



# A Text-to-SQL strategy based on large language models and knowledge graphs for real-world databases<sup>☆</sup>

Eduardo R. Nascimento<sup>a,b</sup>, Caio Viktor S. Avila<sup>c</sup>, Yenier T. Izquierdo<sup>a</sup>,  
Grettel M. García<sup>a</sup>, Lucas Feijó L. Andrade<sup>a</sup>, Matheus O. Silva<sup>b</sup>,  
Michelle S.P. Facina<sup>d</sup>, Melissa Lemos<sup>a,b</sup>, Marco A. Casanova<sup>a,b</sup> <sup>\*</sup>

<sup>a</sup> Tecgraf Institute, PUC-Rio, Rio de Janeiro, 22451-900, RJ, Brazil

<sup>b</sup> Department of Informatics, PUC-Rio, Rio de Janeiro, 22451-900, RJ, Brazil

<sup>c</sup> Department of Computing, UFC, Fortaleza, 60440-900, CE, Brazil

<sup>d</sup> Petrobras, Rio de Janeiro, 20231-030, RJ, Brazil

## ARTICLE INFO

### Keywords:

Text-to-SQL  
Large language models  
Knowledge graphs

## ABSTRACT

The *Text-to-SQL* task is defined as “given a relational database and a natural language sentence that describes a question on the database, generate an SQL query over the database that expresses the question”. Text-to-SQL strategies based on Large Language Models (LLMs) achieve remarkable performance on well-known benchmarks, but their performance is significantly less for real-world databases. Some of the reasons for this performance decrease lie in the mismatch between the end user’s view of the data and the organization and naming conventions of the database schema, and in the differences between the end user’s data semantics and the encoding of such semantics in the database. This article then proposes an LLM-based strategy to compile natural language questions into SQL queries that uses a knowledge graph and incorporates a dynamic few-shot examples technique. The implementation of the strategy leverages a database keyword search tool and specific naming conventions to expose the knowledge graph to an LLM thereby simplifying the text-to-SQL task. The article includes experiments with a real-world, proprietary relational database and the Mondial database to assess the performance of the proposed strategy. The experiments suggest that the strategy achieves an accuracy on challenging relational databases that surpasses state-of-the-art approaches on the same databases.

## 1. Introduction

The *Text-to-SQL* task is defined as “given a relational database  $D$  and a natural language ( $NL$ ) sentence  $Q_N$  that describes a question on  $D$ , generate an SQL query  $Q_{SQL}$  over  $D$  that expresses  $Q_N$ ” [1,2].

Numerous tools have addressed this task with relative success [1–4] over well-known benchmarks, such as Spider – Yale Semantic Parsing and Text-to-SQL Challenge [5] and BIRD – Big Bench for LaRge-scale Database Grounded Text-to-SQL Evaluation [6]. The leaderboards of these benchmarks point to a firm trend: the best text-to-SQL tools are all based on Large Language Models (LLMs) [4].

<sup>☆</sup> This article is part of a Special issue entitled: ‘LLMs and KGs for Semantics-driven Sys Engineering - Woo’ published in Data & Knowledge Engineering.

<sup>\*</sup> Corresponding author at: Department of Informatics, PUC-Rio, Rio de Janeiro, 22451-900, RJ, Brazil.

E-mail address: [casanova@inf.puc-rio.br](mailto:casanova@inf.puc-rio.br) (M.A. Casanova).

<https://doi.org/10.1016/j.datak.2026.102580>

Received 26 March 2025; Received in revised form 21 December 2025; Accepted 12 February 2026

Available online 18 February 2026

0169-023X/© 2026 Published by Elsevier B.V.

Text-to-SQL tools must face several challenges. To begin with, they must be able to process NL questions that require multiple SQL constructs [5]. For example, processing the NL question:

“Which has more open orders, P-X or P-Y?”

requires:

- Recognizing that P-X and P-Y are industrial installations.
- Joining installations and orders.
- Understanding what an open order is.
- Computing the number of open orders for each installation.
- Returning the installation with the largest number of open orders.

Omitting the details, the following SQL query would answer the above NL question:

```
SELECT t.name, COUNT(*) AS number_open_orders
FROM Installation t JOIN Order o ON t.code = o.inst_code
WHERE (t.name = 'P-X' OR t.name = 'P-Y')
AND LOWER(o.status) LIKE LOWER ('%0pen%')
GROUP BY t.code
ORDER BY number_open_orders DESC
FETCH 1
```

This is an example of a challenging NL question that the strategy proposed in this article can compile into a correct SQL query. *Real-world databases* raise a different set of challenges for several reasons, among which:

**Vocabulary mismatch.** The relational schema is often an inappropriate specification of the database from the point of view of the end user — the table and column names are often different from the terms the user adopts to formulate NL questions.

**Vocabulary ambiguity.** Metadata and data are often ambiguous, leading to unwanted results.

**Schema complexity.** The relational schema is often large, in the number of tables, columns per table, and foreign keys — which may lead to queries with many joins, which are difficult to synthesize.

**Data semantics complexity.** The data semantics are often complex; for example, some data values may encode enumerated domains, which implies that the terms the users adopt to formulate their NL questions must be mapped to this internal semantics.

Indeed, the performance of some of the best LLM-based text-to-SQL tools on real-world databases is significantly less than that observed for the Spider and BIRD benchmarks [7,8].

This article then focuses on the *real-world text-to-SQL problem*, which is the version of the text-to-SQL problem applicable to real-world databases. Although the original problem has been investigated for some time, this version is considered far from solved, as argued in [8,9].

The first contribution of the article is a novel LLM-based strategy for the real-world text-to-SQL task that combines several approaches to face the above challenges:

**Vocabulary mismatch.** Addressed by using a knowledge graph (KG) designed to describe the database in terms closer to the end user’s terms.

**Vocabulary ambiguity.** Addressed by heuristics to link the terms used in an NL question to KG metadata and data.

**Schema complexity.** Addressed by defining a view that exposes a fragment of the KG to the LLM in such a way that the view contains all the information the LLM requires to compile the user’s NL question into an SQL query.

**Data semantics complexity.** Addressed by exposing the data semantics to an LLM via examples, that is, pairs  $(Q_N, Q_{SQL})$ , where  $Q_N$  is an NL question and  $Q_{SQL}$  is its SQL translation. A large dataset of examples is pre-computed to capture how the data semantics are encoded in the database, and, at runtime, a small set is dynamically selected among the examples whose NL question is similar to the user’s NL question.

The implementation of the strategy leverages the services of a database keyword search tool, called DANKE [10,11], to manage the knowledge graph and its mapping to the relational schema, to link the terms used in an NL question to the KG metadata and data, and to create the required views.

In more detail, Section 5.2 describes how DANKE’s matching service helps find a set of classes of the KG that suffices to compile an input NL question. Then, Section 5.3 shows how the SQL query compilation process is improved by calling DANKE to synthesize a view  $V$  that captures the required joins to answer the user’s NL question  $Q_N$ , and then calling an LLM to compile  $Q_N$  into an SQL query  $Q_{SQL}$  over  $V$ , which can be remapped to the database schema with the help of the definition of  $V$ .

To create a dataset with examples, the implementation includes a variation of a technique that proved helpful in conveying the data semantics to an LLM [12]. Section 4.2 describes how the technique was adapted to the context of KGs.

The second contribution of the article is a set of experiments with two benchmarks to assess the performance of the proposed strategy (Spider and BIRD, two of the familiar text-to-SQL benchmarks, were not adopted for the reasons explained in Section 2.1). The first benchmark is built upon a proprietary, real-world relational database with a challenging schema, which is in production at an energy company, and a set of 100 NL questions carefully defined to reflect the NL questions users submit and to cover a wide range of SQL constructs. The second benchmark was first introduced in [7] and is based on the openly available Mondial relational database and, again, a set of 100 NL questions.

These new results, combined with results from [7], indicate that the proposed strategy performs, on two challenging benchmarks, significantly better than LangChain SQLQueryChain, SQLCoder,<sup>1</sup> “C3 + ChatGPT + Zero-Shot” [13], and “DIN-SQL + GPT-4” [14].

This article is a much-extended version of [15]. It contains a detailed description of DANKE, provides a careful definition of the knowledge graph variety that DANKE supports, discusses how to pass information from a knowledge graph to an LLM, describes in much more detail the proposed strategy, and includes new experiments with the Mondial relational database.

The article is organized as follows. Section 2 covers related work. Section 3 describes DANKE and the variety of knowledge graphs that DANKE supports. Section 4 discusses how to pass information from a knowledge graph to an LLM. Section 5 details the proposed text-to-SQL strategy. Section 6 presents the experiments with the real-world, proprietary database. Section 7 covers the experiments with the Mondial database. Finally, Section 8 contains the conclusions.

## 2. Related work

### 2.1. Text-to-SQL benchmarks

The Spider — Yale Semantic Parsing and Text-to-SQL Challenge [5] defines 200 datasets, covering 138 different domains, for training and testing text-to-SQL tools.

For each database, Spider lists 20–50 hand-written NL questions and their SQL translations. An NL question  $Q_N$ , with an SQL translation  $Q_{SQL}$ , is classified as easy, medium, hard, and extra-hard, where the difficulty is based on the number of SQL constructs of  $Q_{SQL}$  – GROUP BY, ORDER BY, INTERSECT, nested sub-queries, column selections, and aggregators – so that an NL query whose translation  $Q_{SQL}$  contains more SQL constructs is considered more complex. The sets of NL questions introduced in Sections 6.1.3 and 7.1.3 follow this classification, but do not consider extra-hard NL questions.

Spider proposes three evaluation metrics: *component matching* checks whether the components of the prediction and the ground-truth SQL queries match exactly; *exact matching* measures whether the predicted SQL query as a whole is equivalent to the ground-truth SQL query; *execution accuracy* requires that the predicted SQL query selects a list of gold values and fills them into the correct slots. Section 6.2 describes the metric used in the experiments of this article, which is a variation of execution accuracy.

Most databases in Spider have very small schemas; the largest five data-bases have between 16 and 25 tables, and approximately half have schemas with five tables or fewer. Furthermore, all Spider NL questions are phrased in 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.

Spider has two interesting variations. Spider-Syn [16] is used to test how well text-to-SQL tools handle synonym substitution, and Spider-DK [17] addressed testing how well text-to-SQL tools deal with domain knowledge.

BIRD — Big Bench for LaRge-scale Database Grounded Text-to-SQL Evaluation [6] is a large-scale, cross-domain text-to-SQL benchmark in the English language. The dataset comprises 12,751 text-to-SQL pairs and 95 databases, totaling 33.4 GB across 37 domains. 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, of the 11 databases used for development, the largest one has only 13 tables. Again, all NL questions are phrased in the terms used in the database schemas.

Finally, the `sql-create-context`<sup>2</sup> dataset also addresses the text-to-SQL task, and was built from WikiSQL and Spider. It contains 78,577 examples of NL questions, 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.

Despite the availability of these benchmark datasets for the text-to-SQL task, and inspired by them, Sections 6.1 and 7.1 describe benchmark datasets constructed specifically to test strategies designed for the real-world text-to-SQL task.

The benchmark dataset in Section 6.1 consists of a real-world, proprietary relational database, and a set of 100 test NL questions and their ground-truth SQL translations. The database schema is far more challenging than most database schemas available in Spider or BIRD. The database is populated with real data with a semantics that is sometimes not easily mapped to the semantics of the terms the users adopt (such as “`criticality_level = 5`” encodes “critical orders”), which is a challenge for the text-to-SQL task not captured by unpopulated databases, as in the case of Spider. Finally, the NL questions mimic those posed by real users and cover a wide range of SQL constructs (see Table 1).

The benchmark dataset in Section 7.1 is based on the Mondial database,<sup>3</sup> which features a relational schema much larger than those of most databases in the Spider and BIRD benchmarks. The Mondial database is an open-source, suitable proxy for the

<sup>1</sup> <https://huggingface.co/defog/sqlcoder-34b-alpha>.

<sup>2</sup> <https://huggingface.co/datasets/b-mc2/sql-create-context>.

<sup>3</sup> Available at [https://github.com/dudursn/text\\_to\\_sql\\_chatgpt\\_real\\_world/](https://github.com/dudursn/text_to_sql_chatgpt_real_world/).

proprietary industrial database that motivated this article but could not be made openly available. The benchmark includes 100 NL questions, with balanced difficulty, and their translation to SQL.

Both benchmarks also include the necessary data structures to use DANKE and a synthetic dataset that allows applying the dynamic few-shot examples technique described in Section 4.2.

## 2.2. Text-to-SQL tools

An account of text-to-SQL tools of the pre-LLM era can be found in [3]. Recent text-to-SQL tools are surveyed in [18–21], which include a discussion of prompt engineering and fine-tuning methods. The Text-to-SQL Handbook<sup>4</sup> lists the best-performing text-to-SQL tools on several benchmarks.

The Spider Web site<sup>5</sup> publishes a leaderboard with the best-performing text-to-SQL tools. In April 2024, the top five tools achieved an accuracy that ranged from an impressive 85.3% to 91.2%. Four tools used GPT-4. Two of the tools were not openly documented. The three tools that provided detailed documentation have an elaborate first prompt that tries to select the tables and columns that best match the NL question. Therefore, this first prompt is prone to failure if the database schema results in a vocabulary that is disconnected from the NL question terms. This failure cannot be fixed by even more elaborate prompts that try to match the schema and the NL question vocabularies, but it should be addressed as proposed in this article.

The BIRD Web site<sup>6</sup> also publishes a leaderboard with the best-performing tools. Again in April 2024, out of the top 5 tools, two use GPT-4, one uses CodeS-15B, one CodeS-7B, and one is not documented. The sixth and seventh tools also use GPT-4, appear in the Spider leaderboard, and are well-documented.

The Awesome Text2SQL Web site<sup>7</sup> lists the best-performing text-to-SQL tools on WikiSQL, Spider (Exact Match and Exact Execution) and BIRD (Valid Efficiency Score and Execution Accuracy).

The DB-GPT-Hub<sup>8</sup> is a project exploring the use of LLMs for text-to-SQL translation. It contains data collection, data preprocessing, model selection and building, fine-tuning weights, and evaluations of several LLMs fine-tuned for text-to-SQL.

Nascimento et al. [7] ran SQLCoder, LangChain SQLQueryChain, C3, and DIN+SQL over the benchmarks adopted in this article, obtaining much poorer results than those reported in the leaderboards of Spider and BIRD. The reasons were already pointed out in the introduction. By contrast, Sections 6 and 7 demonstrate that the proposed text-to-SQL strategy achieves very good results on the benchmarks presented in the article. Thus, there is reason to believe that the proposed text-to-SQL strategy would yield good performance on Spider and BIRD, thereby avoiding the need for lengthy experiments.

SQLCoder<sup>9</sup> is a specialized text-to-SQL model, open-sourced under the Apache-2 license. The sqlcoder-34b-alpha model features 34B parameters and was fine-tuned on a base CodeLlama model, on more than 20,000 human-curated questions, classified as in Spider, based on ten different schemas.

The experiments in [7] used the “classic” Langchain SQLQueryChain.<sup>10</sup> A newer LangChain tutorial<sup>11</sup> shows how to build an agent that can answer questions about an SQL database.

“C3 + ChatGPT + Zero-Shot” [13] (or briefly C3) is a prompt-based strategy, originally defined for ChatGPT, that uses only approximately 1000 tokens per query and achieves a better performance than fine-tuning-based methods. C3 has three key components: *Clear Prompting* (CP); *Calibration with Hints* (CH); *Consistent Output* (CO). In April 2024, C3 was the sixth strategy listed in the Spider leaderboard, achieving 82.3% in terms of execution accuracy on the test set. It outperformed state-of-the-art fine-tuning-based approaches in execution accuracy on the test set.

“DIN-SQL + GPT-4” [14] (or briefly DIN) uses only prompting techniques and decomposes the text-to-SQL task into four steps: *schema linking*; *query classification and decomposition*; *SQL generation*; and *self-correction*. When released, DIN was the top-performing tool listed in the Spider leaderboard, achieving 85.3% in terms of execution accuracy.

Despite the impressive results of C3 and DIN on Spider, and of SQLCoder on a specific benchmark, the performance of these tools on the benchmarks used in this article was significantly lower [7]. A similar remark applies to LangChain SQLQueryChain, whose results are shown in Line 1 of Table 3.

Different from the usual text-to-SQL tools, the strategy described in Section 5 requires executing two offline steps before the database interface goes into production. First, it requires preparing the keyword search tool, DANKE, for the database in question, as described in Section 3. Section 4 discusses the benefits and limitations of using DANKE. Second, it requires generating a synthetic dataset for the database, as summarized in Section 4.2. The effects of using a dynamic few-shot examples technique, based on a synthetic dataset, was discussed in detail in [12].

By relying on DANKE and the dynamic few-shot examples technique, the proposed strategy, described in Section 5, surpasses all strategies summarized above on both benchmarks.

<sup>4</sup> [https://github.com/HKUSTDial/NL2SQL\\_Handbook](https://github.com/HKUSTDial/NL2SQL_Handbook).

<sup>5</sup> <https://yale-lily.github.io/spider>.

<sup>6</sup> <https://bird-bench.github.io>.

<sup>7</sup> <https://github.com/eosphoros-ai/Awesome-Text2SQL>.

<sup>8</sup> <https://github.com/eosphoros-ai/DB-GPT-Hub>.

<sup>9</sup> <https://huggingface.co/defog/sqlcoder-34b-alpha>.

<sup>10</sup> [https://api.python.langchain.com/en/latest/chains/langchain.chains.sql\\_database.query.create\\_sql\\_query\\_chain.html](https://api.python.langchain.com/en/latest/chains/langchain.chains.sql_database.query.create_sql_query_chain.html).

<sup>11</sup> <https://docs.langchain.com/oss/python/langchain/sql-agent>.

### 2.3. Retrieval-augmented and dynamic few-shot examples prompting

Retrieval-Augmented Generation (RAG), introduced in [22], is a strategy to incorporate data from external sources. This process ensures that the responses are grounded in retrieved evidence, thereby significantly enhancing the accuracy and relevance of the output. There is an extensive literature on RAG. A recent survey [23] classified RAG strategies into *naive*, *advanced*, and *modular* RAG. Naive RAG follows the traditional process that includes indexing, retrieval, and generation of document “chunks”. Advanced RAG introduces various methods to optimize retrieval. Modular RAG integrates strategies to enhance functional modules, such as incorporating a search module for similarity retrieval and applying a fine-tuning approach in the retriever.

A LangChain tutorial<sup>12</sup> shows how to build a simple Q&A application over an unstructured text data source using RAG.

As for text-to-SQL, recent references include a RAG technique [24] to retrieve table and column descriptions from a metadata store that are related to the NL question, based on a similarity search.

A retrieval-augmented prompting method for an LLM-based text-to-SQL framework is proposed in [25], involving sample-aware prompting and a dynamic revision chain. The method uses two strategies to retrieve questions sharing similar intents with input questions. Firstly, using LLMs, the method simplifies the original questions, unifying the syntax and thereby clarifying the users' intentions. To generate executable and accurate SQL queries without human intervention, the method incorporates a dynamic revision chain, which iteratively adapts fine-grained feedback from the previously generated SQL queries.

A similar method was proposed in [12], which also describes a technique to create synthetic datasets with sets of examples  $(Q_N, Q_{SQL})$  where  $Q_N$  is an NL question and  $Q_{SQL}$  is its SQL translation. This method was incorporated into the proposed strategy, as described in Section 4.2.

## 3. A database keyword query processing tool

DANKE is the keyword search platform currently deployed for the industrial database described in Section 6.1 and used for the experiments. The reader is referred to [10,11] for the details of the platform.

DANKE operates over relational databases and RDF datasets and is designed to compile a keyword query into an SQL or SPARQL query that returns the best data matches. Since this article deals with the text-to-SQL problem, only the relational version of DANKE is considered in what follows.

DANKE's architecture comprises three main components: (1) *Storage Module*; (2) *Preparation Module*; and (3) *Data and Knowledge Extraction Module*.

### 3.1. The storage and the preparation modules

The *Storage Module* houses a centralized relational database, constructed from various data sources. It permits defining a knowledge graph that provides a conceptual description of the database and a mapping from the knowledge graph to the database relational schema.

To some extent, the definition of a knowledge schema follows OWL 2 [26]. However, for simplicity, primary key declarations and other details are omitted in the following discussion.

A *class* is a triple  $c = (r_c, l_c, L_c)$ , where  $r_c \in [1, \infty)$  is the *ranking* of  $c$ ,  $l_c$  is a literal, the *primary label* or *primary name* of  $c$ , and  $L_c$  is a set of literals, the *synonym labels* or *synonym names* of  $c$ .

A *datatype property* is a tuple  $p = (r_p, l_p, L_p, c_p, T_p)$ , where  $r_p \in [1, \infty)$  is the *ranking* of  $p$ ,  $l_p$  is a literal, the *primary label* or *primary name* of  $p$ ,  $L_p$  is a set of literals, the *synonym labels* or *synonym names* of  $p$ ,  $c_p$  is a class, the *domain* of  $p$ , and  $T_p$  is a set of data types. The *range* of  $p$  is the union of the data types in  $T_p$ .

The ranking assigned to each class and datatype property reflects the relative relevance of the element within the knowledge graph. These values are determined according to the contextual characteristics of the dataset and may be obtained from domain experts when no explicit specification is provided. Alternatively, they may be computed using the metrics described in [27].

An *object property* is a tuple  $o = (r_o, l_o, L_o, c_o^1, c_o^2)$ , where  $r_o \in [1, \infty)$  is the *ranking* of  $o$ ,  $l_o$  is a literal, the *primary label* or *primary name* of  $o$ ,  $L_o$  is a set of literals, the *synonym labels* or *synonym names* of  $o$ , and  $c_o^1$  and  $c_o^2$  are classes, the *domain* and *range* of  $o$ . In the current version of DANKE, the names of object properties are not used for matching.

The ranking of an object property represents the weight of the relationship between the two classes. For example, suppose that an object property between  $a$  and  $b$  is assigned a weight of 1.0, an object property between  $b$  and  $c$ , a weight of 1.0, while an object property between  $a$  and  $c$  receives a weight of 2.5. This may indicate that, to relate classes  $a$  and  $c$ , the path traversing class  $b$  is considered more relevant, as it has a lower overall weight.

A *knowledge schema* is a triple  $S^K = (C, P, O)$  such that:

- $C$  is a set of classes
- $P$  is a set of datatype properties whose domains are in  $C$
- $O$  is a set of object properties whose domains and ranges are in  $C$
- The names of the classes, datatype properties, and object properties are unique

<sup>12</sup> <https://docs.langchain.com/oss/python/langchain/rag>.

A knowledge schema  $S^K = (C, P, O)$  induces a *knowledge graph*, defined as a directed multi-graph  $G^K = (V, E)$ , where

- $V = C \cup P$
- $(a, b) \in E$  iff there is an object property  $o \in O$  such that  $a$  and  $b$  are the domain and range of  $o$ , or  $a \in C$  is the domain of the datatype property  $b \in P$ .

The knowledge schema and the knowledge graph will be used interchangeably in what follows, depending on the context.

Intuitively, a knowledge schema corresponds to a much-simplified form of an OWL 2 ontology. On the other hand, a knowledge schema introduces the notions of synonyms and rankings that help uncover and rank matches.

A *mapping*  $\mu$  from a knowledge schema  $S^K$  to a relational schema  $S^R$  is specified in much the same way as RDB-to-RDF mappings [28].

An *indexed* datatype property  $p = (r_p, l_p, L_p, c_p, T_p)$  indicates that each value  $v$  of  $p$  for an entity of class  $c_p$  will participate in the keyword-matching process. These typically are real-world entity names or the possible values of an enumerated type.

The *Storage Module* maintains a *dictionary* with *metadata entries* describing the knowledge schema  $S^K$ , the relational schema  $S^R$ , and the mapping  $\mu$ . Also, for each indexed datatype property  $p = (r_p, l_p, L_p, c_p, T_p)$ , for each value  $v$  of  $p$  for an entity of class  $c_p$ , the dictionary has a *data entry* of the form  $(v, c_p, p)$ . For example, for the database used in the experiments in Section 6.1.1, the dictionary contains a few hundred metadata entries and approximately 1.1 million data entries.

Lastly, the *Preparation Module* has tools for creating the knowledge schema  $S^K$  and the mapping  $\mu$  from  $S^K$  to  $S^R$ . The tools also allow the designer to indicate the indexed datatype properties.

### 3.2. The data and knowledge extraction module

The *Data and Knowledge Extraction Module* has two sub-modules, *Query Compilation* and *Query Processing*.

Given a keyword query, represented by a list of keywords, the *Query Compilation Module* in turn has three sub-modules:

1. *Matching Discovery Module*: matches each keyword in the keyword query with the entries in the dictionary.
2. *Matching Optimization Module*: selects the most relevant matches.
3. *Conceptual Query Synthesis Module*: compiles a *conceptual query* over the knowledge schema from the most relevant matches.

The *Matching Discovery Module* implements the process of finding the dictionary entries that match each keyword  $k$ . The matching of  $k$  is always with a class name or a synonym, a datatype property name or a synonym, or a value. Furthermore, the match of  $k$  with an entry  $d_k$  in the dictionary is always *associated* with a class  $c_k$ . Indeed, if the entry is of the form  $c = (r, l_c, L_c)$ , that is, it describes a class  $c$ , then  $c_k = c$ ; if the entry is of the form  $p = (r, l_p, L_p, c_p, T_p)$ , that is, it describes a datatype property  $p$ , or of the form  $(v, c_p, p)$ , that is, it describes a value  $v$ , then  $c_k = c_p$ . The pair  $(k, d_k)$  is called a *match*. A *nucleous* is a set of matches associated with the same class. A nucleus *covers* the set of keywords in its matches.

To find matches, the current implementation of DANKE first employs an exact-match retrieval approach. If no exact match is found, a secondary retrieval phase is applied using fuzzy matching techniques to approximate the user-provided keyword. For example, in Oracle-based implementations, this fuzzy matching process may leverage the native fuzzy query functionality.<sup>13</sup> If the fuzzy matching process also fails, the user-provided keyword is considered *unmatched*, and a feedback message is generated. The *Matching Optimization Module* implements heuristics that avoid enumerating all possible sets of nuclei to select a minimal set that covers the largest set of keywords, which would be infeasible for a large and ambiguous dataset.

A detailed description of the heuristics can be found in [29]. Briefly, the first heuristic discards matches according to a threshold, the order of the keywords, and the Boolean function present in the keyword query. The second heuristic implements a greedy algorithm that first creates the nucleus from the matches found, and then selects the smallest set of nuclei that covers the largest set of keywords. The last heuristic finds a minimal way of connecting the classes in the nuclei, that is, it finds a Steiner tree of the knowledge graph  $G_S$  whose end nodes are such classes. This is a central point since it guarantees that the final SQL query will not return unconnected data, as explored in detail in [30]. If  $G_S$  is connected, then it is always possible to construct one such Steiner tree; otherwise, one would have to find a Steiner forest to cover all classes associated with the matches.

Using the Steiner tree, the *Conceptual Query Synthesis Module* compiles the keyword query into an abstract query [29] over the knowledge schema that includes restriction clauses representing the matches and join clauses connecting the restriction clauses. The generation of the join clauses uses the Steiner tree edges.

Lastly, the *Query Processing Module* has two sub-modules:

1. *Query Translation Module*: compiles the abstract query into an SQL query over the relational schema.
2. *Query Execution Module*: submits the SQL query for execution, collects the results, and displays them to the user.

Compared to the *Query Compilation Module*, the *Query Processing Module* consists of relatively simple steps and leverages the mapping from the knowledge schema to the relational schema. The reader is referred to [29] for the details.

<sup>13</sup> [https://docs.oracle.com/cd/F26413\\_55/books/DataQUCM/c-Fuzzy-Query-ht202169.html](https://docs.oracle.com/cd/F26413_55/books/DataQUCM/c-Fuzzy-Query-ht202169.html).

### 3.3. API extensions

DANKE's internal API was expanded to support the text-to-SQL strategy described in Section 5. Briefly, it now offers the following services:

- *Matching Discovery Service*: exposes the internal *Matching Discovery Module* interface. It receives a set  $K$  of keywords and returns the set  $K_M$  of pairs  $(k, d_k)$  such that  $k \in K$  and  $d_k$  is the dictionary entry that best matches  $k$ , using the matching optimization heuristic mentioned above. The dictionary entry  $d_k$  will be called the *data associated* with  $k$ . If  $k$  does not match any dictionary entry, then  $d_k$  is set to “noMatch”.
- *View Synthesis Service*: receives a set  $S'$  of classes of the knowledge schema and returns a view  $V$  that best joins all classes in  $S'$ , using the Steiner tree optimization heuristic mentioned above. The view  $V$  is specified in much the same way as an RDB-to-RDF mapping: its target clause is over the knowledge schema, but its definition is over the underlying relational schema.

## 4. Exposing the knowledge graph and data semantics to an LLM

### 4.1. Exposing classes and datatype properties through DANKE views

Let  $D$  be a relational database,  $S_D^R$  be its relational schema, and  $S_D^K$  be its knowledge schema represented in DANKE. Let  $Q_N$  be an NL question.

DANKE synthesizes a view  $V$  that incorporates most of what is required by an LLM to compile  $Q_N$  into an SQL query over  $S_D^R$ :

1. The matchings of terms in  $Q_N$  with class names and datatype property names of  $S_D^K$ .
2. The mappings from classes and datatype properties of  $S_D^K$  to tables and columns in  $S_D^R$ .
3. The joins that connect the tables involved.

The matchings of terms in  $Q_N$  with values of indexed datatype properties are passed to an LLM as examples, as explained in Section 4.2.

The mappings and joins are hidden in the WHERE clause of  $V$ , whereas the matchings with class and datatype property names are captured in the view name and the column names of  $V$ , as follows.

Assume that  $V$  is a view that DANKE creates from a Steiner tree of the knowledge graph  $G_D^K$ . Assume that the end nodes of the Steiner tree are the classes whose names are  $c_1, \dots, c_m$  and that  $p_1^i, \dots, p_{n_i}^i$  are the names of the datatype properties of  $G_D^K$  with domain  $c_i$ , for each  $i \in [1, m]$  and  $j \in [1, n_i]$ . The classes  $c_1, \dots, c_m$  are said to be *covered* by  $V$ .

Then, DANKE constructs  $V$  in such a way that:

- The name of  $V$  is “view\_ $c_1 \dots c_m$ ”, that is, a concatenation of the term “view” with the names of the classes the view covers, separated by “\_”.
- The column names of  $V$  are terms of the form “ $c_i p_j^i$ ”, for each  $i \in [1, m]$  and  $j \in [1, n_i]$ .

Appendix C contains an example of how an NL question is compiled into an SQL query, clarifying how this naming convention helps.

### 4.2. Exposing the data semantics to an LLM

The technique proposed in [12] helps an LLM understand how NL language terms map to SQL restrictions and joins. The technique does so by automatically creating pairs  $(Q_N, Q_{SQL})$ , where  $Q_N$  is an NL question and  $Q_{SQL}$  is its SQL translation.

In the context of this article, DANKE views capture the required joins to process an NL question. Therefore, the technique can be simplified to provide just examples  $(Q_N, Q_{SQL})$  that help an LLM understand how NL language terms map to SQL restrictions:  $Q_N$  provides an example of how a user would express a question involving a restriction on values of a property of a class instance in his/her own terms, and  $Q_{SQL}$  expresses the question in SQL. Furthermore,  $Q_{SQL}$  does not need to use table and column names from the relational schema, but rather class and datatype property names from the knowledge schema, which are more closely aligned with user terms. This simplified requirement – capture only SQL restrictions – leads to a modified technique, used in this article, that generates examples, each of which involves only one class, two datatype properties, one of which is the class identifier, and two values.

In general, the examples are collected in a *synthetic dataset*, and created by repeatedly calling Algorithm 1. Appendix A contains an example illustrating how the algorithm generates a sample pair, including the required prompts. The rest of this section comments on the steps of Algorithm 1.

Let  $D$  be a database with relational schema  $S_D^R$ . Let  $S_D^K$  be a knowledge schema defined for  $D$ , and  $\mu_D$  be the mapping from  $S_D^K$  to  $S_D^R$ .

**SelectClassPropertiesValues** selects a tuple  $P = (c, id, p, t_{id}, t_p, v_{id}, v_p)$ , where  $c$  is a class,  $id$  is an identifier of  $c$ ,  $p$  is a datatype property with domain  $c$ ,  $t_{id}$  and  $t_p$  are the types (ranges) of  $id$  and  $p$ , and  $v_{id}$  and  $v_p$  are values of  $id$  and  $p$ . The selection of  $c$  employs

**Algorithm 1:** Example Generation.

**Data:** the relational database  $D$  with schema  $S_D^R$ , the knowledge schema  $S_D^K$ , the mapping  $\mu_D$  from  $S_D^K$  to  $S_D^R$ , and the database documentation  $D_{doc}$ , if available.

**Result:** a pair  $(Q_N, Q_{SQL})$  where  $Q_N$  is an NL question and  $Q_{SQL}$  is the corresponding SQL query.

```

1 Function CreateExample( $D, S_D^K, \mu_D, D_{doc}$ ):
2    $P \leftarrow$  SelectClassPropertiesValues( $D, S_D^K, \mu_D$ );
3    $Q'_N \leftarrow$  CreateQuestion( $P$ );
4    $Q_{SQL} \leftarrow$  GenerateSQL( $P, Q'_N$ );
5    $Q_N \leftarrow$  ImproveQuestion( $P, Q'_N, D_{doc}$ );
6   return ( $Q_N, Q_{SQL}$ );

```

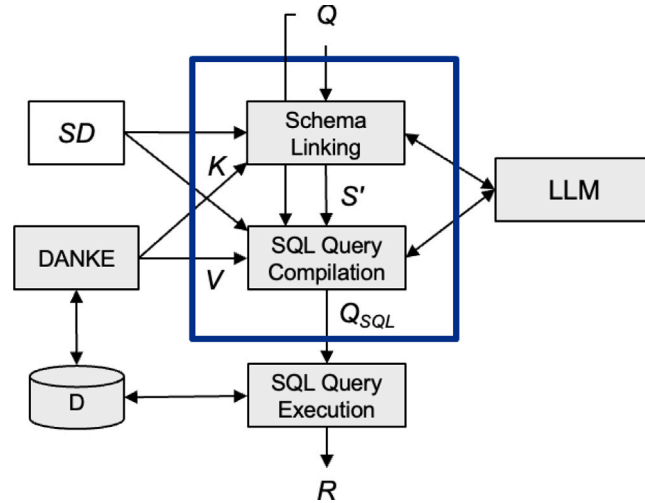


Fig. 1. Proposed strategy.

a weighted random distribution, which reflects the likelihood of each class being chosen by an average user. Note that, since users may choose some classes more often than others, employing a weighted random distribution is justified, but a user's access log must be available to estimate the distribution (as was the case with the experiments in Section 6). The selection of  $p$  follows the same principle. *SelectClassPropertiesValues* selects values from the column/table to which  $\mu_D$  maps  $id$ , and likewise for  $p$ .

*CreateQuestion* prompts an LLM with instructions to create an NL question  $Q'_N$  from  $P = (c, id, p, t_{id}, t_p, v_{id}, v_p)$ . The types of  $id$  and  $p$  allow the LLM to incorporate the relevant details. For example, numeric-type properties can be used to create queries with aggregations.

*GenerateSQL* prompts an LLM with instructions to translate  $Q'_N$  into an SQL query  $Q_{SQL}$  that responds to  $Q'_N$ , based on  $P = (c, id, p, t_{id}, t_p, v_{id}, v_p)$ . As illustrated in Appendix A, the prompt instructs the LLM to generate  $Q_{SQL}$  in such a way that the FROM clause has the single class name  $c$  and the target clause has as column names “ $c_{id}$ ” and “ $c_p$ ”, that is, the datatype property names prefixed by the class name, consistently with the convention of the DANKE views.

Finally, *ImproveQuestion* prompts an LLM to rewrite  $Q'_N$  into an improved NL question  $Q_N$ , closer to natural language, using the descriptions of  $c$ , “ $c_{id}$ ”, and “ $c_p$ ”, along with their synonyms.

## 5. A strategy for the text-to-SQL task

### 5.1. Outline of the proposed strategy

Briefly, the proposed strategy comprises two modules, the *Schema-Linking Module* and *SQL Query Compilation Module*, as typical of text-to-SQL prompt strategies. Fig. 1 summarizes the proposed strategy, leaving the details to the next sections. Appendix B lists the prompts that implement the major steps of the strategy, and Appendix C contains an example.

The two modules run under LangChain. They use a dynamic few-shot examples strategy that retrieves a set of samples from a *synthetic dataset*  $SD$ , indexed with the help of the FAISS similarity search library.<sup>14</sup> The key point is the use of services provided by

<sup>14</sup> <https://ai.meta.com/tools/faiss/>.

DANKE to enhance schema-linking and simplify SQL compilation, as explained in the following sections. In particular, DANKE will generate a single view encapsulating all joins necessary to answer the user's NL question.

The current implementation runs in-house: LangChain, FAISS, DANKE, and Oracle. The experiments used OpenAI models and several other LLMs, as detailed in Sections 6 and 7.

### 5.2. The schema-linking module

Let  $D$  be a relational database,  $S_D^R$  be its relational schema,  $S_D^K$  be its knowledge schema represented in DANKE, and  $SD$  be the synthetic dataset created for  $D$ . Let  $Q_N$  be an NL question.

The *Schema-Linking Module* identifies a minimal set  $S'$  of classes in  $S_D^K$  that is sufficient to answer  $Q_N$ . Note that Schema-Linking returns classes from a knowledge schema and is therefore different from the usual process, which returns tables from a relational schema. [Appendix C](#) provides an example of an NL question, the set of keywords  $K'$ , and the set of classes  $S'$ , mentioned in what follows.

The module has the following major steps (see [Fig. 2](#)):

#### 1. Keyword Extraction and Matching

- (a) Receive as input an NL question  $Q_N$ .
- (b) Call the LLM to extract a set  $K'$  of keywords from  $Q_N$ .
- (c) Call DANKE's *Matching Discovery Service* to match  $K'$  with the dictionary, creating a final set  $K_M = \{(k, d_k) / k \text{ is in } K' \text{ and } k \text{ matches a dictionary entry } d_k\}$ .
- (d) Return  $K_M$ .

#### 2. Dynamic Few-shot Examples Retrieval

- (a) Receive as input an NL question  $Q_N$ .
- (b) Retrieve from the synthetic dataset  $SD$  a set of  $k$  examples whose NL questions are most similar to  $Q_N$ , generating a list

$$L = [(N_1, SQL_1), \dots, (N_k, SQL_k)]$$

- (c) Create a list  $T$

$$T = [(N_1, F_1), \dots, (N_k, F_k)]$$

where  $F_i$  is the class name in the FROM clause of  $SQL_i$ .

- (d) Return  $T$ .

#### 3. Schema Linking

- (a) Receive as input an NL question  $Q_N$ , a set  $K_M$  of keywords and associated data, and a list  $T$  as above.
- (b) Retrieve the set  $S$  of classes in  $S_D^K$ .
- (c) Call the LLM to create  $S'$  prompted by  $Q_N$ ,  $K_M$ ,  $S$ , and  $T$ .
- (d) Return  $S'$  and  $K_M$ .

A few remarks must be made about some of the steps of the *Schema-Linking Module*.

As for Step (1.c), recall from Section 3 that, given a pair  $(k, d_k) \in K_M$ ,  $d_k$  always involves a class name or one of its synonyms. If  $k$  does not match any dictionary entry,  $k$  is ignored. Hence,  $K_M$  contains class names of the knowledge schema that help the schema-linking process. In particular, DANKE may find matches between keywords and database values, which are otherwise inaccessible to an LLM unless the database values are passed in the prompt. For example, DANKE indexes over 1.1M values for the database used in the experiments reported in Section 6, a set well beyond the current prompt limits of most LLMs.

In Step (2.b), note that the synthetic dataset is the same as that prepared for the *SQL Query Compilation Module* — there is no need to create a separate synthetic dataset for the *Schema-Linking Module*. By construction (see Section 4.2), each pair  $(N_i, SQL_i)$  in the synthetic dataset is such that the FROM clause of  $SQL_i$  has just one class name, which Step (2.c) extracts to create list  $T$ .

Albeit Step (1.a) already returns a set of classes that are candidates for schema-linking, DANKE may return more classes than necessary. This is clear if one compares the precision values in Lines 3 and 4 of [Table 2](#). Step (3.c) then calls the LLM to execute the final schema-linking, which opens the possibility for adjusting the set of classes that comes from DANKE.

### 5.3. The SQL query compilation module

The *SQL Query Compilation Module* receives as input the NL question  $Q_N$  and the output of the *Schema-Linking Module* – a set of classes  $S'$  of  $S_D^K$  and a set  $K_M$  of keywords and associated data – and returns an SQL query  $Q_{SQL}$  over  $S_D^R$ . [Appendix C](#) contains an example of a view  $V$ , the intermediate SQL query  $Q'_{SQL}$  over  $V$ , and the final predicted query  $Q_{SQL}$  over  $S_D^R$ , mentioned in what follows.

The module has the following major steps (see [Fig. 3](#)):

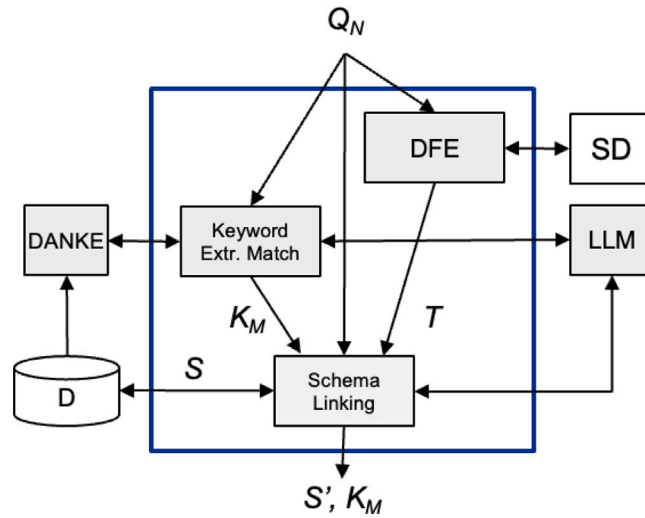


Fig. 2. Schema linking module.

### 1. View Synthesis

- (a) Receive as input a set of classes  $S'$ .
- (b) Call DANKE's *View Synthesis Service* to create a view  $V$  that joins the classes in  $S'$  and whose name and column names follow the convention described in Section 4.1.
- (c) Return  $V$ .

### 2. Question Decomposition

- (a) Receive as input an NL question  $Q_N$ .
- (b) Call an LLM to decompose  $Q_N$  into sub-questions  $Q_1, \dots, Q_m$ .
- (c) Return  $Q_1, \dots, Q_m$ .

### 3. Dynamic Few-shot Examples Retrieval

- (a) Receive as input a list of NL questions  $Q_1, \dots, Q_m$ .
- (b) Let  $p = \lceil k/m \rceil$ . For each  $i \in [1, m]$ , retrieve from the synthetic dataset  $SD$  a set of  $p$  examples whose NL questions are most similar to  $Q_i$  and whose SQL queries are such that the only class name in the FROM clause is in  $S'$ , generating a list

$$L_i = [(N_{i_1}, SQL_{i_1}), \dots, (N_{i_p}, SQL_{i_p})]$$

in decreasing order of similarity of  $N_{i_j}$  to  $Q_i$ .

- (c) Create the final list  $L$ , with  $k$  elements, by intercalating the lists  $L_i$  and retaining the top- $k$  pairs.
- (d) Return  $L$ .

### 4. SQL Compilation

- (a) Receive as input a view  $V$ , a set  $K_M$  of keywords and associated data, an NL question  $Q_N$ , and a list  $L$  as above.
- (b) In each SQL query  $SQL_{i_j}$  in  $L$ , replace the class name in the FROM clause by  $V$ , creating a new list  $L'$ .
- (c) Retrieve from  $D$  a set  $M$  of row samples of  $V$ .
- (d) Call the LLM to compile  $Q_N$  into an intermediate SQL query  $Q'_{SQL}$  over  $V$ , when prompted with  $Q_N$ ,  $V$ ,  $K_M$ ,  $M$  and  $L'$ .
- (e) Remap  $Q'_{SQL}$  to an SQL query  $Q_{SQL}$  over  $S_D^R$ , using the definition of  $V$ .
- (f) Return  $Q_{SQL}$ .

In Step (3.c), note that the final list  $L$  may not contain an example for each sub-question  $Q_1, \dots, Q_m$ , if  $k < m$ . If this is frequently the case, then  $k$  should be increased.

Let  $(N_{i_j}, SQL_{i_j})$  be one of the examples that Step (3.b) retrieves from the synthetic dataset  $SD$ .

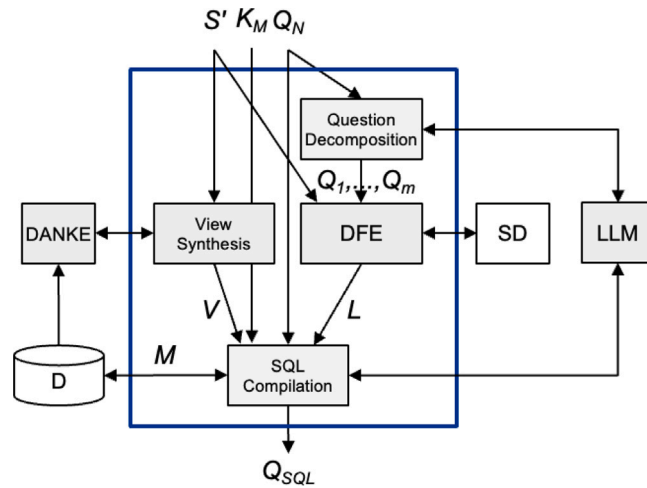


Fig. 3. SQL query compilation module.

By construction of  $SD$  (see Section 4.2), the FROM clause of  $SQL_{i_j}$  has just one class name, say  $c$ , and the column names of  $SQL_{i_j}$  then are “ $c_{id}$ ” and “ $c_p$ ”.

But Step (3.b) guarantees that  $c$  is in  $S'$ , and thus a class that  $V$  covers. Hence, by construction (see Section 4.1), “ $c_{id}$ ” and “ $c_p$ ” are also column names of  $V$ , since  $V$  covers  $c$ .

This guarantees that, when Step (4.b) creates a query  $SQL'_{i_j}$  by replacing  $c$  by  $V$  in the FROM clause, the target clause of  $SQL'_{i_j}$  is well-formed, since “ $c_{id}$ ” and “ $c_p$ ” are column names of  $V$ .

Thus, the resulting query  $SQL'_{i_j}$ , which Step (4.d) passes to the LLM, will have a FROM clause with only  $V$ , will include column names “ $c_{id}$ ” and “ $c_p$ ”, and will have a WHERE clause that exemplifies restrictions involving “ $c_{id}$ ” and “ $c_p$ ”, as desired.

Lastly, Step (4.e) creates the SQL query  $Q_{SQL}$  over the relational schema simply by using the SQL WITH clause to specify a sub-query block with the view definition, followed by  $Q'_{SQL}$  (see Appendix C for an example).

#### 5.4. Discussion

The proposed text-to-SQL strategy strongly depends on how the knowledge schema is constructed, how matching discovery is implemented, and how the synthetic dataset is generated, as discussed below.

##### Schema-linking

A quick way to apply the proposed text-to-SQL strategy to a database would be to start with the relational schema and create a knowledge schema using a direct mapping: tables map to classes, columns to datatype properties, and referential integrity constraints to object properties. The knowledge schema and the mappings would be stored in DANKE’s dictionary.

However, to be effective, any knowledge schema construction process must be complemented by enriching the dictionary as follows. First, one must add class and property synonyms to the dictionary to account for variations in the terms users adopt to refer to the concepts. The synonyms may come from an application log that registers how users access the data. Class and property descriptions should also be added to the dictionary. Second, one must decide which datatype properties should be indexed. The natural candidates are the external names users adopt to refer to entities and the possible values of an enumerated type. As already pointed out in Section 3.2, the matching of the user terms in the NL question and the dictionary entries must be carefully implemented.

In summary, the accuracy of the schema-linking process, and consequently, the accuracy of the complete process, is strongly dependent on the matching of user terms with dictionary entries. But this is true of any schema-linking strategy.

##### SQL Query Compilation

The referential integrity constraints, which will map to object properties, are quite important since they determine the knowledge graph structure that the matching optimization module explores to connect the nuclei using a Steiner tree. It is such Steiner trees that directly drive the synthesis of DANKE’s views (see Section 3.2). If the database does not have referential integrity constraints, the object properties should be defined directly in the dictionary, which is time-consuming but necessary.

The mappings from the knowledge schema to the relational schema must also be present in the dictionary for the view synthesis to succeed.

The generation of the examples ( $Q_N, Q_{SQL}$ ) in the synthetic dataset is strongly dependent on two points (see the example in Appendix A):

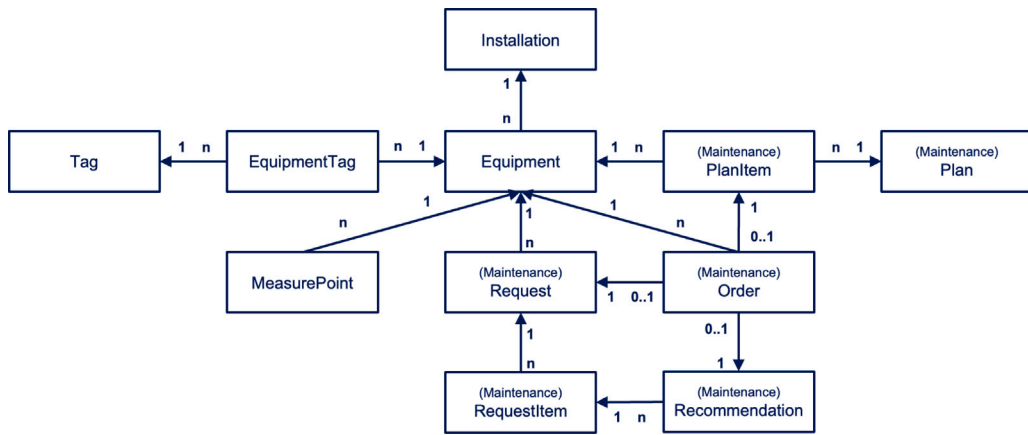


Fig. 4. The knowledge graph of the real-world, proprietary database.

- The synonyms and descriptions of the class and properties sampled, obtained from the dictionary, as well as the data values sampled, which determine in part how the prompt that generates  $Q_N$  is constructed.
- The data types of the datatype properties, which determine in part how the prompt that generates  $Q_{SQL}$  is constructed.

In summary, the accuracy of the SQL query compilation process strongly depends on the quality of the view that DANKE synthesizes, and, to some extent, on the quality of the examples, which in turn depends on the quality of the dictionary. These two points are peculiar to the specific strategy proposed in the article.

## 6. Experiments with a real-world relational database

### 6.1. A benchmark based on a real-world relational database

This section describes a benchmark to help investigate the real-world text-to-SQL task. The benchmark consists of a relational database, a set of 100 test NL questions, and their *ground-truth* SQL translations. It also includes DANKE’s knowledge schema and dictionary, and a synthetic dataset.

#### 6.1.1. The real-world relational database

The selected database is a real-world, proprietary relational database (in Oracle) that stores data related to the integrity management of an energy company’s industrial assets. The relational schema of the adopted database contains 27 relational tables with, in total, 585 columns and 30 foreign keys (some multi-column), where the largest table has 81 columns.

This database will be referred to as the *P database* (for proprietary database) in what follows.

#### 6.1.2. DANKE and synthetic dataset preparation

Recall that the proposed strategy depends on the use of DANKE and a synthetic dataset.

Fig. 4 partially shows the knowledge graph  $G_p^K$  induced by the knowledge schema  $S_p^K$  stored in DANKE for the real-world database. The class and property names in  $S_p^K$  do not follow the original relational schema  $S_p^R$ , but are closely aligned with the terms users adopt. This greatly improved schema-linking, as argued in [31]. By construction, the classes and properties in  $S_p^K$  map to unique tables and columns in  $S_p^R$ , and the datatype property types are those of the underlying relational database.

The knowledge graph is connected, which implies that, given any set of classes  $C$  of  $G_p^K$ , it is always possible to create a Steiner tree of the graph that covers all classes in  $C$ . This implies that Step (1.b) of the *SQL Query Compilation Module* will always succeed in creating the required view.

DANKE’s dictionary has a few hundred metadata entries and about 1.1M data entries for the real-world database, as already mentioned.

The synthetic dataset for the real-world database was prepared as described in Section 4.2 and contains approximately 4750 pairs  $(Q_N, Q_{SQL})$ , where  $Q_N$  is an NL question and  $Q_{SQL}$  is its SQL query translation. The generation of the synthetic dataset was considerably simplified since the database had a structured documentation clearly indicating:

- The mappings from classes and properties to tables and columns.
- The synonyms and descriptions of the classes and properties.
- For each enumerated type  $E$  and each value  $e$  in  $E$ , the user terms that map to  $e$  (for example, for the type “status”, the term “immediate” maps to the value “IMED”).

**Table 1**  
Basic statistics of the sets of queries [7].

	Type	#cols	#joins	#filters	#aggr
Average	Complex	2.00	3.03	2.55	0.39
	Medium	1.44	2.47	1.68	0.29
	Simple	1.30	0.21	0.82	0.15
Maximum	Complex	19	5	4	2
	Medium	9	5	3	2
	Simple	6	2	2	1

### 6.1.3. The set of test questions and their ground-truth SQL translations

The benchmark contains a set of 100 NL questions that consider the terms and questions experts use when requesting information related to the maintenance and integrity processes. The ground-truth SQL queries were manually defined over the relational schema  $S_P^R$  of the real-world database so that the execution of a ground-truth SQL query returns the expected answer to the corresponding NL question.

An NL question is classified into *simple*, *medium*, and *complex*, based on the complexity of its corresponding ground-truth SQL query, as in the Spider benchmark (extra-hard questions were not considered). The set  $L$  contains 33 simple, 33 medium, and 34 complex NL questions, with the basic statistics shown in Table 1.

## 6.2. Evaluation procedure

The experiments used an automated procedure to compare the *predicted* and the *ground-truth* SQL queries, entirely based on column and table values, and not just column and table names. Therefore, a text-to-SQL tool may generate SQL queries over the relational schema or any set of views, and the resulting SQL queries may be compared with the ground-truth SQL queries based on the results returned. The results of the automated procedure were manually checked to eliminate false positives and false negatives.

Indeed, false positives were relatively rare, but false negatives mainly occurred as a consequence of variations in the TARGET clauses of the predicted and the ground-truth SQL queries. For example, the NL question may ask for installations, without specifying whether the installation code or the installation name should be returned. This opens the possibility that the TARGET clause of the predicted SQL query diverges from that of the ground-truth SQL query, without actually incurring an error.

## 6.3. Experiments with schema-linking

### 6.3.1. Experimental setup

The first set of experiments evaluated several alternatives for the schema-linking process.

The experiments adopted the benchmark described in Section 6.1. For each NL question, the ground-truth minimum set  $T$  of tables necessary to answer the NL question is the set of tables in the FROM clause of the ground-truth SQL query. Since Alternatives (3) to (6) below return a set  $S'$  of classes in  $S_P^K$ , the classes are mapped to tables in  $S_P^R$  before  $S'$  is compared to  $T$ , under the assumption that each class in  $S_P^K$  maps to a unique table in  $S_P^R$ . The experiments used GPT-3.5 Turbo and GPT-4, but only the results obtained with GPT-4 were noteworthy. The complete schema-linking strategy was also tested with GPT-4o.

The experiments considered the following alternatives:

1. (*LLM*): A strategy that prompts an LLM with  $Q_N$  and  $S_P^R$  to find the set of tables.
2. (*DFE+LLM*): A strategy that, first, finds a set of examples  $T$  from  $SD$  using  $Q_N$ , and then prompts an LLM with  $Q_N$ ,  $S_P^R$ , and  $T$  to find the set of tables.
3. (*DANKE*): A strategy that, first, uses DANKE to extract a set of keywords  $K$  from  $Q_N$ , and then extracts a set of classes  $S'$  from the information associated with  $K$  in DANKE's dictionary.
4. (*DANKE+LLM*): A strategy that, first, uses DANKE to extract a set of keywords  $K$  from  $Q_N$  and retrieve the information associated with  $K$  from the dictionary, creating a set  $K_M$ , and then prompts an LLM with  $Q_N$ , the classes in  $S_P^K$ , and  $K_M$  to find the set of classes  $S'$ .
5. (*DANKE+DFE+LLM*): A strategy that, first, uses DANKE to extract a set of keywords  $K$  from  $Q_N$  and retrieve the information associated with  $K$  from the dictionary, creating a set  $K_M$ , finds a set of examples  $T$  from  $SD$  using  $Q_N$ , and then prompts an LLM with  $Q_N$ , the classes in  $S_P^K$ ,  $K_M$ , and  $T$  to find the set of classes  $S'$ .
6. (*Complete*): The entire schema-linking process uses an LLM to extract a set of keywords  $K$  from  $Q_N$ , calls DANKE to retrieve the information associated with  $K$  from the dictionary, creating a set  $K_M$ , finds a set of examples  $T$  from  $SD$  using  $Q_N$ , and then prompts an LLM with  $Q_N$ , the classes in  $S_P^K$ ,  $K_M$ , and  $T$  to find the set of classes  $S'$ .

**Table 2**

Results for the schema-linking alternatives (all with GPT-4, except the last line).

#	Method	Precision	Recall	F1-score
1	LLM	0.864	0.886	0.851
2	DFE+LLM	0.940	0.843	0.868
3	DANKE	0.860	0.983	0.900
4	DANKE+LLM	0.930	0.930	0.930
5	DANKE+DFE+LLM	0.993	0.983	0.950
6	Complete — GPT-4	0.993	<b>1.000</b>	<b>0.996</b>
7	Complete — GPT-4o	<b>1.000</b>	0.995	0.995

### 6.3.2. Results

Table 2 presents the precision, recall, and F1-score for the experiments using the schema-linking process. Briefly, the results show that:

1. (*LLM*): Alternative 1 obtained an F1-score of 0.851. It had a performance poorer than Alternative 2.
2. (*LLM+DFE*): Alternative 2 obtained an F1-score of 0.868. The use of DFE improved the results achieved by Alternative 1, but the results were still lower than those of Alternative 3.
3. (*DANKE*): Alternative 3 obtained an F1-score of 0.900, which is better than those of Alternatives 1 and 2 (which do not use DANKE). However, DANKE reached a precision which is lower than that of Alternative 2, that is, DANKE may return more classes resulting from matches than necessary, which has a negative impact on SQL Query Compilation. But DANKE increased the recall w.r.t. Alternatives 1 and 2, that is, DANKE's ability to find matches between keywords and database values, associating the keywords with the properties where the values occur, has a positive effect on recall. In other words, DANKE's matching service can find references that were previously impossible for the LLM to discover since the LLM does not know the database values.
4. (*DANKE+LLM*): Alternative 4 increased the F1-score to 0.930. Enriching the prompt with the keywords extracted by DANKE from  $Q_N$ , and associated information from the dictionary, yielded an improved precision.
5. (*DANKE+DFE+LLM*): Alternative 5 further increased the F1-score to 0.950. Note that, as in Alternative 2, DFE was responsible for improving the F1-score also in this case.
6. (*Complete — GPT-4*): The complete schema-linking process achieved an F1-score of 0.996, the best result. Using the LLM to extract keywords from  $Q_N$  improved the results with respect to extracting keywords using only DANKE. Furthermore, leaving it to the LLM to execute the final schema-linking opened the possibility to adjust the set of classes that comes from DANKE.
7. (*Complete — GPT-4o*): Using GPT-4o resulted in a slight decrease in the F1-score to 0.995, possibly due to the non-deterministic behavior of LLMs.

In general, these results show that DANKE, together with an LLM, performed effectively in the schema-linking process for NL questions. Considering that the complete Schema-Linking Module achieved a recall of 1.0, it returned all classes required to answer each NL question. Thus, the *Schema-Linking Module* does not impact the *SQL Query Compilation Module*, although the extra classes (i.e., precision is not 1.0) may create distractions for the LLM (see Section 6.4).

## 6.4. Experiments with SQL query compilation

### 6.4.1. Experimental setup

The experiments were based on LangChain SQLQueryChain,<sup>15</sup> which automatically extracts metadata from the database, creates a prompt with the metadata and passes it to the LLM. This chain greatly simplifies creating prompts to access databases through views since it passes a view specification as if it were a table specification.

The experiments executed the 100 questions introduced in Section 6.1.3 in eleven alternatives:

1. (*Relational Schema*): SQLQueryChain executed over the relational schema of the benchmark database, as in [7].
2. (*Partially Extended Views*): SQLQueryChain executed over the partially extended views of the benchmark database, as in [31].
3. (*Partially Extended Views and DFE*): SQLQueryChain executed over the partially extended views using only the DFE technique, as in [12].
4. (*Partially Extended Views, DFE, and Question Decomposition*): SQLQueryChain executed over the partially extended views, using Question Decomposition and the DFE technique as in [32].
5. (*The Proposed Text-to-SQL Strategy — GPT-4*): The proposed Text-to-SQL Strategy, using GPT-4-32K.
6. (*The Proposed Text-to-SQL Strategy — GPT-4o*): The proposed Text-to-SQL Strategy, using GPT-4o.
7. (*The Proposed Text-to-SQL Strategy — GPT-5*): The proposed Text-to-SQL Strategy, using GPT-5.
8. (*The Proposed Text-to-SQL Strategy — o3*): The proposed Text-to-SQL Strategy, using o3.

<sup>15</sup> <https://docs.langchain.com>.

**Table 3**

Summary of the results for the real-world relational database benchmark.

#Line	Strategy / Model	#Samples	#Correct Predicted Questions				Accuracy				Total Elapsed Time
			Simple	Medium	Complex	Total	Simple	Medium	Complex	Total	
<i>SQLQueryChain with the Relational Schema</i>											
1	GPT-4-32K	-	22	11	8	41	0.67	0.33	0.23	0.41	23min
<i>SQLQueryChain with the Partially Extended Views</i>											
2	GPT-4-32K	-	30	25	19	74	0.91	0.76	0.56	0.74	N/A
<i>SQLQueryChain with the Partially Extended Views and DFE</i>											
3	GPT-4-32K	Top 8	32	24	23	79	0.97	0.73	0.68	0.79	11min
<i>SQLQueryChain with the Partially Extended Views, DFE, and Question Decomposition</i>											
4	GPT-4-32K	Top 8	32	28	24	84	0.97	0.85	0.71	0.84	31min
<i>The Proposed Text-to-SQL Strategy</i>											
5	GPT-4-32K	Top 8	31	32	30	93	0.94	0.97	0.98	<b>0.93</b>	17min
<i>The Proposed Text-to-SQL Strategy</i>											
6	GPT-4o	Top 8	30	30	30	90	0.91	0.91	0.88	0.90	7min
<i>The Proposed Text-to-SQL Strategy</i>											
7	GPT-5	Top 8	32	30	28	90	0.97	0.91	0.82	0.90	60min
<i>The Proposed Text-to-SQL Strategy</i>											
8	o3	Top 8	31	28	28	87	0.94	0.85	0.82	0.87	40min
<i>The Proposed Text-to-SQL Strategy</i>											
9	LLaMA 3.1-405B-Instruct	Top 8	25	28	28	81	0.76	0.85	0.82	0.81	19min
<i>The Proposed Text-to-SQL Strategy</i>											
10	Mistral Large	Top 8	26	27	24	77	0.79	0.82	0.71	0.77	16min
<i>The Proposed Text-to-SQL Strategy</i>											
11	Claude 3.5-Sonnet	Top 8	24	24	26	74	0.73	0.73	0.76	0.74	17min

9. (*The Proposed Text-to-SQL Strategy — LLaMA 3.1-405B-Instruct*): The proposed Text-to-SQL Strategy, using LLaMA 3.1-405B-Instruct.

10. (*The Proposed Text-to-SQL Strategy — Mistral Large*): The proposed Text-to-SQL Strategy, using Mistral Large.

11. (*The Proposed Text-to-SQL Strategy — Claude 3.5-Sonnet*): The proposed Text-to-SQL Strategy, using Claude 3.5-Sonnet.

Alternatives 1–8 ran on the OpenAI platform, and Alternatives 9–11 on the AWS Bedrock platform.

Also, recall that:

- GPT-4-32K has a context window of 32K tokens.
- GPT-4o has a context window of 128K tokens.
- GPT-5 has a context window of 400K tokens.
- o3 has a context window of 200K tokens.
- Llama 3.1-405B Instruct has a context window of 128K tokens.
- Mistral Large has 123B parameters and a context window of 32K tokens.
- Claude 3.5 Sonnet has 175B parameters and a context window of 200K tokens.

#### 6.4.2. Results

Table 3 summarizes the results for the various alternatives. Column “#Samples” indicates the number of examples the “Dynamic Few-shot Examples” step of the SQL Query Compilation Module retrieves. Columns under “#Correct Predicted Questions” show the number of NL questions per type correctly translated to SQL (recall that there are 33 simple, 33 medium, and 34 complex NL questions, with a total of 100). Columns under “Accuracy” indicate the accuracy results per NL question type and the overall accuracy. The last column shows the total elapsed time to run all 100 NL questions.

The results for Alternatives 1, 2, and 3 were reported in [7,12,31], respectively. They are repeated in Table 3 for comparison with the results of this article.

The results for Alternative 4 show that Question Decomposition produced an improvement in total accuracy from 0.79 to 0.84. This reflects the diversity of examples passed to the LLM when they are retrieved for each sub-question, as already pointed out in [32].

The results for Alternative 5 show that the key contribution of this article, the text-to-SQL strategy described in Section 5, indeed leads to a significant improvement in the total accuracy for the real-world, proprietary database, as well as the accuracy for the medium and complex NL questions.

The results for Alternative 6 indicate a slight decrease in the total accuracy to 0.90 when GPT-4o is adopted, possibly due to the non-deterministic behavior of the models. However, while GPT-4-32K took 17 min to run all 100 questions, GPT-4o took only 7 min.

The results for Alternatives 7 and 8 show that GPT-5 and o3 did not improve accuracy, and yet had very high total elapsed times. Briefly, GPT-5 is designed as a powerful reasoning model (and was ran with high reasoning effort), whereas the o3 model is a high-performance reasoning model designed for complex, precision-critical tasks. GPT-5 demonstrated greater care when returning data to the user. The query results were always ordered, even without being asked for. When applicable, GPT-5 used decreasing

order by date (creation or closing dates), and NULLS LAST (Oracle), moving null values to the end. GPT-5 and o3 tried to consider multiple interpretations for ambiguous NL questions, generating restrictions to cope with synonyms, text variations, and semantically similar columns.

The results for Alternatives 9–11 show a decrease in the total accuracy to 0.81%, 0.77%, and 0.74%, respectively, with a much higher total elapsed time when compared with GPT-4o, but comparable to that of GPT-4-32K. However, recall that Alternatives 9–11 were run on a different platform.

### 6.4.3. Discussion

In Alternatives 1–4, which did not resort to DANKE, the LLM had all the schema-linking burden and had to synthesize all the joins required to process the NL question correctly. By contrast, the strategy outlined in Section 5, which utilizes DANKE, alleviates these burdens. In a few cases, schema linking returned more classes than were required. However, in most of these cases, the SQL Query Compilation process was not compromised; in only one instance, the extra classes and, consequently, the extra columns in the view led the LLM to confuse the choice of columns and synthesize an incorrect WHERE clause.

The results in Table 3 show that the proposed strategy (Line 5) correctly processed four more medium NL questions and six more NL complex questions than the previous best strategy (Line 4). However, these results hide the fact that the proposed strategy processed four complex NL questions that none of the strategies previously tested on the same database and set of questions have correctly handled, including C3 and DIN+SQL, tested in [7].

For example, in one question of the benchmark (Question 29), the previous strategies did not correctly synthesize the required join, whereas the view created in Step 1 of the SQL Query compilation process (see Section 5.3) indeed includes such a join, making it easier for the LLM to compile the required SQL query. In another question (Question 93), all previous strategies failed to remap the installation name “E176” in the user question to the installation name “E-176” stored in the database; the proposed strategy utilized DANKE’s matches to correct this issue.

Similar observations apply to the medium NL questions. For example, the previous strategies did not correctly process the question used in the example of Appendix C, whereas the proposed strategy did, using DANKE’s matches.

An analysis of the NL questions compiled into incorrect SQL queries uncovered that the proposed strategy failed for two basic reasons: (1) the semantics of a term of the NL question was mapped into an incorrect SQL filter; and (2) a term of the NL question was associated with an incorrect column name.

As for the other models – Llama 3.1-405B Instruct, Mistral Large, and Claude 3.5 Sonnet – the most common source of error was the use of the CONTAINS function, which requires the target column to be indexed; the correct filter would have to use LIKE.

## 7. Experiments with the Mondial database

### 7.1. The Mondial benchmark

The Mondial benchmark adopted in this article was first introduced in [7] and is based on the openly available Mondial database, a set of 100 NL questions, and a synthetic dataset.

#### 7.1.1. The Mondial relational database

The Mondial database<sup>16</sup> stores geographic data, as the name suggests. The relational version, adopted in this article, has a total of 47,699 instances and features a schema with 46 tables, a total of 184 columns, and 49 foreign keys (some of which are multi-column). The Mondial schema is sophisticated, but uses table and column names which are familiar terms, such as countries, cities, rivers, etc. A Mondial version with 34 tables is also part of the BIRD benchmark.

The Mondial database will be referred to as the *M database* in what follows.

#### 7.1.2. DANKE and synthetic dataset preparation

The knowledge schema for Mondial in DANKE is quite similar to the original Mondial relational schema, since the latter already has table and column names close to what users adopt, and requires no further comments.

The synthetic dataset for Mondial has approximately 8530 pairs and was obtained using the technique outlined in Section 4.2, adopting GPT-4. The generation of the synthetic dataset was again considerably simplified since Mondial has a straightforward, structured documentation.<sup>17</sup>

#### 7.1.3. The set of test questions and their ground-truth SQL translations

The Mondial benchmark contains a set of 100 NL questions and their ground-truth SQL queries, collected from the Internet or defined manually over the Mondial relational schema. As before, the NL questions are manually classified into *simple*, *medium*, and *complex*, which correspond to the easy, medium, and hard classes of the Spider benchmark. The list of NL questions contains 33 simple, 33 medium, and 34 complex.

Table 4 shows an example of each type of NL question  $Q_N$  and its ground-truth SQL translation  $Q_{SQL}$ . Note that, in the examples,  $Q_{SQL}$  has just a restriction, when  $Q_N$  is simple,  $Q_{SQL}$  has a join and a restriction, when  $Q_N$  is medium, and  $Q_{SQL}$  has two joins and an aggregation, when  $Q_N$  is complex.

<sup>16</sup> <https://relational.fit.cvut.cz/dataset/Mondial>.

<sup>17</sup> <https://www.dbis.informatik.uni-goettingen.de/Mondial/mondial-RS.pdf>.

**Table 4**

A sample of the benchmark NL questions and their ground-truth SQL translations for Mondial.

Type	NL Question	Ground-truth SQL Query	ID
simple	Show the Airports with elevation more than 3000.	SELECT NAME FROM MONDIAL_AIRPORT WHERE ELEVATION > 3000.0	33
medium	What type of government does Iran have?	SELECT p.GOVERNMENT  FROM MONDIAL_COUNTRY c INNER JOIN MONDIAL_POLITICS p ON p.COUNTRY = c.CODE WHERE c.NAME = "Iran"	99
complex	How much area do the countries that are adjacent to (or encompassed by) the Caribbean Sea cover in total?	SELECT SUM(DISTINCT c.area) AS total_area FROM mondial.Country c JOIN mondial.encompasses e ON c.code = e.country JOIN mondial.geo_sea gs ON e.country = gs.country WHERE gs.sea = "Caribbean Sea"	24

**Table 5**

Summary of the results for the Mondial benchmark.

#Line	Alternatives / Model	#Samples	#Correct Predicted Queries				Accuracy			
			Simple	Medium	Complex	Total	Simple	Medium	Complex	Total
<b>Relational Schema</b>										
1	SQLQueryChain with GPT-4-32K	Top 8	27	28	15	70	0.82	0.85	0.44	0.70
<b>Relational Schema</b>										
2	C3 with GPT-4-32K	-	29	25	24	78	0.88	0.76	0.71	0.78
<b>Relational Schema, DFE, and Question Decomposition</b>										
3	SQLQueryChain with GPT-4-32K	Top 8	28	31	27	86	0.85	0.94	0.79	0.86
<b>The Proposed Text-to-SQL Strategy</b>										
4	GPT-4o	Top 8	33	29	31	93	1.00	0.87	0.91	<b>0.93</b>

## 7.2. Experiments

The evaluation procedure remained as described in Section 6.2.

The experiments with the *Schema-Linking Module* revealed that Step 3 — “Dynamic Few-shot Examples Retrieval” confused the LLM and led to sub-optimal results. However, when the *Schema-Linking Module* was run without Step 3, it obtained results similar to those of Alternatives 6 and 7 reported in Table 2.

Table 5 summarizes the results obtained with the proposed SQL Query Compilation Module for the Mondial benchmark. The results for Alternatives 1, 2, and 3 were reported in [32]. Again, they are repeated in Table 3 for comparison with the results of this article. In view of the discussion in Sections 6.4.2 and 6.4.3, the experiments with the Mondial benchmark adopting the proposed strategy used only GPT-4o.

The results in Table 5 are very similar to those reported in Table 3, albeit the real-world, proprietary database is very different from the Mondial database, which reinforces the usefulness of the proposed strategy. Out of coincidence, observe that the complete procedure had an accuracy of 93% in both benchmarks, but with a different mix: for the Mondial benchmark, it obtained slightly better results for the simple and complex queries, but slightly worse results for the medium queries.

## 8. Conclusions

This article proposed a novel LLM-based strategy for the real-world text-to-SQL task whose implementation leverages knowledge graphs and adopts a dynamic few-shot examples technique.

Knowledge graphs are supported by the DANKE platform, which offers two basic services: matching user terms with knowledge graph data and metadata stored in a dictionary, and synthesizing views from the knowledge graph to the underlying relational database. The matching service improves the schema-linking process and the view synthesis service simplifies the SQL query compilation process.

Both the schema-linking and the SQL query compilation processes benefit from examples of text-to-SQL constructed from the knowledge graph, using a technique that is a variant of that introduced in [12].

The article included two sets of experiments to assess the performance of the proposed strategy. The first set of experiments used a benchmark based on a proprietary, real-world database. The results in Section 6.3 showed that the precision and recall of the schema-linking process indeed improved with the help of the matching service that DANKE provides. The discussion in Section 6.4

**Table A.6**  
Sample NL question and the corresponding SQL query.

#	Variable	Value
1	Selected Objects	<ul style="list-style-type: none"> <li>• Pipe: selected Class</li> <li>• id: selected identifier, indexed, with data type VARCHAR(12), and sample value "XYZ"</li> <li>• Material: selected datatype property, non-indexed, with data type VARCHAR(50), and sample value "reinforced steel"</li> </ul>
2	NL question	<i>"Has pipe id='XYZ' material= 'reinforced steel' ?"</i>
3	SQL Query	<pre>SELECT Pipe_id, Pipe_material FROM Pipe WHERE CONTAINS (Pipe_id, '{XYZ}', 1) &gt; 0 AND LOWER(Pipe_material) LIKE LOWER('reinforced steel');</pre>
4	Improved NL question	<i>"Is pipe XYZ made of reinforced steel?"</i>

suggested that creating a view with the help of DANKE improved the SQL query compilation process. In conjunction, these results indicate that the proposed strategy achieved a total accuracy of 93% over a benchmark built upon a relational database with a challenging schema and a set of 100 questions carefully defined to reflect the questions users submit and to cover a wide range of SQL constructs.

The second benchmark was first introduced in [7] and is based on the openly available Mondial relational database, and again a set of 100 NL questions. The results in Section 7.2 indicate that the proposed strategy also achieved a total accuracy of 93% in this second benchmark, albeit with a different mix of correctly processed NL questions.

In summary, the results showed that the proposed strategy performed, on two challenging benchmarks, significantly better than SQLCoder, LangChain SQLQueryChain, C3, and DIN+SQL on the same benchmarks, as previously reported in [7,32], and summarized in part in Lines 1 to 4 of Table 3 and Lines 1 to 3 of Table 5.

As future work, a pressing demand is to address the problem that NL questions are intrinsically ambiguous. The use of DANKE's matching service helps, but it should be complemented with a different approach that incorporates the user in a disambiguation loop.

Finally, it should be stressed that the proposed strategy is generic, but depends on a knowledge graph and a synthetic dataset with text-to-SQL examples, both of which could be costly to construct. The strategy should therefore be considered when it is worth investing in the construction of a natural language interface for a serious database where accuracy is at stake.

#### CRediT authorship contribution statement

**Eduardo R. Nascimento:** Software. **Caio Viktor S. Avila:** Software. **Yenier T. Izquierdo:** Conceptualization. **Grettel M. García:** Software. **Lucas Feijó L. Andrade:** Software. **Matheus O. Silva:** Software. **Michelle S.P. Facina:** Writing – review & editing. **Melissa Lemos:** Writing – review & editing. **Marco A. Casanova:** Conceptualization.

#### Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

#### Acknowledgments

This work was partly funded by FAPERJ under grant E-26/204.322/2024; by CNPq under grant 305587/2021-8; and by Petrobras, Brazil, under research agreement 2022/00032-9 between CENPES and PUC-Rio.

#### Appendix A. Example of the generation of a natural language question and the corresponding SQL query

This appendix illustrates how Algorithm 1 generates a sample pair for the real-world database used in Section 6 (the data has been modified for privacy reasons). The example involves the objects shown in Table A.6.

In general, Algorithm 1 first samples the knowledge graph and the database to select a class, datatype properties, and data values, and then executes three major tasks — generate an NL question, create the corresponding SQL query, and improve the NL question.

The algorithm executes each of these tasks in two stages: the first stage synthesizes a distinct prompt for the task, based on the sampled data; the second stage passes the prompt to guide an LLM in the desired task. This appendix concentrates on the prompt synthesis stages.

### CreateQuestion

The prompt synthesized to generate an NL question contains:

- The class, properties, and data values sampled, as well as the data types of the sampled properties, as shown in Line 1 of [Table A.6](#). Since the LLM is supposed to understand only SQL, classes are referred to as tables, and datatype properties as columns.
- A detailed description of how the NL question should be generated:
  - The question should include a restriction based on the data types. For example, categories can be grouped, numbers can be added, and so on.
  - The question must clearly contain the class and property names to facilitate the next step of creating the SQL query.
- The model should answer with only the NL question, without providing any additional information.

For the sampled class, properties, and values shown in Line 1 of [Table A.6](#), the prompt goes as follows:

```

1 "You must generate a natural language question about the columns of
2 a table in a database related to platform management. Consider
3 the following information about the table, columns, and values:
4
5 **id** VARCHAR(12) IDENTIFIER of table *Pipe*.
6 **material** VARCHAR(50) of table *Pipe*.
7
8 Example values for the column **id** are: ['XYZ']
9 Example values for the column **material** are: ['reinforced steel']
10
11 The column *id* is a primary key. That is, the values are unique.
12
13 With this information, write a question that a user of this database
14 would ask, involving these columns.
15
16 More specifically, the question must refer to one column with
17 a restriction on the values of the other column.
18
19 To construct this restriction, use the examples provided.
20 This restriction will depend on the data type of the columns.
21 For example, for numerical data, you may ask about maximum or minimum
22 values, sums, or request records with values below or above a
23 specific value.
24 If the column is of text type, you may ask about the expressions
25 contained within that text, or request records with an exact
26 specific value, if applicable.
27 If the column is of date type, you may ask for records from a certain
28 year, month, or day, for example, or all records from a specific date
29 onward.
30
31 Do not use more than one restriction to formulate the question.
32 Ask only about one column with some restriction on the other.
33
34 Remember to use the vocabulary of the database in question, necessarily
35 using the table and column names. Imagine you are asking this question
36 to someone who will answer it with an SQL query to be executed in
37 this database.
38
39 There are other tables in this database. Therefore, your question must
40 make it explicit that it refers to this table.
41
42 Respond with only the question and nothing else."

```

The NL question generated is shown in Line 2 of [Table A.6](#).

### GenerateSQL

To automatically generate the SQL query that answers the NL question created in the previous step, this step also runs in two stages and creates a specific prompt with the following information:

- The question that must be answered.
- A CREATE TABLE statement describing the table and columns sampled. The column names are prefixed with the table name to facilitate linking the generated SQL query with a DANKE view, as explained in Section 4.2.
- Some specific instructions on how the SQL query should be generated.

The prompt goes as follows:

```

1 "Based on a natural-language question, you must provide the SQL that
2 answers the question.
3 The SQL you provide will be executed in an Oracle database later.
4 The question you must answer is the following:
5
6 {question}
7
8 To answer this question, consider the description of the following
9 table:
10
11 '''
12 CREATE TABLE Pipe (
13     Pipe_id VARCHAR2 IDENTIFIER
14     Pipe_date NUMBER
15 );
16 '''
17
18 Information about the columns:
19
20 * **Pipe_id**:
21   range_type: DATA_PROPERTY, INDEXED
22   example: ['XYZ']
23   rule for SQL: Use: **CONTAINS**
24 * **Pipe_material**:
25   range_type: DATA_PROPERTY, NON INDEXED
26   example: ['reinforced steel']
27   rule for SQL: Use: **LIKE**
28
29 From this database description, return the SQL that answers the given
30 question.
31
32 Regarding VARCHAR2 or BOOLEAN columns of range types CATEGORICAL,
33 DATA_PROPERTY, or LARGE_STRING, note:
34
35 * Filters using VARCHAR2 columns with INDEXED must **always** be
36 generated using the **CONTAINS**:
37
38 (CONTAINS(<column>, '{<value>}', <number>) > 0)
39 <column>: column name
40 <value>: the value, enclosed in single quotes and braces
41 <number>: a numeric identifier that increases sequentially
42 for each restriction using CONTAINS. Always start at 1.
43
44 Example:
45 1) Pipe_id IDENTIFIER -- use in the restriction:
46   (CONTAINS(Pipe_id, '{VALUE}', 1) > 0)
47
48 * All other text columns that are **not** INDEXED must use
49 **LOWER** and **LIKE** together, always in this format:
50 LOWER(<column>) LIKE LOWER('%<value>%')
51 to search for any string containing <value> in any position and
52 avoid case differences.
53
54 Examples:
55 1) Column material, range type DATA_PROPERTY -- use restriction:
56   LOWER(material) LIKE LOWER('%VALUE%')
57 2) Column description, range type LARGE_STRING -- same restriction.
58
59 * The following cases may exist:
60 1. (Value different from ABC and CEF) --
61   LOWER(col_DESCRIPTION) NOT LIKE LOWER('%ABC%')
62   AND LOWER(col_DESCRIPTION) NOT LIKE LOWER('%CEF%')
63 2. (Value equal to ABC and CEF) --
64   LOWER(col_DESCRIPTION) LIKE LOWER('%ABC%')
65   AND LOWER(col_DESCRIPTION) LIKE LOWER('%CEF%')

```

```

66
67 Never select all columns using *. Instead, specify the column you wish
68 to select.
69
70 Return only the SQL and nothing else."

```

The SQL query generated is shown in Line 3 of [Table A.6](#).

### ImproveQuestion

To rewrite the NL question while maintaining its original intent, this step creates a specific prompt with the following information:

- The NL question that must be rewritten.
- Synonyms and descriptions of the tables and columns, obtained from the database documentation.
- For an enumerated type  $E$  and a value  $e$  in  $E$ , the user terms that map to  $e$  (for example, for the type “status”, the term “immediate” maps to the value “IMED”)
- Some specific instructions.

The prompt goes as follows:

```

1 "Consider the following question made about a database:
2
3 {question}
4
5 Consider the following information:
6
7 Synonyms for the property **id**: "id", "identifier", "name".
8 Synonyms for the property **material**: "material", "components".
9 Synonyms for the class **Pipe**: "pipe", "pipe segment".
10
11 Description of the column *id*: Corresponds to the name of the pipe.
12 Description of the column *material*: A description of the material
13 the pipe is made of.
14
15 Based on the information provided, rewrite the original question from
16 the database vocabulary into the domain vocabulary, using the synonyms
17 and descriptions given.
18
19 Additionally, try to rewrite property values in a more natural way.
20
21 For example, if the question includes dates, transform the format
22 'YYYY-MM-DDT00:00' into something more compatible with natural
23 language, mentioning the year, month, and day.
24
25 If the question includes a term in single or double quotes, remove
26 the quotes and try to replace it with a more natural term, if possible.
27
28 If the question refers to the value of a column, try to use the column
29 description to convert that code into something more natural for
30 the user.
31
32 The rewritten question must not contain expressions such as "table"
33 or "column".
34
35 You may replace the name of the identifier attribute with the name
36 of the table.
37
38 Remember that the intent of the question must remain the same.
39
40 Additionally, the question must have at most 100 characters.
41
42 The question must clearly indicate that it refers to *tables*.
43 You must include this term or its synonyms in the question.
44 Remember: **Do not use expressions like "column", "table", or "record"
45 under any circumstances.**
46
47 Respond only with a list containing one rewritten question(s), and
48 nothing else."

```

The rewritten question is shown in Line 4 of [Table A.6](#).

## Appendix B. Prompts used for schema-linking and SQL query compilation

This appendix lists the prompts used in the major steps of the *Schema-linking* and the *SQL Query Compilation* modules. The prompt that implements Step (3) of the *Schema-linking Module* has the following parameters:

- **schema** — An NL description of the knowledge graph, listing the classes and, for each class, all its associated properties.
- **samples** — A list of example pairs  $(N_i, F_i)$  such that  $N_i$  is similar to the original question and  $F_i$  is a class name (as explained in Section 5.2).
- **synonyms** — A list of class/properties and their respective synonyms.
- **result\_kws** — The matches returned by DANKE's *Matching Discovery Service*, in a simplified format (in the current implementation): keyword, property, class.

```

1 You are a system that helps the user query a database.
2 You know the schema of the database being queried.
3 Your task is to receive a question from the user and return a list of
4 tables relevant to this question.
5
6 Schema:
7 {schema}
8
9 {samples}
10
11 Use the following synonyms to help with this task:
12 {synonyms}
13
14 Here are some data related to the conceptual query retrieved from
15 the keyword search:
16 {result_kws}
17
18 Generate the response as a json list containing only the list of
19 relevant tables to answer the user's original question. For example:
20 human: return the tables "table_1", "table_2" and "table_3"
21 output:
22   ' ' ' json
23   ["table_1", "table_2", "table_3"]
24   ' ' '

```

The prompt that implements Step (4) of the *SQL Query Compilation Module* has the following parameters:

- **tables** — A view generated by DANKE, in DDL format.
- **synonyms** — A list of class/properties and their respective synonyms.
- **result\_kws** — The matches returned by DANKE's *Matching Discovery Service*, in a simplified format: keyword, property, class.
- **samples** — A list of example pairs  $(N_i, SQL_i)$  such that  $N_i$  is similar to the original question and  $SQL_i$  is the corresponding SQL query.
- **input** — The NL question.

```

1 # You are an Oracle SQL expert. Given an input question, first create
2 a syntactically correct Oracle SQL query to run, then look at the
3 results of the query and return the answer to the input question.
4 Unless the user specifies in the question a specific number of
5 examples to obtain, don't query for at 0 most results or any using
6 the FETCH FIRST n ROWS ONLY clause as per Oracle SQL.
7 You can order the results to return the most informative data in
8 the database. Never query for all columns from a table.
9 You must query only the columns that are needed to answer the
10 question. Pay attention to using only the column names you can see
11 in the tables below. Be careful not to query for columns that do
12 not exist. Also, pay attention to which column is in which table.
13 Pay attention to using TRUNC(SYSDATE) function to get the current
14 date, if the question involves "today".
15
16 # Some hints:
17 - Don't use double quotes in column names
18 - Don't use LEFT JOIN, only JOIN
19 - About string columns, pay attention to:
20   - INDEXED columns must be generated with the CONTAINS function:

```

```

21     (CONTAINS (<column>, '<value>', <number>) > 0)
22     <column>: column name
23     <value>: the value that must be between single quotes and
24     braces
25     <number>: It is a number and will be in sequential order
26     for each constraint using the contains.
27     Always start with 1.
28     Example:
29     1) col_Entity_IDENTIFIER VARCHAR2 IDENTIFIER => use
30     the restriction
31     (CONTAINS (col_Entity_IDENTIFIER, '{{VALUE}}', 1) > 0)
32
33     - The remaining string columns that are not INDEXED must use
34     the LOWER and LIKE functions together, of the form
35     LOWER(<column>) LIKE LOWER('%<value>%')
36     to search for any string that has <value> in any position and
37     avoid differences between upper and lower case.
38     Example:
39     1) col_DESCRIPTION VARCHAR2 => use the restriction
40     LOWER(col_DESCRIPTION) LIKE LOWER('%VALUE%')
41
42     - Never select all columns using *. Instead, specify the column you
43     want to be selected.
44
45     - Use the synonym only to help identify the tables that will be used
46     in queries
47
48     ## Only use the following tables:
49     {tables}
50
51     ## Use the following synonyms to help with this task:
52     {synonyms}
53
54     ## Here are some tips related to the conceptual query retrieved from
55     the keyword search:
56     {result_kws}
57
58     ## Here are some examples of question-sql pairs:
59     {samples}
60
61     # Answer the user query.\n{format_instructions}\n
62     If you feel there is insufficient information to generate a query,
63     do it anyway, even if it is incomplete.
64     -----
65     Question: {input}

```

### Appendix C. Example of compiling an NL question into an SQL query

This appendix illustrates the main steps involved in compiling an NL question into an SQL query (modified for privacy). [Table C.7](#) summarizes the variables used in the example.

#### Schema-Linking

Consider the NL question shown in Line 1 of [Table C.7](#). The *Schema-Linking Module* first calls an LLM to extract the three keywords shown in Line 2. Then, it calls DANKE's *Matching Discovery Service*, which returns the matches found in a simplified format (in the current implementation): keyword, property, class. The matches shown in Line 3 are:

- The term “*recommendations*” of the NL question matches the class name Recommendation.
- The term “*Búzinás*” matches the value Búzinás of the datatype property business-unit of class Installation.
- The term “*released*” matches the value released of the datatype property situation of class Recommendation.

Finally, the *Schema-Linking Module* returns the two classes shown in Line 4, Installation and Recommendation, which cover these matches.

#### SQL Query Compilation

The *SQL Query Compilation Module* first calls DANKE's *View Synthesis Module* to synthesize a view that joins the classes the *Schema-Linking Module* returned. The module uses the knowledge graph in [Fig. 4](#) to find the required join. Then, it uses the following mappings between the knowledge schema and the relational schema to synthesize the view in Line 5 (some details of the view definition are omitted for brevity):

- Class Recommendation maps to table Maintenance\_recommendation.

**Table C.7**  
Compilation of an NL question into an SQL query.

#	Variable	Value
1	NL Question	<i>Which recommendations from Búzinas are critical?</i>
2	Keywords	<i>recommendations, Búzinas, critical</i>
3	$K_m$	<i>recommendations, identifier, Recommendation, Búzinas, business-unit, Installation, critical, situation, Recommendation</i>
4	$S'$	Installation, Recommendation
5	View $V$	CREATE VIEW view_Recommendation_Installation AS SELECT r.id AS Recommendation_identifier, r.sys_sit AS Recommendation_situation, p.BU AS Installation_business_unit  ... FROM Maintenance_recommendation r JOIN Installation p ON r.inst_code = p.name
6	Intermediate SQL Query $Q'_{SQL}$	SELECT Recommendation_Identifier FROM view_Recommendation_Installation WHERE Recommendation_situation = 5 AND LOWER(Installation_business_unit) LIKE LOWER('%BUZINAS%')
7	Final Predicted SQL Query $Q_{SQL}$	WITH view_Recommendation_Installation AS (SELECT r.id AS Recommendation_identifier, r.sys_sit AS Recommendation_situation, p.BU AS Installation_business_unit  ... FROM Maintenance_recommendation r JOIN Installation p ON r.inst_code = p.name ) SELECT Recommendation_Identifier FROM view_Recommendation_Installation WHERE Recommendation_situation = 5 AND LOWER(Installation_business_unit) LIKE LOWER('%BUZINAS%')
8	Ground-truth SQL Query	SELECT r.id FROM Maintenance_recommendation r JOIN Installation p ON r.inst_code = p.code WHERE r.sys_sit = 5 AND LOWER(p.BU) LIKE LOWER('%BUZINAS%')

- Class Installation maps to table Installation.
- The datatype property identifier of class Recommendation maps to column id of table Maintenance\_recommendation.
- The datatype property situation of class Recommendation maps to column sys\_sit of table Maintenance\_recommendation.
- The datatype property business\_unit of class Installation maps to column BU of table Installation.

Note that the view definition indeed hides the join between tables. Maintenance\_recommendation and Installation. The *SQL Compilation* step of the *SQL Query Compilation Module* prompts this view to the LLM as if it were a table in DDL format:

```
CREATE TABLE view_Recommendation_Installation
(Recommendation_identifier, Recommendation_situation, ...,
Installation_business_unit, ...)
```

As a result, the *SQL Query Compilation Module* generates the intermediate SQL query in Line 6 of Table C.7, which does not include the join used in the ground-truth SQL query shown in Line 8.

Lastly, the module generates the final predicted SQL query, as shown in Line 7, which is equivalent to the ground-truth SQL query shown in Line 8.

## Data availability

Data will be made available on request.

## References

- [1] G. Katsogiannis-Meimarakis, G. Koutrika, A survey on deep learning approaches for text-to-SQL, *VLDB J.* 32 (4) (2023) 905–936.
- [2] H. Kim, B.-H. So, W.-S. Han, H. Lee, Natural language to SQL: Where are we today? *Proc. VLDB Endow.* 13 (10) (2020) 1737–1750.
- [3] K. Affolter, K. Stockinger, A. Bernstein, A comparative survey of recent natural language interfaces for databases, *VLDB J.* 28 (2019) 793–819.
- [4] L. Shi, Z. Tang, N. Zhang, X. Zhang, Z. Yang, A survey on employing large language models for text-to-SQL tasks, 2024, *ArXiv Preprint*. <https://doi.org/10.48550/arXiv.2407.15186>.
- [5] T. Yu, R. Zhang, K. Yang, M. Yasunaga, D. Wang, Z. Li, J. Ma, I. Li, Q. Yao, S. Roman, Z. Zhang, D. Radev, Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task, in: E. Riloff, D. Chiang, J. Hockenmaier, J. Tsujii (Eds.), *Proc. 2018 Conference on Empirical Methods in Natural Language Processing, Association for Computational Linguistics, Brussels, Belgium, 2018*, pp. 3911–3921.
- [6] J. Li, B. Hui, G. Qu, J. Yang, B. Li, B. Li, B. Wang, B. Qin, R. Geng, N. Huo, X. Zhou, C. Ma, G. Li, K. Chang, F. Huang, R. Cheng, Y. Li, Can LLM already serve as a database interface? a big bench for large-scale database grounded text-to-SQLs, in: *Proceedings of the 37th International Conference on Neural Information Processing Systems, NIPS '23, Curran Associates Inc., Red Hook, NY, USA, 2024*.
- [7] E. Nascimento, G. García, L. Feijó, W. Victorio, Y. Izquierdo, A. Oliveira, G. Coelho, L. M., R. García, L. Leme, M. Casanova, Text-to-SQL meets the real-world, in: *Proceedings of the 26th International Conference on Enterprise Information Systems - Volume 1: ICEIS, INSTICC, SciTePress, Setúbal, Portugal, 2024*, pp. 61–72.
- [8] F. Lei, et al., Spider 2.0: Evaluating language models on real-world enterprise text-to-SQL workflows, 2024, *ArXiv Preprint*. <https://doi.org/10.48550/arXiv.2411.07763>.
- [9] A. Floratou, et al., NL2SQL is a solved problem... Not!, in: *Conference on Innovative Data Systems Research, 2024*.
- [10] Y. Izquierdo, G. García, M. Lemos, A. Novello, B. Novelli, C. Damasceno, L. Leme, M. Casanova, A platform for keyword search and its application for COVID-19 pandemic data, *J. Inf. Data Manag.* 12 (5) (2021) 521–535, URL <https://sol.sbc.org.br/journals/index.php/jidm/article/view/1904>.
- [11] Y. Izquierdo, M. Lemos, C. Oliveira, B. Novelli, G. García, G. Coelho, L. Feijó, B. Coutinho, T. Santana, R. Garcia, M. Casanova, Busca360: A search application in the context of top-side asset integrity management in the oil & gas industry, in: *Anais do XXXIX Simpósio Brasileiro de Bancos de Dados, SBC, Porto Alegre, RS, Brasil, 2024*, pp. 104–116.
- [12] G. Coelho, E.S. Nascimento, Y. Izquierdo, G. García, L. Feijó, M. Lemos, R. Garcia, A. de Oliveira, J. Pinheiro, M. Casanova, Improving the accuracy of text-to-SQL tools based on large language models for real-world relational databases, in: C. Strauss, T. Amagasa, G. Manco, G. Kotsis, A. Tjoa, I. Khalil (Eds.), *Database and Expert Systems Applications, Springer Nature Switzerland, Cham, 2024*, pp. 93–107.
- [13] X. Dong, C. Zhang, Y. Ge, Y. Mao, Y. Gao, L. Chen, J. Lin, D. Lou, C3: Zero-shot text-to-SQL with ChatGPT, 2023, *ArXiv Preprint*. <https://doi.org/10.48550/arXiv.2307.07306>.
- [14] M. Pourreza, D. Rafiei, DIN-SQL: Decomposed in-context learning of text-to-SQL with self-correction, in: *Proceedings of the 37th International Conference on Neural Information Processing Systems, NIPS '23, Curran Associates Inc., Red Hook, NY, USA, 2024*.
- [15] C. Nascimento, Y. Izquierdo, G. García, L. Feijó, M. Facina, L. M., M. Casanova, On the text-to-SQL task supported by database keyword search, in: *(The 27th International Conference on Enterprise Information Systems), Porto, Portugal, 2025, Accepted To*.
- [16] Y. Gan, X. Chen, Q. Huang, M. Purver, J. Woodward, J. Xie, P. Huang, Towards robustness of text-to-SQL models against synonym substitution, in: *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, Association for Computational Linguistics, 2021*, pp. 2505–2515.
- [17] Y. Gan, X. Chen, M. Purver, Exploring underexplored limitations of cross-domain text-to-sql generalization, in: M.-F. Moens, X. Huang, L. Specia, S. Yih (Eds.), *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, Association for Computational Linguistics, 2021*, pp. 8926–8931.
- [18] X. Liu, S. Shen, B. Li, P. Ma, R. Jiang, Y. Zhang, J. Fan, G. Li, N. Tang, Y. Luo, A survey of text-to-SQL in the era of LLMs: Where are we, and where are we going? *IEEE Trans. Knowl. Data Eng.* 37 (10) (2025) 5735–5754.
- [19] Y. Luo, G. Li, J. Fan, C. Chai, N. Tang, Natural language to SQL: State of the art and open problems, *Proc. VLDB Endow.* 18 (12) (2025) 5466–5471.
- [20] L. Shi, Z. Tang, N. Zhang, X. Zhang, Z. Yang, A survey on employing large language models for text-to-SQL tasks, *ACM Comput. Surv.* (2025).
- [21] A. Singh, A. Shetty, A. Ehtesham, S. Kumar, T.T. Khoei, A survey of large language model-based generative AI for text-to-SQL: Benchmarks, applications, use cases, and challenges, in: *2025 IEEE 15th Annual Computing and Communication Workshop and Conference, CCWC, 2025*, pp. 00015–00021.
- [22] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, D. Kiela, Retrieval-augmented generation for knowledge-intensive NLP tasks, in: H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, H. Lin (Eds.), *Advances in Neural Information Processing Systems, vol. 33, Curran Associates, Inc., Red Hook, NY, USA, 2020*, pp. 9459–9474.
- [23] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi, Y. Dai, J. Sun, Q. Guo, M. Wang, H. Wang, Retrieval-augmented generation for large language models: A survey, 2024, *ArXiv Preprint*. <https://doi.org/10.48550/arXiv.2312.10997>.
- [24] S. Panda, B. Gozluklu, Build a robust text-to-SQL solution generating complex queries, self-correcting, and querying diverse data sources, *AWS Mach. Learn. Blog* (2024).
- [25] C. Guo, Z. Tian, J. Tang, S. Li, Z. Wen, K. Wang, T. Wang, Retrieval-augmented GPT-3.5-based text-to-SQL framework with sample-aware prompting and dynamic revision chain, in: *Neural Information Processing, Springer Nature Singapore, Singapore, 2024*, pp. 341–356.
- [26] P. Hitzler, M. Krötzsch, B. Parsia, P. Patel-Schneider, S. Rudolph, OWL 2 web ontology language primer (second edition), 2012, <https://api.semanticscholar.org/CorpusID:57542951>.

- [27] E.S. Menendez, M.A. Casanova, L.A.P. Paes Leme, M. Boughanem, Novel node importance measures to improve keyword search over rdf graphs, in: S. Hartmann, J. Küng, S. Chakravarthy, G. Anderst-Kotsis, A.M. Tjoa, I. Khalil (Eds.), Database and Expert Systems Applications, Springer International Publishing, Cham, 2019, pp. 143–158.
- [28] M. Hert, G. Reif, H. Gall, A comparison of RDB-to-RDF mapping languages, in: Proceedings of the 7th International Conference on Semantic Systems, in: I-Semantics '11, Association for Computing Machinery, New York, NY, USA, 2011, pp. 25–32.
- [29] G. García, A Keyword-based Query Processing Method for Datasets with Schemas (Ph.D. thesis), Thesis presented to the Graduate Program in Informatics, PUC-Rio, 2020.
- [30] G. García, Y. Izquierdo, E. Menendez, F. Dartayre, M. Casanova, RDF keyword-based query technology meets a real-world dataset, in: Proceedings of the 20th International Conference on Extending Database Technology, EDBT, OpenProceedings.org, Venice, Italy, 2017, pp. 656–667.
- [31] E. Nascimento, Y. Izquierdo, G. García, G. Coelho, L. Feijó, M. Lemos, L. Leme, C. M.A., My database user is a large language model, in: Proceedings of the 26th International Conference on Enterprise Information Systems - Volume 1: ICEIS, INSTICC, SciTePress, Setúbal, Portugal, 2024, pp. 800–806.
- [32] A. Oliveira, E. Nascimento, J. Pinheiro, C. Avila, G. Coelho, L. Feijó, Y. Izquierdo, G. García, L. Leme, M. Lemos, M. Casanova, Small, medium, and large language models for text-to-SQL, in: W. Maass, H. Han, H. Yasar, N. Multari (Eds.), Conceptual Modeling, Springer Nature Switzerland, Cham, 2025, pp. 276–294.

**Eduardo Roger Silva Nascimento** is a Researcher at the Tecgraf Institute/PUC-Rio. He holds an M.Sc. in Informatics from PUC-Rio (2024). His Master's dissertation won the First Place in the Database Thesis and Dissertation Contest — CTDBD in 2024. He is currently pursuing a D.Sc. in Informatics at PUC-Rio. His interests include Databases, Artificial Intelligence, Data Science and software development (backend).

**Caio Viktor da Silva Avila** is an analyst at the Justice Court of the State of Ceará, working at the AI Sector. He holds a D.Sc. (2024) in Computer Science from the Federal University of Ceará. His research interests include ChatBots, Large Language Models, Natural Language Processing and Semantic Search.

**Yenier Torres Izquierdo** is a Consultant I - AAT - Technology at the Tecgraf Institute/PUCRio. He obtained a Bachelor's degree in Computer Science from the Universidad de La Habana, Cuba. He obtained an M.Sc. (2017) and a D.Sc. (2021) in Informatics, both from PUC-Rio. He was a "Nota 10 Student" with a FAPERJ Scholarship in 2016 and 2019 for his academic results during his M.Sc. and D.Sc., respectively. His research interests include Databases, Semantic Web, Linked Data, Information Retrieval, and Data Science.

**Grettel Monteagudo García** is a Consultant at the Tecgraf Institute/PUC-Rio. She obtained a Bachelor's degree in Computer Science (2012) from the Universidad de La Habana, Cuba. He obtained an M.Sc. (2016) and a D.Sc. (2020) in Informatics, both from PUC-Rio. She was a "Nota 10 Student" with a FAPERJ Scholarship. Her research interests include ontologies, Semantic Web, keyword search, and graph algorithms.

**Lucas Feijó Lobo de Andrade** graduated in Computer Engineering (2024) at PUC-Rio. His research interests cover databases and machine learning.

**Matheus Oliveira Silva** graduated in Computer Science (2024) at UFRJ and obtained an M.Sc in Informatics from PUC-Rio (2025). His interests include Generative AI and Deep Learning, focusing on Large Language Models and Computer Vision.

**Michelle Soares Pereira Facina** is a Data Scientist at Petrobras. She graduated in Electrical Engineering (2013) and obtained an M.Sc. (2015) also in Electrical Engineering from the Federal University of Juiz de Fora. She obtained a D.Sc. (2020) in Electrical Engineering from the State University of Campinas (UNICAMP). Her research interests cover signal processing and machine learning.

**Melissa Lemos** is an Assistant Professor at the Department of Informatics at PUC-Rio and a Researcher at the Tecgraf Institute/PUC-Rio. She graduated in Computer Engineering (1998) at PUC-Rio. She obtained an M.Sc. (2000) and a D.Sc. (2004) in Informatics, both from PUC-Rio. She has experience in Computer Science, with an emphasis on Databases, working mostly on projects related to data and document search, access, interpretation, and integration.

**Marco Antonio Casanova** is a Researcher at the Tecgraf Institute and collaborates with the Graduate Program in Informatics of the Pontifical Catholic University of Rio de Janeiro – PUC-Rio. He graduated in Electronic Engineering at the Military Institute of Engineering (1974), obtained an M.Sc. in Informatics from PUC-Rio (1976) and an M.Sc. (1978) and a Ph.D. (1980) in Applied Mathematics from Harvard University. He was Graduate Program Coordinator (2005-2007), Director (2007-2011), and Full Professor (2015-2025) of the Department of Informatics of PUC-Rio, and Coordinator of the Central Planning and Evaluation Office of PUC-Rio (2012-2025). His research interests concentrate on database conceptual modeling, construction of database management systems, and applications of Large Language Models. For his contributions, in July 2012, he received the Scientific Merit Award from the Brazilian Computer Society.