in US Dollars.

With respect to overall accuracy, the Top-5 strategies used GPT-4, albeit they were also associated with the highest costs. C3 had the best overall accuracy of 0.78. Then, *SQLQueryChain* with samples, DIN, and C3+DIN had the same overall accuracy of 0.70. Lastly, *SQLQueryChain*, without samples, achieved 0.69.

The top-performing strategies for each query type were all based on GPT-4. C3+DIN achieved the highest accuracy for simple queries, reaching 0.91, whereas *SQL Query Chain* with samples outperformed the other strategies for medium-type queries with an accuracy of 0.85. Finally, C3 demonstrated excellent performance for complex queries, attaining an accuracy of 0.71.

Despite the very low cost, the GPT-3.5 model proved to be very poor for generating SQL. The best strategy was *SQLQueryChain* with an overall accuracy of 0.60. The model was unable to correctly generate medium and complex SQL queries, precisely those with many filters, aggregations, ambiguous queries and joins. However, considering only simple queries, the top-5 strategies with the GPT-3.5 model had an accuracy greater than 0.75.

SQLQueryChain was the best among the LangChain-based strategies. It had the best cost-benefit ratio among the Top-5 strategies. This shows the importance of providing the entire schema to achieve higher overall accuracy. Additionally, incorporating samples into the prompt, helping to capture the semantics of the data, improved accuracy. However, the size of the prompt presents a challenge, as it requires passing the entire database schema and other relevant data. This large data volume occasionally lead the LLM to be confused during the SQL conversion for certain queries.

SQLDatabaseSequentialChain had minimal cost due to smaller prompts, obtained by filtering schemas for tables. However, the GPT models incorrectly identified relevant tables for the NL query, leading to incorrect SQL queries, thus exhibiting poor performance in both models. This occurred due to the simplicity of the *decider chain prompt*, whose task is to identify tables relevant to the queries. It lacks detailed descriptions of the tables, such as synonyms, definitions, or other guiding tips for the LLM. While providing samples of the data in the prompt proved beneficial in *SQLQueryChain*, it did not yield positive results in this strategy. However, this error was primarily caused by inaccuracies in the initial step, where incorrect table selection rendered sample provisioning irrelevant for SQL generation.

SQLAgent also had a low cost in both the number of tokens used and the price spent, due to the size of the prompt. Similarly to *SQLDatabaseSequentialChain*, the strategy made mistakes in identifying the tables and, consequently, generated incorrect SQL. *SQLAgent* became lost or hallucinated using GPT-4. In fact, *SQLAgent* is not fully compatible with GPT-4. Also, *SQLAgent* had a poor performance with GPT-3.5-turbo. Another problem found was in the correction mechanism of this chain. Upon generating incorrect SQL and executing it in the database, the error feedback was returned to the LLM to fix the SQL query, but it continued to generate the query incorrectly.

C3 significantly enhances LLM's performance in handling simple and complex queries. Clear Prompting offers effective prompts that help the LLM to better comprehend the query intent and the database schema structure. As it captures only specific tables and columns, it increases the clarity on how joins should be synthesized.

On the other hand, for medium queries, characterized by numerous filters, its efficacy is reduced as it fails to assist LLM in understanding the semantics of the data. Also, Consistent Output generated many output tokens since it produces ten answers in each Clear Prompting stage and twenty results in SQL generation. If the output does not align with the strategy's expectations, the LLM is required to generate the outputs again. This became problematic when translating medium and complex queries into SQL, with many queries taking more than two minutes for the LLM to provide a response. The output token price of GPT models is higher than the input token price, as shown in Table 3.1. Thus, albeit C3 with GPT-4 had the best overall accuracy, it generated 426,937 output tokens and had a high cost of $30.23.

DIN incurred a cost of $44.80, the second-highest among all strategies. *DIN* with GPT-3.5-turbo had the largest number of input tokens, followed closely by *DIN with GPT-4*, both with over 1.4 MM input tokens. Indeed, *DIN* generates large prompts since it passes the complete database schema and uses a few examples to indicate how the LLM should reason and generate SQL code in each stage, except for the self-correction stage. While providing a few examples in the prompts proved effective for helping the LLM correctly answer simple and medium queries, it was not sufficient for complex queries involving many joins. This limitation arises from the static nature of the examples in *DIN*, which involve vocabularies from other databases rather than Mondial.

C3+DIN was the highest among all experiments, totaling $58.19. *C3+Din* with GPT-4 exhibited the same overall accuracy as *DIN*, albeit lower than *C3 with GPT-4*. Whereas the strategy proved effective for simple queries, it inherited the issues of both *C3* and *DIN*, such as prolonged table and column recall times and large prompt sizes. This resulted in the consumption of a substantial number of tokens, both input and output, despite generating fewer input tokens than *DIN*, as it did not utilize all *DIN* modules. Consequently, the combination becomes impractical for real-world applications.

5.5 Analysis of the predicted SQL queries

The LLM-based Text-to-SQL strategies were able to understand aggregations and translate them correctly into SQL as shown below:

```
Question: What is the average infant mortality rate for each continent?
LLM: SELECT continent.name, AVG(population.infant\_mortality) FROM continent JOIN encompasses ON continent.name = encompasses.continent JOIN population ON encompasses.country = population.country GROUP BY continent.name;


Question: What is the area of the largest continent?
LLM: SELECT MAX(area) FROM continent;


Question: What is the total of provinces of Netherlands?
LLM: SELECT COUNT(*) FROM province WHERE country = 'Netherlands';
```

In order to comprehend the areas where the LLM-based Text-to-SQL strategies struggle with SQL query generation, the incorrect answers generated by the best strategy, C3, were manually examined. The aim was to understand the reasons behind their failure. Inspired by Pourreza e Rafiei (2023), these errors were classified into five categories: Schema Linking, JOIN, Nested, Invalid SQL and Miscellaneous.

Figure 5.2 shows the error analysis of C3 with GPT-4. When compared with that of C3 for the Spider benchmark (DONG et al., 2023b), it indicates that the errors resulting from schema linking and joins are exacerbated in the experiments with Mondial, which would be expected, given that the Mondial schema is far more complex than the majority of the datasets in the Spider benchmark.

The most prevalent errors occurred during schema linking and subsequent joins. In the case of joins, while the LLM managed to correctly identify tables, it frequently made mistakes when determining which columns should be used. This was a consistent issue observed across all approaches.

The cause of this problem lies in that, in the Mondial schema, the foreign keys in the tables had the same name as the table they related to, such as *mondial_country*, *mondial_sea*, *mondial_river*, *mondial_city*, *mondial_province*. Listing 4 shows the result for the question "*What are the languages spoken in Poland?*". A join between the *mondial_country* table and the *mondial_language* table was required, linking the *mondial_country code* primary key with the foreign key *country* referencing the table of the same name in the *mondial_language* table, as depicted in line 27. However, the LLM confused the instruction at line 25, generating SQL that simply searched for the

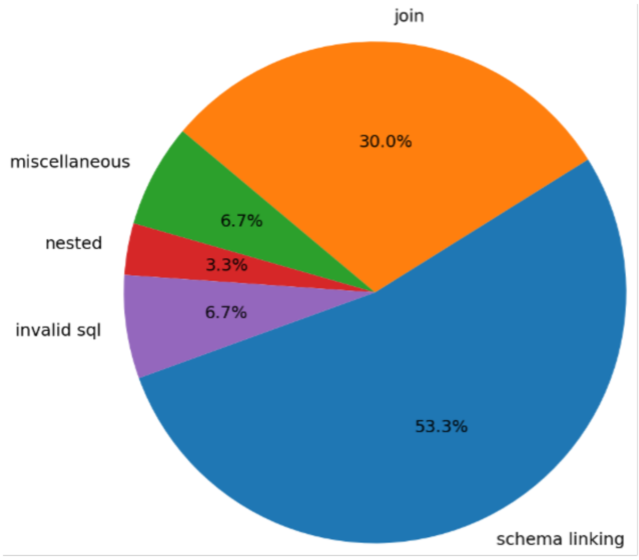

Figure 5.2: Error analysis chart in the C3-GPT4 experiment in the Mondial database.

country using the foreign key *country* instead of executing the join. Therefore, this ambiguity in the schema caused the LLM to get confused and project a column instead of doing the join correctly.

Listing 4: A common error in the SQL generated for LLM in the Mondial database.

```

CREATE TABLE country (
  name VARCHAR(50 CHAR) NOT NULL,
  code VARCHAR(4 CHAR) NOT NULL,
  capital VARCHAR(50 CHAR),
  province VARCHAR(50 CHAR),
  area NUMBER,
  population NUMBER,
  CONSTRAINT countrykey PRIMARY KEY (code),
  CONSTRAINT countryarea CHECK (Area >= 0),
  CONSTRAINT countrypop CHECK (Population >= 0)
)

CREATE TABLE language (
  country VARCHAR(4 CHAR) NOT NULL,
  name VARCHAR(50 CHAR) NOT NULL,
  percentage NUMBER,
  CONSTRAINT languagekey PRIMARY KEY (name, country),
  CONSTRAINT language_country_fk FOREIGN KEY(country) REFERENCES country (code),
  CONSTRAINT languagepercent CHECK ((Percentage > 0) AND (Percentage
      <= 100))
)

Question: What are the languages spoken in Poland?

```

```
LLM: SELECT name FROM mondial_language WHERE country = "Poland"

Expected: SELECT mondial_language.name FROM mondial_language INNER
    JOIN mondial_country ON  mondial_language.country =
    mondial_country.code WHERE mondial_country.name = "Poland"
```

In some cases, samples of instances from the database were passed and LLM was able to capture the semantics of the data in the database, correctly identifying that *country* refers to the country's acronym. Listing 5.5 shows a correctly generated SQL query, although a join was expected:

```

CREATE TABLE language (
  country VARCHAR(4 CHAR) NOT NULL,
  name VARCHAR(50 CHAR) NOT NULL,
  percentage NUMBER,
  CONSTRAINT languagekey PRIMARY KEY (name, country),
  CONSTRAINT language_country_fk FOREIGN KEY(country) REFERENCES country (code),
  CONSTRAINT languagepercent CHECK ((Percentage > 0) AND (Percentage
      <= 100))
)

/*
3 rows from language table:
country  name  percentage
AL  Albanian  98.8
AL  Greek 0.5
GR  Greek 99.0
*/


Question: What are the languages spoken in Poland?
LLM: SELECT name FROM mondial_language WHERE country = "PL";
```

A limited experiment was done by changing the *country* column to *code_country*. As a result, the LLM was able to generate the SQL query correctly. Perhaps the accuracy would be greater if the names of these foreign keys were changed, such as *code_country*, *code_sea*, *code_province* and *code_river*.

Table 5.2: Results for Mondial database

	Simple	Medium	Complex	Overall	Input Tokens	Output Tokens	Tokens	Estimate Cost	Comments
SQL Database Agent (LangChain) - with samples									
GPT-3.5-turbo	0.58	0.37	0.12	0.35	441785	17939	459724	$0.70	(2)
GPT-4	—	—	—	—	—	—	—	—	(4)
DIN-SQL									
GPT-3.5-turbo	0.82	0.45	0.41	0.56	1431645	33050	1464695	$4.43	(1)(5)
GPT-4	0.85	0.76	0.50	0.70	1428284	32473	1460757	$44.80	-
C3									
GPT-3.5-turbo	0.79	0.48	0.38	0.55	175298	754619	929917	$1.77	(6)
GPT-4	0.82	0.51	0.71	0.78	153705	426937	580642	$30.23	(6)
C3+DIN									
GPT-3.5-turbo	0.82	0.51	0.44	0.59	1147061	719891	1866952	$6.32	(1)(5)(6)
GPT-4	0.91	0.70	0.5	0.70	1137152	401191	1538343	$58.19	(5)(6)

(1) GPT-3.5-turbo-16k was used due to its larger token limit.
(2) Two samples were passed in the prompt.
(3) The complete schema could be passed in the prompt.
(4) GPT-4 is not entirely compatible with SQLAgent.
(5) The DIN prompt requires a larger token limit.
(6) Table and column recall took a long time.

6
Experiments to evaluate the effect of using views

The previous chapter indicated that the performance of LLM-based Text-to-SQL tools is significantly less than that reported for well-known benchmark when applied to complex database. This chapter shows that the Text-to-SQL task can be significantly facilitated by providing a database specification based on the use of LLM-friendly views that are close to the language of the users' questions and that eliminate frequently used joins, and LLM-friendly data descriptions of the database values, applying the *SQLQueryChain* strategy to a complex, real-world database.

6.1
A new approach: Views

The results of the experiments in Chapter 5 indicated that an LLMs makes mistakes in schema linking and joins because:

- The relational schema is often an inappropriate specification of the database from the point of view of the LLM – the table and column names are often different from the terms the users adopt to formulate their NL questions;
- The database schema is often large, in the number of tables, columns per table, and foreign keys – but a large schema may not fit in the prompt area, and opens space to queries with many joins, which are difficult to synthesize;
- The data semantics is often complex; for example, some data values may encode enumerated domains – again, the terms the users adopt to formulate their NL questions may have to be mapped to this internal semantics;

Consider, for example, the NL sentence S: "*What is the installation with the largest number of open maintenance orders?*". Sentence S uses the end user's terms "*installation*", "*open*", and "*maintenance order*" which, in the best scenario, would match the database table names "`Installation`" and "`Maintenance_Order`", and the column name "`Situation`", which has "`open`" as a value. However, in a real-world scenario, the relational schema may induce a quite different vocabulary, such as the table names "`TB_IN`" and "`TB_MO`", and the column name "`MO_ST`", and the database may use "`1`" as a value of "`MO_ST`" to indicate that the order is *open*.

An LLM trained for the Text-to-SQL task would translate the expression "*the largest number*" to the correct SQL constructs, which is not a trivial feat (compare it to the treatment of aggregations in earlier approaches reported in (AFFOLTER; STOCKINGER; BERNSTEIN, 2019)). It would then produce the correct SQL query under the best scenario but it would fail in the real-world scenario due to the use of database terms which are thoroughly inappropriate to the LLM. Thus, for the LLM to succeed in the Text-to-SQL task, it should, first of all, be able to match the user and the database vocabularies.

This dissertation then argues that the Text-to-SQL task can be greatly facilitated by a database specification that provides:

– *LLM-friendly views* that map (fragments of) the database schema to terms close to the terms users frequently adopt and that try to pre-define fre- quently used joins.
– *LLM-friendly descriptions* of the database values.

LLM-friendly views are nothing but the familiar concept of views, designed to present (fragments of) the relational schema (that is, database metadata) to the LLM. Views represent a subset of the data contained in a table. It can join and simplify multiple tables into a single virtual table, hiding the complexity of data (GROFF; WEINBERG, 1999). As such, they can be implemented with the usual DBMS mechanisms, within the database. *LLM-friendly descriptions* refer to a set of constructs that try to capture the data semantics. The experiments in this chapter include data samples in the LLM prompt through Langchain. Alternatively, RAG can be used to capture the data semantics, or the LLM can be fine-tuned using a set of *prompt completion pairs* such as (in OpenAI GPT syntax[1]):

```
{"prompt": "the order is open", "completion": "Situation='open'"}
```

Figure 6.1 summarizes the approach adopted in this chapter. The view definitions will create a vocabulary that better matches user terms, and they will also pre-define frequently required joins. The first step aims at reducing schema linking errors, while the second focuses on join errors, thereby reducing the complexity of generating the SQL query.

Lastly, the experiments in this chapter will use a complex real-world database, discussed in Section 6.2.

6.2
A real-world benchmark

[1]<https://platform.openai.com/docs/guides/fine-tuning>

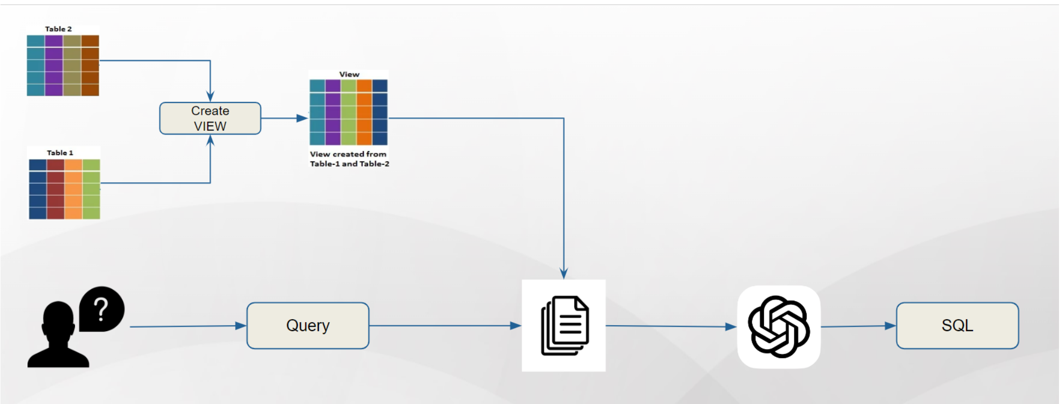

Figure 6.1: Architecture using views for the Text-to-SQL task.

6.2.1
The real-world relational database

The benchmark adopts a real-world relational database (in Oracle) that stores data related to the integrity management of Petrobras' industrial assets.[2]

The relational schema contains 27 relational tables with, in total, 585 columns and 30 foreign keys (some multi-column), where the largest table has 81 columns.

Table and column names in the relational schema do not follow a specific vocabulary. They are assigned using mnemonic terms based on an internal company specification for naming database objects. This scenario implies that users who do not know the relational schema have difficulties in understanding the semantics of the stored data and must turn to database specialists when retrieving data related to maintenance and integrity management processes, even if the user has access to a description of the tables and their columns.

Also, some column values are not end-user-friendly – coding values and combinations of different values may hide semantic information. To overcome this situation, database experts often create SQL functions that contain the logic to represent the semantics hidden in the column values.

6.2.2
The sets of views

To test how the proposed approach affects the Text-to-SQL task, the benchmark introduces three sets of LLM-friendly views of increasing complexity:

Conceptual schema views: a set of views that define a one-to-one mapping of the relational schema to end users' terms; the views basically rename tables and columns. Figure 6.2 shows the referential dependencies diagram of a much-

[2] <https://petrobras.com.br>

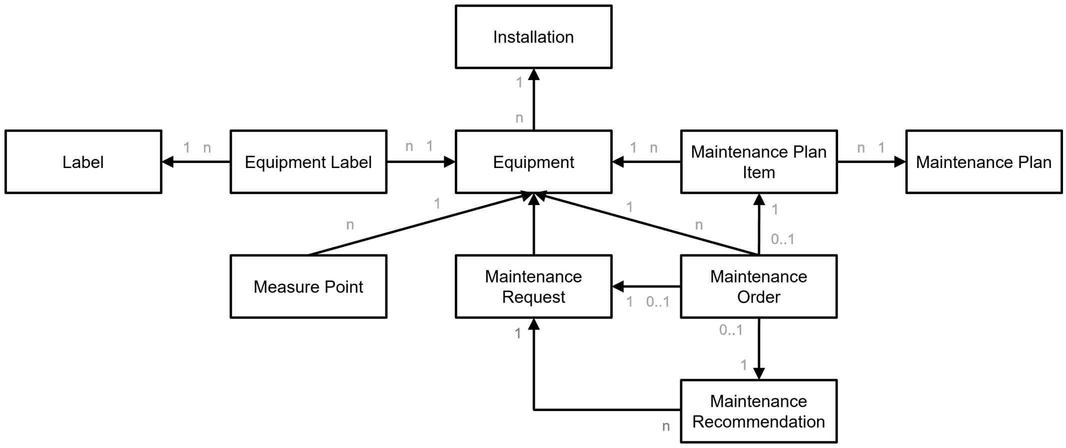

Figure 6.2: The referential dependency diagram of a simplified version.

simplified version of the conceptual schema views, where an arrow represents a foreign key and points to the referenced table, as usual.

Partially extended views: a set of views that extend the conceptual schema views with new columns that predefine joins that follow foreign keys, as well as other selected columns. Figure 6.3 shows that previously it was necessary to have at least three joins between the *Installation* and *Maintenance_Order* tables. Creating an *installation_id* column in *Maintenance_Order*, reduced it to just one join between the tables.

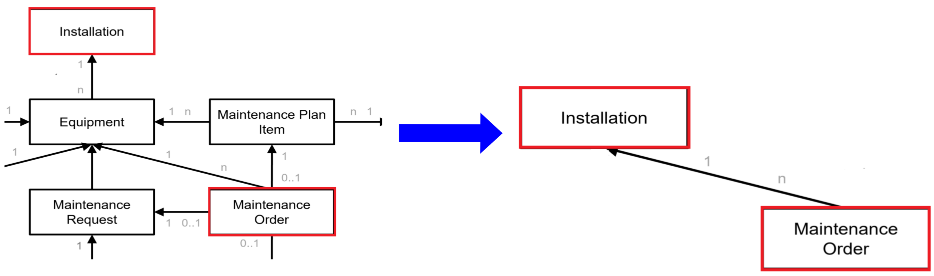

Figure 6.3: An example of how *partially extended views* were constructed.

In practice, the set of *partially extended views* was defined by including the non-primary key columns of the view `Installation` into the views `Equipment`, `Maintenance_Order`, `Maintenance_Request`, `Maintenance_Recommendation`, and `Maintenance_Plan_Item`, respectively.

Fully extended views: a set of views such that each view is a single view formed from the combination of one or more views created from the conceptual schema. This way, it aims to eliminate the generation of SQLs with joins by LLM.

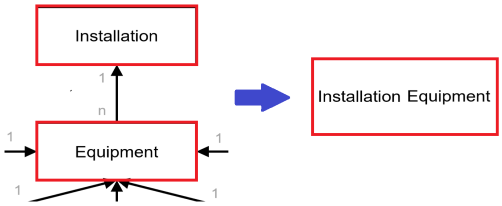

Figure 6.4: An example of how *fully extended views* were constructed.

The following statement shows an example SQL code to create a fully extended view by combining the views `Installation` and `Equipment` in a new and single view named `Installation_Equipment`[3] as illustrated in Figure 6.4:

```
CREATE VIEW pe_equipment AS
SELECT inst.name AS installation_name, inst.asset,
       inst.main_hub, inst.business_unit, equip.*
FROM equipment equip JOIN installation inst
     ON inst.id = equip.installation_id
```

where it has all the columns of `Equipment` and `Installation` views.

6.2.3 The test questions and their ground truth SQL translations

The benchmark contains a set of 100 NL questions, $L = \{L_1, ..., L_{100}\}$, that consider the terms and questions experts use when requesting information related to the maintenance and integrity processes.

The ground truth SQL queries, $G = \{G_1, ..., G_{100}\}$, were manually defined over the conceptual schema views so that the execution of G_i returns the expected answer to the NL question L_i. The use of the conceptual schema views facilitated this manual task, since these views use a vocabulary close to that of the NL questions.

An NL question L_i is classified into *simple*, *medium*, and *complex*, based on the complexity of its ground truth SQL query G_i, as in the Spider benchmark (extra-hard questions were not considered). The set of questions L contains 33 simple, 33 medium and 34 complex questions.

Note that the NL questions classification is anchored on the conceptual schema views. But, since these views map one-to-one to the tables of the relational schema, a classification anchored on the relational schema would remain the same. The classification is maintained for the other sets of views, even knowing that the definition of these other sets of views might simplify the

[3]Oracle names are case insensitive by default

translation of some NL questions (which was one of the reasons for considering these sets of views, in any case).

Table 6.1 shows examples of NL questions and their ground truth SQL translations. Column "**ID**" indicates the question identifiers; column "**NL Question**" indicates the test NL questions suggested by experts; column "**Ground Truth SQL Query Table**" indicates the ground truth SQL query that represents the answer table for the NL question; and column "**Question Type**" represents the question type classification (as explained above).

Table 6.1: A sample of the designed benchmark dataset.

ID	NL Question	Ground Truth SQL Query Table	Question Type
1	Which IBX15 installation recommendations are expired?	`SELECT id FROM vw_maintenance_recommentation WHERE expired ='true' AND installation_name = 'IBX-15'`	medium
		…	
25	Show all orders with type MO05	`SELECT id FROM vw_maintenance_order WHERE type = 'MO05'`	simple
		…	
47	Which IBX-67 installation label has the most approved maintenance requests with a priority greater than 6?	`SELECT el.label_id FROM vw_maintenance_request mr JOIN vw_equipment_label el ON mr.equipment_id = el.equipment_id WHERE mr.installation_name = 'IBX-67' AND mr.situation = 'Approved' AND mr.priority > 6 GROUP BY el.label_id ORDER BY COUNT(mr.id) DESC FETCH FIRST 1 ROWS ONLY`	complex
		…	

6.3 Experimental setup

The experiments used a Text-to-SQL implementation based on LangChain's SQLQueryChain (see Section 4.1). The results in Table 5.2 show that this strategy had good performance and a much lower cost than other tested strategies.

This chain greatly simplifies creating prompts to access databases through views since it passes a view specification as if it were a table specification. Figure 6.5 illustrates the prompt implemented: (A) contains instructions for the LLM; (B) defines the output format; (C) partly illustrates how the `maintenance_order` view is passed to the LLM as a `CREATE TABLE` statement; (D) shows 3 data samples from the `maintenance_order` view; and (E) passes the NL question.

The experiments applied the LangChain-based strategy with GPT-3.5-turbo-16k and GPT-4 against the 100 questions introduced in Section 6.2, separately for the database relational schema of Section 6.2.1, and each of the three sets of views outlined in Section 6.2.2, all with data samples. Finally, the

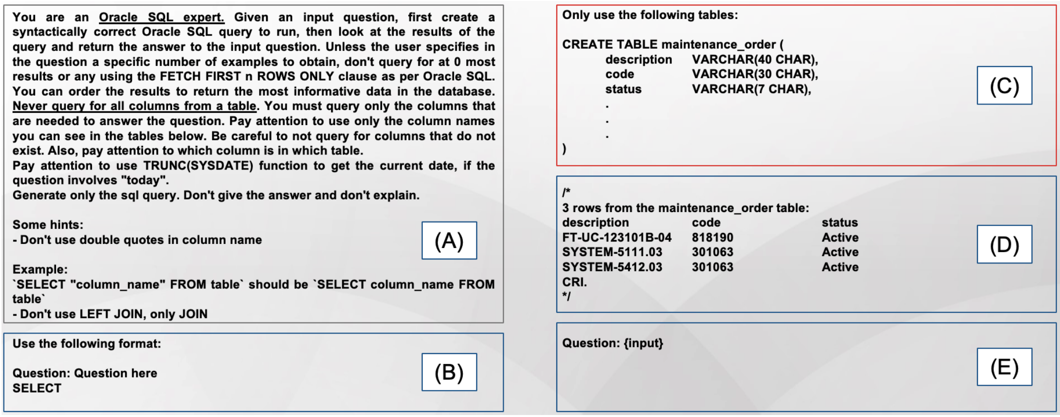

Figure 6.5: SQLQueryChain's prompt with some tips used in the experiments.

metrics and evaluation procedure were the same as those described in Section 5.2.

6.4 Results

Table 6.2 shows the results. Overall, the accuracy results with GPT-4 were much better than those with GPT-3.5-turbo-16k; if we compare the best accuracy results (the gray cells), GPT-4 achieved an overall accuracy 22% better than GPT-3.5-turbo-16k. Let us concentrate on the accuracy results with GPT-4.

The results demonstrated that running LLM for Text-to-SQL tasks on the relational schema of a real-world database resulted in the lowest performance, with an accuracy of 41%. An experiment was conducted on an extended version of the relational schema, introducing new foreign keys to minimize table joins. This adjustment alone led to a 13% increase in accuracy. However, the accuracy remained notably lower compared to benchmarks such as Mondial and Spider, primarily due to persistent challenges with schema linking and joins.

Implementing the strategy with LLM-friendly views yielded significant improvements. A comparison between the results of experiments on the relational schema and *Conceptual Schema Views* indicates that the overall accuracy achieved with the view-based strategy was 24% higher than that achieved with the relational schema alone. This means that simply renaming the tables and columns to terms closer to the end-user vocabulary sufficed to improve accuracy substantially.

The *Partially Extended Views* experiment achieved an accuracy of 74%, representing the highest accuracy among all employed strategies. This approach simplified the Text-to-SQL task by renaming columns to more descrip-

Table 6.2: Results for Views

	Simple	Medium	Complex	Overall	Input Tokens	Output Tokens	Total Tokens	Estimate Cost	Comments
Partially Extended - Conceptual Schema - Views									
GPT-3.5-turbo	0.79	0.54	0.23	0.52	810903	5307	816210	$2.45	(1)
GPT-4	0.91	0.76	0.56	0.74	810903	10259	821162	$24.94	-
Fully Extended Conceptual Schema Views									
GPT-3.5-turbo	0.85	0.40	0.18	0.47	209601	6616	216217	$0.33	(2)
GPT-4	0.85	0.61	0.53	0.66	206178	7388	213566	$6.63	(2)
Conceptual Schema Views									
GPT-3.5-turbo	0.82	0.49	0.15	0.48	750503	7849	758352	$2.28	(1)
GPT-4	0.97	0.55	0.44	0.65	750503	14345	764848	$23.38	-
Partially Extended Relational Schema									
GPT-3.5-turbo	0.73	0.24	0.09	0.35	559204	4602	563806	$1.70	(1)
GPT-4	0.82	0.42	0.39	0.54	559204	11746	570950	$17.48	-
Relational Schema									
GPT-3.5-turbo	0.73	0.24	0.12	0.36	1011903	5377	1017280	$3.06	(1)(3)
GPT-4	0.67	0.33	0.23	0.41	1011903	11405	1023308	$31.04	(3)

(1) GPT-3.5-turbo-16k was used due to its larger token limit.
(2) Only a single view that was related to the question was passed in the prompt
(3) The relational schema does not have foreign keys defined

tive names, facilitating for the LLM the identification of columns and tables during SQL generation. Additionally, it created new columns simulating foreign keys to reduce the need for joins. The overall accuracy achieved with these views was substantially better (33% improvement) than that achieved with the relational schema. Moreover, it demonstrated greater accuracy in medium and complex queries, reaching 76% and 56%, respectively.

Also, note that the *Partially Extended Views* experiment failed to translate two more simple NL questions than *Conceptual Schema Views* experiment. One explanation is that LLMs are non-deterministic; if the experiments were repeated several times, Table 6.2 could report slightly different accuracy results for *Conceptual Schema Views* and *Partially extended views* experiments.

A comparison between the results of the *Partially Extended Views* and *Fully Extended Views* experiments shows a decrease of 8%. Indeed, the fully extended views save more joins, facilitating the Text-to-SQL task, but they require passing much larger view specifications in the prompt. Furthermore, the definition of a fully extended view, which combines several views, requires renaming several columns, which may create columns with similar names. In conjunction – views with many columns and similar column names – confuse the LLM, leading to ambiguous matches with an NL question.

Another consideration is that, despite the *Fully Extended Views* achieving the second-best results, with 66%, the views related to each question were already provided to the LLM. In other words, the model did not have to choose which views to select; they were explicitly given. Also, this is reflected in the approach having the lowest cost of all, since in other approaches, all available views or tables were described in the prompt.

In summary, the results suggest that the partially extended views, with just a few extra columns that pre-define joins, is a better alternative than fully extended views, that combine several views. These views also proved to be a much better alternative than using the relational schema or the set of conceptual schema views. From a broader perspective, the accuracy increases when one moves from prompting the LLM with the relational schema to prompting the LLM with LLM-friendly views and data samples, corroborating the position argued in this work.

6.5
Analysis of the predicted SQL queries in LLM-Friendly views

As the results in Section 6.4 demonstrate, the issue of schema linking and joins has been mitigated. Consider the NL sentence S: "*What is the installation with the largest number of open maintenance orders?*". The tables involved are *Installation* and *Maintenance_Order*. At least two joins are required as we can see in Figure 6.2: *Installation* -> *Equipment* and, *Equipment* -> *Maintenance_Order*. Extending to views, the *Installation_ID* column was created in the *Maintenance_Order* view as a foreign key referencing the *Installation* view, reducing it to just a join. Renaming the *Situation* column to *Open_situation* in the *Maintenance_Order* view facilitated the identification of the meaning of "open maintenance orders". By providing samples of the instances in the prompt, the LLM was able to identify that the value of this column is "open". As result, the language model successfully generated the SQL query, as shown below:

```
Question: What is the installation with the largest number of open
    maintenance orders?

LLM: SELECT i.Installation_Name
FROM Installation i JOIN Maintenance_Order mo ON i.ID = mo.Installation_ID
WHERE mo.Open_situation = 'open'
GROUP BY i.Installation_Name
ORDER BY COUNT(*) DESC
FETCH FIRST 1 ROWS ONLY
```

In Table 5.1, we have the question "*Which IBX15 installation recommendations are expired?*", while the gold SQL searches for *installation_name = "IBX-15"*, the user's question references "IBX15" without the hyphen. Consequently, when the LLM searches for *installation_name = "IBX15"*, it returns an empty set as there is no matching value in the database.

In another question, "*What are the requests located in MX10 ?*", SQL gold is expected to use the LIKE operator and thus *SELECT * FROM maintenance_request WHERE localization LIKE "%MX10%"*, but LLM generates *SELECT * FROM maintenance_request WHERE localization ="MX10"*, and again, returns an empty set.

Assembling the filters in the SQL query posed the greatest challenge for the LLM. Despite having samples of some instances in the prompt, it is not possible to pass all possible values that a column can have, partly due to the limitation of the size of the context window. Consequently, in cases where the LLM did not have sufficient information about the data, it made mistakes in constructing the generated SQL filter.

7
Conclusions and future work

7.1
Conclusions

This dissertation described experiments with various strategies using LLMs for the text-to-SQL task in two challenging scenarios characterized by: (1) an openly available database with a large complex schema, using terms close to those of the users' NL questions; (2) a real-world database with a large complex schema, using terms different from those of the users' NL questions.

As a first contribution of this dissertation, the results of the experiments in Chapter 5 permitted concluding that, while some of these strategies showed promising results in benchmarks such as Spider, their performance decreased when applied to a complex database. Among the strategies tested, SQLQueryChain with samples using GPT-4 proved more effective, because C3 with GPT-4, despite having the best overall accuracy, incurred higher costs, longer runtime, and more output tokens. In general, passing the entire schema to the LLM achieved better results than filtering the schema and passing just a few tables, but this approach is limited by the number of tokens the LLM allows, especially for complex schemas, such as that of Mondial.

As a second contribution of this dissertation, the results of the experiments in Chapter 6 revealed that the overall accuracy of a text-to-SQL strategy (SQLQueryChain with samples using GPT-4) improved by customizing the database specification. On a real-world benchmark, the experiments suggested that there is a dramatic increase in accuracy when one moves from prompting the LLM with the relational schema to prompting the LLM with LLM-friendly views and data samples. These views help reduce SQL query complexity, thereby improving the overall accuracy of the Text-to-SQL task. They rename tables and columns to more descriptive terms aimed at facilitating the identification of columns and tables by the LLM during SQL generation, and introduce new columns that reduce the number of joins.

Partial results related to this dissertation, as well other relevant results, were reported in the following articles:

– Pinheiro, J.; Victorio, W.; Nascimento, E. R.; Seabra, A.; Izquierdo, Y.; García, G.; Coelho, G.; Lemos, M.; Leme, L.; Furtado, A. and Casanova, M. (2023).**On the Construction of Database Interfaces Based on Large Language Models**. In Proceedings of the 19th International

Conference on Web Information Systems and Technologies - WEBIST; ISBN 978-989-758-672-9; ISSN 2184-3252, SciTePress, pages 373-380. DOI: 10.5220/0012204000003584.

- Nascimento, E.R., Garcia, G.M., Victorio, W.Z., Lemos, M., Izquierdo, Y.T., Garcia, R.L., Leme, L.A.P., Casanova, M.A.: **A family of natural language interfaces for databases based on chatgpt and langchain**. In: Proceedings of the 42nd International Conference on Conceptual Modeling – Posters&Demos. Lisbon, Portugal (nov 2023).
- Nascimento, E.; García, G.; Feijó, L.; Victorio, W.; Izquierdo, Y.; R. de Oliveira, A.; Coelho, G.; Lemos, M.; Garcia, R.; Leme, L. and Casanova, M. A. (2024). (2024). **Text-to-sql meets the real-world**. In: *Proc. 26th Int. Conf. on Enterprise Info. Sys.*
- Nascimento, E. R., Izquierdo, Y. T., Garcia, G. M., Coelho, G., Feijó, L., Lemos, M., Leme, L. A. P., and Casanova, M. A. (2024). **My Database User is a Large Language Model**. In *Proc. 26th Int. Conf. on Enterprise Info. Sys.*
- Coelho, G., Nascimento, E. R., Izquierdo, Y. T., Garcia, G. M., Feijó, L., Lemos, M., Garcia, R. L., R. de Oliveira, A., Pinheiro, J., and Casanova, M. A. (2024). **Improving the Accuracy of Text-to-SQL Tools based on Large Language Models for Real-World Relational Databases**. (Submitted for publication).

7.2 Future Work

Future work will consider four alternatives for text-to-SQL, which are not mutually exclusive.

The first alternative is to improve some of the text-to-SQL strategies tested. This can be achieved by implementing self-correction and self-consistency mechanisms in LangChain-based strategies, offering more comprehensive information about the database tables in the *decider_chain* prompt of *SQLDatabaseSequentialChain*, refining self-consistency in C3, and incorporating a few examples of NL question/SQL query pairs in clear prompting.

The LLM-friendly views used in the experiments were created by inspecting the database documentation and by mining a log of user questions. Albeit this process was tedious but not too difficult, a second suggestion is to develop on a tool that automatically creates views on the fly, depending on the NL question submitted.

The third alternative is to explore the use of RAG (discussed in Section 3.2.3.5). While DIN relies on static examples and *SQLQueryChain* provides data samples, both strategies demonstrated that having access to external information helps LLMs generate SQL queries. RAG further assists LLMs by providing access to external data, enabling them to generate responses with additional context.

In this approach, a dataset comprising NL question/SQL query pairs would be encoded and stored in a vector database. Subsequently, the input NL question would also be encoded, and based on this encoding, the pairs whose NL question is most similar to the input NL question would be retrieved from the dataset. Leveraging the context formed by these pairs along with the input NL question, the LLM would be prompted to generate the SQL query. This approach would help the LLM identify the tables and columns involved in the input NL question, as well as understand the semantics of the data.

The last alternative is to fine-tune an open-source LLM, stored locally, for a given database with a large schema. The training dataset can be quite laborious to create, but GPT-4 may come in hand to augment the training set from a seed set of NL questions and their SQL translations. Running locally an LLM, rather than as a service provided by a third party, has the advantage that proprietary data and metadata will remain in-house when running the LLM, a concern often voiced by companies.

8
Bibliography

ABDULLAH, M.; MADAIN, A.; JARARWEH, Y. Chatgpt: Fundamentals, applications and social impacts. In: **2022 Ninth International Conference on Social Networks Analysis, Management and Security (SNAMS'22)**. [S.l.: s.n.], 2022. p. 1–8.

AFFOLTER, K.; STOCKINGER, K.; BERNSTEIN, A. A comparative survey of recent natural language interfaces for databases. **The VLDB Journal**, Springer Science and Business Media LLC, v. 28, n. 5, p. 793–819, ago. 2019. ISSN 0949-877X. Available at: <http://dx.doi.org/10.1007/s00778-019-00567-8>.

ANDERSON, C. **Guardrails on Large Language Models, Part 3: Prompt Design**. 2023. Available at: <https://avidml.org/blog/llm-guardrails-3/>. Accessed on February 14, 2024.

BROWN, T. B. et al. **Language Models are Few-Shot Learners**. 2020. Available at: <https://arxiv.org/abs/2005.14165>.

DAHL, D. A. et al. Expanding the scope of the ATIS task: The ATIS-3 corpus. In: **Human Language Technology: Proceedings of a Workshop held at Plainsboro, New Jersey, March 8-11, 1994**. [S.l.: s.n.], 1994. Available at: <https://aclanthology.org/H94-1010>.

DAS, S. **Fine Tune Large Language Model (LLM) on a Custom Dataset with QLoRA**. 2024. Available at: <https://dassum.medium.com/fine-tune-large-language-model-llm-on-a-custom-dataset-with-qlora-fb60abdeba07>.

Defog AI. **sqlcoder**. 2023. <https://github.com/defog-ai/sqlcoder>.

DESAI, G. **Parameters for LLM Models: A Simple Explanation**. 2024. Linkedin. Available at: <https://www.linkedin.com/pulse/parameters-llm-models-simple-explanation-gaurang-desai-kabfe/>. Accessed on February 15, 2024.

DETTMERS, T. et al. **QLoRA: Efficient Finetuning of Quantized LLMs**. 2023. Available at: <https://arxiv.org/abs/2305.14314>.

DONG, Q. et al. **A Survey on In-context Learning**. 2023. Available at: <https://arxiv.org/abs/2301.00234>.

DONG, X. et al. **C3 Zero-shot Text-to-SQL with ChatGPT**. 2023. Available at: <https://arxiv.org/abs/2307.07306>.

Eosphoros AI. **DB-GPT-Hub**. 2023. <https://github.com/eosphoros-ai/DB-GPT-Hub>.

Eosphoros AI. **Awesome-Text2SQL**. 2024. <https://github.com/eosphoros-ai/Awesome-Text2SQL>.

GAN, Y. et al. Towards robustness of text-to-sql models against synonym substitution. **CoRR**, abs/2106.01065, 2021. Available at: <https://arxiv.org/abs/2106.01065>.

GAN, Y.; CHEN, X.; PURVER, M. Exploring underexplored limitations of cross-domain text-to-sql generalization. In: **Conference on Empirical Methods in Natural Language Processing**. [S.l.: s.n.], 2021. Available at: <https://api.semanticscholar.org/CorpusID:237491444>.

GAO, D. et al. **Text-to-SQL Empowered by Large Language Models A Benchmark Evaluation**. 2023. Available at: <https://arxiv.org/abs/2308.15363>.

GAO, Y. et al. **Retrieval-Augmented Generation for Large Language Models: A Survey**. 2024. Available at: <https://arxiv.org/abs/2312.10997>.

GOOGLE. **Official blog: AI**. 2024. Available at: <https://blog.google/technology/ai/>.

GROFF, J. R.; WEINBERG, P. N. **SQL: The Complete Reference**. [S.l.]: Osborne/McGraw-Hill, 1999. 291–292 p. ISBN 0072118458.

GUO, C. et al. **Prompting GPT-3.5 for Text-to-SQL with De-semanticization and Skeleton Retrieval**. 2023. Available at: <https://arxiv.org/abs/2304.13301>.

GUO, C. et al. **Retrieval-augmented GPT-3.5-based Text-to-SQL Framework with Sample-aware Prompting and Dynamic Revision Chain**. 2023. Available at: <https://arxiv.org/abs/2307.05074>.

GUO, J. et al. Chase: A large-scale and pragmatic chinese dataset for cross-database context-dependent text-to-sql. In: **Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing**. [S.l.]: Association for Computational Linguistics, 2021. p. 2316–2331. Available at: <https://aclanthology.org/2021.acl-long.180>.

HABIB, R.; OZDEMIR, S. **Prompt Engineering 101**. 2022. Available at: <https://humanloop.com/blog/prompt-engineering-101>. Accessed on February 14, 2024.

HENDRIX, G. G. Natural-language interface. **American Journal of Computational Linguistics**, v. 8, n. 2, p. 56–61, 1982. Available at: <https://aclanthology.org/J82-2002>.

HU, E. J. et al. **LoRA: Low-Rank Adaptation of Large Language Models**. 2021. Available at: <https://arxiv.org/abs/2106.09685>.

HU, Z. et al. **LLM-Adapters: An Adapter Family for Parameter-Efficient Fine-Tuning of Large Language Models**. 2023. Available at: <https://arxiv.org/abs/2304.01933>.

IYER, S. et al. **Learning a Neural Semantic Parser from User Feedback**. 2017. Available at: <https://arxiv.org/abs/1704.08760>.

KAUFMANN, E.; BERNSTEIN, A. How useful are natural language interfaces to the semantic web for casual end-users? In: **The Semantic Web**. [S.l.]: Springer, 2007. p. 281–294.

KOJIMA, T. et al. **Large Language Models are Zero-Shot Reasoners**. 2023.

LAN, W. et al. **UNITE: A Unified Benchmark for Text-to-SQL Evaluation**. 2023. Available at: <https://arxiv.org/abs/2305.16265>.

LANGCHAIN. **LangChain is a framework for developing applications powered by language models**. 2024. Available at: <https://python.langchain.com/docs/get_started/introduction>.

LEWIS, P. et al. **Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks**. 2021. Available at: <https://arxiv.org/abs/2005.11401>.

LI, J. et al. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. 2023. Available at: <https://arxiv.org/abs/2305.03111>.

LI, Z.; YANG, Z.; WANG, M. **Reinforcement Learning with Human Feedback: Learning Dynamic Choices via Pessimism**. 2023. Available at: <https://arxiv.org/abs/2305.18438>.

LIU, A. et al. **A comprehensive evaluation of ChatGPT's zero-shot Text-to-SQL capability**. 2023. Available at: <https://arxiv.org/abs/2303.13547>.

LOCK, S. What is ai chatbot phenomenon chatgpt and could it replace humans? **The Guardian**, 2022. Accessed on February 27, 2024.

LU, H. et al. **Utilizing Large Language Models for Natural Interface to Pharmacology Databases**. 2023. Available at: <https://arxiv.org/abs/2307.15717>.

MEHRA, A. **Fine-Tuning Tutorial: Falcon-7b LLM To A General Purpose Chatbot**. 2023.

META. **Getting started with Llama**. 2024. Available at: <https://llama.meta.com>.

NASCIMENTO, E. et al. A family of natural language interfaces for databases based on chatgpt and langchain. In: **ER2023: Companion Proceedings of the 42nd International Conference on Conceptual Modeling: ER Forum, 7th SCME, Project Exhibitions, Posters and Demos, and Doctoral Consortium**. Lisbon, Portugal: [s.n.], 2023. Available at: <https://ceur-ws.org/Vol-3618/pd_paper_1.pdf>.

NIHALANI, N.; SILAKARI, S.; MOTWANI, M. Natural language interface for database: A brief review. In: . [S.l.: s.n.], 2011. Available at: <https://api.semanticscholar.org/CorpusID:18731245>.

OPENAI. Gpt-4 technical report. **arXiv**, 2023. Doi: 10.48550/arXiv.2303.08774.

OPENAI. **OpenAI Documentation**. 2023. Available at: <https://platform.openai.com/docs>.

OPENAI. **OpenAI Blog**. 2024. Available at: <https://openai.com/blog/new-embedding-models-and-api-updates>.

OUYANG, L. et al. **Training language models to follow instructions with human feedback**. 2022. Available at: <https://arxiv.org/abs/2203.02155>.

PERRIGO, B. Exclusive: Openai used kenyan workers on less than $2 per hour to make chatgpt less toxic. **The Time**, 2023. Available at: <https://time.com/6247678/openai-chatgpt-kenya-workers>. Accessed on February 27, 2024.

PING, W. J. **Open-sourcing SQLEval: our framework for evaluating LLM-generated SQL**. 2023. Disponível em: <https://defog.ai/blog/open-sourcing-sqleval/>.

PINHEIRO., J. et al. On the construction of database interfaces based on large language models. In: INSTICC. **Proceedings of the 19th International Conference on Web Information Systems and Technologies - WEBIST**. [S.l.]: SciTePress, 2023. p. 373–380. ISBN 978-989-758-672-9. ISSN 2184-3252.

POURREZA, M.; RAFIEI, D. **DIN-SQL: Decomposed In-Context Learning of Text-to-SQL with Self-Correction**. 2023. Available at: <https://arxiv.org/abs/2304.11015>.

QUAMAR, A. et al. Natural language interfaces to data. **Foundations and Trends in Databases**, v. 11, n. 4, p. 319–414, 2022.

RIEDEL, S. et al. **Retrieval Augmented Generation: Streamlining the creation of intelligent natural language processing models**. 2020. Available at: <https://ai.meta.com/blog/retrieval-augmented-generation-streamlining/the-creation-of-intelligent-natural-language-processing-models/>. Accessed on February 26, 2024.

SAFJAN, K. Understanding retrieval-augmented generation (rag) empowering llms. **Krystian's Safjan Blog**, 2023.

SANDHU, J. A. **What are LLMs and Generative AI? A Beginner's Guide to the Technology Turning Heads**. 2024. Available at: <https://srinstitute.utoronto.ca/news/gen-ai-llms-explainer>. Accessed on February 27, 2024.

SARAVIA, E. Prompt Engineering Guide. **https://github.com/dair-ai/Prompt-Engineering-Guide**, 12 2022.

SINGH, A. **Large Language Models: A Guide on Its Benefits, Use Cases, and Types**. 2023. Available at: <https://yellow.ai/blog/large-language-models>. Accessed on February 17, 2024.

TAM, A. **What Are Zero-Shot Prompting and Few-Shot Prompting**. 2023. Disponível em: <https://machinelearningmastery.com/what-are-zero-shot-prompting-and-few-shot-prompting/>.

VASWANI, A.; AL. et. Attention is all you need. In: **Advances in Neural Information Processing Systems**. [S.l.: s.n.], 2017. v. 30.

WANG, X. et al. **Self-Consistency Improves Chain of Thought Reasoning in Language Models**. 2023. Available at: <https://arxiv.org/abs/2203.11171>.

WEI, J. et al. **Chain-of-Thought Prompting Elicits Reasoning in Large Language Models**. 2023. Available at: <https://arxiv.org/abs/2201.11903>.

WOLFE, C. R. **Chain of Thought Prompting for LLMs**. 2023. Available at: <https://cameronrwolfe.substack.com/p/chain-of-thought-prompting-for-llms>. Accessed on February 28, 2024.

XU, X.; LIU, C.; SONG, D. Sqlnet: Generating structured queries from natural language without reinforcement learning. **arXiv preprint arXiv:1711.04436**, 2017.

YAGHMAZADEH, N. et al. Sqlizer: query synthesis from natural language. **Proceedings of the ACM on Programming Languages**, v. 1, n. OOPSLA, p. 1–26, 2017.

YAO, S. et al. **ReAct: Synergizing Reasoning and Acting in Language Models**. 2023. Available at: <https://arxiv.org/abs/2210.03629>.

YU, T. et al. Typesql: Knowledge-based type-aware neural text-to-sql generation. In: **Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)**. [S.l.: s.n.], 2018. p. 588–594.

YU, T. et al. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In: RILOFF, E. et al. (Ed.). **Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing**. Brussels, Belgium: Association for Computational Linguistics, 2018. p. 3911–3921. Available at: <https://aclanthology.org/D18-1425>.

YU, X. et al. **Automatic Hallucination Assessment for Aligned Large Language Models via Transferable Adversarial Attacks**. 2024. Available at: https://openreview.net/forum?id=VQZCXoteoP.

ZELLE, J. M.; MOONEY, R. J. Learning to parse database queries using inductive logic programming. In: **Proceedings of the AAAI Conference on Artificial Intelligence**. Portland, Oregon: The AAAI Press, 1996. v. 13. ISBN 978-0-262-51091-2.

ZHONG, R.; YU, T.; KLEIN, D. **Semantic Evaluation for Text-to-SQL with Distilled Test Suites**. 2020. Available at: <https://arxiv.org/abs/2010.02840>.

ZHONG, V.; XIONG, C.; SOCHER, R. Seq2sql: Generating structured queries from natural language using reinforcement learning. **CoRR**, abs/1709.00103, 2017. Available at: <http://arxiv.org/abs/1709.00103>.

ZHOU, Y. et al. **Large Language Models Are Human-Level Prompt Engineers**. 2023. Available at: <https://arxiv.org/abs/2211.01910>.

ZIEGLER, A.; BERRYMAN, J. **A Developer's Guide to Prompt Engineering and LLMs - The GitHub Blog**. 2023. Available at: <https://github.blog/2023-07-17-developers-guide-prompt-engineering-llms/>. Prompt engineering is the art of communicating with a generative AI model.

A
Details of the strategies used in the text-to-SQL task

This section presents a comprehensive list of all the prompts utilized in the strategies used to allow for easy replication and understanding of the approaches.

A.1
LangChain-Based Strategies Prompts

A.1.1
SQLQueryChain

Listing 5: *SQLQueryChain* prompt in LangChain strategy

```
You are an Oracle SQL expert. Given an input question, first create a syntactically correct Oracle SQL query to run, then look at the results of the query and return the answer to the input question.
Unless the user specifies in the question a specific number of examples to obtain, don't query for at {top_k} most results or any using the FETCH FIRST n ROWS ONLY clause as per Oracle SQL.
Never query for all columns from a table. You must query only the columns that are needed to answer the question.
Pay attention to use only the column names you can see in the tables below. Be careful to not query for columns that do not exist. Also, pay attention to which column is in which table.
Pay attention to use TRUNC(SYSDATE) function to get the current date, if the question involves "today".

Some hints:
- Don't use double quotes in column name
- Don't use LEFT JOIN, only JOIN

Use the following format:

Question: Question here
SQLQuery: SQL Query to run
SQLResult: Result of the SQLQuery
Answer: Final answer here

Only use the following tables:
{schema}

Question: {question}
```

A.1.2 *SQLDatabaseSequentialChain*

Listing 6: *DeciderChain prompt template* from *SQLDatabaseSequentialChain* in LangChain strategy

```
Given the below input question and list of potential tables, output a comma separated list of the table names that may be necessary to answer this question.

Question: {question}

Table Names: {table_names}

Relevant Table Names:
```

A.1.3 *SQLAgent*

Listing 7: *SQLAgent* in LangChain strategy

```

You are an agent designed to interact with a SQL database.
Given an input question, create a syntactically correct oracle query to run, then look at query and return only the sql query.
You can order the results by a relevant column to return the most interesting examples in the database.
Never query for all the columns from a specific table, only ask for the relevant columns given the question.
You have access to tools for interacting with the database.
Only use the below tools. Only use the information returned by the below tools to construct your final answer.
You MUST double check your query before executing it. If you get an error while executing a query, rewrite the query and try again.

DO NOT make any DML statements (INSERT, UPDATE, DELETE, DROP etc.) to the database.

custom_sql_db_query: Input to this tool is a detailed and correct SQL query, output is a sql query. If the query is not correct, an error message will be returned. If an error is returned, rewrite the query, check the query, and try again. If you encounter an issue with Unknown column 'xxxx' in 'field list', using sql_db_schema to query the correct table fields.
sql_db_schema: Input to this tool is a comma-separated list of tables, output is the schema and sample rows for those tables. Be sure that the tables actually exist by calling sql_db_list_tables first! Example Input: 'table1, table2, table3'
```

```
sql_db_list_tables: Input is an empty string, output is a comma separated list of tables in the database.
sql_db_query_checker: Use this tool to double check if your query is correct before executing it. Always use this tool before executing a query with custom_sql_db_query!

Use the following format:

Question: the input question you must answer
Thought: you should always think about what to do
Action: the action to take, should be one of [sql_db_query, sql_db_schema, sql_db_list_tables, sql_db_query_checker]
Action Input: the input to the action
Observation: the result of the action
... (this Thought/Action/Action Input/Observation can repeat N times)
Thought: I now know the SQL QUERY generated
Final Answer: the SQL QUERY generated

Begin!

Question: {input}
Thought: I should look at the tables in the database to see what I can query. Then I should query the schema of the most relevant tables.
```

A.2 Prompts used in the C3 strategy

A.2.1 Table and Column recall prompts

Listing 8: Table Recall Prompt in C3 strategy

```
Given the database schema and question, perform the following actions:
1 - Rank all the tables based on the possibility of being used in the SQL according to the question from the most relevant to the
least relevant, Table or its column that matches more with the question words is highly relevant and must be placed ahead.
2 - Check whether you consider all the tables.
3 - Output a list object in the order of step 2, Your output should contain all the tables. The format should be like:
[
    "table_1", "table_2", ...
]

Schema:
{schema}
```

```

Question:
{question}
```

Listing 9: Column Recall Prompt in C3 strategy

```
Given the database tables and question, perform the following
    actions:
1 - Rank the columns in each table based on the possibility of being
     used in the SQL, Column that matches more with the question
words or the foreign key is highly relevant and must be placed ahead
    . You should output them in the order of the most
relevant to the least relevant.
Explain why you choose each column.
2 - Output a JSON object that contains all the columns in each table
     according to your explanation. The format should be like:
{{
    "table_1": ["column_1", "column_2", ......],
    "table_2": ["column_1", "column_2", ......],
    "table_3": ["column_1", "column_2", ......],
    ......
}}

Schema:
{schema}
{foreign_keys}

Question:
{question}
```

A.2.2
Calibrations with hints

Listing 10: Calibrations with hints provided to the LLM in C3 strategy using the LangChain

```

C3_HINTS_PROMPT =[
    SystemMessage(
        content="""
            You are now an excellent SQL writer, first I'll give you
                 some tips and examples, and I need you to
            remember the tips, and do not make same mistakes
            """
    ),
    HumanMessage(
        content="""
        Tips 1:
        Question: Which A has most number of B?
```

```
        Gold SQL: select A from B group by A order by count (*) desc fetch first 1 rows only;
        Notice that the Gold SQL doesn't select COUNT(*) because the question only wants to know the A and
        the number should be only used in ORDER BY clause, there are many questions asks in this way, and I
        need you to remember this in the the following questions.
        """
    ),
    AIMessage(
        content="""
            Thank you for the tip! I'll keep in mind that when the question only asks for a certain field, I should not
            include the COUNT(*) in the SELECT statement, but instead use it in the ORDER BY clause to sort the
            results based on the count of that field.
        """
    ),
    HumanMessage(
        content="""
        Tips 2:
        Don't use "IN", "OR", "LEFT JOIN" as it might cause extra results, use "INTERSECT" or "EXCEPT"
        instead, and remember to use "DISTINCT" or "FETCH FIRST" when necessary.
        For example,
        Question: Who are the A who have been nominated for both B award and C award?
        Gold SQL should be: select A from X where award = 'B' intersect select A from X where award = 'C';
        """
    ),
    AIMessage(
        content="""
        Thank you for the tip! I'll remember to use "INTERSECT" or "EXCEPT" instead of "IN", "NOT IN", or
        "LEFT JOIN" when I want to find records that match or don't match across two tables. Additionally, I'll
        make sure to use "DISTINCT" or "FETCH FIRST" when necessary to avoid repetitive results or limit the number
        of results returned.
        """
    )
]
```

A.3 Prompts used in DIN strategy

A.3.1 Schema Linking

Listing 11: Schema linking prompt

```
# Find the schema_links for generating SQL queries for each question
    based on the database schema and Foreign keys.
Table advisor, columns = [*,s_ID,i_ID]
Table classroom, columns = [*,building,room_number,capacity]
Table course, columns = [*,course_id,title,dept_name,credits]
Table department, columns = [*,dept_name,building,budget]
Table instructor, columns = [*,ID,name,dept_name,salary]
Table prereq, columns = [*,course_id,prereq_id]
Table section, columns = [*,course_id,sec_id,semester,year,building,
    room_number,time_slot_id]
Table student, columns = [*,ID,name,dept_name,tot_cred]
Table takes, columns = [*,ID,course_id,sec_id,semester,year,grade]
Table teaches, columns = [*,ID,course_id,sec_id,semester,year]
Table time_slot, columns = [*,time_slot_id,day,start_hr,start_min,
    end_hr,end_min]
Foreign_keys = [course.dept_name = department.dept_name,instructor.
    dept_name = department.dept_name,section.building = classroom.
    building,section.room_number = classroom.room_number,section.
    course_id = course.course_id,teaches.ID = instructor.ID,teaches.
    course_id = section.course_id,teaches.sec_id = section.sec_id,
    teaches.semester = section.semester,teaches.year = section.year,
    student.dept_name = department.dept_name,takes.ID = student.ID,
    takes.course_id = section.course_id,takes.sec_id = section.
    sec_id,takes.semester = section.semester,takes.year = section.
    year,advisor.s_ID = student.ID,advisor.i_ID = instructor.ID,
    prereq.prereq_id = course.course_id,prereq.course_id = course.
    course_id]
Q: "Find the buildings which have rooms with capacity more than 50."
A: Let's think step by step. In the question "Find the buildings
    which have rooms with capacity more than 50.", we are asked:
"the buildings which have rooms" so we need column = [classroom.
    capacity]
"rooms with capacity" so we need column = [classroom.building]
Based on the columns and tables, we need these Foreign_keys = [].
Based on the tables, columns, and Foreign_keys, The set of possible
    cell values are = [50]. So the Schema_links are:
Schema_links: [classroom.building,classroom.capacity,50]

Table department, columns = [*,Department_ID,Name,Creation,Ranking,
    Budget_in_Billions,Num_Employees]
Table head, columns = [*,head_ID,name,born_state,age]
Table management, columns = [*,department_ID,head_ID,
    temporary_acting]
Foreign_keys = [management.head_ID = head.head_ID,management.
    department_ID = department.Department_ID]
Q: "How many heads of the departments are older than 56 ?"
```

```
A: Let's think step by step. In the question "How many heads of the departments are older than 56 ?", we are asked:
"How many heads of the departments" so we need column = [head.*]
"older" so we need column = [head.age]
Based on the columns and tables, we need these Foreign_keys = [].
Based on the tables, columns, and Foreign_keys, The set of possible cell values are = [56]. So the Schema_links are:
Schema_links: [head.*,head.age,56]

Table department, columns = [*,Department_ID,Name,Creation,Ranking,Budget_in_Billions,Num_Employees]
Table head, columns = [*,head_ID,name,born_state,age]
Table management, columns = [*,department_ID,head_ID,temporary_acting]
Foreign_keys = [management.head_ID = head.head_ID,management.department_ID = department.Department_ID]
Q: "what are the distinct creation years of the departments managed by a secretary born in state 'Alabama'?"
A: Let's think step by step. In the question "what are the distinct creation years of the departments managed by a secretary born in state 'Alabama'?", we are asked:
"distinct creation years of the departments" so we need column = [department.Creation]
"departments managed by" so we need column = [management.department_ID]
"born in" so we need column = [head.born_state]
Based on the columns and tables, we need these Foreign_keys = [department.Department_ID = management.department_ID,management.head_ID = head.head_ID].
Based on the tables, columns, and Foreign_keys, The set of possible cell values are = ['Alabama']. So the Schema_links are:
Schema_links: [department.Creation,department.Department_ID = management.department_ID,head.head_ID = management.head_ID,head.born_state,'Alabama']

Table Addresses, columns = [*,address_id,line_1,line_2,city,zip_postcode,state_province_county,country]
Table Candidate_Assessments, columns = [*,candidate_id,qualification,assessment_date,asessment_outcome_code]
Table Candidates, columns = [*,candidate_id,candidate_details]
Table Courses, columns = [*,course_id,course_name,course_description,other_details]
Table People, columns = [*,person_id,first_name,middle_name,last_name,cell_mobile_number,email_address,login_name,password]
Table People_Addresses, columns = [*,person_address_id,person_id,address_id,date_from,date_to]
Table Student_Course_Attendance, columns = [*,student_id,course_id,date_of_attendance]
Table Student_Course_Registrations, columns = [*,student_id,course_id,registration_date]
Table Students, columns = [*,student_id,student_details]
```

```
Foreign_keys = [Students.student_id = People.person_id,
    People_Addresses.address_id = Addresses.address_id,
    People_Addresses.person_id = People.person_id,
    Student_Course_Registrations.course_id = Courses.course_id,
    Student_Course_Registrations.student_id = Students.student_id,
    Student_Course_Attendance.student_id =
    Student_Course_Registrations.student_id,
    Student_Course_Attendance.course_id =
    Student_Course_Registrations.course_id,Candidates.candidate_id =
     People.person_id,Candidate_Assessments.candidate_id =
    Candidates.candidate_id]
Q: "List the id of students who never attends courses?"
A: Let's think step by step. In the question "List the id of
    students who never attends courses?", we are asked:
"id of students" so we need column = [Students.student_id]
"never attends courses" so we need column = [
    Student_Course_Attendance.student_id]
Based on the columns and tables, we need these Foreign_keys = [
    Students.student_id = Student_Course_Attendance.student_id].
Based on the tables, columns, and Foreign_keys, The set of possible
    cell values are = []. So the Schema_links are:
Schema_links: [Students.student_id = Student_Course_Attendance.
    student_id]

Table Country, columns = [*,id,name]
Table League, columns = [*,id,country_id,name]
Table Player, columns = [*,id,player_api_id,player_name,
    player_fifa_api_id,birthday,height,weight]
Table Player_Attributes, columns = [*,id,player_fifa_api_id,
    player_api_id,date,overall_rating,potential,preferred_foot,
    attacking_work_rate,defensive_work_rate,crossing,finishing,
    heading_accuracy,short_passing,volleys,dribbling,curve,
    free_kick_accuracy,long_passing,ball_control,acceleration,
    sprint_speed,agility,reactions,balance,shot_power,jumping,
    stamina,strength,long_shots,aggression,interceptions,positioning
    ,vision,penalties,marking,standing_tackle,sliding_tackle,
    gk_diving,gk_handling,gk_kicking,gk_positioning,gk_reflexes]
Table Team, columns = [*,id,team_api_id,team_fifa_api_id,
    team_long_name,team_short_name]
Table Team_Attributes, columns = [*,id,team_fifa_api_id,team_api_id,
    date,buildUpPlaySpeed,buildUpPlaySpeedClass,buildUpPlayDribbling
    ,buildUpPlayDribblingClass,buildUpPlayPassing,
    buildUpPlayPassingClass,buildUpPlayPositioningClass,
    chanceCreationPassing,chanceCreationPassingClass,
    chanceCreationCrossing,chanceCreationCrossingClass,
    chanceCreationShooting,chanceCreationShootingClass,
    chanceCreationPositioningClass,defencePressure,
    defencePressureClass,defenceAggression,defenceAggressionClass,
    defenceTeamWidth,defenceTeamWidthClass,defenceDefenderLineClass]
Table sqlite_sequence, columns = [*,name,seq]
```

```
Foreign_keys = [Player_Attributes.player_api_id = Player.player_api_id,Player_Attributes.player_fifa_api_id = Player.player_fifa_api_id,League.country_id = Country.id,Team_Attributes.team_api_id = Team.team_api_id,Team_Attributes.team_fifa_api_id = Team.team_fifa_api_id]
Q: "List the names of all left-footed players who have overall rating between 85 and 90."
A: Let's think step by step. In the question "List the names of all left-footed players who have overall rating between 85 and 90.", we are asked:
"names of all left-footed players" so we need column = [Player.player_name,Player_Attributes.preferred_foot]
"players who have overall rating" so we need column = [Player_Attributes.overall_rating]
Based on the columns and tables, we need these Foreign_keys = [Player_Attributes.player_api_id = Player.player_api_id].
Based on the tables, columns, and Foreign_keys, The set of possible cell values are = [left,85,90]. So the Schema_links are:
Schema_links: [Player.player_name,Player_Attributes.preferred_foot,Player_Attributes.overall_rating,Player_Attributes.player_api_id = Player.player_api_id,left,85,90]

Table advisor, columns = [*,s_ID,i_ID]
Table classroom, columns = [*,building,room_number,capacity]
Table course, columns = [*,course_id,title,dept_name,credits]
Table department, columns = [*,dept_name,building,budget]
Table instructor, columns = [*,ID,name,dept_name,salary]
Table prereq, columns = [*,course_id,prereq_id]
Table section, columns = [*,course_id,sec_id,semester,year,building,room_number,time_slot_id]
Table student, columns = [*,ID,name,dept_name,tot_cred]
Table takes, columns = [*,ID,course_id,sec_id,semester,year,grade]
Table teaches, columns = [*,ID,course_id,sec_id,semester,year]
Table time_slot, columns = [*,time_slot_id,day,start_hr,start_min,end_hr,end_min]
Foreign_keys = [course.dept_name = department.dept_name,instructor.dept_name = department.dept_name,section.building = classroom.building,section.room_number = classroom.room_number,section.course_id = course.course_id,teaches.ID = instructor.ID,teaches.course_id = section.course_id,teaches.sec_id = section.sec_id,teaches.semester = section.semester,teaches.year = section.year,student.dept_name = department.dept_name,takes.ID = student.ID,takes.course_id = section.course_id,takes.sec_id = section.sec_id,takes.semester = section.semester,takes.year = section.year,advisor.s_ID = student.ID,advisor.i_ID = instructor.ID,prereq.prereq_id = course.course_id,prereq.course_id = course.course_id]
Q: "Give the title of the course offered in Chandler during the Fall of 2010."
A: Let's think step by step. In the question "Give the title of the course offered in Chandler during the Fall of 2010.", we are
```

```
asked:
"title of the course" so we need column = [course.title]
"course offered in Chandler" so we need column = [SECTION.building]
"during the Fall" so we need column = [SECTION.semester]
"of 2010" so we need column = [SECTION.year]
Based on the columns and tables, we need these Foreign_keys = [course.course_id = SECTION.course_id].
Based on the tables, columns, and Foreign_keys, The set of possible cell values are = [Chandler,Fall,2010]. So the Schema_links are:
Schema_links: [course.title,course.course_id = SECTION.course_id,SECTION.building,SECTION.year,SECTION.semester,Chandler,Fall,2010]

Table city, columns = [*,City_ID,Official_Name,Status,Area_km_2,Population,Census_Ranking]
Table competition_record, columns = [*,Competition_ID,Farm_ID,Rank]
Table farm, columns = [*,Farm_ID,Year,Total_Horses,Working_Horses,Total_Cattle,Oxen,Bulls,Cows,Pigs,Sheep_and_Goats]
Table farm_competition, columns = [*,Competition_ID,Year,Theme,Host_city_ID,Hosts]
Foreign_keys = [farm_competition.Host_city_ID = city.City_ID,competition_record.Farm_ID = farm.Farm_ID,competition_record.Competition_ID = farm_competition.Competition_ID]
Q: "Show the status of the city that has hosted the greatest number of competitions."
A: Let's think step by step. In the question "Show the status of the city that has hosted the greatest number of competitions.", we are asked:
"the status of the city" so we need column = [city.Status]
"greatest number of competitions" so we need column = [farm_competition.*]
Based on the columns and tables, we need these Foreign_keys = [farm_competition.Host_city_ID = city.City_ID].
Based on the tables, columns, and Foreign_keys, The set of possible cell values are = []. So the Schema_links are:
Schema_links: [city.Status,farm_competition.Host_city_ID = city.City_ID,farm_competition.*]

Table advisor, columns = [*,s_ID,i_ID]
Table classroom, columns = [*,building,room_number,capacity]
Table course, columns = [*,course_id,title,dept_name,credits]
Table department, columns = [*,dept_name,building,budget]
Table instructor, columns = [*,ID,name,dept_name,salary]
Table prereq, columns = [*,course_id,prereq_id]
Table section, columns = [*,course_id,sec_id,semester,year,building,room_number,time_slot_id]
Table student, columns = [*,ID,name,dept_name,tot_cred]
Table takes, columns = [*,ID,course_id,sec_id,semester,year,grade]
Table teaches, columns = [*,ID,course_id,sec_id,semester,year]
Table time_slot, columns = [*,time_slot_id,day,start_hr,start_min,end_hr,end_min]
```

```
Foreign_keys = [course.dept_name = department.dept_name,instructor.dept_name = department.dept_name,section.building = classroom.building,section.room_number = classroom.room_number,section.course_id = course.course_id,teaches.ID = instructor.ID,teaches.course_id = section.course_id,teaches.sec_id = section.sec_id,teaches.semester = section.semester,teaches.year = section.year,student.dept_name = department.dept_name,takes.ID = student.ID,takes.course_id = section.course_id,takes.sec_id = section.sec_id,takes.semester = section.semester,takes.year = section.year,advisor.s_ID = student.ID,advisor.i_ID = instructor.ID,prereq.prereq_id = course.course_id,prereq.course_id = course.course_id]
Q: "Find the id of instructors who taught a class in Fall 2009 but not in Spring 2010."
A: Let's think step by step. In the question "Find the id of instructors who taught a class in Fall 2009 but not in Spring 2010.", we are asked:
"id of instructors who taught " so we need column = [teaches.id]
"taught a class in" so we need column = [teaches.semester,teaches.year]
Based on the columns and tables, we need these Foreign_keys = [].
Based on the tables, columns, and Foreign_keys, The set of possible cell values are = [Fall,2009,Spring,2010]. So the Schema_links are:
Schema_links: [teaches.id,teaches.semester,teaches.year,Fall,2009,Spring,2010]

Table Accounts, columns = [*,account_id,customer_id,date_account_opened,account_name,other_account_details]
Table Customers, columns = [*,customer_id,customer_first_name,customer_middle_initial,customer_last_name,gender,email_address,login_name,login_password,phone_number,town_city,state_county_province,country]
Table Financial_Transactions, columns = [*,transaction_id,account_id,invoice_number,transaction_type,transaction_date,transaction_amount,transaction_comment,other_transaction_details]
Table Invoice_Line_Items, columns = [*,order_item_id,invoice_number,product_id,product_title,product_quantity,product_price,derived_product_cost,derived_vat_payable,derived_total_cost]
Table Invoices, columns = [*,invoice_number,order_id,invoice_date]
Table Order_Items, columns = [*,order_item_id,order_id,product_id,product_quantity,other_order_item_details]
Table Orders, columns = [*,order_id,customer_id,date_order_placed,order_details]
Table Product_Categories, columns = [*,production_type_code,product_type_description,vat_rating]
Table Products, columns = [*,product_id,parent_product_id,production_type_code,unit_price,product_name,product_color,product_size]
```

```
Foreign_keys = [Orders.customer_id = Customers.customer_id,Invoices.order_id = Orders.order_id,Accounts.customer_id = Customers.customer_id,Products.production_type_code = Product_Categories.production_type_code,Financial_Transactions.account_id = Accounts.account_id,Financial_Transactions.invoice_number = Invoices.invoice_number,Order_Items.order_id = Orders.order_id,Order_Items.product_id = Products.product_id,Invoice_Line_Items.product_id = Products.product_id,Invoice_Line_Items.invoice_number = Invoices.invoice_number,Invoice_Line_Items.order_item_id = Order_Items.order_item_id]
Q: "Show the id, the date of account opened, the account name, and other account detail for all accounts."
A: Let's think step by step. In the question "Show the id, the date of account opened, the account name, and other account detail for all accounts.", we are asked:
"the id, the date of account opened, the account name, and other account detail for all accounts." so we need column = [Accounts.account_id,Accounts.account_name,Accounts.other_account_details,Accounts.date_account_opened]
Based on the columns and tables, we need these Foreign_keys = [].
Based on the tables, columns, and Foreign_keys, The set of possible cell values are = []. So the Schema_links are:
Schema_links: [Accounts.account_id,Accounts.account_name,Accounts.other_account_details,Accounts.date_account_opened]

Table city, columns = [*,City_ID,Official_Name,Status,Area_km_2,Population,Census_Ranking]
Table competition_record, columns = [*,Competition_ID,Farm_ID,Rank]
Table farm, columns = [*,Farm_ID,Year,Total_Horses,Working_Horses,Total_Cattle,Oxen,Bulls,Cows,Pigs,Sheep_and_Goats]
Table farm_competition, columns = [*,Competition_ID,Year,Theme,Host_city_ID,Hosts]
Foreign_keys = [farm_competition.Host_city_ID = city.City_ID,competition_record.Farm_ID = farm.Farm_ID,competition_record.Competition_ID = farm_competition.Competition_ID]
Q: "Show the status shared by cities with population bigger than 1500 and smaller than 500."
A: Let's think step by step. In the question "Show the status shared by cities with population bigger than 1500 and smaller than 500.", we are asked:
"the status shared by cities" so we need column = [city.Status]
"cities with population" so we need column = [city.Population]
Based on the columns and tables, we need these Foreign_keys = [].
Based on the tables, columns, and Foreign_keys, The set of possible cell values are = [1500,500]. So the Schema_links are:
Schema_links: [city.Status,city.Population,1500,500]

{schema}
Foreign_keys = {foreign_keys}
Q: {question}
A: Let's think step by step.
```

A.3.2
Classification & decomposition

Listing 12: Classification & decomposition prompt

```
# For the given question, classify it as EASY, NON-NESTED, or NESTED based on nested queries and JOIN.

if need nested queries: predict NESTED
elif need JOIN and don't need nested queries: predict NON-NESTED
elif don't need JOIN and don't need nested queries: predict EASY

{schema}
Foreign_keys = {foreign_keys}

Q: "Find the buildings which have rooms with capacity more than 50."
schema_links: [classroom.building,classroom.capacity,50]
A: Let's think step by step. The SQL query for the question "Find the buildings which have rooms with capacity more than 50." needs these
tables = [classroom], so we don't need JOIN.
Plus, it doesn't require nested queries with (INTERSECT, UNION, EXCEPT, IN, NOT IN), and we need the answer to the questions = [""].
So, we don't need JOIN and don't need nested queries, then the the SQL query can be classified as "EASY".
Label: "EASY"

Q: "What are the names of all instructors who advise students in the math depart sorted by total credits of the student."
schema_links: [advisor.i_id = instructor.id,advisor.s_id = student.id,instructor.name,student.dept_name,student.tot_cred,math]
A: Let's think step by step. The SQL query for the question "What are the names of all instructors who advise students in the math depart sorted by total credits of the student." needs these tables = [advisor,instructor,student], so we need JOIN.
Plus, it doesn't need nested queries with (INTERSECT, UNION, EXCEPT, IN, NOT IN), and we need the answer to the questions = [""].
So, we need JOIN and don't need nested queries, then the the SQL query can be classified as "NON-NESTED".
Label: "NON-NESTED"

Q: "Find the room number of the rooms which can sit 50 to 100 students and their buildings."
schema_links: [classroom.building,classroom.room_number,classroom.capacity,50,100]
A: Let's think step by step. The SQL query for the question "Find the room number of the rooms which can sit 50 to 100 students and their buildings." needs these tables = [classroom], so we don't need JOIN.
```

```
Plus, it doesn't require nested queries with (INTERSECT, UNION, EXCEPT, IN, NOT IN), and we need the answer to the questions = [""].
So, we don't need JOIN and don't need nested queries, then the the SQL query can be classified as "EASY".
Label: "EASY"

Q: "How many courses that do not have prerequisite?"
schema_links: [course.*,course.course_id = prereq.course_id]
A: Let's think step by step. The SQL query for the question "How many courses that do not have prerequisite?" needs these tables = [course,prereq], so we need JOIN.
Plus, it requires nested queries with (INTERSECT, UNION, EXCEPT, IN, NOT IN), and we need the answer to the questions = ["Which courses have prerequisite?"].
So, we need JOIN and need nested queries, then the the SQL query can be classified as "NESTED".
Label: "NESTED"

Q: "Find the title of course that is provided by both Statistics and Psychology departments."
schema_links: [course.title,course.dept_name,Statistics,Psychology]
A: Let's think step by step. The SQL query for the question "Find the title of course that is provided by both Statistics and Psychology departments." needs these tables = [course], so we don't need JOIN.
Plus, it requires nested queries with (INTERSECT, UNION, EXCEPT, IN, NOT IN), and we need the answer to the questions = ["Find the titles of courses that is provided by Psychology departments"].
So, we don't need JOIN and need nested queries, then the the SQL query can be classified as "NESTED".
Label: "NESTED"

Q: "Find the id of instructors who taught a class in Fall 2009 but not in Spring 2010."
schema_links: [teaches.id,teaches.semester,teaches.year,Fall,2009,Spring,2010]
A: Let's think step by step. The SQL query for the question "Find the id of instructors who taught a class in Fall 2009 but not in Spring 2010." needs these tables = [teaches], so we don't need JOIN.
Plus, it requires nested queries with (INTERSECT, UNION, EXCEPT, IN, NOT IN), and we need the answer to the questions = ["Find the id of instructors who taught a class in Spring 2010"].
So, we don't need JOIN and need nested queries, then the the SQL query can be classified as "NESTED".
Label: "NESTED"

Q: "Find the name of the department that offers the highest total credits?"
schema_links: [course.dept_name,course.credits]
```

```
A: Let's think step by step. The SQL query for the question "Find the name of the department that offers the highest total credits?." needs these tables = [course], so we don't need JOIN.
Plus, it doesn't require nested queries with (INTERSECT, UNION, EXCEPT, IN, NOT IN), and we need the answer to the questions = [""].
So, we don't need JOIN and don't need nested queries, then the the SQL query can be classified as "EASY".
Label: "EASY"

Q: "What is the name of the instructor who advises the student with the greatest number of total credits?"
schema_links: [advisor.i_id = instructor.id,advisor.s_id = student.id,instructor.name,student.tot_cred ]
A: Let's think step by step. The SQL query for the question "What is the name of the instructor who advises the student with the greatest number of total credits?" needs these tables = [advisor,instructor,student], so we need JOIN.
Plus, it doesn't need nested queries with (INTERSECT, UNION, EXCEPT, IN, NOT IN), and we need the answer to the questions = [""].
So, we need JOIN and don't need nested queries, then the the SQL query can be classified as "NON-NESTED".
Label: "NON-NESTED"

Q: "Find the total number of students and total number of instructors for each department."
schema_links = [department.dept_name = instructor.dept_name,student.id,student.dept_name = department.dept_name,instructor.id]
A: Let's think step by step. The SQL query for the question "Find the total number of students and total number of instructors for each department." needs these tables = [department,instructor,student], so we need JOIN.
Plus, it doesn't need nested queries with (INTERSECT, UNION, EXCEPT, IN, NOT IN), and we need the answer to the questions = [""].
So, we need JOIN and don't need nested queries, then the the SQL query can be classified as "NON-NESTED".
Label: "NON-NESTED"

Q: "Give the name and building of the departments with greater than average budget."
schema_links: [department.budget,department.dept_name,department.building]
A: Let's think step by step. The SQL query for the question "Give the name and building of the departments with greater than average budget." needs these tables = [department], so we don't need JOIN.
Plus, it requires nested queries with (INTERSECT, UNION, EXCEPT, IN, NOT IN), and we need the answer to the questions = ["What is the average budget of the departments"].
So, we don't need JOIN and need nested queries, then the the SQL query can be classified as "NESTED".
```

```
Label: "NESTED"

Q: {question}
schema_links: {schema_links}
A: Let's think step by step.
```

A.3.3 SQL Generation

Listing 13: Easy prompt

```
# Use the the schema links to generate the SQL queries for each of
    the questions.

{schema}

Q: "Find the buildings which have rooms with capacity more than 50."
Schema_links: [classroom.building,classroom.capacity,50]
SQL: SELECT DISTINCT building FROM classroom WHERE capacity  >  50

Q: "Find the room number of the rooms which can sit 50 to 100
    students and their buildings."
Schema_links: [classroom.building,classroom.room_number,classroom.
    capacity,50,100]
SQL: SELECT building ,  room_number FROM classroom WHERE capacity
    BETWEEN 50 AND 100

Q: "Give the name of the student in the History department with the
    most credits."
Schema_links: [student.name,student.dept_name,student.tot_cred,
    History]
SQL: SELECT name FROM student WHERE dept_name  =  'History' ORDER BY
     tot_cred DESC FETCH FIRST 1 ROWS ONLY

Q: "Find the total budgets of the Marketing or Finance department."
Schema_links: [department.budget,department.dept_name,Marketing,
    Finance]
SQL: SELECT sum(budget) FROM department WHERE dept_name  =  '
    Marketing' OR dept_name  =  'Finance'

Q: "Find the department name of the instructor whose name contains '
    Soisalon'."
Schema_links: [instructor.dept_name,instructor.name,Soisalon]
SQL: SELECT dept_name FROM instructor WHERE name LIKE '%Soisalon%'

Q: "What is the name of the department with the most credits?"
Schema_links: [course.dept_name,course.credits]
SQL: SELECT dept_name FROM course GROUP BY dept_name ORDER BY sum(
    credits) DESC FETCH FIRST 1 ROWS ONLY

```

```
Q: "How many instructors teach a course in the Spring of 2010?"
Schema_links: [teaches.ID,teaches.semester,teaches.YEAR,Spring,2010]
SQL: SELECT COUNT (DISTINCT ID) FROM teaches WHERE semester = 'Spring' AND YEAR = 2010

Q: "Find the name of the students and their department names sorted by their total credits in ascending order."
Schema_links: [student.name,student.dept_name,student.tot_cred]
SQL: SELECT name , dept_name FROM student ORDER BY tot_cred

Q: "Find the year which offers the largest number of courses."
Schema_links: [SECTION.YEAR,SECTION.*]
SQL: SELECT YEAR FROM SECTION GROUP BY YEAR ORDER BY count(*) DESC FETCH FIRST 1 ROWS ONLY

Q: "What are the names and average salaries for departments with average salary higher than 42000?"
Schema_links: [instructor.dept_name,instructor.salary,42000]
SQL: SELECT dept_name , AVG (salary) FROM instructor GROUP BY dept_name HAVING AVG (salary) > 42000

Q: "How many rooms in each building have a capacity of over 50?"
Schema_links: [classroom.*,classroom.building,classroom.capacity,50]
SQL: SELECT count(*) , building FROM classroom WHERE capacity > 50 GROUP BY building

Q: "Find the names of the top 3 departments that provide the largest amount of courses?"
Schema_links: [course.dept_name,course.*]
SQL: SELECT dept_name FROM course GROUP BY dept_name ORDER BY count(*) DESC FETCH FIRST 3 ROWS ONLY

Q: "Find the maximum and average capacity among rooms in each building."
Schema_links: [classroom.building,classroom.capacity]
SQL: SELECT max(capacity) , avg(capacity) , building FROM classroom GROUP BY building

Q: "Find the title of the course that is offered by more than one department."
Schema_links: [course.title]
SQL: SELECT title FROM course GROUP BY title HAVING count(*) > 1

Q: {question}
Schema_links: {schema_links}
```

Listing 14: Non-Nested prompt

```
 # Use the the schema links and Intermediate_representation to generate the SQL queries for each of the questions.

```

```
{schema}
Foreign_keys = {foreign_keys}

Q: "Find the total budgets of the Marketing or Finance department."
Schema_links: [department.budget,department.dept_name,Marketing,Finance]
A: Letâ s think step by step. For creating the SQL for the given question, we need to join these tables = []. First, create an intermediate representation, then use it to construct the SQL query.
Intermediate_representation: select sum(department.budget) from department  where  department.dept_name = \"Marketing\"  or department.dept_name = \"Finance\"
SQL: SELECT sum(budget) FROM department WHERE dept_name  =  'Marketing' OR dept_name  =  'Finance'

Q: "Find the name and building of the department with the highest budget."
Schema_links: [department.budget,department.dept_name,department.building]
A: Letâ s think step by step. For creating the SQL for the given question, we need to join these tables = []. First, create an intermediate representation, then use it to construct the SQL query.
Intermediate_representation: select department.dept_name , department.building from department  order by department.budget desc limit 1
SQL: SELECT dept_name ,  building FROM department ORDER BY budget DESC FETCH FIRST 1 ROWS ONLY

Q: "What is the name and building of the departments whose budget is more than the average budget?"
Schema_links: [department.budget,department.dept_name,department.building]
A: Letâ s think step by step. For creating the SQL for the given question, we need to join these tables = []. First, create an intermediate representation, then use it to construct the SQL query.
Intermediate_representation:  select department.dept_name , department.building from department  where  @.@ > avg ( department.budget )
SQL: SELECT dept_name ,  building FROM department WHERE budget  >  (SELECT avg(budget) FROM department)

Q: "Find the total number of students and total number of instructors for each department."
Schema_links: [department.dept_name = student.dept_name,student.id,department.dept_name = instructor.dept_name,instructor.id]
A: Letâ s think step by step. For creating the SQL for the given question, we need to join these tables = [department,student,instructor]. First, create an intermediate representation, then
```

```
use it to construct the SQL query.
Intermediate_representation: "select count( distinct student.ID) , count( distinct instructor.ID) , department.dept_name from department  group by instructor.dept_name
SQL: SELECT count(DISTINCT T2.id) ,  count(DISTINCT T3.id) ,  T3.dept_name FROM department T1 JOIN student T2 ON T1.dept_name  =  T2.dept_name JOIN instructor AS T3 ON T1.dept_name  =  T3.dept_name GROUP BY T3.dept_name

Q: "Find the title of courses that have two prerequisites?"
Schema_links: [course.title,course.course_id = prereq.course_id]
A: Letâ s think step by step. For creating the SQL for the given question, we need to join these tables = [course,prereq]. First, create an intermediate representation, then use it to construct the SQL query.
Intermediate_representation: select course.title from course  where  count ( prereq.* )  = 2  group by prereq.course_id
SQL: SELECT T1.title FROM course T1 JOIN prereq T2 ON T1.course_id  =  T2.course_id GROUP BY T2.course_id HAVING count(*)  =  2

Q: "Find the name of students who took any class in the years of 2009 and 2010."
Schema_links: [student.name,student.id = takes.id,takes.YEAR,2009,2010]
A: Let's think step by step. For creating the SQL for the given question, we need to join these tables = [student,takes]. First, create an intermediate representation, then use it to construct the SQL query.
Intermediate_representation: select  distinct student.name from student  where  takes.year = 2009  or  takes.year = 2010
SQL: SELECT DISTINCT T1.name FROM student T1 JOIN takes T2 ON T1.id  =  T2.id WHERE T2.YEAR  =  2009 OR T2.YEAR  =  2010

Q: "list in alphabetic order all course names and their instructors' names in year 2008."
Schema_links: [course.title,course.course_id = teaches.course_id,teaches.id = instructor.id,instructor.name,teaches.year,2008]
A: Let's think step by step. For creating the SQL for the given question, we need to join these tables = [course,teaches,instructor]. First, create an intermediate representation, then use it to construct the SQL query.
Intermediate_representation: select course.title , instructor.name from course  where  teaches.year = 2008  order by course.title asc
SQL: SELECT T1.title ,  T3.name FROM course T1 JOIN teaches T2 ON T1.course_id  =  T2.course_id JOIN instructor AS T3 ON T2.id  =  T3.id WHERE T2.YEAR  =  2008 ORDER BY T1.title

Q: {question}
Schema_links: {schema_links}
A: Let's think step by step.
```

Listing 15: Nested prompt

```
# Use the the schema links and Intermediate_representation to generate the SQL queries for each of the questions.

{schema}
Foreign_keys = {foreign_keys}

Q: "Find the title of courses that have two prerequisites?"
Schema_links: [course.title,course.course_id = prereq.course_id]
A: Let's think step by step. "Find the title of courses that have two prerequisites?" can be solved by knowing the answer to the following sub-question "What are the titles for courses with two prerequisites?".
The SQL query for the sub-question "What are the titles for courses with two prerequisites?" is SELECT T1.title FROM course T1 JOIN prereq T2 ON T1.course_id  =  T2.course_id GROUP BY T2.course_id HAVING count(*)  =  2
So, the answer to the question "Find the title of courses that have two prerequisites?" is =
Intermediate_representation: select course.title from course  where  count ( prereq.* )  = 2  group by prereq.course_id
SQL: SELECT T1.title FROM course T1 JOIN prereq T2 ON T1.course_id  =  T2.course_id GROUP BY T2.course_id HAVING count(*)  =  2

Q: "Find the name and building of the department with the highest budget."
Schema_links: [department.dept_name,department.building,department.budget]
A: Let's think step by step. "Find the name and building of the department with the highest budget." can be solved by knowing the answer to the following sub-question "What is the department name and corresponding building for the department with the greatest budget?".
The SQL query for the sub-question "What is the department name and corresponding building for the department with the greatest budget?" is SELECT dept_name ,  building FROM department ORDER BY budget DESC FETCH FIRST 1 ROWS ONLY
So, the answer to the question "Find the name and building of the department with the highest budget." is =
Intermediate_representation: select department.dept_name , department.building from department  order by department.budget desc limit 1
SQL: SELECT dept_name ,  building FROM department ORDER BY budget DESC FETCH FIRST 1 ROWS ONLY

Q: "Find the title, credit, and department name of courses that have more than one prerequisites?"
Schema_links: [course.title,course.credits,course.dept_name,course.course_id = prereq.course_id]
A: Let's think step by step. "Find the title, credit, and department name of courses that have more than one prerequisites?" can be
```

```
solved by knowing the answer to the following sub-question "What is the title, credit value, and department name for courses with more than one prerequisite?".
The SQL query for the sub-question "What is the title, credit value, and department name for courses with more than one prerequisite?" is SELECT T1.title ,  T1.credits , T1.dept_name FROM course T1 JOIN prereq T2 ON T1.course_id  =  T2.course_id GROUP BY T2.course_id HAVING count(*)  >  1
So, the answer to the question "Find the name and building of the department with the highest budget." is =
Intermediate_representation: select course.title , course.credits , course.dept_name from course  where  count ( prereq.* )  > 1 group by prereq.course_id
SQL: SELECT T1.title ,  T1.credits , T1.dept_name FROM course T1 JOIN prereq T2 ON T1.course_id  =  T2.course_id GROUP BY T2.course_id HAVING count(*)  >  1

Q: "Give the name and building of the departments with greater than average budget."
Schema_links: [department.dept_name,department.building,department.budget]
A: Let's think step by step. "Give the name and building of the departments with greater than average budget." can be solved by knowing the answer to the following sub-question "What is the average budget of departments?".
The SQL query for the sub-question "What is the average budget of departments?" is SELECT avg(budget) FROM department
So, the answer to the question "Give the name and building of the departments with greater than average budget." is =
Intermediate_representation: select department.dept_name , department.building from department  where  @.@ > avg ( department.budget )
SQL: SELECT dept_name ,  building FROM department WHERE budget  >  (SELECT avg(budget) FROM department)

Q: "Find the id of instructors who taught a class in Fall 2009 but not in Spring 2010."
Schema_links: [teaches.id,teaches.semester,teaches.YEAR,Fall,2009,Spring,2010]
A: Let's think step by step. "Find the id of instructors who taught a class in Fall 2009 but not in Spring 2010." can be solved by knowing the answer to the following sub-question "Find the id of instructors who taught a class in Spring 2010".
The SQL query for the sub-question "Find the id of instructors who taught a class in Spring 2010" is SELECT id FROM teaches WHERE semester  =  'Spring' AND YEAR  =  2010
So, the answer to the question "Find the id of instructors who taught a class in Fall 2009 but not in Spring 2010." is =
Intermediate_representation: select teaches.ID from teaches  where teaches.semester = \"Fall\"  and  teaches.year = 2009  and teaches.semester != \"Spring\"  and  teaches.year = 2010
```

```
SQL: SELECT id FROM teaches WHERE semester  =  'Fall' AND YEAR  =  2009 EXCEPT SELECT id FROM teaches WHERE semester  =  'Spring' AND YEAR  =  2010

Q: "Find the name of the courses that do not have any prerequisite?"
Schema_links: [course.title,course.course_id]
A: Let's think step by step. "Find the name of the courses that do not have any prerequisite?" can be solved by knowing the answer to the following sub-question "What are the courses that have any prerequisite?".
The SQL query for the sub-question "What are the courses that have any prerequisite?" is SELECT course_id FROM prereq
So, the answer to the question "Find the name of the courses that do not have any prerequisite?" is =
Intermediate_representation: select course.title from course  where @.@ not in prereq.course_id
SQL: SELECT title FROM course WHERE course_id NOT IN (SELECT course_id FROM prereq)

Q: "Find the salaries of all distinct instructors that are less than the largest salary."
Schema_links: [instructor.salary]
A: Let's think step by step. "Find the salaries of all distinct instructors that are less than the largest salary." can be solved by knowing the answer to the following sub-question "What is the largest salary of instructors".
The SQL query for the sub-question "What is the largest salary of instructors" is SELECT max(salary) FROM instructor
So, the answer to the question "Find the salaries of all distinct instructors that are less than the largest salary." is =
Intermediate_representation: select  distinct instructor.salary from instructor  where  @.@ < max ( instructor.salary )
SQL: SELECT DISTINCT salary FROM instructor WHERE salary  <  (SELECT max(salary) FROM instructor)

Q: "Find the names of students who have taken any course in the fall semester of year 2003."
Schema_links: [student.id,student.name,takes.id,takes.semester,fall,2003]
A: Let's think step by step. "Find the names of students who have taken any course in the fall semester of year 2003." can be solved by knowing the answer to the following sub-question "Find the students who have taken any course in the fall semester of year 2003.".
The SQL query for the sub-question "Find the students who have taken any course in the fall semester of year 2003." is SELECT id FROM takes WHERE semester  =  'Fall' AND YEAR  =  2003
So, the answer to the question "Find the names of students who have taken any course in the fall semester of year 2003." is =
Intermediate_representation: select student.name from student  where  takes.semester = \"Fall\"  and  takes.year = 2003
```

```
SQL: SELECT name FROM student WHERE id IN (SELECT id FROM takes WHERE semester  =  'Fall' AND YEAR  =  2003)

Q: "Find the minimum salary for the departments whose average salary is above the average payment of all instructors."
Schema_links: [instructor.salary,instructor.dept_name]
A: Let's think step by step. "Find the minimum salary for the departments whose average salary is above the average payment of all instructors." can be solved by knowing the answer to the following sub-question "What is the average payment of all instructors.".
The SQL query for the sub-question "What is the average payment of all instructors." is SELECT avg(salary) FROM instructor
So, the answer to the question "Find the minimum salary for the departments whose average salary is above the average payment of all instructors." is =
Intermediate_representation: select min(instructor.salary) , instructor.dept_name from instructor  where  avg ( instructor.salary )  > avg ( instructor.salary )   group by instructor.dept_name
SQL: SELECT min(salary) ,  dept_name FROM instructor GROUP BY dept_name HAVING avg(salary)  >  (SELECT avg(salary) FROM instructor)

Q: "What is the course title of the prerequisite of course Mobile Computing?"
Schema_links: [course.title,course.course_id = prereq.course_id,prereq.prereq_id,course.title,Mobile Computing]
A: Let's think step by step. "What is the course title of the prerequisite of course Mobile Computing?" can be solved by knowing the answer to the following sub-question "What are the ids of the prerequisite of course Mobile Computing?".
The SQL query for the sub-question "What are the ids of the prerequisite of course Mobile Computing?" is SSELECT T1.prereq_id FROM prereq T1 JOIN course T2 ON T1.course_id  =  T2.course_id WHERE T2.title  =  'Mobile Computing'
So, the answer to the question "What is the course title of the prerequisite of course Mobile Computing?" is =
Intermediate_representation: select course.title from course  where  @.@ in prereq.*  and  course.title = \"Mobile Computing\"
SQL: SELECT title FROM course WHERE course_id IN (SELECT T1.prereq_id FROM prereq T1 JOIN course T2 ON T1.course_id  =  T2.course_id WHERE T2.title  =  'Mobile Computing')

Q: "Give the title and credits for the course that is taught in the classroom with the greatest capacity."
Schema_links: [classroom.capacity,classroom.building = SECTION.building,classroom.room_number = SECTION.room_number,course.title,course.credits,course.course_id = SECTION.course_id]
A: Let's think step by step. "Give the title and credits for the course that is taught in the classroom with the greatest
```

```
    capacity." can be solved by knowing the answer to the following
    sub-question "What is the capacity of the largest room?".
The SQL query for the sub-question "What is the capacity of the
    largest room?" is (SELECT max(capacity) FROM classroom)
So, the answer to the question "Give the title and credits for the
    course that is taught in the classroom with the greatest
    capacity." is =
Intermediate_representation: select course.title , course.credits
    from classroom  order by classroom.capacity desc limit 1"
SQL: SELECT T3.title ,  T3.credits FROM classroom T1 JOIN SECTION T2
     ON T1.building  =  T2.building AND T1.room_number  =  T2.
    room_number JOIN course AS T3 ON T2.course_id  =  T3.course_id
    WHERE T1.capacity  =  (SELECT max(capacity) FROM classroom)

Q: {question}
Schema_links: {schema_links}
A: Let's think step by step. {question} can be solved by knowing the
     answer to the following sub-question "{sub_questions}".
The SQL query for the sub-question
```

A.3.4 Self-Correction

Listing 16: Self-Correction prompt

```
#### For the given question, use the provided tables, columns,
    foreign keys, and primary keys to fix the given Oracle SQL QUERY
     for any issues. If there are any problems, fix them. If there
    are no issues, return the Oracle SQL QUERY as is.
#### Use the following instructions for fixing the SQL QUERY:
1) Use the database values that are explicitly mentioned in the
    question.
2) Pay attention to the columns that are used for the JOIN by using
    the Foreign_keys.
3) Use DESC and DISTINCT when needed.
4) Pay attention to the columns that are used for the GROUP BY
    statement.
5) Pay attention to the columns that are used for the SELECT
    statement.
6) Only change the GROUP BY clause when necessary (Avoid redundant
    columns in GROUP BY).
7) Use GROUP BY on one column only.
8) Use FETCH FIRST <NUMBER> ROWS ONLY when needed

{schema}
Foreign_keys = {foreign_keys}
Primary_keys = {primary_keys}
#### Question: {question}
#### {dialect} SQL QUERY
{query}
```

```
#### {dialect} FIXED SQL QUERY
SELECT
```

A.4 Description of the Mondial schema in DIN and C3 prompts

Listing 17: Representation of the Mondial schema in OpenAI Demostration Prompt of the C3 strategy

```
# continent (name, area)
# country (name, code, capital, province, area, population)
# desert (name, area, coordinates)
# island (name, islands, area, elevation, type, coordinates)
# lake (name, river, area, elevation, depth, height, type,
    coordinates)
# mountain (name, mountains, elevation, type, coordinates)
# river (name, river, lake, sea, length, area, source, mountains,
    sourceelevation, estuary, estuaryelevation)
# sea (name, area, depth)
# borders (country1, country2, length)
# countrylocalname (country, localname)
# countryothername (country, othername)
# countrypops (country, year, population)
# economy (country, gdp, agriculture, service, industry, inflation,
    unemployment)
# encompasses (country, continent, percentage)
# ethnicgroup (country, name, percentage)
# islandin (island, sea, lake, river)
# lakeonisland (lake, island)
# language (country, name, percentage)
# mergeswith (sea1, sea2)
# mountainonisland (mountain, island)
# politics (country, independence, wasdependent, dependent,
    government)
# population (country, population_growth, infant_mortality)
# province (name, country, population, area, capital, capprov)
# religion (country, name, percentage)
# riveronisland (river, island)
# riverthrough (river, lake)
# city (name, country, province, population, latitude, longitude,
    elevation)
# geo_desert (desert, country, province)
# geo_estuary (river, country, province)
# geo_island (island, country, province)
# geo_lake (lake, country, province)
# geo_mountain (mountain, country, province)
# geo_river (river, country, province)
# geo_sea (sea, country, province)
# geo_source (river, country, province)
# provincelocalname (province, country, localname)
```

```
# provinceothername (province, country, othername)
# provpops (province, country, year, population)
# airport (iatacode, name, country, city, province, island, latitude, longitude, elevation, gmtoffset)
# citylocalname (city, country, province, localname)
# cityothername (city, country, province, othername)
# citypops (city, country, province, year, population)
# located (city, province, country, river, lake, sea)
# locatedon (city, province, country, island)
# organization (abbreviation, name, city, country, province, established)
# ismember (country, organization, type)
```

Listing 18: Representation of the Mondial schema in Basic Prompt of the DIN strategy

```
Table continent, columns = [*,name,area]
Table country, columns = [*,name,code,capital,province,area,population]
Table desert, columns = [*,name,area,coordinates]
Table island, columns = [*,name,islands,area,elevation,type,coordinates]
Table lake, columns = [*,name,river,area,elevation,depth,height,type,coordinates]
Table mountain, columns = [*,name,mountains,elevation,type,coordinates]
Table river, columns = [*,name,river,lake,sea,length,area,source,mountains,sourceelevation,estuary,estuaryelevation]
Table sea, columns = [*,name,area,depth]
Table borders, columns = [*,country1,country2,length]
Table countrylocalname, columns = [*,country,localname]
Table countryothername, columns = [*,country,othername]
Table countrypops, columns = [*,country,year,population]
Table economy, columns = [*,country,gdp,agriculture,service,industry,inflation,unemployment]
Table encompasses, columns = [*,country,continent,percentage]
Table ethnicgroup, columns = [*,country,name,percentage]
Table islandin, columns = [*,island,sea,lake,river]
Table lakeonisland, columns = [*,lake,island]
Table language, columns = [*,country,name,percentage]
Table mergeswith, columns = [*,sea1,sea2]
Table mountainonisland, columns = [*,mountain,island]
Table politics, columns = [*,country,independence,wasdependent,dependent,government]
Table population, columns = [*,country,population_growth,infant_mortality]
Table province, columns = [*,name,country,population,area,capital,capprov]
Table religion, columns = [*,country,name,percentage]
Table riveronisland, columns = [*,river,island]
Table riverthrough, columns = [*,river,lake]
```

```
Table city, columns = [*,name,country,province,population,latitude,longitude,elevation]
Table geo_desert, columns = [*,desert,country,province]
Table geo_estuary, columns = [*,river,country,province]
Table geo_island, columns = [*,island,country,province]
Table geo_lake, columns = [*,lake,country,province]
Table geo_mountain, columns = [*,mountain,country,province]
Table geo_river, columns = [*,river,country,province]
Table geo_sea, columns = [*,sea,country,province]
Table geo_source, columns = [*,river,country,province]
Table provincelocalname, columns = [*,province,country,localname]
Table provinceothername, columns = [*,province,country,othername]
Table provpops, columns = [*,province,country,year,population]
Table airport, columns = [*,iatacode,name,country,city,province,island,latitude,longitude,elevation,gmtoffset]
Table citylocalname, columns = [*,city,country,province,localname]
Table cityothername, columns = [*,city,country,province,othername]
Table citypops, columns = [*,city,country,province,year,population]
Table located, columns = [*,city,province,country,river,lake,sea]
Table locatedon, columns = [*,city,province,country,island]
Table organization, columns = [*,abbreviation,name,city,country,province,established]
Table ismember, columns = [*,country,organization,type]
```