Institut für Maschinelle Sprachverarbeitung
Universität Stuttgart
Pfaffenwaldring 5B
D-70569 Stuttgart

Master Thesis

An Analysis of the Domain-specific Applicability of Text-to-SQL Systems on a Linguistic Database

Maria Vittoria Ateri

Studiengang: M.Sc. Computational Linguistics

Erstprüfer: Dr. Michael Roth
Betreuerin: Dr. Kerstin Jung

Beginn der Arbeit: 28.02.2024
Ende der Arbeit: 28.08.2024

Erklärung (Statement of Authorship)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und dabei keine andere als die angegebene Literatur verwendet habe. Alle Zitate und sinngemäßen Entlehnungen sind als solche unter genauer Angabe der Quelle gekennzeichnet. Die eingereichte Arbeit ist weder vollständig noch in wesentlichen Teilen Gegenstand eines anderen Prüfungsverfahrens gewesen. Sie ist weder vollständig noch in Teilen bereits veröffentlicht. Die beigefügte elektronische Version stimmt mit dem Druckexemplar überein.[1]

(Maria Vittoria Ateri)

[1]Non-binding translation for convenience: This thesis is the result of my own independent work, and any material from work of others which is used either verbatim or indirectly in the text is credited to the author including details about the exact source in the text. This work has not been part of any other previous examination, neither completely nor in parts. It has neither completely nor partially been published before. The submitted electronic version is identical to this print version.

Abstract

The applicability and adaptability of text-to-SQL systems trained on reference databases to more complicated ones is an open question. This thesis attempts to provide intuitions on the challenges and limitations when applying benchmark systems to more complicated databases. For this, two exemplary systems, namely the IRNet and SmBop, both trained on the Spider dataset, are applied to the complex linguistic relational database DIRNDL.

The primary aim is to analyze to what extent the systems manage to produce accurate queries and retrieve correct information when the inference is conducted on a database of greater complexity and dimensions compared to the databases contained in the Spider dataset (the main benchmark in the field). Intentionally, no re-training is performed. A comparison between the two systems is also conducted.

In addition to this, the sensitivity to lexical changes and question complexity variation is part of the analysis carried out in this work.

Through a qualitative evaluation, the current work provides insights into which model architecture works better for complex linguistic databases, and the limits of both systems. The main findings are that the SmBop system is superior to the IRNet one, and that SmBop is also more sensitive to lexical changes in the database schema. Nevertheless, neither system shows a satisfactory performance when the goal is the synthesis of more complex queries which are used in real-world research settings.

Contents

1 Introduction and Motivation **9**

2 Preliminaries **13**

2.1 SQL 13

2.2 Relational Databases and RDBMS 15

2.3 Semantic Parsing and Grammar Engineering 17

2.4 The LAF model 19

2.5 Annotations in the DIRNDL 20

2.5.1 The RefLex scheme 21

2.5.2 The TIGER treebank and XML encoding 22

2.5.3 LFG and XLE 23

2.5.4 GToBI pitch annotations 24

3 Related Work **27**

4 Datasets **33**

4.1 Spider 33

4.2 DIRNDL 35

5 Tools **41**

5.1 IRNet 41

5.2 SmBoP 43

6 Methodology and Contributions **47**

7 Input Formats **49**

8 Evaluation Methods **55**

9 Experiments **61**

9.1 Experiment 1 68

9.1.1 Evaluation of Experiment 1 68

9.2 Experiment 2 73

9.2.1 Evaluation of Experiment 2 73

9.3 Experiment 3 78

9.3.1 Evaluation of Experiment 3 78

10 Results **83**

11 Conclusion and Future Directions **87**

A Custom Evaluation Script **101**

B Custom Evaluation Outputs **108**

1 Introduction and Motivation

Structured Query Language (SQL) is the language that enables definition and interaction with relational databases. Whereas the syntax of SQL queries is comparably easy, it requires familiarity of the user with the framework to use it confidently. On the other hand, natural language processing (NLP) has witnessed remarkable progress in recent years in its intersection with database query generation. Text-to-SQL models are indeed designed to bridge the semantic gap between natural language queries and SQL statements, enabling more intuitive interaction with databases and streamlining its usage therefore. In text-to-SQL parsing, the goal is to produce a SQL query that, when executed against a database, answers a given natural language question, as illustrated in Figure 1. This task involves utilizing the database schema, which includes information about tables, columns, and primary-foreign key relations (Elgohary et al., 2021).

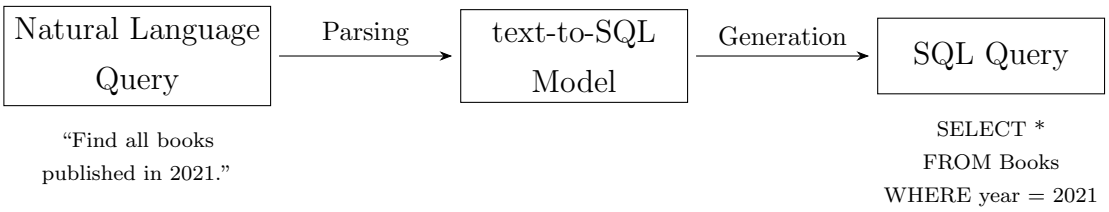


Figure 1: Basic text-to-SQL pipeline with an example query.

However, while most models have demonstrated impressive performance within specific domains and with standard benchmarks, e.g., Spider and other datasets described in Table 1, their robustness and adaptability to diverse and unexplored datasets and domains remain an open research question (Elgohary et al., 2021). Moreover, the capability to generalize across domains, which is crucial for real-world applications, poses still challenges (Julavanich and Aizawa, 2023).

Motivated by these limitations of currently existing text-to-SQL systems, this thesis seeks to analyze the applicability of two different exemplary text-to-SQL sys-

tems to a previously-unseen linguistic-annotated dataset as new benchmark, addressing a critical gap in current research.

The text-to-SQL systems selected for the current work are IRNet (Guo et al., 2019) and SmBop (Rubin and Berant, 2021). Both have been originally trained and tested on Spider, a dataset of schemas, questions and gold parses spanning several databases in different domains (Elgohary et al., 2021). The new database on which inference is endeavored is DIRNDL (**D**iscourse **I**nformation **R**adio **N**ews **D**atabase for **L**inguistic Analysis) (Eckart et al., 2012). Further details on the data and the systems used are provided in the Sections 4 and 5.

The main challenge consists in the fact that the structure of the DIRNDL is not directly deployable as input for the IRNet and for the SmBop systems. Hence, the first research question is:

1. What database design changes are necessary in order to deploy a LAF-format database like the DIRNDL as input for text-to-SQL systems? What are the limits (if any)?

The second point investigated in this setting is the extent to which the parsers of the text-to-SQL systems manage to understand the relations existing in the database. Hence, the second research question is:

2. How sophisticated and accurate are the queries that the systems output? How can it be evaluated at inference time?

Another matter consists in analyzing the effects of changes in the vocabulary of the natural language questions and/or of the tables and columns names. Hence, the third research question is:

3. What semantic/lexical/syntactic changes in the natural language input questions and/or in the tables and columns names change the output query quality? Which system performs better?

Dataset	#Size	#DB	#D	#T/DB	Issues addressed	Sources for data
Spider (Yu et al., 2018)	10,181	200	138	5.1	Domain generalization	College courses, DatabaseAnswers, WikiSQL
WikiSQL (Zhong et al., 2017)	80,654	26,521	-	1	Data size	Wikipedia
Squall (Shi et al., 2020)	11,468	1,679	-	1	Lexicon-level supervision	WikiTableQuestions
KaggleDBQA (Lee et al., 2021)	272	8	8	2.3	Domain generalization	Real web databases
IMDB (Yaghmazadeh et al., 2017)	131	1	1	16	-	Internet Movie Database
Yelp (Yaghmazadeh et al., 2017)	128	1	1	7	-	Yelp website
Advising (Finegan-Dollak et al., 2018)	3,898	1	1	10	-	University of Michigan course information
MIMICSQL (Wang et al., 2020b)	10,000	1	1	5	-	Healthcare domain
SEDE (Hazoom et al., 2021)	12,023	1	1	29	SQL template diversity	Stack Exchange

Table 1: Recent text-to-SQL datasets. #Size, #DB, #D, and #T/DB denote the numbers of question-SQL pairs, databases, domains, and the average number of tables per database, respectively (in the original paper it is stated that #T/DB denotes the average number of tables per domain but is believed to be a typo). A "-" in the #D column signifies an unknown number of domains, while a "-" in the "Issues Addressed" column indicates that the dataset does not specifically address any issues. Datasets above and below the line are categorized as cross-domain and single-domain, respectively (Deng et al., 2022).

The last and broader question that this thesis aims at discovering is whether the IRNet and/or the SmBoP systems can be applied to the DIRNDL and support the extraction of accurate information in a linguistics research setting, in this case based on a study about information status and relative givenness. Hence, the last research question is:

4. Is it possible to apply text-to-SQL systems on a database like the DIRDNL and aid accurate information retrieval in a linguistics research setting? An analysis based on the linguistics research paper "Anarchy in the NP. When new nouns get deaccented and given nouns don't" (Riester and Piontek, 2015).

On the structural level the current thesis is divided in various chapters. In order, Section 2 provides an explanation of the fundamentals required to understand the concepts, topics, and methods discussed throughout the work; Section 3 provides an overview of relevant related work and state of the art in the field of text-to-SQL; Section 4 describes the Spider and the DIRNDL datasets contents and structures; Section 5 describes the two text-to-SQL systems used and compared in this thesis, namely IRNet and SmBoP; Section 6 describes the methodology applied; Section 7 provides fundamental information about the input formats; Section 8 describes the evaluation methods applied; Section 9 describes the three experiments conducted; in Section 10 the results and findings of the research are discussed; last, Section 11 describes some possible future research directions and serves as conclusion of the current thesis.

2 Preliminaries

2.1 SQL

SQL stands for **S**tructured **Q**uery **L**anguage and is the main programming language designed to manage data stored in database systems (Silva et al., 2016). SQL consists of a relatively simple syntax in the form of commands in English language to define, manipulate, and control data. Indeed, it has different components which correspond to these main types of operations. The first component is the data definition language (DDL), which allows to define the structure of the database. The second is the data manipulation language (DML), which allows operations to modify and delete data. The third is the data control language (DCL), which is used by database administrators to manage the permissions and the security of the database (Silva et al., 2016). There is also a fourth one, which is the transaction control language (TCL) and is used to manage transactions in the database. In other words, TCL commands are used to manage the changes made by DML statements (Chapple and Nijim, 2023). Last, SQL is also composed by a data query language (DQL), which allows to fetch the data and is arguably the most relevant SQL component for the current thesis (Sequeda et al., 2011). Figure 2 shows the main SQL commands classified by component.

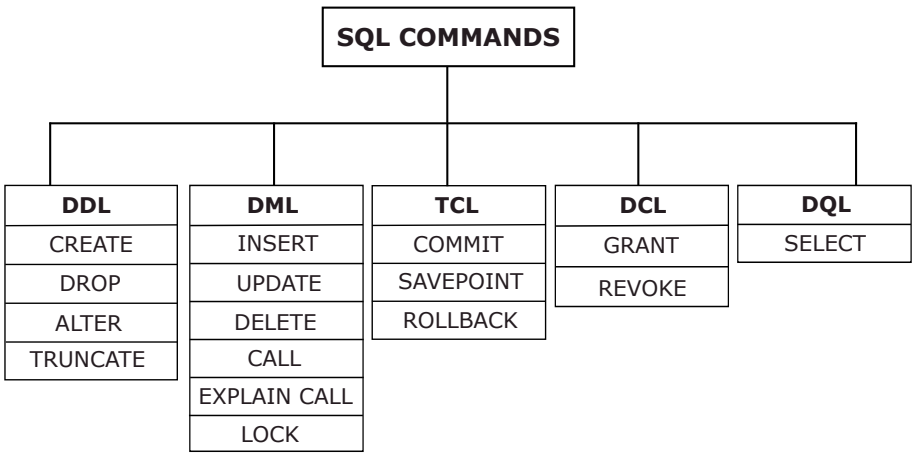


Figure 2: The main SQL commands classified by component.

Importantly, SQL became a standard of the American Standards Institute (ANSI) and of the International Organization for Standardization (ISO, ISO/IEC 9075), and is portable among different database management systems (DBMS) with the aid of some partial adjustments (Silva et al., 2016).

In the current thesis the focus is on DQL. The SELECT statement is the cornerstone of DQL and allows retrieving data from a single or multiple tables in the database. Most often the SELECT statement appears together with a variety of clauses, keywords and operators. A brief description of the main ones follows.

The FROM keyword specifies the table that the user wants to select the data from, and the WHERE clause allows to specify one or more conditions so that the user can filter out the data and get a specific result set. Often tables and columns are assigned aliases, which are temporary names used to make tables and columns names more readable. Aliases are specified with the AS keyword.

The most common operator used in the WHERE clauses is the = (equal) operator, which checks if the result set matches the value that the user is looking for. Other popular operators are != (not equal), > (greater than), >= (greater than or equal), < (less than), <= (less than or equal). In case the user wants to specify multiple criteria, the AND keyword can be used, and the result set contains the data that matches all the specified conditions. Similarly, the OR keyword can be used to specify multiple criteria and obtain data which match either of the conditions. The IN and NOT IN operators allow providing a list expression and return the results that match or do not match that list of values. The IS and IS NOT operators are used to check NULL values and to check or negate a condition or Boolean expression in a WHERE clause. The BETWEEN operator allows selecting values with a given range. The LIKE operator allows wildcard matching, where a wildcard character is a placeholder that represents a single character.

In addition to this, the user often wants to order the results in a specific way based on a particular column. For instance the goal could be to obtain the output in alphabetical order based on the column containing students' surnames in a school database. This can be achieved using the ORDER BY clause. The LIMIT clause

instead allows limiting the number of results that are output to the user. Moreover, in some cases, there might be duplicate entries in a table, and in order to get only the unique values, the user can use the DISTINCT keyword.

Also, with regards to sorting and ordering data, the GROUP BY statement allows using an aggregate function like COUNT, AVG, MIN, MAX, SUM with multiple columns. In addition to this, the HAVING clause allows filtering out the results on the groups formed by the GROUP BY clause.

One last fundamental clause often used in queries is the JOIN clause, which allows combining the data from two or more tables into one result set. Different types of JOIN exist, namely INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL JOIN (Iliev, 2023).

2.2 Relational Databases and RDBMS

A database is a collection of interrelated data, typically stored according to a data model in a way that its contents can be easily accessed, managed, and updated (Taipalus, 2023; Adams and Riede, 2002).

A relational database is a collection of tables that have data organized in a structured way. Each table focuses on a specific category and has columns that define what types of information fit in the table. The relational nature of such databases allows pulling together and observing data from different tables with the aid of SQL commands (Adams and Riede, 2002). Primary keys and foreign keys are two fundamental constraints for relational databases, indicating the entity integrity and referential integrity that databases need to follow (Jiang and Naumann, 2020). Figure 3 shows an example of relational database schema.

A relational database management system (RDBMS) is a program used to create, update, and manage relational databases. Some famous RDBMS are ORACLE, MySQL, PostgreSQL, or SQLite (Adams and Riede, 2002). Two of them, namely PostgreSQL and SQLite are relevant for the current work and hence a short description follows.

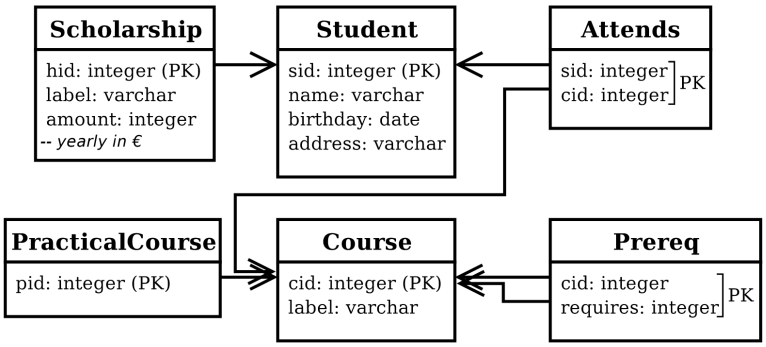


Figure 3: Example of a relational database schema from Champin et al. (2007). PK stands for primary key and the arrows represent foreign keys relations.

Both SQLite and PostgreSQL are fully SQL-compliant databases and support the SQL standard for creating, querying, and managing databases, but differently compared to other database engines, in SQLite a database is stored in a single file. Indeed, while most of the other RDBMS are server-based, SQLite is file-based and does not have a client/server architecture. SQLite encapsulates the entire database into a single file, which includes both the database structure and the actual data within all tables and indexes. This file format is cross-platform, allowing access on any machine regardless of native byte order or word size. Moreover, the fact of having the entire database in one file simplifies creating, copying, and backing up the on-disk database image (Kreibich, 2010).

Nevertheless, although SQLite is highly flexible, it does have certain limitations. For instance, there are practical limits to how much data should be stored in a SQLite database, and since SQLite stores everything in a single file, large databases can strain the operating system or filesystem. Despite most modern filesystems being capable of handling files that are a terabyte or larger, their performance can degrade significantly for random access patterns when the file size reaches multiple gigabytes (Kreibich, 2010).

PostgreSQL instead is a highly robust and sophisticated open-source database. It utilizes an object-relational model to manage data, capable of handling complex routines and rules. Its advanced features include declarative SQL queries, multi-user

support, query optimization, and the use of arrays. Unlike SQLite, PostgreSQL is specifically designed for multi-user access, allowing multiple users to connect and interact with the database concurrently (Worsley and Drake, 2001). However, since PostgreSQL databases depend on servers, moving them between systems can be more complicated and cumbersome than moving SQLite databases.

With regards to data types, unlike some other database systems, SQLite provides a more constrained selection of data types, with its fundamental data type classes being text, floats, integers, NULL, and blobs. These address most everyday use cases, but they may not be sufficient for applications requiring advanced or specialized data types (Kreibich, 2010). On the contrary, one of the main advantages of PostgreSQL is its extensibility, which allows users to define custom data types, functions, and operators. Its extensibility renders it a flexible platform capable of managing diverse data types and intricate use cases (Douglas and Douglas, 2003).

Last, it is relevant to state that nearly every database product, among which SQLite and PostgreSQL, incorporates custom extensions and enhancements to the core SQL syntax. These additions help differentiate each product or provide access to features and control systems not covered by the core SQL standard (Kreibich, 2010). However, these differences also cause slight modifications in the syntax of the SQL queries that are run on one database system or another.

2.3 Semantic Parsing and Grammar Engineering

Semantic parsing is the task of mapping natural language utterances into programs, or, better, it is the task of converting natural language (NL) sentences into their meaning representations (MRs) which a computer program can understand and execute to perform some specific task. Examples of such domain-specific tasks are controlling a robot, or answering questions in a database. These MRs are expressed in a formal meaning representation language (MRL), which is unique and specific to the domain, like some specific command language for robots or some special query language for databases. A machine learning system for semantic parsing is trained on examples of natural language paired with their MRs, and it uses those pairs to

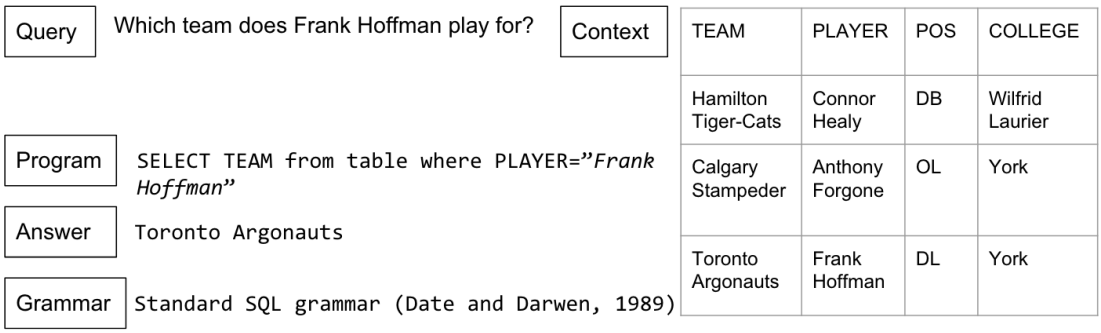


Figure 4: Example of a semantic parsing task with its various components (Kamath and Das, 2019).

learn how to take new NL sentences as input and understand the MR of what they mean. This way, the system can map any novel sentence to the right meaning a computer can understand and use to reach the final goal (Kate, 2008).

Text-to-SQL semantic parsing specifically allows non expert programmers to state questions in natural language, turn the questions into SQL code, and inspect the results of the query execution (Eyal et al., 2023). Figure 4 shows the components of the text-to-SQL semantic parsing task based on the SQL grammar from Date and Darwen (1989). By grammar it is meant a set of rules which have the function to define a set of candidate derivations for every input NL utterance. The type of grammar used determines not only the expressivity of the semantic parser, but also the computational complexity associated with building it (Kamath and Das, 2019). Indeed, grammar engineering is another fundamental concept in the realm of text-to-SQL parsing.

Many text-to-SQL models have been developed with a standard sequence-to-sequence approach, where the natural language input is encoded as a sequence of tokens and the output is decoded as a sequence of SQL tokens. However, some research in the area of semantic parsing has found that using a grammar is often better. With a grammar-based approach, instead of predicting a flat sequence of tokens, the model predicts a hierarchical structure of grammar rules. This way the output is not just a list of tokens, but a structured sequence of rule applications

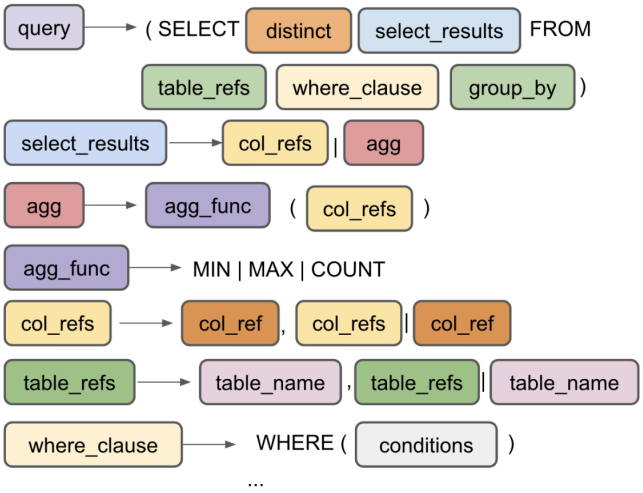


Figure 5: An example of base SQL grammar (Lin et al., 2019).

based on the grammar (Lin et al., 2019).

When designing a grammar for a general programming language like SQL, there are various considerations to be taken. For instance, there is a trade-off between having a compact grammar that minimizes over-generation and having a deeper grammar that can capture more fine-grained phenomena. It is important to highlight that the more compact the grammar is, the more easily learnable it is for the semantic parser (Lin et al., 2019).

Traditional approaches often leverage compiler grammars and Abstract Syntax Trees (ASTs), but these methods can lead to significant over-generation of potential queries and introduce deep, complex derivations that hinder the learning process of semantic parsers (Lin et al., 2019).

SQL grammars can be dataset-specific, but they share a common base. An excerpt of a simplified base grammar is shown in Figure 5 (Lin et al., 2019).

2.4 The LAF model

As the data structures of the micro layer of the B3DB are based on the Linguistic Annotation Framework (LAF), a short description of it follows here.

The Linguistic Annotation Framework (LAF, ISO 24612) was developed by the International Standards Organization (ISO)'s TC37 SC4, the ISO sub-committee on Language Resource Management. The first works on LAF aimed at identifying the fundamental properties and principles for representing linguistic annotations, which led to the design of an abstract data model that has since served as the basis for a standard representation format of morpho-syntactic and syntactic annotations together with various types of semantic annotations (Ide and Suderman, 2014).

The existence of such generic data model allows for easy exchange of linguistic annotations, and it was to this end that the ISO Language Resource Management committee developed principles and methods for creating, coding, processing and managing language resources, such as written corpora, lexical corpora, speech corpora, and dictionary compiling and classification schemes (Ide and Romary, 2003).

One of the most important concepts of LAF is the distinction of annotation structure and annotation content. Indeed, the annotations must attach to the original primary data objects and the semantics of the annotations must be encoded outside of the representation, for instance with the aid of persistent identifiers from data categories stored in a Data Category Registry (Eckart, 2017).

LAF introduces a layered graph structure, where graphs consist of nodes, edges and annotations, and all standard annotation layers for linguistic corpora can be mapped onto this model (Eckart, 2017).

2.5 Annotations in the DIRNDL

The primary data of the DIRNDL corpus consists of an audio file and a manuscript for each of the broadcasts. Moreover, the DIRNDL corpus has been annotated partly automatically and partly manually on three main levels, namely:

- on the pragmatic level with information status annotations based on the RefLex scheme (Riester and Baumann, 2017);
- on the prosodic level with annotations on pitch accents and prosodic boundaries based on the GToBI(S) framework (Mayer, 1995);

Tag	Contextual class
r-given-sit	Referents contained in text-external context (communicative situation)
r-environment	
r-given	Referents mentioned in previous discourse context
r-given-displaced	
r-cataphor	Discourse-new entities that depend on other expressions in the discourse context
r-bridging	
r-bridging-contained	Globally unique entities that are discourse-new and independent of the discourse context
r-unused-unknown	
r-unused-known	
r-new	Non-unique, discourse-new entities
r-expletive	Non-referring expressions
r-idiom	
+generic	Optional features
+predicative	

Figure 6: Annotation tags of the r-level (Riester and Baumann, 2017).

- on the morphosyntactic level with annotations resulting from a parser based on an LFG grammar by Rohrer and Forst and the XLE system (Rohrer and Forst, 2006).

The flexible linking mechanism of the B3DB has been utilized to enable joint querying of the annotation layers of intonation, syntax, and information status (Eckart, 2017).

In the following subsections a short description of the main concepts related to the annotations contained in the DIRNDL is provided.

2.5.1 The RefLex scheme

The RefLex annotation scheme is based on the concept that information status should be analyzed on two levels or dimensions: referential and lexical (or conceptual). This idea originates from theories of information structure, particularly those concerning focus and background (Riester and Baumann, 2017).

Referring expressions (and non-referring terms) are classified according to the scheme described in Figure 6.

Tag	Salience class
l-given-same	active, i.e. salient concepts
l-given-syn	
l-given-super	
l-given-whole	
l-accessible-sub	semi-active, i.e. derivable concepts
l-accessible-part	
l-acessible-stem	
l-new	inactive concepts

Figure 7: Annotation tags of the l-level (Riester and Baumann, 2017).

The lexical level applies to the word domain, more specifically to the content words such as nouns, adjectives, (content) adverbs, and verbs. Pronouns and other functional categories are not annotated at the lexical level. The classification of words follows the scheme described in Figure 7.

2.5.2 The TIGER treebank and XML encoding

The TIGER treebank (Brants et al., 2004) is a newspaper corpus derived from various sections of the Frankfurter Rundschau. Comprising around 900,000 tokens, it includes annotations for lemmas, part-of-speech (POS) tags, morphological features, and syntactic structures.

The dataset is available in multiple formats and encodings, including TIGER-XML, a unique XML-based representation (König et al., 2003) that has also been adopted as the serialization format for the Syntactic Annotation Framework (SynAF), as referenced in ISO 24615-1:2014 and ISO/PRF 24615-2 (Eckart, 2017).

Figure 8 illustrates the terminal and non-terminal nodes in TIGER-XML.

Annotations within the database's micro layer are linked to the nodes and edges of the graph representation. Since annotations are not mandatory for every node or edge, they reference the nodes and edges rather than being inherently attached to them. Furthermore, a node or edge can be annotated with multiple entries. The annotation structure is based on the XML serialization GrAF and has been extended in

```
<terminals>
   <t id="s10_1" word="Frau" pos="N[addr]"/>
   <t id="s10_2" word="Merkel" pos="NAME"/>
   [...]
</terminals>
<nonterminals>
   <nt id="s10_506" cat="NAMEP">
      <edge label="--" idref="s10_1"/>
      <edge label="--" idref="s10_2"/>
   </nt>
   [...]
</nonterminals>
```

Figure 8: Terminal and non-terminal nodes in TIGER-XML (Eckart, 2017).

accordance with Kountz et al. (2008). Regarding feature structures, GrAF complies with ISO 24610-1:2006. Figure 9 provides an example of an annotation encoded in GrAF.

```
<a label="NN" ref="node01" as="TIGER">
  <fs>
    <f name="pos" value="NN" />
    <f name="morph">
      <fs>
        <f name="gender" value="Neut" />
        <f name="case" value="Gen" />
        <f name="number" value="Sg" />
      </fs>
    </f>
  </fs>
</a>
```

Figure 9: Morphosyntactic annotation of a common noun (part-of-speech NN), based on the annotation scheme of the TIGER corpus, and encoded according to GrAF (Eckart, 2017).

2.5.3 LFG and XLE

The term "LFG parser" denotes a system that combines Lexical Functional Grammar (Kaplan and Bresnan, 1982) with a parser. In this context, the Xerox Linguistic Environment (XLE) (Crouch et al., 2011) serves as both the parsing engine and the

development platform (Langer, 2004; Eckart, 2017). An LFG parser generates two primary types of output: the c-structure (constituent structure) and the f-structure (functional structure). For the purposes of this study, only the c-structure, illustrated in Figure 10, is pertinent. In this context, the LFG parser utilized in this work is classified as a rule-based constituency parser. The grammar employed has been extensively tested and refined using the TIGER treebank (Rohrer and Forst, 2006).

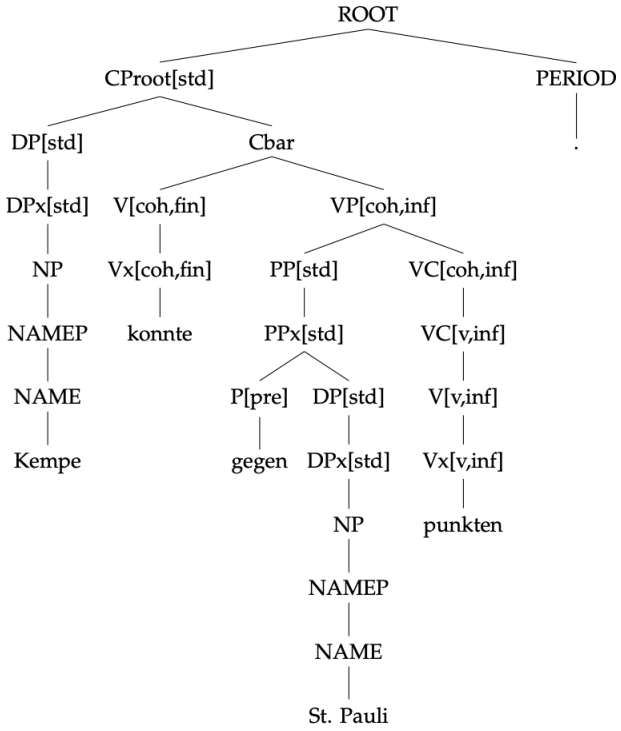


Figure 10: Analysis of example *Kempe konnte gegen St. Pauli punkten.* by the LFG parser (Eckart, 2017).

2.5.4 GToBI pitch annotations

The speech melodies of individual utterances can be described as variations of specific pitch contours. These contours are represented either as configurations, such as falls and rises, or as sequences of discrete tonal units, such as high and low tones (Peters, 2018).

The DIRNDL broadcasts have been manually annotated for pitch accents and prosodic boundaries following GToBI(S), (Mayer, 1995), which is a system of conventions designed for labeling the phonological aspects of German intonation. GToBI closely aligns with the English ToBI system (E-ToBI), which is based on autosegmental-metrical phonology.

GToBI uses a set of high (H) and low (L) tones that are associated with prominent syllables (pitch accents) and the edges of phrases (boundary tones).

The use of different pitch accents can indicate the informational status of a word or phrase, such as whether it is new, given, or accessible information. Pitch accents are marked with a star '*' following the tone, e.g. H*.

The tonal inventory of GToBI includes two monotonal pitch accents (H* and L*) and four bitonal pitch accents (L+H*, L*+H, H+L*, H+!H*). It also features edge tones associated with minor (intermediate) phrases (L- or H-) and major (intonation) phrases (L% or H%) (Grice et al., 2000).

3 Related Work

Text-to-SQL has caught the attention of various research communities, as it can bring huge benefits by allowing building natural language interfaces to database systems. However, this entails a variety of challenges, which numerous researchers have been addressing with a wide range of methodological approaches. In this regard, Deng et al. (2022) provide an overview of the main challenges, current datasets, methods, and potential directions for further research in this field. Essentially, the challenges in text-to-SQL break down to three aspects (Deng et al., 2022):

1. extracting the meaning of natural utterances, which is not trivial given the ambiguous nature of human language (*encoding*);
2. transforming the extracted meaning into another expression which is equivalent to the natural language (NL) meaning, bridging the gap between the two forms of representation (*translating*);
3. generating the corresponding correct SQL queries (*decoding*).

Furthermore, a variety of more specific issues can be mapped to these three main aspects. For instance, multi-turn text-to-SQL systems may need to consider the context of the conversation or the domain-specific knowledge to accurately translate natural language into SQL queries Liu et al. (2022). Significant efforts have been dedicated to developing models that allow the incorporation of user feedback. To cite a few, Zhang et al. (2019) made use of the interaction history by editing the previous predicted query to improve the generation quality, based on the fact that adjacent natural language questions are often linguistically dependent and their corresponding SQL queries tend to overlap. Elgohary et al. (2020) investigated an interactive scenario where humans can interact with the system by providing free-form natural language feedback to correct the system when it generates an inaccurate interpretation of an initial utterance. Elgohary et al. (2021) later presented NL-EDIT, a model designed to interpret natural language feedback within the context of interaction. This model generates a series of edits that can be applied to the

initial SQL parse, correcting any errors present. Nevertheless, although a range of important studies has been focusing on improving multi-turn settings, these are out of the scope of this thesis, since from past work on the DIRNDL database a number of NL queries and respective SQL queries are available, but no multi-turn data.

Another problem lies in how to handle the variations in language and query structures. Indeed, natural language expressions can vary greatly in terms of syntax, grammar, and vocabulary. Similarly, SQL queries can have different structures and writing conventions. Text-to-SQL models need to handle these variations to ensure robustness and generalizability. Xu et al. (2017) addressed specifically the issue of *order matters* in SQL. The *order matters* issue consists in the fact that the same SQL query may have multiple equivalent serializations due to commutativity and associativity, but the de-facto approach of employing a sequence-to-sequence model is sensitive to the choice of one of them. Xu et al. (2017) proposed SQLNet, an approach that avoids the sequence-to-sequence structure when the order does not matter, and instead employs a sketch-based approach. Their approach is an alternative to the one developed by Zhong et al. (2017), namely Seq2SQL, which uses a sequence-to-sequence structure with policy-based reinforcement learning.

One last major problem in text-to-SQL tasks is the fact that developing accurate and effective text-to-SQL models requires access to large-scale, high-quality training data and annotated resources. However, such resources are often limited, especially for complex or specialized domains. Indeed, Kuznia (2023) claimed that human annotation is the bottleneck for text-to-SQL datasets and therefore provided the Generative Pretrained Transformer 3 model (GPT-3) with particular instructions to build a rigorous text-to-SQL dataset, and showed that the created pairs, consisting of a question in natural language and the corresponding SQL code, have excellent quality and diversity, and when utilized as training data, they can enhance the accuracy of SQL generation models. Also Zhao et al. (2022) pointed out that in order to be applicable in real scenarios, real parsers must be able to generalize to new domains since collecting domain specific labeled data is often too expensive. The main challenge in new domains according to the authors consists in two types of operations, namely:

- column matching: the task of mapping natural language phrases to the most relevant columns. For instance, if a user types *"What was Armani's income in 2020?"*, the system might have to map the word *"income"* to the *"Wages"* column in the corresponding table. This can be challenging because some mappings may be implicit or may require domain knowledge;
- column operations: the task of mapping natural language phrases to composite expressions over table columns. For instance, if a user types *"What was Armani's income in 2020?"*, the system might have to map the word *"income"* to the table columns *"Salary"* + *"Stock"*.

The authors claim that especially the challenge of column operations remains unexplored due to the lack of evaluation benchmarks, and thus they propose a synthetic dataset and a new train/test repartitioning of the Squall dataset (Shi et al., 2020) as new benchmarks to quantify domain generalization over column operations. Their results indicate that existing state-of-the-art parsers struggle in these benchmarks. Therefore, they propose to incorporate prior domain knowledge by preprocessing table schemas. They design a method that consists of two components: schema expansion and schema pruning. This method can be easily applied to multiple existing base parsers, and they show that it significantly outperforms baseline parsers on this domain generalization problem, boosting the underlying parsers' overall performance by up to 13.8% relative accuracy gain (5.1% absolute) on the new Squall data split.

The need of better benchmarks to evaluate NL-to-SQL systems (where NL possibly denotes other modalities in addition to text) was also highlighted by Zhang et al. (2024), who claim that the high accuracy reached by some systems is partly justified by the fact that such systems are evaluated on the Spider benchmark, which mainly contains simple databases with few tables, columns, and entries, and thus does not reflect a realistic setting. In fact, complex real-world databases with domain-specific content have little to no training data available in the form of NL/SQL-pairs leading to poor performance of existing NL-to-SQL systems. Hence, Zhang et al. (2024) introduced ScienceBenchmark, a new NL-to-SQL benchmark for three real-

world, highly domain-specific databases. More specifically, they used a policy making database, an astrophysics database, and a cancer research database. For this new benchmark, SQL experts and domain experts created high-quality NL/SQL-pairs for each domain. They then extended the small amount of human-generated data with synthetic data generated using GPT-3. They show that their benchmark is highly challenging, as the top performing systems on Spider achieve a very low performance on their benchmark. The three state-of-the-art NL-to-SQL systems tested on ScienceBenchmark are T5-Large, SmBoP, and ValueNet, where T5 (**T**ext-**t**o-**T**ext **T**ransfer **T**ransformer) is a large language model that allows adaptability to a variety of NLP tasks (in this case, text-to-SQL) (Raffel et al., 2020), SmBop (Semi-autoregressive Bottom-up Semantic Parsing) is a parser that uses RAT-SQL (an encoder framework that uses relation-aware self-attention) as encoder and a bottom-up decoder (Rubin and Berant, 2021), and ValueNet is an end-to-end NL-to-SQL system largely based on IRNet, which uses schema-linking and intermediate representations to tackle the mismatch between the intention expressed in the natural utterance and the query produced (Brunner and Stockinger, 2021; Guo et al., 2019).

Regarding the domain generalization issue, Wang et al. (2021) pointed out that little attention has been dedicated to learning algorithms which promote domain generalization, with most existing approaches instead relying on standard supervised learning. Moved by this motivation, they proposed a meta-learning framework which targets zero-shot domain generalization for semantic parsing, as illustrated in Figure 11. They showed that even when parsers are augmented with pre-trained models, e.g., BERT, their method can still effectively improve domain generalization in terms of accuracy.

All the above-mentioned challenges have been addressed to varying extents by recent advancements in text-to-SQL research, but there is still room for improvement in achieving accurate and robust conversion from natural language to SQL queries. The current thesis mainly addresses the problem of domain generalization by attempting the application of two exemplary text-to-SQL systems on a complex linguistics relational database.

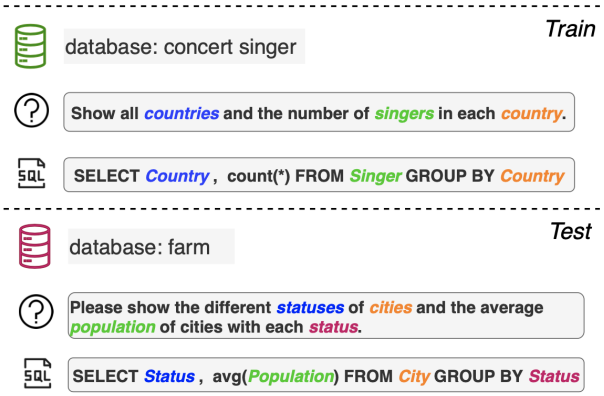


Figure 11: In zero-shot semantic parsing, during the training phase, a parser is exposed to examples related to the concert singer database. However, during the testing phase, it must generate SQL queries for questions that are related to a previously unseen database, namely the farm database (Wang et al., 2021).

4 Datasets

This thesis aims at analyzing the applicability of two exemplary but yet powerful text-to-SQL systems, namely IRNet and SmBoP. Both have been trained and tested on the Spider dataset.

In this thesis, only the pretrained models are deployed, which means that no re-training is performed. For evaluation of the research question, the inference is performed on the DIRNDL database. The main challenge arises in its intrinsic complex structure, comprising a macro and a micro layer, linguistic annotations, and multiple schemas. The Spider and DIRNDL datasets are described in the following subsections.

4.1 Spider

Spider[2] is a cross-domain semantic parsing and text-to-SQL dataset developed at Yale University. It consists of 10,181 sample questions and is annotated with 5,693 unique SQL queries, extracting information from 200 databases with multiple tables, covering 138 different domains. The 200 databases have been collected from three resources. First, about 70 databases from different college database courses, SQL tutorial websites, online csv files, and textbook examples have been collected. Second, about 40 databases have been collected from *DatabaseAnswers*, which contains over a thousand data models across different domains. These data models contain only database schemas, hence they have been converted into SQLite, and populated using an online database population tool. Finally, the remaining 90 databases have been created based on WikiSQL. The databases in Spider have been developed and encoded using the SQLite database engine(Yu et al., 2018).

The data fields in the training and development sets json files are as follows:

- **db_id:** database name;

[2] `https://yale-lily.github.io/spider`

- **question:** NL question to be interpreted into SQL;
- **query:** gold SQL query corresponding to the NL question;
- **query_toks:** list of tokens of the query;
- **query_toks_no_value:** list of tokens of the query with the omission of the explicit values if existing (e.g. if the query contains the clause "WHERE age $>$ 25", here it would become ["where", "age", ">", "value"]);
- **question_toks:** list of tokens of the NL question;
- **sql:** parsed version of the SQL query broken down in its various components as a nested dictionary.

In comparison with other previously-developed datasets for semantic parsing and text-to-SQL, Spider entails a large level of both complexity and diversity, quantified by the number of different SQL operations within the samples and the domain diversity of databases. Indeed the text-to-SQL datasets ATIS, GeoQuery, and Academic contain each only one database and most of them contain less than 500 unique SQL queries, which leads to the fact that models trained on these datasets are prune to bad generalization across different domains. Differently, the WikiSQL dataset has a large number of SQL queries and tables, but it is very simple, meaning that all SQL queries are basic and not particularly articulated, and each database in the dataset is a single table without any foreign key. The discrepancy between Spider and the other mentioned datasets is illustrated in Figure 12.

The Spider dataset has been created and made available for the Spider challenge, where the goal is to develop NL interfaces to cross-domain databases. In Spider, different complex SQL queries and databases appear in train and test sets. Hence, to achieve high test scores, systems must generalize well across different SQL queries and database schemas. In order to analyze the difficulty and demonstrate the purpose of the corpus, Yu et al. (2018) initially experimented with several state-of-the-art semantic parsing models, with the best model achieving only 12.4% exact matching accuracy in the database split setting, which suggested that there would be large

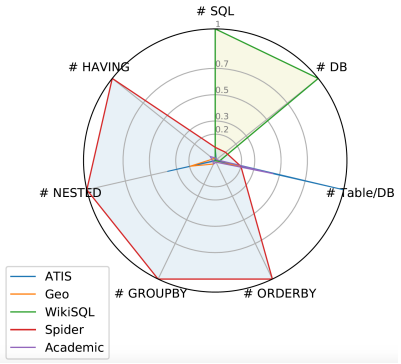


Figure 12: Spider spans the largest area in the chart. The large amount of diverse operations demonstrates its higher complexity (Yu et al., 2018).

room for improvement. Eventually, this lead to considerable research efforts aimed at enhancing existing text-to-SQL systems or developing new ones which could perform well on this benchmark.

4.2 DIRNDL

DIRNDL (**D**iscourse **I**nformation **R**adio **N**ews **D**atabase for **L**inguistic Analysis) [3] is an annotated corpus resource based on news broadcasts from the German radio station Deutschlandfunk, which has been prepared for the investigation of the interfaces between prosody, information status and syntax (Eckart et al., 2012).

The database contains two primary datasets, namely spoken data in the form of audio files and a written version of each broadcast (Eckart, 2017). The total length of the audio files is of approximately five hours, performed by five male speakers and four female speakers. They were annotated for pitch accents and prosodic boundaries following GToBI(S) (Mayer, 1995). The transcribed broadcast manuscripts comprise of 3221 sentences, annotated for referential information status (given-new distinction), following the RefLex annotation scheme from Riester and Baumann (2017).

The annotation layers are hence the results of two different processing pipelines,

[3] `https://www.ims.uni-stuttgart.de/en/research/resources/corpora/dirndl/`

namely one from the spoken primary data to prosodic annotations, and the other from the written primary data to recursive information status labels. When primary data is processed in different annotation pipelines, conflicting tokenizations tend to arise (Chiarcos et al., 2009). In the case of DIRNDL, the processing of the data in different pipelines introduces indeed deviations. In the prosodic pipeline, the term "tokens" refers to items that are actually pronounced. This introduces an uneven treatment of punctuation symbols. For instance, hyphens, as in *EU-Außenbeauftragter* (EU High Representative), are silent and are omitted in the speech data transcriptions. Conversely, the comma symbol in a numerical expression like *6,9* becomes a distinct token and is transcribed as the word "Komma." Opting for only one of the primary datasets results in a loss of information during processing, and removing segments of the speech data like fillers or slips of the tongue is not recommended. To address disparities between the primary datasets and variations in the outputs of the processing pipelines, links between the tokens generated by each pipeline are established, as shown in Figure 13. This approach allows retaining as much information as possible in the corpus and facilitates the extraction of data for studying specific phenomena (Eckart et al., 2012). The alignment of spoken discourse and its written counterpart in a database which bridges over the slight deviations between the two versions allows formulating complex queries that involve syntactic, information-structural, and prosodic properties (Riester and Piontek, 2015).

It is also relevant to highlight that in order to systematically annotate information status within complex news language, an analysis of syntactic structure is indispensable in order to highlight hierarchical relations. As referential expressions are often embedded inside each other, so are information status labels, which cannot be adequately represented within a linearly organised phonetic analysis tool. For this reason, the written manuscripts were parsed with the XLE system and the German LFG-grammar by Rohrer and Forst (2006).

The two types of data are aligned in a generic relational database management system described in Eckart et al. (2010) (Eckart et al., 2012). Indeed, to be able to run SQL queries that support linguistic hypotheses and retrieve linguistically significant statistics from the annotated data, the DIRNDL corpus was loaded into

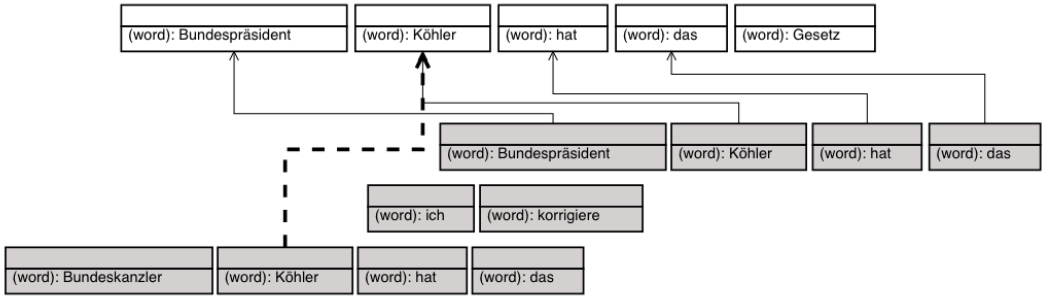


Figure 13: This example shows the primary (solid) and the secondary (dashed) links between the tokens from the written version (white), namely *Bundeskanzler Köhler hat das ich korrigiere Bundespräsident Köhler hat das...*, and the spoken version (grey), namely *Bundespräsident Köhler hat das Gesetz...*, of a sentence piece from DIRNDL (Eckart et al., 2012).

the B3 database (B3DB), implemented based on PostgreSQL [4].

The B3DB has evolved in several years and has been successfully applied in several projects, as in fact it was developed to allow for a generic treatment of different resources. Indeed, to test the tools' applicability, the current work uses a set of NL formulations and SQL queries which are available as they were used in the study from Riester and Piontek (2015), which made use of the B3DB infrastructure (Eckart, 2017). In this study the authors investigate relative givenness (Wagner, 2006) on the DIRNDL corpus and show that it is not givenness (defined as availability in the previous discourse context) that causes a modified noun in an adjective-noun combination to be deaccented. They also demonstrate that the claim from Wagner (2006) that the reason for deaccentuation is the presence of a local alternative (relative givenness) must be adapted, since also discourse-new nouns may be deaccented (Riester and Piontek, 2015). An example of query made available from this study is shown in Listing 1 in Section 6.

In-depth explanations of the B3DB structure are provided in Eckart (2017). Nevertheless, it is important to highlight some major design decisions.

[4] `https://www.postgresql.org/`

First, the database is conceptually divided into a macroscopic and a microscopic layer, which is indeed reflected in the data structures as each table is either part of the micro or the macro layer. The macro layer and some parallel tables from the micro layer B3DB are conceptually visualized in Figure 14. The micro layer contains generated annotations as structured and searchable information and generic object relation structures (structures of typed objects and relations appear on the micro layer with the tables "node", "edge", and "graph_type_definition"). Diversely, the macro layer contains process metadata, generated annotations as workflow objects, information about tools (for instance the version of the database), and generic object relations (structures of typed objects and relations appear on the macro layer with the tables "obj_definition", "obj_relation", and "type_definition").

A second major decision connected to this is the one of separating objects from contents. This is done in order to avoid unbalanced tables, as sometimes the bare existence of an object has to be declared, while other times the objects actually denote a content and the content could possibly be of huge dimensions. For this reason, an "obj_content" table is introduced on the macro layer, into which content strings are inserted and referenced by respective entries in the "obj_definition" table with a foreign key.

Third, all annotations on the micro layer are graphs, and each structure is mapped to the concept of nodes and edges, which are stored in respectively-named tables on the micro layer. Their entries are typed to allow a distinction among different groups of nodes and edges.

The last thing worth mentioning in terms of design choices is that the data structures of the micro-layer are based on the structures proposed by LAF ISO 24612:2012, an ISO standard on the Linguistic Annotation Framework (LAF) (see Section 2.4), its XML-serialization GrAF, the Graph Annotation Format, and an extension by Kountz et al. (2008) (Eckart, 2017). Standard formats play indeed an important role in interoperability and tend to be used to exchange data between more resource-specific formats without losing information in the process of mapping (Stede and Huang, 2012). The LAF data model introduces a layered graph structure, where graphs consist of nodes, edges and annotations, and accommodates all

standard annotation layers found in linguistic corpora. The referencing of primary data is established by encoding their minimal addressable unit, be it characters for textual representation or frames for video data. As a result, multiple modalities are encompassed. This makes the LAF data model particularly well suited for the applications of the DIRNDL corpus which has two types of primary data (Eckart, 2017). In relation to this, it is worth noting that schemas have been built to represent DIRNDL more closely to its actual annotation layers.

For the purposes of this thesis, not the totality of the B3DB structure that encodes the DIRNDL corpus is taken into account, but only a simplified sub-part of it. More details are provided in Section 9.

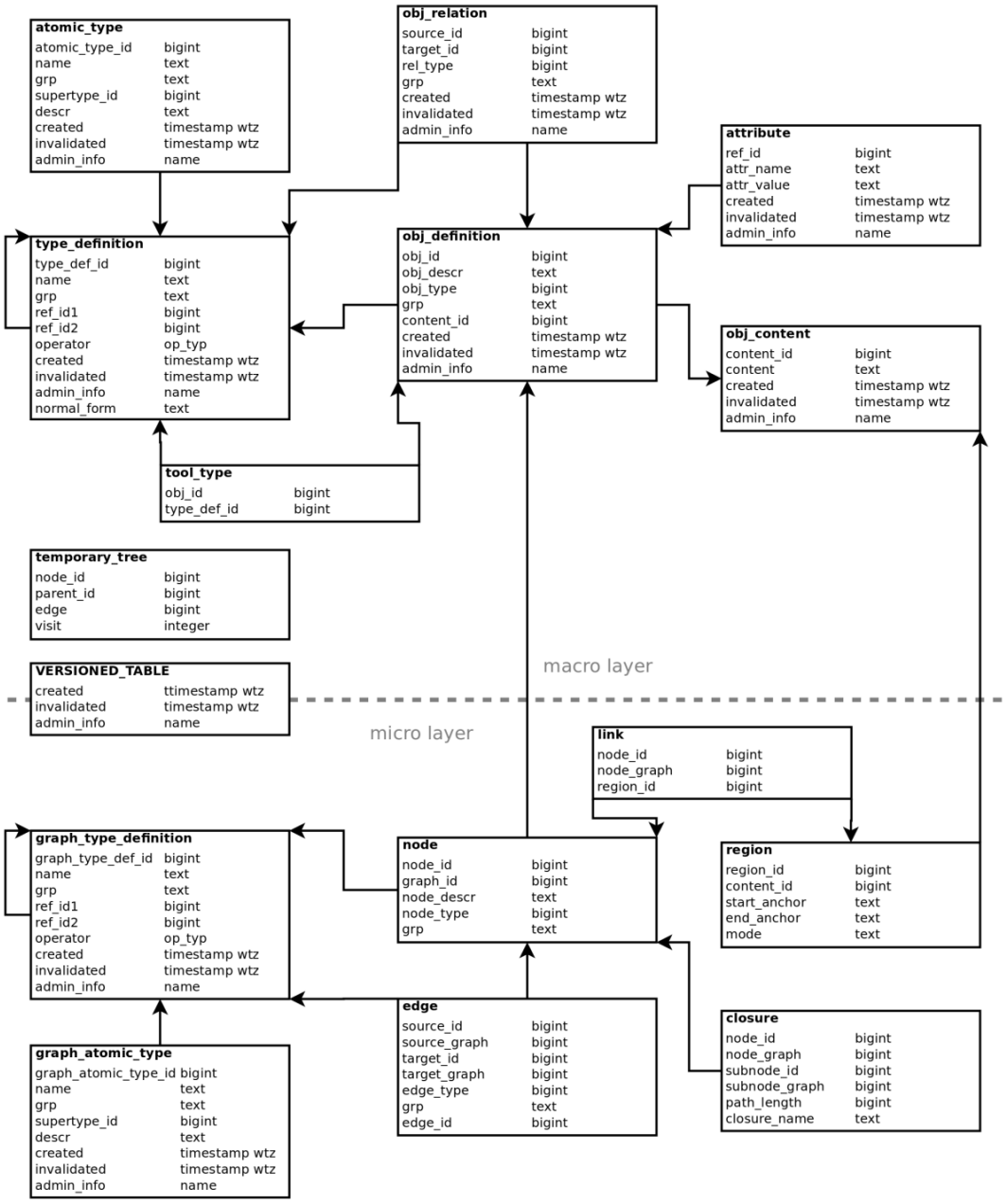


Figure 14: Partial representation of the B3DB schema where the macro layer and some parallel tables from the micro layer are shown (Eckart, 2017).

5 Tools

To test the adaptability and performance of pretrained models on the unseen multi-layered annotated DIRNDL relational database, two text-to-SQL systems trained on Spider are deployed. Based on the open-source availability of both the model implementation and the pretrained weights, the **I**ntermediate **R**epresentation **Net**work (IRNet)[5] and **Se**m**i**-autoregressive **Bo**ttom-up Semantic **P**arsing (SmBoP)[6] systems are selected within the scope of this thesis. Both networks are therefore briefly introduced and explained in the following.

5.1 IRNet

IRNet is a neural approach for complex and cross-domain text-to-SQL, which aims to address two main challenges in text-to-SQL tasks, namely the mismatch between expressed intents in NL, and the columns prediction hardship caused by a greater number of out-of-domain words (Guo et al., 2019). Differently from other approaches, IRNet decomposes NL into three phases instead of an end-to-end synthesis of the SQL query. The first step consists in a schema linking process over a database schema and a question, where schema linking means to identify references of columns, tables and condition values in NL queries, as exemplified in Figure 15 (Lei et al., 2020).

IRNet uses SemQL, which is a semantic query language that serves as an intermediate representation between SQL and NL. An illustrative example of SemQL is given in Figure 16a, whereas the context-free grammar is displayed in Figure 16b.

The model architecture includes a NL encoder, a schema encoder and a decoder, as shown in Figure 17. The NL encoder takes the NL input and encodes it into an embedding vector. These embedding vectors are there used to construct hidden states using a bi-directional LSTM. The schema encoder takes database schema as input and outputs representations for columns and tables. Finally, the decoder is used to synthesize SemQL queries using a context-free grammar. When the decoder

[5]Code is available at: `https://github.com/microsoft/IRNet`
[6]Code is available at: `https://github.com/OhadRubin/SmBop`

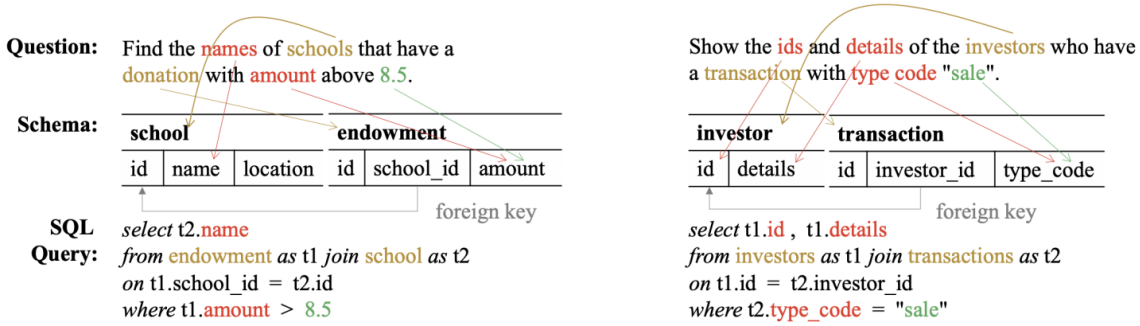
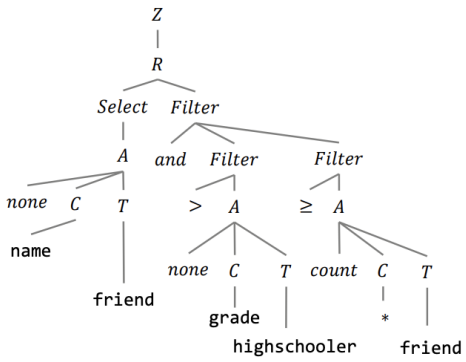


Figure 15: Two examples of schema linking. Column, table and value references are marked in red, yellow and green, respectively. The arrows of column and table references indicate their respective referents in the schema. For value references, the arrows point to the columns they compare with (Lei et al., 2020).



(a) An illustrative example of SemQL. Its corresponding natural language question is "Show the names of students who have a grade higher than 5 and have at least 2 friends" (Guo et al., 2019).

$$
\begin{aligned}
Z &::= \textit{intersect } R\ R \mid \textit{union } R\ R \mid \textit{except } R\ R \mid R \\
R &::= \textit{Select} \mid \textit{Select Filter} \mid \textit{Select Order} \\
&\quad \mid \textit{Select Superlative} \mid \textit{Select Order Filter} \\
&\quad \mid \textit{Select Superlative Filter} \\
\textit{Select} &::= A \mid A\ A \mid A\ A\ A \mid A\ A\ A\ A \mid A\ A \cdots A \\
\textit{Order} &::= \textit{asc } A \mid \textit{desc } A \\
\textit{Suerlative} &::= \textit{most } A \mid \textit{least } A \\
\textit{Filter} &::= \textit{and Filter Filter} \mid \textit{or Filter Filter} \\
&\quad \mid > A \mid > A\ R \mid < A \mid < A\ R \\
&\quad \mid \geq A \mid \geq A\ R \mid = A \mid = A\ R \\
&\quad \mid \neq A \mid \neq A\ R \mid \textit{between } A \\
&\quad \mid \textit{like } A \mid \textit{not like } A \mid \textit{in } A\ R \mid \textit{not in } A\ R \\
A &::= \max C\ T \mid \min C\ T \mid \textit{count } C\ T \\
&\quad \mid \textit{sum } C\ T \mid \textit{avg } C\ T \mid \textit{none } C\ T \\
C &::= \textit{column} \\
T &::= \textit{table}
\end{aligned}
$$

(b) The context-free grammar of SemQL. *column* ranges over distinct column names in a schema, *table* ranges over tables in a schema (Guo et al., 2019).

Figure 16: SemQL context grammar and example.

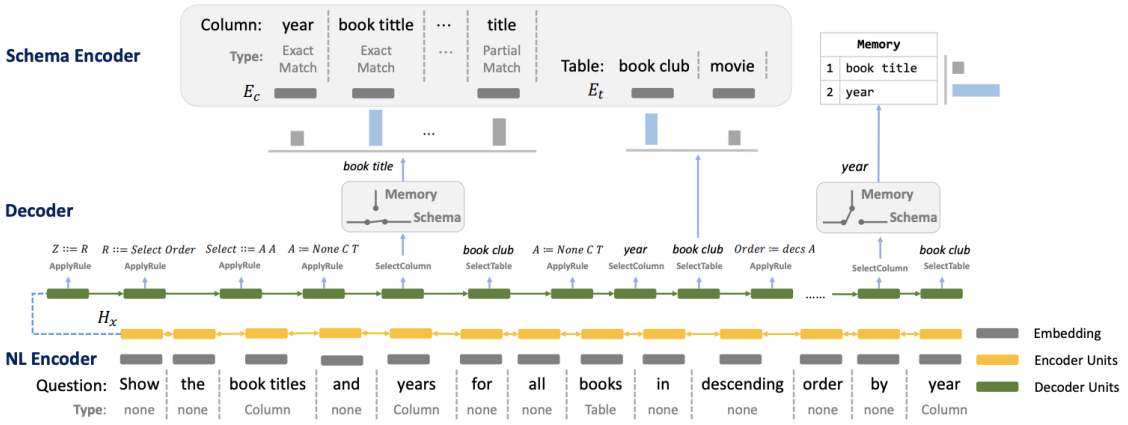


Figure 17: An overview of the neural model to synthesize SemQL queries. Basically, IRNet is constituted by an NL encoder, a schema encoder and a decoder. As shown in the figure, the column 'book title' is selected from the schema, while the second column 'year' is selected from the memory (Guo et al., 2019).

is going to select a column, it first makes a decision on whether to select from the memory or not, and then selects a column from the memory or the schema based on the decision. Once a column is selected, it will be removed from the schema and be recorded in the memory. IRNet yields a significant improvement of 19.5% over previous benchmark models by achieving 46.7% exact match (EM) accuracy at test time on the Spider dataset.

5.2 SmBoP

SmBoP is a model that uses a RAT-SQL encoder (Wang et al., 2020a) on top of the GraPPa (Yu et al., 2020), another pre-trained encoder tailored to semantic parsing. The RAT-SQL encoder provides a joint contextualized representation of the input (i.e. the utterance and the database schema), and uses relational-aware self-attention (Shaw et al., 2018) to encode the structure of the schema and other prior knowledge on relations between encoded tokens.

While the standard decoding method in recent years has mostly been to decode the output SQL query token-by-token from left to right or to decode the abstract

Algorithm 1: SmBoP

input: utterance x, schema S
$\mathbf{x}, \mathbf{s} \leftarrow \text{Encode}_{\text{RAT}}(x, S)$
$Z_0 \leftarrow$ Top-K schema constants and DB values
for $t \leftarrow 0 \dots T-1$ **do**
    $Z'_t \leftarrow \text{Attention}(Z_t, \mathbf{x}, \mathbf{x})$
    $F_{t+1} \leftarrow \text{Score-frontier}(Z'_t)$
    $F'_{t+1} \leftarrow \arg\max_K(F_{t+1})$
    $Z_{t+1} \leftarrow \text{Represent-beam}(Z_t, F'_{t+1})$
return $\arg\max_z(Z_T)$

Figure 18: The SmBoP algorithm (Rubin and Berant, 2021).

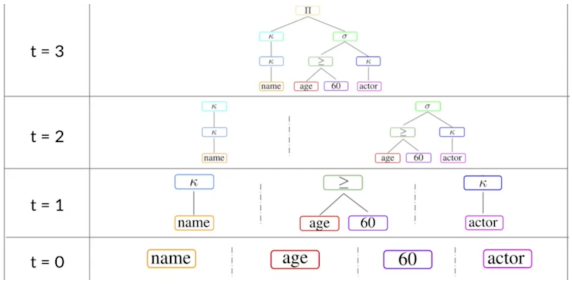
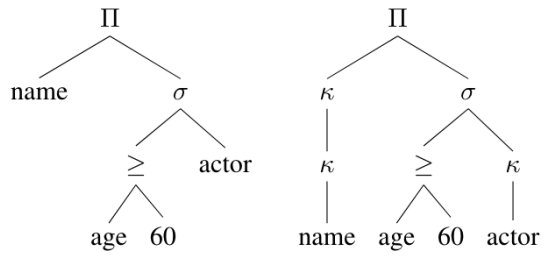


Figure 19: Illustration of the tree decoding process at different heights (t) (Rubin and Berant, 2021).

syntax tree of the target program in a top-down manner, bottom-up decoding has not received significant attention. However, the semi-autoregressive bottom-up decoder implemented in SmBoP is shown to counteract two drawbacks of the standard practice, namely the fact that the decoding steps is linear in the size of the gold SQL query (since it is a token-by-token decoding), and the fact that the decoder hidden state that is output in every decoding step corresponds to the prefix of the SQL query that has been decoded so far, and this representation does not have any clear semantics (Rubin and Berant, 2021). In SmBoP, after encoding the input, the decoder iteratively constructs the top-K abstract sub-trees, representing the query creation process. This process, visualized in Figure 19, leads to logarithmic inference time, rather than linear, since all trees of height t are decoded in parallel. Moreover, the representation that is held at every time step corresponds to well-formed and meaningful semantic trees. The SmBoP algorithm is shown in Figure 18.



(a) Unbalanced tree (b) Balanced tree

Figure 20: An unbalanced and balanced relational algebra tree (with the "KEEP" operation) for the utterance "What are the names of actors older than 60?", where the corresponding SQL query is "SELECT name FROM actor WHERE age $\geq$ 60" (Rubin and Berant, 2021).

Note in Figure 19, that the trees are not directly built with SQL syntax. Indeed, SQL is not well-suited for semantic parsing due to the mismatch between SQL and NL, hence they use relational algebra (Codd, 2001) in the decoding process. The rationale behind the use of relational algebra as intermediate representation is similar to Guo et al. (2019)'s decision of using SemQL, as both these query languages have a better alignment with NL. Rubin and Berant (2021) have additionally introduced a "KEEP" operation in the relational algebra grammar, which does not change the semantics of the subtree it is applied on, but only ensures that the distance from the root to all leaves is equal and hence the trees are balanced, as illustrated in Figure 20.

The EM accuracy of SmBoP on the Spider test dataset is of 69.5%, which is comparable to the EM score of the standard RATSQL+GraPPa architecture, which is 69.6%. In terms of Execution Accuracy, namely whether the executed SQL query maps to the correct answer in the database, SmBoP reaches a score of 71.1%, outperforming by roughly three percentage points the highest-scoring model in terms of Execution Accuracy on Spider.

6 Methodology and Contributions

The approach adopted in the current thesis is an application approach, where the main challenge and the main contribution is a qualitative analysis on what modifications on the DIRNDL database structure are necessary in order to make the IRNet and the SmBoP deployable in a way that they can retrieve linguistically significant information and hence possibly ease linguistic research.

The task is challenging since the Spider dataset contains 200 databases with one schema each, comprising relatively low numbers of tables, and with tables'/columns' names and values of straightforward interpretation. Moreover, the schemas of the databases are represented in SQLite format. On the other hand, the DIRNDL corpus encoded in the B3 database contains four schemas of respectively three, five, eight, and thirty tables, each with a varying number of columns that are sparsely populated, and where the table and column names and the value types do not allow for a straightforward interpretation. For instance, a node can be in DIRNDL a prosody token or a terminal or non-terminal node in a syntax tree, or other. In addition to this, the B3 database was implemented for PostgreSQL.

These aspects constitute the first and main challenge, namely bringing data structures modifications, for instance moving tables from some schemas to others and renaming columns and tables, as well as transferring from one RDBMS to another, with eventual changes in the syntax.

Furthermore, the SmBoP and IRNet systems take question-schema pairs as inputs, for instance *("What is the name of the players that scored more than 30 goals in 2018?", "soccer")*. Given that DIRNDL contains four schemas, and given that they come into play together for several linguistic queries for which I intend to perform inference, as shown in Listing 1, this is an additional challenge to investigate in the current thesis.

The methodology approach intended here is one of gradual adaptation, meaning that the modifications will be divided in gradual levels of complexity and the analysis of results will also follow this multi-step workflow. Hence, the first step of this

thesis is to obtain a format of the DIRNDL dataset that is at least partially (for instance, just with regards to one schema instead of all four directly) deployable as input for the IRNet and SmBoP systems. Also the attempts of inference will first be based on very simple requests. Once a positive outcome is reached, more challenging structure modifications and information requests will be taken into account. For this reason, Section 9 is divided in sub-sections of incremental complexity level.

```
select NT.graph_id, NT.node_id, NT.cat
    , array_agg(T.word)
    , array_agg(T.pos)
from syntax_nonterminal_nodes NT
    , closure C
    , syntax_terminal_nodes T
where NT.cat like 'DP[%'
    and C.node_id = NT.node_id and C.node_graph = NT.graph_id
    and C.subnode_id = T.node_id and C.node_graph = T.graph_id
group by NT.graph_id, NT.node_id, NT.cat
having every(T.pos not like 'A[%'); --(boolean_and)
```

Listing 1: Exemplary SQL query for the information need *"Get all DPs without an adjective"* in the DIRNDL database.

7 Input Formats

The first step in this thesis is the analysis of the input structure of the two systems. SmBoP uses the original Spider dataset in the original format. In particular, the dataset directory in the system contains (among other files related to training) a database sub-directory. This sub-directory contains one folder for each database instance, and each folder contains the *[db_name]*.sqlite file. In most cases, a schema.sql file with SQL statements to create the database is also provided together with the .sqlite file. In addition to this, there is a file named `tables.json` which contains a description of the tables and columns of each Spider database and the respective primary and foreign keys.

```
CREATE TABLE "captain" (
"Captain_ID" int,
"Name" text,
"Ship_ID" int,
"age" text,
"Class" text,
"Rank" text,
PRIMARY KEY ("Captain_ID"),
FOREIGN KEY ("Ship_ID") REFERENCES "Ship"("Ship_ID")
);

CREATE TABLE "Ship" (
"Ship_ID" int,
"Name" text,
"Type" text,
"Built_Year" real,
"Class" text,
"Flag" text,
PRIMARY KEY ("Ship_ID")
);

INSERT INTO "Ship" VALUES (1,"HMS Manxman","Panamax","1997","KR","Panama");
INSERT INTO "Ship" VALUES (2,"HMS Gorgon","Panamax","1998","KR","Panama");
INSERT INTO "Ship" VALUES (3,"HM Cutter Avenger","Panamax","1997","KR","Panama");
INSERT INTO "Ship" VALUES (4,"HM Schooner Hotspur","Panamax","1998","KR","Panama");
INSERT INTO "Ship" VALUES (5,"HMS Destiny","Panamax","1998","KR","Panama");
INSERT INTO "Ship" VALUES (6,"HMS Trojan","Panamax","1997","KR","Panama");
INSERT INTO "Ship" VALUES (7,"HM Sloop Sparrow","Panamax","1997","KR","Panama");
INSERT INTO "Ship" VALUES (8,"HMS Phalarope","Panamax","1997","KR","Panama");
INSERT INTO "Ship" VALUES (9,"HMS Undine","Panamax","1998","GL","Malta");

INSERT INTO "captain" VALUES (1,"Captain Sir Henry Langford",1,"40","Third-rate ship of the line","Midshipman");
INSERT INTO "captain" VALUES (2,"Captain Beves Conway",2,"54","Third-rate ship of the line","Midshipman");
INSERT INTO "captain" VALUES (3,"Lieutenant Hugh Bolitho",3,"43","Cutter","Midshipman");
INSERT INTO "captain" VALUES (4,"Lieutenant Montagu Verling",4,"45","Armed schooner","Midshipman");
INSERT INTO "captain" VALUES (5,"Captain Henry Dumaresq",5,"38","Frigate","Lieutenant");
INSERT INTO "captain" VALUES (6,"Captain Gilbert Pears",2,"60","Third-rate ship of the line","Lieutenant");
INSERT INTO "captain" VALUES (7,"Commander Richard Bolitho",3,"38","Sloop-of-war","Commander, junior captain");
```

Listing 2: Schema of the Spider database ship_1.

To illustrate the concept, Listing 2 shows the schema.sql file of the database ship_1, and this same database instance is represented in the `tables.json` file, as shown in Listing 3.

```
{
  "column_names": [
    [-1, "*"],
    [0, "captain id"],
    [0, "name"],
    [0, "ship id"],
    [0, "age"],
    [0, "class"],
    [0, "rank"],
    [1, "ship id"],
    [1, "name"],
    [1, "type"],
    [1, "built year"],
    [1, "class"],
    [1, "flag"]
  ],
  "column_names_original": [
    [-1, "*"],
    [0, "Captain_ID"],
    [0, "Name"],
    [0, "Ship_ID"],
    [0, "age"],
    [0, "Class"],
    [0, "Rank"],
    [1, "Ship_ID"],
    [1, "Name"],
    [1, "Type"],
    [1, "Built_Year"],
    [1, "Class"],
    [1, "Flag"]
  ],
  "column_types": ["text", "number", "text", "number", "text", "text", "text", "number", "text", "text", "number", "text", "text"],
  "db_id": "ship_1",
  "foreign_keys": [[3, 7]],
  "primary_keys": [1, 7],
  "table_names": ["captain", "ship"],
  "table_names_original": ["captain", "Ship"]
}
```

Listing 3: Instance of the Spider database ship_1 in the `tables.json` file.

By observing Listing 3, it can be noticed that **column_names** (key 0) has a value that is a list of sublists, where each sublist contains a number representing the table index and a column name. The first sublist represents the * function, which selects all columns, and has index -1. Then, for instance, the next six sublists, namely [0, “captain id”], [0, “name”], [0, “ship id”], [0, “age”], [0, “class”], [0, “rank”] indicate that the columns “captain id”, “name”, “ship id”, “age”, “class”, “rank” belong to the table with index 0.

Column_names_original (key 1) is the equivalent of column_names, but the column names are stated precisely like the original ones, so in this case with underscores and uppercase letters.
Column_types (key 2) specifies the datatype for every column described in column_names and in columns_names_original.
Db_id (key 3) specifies the database id name.
Foreign_keys (key 4) lists the foreign keys relations present in the database. In this case there is only a foreign key relation between the column with index 3 and the one with index 7. Index 3 corresponds to "ship id" in the table "captain", while index 7 corresponds to "ship id" in the table "ship" (and it is the primary key in this table).
Primary_keys (key 5) specifies which the primary keys are, namely the columns corresponding to the indices 1 and 7 in column_names, hence "captain id" and "ship id".
Table_names (key 6) specifies the table names just like column_names does for the columns. In the description of column_names, the concept of table index was already introduced. With respect to this database, the table with index 0 is "captain", whereas the one with index 1 is "ship".
Table_names_original (key 7) specifies the original table names just like column_names_original does for the column names. The official Spider GitHub repository provides a script to produce the corresponding `tables.json` file given a schema as input [7].

Another Spider file which is relevant for the current thesis is the `dev.json` file, which is a list of dictionaries where each dictionary contains a development set instance. Listing 4 shows a development instance. Here, **db_id** (key 0) states the database id name. **Query** (key 1) represents the gold SQL query corresponding to a certain natural language question (which is found in key 4). **Query_toks** (key 2) and **question_toks** (key 5) represent the tokenized versions of the query and question respectively. **Query_toks_no_value** (key 3) represents the tokenized query but

[7]Script can be found at: `https://github.com/taoyds/spider/blob/master/preprocess/get_tables.py`

omitting the specific values in case they appear in the query. For instance, if the query included the condition "WHERE age > 25", the "25" would be only represented by the word "value" in query_toks_no_value. Last, **sql** (key 6) represents the parsed result of this SQL query. This parsed version of the SQL query is the output of another script provided in the original Spider repository [8], which breaks down the components of the query in the form of a dictionary.

```
{
    "db_id": "concert_singer",
    "query": "SELECT count(*) FROM singer",
    "query_toks": ["SELECT", "count", "(", "*", ")", "FROM", "singer"],
    "query_toks_no_value": ["select", "count", "(", "*", ")", "from", "singer"],
    "question": "How many singers do we have?",
    "question_toks": ["How", ‘‘many", "singers", "do", "we", "have", "?"],
    "sql": {
    "from": {"table_units": [["table_unit", 1]], "conds": []},
    "select": [false, [[3, [0, [0, 0, false], null]]]],
    "where": [],
    "groupBy": [],
    "having": [],
    "orderBy": [],
    "limit": null,
    "intersect": null,
    "union": null,
    "except": null
    }
}
```

Listing 4: Instance of the Spider `dev.json` file based on the concert_singer database.

The input format of IRNet is of less trivial interpretation compared to SmBoP. The data which the IRNet authors indicate to use to run the system consist of the GloVe file containing the GloVe word embeddings, a `tables.json` file, a `dev.json` file, and a `train.json` file. However, while `tables.json` has the same structure as the original Spider file as described earlier, the `dev.json` and `train.json` files are the result of a IRNet-specific processing method and hence have a different structure. While the training-related data are not relevant for the current thesis, it is important to point out the extent to which the IRNet `dev.json` format differs from the original Spider one.

[8] Script can be found at: `https://github.com/taoyds/spider/blob/master/process_sql.py`

```
{
    "db_id": "concert_singer",
    "query": "SELECT count(*) FROM singer",
    "query_toks": ["SELECT", "count", "(", "*", ")", "FROM", "singer"],
    "query_toks_no_value": ["select", "count", "(", "*", ")", "from", "singer"],
    "question": "How many singers do we have?",
    "question_toks": ["How", "many", "singers", "do", "we", "have", "?"],
    "sql": {
    "from": {"table_units": [["table_unit", 1]], "conds": []},
    "select": [false, [[3, [0, [0, 0, false], null]]]],
    "where": [],
    "groupBy": [],
    "having": [],
    "orderBy": [],
    "limit": null,
    "intersect": null,
    "union": null,
    "except": null
    },
    "names": ["*", "stadium id", "location", "name", "capacity", "highest", "lowest", "average", "singer id", "name", "country", "song
           name", "song release year", "age", "is male", "concert id", "concert name", "theme", "stadium id", "year", "concert id",
          "singer id"],
    "table_names": ["stadium", "singer", "concert", "singer in concert"],
    "col_set": ["*", "stadium id", "location", "name", "capacity", "highest", "lowest", "average", "singer id", "country", "song name"
          , "song release year", "age", "is male", "concert id", "concert name", "theme", "year"],
    "col_table": [-1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 3, 3],
    "keys": {
    "18": 1,
    "1": 1,
    "21": 8,
    "8": 8,
    "20": 20,
    "15": 15
    },
    "origin_question_toks": ["How", "many", "singers", "do", "we", "have", "?"],
    "question_arg": [["how"], ["many"], ["singer"], ["do"], ["we"], ["have"], ["?"]],
    "question_arg_type": [["NONE"], ["NONE"], ["table"], ["NONE"], ["NONE"], ["NONE"], ["NONE"]],
    "nltk_pos": [
    ["how", "WRB"],
    ["many", "JJ"],
    ["singer", "NN"],
    ["do", "VBP"],
    ["we", "PRP"],
    ["have", "VB"],
    ["?", "."]
    ],
    "rule_label": "Root1(3) Root(5) Sel(0) N(0) A(3) C(0) T(1)"
}
```


Listing 5: Instance of the IRNet `dev.json` file based on the concert_singer database.

As shown in Listing 5, there are some extra keys compared to the original Spider `dev.json` file. Indeed, after sql (key 7) there is **names** (key 8), which specifies the column names in order. Then, there is **table_names** (key 9) which specifies the table names. Then, there is **col_set** (key 10), which specifies the column names avoiding repetitions (only distinct values). Then, there is **col_table** (key 11) which specifies

the index of the table which each column from names (key 8) belongs to; the "*" column is assigned index -1. **Keys** (key 12) specifies key-value pairs where each key is a column index and the value is the primary key which the column has a relation to. **Origin_question_toks** (key 13) shows the tokenized version of question (key 5). **Question_arg** (key 14) shows not only the tokenized version of question, but it also normalizes the tokens. **Question_arg_type** (key 15) is a list of the same length as question_arg and states the type of argument or entity corresponding to each element in question_arg; for instance, "singer" is a table, and hence "[table]" is to be found at the corresponding index in the question_arg_type list. **Nltk_pos** (key 16) shows the POS tags of the question tokens based on the segmentation found in question_toks. **Rule_label** (key 17) shows the result of a parsing rule applied to generate the SQL query from the NL question.

8 Evaluation Methods

The evaluation of SQL parsing models involves measuring the ability to correctly translate NL questions into SQL statements. This concept can be broken down into two main types of metrics, namely Exact Match (EM) and Execution (EX) Accuracy. EM evaluates whether the generated queries exactly match the gold queries, while EX evaluates the correctness of generated answers based on the execution results. It is important to point out that SQL is flexible in syntax, meaning that there are multiple ways to build a query which in the end outputs the correct information, hence EX is generally considered a more meaningful metric (Liu and Tan, 2023).

In the context of the current thesis, the performance of the two systems on the DIRNDL database can be evaluated using the official Spider evaluation script [9] evaluation script or with a custom script.

The Spider evaluation script first loads the data containing the NL questions and their corresponding gold SQL queries, and it parses the SQL statement into its fundamental components, like SELECT, WHERE, and others. The queries predicted by the systems are then compared to the gold queries using a series of evaluation metrics, namely EM accuracy, EX accuracy, partial match accuracy, recall, and F1-score. These metrics are defined as follows:

- EM Accuracy: the percentage of queries for which the predicted SQL matches the gold SQL exactly and completely.
- EX Accuracy: evaluation of the functional correctness of the predicted SQL by executing both the predicted and gold SQL statements on the database and comparing the results.
- Partial Match Accuracy/Recall/F1 score: the accuracy/recall/F1 score of individual SQL components by comparing each component of the predicted SQL to the corresponding component in the gold SQL. Each component of the SQL query is compared separately, which involves comparing:

[9] Script can be found at: `https://github.com/taoyds/spider/blob/master/evaluation.py`

– SELECT columns;
– SELECT columns ignoring any aggregate functions like SUM or COUNT;
– WHERE conditions;
– WHERE conditions ignoring the operators like > or =;
– GROUP BY columns ignoring any HAVING clause that might follow;
– GROUP BY columns including any HAVING clause that might follow;
– ORDER BY criteria;
– AND/OR operators used in the WHERE clause;
– INTERSECTION, UNION, EXCEPT, and NOT IN operations;
– keywords on the whole.

Category	**Components**
Group A	WHERE, GROUP BY, ORDER BY, LIMIT, JOIN, OR, LIKE, HAVING.
Group B	EXCEPT, UNION, INTERSECT, NESTED.
Group C	number of agg > 1, number of select columns > 1, number of where conditions > 1, number of group by clauses > 1.

Table 2: SQL Hardness Criteria.

As shown in Figure 21, in addition to the evaluation results according to the above-mentioned metrics, the output of the Spider script also displays the pairs of gold and predicted queries categorized in various hardness levels. Considering the three groups of SQL components stated in Table 2, the hardness levels are defined as follows:

1. **Easy (or low):**
    - the SQL key words have zero or exactly one element from group A and the query does not satisfy any conditions in group C. No element from group B is included.

2. **Medium:**

   - SQL satisfies no more than two rules in group C and does not have more than one word from group A and no word from group B.

     or

   - SQL includes exactly two words from group A and less than two rules in group C, and no word from group B.

3. **Hard:**

   - SQL satisfies more than two rules in group C, with no more than two words in group A and no word from group B.

     or

   - The number of key words from group A is at least two but maximum three, and the query satisfies no more than two rules in group C, but no word in group B.

     or

   - SQL has no more than one word from group A and no rule in group C, but exactly one word from group B.

4. **Extra hard:**

   - All other configurations left.

Nevertheless, the official evaluation script presents some limitations and does not take into account for instance n-gram overlap between the gold and predicted queries. For this reason a custom evaluation script has been developed, which provides a detailed comparison of various components of the SQL queries, such as keywords, arguments, and WHERE clause elements.

This evaluation is performed by analyzing the structural elements of the queries and computing their similarity using set operations and n-gram overlap techniques.

```
1. medium pred: SELECT prosody_nodes.word FROM prosody_nodes, syntax_terminal_nodes WHERE prosody_graph.filesequence = 1
                AND syntax_terminal_nodes.word = 3
1. medium gold: SELECT word FROM syntax_terminal_nodes WHERE s_num = 3 AND seq = 1
2. easy pred: SELECT COUNT(*) FROM prosody_nodes WHERE prosody_nodes.word LIKE '%3%'
2. easy gold: SELECT MAX(seq) FROM syntax_terminal_nodes WHERE s_num = 3
3. medium pred: SELECT prosody_nodes.word FROM prosody_nodes WHERE prosody_nodes.ts = 3
3. medium gold: SELECT pos FROM syntax_terminal_nodes WHERE s_num = 3 ORDER BY seq
4. medium pred: SELECT syntax_terminal_nodes.pos FROM prosody_nodes WHERE prosody_nodes.word = 3
4. medium gold: SELECT pos FROM syntax_terminal_nodes WHERE s_num = 3 AND seq = 1
5. easy pred: SELECT prosody_graph.graph_id FROM prosody_graph WHERE prosody_graph.filesequence = 3
5. easy gold: SELECT graph_id FROM syntax_graph WHERE s_num = 3
6. medium pred: SELECT prosody_nodes.word FROM prosody_nodes WHERE closure.closure_name = 'dlf-nachrichten-200703262000'
6. medium gold: SELECT word FROM prosody_nodes WHERE tone = 'NONE' AND filesequence = 'dlf-nachrichten-200703262000'
7. easy pred: SELECT is_target_syn_root.is_label FROM node WHERE node.node_id = 900514
7. easy gold: SELECT is_label FROM is_target_syn_root WHERE syn_root_id = 900614
8. medium pred: SELECT node.node_type FROM node WHERE node.node_id = 900620
8. medium gold: SELECT node_type, grp FROM node WHERE node_id = 900620
9. medium pred: SELECT edge.edge_type FROM edge WHERE edge.source_id = 56666 AND edge.target_graph = 56690
9. medium gold: SELECT grp FROM edge WHERE source_id = 56666 AND source_graph = 1349 AND target_id = 56690
10. easy pred: SELECT graph_type_definition.name FROM graph_type_definition
               JOIN node ON graph_type_definition.graph_type_def_id = node.node_type
               WHERE syntax_nonterminal_nodes.tiger_node_id = 'tigerXML_node'
10. easy gold: SELECT name FROM graph_type_definition WHERE grp = 'tigerXML_node'
```

	easy	medium	hard	extra	all
count	4	6	0	0	10
EXECUTION ACCURACY					
execution	0.000	0.000	0.000	0.000	0.000
EXACT MATCHING ACCURACY					
exact match	0.000	0.000	0.000	0.000	0.000
PARTIAL MATCHING ACCURACY					
select	0.750	0.333	0.000	0.000	0.500
select(no AGG)	0.750	0.333	0.000	0.000	0.500
where	0.250	0.167	0.000	0.000	0.200
where(no OP)	0.250	0.167	0.000	0.000	0.200
group(no Having)	0.000	0.000	0.000	0.000	0.000
group	0.000	0.000	0.000	0.000	0.000
order	0.000	0.000	0.000	0.000	0.000
and/or	1.000	0.667	0.000	0.000	0.800
IUEN	0.000	0.000	0.000	0.000	0.000
keywords	0.750	0.833	0.000	0.000	0.800
PARTIAL MATCHING RECALL					
select	0.750	0.333	0.000	0.000	0.500
select(no AGG)	0.750	0.333	0.000	0.000	0.500
where	0.250	0.167	0.000	0.000	0.200
where(no OP)	0.250	0.167	0.000	0.000	0.200
group(no Having)	0.000	0.000	0.000	0.000	0.000
group	0.000	0.000	0.000	0.000	0.000
order	0.000	0.000	0.000	0.000	0.000
and/or	1.000	1.000	0.000	0.000	1.000
IUEN	0.000	0.000	0.000	0.000	0.000
keywords	0.750	0.833	0.000	0.000	0.800
PARTIAL MATCHING F1					
select	0.750	0.333	0.000	0.000	0.500
select(no AGG)	0.750	0.333	0.000	0.000	0.500
where	0.250	0.167	0.000	0.000	0.200
where(no OP)	0.250	0.167	0.000	0.000	0.200
group(no Having)	1.000	1.000	0.000	0.000	1.000
group	1.000	1.000	0.000	0.000	1.000
order	1.000	1.000	0.000	0.000	1.000
and/or	1.000	0.800	0.000	0.000	0.889
IUEN	1.000	1.000	0.000	0.000	0.000
keywords	0.750	0.833	0.000	0.000	0.800

Figure 21: Exemplary output of the official Spider evaluation script for Experiment 1 of Section 9.

The output of this custom evaluation script [10] for the first query produced by the SmBop system for the first question of Experiment 1 (explained later in Section 9 and in Table 3) is shown in Figure 6.

The script contains an `extract_clause_components` function, which parses SQL queries to extract essential components, including keywords (e.g., SELECT, FROM, WHERE), arguments (e.g., column names, table names), and elements within the WHERE clause (e.g., columns, operators, values). This function utilizes the sql-parse library to tokenize the query and identify relevant components. It handles nested queries and recognizes aggregate functions (e.g., MAX, COUNT) and logical operators (e.g., AND, OR, IN).

The similarity between two sets A and B of extracted components is calculated using the Jaccard similarity coefficient. This measure is defined as the size of the intersection divided by the size of the union of the sets

$$\text{Similarity} = \frac{|A \cap B|}{|A \cup B|}$$

where $|A \cap B|$ is the number of elements common to both sets, and $|A \cup B|$ is the total number of unique elements in both sets. The `calculate_similarity` function implements this formula, providing a quantitative measure of similarity ranging from 0 to 1, where 1 indicates identical sets.

To capture more nuanced differences, the script also calculates n-gram overlap between sets of components. The `calculate_ngram_overlap` function decomposes each component into n-grams (substrings of length n, set to 3 in this case) and computes the overlap between the sets of n-grams. This approach helps in identifying partial matches and structural similarities beyond exact matches.

The `evaluate_similarity` function reads gold standard queries and predicted queries from input files, extracts their components, and computes similarity scores

[10] The script can be found in Appendix B and at: `https://github.com/mariavittoriaateri/MasterThesis/blob/main/dirndl_eval.py`

for keywords, arguments, and WHERE clause elements. It then prints detailed similarity metrics, including both Jaccard similarity and n-gram overlap, for each pair of queries. This function facilitates a comprehensive comparison by providing insights into which components match and which do not.

```
Query 1 similarity:
  Keyword similarity: 1.00
  Keyword n-gram overlap: 1.00
  Argument similarity: 0.33
  Argument n-gram overlap: 0.21
  Operator similarity: 1.00
  Operator n-gram overlap: 1.00
    Gold keywords: {'FROM', 'WHERE', 'SELECT'}
    Predicted keywords: {'FROM', 'WHERE', 'SELECT'}
    Gold arguments: {'word', 'syntax_terminal_nodes'}
    Predicted arguments: {'word', 'prosody_nodes'}
    Gold operators: 'None'
    Predicted operators: 'None'
  WHERE clause component similarities:
    Column similarity: 0.00
    Column n-gram overlap: 0.07
    Operator similarity: 1.00
    Operator n-gram overlap: 1.00
    Value similarity: 1.00
    Value n-gram overlap: 0.00
      Gold WHERE columns: {'seq', 's_num'}
      Predicted WHERE columns: {'filesequence', 'word'}
      Gold WHERE operators: {'AND', '='}
      Predicted WHERE operators: {'AND', '='}
      Gold WHERE values: {'1', '3'}
      Predicted WHERE values: {'1', '3'}
```

Listing 6: Exemplary output of the custom evaluation script for the first predicted query in Experiment 1 of Section 9.

Last, in order to effectively observe the results of the execution of the predicted SQL queries, an execution script has been developed. This script takes three arguments, namely the path to the .sqlite database from which it is supposed to retrieve the information, the path to the .sql file containing the predicted queries, and the name of the output file to be written.

9 Experiments

Three experiments are conducted in the current thesis to observe the performance potential of the IRNet and SmBop systems on the DIRNDL.

Although the three versions of database used in the three experiments display some differences, they all share a common base architecture for which a description follows. Indeed, the DIRNDL database has to be adapted to a type of format that can be processed as input by the two systems and the systems' performance evaluation method.

The B3DB representing the DIRNDL corpus is implemented based on PostgreSQL and the schemas were built to represent DIRNDL as close as possible to its actual annotation layers. On a less abstract level this means that when accessing the DIRNDL corpus B3DB in PostgreSQL, one can observe the presence of four schemas, each of which encompasses a comprehensive representation of the data model, reflecting the structure and relationships inherent in the underlying data.

The first schema is the nuc schema, containing 5 tables. This schema contains mainly information on which words contain nuclear and prenuclear accents and the tone sequences of words in the corpus sentences.

The second schema is the public schema, containing 30 tables. This schema is the main skeleton of the database, since all the graphs representing different annotation layers in the other three schemas map to the entities defined in this schema. For instance, the prosody_nodes table in the reflex schema contains the columns filesequence, graph_id, node_id, ts, word, tone, accents, and here the values graph_id and node_id are inherited from and mappable to the node table in the public schema, which belongs to the micro layer of the B3DB and has the aim of uniquely defining the node entities.

The third schema is the reflex schema, containing 8 tables. This schema contains (morpho)syntactic and prosodic information, as well as information status labels.

The fourth schema is the syl schema, containing 3 tables. This schema mainly contains information on pitch accents at the syllable level.

Before diving into the technical modifications carried out, it is relevant to highlight again that the main motivation of this thesis is to analyze and test the applicability of the two text-to-SQL systems on the DIRNDL in order to aid information retrieval in linguistics research, in this case based on the case study of the paper "Anarchy in the NP. When new nouns get deaccented and given nouns don't" (Riester and Piontek, 2015). Based on this, the modifications carried out involve only a sub-part of the database, which concretely means that a piece of database is extracted from the bigger structure and it includes only a subset of tables and relations, which have been processed in order to be used as input into the systems. Subsequently the performance of the systems is tested with a set of questions of variable complexity in the three experiments. Figure 22 shows the sub-part of the database taken into account for the first experiment, which also corresponds to the ground architecture used for the second and third experiments, as further explained in the corresponding sections. This database includes 7 tables from the reflex schema, namely syntax_graph, syntax_terminal_nodes, syntax_nonterminal_nodes, is_graph, is_target_syn_root, prosody_graph, and prosody_nodes, and 4 from the public schema, namely graph_type_definition, node, edge, and closure. A short explanation follows. For better readability, every time a table is introduced, it is in bold print.

Starting from the bottom of Figure 22, three tables can be seen. **Syntax_graph** contains 3221 values of graph_id and 3221 values of s_num (which stands for "sentence number"), as each of the 3221 sentences in the corpus is represented as a graph and hence has one corresponding graph_id.
The same graph_id - s_num pairs appear in both the **syntax_nonterminal_nodes** and the **syntax_terminal_nodes** tables. Nevertheless, in syntax_terminal_nodes the 3221 distinct values appear as many times as many words exist in the corpus, and each word is connected to a node_id and has a corresponding part-of-speech (pos) tag. For instance, if the first sentence has nine words, then each word has its own node_id and pos, but all nine share the same corresponding s_num and graph_id. The seq column contains values which correspond to the indices of words. For instance, if the first sentence starts as "Der Iran will [...]", then *Der* has seq=1, *Iran* has seq=2, *will* has seq=3, and so on. Last, both syntax_terminal_nodes and syn-

tax_nonterminal_nodes contain syntactic information based on the TIGER annotation scheme (Uszkoreit et al., 2003) and for this reason they both have a tiger_node_id attribute. In the case of syntax_terminal_nodes a tiger_node_id could for instance have the value of s1_1, and this would mean that this tiger node is the one corresponding to the first word (_1) of the first sentence (s1). The interpretation of syntax_nonterminal_nodes is similar to syntax_terminal_nodes, but the main difference lies in the fact that here the values of node_id do not correspond to the words of the corpus sentences, but to the non-terminal nodes building the syntax trees which have words as their terminal nodes. Last, cat assigns a syntactic category based on a LFG-grammar (e.g. ROOT, Cbar, NP, etc.) to each non-terminal node of each syntactic tree.

Observing now the top right corner of the diagram, the tables **is_graph** and **is_target_syn_root** are displayed. They contain information on information status (is). Is_graph contains a total of 5488 graph_id values, where half of these correspond to the lexical layer (stated as DIRNDL-RefLex-L-2013-10-25_is.xml in the layer column), while the other half corresponds to the referential layer (stated as DIRNDL-RefLex-R-2013-10-25_is.xml in the layer column). All the 5488 distinct graph_id values appear also in is_graph_id in the table is_target_syn_root. This table also has a s_num column, which has 2774 distinct values, each of which representing a sentence number. However, not the totality of the corpus sentences has been annotated, which explains why they are 2774 instead of 3221 in this case. Each sentence exists as representation on the referential layer (R-layer) graph and on the lexical layer (L-layer) graph. This means that each s_num value has two distinct is_graph_id associated to it. Then, the specific lexical or referential status annotation (e.g. "L-NEW", "R-GENERIC", etc.) is represented in the is_label column. It can be said that the graph_id - layer pairs from the is_graph table have a projection on the is_target_syn_root table as is_graph_id - is_label pairs. As far as syn_root_id is concerned, the values found in this column correspond to the values of node_id either in syntax_terminal_nodes (when the is_label value concerns the L-layer) or in syntax_nonterminal_nodes (when the is_label value concerns the R-layer). In the case of syntax_nonterminal_nodes, both single words and phrases

can be labeled. This correspondence between syntax_terminal_nodes and L-layer and syntax_nonterminal_nodes and R-layer does not hold in the totality of the cases, as some rare outliers exist. Nevertheless, it can be stated that each value in syn_root_id corresponds to a terminal or non-terminal node from the two syntax nodes tables, and following the same logic, the values in syn_graph_id correspond to the ones of graph_id in the syntax nodes tables and hence also in the syntax_graph table (which also means that each syn_graph_id matches one s_num). Last, each value in is_node_id is a representation at the level of the information status graph of a syn_root_id node. The column is_node_descr contains values which are identifiers resulting from the application of the SALSA annotation tool (Burchardt et al., 2006) and paired to the respective is_node_id.

On the left of the diagram the tables **prosody_graph** and **prosody_nodes** can be observed. Prosody_graph contains 55 distinct pairs of filesequence - graph_id values, where each filesequence represents one of the news files that make up the corpus, and since each of these is represented as a graph, it also has a graph_id connected to it. In the prosody_nodes table one can find all the words contained in these news files and the timestamp (ts) showing when the word was pronounced. Every word has its correspondent node_id and annotations about the tone and accents of the word itself. Each word corresponds to a certain filesequence - graph_id pair. For instance, there could be a sentence starting as "Der Iran will...", which is contained in the file dfl-nachrichten-20070325000, which corresponds to the graph_id with value 74471; this would mean that the individual words *der*, *Iran*, *will* all map to this filesequence - graph_id pair in the prosody_nodes table.

The last tables left to describe are the ones which are originally part of the public schema, and hence the core part of the B3DB.
The **node** table contains information about all the nodes that are part of the various graphs representing different types of annotations (e.g. the graph representing the annotation about the syntactic structure of the sentences). Each node has a unique node_id and a reference to the graph object it belongs to (graph_id). Nodes can also carry a node description (node_descr), for instance an important part of the annotation or information on the order of the nodes. Nodes are also typed, which means

that they have a node_type attribute which references graph_type_def_id from the graph_type_definition table. Each value in graph_type_def_id is a numeric id associated to a value in the name column, which describes the entity type with a keyword. For instance, the node with node_id = 900620 in the node table has the attribute node_type = 145. This node_type value is to be found also in graph_type_def_id in the graph_type_definition table, and by observing the corresponding value in the name column, the meaning behind the value "145" can be understood. In this example, the node type 145 means "non-terminal node". The node table also contains a grp attribute, which references the homonymous column in the graph_type_definition table and has a function similar to the name attribute, with the difference that it introduces a classification into broader groups. For instance, a value of grp can be tigerXML_node or bitpar_node, to signal that a certain node results from the application of a certain parser or tool, but omitting more fine-grained specifications. In other words, the group of a node (grp) mostly denotes the annotation layer a node belongs to. The attribute node_descr describes the node in the more narrow sense, e.g. if a node is a tigerXML node, then its node description can be for instance s1_525, which is an identifier for a syntactically annotated node in the TIGER-XML encoding.

The node table is closely connected to the **graph_type_definition** table, as stated earlier. Indeed, the type - group pairs of the tables node and edge refer to this table, which was developed with the task of storing all the entity types and groups. In addition to this, the table contains metadata about the resource information, for instance the admin name and the creation date. Last, the table contains an operator attribute and two ref_id attributes. This is due to the fact that in some cases a type can be combined by means of subtypes (ref_id1,ref_id2) and a specific operator taking one of the values from {id, neg, or, and}. Types with "id" as operator are types which do not result from the combination of subtypes.

The node and graph_type_definition tables are also closely connected to the **edge** table. Indeed, as mentioned in 2.4, LAF introduces a layered graph structure where graphs consist of nodes, edges and annotations, and the nodes and edges are stored in respectively named tables, with their entries being typed to distinguish differ-

ent groups of nodes and edges (Eckart, 2017). The edge table stores the source_id and target_id values, as well as source_graph and target_graph values, which have a correspondence respectively with node_id and graph_id in the node table. In this setting with only a sub-part of the database, the role of the edge table is particularly important because it connects the prosody_nodes table to the syntax_terminal_nodes table. Indeed, as explained earlier, prosody_nodes contains a node_id column, where each value identifies one word, and it also contains a graph_id identifying the news file that contains a certain number of nodes and hence words. All the node_id - graph_id pairs from prosody_nodes are stored as source_id - source_graph pairs in the edge table and also as node_id - graph_id pairs in the node table. Conceptually, the nodes in prosody_nodes are the source nodes mapping to the the nodes in syntax_terminal_nodes, which can be considered as the target nodes encoding syntactic information (see Section 4.2). All the node_id - graph_id pairs from syntax_terminal_nodes are stored as target_id - target_graph pairs in the edge table and also as node_id - graph_id pairs in the node table. The edge_id is unique and a numeric identifier is assigned to a new edge. However some edges are not annotated, and in such cases the the edge_id is not strictly necessary. Annot_id and annot_graph are necessary attributes if some type of annotation on the edge itself exists.

To conclude, it is relevant to also state that syntax_terminal_nodes and syntax_nonterminal_nodes are connected to each other. Indeed, the node_id - graph_id pairs in syntax_terminal_nodes are stored as subnode_id - subnode_graph pairs in the **closure** table and also as node_id - graph_id pairs in the node table. Similarly, the node_id - graph_id pairs in syntax_nonterminal_nodes are stored as node_id - node_graph pairs in the closure table and also as node_id - graph_id pairs in the node table. Building a transitive closure is one of the possible approaches that allow traversing deep structures like graphs or trees, and this is exactly the reason why a closure table exists in the database. Indeed, the closure table stores the path in the graph from a start node (node_id - node_graph) to a target node (subnode_id - subnode_graph). The givenness of the path length is optional but present in most of the cases and stored as path_length attribute. Each closure must also have a name (in the closure_name column).

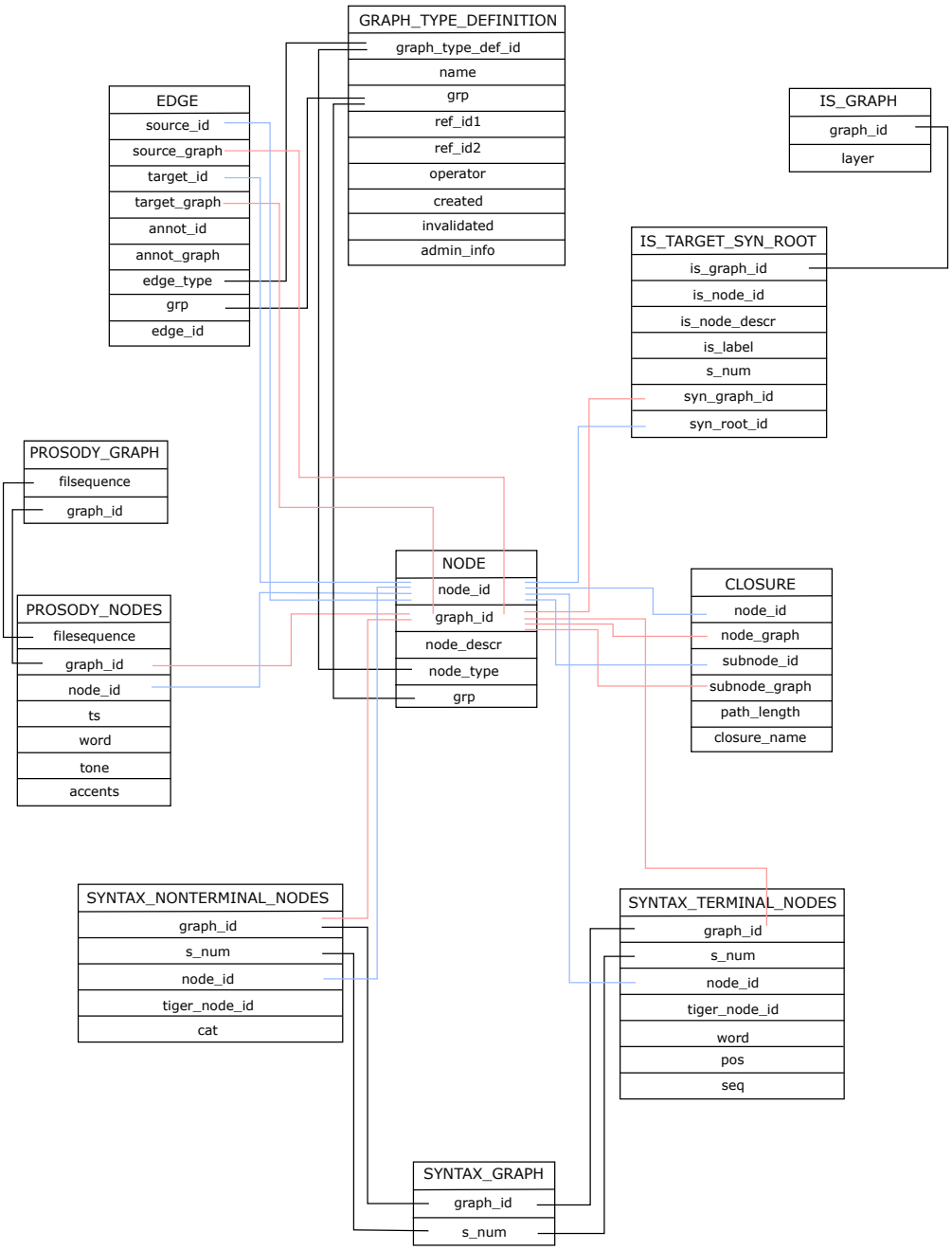


Figure 22: DIRNDL database tables and foreign key relations included in Experiment 1. The red lines show the foreign keys that map to graph_id in the node table; the blue lines show the foreign keys that map to node_id in the node table; the black lines show the foreign keys that map to other entities.

9.1 Experiment 1

The first experiment involves the first step of adaptation of the DIRNDL database, in which a subset of tables, columns, and foreign key relations has been extracted from the original database, and an initial set of questions of low-medium complexity shown in Table 3. The database structure is shown in Figure 22. As specified in the official Spider page [11], the SQL queries are classified in four levels of increasing hardness, namely "easy", "medium", "hard", and "extra hard". The decision of using a set of easy and medium hardness queries as first experimental data stems from the fact that the "easy" category is very restrictive and too limiting. The hardness classification and the evaluation methods adopted are described in detail in Section 8.

9.1.1 Evaluation of Experiment 1

This section presents the results obtained from this experiment and how they contribute to the overall research objectives.

An overview of the NL questions, the gold and predicted queries, as well as the hardness classification for the current experiment is shown in Table 4. As it can be observed in the table, while the overall performance of both systems is not satisfactory and neither system achieves perfect accuracy, the analysis of individual components reveals some insights into where the models perform better and where they struggle.

The IRNet system demonstrates a reasonably good performance in identifying the main SQL keywords such as SELECT, FROM, and WHERE. Most queries have a keyword similarity score around or above 0.75, with some reaching 1.00, indicating that the structure of the queries is often correctly recognized. However, it tends to insert JOIN clauses even when they are not needed. However, the identification of arguments (such as table names and columns) shows more variability and generally lower similarity scores compared to keyword identification. In particular, the

[11] `https://github.com/taoyds/spider`

evaluation of WHERE clause components reveals specific challenges:

- column similarity: scores are consistently low, with many queries having a column similarity of 0.00. This indicates difficulty in correctly identifying which columns are being queried;
- operator similarity: although there are occasional perfect scores (e.g., query 5 from Table 4), the overall performance is inconsistent. The model sometimes correctly identifies operators like = but fails with others, and interestingly it tends to use the LIKE operator instead of the = operator quite often without following a pattern. It is not the case, for example, that the LIKE operator appears more often with text instead of digits or the other way around;
- value similarity: the model frequently fails to match the correct values in the WHERE clauses, resulting in low similarity scores across the board.

Interestingly, the observation of n-gram overlap in similar column names indicates that refining the model to better understand context-specific naming conventions or modifying the database to ease the task for the models could lead to improved performance in argument identification. Indeed, a pattern can be observed in all queries. This pattern consists in the fact that the system selects some tables or columns which are not the same as the gold ones, but they have a partial lexical overlap, for instance the FROM clause in the gold query 2 from Table 4 selects the syntax_terminal_nodes table, whereas the respective predicted query selects the prosody_nodes table. For further details, the whole evaluation output of the IRNet system in this experiment is shown in Listing 8 in Appendix B.

The SmBop system also demonstrates a reasonably good ability to identify SQL keywords, often achieving perfect or near-perfect scores, and in contrast to the IRNet system, it tends to make less use of JOIN clauses. The recognition of arguments is less consistent and generally achieves lower similarity scores, but nevertheless overall better than IRNet. As far as the WHERE clause is concerned, the operator similarity is quite high and also the value similarity scores outperform the IRNet ones, as shown in Figure 23. However, similarly to IRNet, column similarity scores are consistently

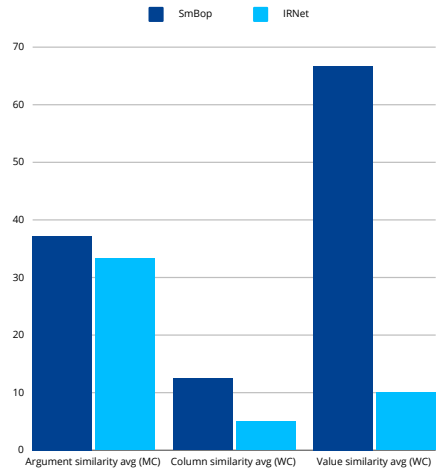


Figure 23: Average percentile scores produced by the systems with regards to argument similarity in the main clause (MC), and column and value similarity in the where clause (WC).

low. Moreover, also in the case of SmBop the system tends to select wrong columns and tables which still have a partial lexical overlap with the gold standard. For instance in the first predicted query (visualized in Table 4) an example of a table swap due to columns with the same name (namely "word", which appears both in syntax_terminal_nodes and in prosody_nodes) can be observed. For further details, the whole evaluation output of the SmBop system in this experiment is shown in Listing 7 in Appendix B.

Figure 23 shows a summary of the experiment results highlighting the differences between the two systems. This graph does not include the similarity scores of all query components, but only of the three which are most significant for the current research, namely argument similarity in the main clause, and column and value similarity in the where clause.

Concerning what was defined in Section 8 as Execution Accuracy, both systems perform poorly and fail almost completely to retrieve the correct information. The only query which manages to partially retrieve the correct information is the query 8 produced by the SmBop system, which retrieves the correct value for the node_type attribute but not for the grp attribute.

Question	Gold Query	Gold Output
1. What is the 1st word of sentence number 3?	`SELECT word FROM syntax_terminal_nodes WHERE s_num = 3 AND seq = 1`	der
2. How many words does sentence 3 have?	`SELECT MAX(seq) FROM syntax_terminal_nodes WHERE s_num = 3`	20
3. What is the pos of each word in sentence 3?	`SELECT pos FROM syntax_terminal_nodes WHERE s_num = 3 ORDER BY seq`	D[std], N[comm], ...
4. What is the pos of the first word in sentence 3?	`SELECT pos FROM syntax_terminal_nodes WHERE s_num = 3 AND seq = 1`	D[std]
5. What is the graph id of sentence number 3?	`SELECT graph_id FROM syntax_graph WHERE s_num = 3`	62632
6. Which words belonging to the file dlf-nachrichten-200703262000 have no marked tone?	`SELECT word FROM prosody_nodes WHERE tone = 'NONE' AND filesequence = 'dlf-nachrichten-200703262000'`	Bundeskanzlerin, rechnet, ...
7. What is the information status label of node 900514?	`SELECT is_label FROM is_target_syn_root WHERE syn_root_id = 900614`	L-NEW
8. What is the type and the group of node 900620?	`SELECT node_type, grp FROM node WHERE node_id = 900620`	145, tigerXML_node
9. What is the group of the edge connecting the source node with id 56666 and source_graph 1349 to the target node with id 56690?	`SELECT grp FROM edge WHERE source_id = 56666 AND source_graph = 1349 AND target_id = 56690`	tigerXML_edge
10. What is the name of the type entities belonging to the group tigerXML_node?	`SELECT name FROM graph_type_definition WHERE grp = 'tigerXML_node'`	terminal, nonterminal

Table 3: Initial test questions, the respective gold queries and the gold outputs of the queries.

NL Question	Hardness	Gold Query	Predicted Query SmBop	Predicted Query IRNet
1. What is the 1st word of sentence number 3?	Medium	SELECT word FROM syntax_terminal_nodes WHERE s_num = 3 AND seq = 1	SELECT prosody_nodes.word FROM prosody_nodes WHERE prosody_graph.filesequence = 1 AND syntax_terminal_nodes.word = 3	SELECT T1.word FROM prosody_nodes AS T1 JOIN node AS T3 JOIN is_target_syn_root AS T2 WHERE T2.is_label like 1
2. How many words does sentence 3 have?	Easy	SELECT MAX(seq) FROM syntax_terminal_nodes WHERE s_num = 3	SELECT COUNT(*) FROM prosody_nodes WHERE prosody_nodes.word LIKE '%3%'	SELECT count(*) FROM prosody_nodes AS T1 WHERE T1.word like 1
3. What is the pos of each word in sentence 3?	Medium	SELECT pos FROM syntax_terminal_nodes WHERE s_num = 3 ORDER BY seq	SELECT prosody_nodes.word FROM prosody_nodes WHERE prosody_nodes.ts = 3	SELECT T1.pos FROM syntax_terminal_nodes AS T1 JOIN node AS T3 JOIN prosody_nodes AS T2 WHERE T2.filesequence like 1
4. What is the pos of the first word in sentence 3?	Medium	SELECT pos FROM syntax_terminal_nodes WHERE s_num = 3 AND seq = 1	SELECT syntax_terminal_nodes.pos FROM prosody_nodes WHERE prosody_nodes.word = 3	SELECT T1.pos FROM syntax_terminal_nodes AS T1 JOIN node AS T3 JOIN prosody_nodes AS T2 WHERE T2.word like 1
5. What is the graph id of sentence number 3?	Easy	SELECT graph_id FROM syntax_graph WHERE s_num = 3	SELECT prosody_graph.graph_id FROM prosody_graph WHERE prosody_graph.filesequence = 3	SELECT T1.is_graph_id FROM is_target_syn_root AS T1 WHERE T1.syn_graph_id = 1
6. Which words belonging to the file dlf-nachrichten-200703262000 have no marked tone?	Medium	SELECT word FROM prosody_nodes WHERE tone = 'NONE' AND filesequence = 'dlf-nachrichten-200703262000'	SELECT prosody_nodes.word FROM prosody_nodes WHERE closure.closure_name = 'dlf-nachrichten-200703262000'	SELECT T1.tone FROM prosody_nodes AS T1 WHERE T1.filesequence NOT IN (SELECT T2.node_id FROM node AS T2)
7. What is the information status label of node 900514?	Easy	SELECT is_label FROM is_target_syn_root WHERE syn_root_id = 900614	SELECT is_target_syn_root.is_label FROM node WHERE node.node_id = 900514	SELECT T1.is_label FROM is_target_syn_root AS T1 WHERE T1.is_node_descr = 1
8. What is the type and the group of node 900620?	Medium	SELECT node_type, grp FROM node WHERE node_id = 900620	SELECT node.node_type FROM node WHERE node.node_id = 900620	SELECT T1.node_type, T2.is_node_descr FROM node AS T1 JOIN is_target_syn_root AS T2 WHERE T2.is_node_id = 1
9. What is the group of the edge connecting the source node with id 56666 and source graph 1349 to the target node with id 56690?	Medium	SELECT grp FROM edge WHERE source_id = 56666 AND source_graph = 1349 AND target_id = 56690	SELECT edge.edge_type FROM edge WHERE edge.source_id = 56666 AND edge.target_graph = 56690	SELECT T1.name FROM graph_type_definition AS T1 JOIN syntax_terminal_nodes AS T2 JOIN prosody_nodes AS T3 WHERE T2.node_id = 1 and T3.filesequence > 1
10. What is the name of the type entities belonging to the group tigerXML node?	Easy	SELECT name FROM graph_type_definition WHERE grp = 'tigerXML node'	SELECT graph_type_definition.name FROM graph_type_definition JOIN node ON graph_type_definition.graph_type_def_id = node.node_type WHERE syntax_nonterminal_nodes.tiger_node_id = 'tigerXML_node'	SELECT T1.name FROM graph_type_definition AS T1 JOIN node AS T3 JOIN closure AS T2 WHERE T2.closure_name = 1

Table 4: Summary of Experiment 1. The first column displays the NL question, the second column specifies the hardness level of the gold query (based on the Spider classification), the third column shows the gold query, and the last two columns display the predictions output by the two systems.

9.2 Experiment 2

The second experiment starts from the observations made by analyzing the results of Experiment 1 and is based on the database structure shown in Figure 24. This new version of the database keeps the same primary key and foreign key relations as in Experiment 1 but many column and table names have been changed on the lexical level and are more explicitly indicating their function. This experiment aims at understanding whether the systems are sensitive to lexical changes and possibly perform better when the content of columns and tables is more explicit, for instance when they are expressed with full words instead of abbreviations and when underscores divide more clearly the words. As an example, the table is_graph is rendered as info_status_graph and the s_num column is rendered as sentence_number.

The set of questions is the same used for Experiment 1 and shown in Table 3, with the only difference being that the names of columns and tables are adapted to the new names as described in Figure 24.

9.2.1 Evaluation of Experiment 2

The evaluation methodology is the same outlined for Experiment 1. An overview of the NL questions, the gold and predicted queries, as well as the hardness classification for the current experiment is shown in Table 5.

Starting with the IRNet system, this one does not show a noteworthy difference in performance compared to the results of Experiment 1. Indeed, half the queries produced in Experiment 2 have the same structure and select the same arguments, operators and keywords as in Experiment 1. The ones that show differences instead do not follow a pattern, meaning that some have higher similarity scores in the main SELECT clause but worse scores in the WHERE clause, whereas others show the inverse behavior. What is interesting is that in both experiments the IRNet system predicts the value "1" in all WHERE clauses. As far as the execution match is concerned, the system fails completely to retrieve correct information with most queries not retrieving information at all. For further details, the whole evaluation output of the IRNet system for this experiment is shown in Listing 10 in Appendix B.

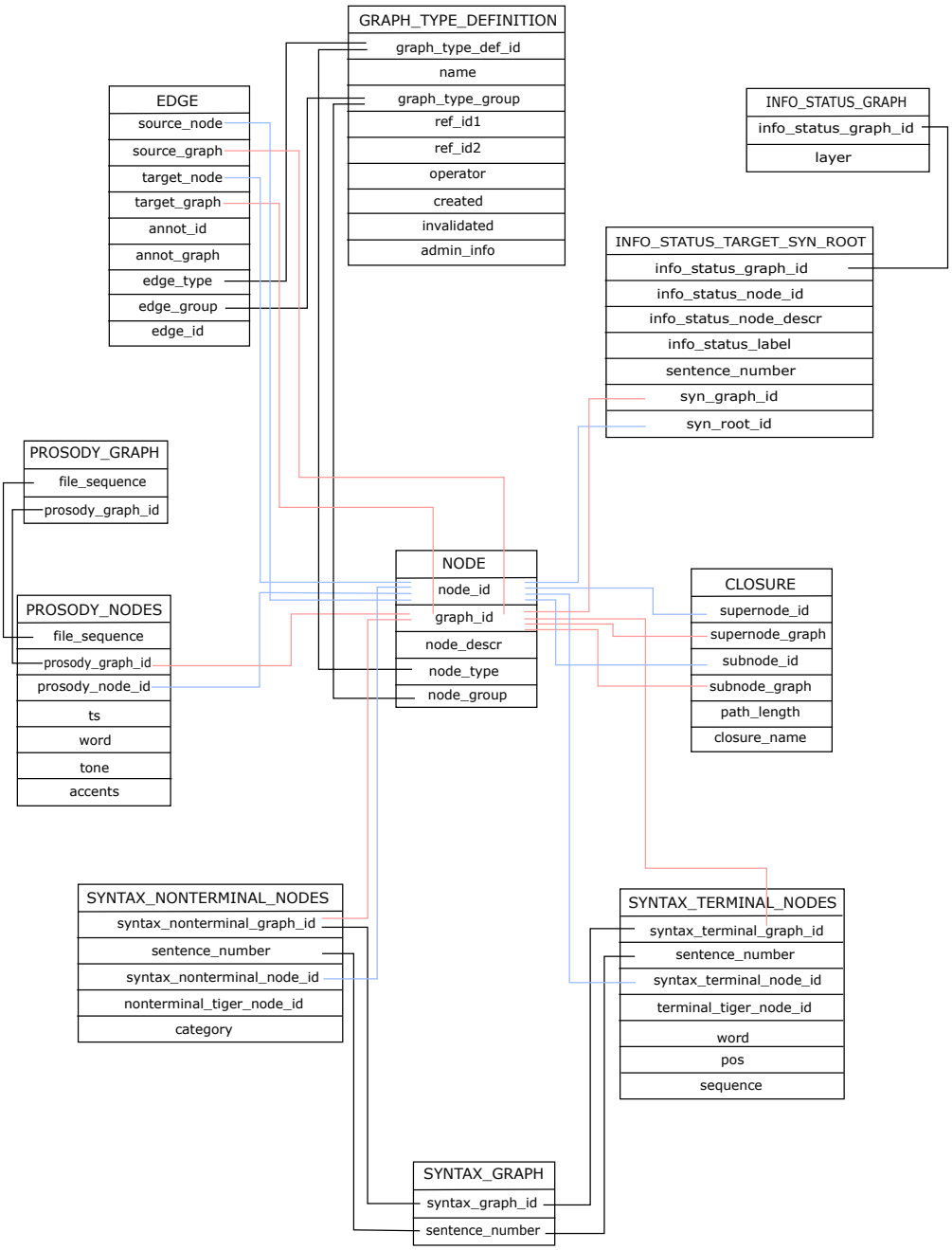


Figure 24: DIRNDL database tables and foreign key relations included in Experiment 2. The red lines show the foreign keys that map to graph_id in the node table; the blue lines show the foreign keys that map to node_id in the node table; the black lines show the foreign keys that map to other entities.

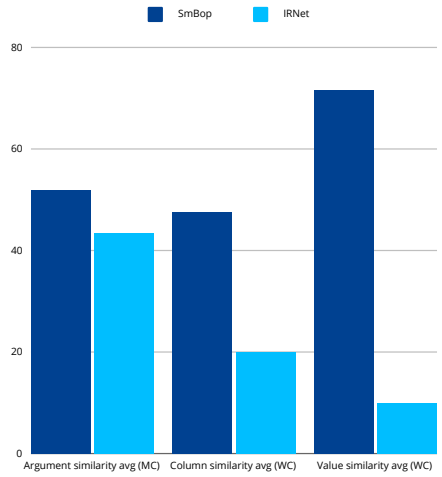


Figure 25: Average percentile scores produced by the systems with regards to argument similarity in the main clause (MC), and column and value similarity in the where clause (WC).

As far as the SmBop system is concerned, a slight improvement can be noticed, with even a perfect match and execution match on query 8 from Table 5. The only query that shows an overall worse similarity score is query 1. Interestingly, the predicted queries 1, 5, 6, and 7 show a higher structure complexity compared to the corresponding predicted queries output in Experiment 1. Nevertheless, apart from query 1, queries number 5, 6, and 7 show improvement in the similarity scores of the WHERE clause. The other queries show overall higher similarity scores, especially on the n-gram level. For questions 3 and 4 the system predicts the same query. Regarding the execution, two queries show particularly interesting insights, namely queries 8 and 10. Query 8 manages to output the correct information, whereas query 10 outputs partially correct information. Indeed, query 10 correctly selects "name" from graph_type_definition and states the WHERE condition that "grp" (rendered as node_group in this experiment) must correspond to "tigerXML_node"; however, it also adds an unnecessary JOIN clause. In the end the query returns 759,472 lines for the column "name", each being either "terminal" or "nonterminal", which is not too different from the gold output, with the only difference being that the gold output would only consist of two lines where one contains the "terminal" name value and

one the “nonterminal” name value. For further details, the whole evaluation output of the SmBop system for this experiment is shown in Listing 9 in Appendix B.

These results overall suggest that the IRNet system performance is only slightly influenced by lexical changes in the database and also that these do not improve the system's performance to a great extent, whereas the SmBop's performance is more positively influenced by such lexical changes. This is also graphically shown in Figure 25, which contains a summary of the experiment results highlighting the main differences between the two systems.

NL Question	Hardness	Gold Query	Predicted Query SmBop	Predicted Query IRNet
1. What is the 1st word of sentence number 3?	Medium	SELECT word FROM syntax_terminal_nodes WHERE sentence_number = 3 AND sequence = 1	SELECT syntax_terminal_nodes.word FROM syntax_terminal_nodes JOIN syntax_graph ON syntax_terminal_nodes.syntax_terminal_graph_id = syntax_graph.sentence_number WHERE info_status_target_syn_root.sentence_number = 3 GROUP BY syntax_terminal_nodes.word	SELECT T1.sentence_number FROM syntax_terminal_nodes AS T1 JOIN node AS T3 JOIN prosody_nodes AS T2 WHERE T2.word = 1
2. How many words does sentence 3 have?	Easy	SELECT MAX(sequence) FROM syntax_terminal_nodes WHERE sentence_num = 3	SELECT COUNT(*) FROM syntax_graph WHERE syntax_graph.sentence_number = 3	SELECT count(*) FROM prosody_nodes AS T1 WHERE T1.word like 1
3. What is the pos of each word in sentence 3?	Medium	SELECT pos FROM syntax_terminal_nodes WHERE sentence_number = 3 ORDER BY sequence	SELECT syntax_terminal_nodes.pos FROM syntax_terminal_nodes WHERE syntax_terminal_nodes.word = 3	SELECT T1.pos FROM syntax_terminal_nodes AS T1 JOIN node AS T3 JOIN prosody_nodes AS T2 WHERE T2.word like 1
4. What is the pos of the first word in sentence 3?	Medium	SELECT pos FROM syntax_terminal_nodes WHERE sentence_number = 3 AND sequence = 1	SELECT syntax_terminal_nodes.pos FROM syntax_terminal_nodes WHERE syntax_terminal_nodes.word = 3	SELECT T1.pos FROM syntax_terminal_nodes AS T1 JOIN node AS T3 JOIN prosody_nodes AS T2 WHERE T2.word like 1
5. What is the graph id of sentence number 3?	Easy	SELECT syntax_graph_id FROM syntax_graph WHERE sentence_number = 3	SELECT node.graph_id FROM syntax_nonterminal_nodes JOIN node ON syntax_nonterminal_nodes.syntax_nonterminal_node_id = node.node_id WHERE info_status_target_syn_root.sentence_number = 3	SELECT T1.graph_id FROM node AS T1 JOIN syntax_terminal_nodes AS T2 WHERE T2.sentence_number = 1
6. Which words belonging to the file dlf-nachrichten-200703262000 have no marked tone?	Medium	SELECT word FROM prosody_nodes WHERE tone = 'NONE' AND file_sequence = 'dlf-nachrichten-200703262000'	SELECT prosody_nodes.accents FROM prosody_nodes WHERE prosody_nodes.file_sequence = 'dlf-nachrichten-200703262000' EXCEPT SELECT prosody_nodes.accents FROM prosody_nodes WHERE prosody_nodes.tone = 'dlf-nachrichten-200703262000'	SELECT T1.tone FROM prosody_nodes AS T1 WHERE T1.file_sequence NOT IN (SELECT T2.node_id FROM node AS T2)
7. What is the information status label of node 900514?	Easy	SELECT info_status_label FROM info_status_target_syn_root WHERE syn_root_id = 900614	SELECT info_status_target_syn_root.info_status_label FROM info_status_target_syn_root JOIN node ON info_status_target_syn_root.syn_graph_id = node.graph_id WHERE node.node_id = 900514	SELECT T1.info_status_label FROM info_status_target_syn_root AS T1 WHERE T1.info_status_node_descr = 1
8. What is the type and the group of node 900620?	Medium	SELECT node_type, node_group FROM node WHERE node_id = 900620	SELECT node.node_type, node.node_group FROM node WHERE node.node_id = '900620'	SELECT T1.node_type, T1.node_group FROM node AS T1 JOIN edge AS T2 WHERE T2.source_node = 1
9. What is the group of the edge connecting the source node with id 56666 and source graph 1349 to the target node with id 56690?	Medium	SELECT edge_group FROM edge WHERE source_node = 56666 AND source_graph = 1349 AND target_node = 56690	SELECT edge.edge_group FROM edge WHERE edge.source_graph = 56666 AND edge.target_graph = 56690	SELECT T1.edge_group FROM edge AS T1 JOIN node AS T2 WHERE T1.target_node = 1 and T1.source_node = 1 and T2.node_id = 1
10. What is the name of the type entities belonging to the group tigerXML node?	Easy	SELECT name FROM graph_type_definition WHERE graph_type_group = 'tigerXML_node'	SELECT graph_type_definition.name FROM graph_type_definition JOIN node ON graph_type_definition.graph_type_def_id = node.node_type WHERE node.node_group = 'tigerXML_node'	SELECT T1.name FROM graph_type_definition AS T1 JOIN node AS T3 JOIN closure AS T2 WHERE T2.closure_name = 1

Table 5: Summary of Experiment 2. The first column displays the NL question, the second column specifies the hardness level of the gold query (based on the Spider classification), the third column shows the gold query, and the last two columns display the predictions output by the two systems.

9.3 Experiment 3

The third and last experiment tests the systems on some queries originally used in the context of the research which led to the paper "Anarchy in the NP. When new nouns get deaccented and given nouns don't" (Riester and Piontek, 2015). Unfortunately the number of original queries which could be retrieved is limited, but it still allows obtaining some insights into the capability of the systems to be deployed in real-word database research settings. The set of questions used together with the respective hardness level, gold query and predicted queries is shown in Table 6. Unlike in the previous experiments, the gold outputs of the queries are omitted here due to space matters. The complexity level of these questions and queries is hard or extra hard, following the hardness definition given in Section 8.

The database structure used here is shown in Figure 26. It differs from the structure used in Experiment 1 only for the presence of two new tables, namely "sentences" and "terminal_and_prosody", which are originally temporary tables used in building sequential queries for the actual questions. The purpose of "sentences" is to list all sentences with each one of them identified by its corresponding s_num, whereas the purpose of "terminal_and_prosody" is to link prosody information for each terminal node. These temporary tables are necessary since the few available queries for this experiment demand information retrieval from them.

9.3.1 Evaluation of Experiment 3

The evaluation methodology is the same outlined for Experiments 1 and 2.

The IRNet system performs extremely poorly in this experiment. Indeed, it does not manage to produce predicted queries for questions number 1, 2, or 4 from Table 6, but only for the question number 3. However, even this only one produced query has a similarity and n-gram overlap score of almost 0.00 when compared to the gold query. Furthermore, the produced query does not run correctly when executed in SQLite as it causes a blockage. For further details, the whole evaluation output is shown in Listing 12 in Appendix B.

On the other hand, the SmBop system performs better than the IRNet, although the results are also not satisfactory. Indeed the similarity and n-gram overlap scores are overall low and the argument scores are never higher than 0.30. The best-performing queries are the number 1 and 4, which are classified as hard queries for their structures, whereas the queries number 2 and 3 are extra hard. An interesting phenomenon can be observed in query number 1, namely here it can be seen that the system fails to unpack the expression 'syntactic_leaf' and only states it as a value in the WHERE clause. This supports the claim from Zhao et al. (2022) that current systems still struggle to capture implicit or domain-specific terms in the NL utterance and fail at mapping them to the correct tables and columns. As far as the execution of the predicted queries is concerned, the first query raises an error related to the INTERSECT operator, whereas the other three queries return no results. For further details, the whole evaluation output is shown in Listing 11 in Appendix B.

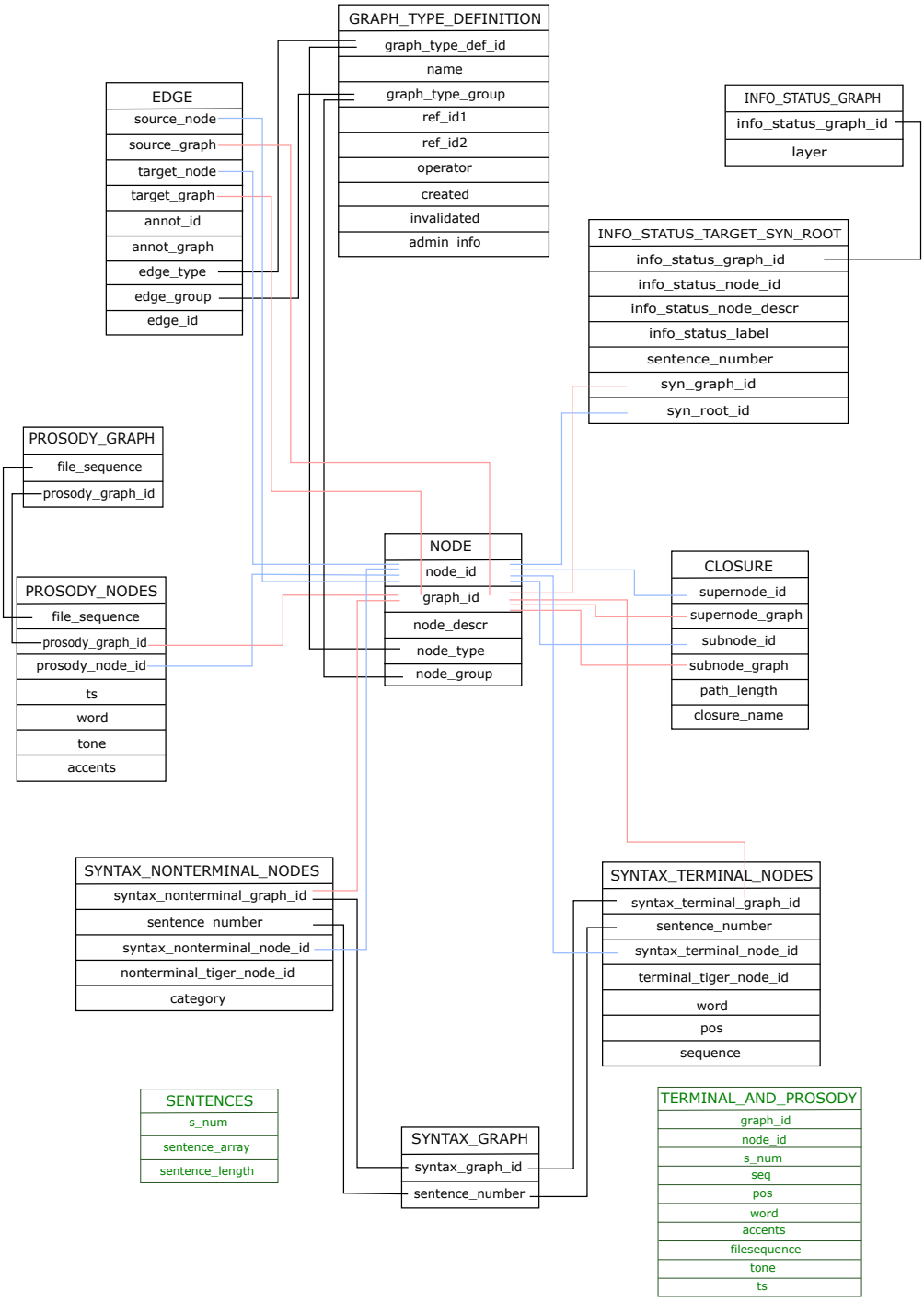


Figure 26: DIRNDL database tables and foreign key relations included in Experiment 3. The green tables are temporary tables.

NL Question	Hardness	Gold Query	Predicted Query SmBop	Predicted Query IRNet
1. Return a list of linked prosody and syntactic leaf nodes together with prosodic features.	Hard	SELECT syntax_terminal_nodes.graph_id, syntax_terminal_nodes.node_id, prosody_nodes.ts, prosody_nodes.word, prosody_nodes.tone, prosody_nodes.accents FROM prosody_nodes JOIN edge ON prosody_nodes.graph_id = edge.source_graph AND prosody_nodes.node_id = edge.source_id JOIN syntax_terminal_nodes ON syntax_terminal_nodes.graph_id = edge.target_graph AND syntax_terminal_nodes.node_id = edge.target_id WHERE edge.grp = 'link_edge'	SELECT node.node_id, prosody_nodes.word FROM prosody_nodes JOIN node ON prosody_nodes.node_id = node.node_id WHERE node.node_type = 'syntactic leaf' INTERSECT SELECT prosody_nodes.word FROM prosody_nodes JOIN node ON prosody_nodes.node_id = node.node_id WHERE node.node_type = 'syntactic leaf'	no output
2. Deliver configurations where the nuclear accent is labeled on the word before the break.	Extra	SELECT graph_id, node_id, terminal_and_prosody.s_num, seq, pos, word, accents, tone, ts, filesequence, sentence_length, sentence_array FROM terminal_and_prosody JOIN sentences ON terminal_and_prosody.s_num = sentences.s_num WHERE accents != '|NONE|' AND tone LIKE '%\\%%' ORDER BY terminal_and_prosody.s_num	SELECT prosody_nodes.word FROM prosody_nodes WHERE prosody_nodes.word LIKE '%nuclear%'	no output
3. Get all DPs without an adjective	Extra	SELECT syntax_nonterminal_nodes.node_id, syntax_nonterminal_nodes.graph_id FROM syntax_nonterminal_nodes WHERE syntax_nonterminal_nodes.cat LIKE 'DP[%' EXCEPT SELECT closure.node_id, closure.node_graph FROM syntax_terminal_nodes JOIN closure ON closure.subnode_id = syntax_terminal_nodes.node_id AND closure.node_graph = syntax_terminal_nodes.graph_id WHERE syntax_terminal_nodes.pos LIKE 'A[%'	SELECT sentences.sentence_array FROM sentences WHERE sentences.sentence_length¡(SELECT MAX(sentences.sentence_length) FROM sentences WHERE sentences.sentence_length LIKE '%DPs%')	SELECT T1.created FROM graph_type_definition AS T1 WHERE T1.graph_type_def_id NOT IN (SELECT T2.source_id FROM edge AS T2 JOIN node AS T4 JOIN prosody_nodes AS T3 WHERE T3.word like 1)
4. Collect the syntactic leaves linked to the prosody nodes.	Hard	SELECT prosody_nodes.filesequence, prosody_nodes.graph_id, prosody_nodes.node_id, prosody_nodes.ts, prosody_nodes.word, prosody_nodes.tone, prosody_nodes.accents, syntax_terminal_nodes.graph_id, syntax_terminal_nodes.node_id FROM syntax_terminal_nodes JOIN edge ON syntax_terminal_nodes.graph_id = edge.target_graph AND syntax_terminal_nodes.node_id = edge.target_id JOIN prosody_nodes ON prosody_nodes.graph_id = edge.source_graph AND prosody_nodes.node_id = edge.source_id WHERE edge.grp = 'link_edge'	SELECT prosody_nodes.word FROM prosody_nodes JOIN syntax_graph ON prosody_nodes.graph_id = syntax_graph.graph_id	no output

Table 6: Summary of Experiment 3. The first column displays the NL question, the second column specifies the hardness level of the gold query (based on the Spider classification), the third column shows the gold query, and the last two columns display the predictions output by the two systems.

10 Results

With regards to the first research question outlined in the introduction, this study has shown that in order to apply text-to-SQL systems to a LAF-format database and in general to a complex relational database, some extensive preprocessing is needed, and most probably the input constraints do not allow to directly pass the whole database as input, which leads the user to the need of extracting a subset of the database structure and relations. Indeed, for the purposes of the current thesis, a precise case study (Riester and Piontek, 2015) has been taken into account, and based on that a subset of tables and relations from the DIRNDL D3DB database has been extracted to create a new schema. In addition to this, the new PostgreSQL schema needed to be converted in the respective SQLite version avoiding information loss, since the two deployed systems are designed to take SQLite files as input.

These steps require users to already possess some knowledge about database design and RDBMS, which goes against the primary reason behind this research, namely to aid the linguistics research and support linguists and in general users who are not specialized in the field of query languages and databases.

In addition to this, converting the NL questions and the data contained in the database schema into the input format required by the systems based on the Spider pipelines is a cumbersome and time-consuming process. With regards to the two systems analyzed here, more preprocessing steps are involved in the IRNet rather than in the SmBop. Moreover, the SmBop system provides a script which allows for direct inference, whereas the IRNet does not. Nevertheless, the SmBop also provides a script for indirect inference, meaning that a NL question together with its gold query must be passed to the system in order to obtain a predicted query, which is however produced only based on the NL question, and not on the gold query. This indirect inference script gives the same outputs as the direct inference one. For this reason it is assumed that the IRNet indirect inference script works analogously to the SmBop one, but in contrast to SmBop, does not allow for a direct inference script. For the purposes of the current thesis, this does not cause problems, since an analysis based on a comparison with the gold queries is essential to effectively

detect the weak and strong points of the systems. However, for a user that needs the support of such systems to retrieve information in a fast and convenient fashion, this is a drawback. Further research could address this with the development of user-friendly interfaces that allow users to directly use existing text-to-SQL systems for inference also based on unseen databases.

With regards to the quality of the systems' output, which is the focus of the second research question, both systems perform quite poorly overall, even when dealing with easy information needs. However, when the names of columns and tables have a lower n-gram overlap among each other, the systems improve their performance. Nevertheless, this means involving further preprocessing before actually being able to use the systems easily.

As far as the evaluation methodology is concerned, the qualitative analysis of the systems' predicted queries is made possible by the custom script developed in the current work, which offers a granular approach to understanding the performance of these systems. Indeed, by breaking down the similarity scores into distinct components, the script provides valuable insights into where the system performs well and where it could be enhanced. Moreover, the inclusion of n-gram overlap as additional evaluation method adds a layer of precision, highlighting also more subtle differences between the gold and predicted queries which might derive from specific lexical features. Such a breakdown can be relevant for fine-tuning text-to-SQL models, as it enables targeted improvements. Ultimately, this script helps to diagnose and address issues in SQL generation, contributing to the development of more accurate and reliable text-to-SQL systems.

Addressing the third research question, the SmBop system has a better performance overall and has demonstrated being more sensitive both to lexical changes in the database schema and to question complexity variations, which might indicate that a neural architecture like the one in SmBop is better and more suitable in the context of the current research. This is not counterintuitive, as SmBop's architecture is crafted to overcome the challenges found in traditional top-down, autoregressive parsers, which is the category which IRNet falls in. The bottom-up, semi-autoregressive strategy allows SmBop to improve efficiency and enhances the

alignment between natural language input and query output. This approach makes SmBop better suited for complex or large-scale semantic parsing tasks, resulting in superior performance compared to IRNet.

More in detail, one key features that is the reason of success of the SmBop system is that unlike traditional top-down approaches that generate the query from the root to the leaves, SmBop starts by generating subtrees and then combines them to form the complete SQL query. Moreover, this system generates multiple components (subtrees) in parallel at each decoding step, which allows for faster processing, and which contrasts with the fully autoregressive models that generate one token or component at a time. Indeed, SmBop maintains a beam of top-K trees (subqueries) at each step and iteratively refines them until the full SQL query is generated. In addition to this, SmBop uses the RAT-SQL encoder, based on relational-aware self-attention, to create joint representations of the natural language query and the database schema. Indeed, the NL utterance is concatenated to a linearized form of the database schema and this joint representation is passed through a stack of transformer layers. The fact that this encoder allows for a joint contextualization improves the model's ability to align natural language with SQL components.

Furthermore, in both systems an intermediate representation that tackles the problem of mismatch between NL and SQL is used, but whereas IRNet uses SemSQL, SmBop uses algebraic trees. This might indicate that relational algebra is a better intermediate representation language when bridging between NL and SQL. This could be related to the fact that relational algebra is a standard query language which is also widely accepted and used in real-world systems as it consists purely of mathematical operators. On the contrary, SemQL is a formal query language, which means that it is a newly created language designed for expressing database queries in a way that is more aligned with logical concepts. The fact that it is a formal human-crafted language might indicate that it is more prone to weaknesses especially in cross-domain settings compared to mathematical operators for which ambiguity is not an issue. This intuition is also supported by Guo et al. (2019), developers and authors of the IRNet system, who state in the respective paper that "designing an effective intermediate representation to bridge NL and SQL is a promising direction

to being there for complex and cross-domain Text-to-SQL" (Guo et al., 2019).

Finally, to answer the last and main research question of this thesis, it is possible to apply text-to-SQL systems on a linguistic relational database like the DIRNDL, but this requires some extensive preprocessing and a partial simplification and/or reduction of the database structure, at least for what concerns the two systems used here. However, the fact that the systems can be applied on such a database does not entail that they perform well on it. Indeed, when the goal is to retrieve research-relevant information and therefore produce complex non-trivial queries, both systems fail.

11 Conclusion and Future Directions

This study, set out to inspect and analyze the applicability of two exemplary text-to-SQL systems on an unseen type of complex linguistic database, has provided interesting insights which allow to state that both the exemplary systems chosen struggle when applied on a complex relational database like the DIRNDL. Nevertheless, SmBop demonstrates better results, indicating that its architecture, based on a bottom-up, semi-autoregressive approach, is more effective than the top-down, autoregressive architecture employed by IRNet. This suggests that SmBop's method of parallel component generation and better representation alignment contributes to its superior performance.

One major drawback of most text-to-SQL systems consists in the fact that they are developed to read only SQLite files as input. For this reason, further work in the realm of text-to-SQL could include developing robust converters able to convert databases developed for a certain RDBMS into the correspondent version in a different RDBMS, in a way that even big databases for instance in PostgreSQL or MySQL can be turned into a SQLite database without relying on extensive work to be manually conducted by the user. Alternatively, new text-to-SQL systems should take into account the existence of various RDBMS and be able to produce queries that adhere to different extensions of the SQL standard, accommodating databases in various formats, and not only SQLite, although the fact that SQLite is file-based clearly makes it an easier deployable tool.

To conclude, generating accurate SQL from NL questions has long been a challenging problem due to the complexities of understanding user requests, comprehending database schemas, and generating correct SQL statements. Earlier text-to-SQL systems have relied on a combination of human engineering and deep neural networks, and more recently also pre-trained language models (PLMs) have demonstrated promising performances. However, as databases grow more complex and user queries become more intricate, the limited comprehension capabilities of PLMs can result in incorrect SQL generation, which leads to the inconvenient need of advanced tailored optimization methods. In this landscape, large language models (LLMs)

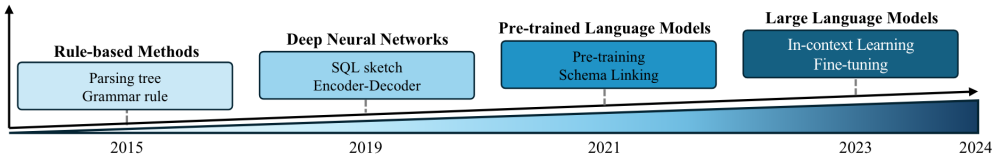


Figure 27: An outline of the evolutionary process of text-to-SQL research from an implementation paradigm perspective, highlighting key stages along with corresponding techniques and representative works (Hong et al., 2024).

have started to show remarkable abilities in natural language understanding, which suggests that LLM-based approaches present potential solutions for advancing text-to-SQL research (Hong et al., 2024) and might tackle the issue related to complex database understanding, which limits the effectiveness of more traditional systems based on human engineering or PLMs, as shown in Figure 27.

References

Till Adams and Klaus Riede. RDBMS - An Introduction to Relational Database Management Systems, January 2002. Digital publication in: Riede, K. (2004): Global Register of Migratory Species - from Global to Regional Scales, on enclosed CD-ROM under: /biblio/PowerUserGuide.pdf.

Sabine Brants, Stefanie Dipper, Peter Eisenberg, Silvia Hansen-Schirra, Esther König, Wolfgang Lezius, Christian Rohrer, George Smith, and Hans Uszkoreit. TIGER: Linguistic Interpretation of a German Corpus. *Research on Language and Computation*, 2(4):597–620, 2004.

Ursin Brunner and Kurt Stockinger. ValueNet: A Natural Language-to-SQL System that Learns from Database Information. *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 2177–2182, April 2021. doi: 10.1109/ICDE51399.2021.00220.

Aljoscha Burchardt, Katrin Erk, Anette Frank, Andrea Kowalski, and Sebastian Pado. SALTO – A Versatile Multi-Level Annotation Tool. In *Proceedings of the 5th International Conference on Language Resources and Evaluation (LREC)*, pages 517–520, Genoa, Italy, 2006.

Pierre-Antoine Champin, Geert-Jan Houben, and Philippe Thiran. Cross: An OWL Wrapper for Reasoning on Relational Databases. volume 4801, pages 502–517, November 2007. ISBN 978-3-540-75562-3. doi: 10.1007/978-3-540-75563-0_34.

Mike Chapple and Sharif Nijim. *CompTIA DataSys+ Study Guide: Exam DS0-001*, pages 41–86. Sybex, November 2023. ISBN 978-1394180059.

Christian Chiarcos, Julia Ritz, and Manfred Stede. By all these lovely tokens... Merging Conflicting Tokenizations. In Manfred Stede, Chu-Ren Huang, Nancy Ide, and Adam Meyers, editors, *Proceedings of the Third Linguistic Annotation Workshop (LAW III)*, pages 35–43, Suntec, Singapore, August 2009. Association for Computational Linguistics. URL `https://aclanthology.org/W09-3005`.

Edgar F. Codd. *A Relational Model of Data for Large Shared Data Banks*, pages 61–98. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. ISBN 978-3-642-48354-7. doi: 10.1007/978-3-642-48354-7_4.

Dick Crouch, Mary Dalrymple, Ron Kaplan, Tracy King, John Maxwell, and Paula Newman. *XLE Documentation*. Palo Alto Research Center, 2011. URL `http://www2.parc.com/isl/groups/nltt/xle/doc/xle_toc.html`.

Chris J. Date and Hugh Darwen. *A Guide to the SQL Standard: A User's Guide to the Standard Relational Language SQL*. Addison-Wesley, 1989.

Naihao Deng, Yulong Chen, and Yue Zhang. Recent Advances in Text-to-SQL: A Survey of What We Have and What We Expect. In Nicoletta Calzolari, Chu-Ren Huang, Hansaem Kim, James Pustejovsky, Leo Wanner, Key-Sun Choi, Pum-Mo Ryu, Hsin-Hsi Chen, Lucia Donatelli, Heng Ji, Sadao Kurohashi, Patrizia Paggio, Nianwen Xue, Seokhwan Kim, Younggyun Hahm, Zhong He, Tony Kyungil Lee, Enrico Santus, Francis Bond, and Seung-Hoon Na, editors, *Proceedings of the 29th International Conference on Computational Linguistics*, pages 2166–2187, Gyeongju, Republic of Korea, October 2022. International Committee on Computational Linguistics. URL `https://aclanthology.org/2022.coling-1.190`.

Korry Douglas and Susan Douglas. PostgreSQL: A Comprehensive Guide to Building, Programming, and Administering PostgreSQL Databases. 2003. URL `https://api.semanticscholar.org/CorpusID:60411823`.

Kerstin Eckart. *Task-based Parser Output Combination: Workflow and Infrastructure*. Dissertation, University of Stuttgart, 2017.

Kerstin Eckart, Kurt Eberle, and Ulrich Heid. An Infrastructure for More Reliable Corpus Analysis. In *Proceedings of the Workshop on Web Services and Processing Pipelines in HLT: Tool Evaluation, LR Production and Validation (LREC'10)*, pages 8–14, Valletta, Malta, May 2010.

Kerstin Eckart, Arndt Riester, and Katrin Schweitzer. A Discourse Information Radio News Database for Linguistic Analysis. In Christian Chiarcos, Sebas-

tian Nordhoff, and Sebastian Hellmann, editors, *Linked Data in Linguistics: Representing and Connecting Language Data and Language Metadat*, pages 65–76. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-28248-5. doi: 10.1007/978-3-642-28249-2_7.

Ahmed Elgohary, Saghar Hosseini, and Ahmed Hassan Awadallah. Speak to your Parser: Interactive Text-to-SQL with Natural Language Feedback. In Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel Tetreault, editors, *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 2065–2077, Online, July 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.187.

Ahmed Elgohary, Christopher Meek, Matthew Richardson, Adam Fourney, Gonzalo Ramos, and Ahmed Hassan Awadallah. NL-EDIT: Correcting Semantic Parse Errors through Natural Language Interaction. In Kristina Toutanova, Anna Rumshisky, Luke Zettlemoyer, Dilek Hakkani-Tur, Iz Beltagy, Steven Bethard, Ryan Cotterell, Tanmoy Chakraborty, and Yichao Zhou, editors, *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 5599–5610, Online, June 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.naacl-main.444.

Ben Eyal, Moran Mahabi, Ophir Haroche, Amir Bachar, and Michael Elhadad. Semantic Decomposition of Question and SQL for Text-to-SQL Parsing. pages 13629–13645, January 2023. doi: 10.18653/v1/2023.findings-emnlp.910.

Catherine Finegan-Dollak, Jonathan K. Kummerfeld, Li Zhang, Karthik Ramanathan, Sesh Sadasivam, Rui Zhang, and Dragomir Radev. Improving Text-to-SQL Evaluation Methodology. In Iryna Gurevych and Yusuke Miyao, editors, *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 351–360, Melbourne, Australia, July 2018. Association for Computational Linguistics. doi: 10.18653/v1/P18-1033.

Martine Grice, Stefan Baumann, and Ralph Benzmüller. GToBI - a phonological

system for the transcription of German intonation. 2000. URL https://api.semanticscholar.org/CorpusID:2788526.

Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. Towards Complex Text-to-SQL in Cross-Domain Database with Intermediate Representation. In Anna Korhonen, David Traum, and Lluís Màrquez, editors, *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4524–4535, Florence, Italy, July 2019. Association for Computational Linguistics. doi: 10.18653/v1/P19-1444.

Moshe Hazoom, Vibhor Malik, and Ben Bogin. Text-to-SQL in the Wild: A Naturally-Occurring Dataset Based on Stack Exchange Data. In Royi Lachmy, Ziyu Yao, Greg Durrett, Milos Gligoric, Junyi Jessy Li, Ray Mooney, Graham Neubig, Yu Su, Huan Sun, and Reut Tsarfaty, editors, *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*, pages 77–87, Online, August 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.nlp4prog-1.9.

Zijin Hong, Zheng Yuan, Qinggang Zhang, Hao Chen, Junnan Dong, Feiran Huang, and Xiao Huang. Next-Generation Database Interfaces: A Survey of LLM-based Text-to-SQL, 2024. URL https://arxiv.org/abs/2406.08426.

Nancy Ide and Laurent Romary. Outline of the International Standard Linguistic Annotation Framework. In *Proceedings of the ACL 2003 Workshop on Linguistic Annotation: Getting the Model Right*, pages 1–5, Sapporo, Japan, July 2003. Association for Computational Linguistics. doi: 10.3115/1119296.1119297. URL https://aclanthology.org/W03-1901.

Nancy Ide and Keith Suderman. The Linguistic Annotation Framework: a standard for annotation interchange and merging. *Lang. Resour. Eval.*, 48(3):395–418, September 2014. ISSN 1574-020X. doi: 10.1007/s10579-014-9268-1. URL https://doi.org/10.1007/s10579-014-9268-1.

Bobby Iliev. *Introduction to SQL*. Bobby Iliev, 2023.

Lan Jiang and Felix Naumann. Holistic primary key and foreign key detection. *Journal of Intelligent Information Systems*, 54:1–23, June 2020. doi: 10.1007/s10844-019-00562-z.

Thanakrit Julavanich and Akiko Aizawa. Measuring Text-to-SQL Semantic Parsing Model on the Question Generalizability. In *Proceedings of the 2022 6th International Conference on Natural Language Processing and Information Retrieval*, NLPIR '22, page 79–83, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450397629. doi: 10.1145/3582768.3582782.

Aishwarya Kamath and Rajarshi Das. A Survey on Semantic Parsing, 2019. URL `https://arxiv.org/abs/1812.00978`.

Ronald M. Kaplan and Joan Bresnan. Lexical-Functional Grammar: A Formal System for Grammatical Representation. In J. Bresnan, editor, *The Mental Representation of Grammatical Relations*, pages 173–281. The MIT Press, Cambridge, MA, USA, 1982. Reprinted in Mary Dalrymple, Ronald M. Kaplan, John Maxwell, and Annie Zaenen, eds., Formal Issues in Lexical-Functional Grammar, 29–130. Stanford: Center for the Study of Language and Information, 1995.

Rohit J. Kate. Transforming Meaning Representation Grammars to Improve Semantic Parsing. In *Proceedings of the Twelfth Conference on Computational Natural Language Learning (CoNLL-2008)*, pages 33–40, Manchester, UK, August 2008. URL `http://www.cs.utexas.edu/users/ai-lab?kate:conll08`.

Manuel Kountz, Ulrich Heid, and Kerstin Eckart. A LAF/GrAF based Encoding Scheme for Underspecified Representations of Syntactic Annotations. In Nicoletta Calzolari (Conference Chair), K. Choukri, B. Maegaard, J. Mariani, J. Odijk, S. Piperidis, and D. Tapias, editors, *Proceedings of the Sixth International Conference on Language Resources and Evaluation (LREC'08)*, pages 2262–2269, Marrakech, Morocco, 2008. European Language Resources Association (ELRA). URL `http://www.lrec-conf.org/proceedings/lrec2008/pdf/569_paper.pdf`.

Jay A. Kreibich. *Using SQLite*. O'Reilly Media, Inc., 1st edition, 2010. ISBN 0596521189.

Kirby Charles Kuznia. Using Language Models to Generate Text-to-SQL Training Data An Approach to Improve Performance of a Text-to-SQL Parser, May 2023. URL `https://keep.lib.asu.edu/items/187426`.

Esther König, Wolfgang Lezius, and Holger Voormann. *TIGERSearch 2.1 User's Manual. Chapter V - The TIGER-XML Treebank Encoding Format.* Universität Stuttgart, Germany, 2003.

Hagen Langer. Syntax und Parsing. In K.-U. Carstensen, C. Ebert, C. Endriss, S. Jekat, R. Klabunde, and H. Langer, editors, *Computerlinguistik und Sprachtechnologie*, chapter 3.4, pages 232–275. Elsevier, München, Germany, second edition, 2004.

Chia-Hsuan Lee, Oleksandr Polozov, and Matthew Richardson. KaggleDBQA: Realistic Evaluation of Text-to-SQL Parsers. In Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli, editors, *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 2261–2273, Online, August 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.acl-long.176.

Wenqiang Lei, Weixin Wang, Zhixin Ma, Tian Gan, Wei Lu, Min-Yen Kan, and Tat-Seng Chua. Re-examining the Role of Schema Linking in Text-to-SQL. In Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6943–6954, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.564.

Kevin Lin, Ben Bogin, Mark Neumann, Jonathan Berant, and Matt Gardner. Grammar-based Neural Text-to-Sql Generation, May 2019.

Qi Liu, Zihuiwen Ye, Tao Yu, Linfeng Song, and Phil Blunsom. Augmenting Multi-Turn Text-to-SQL Datasets with Self-Play. In Yoav Goldberg, Zornitsa Kozareva,

and Yue Zhang, editors, *Findings of the Association for Computational Linguistics: EMNLP 2022*, pages 5608–5620, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.findings-emnlp.411.

Xiping Liu and Zhao Tan. Divide and Prompt: Chain of Thought Prompting for Text-to-SQL, 2023.

Jörg Mayer. Transcription of German Intonation. The Stuttgart System, 1995. URL `https://www.ims.uni-stuttgart.de/institut/arbeitsgruppen/ehemalig/ep-dogil/joerg/labman/STGTsystem.html`.

Jörg Peters. Phonological and semantic aspects of german intonation. *Linguistik Online*, 88(1), January 2018. doi: 10.13092/lo.88.4191. URL `https://bop.unibe.ch/linguistik-online/article/view/4191`.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the Limits of Transfer Learning with a Unified Text-to-text Transformer. *J. Mach. Learn. Res.*, 21(1), January 2020. ISSN 1532–4435.

Arndt Riester and Stefan Baumann. The Reflex Scheme - Annotation Guidelines, 2017. doi: 10.18419/opus-9011.

Arndt Riester and Jörn Piontek. Anarchy in the NP. When new nouns get deaccented and given nouns don't. *Lingua*, 165:230–253, 2015. ISSN 0024-3841. doi: https://doi.org/10.1016/j.lingua.2015.03.006.

Christian Rohrer and Martin Forst. Improving coverage and parsing quality of a large-scale LFG for German. In N. Calzolari, K. Choukri, A. Gangemi, B. Maegaard, J. Mariani, J. Odijk, and D. Tapias, editors, *Proceedings of the Fifth International Conference on Language Resources and Evaluation (LREC'06)*, pages 2206–2211, Genoa, Italy, May 2006. European Language Resources Association (ELRA). URL `http://www.lrec-conf.org/proceedings/lrec2006/pdf/99_pdf.pdf`.

Ohad Rubin and Jonathan Berant. SmBoP: Semi-autoregressive Bottom-up Semantic Parsing. In Kristina Toutanova, Anna Rumshisky, Luke Zettlemoyer, Dilek Hakkani-Tur, Iz Beltagy, Steven Bethard, Ryan Cotterell, Tanmoy Chakraborty, and Yichao Zhou, editors, *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 311–324, Online, June 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.naacl-main.29.

Juan Sequeda, Syed Tirmizi, Oscar Corcho, and Daniel Miranker. Survey of directly mapping SQL databases to the Semantic Web. *Knowledge Eng. Review*, 26:445–486, December 2011. doi: 10.1017/S0269888911000208.

Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-Attention with Relative Position Representations. In Marilyn Walker, Heng Ji, and Amanda Stent, editors, *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 464–468, New Orleans, Louisiana, June 2018. Association for Computational Linguistics. doi: 10.18653/v1/N18-2074.

Tianze Shi, Chen Zhao, Jordan Boyd-Graber, Hal Daumé III, and Lillian Lee. On the Potential of Lexico-logical Alignments for Semantic Parsing to SQL Queries. In Trevor Cohn, Yulan He, and Yang Liu, editors, *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1849–1864, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.167.

Yasin Silva, Isadora Almeida, and Michell Queiroz. SQL: From Traditional Databases to Big Data. pages 413–418, February 2016. doi: 10.1145/2839509.2844560.

Manfred Stede and Chu-Ren Huang. Inter-operability and reusability: The science of annotation. *Language Resources and Evaluation*, 46:91–94, March 2012. doi: 10.1007/s10579-011-9164-x.

Toni Taipalus. Database management system performance comparisons: A systematic literature review. *Journal of Systems and Software*, 208:111872, October 2023. doi: 10.1016/j.jss.2023.111872.

Hans Uszkoreit, George Smith, Anne Schwartz, Bettina Schrader, Marco Rower, Marcus Pußel, Cordula Preis, Oliver Plaehn, Lukas Michelbacher, Robert Langner, Carolin Kirstein, Juliane Janitzek, Silvia Hansen, Peter Eisenberg, Stefanie Dipper, Vera Demberg, Thorsten Brants, Sabine Brants, Tobias Bracht, and Hagen Hirschmann. TIGER Annotationsschema, July 2003.

Michael Wagner. Givenness and locality. *Semantics and Linguistic Theory*, 16, August 2006. doi: 10.3765/salt.v0i0.2938.

Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers. pages 7567–7578, January 2020a. doi: 10.18653/v1/2020.acl-main.677.

Bailin Wang, Mirella Lapata, and Ivan Titov. Meta-Learning for Domain Generalization in Semantic Parsing. In Kristina Toutanova, Anna Rumshisky, Luke Zettlemoyer, Dilek Hakkani-Tur, Iz Beltagy, Steven Bethard, Ryan Cotterell, Tanmoy Chakraborty, and Yichao Zhou, editors, *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 366–379, Online, June 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.naacl-main.33.

Ping Wang, Tian Shi, and Chandan K. Reddy. Text-to-SQL Generation for Question Answering on Electronic Medical Records. In *Proceedings of The Web Conference 2020*, WWW '20, page 350–361, New York, NY, USA, 2020b. Association for Computing Machinery. ISBN 9781450370233. doi: 10.1145/3366423.3380120.

John C. Worsley and Joshua D. Drake. *Practical PostgreSQL*. O'Reilly & Associates, Inc., USA, 1st edition, 2001. ISBN 1565928466.

Xiaojun Xu, Chang Liu, and Dawn Song. SQLNet: Generating Structured Queries From Natural Language Without Reinforcement Learning. November 2017. URL https://doi.org/10.48550/arXiv.1711.04436.

Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. SQLizer: query synthesis from natural language. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017. doi: 10.1145/3133887.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. In Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii, editors, *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921, Brussels, Belgium, October-November 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-1425.

Tao Yu, Chien-Sheng Wu, Xi Victoria Lin, Bailin Wang, Yi Chern Tan, Xinyi Yang, Dragomir Radev, Richard Socher, and Caiming Xiong. GraPPa: Grammar-Augmented Pre-Training for Table Semantic Parsing, 2020. URL https://doi.org/10.48550/arXiv.2009.13845.

Rui Zhang, Tao Yu, Heyang Er, Sungrok Shim, Eric Xue, Xi Victoria Lin, Tianze Shi, Caiming Xiong, Richard Socher, and Dragomir Radev. Editing-Based SQL Query Generation for Cross-Domain Context-Dependent Questions. In Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaoju Wan, editors, *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 5338–5349, Hong Kong, China, November 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-1537.

Yi Zhang, Jan Deriu, George Katsogiannis-Meimarakis, Catherine Kosten, Georgia Koutrika, and Kurt Stockinger. Sciencebenchmark: A complex real-world benchmark for evaluating natural language to sql systems. *Proc. VLDB Endow.*, 17

(4):685–698, March 2024. ISSN 2150-8097. doi: 10.14778/3636218.3636225. URL https://doi.org/10.14778/3636218.3636225.

Chen Zhao, Yu Su, Adam Pauls, and Emmanouil Antonios Platanios. Bridging the Generalization Gap in Text-to-SQL Parsing with Schema Expansion. In Smaranda Muresan, Preslav Nakov, and Aline Villavicencio, editors, *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5568–5578, Dublin, Ireland, May 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.acl-long.381.

Victor Zhong, Caiming Xiong, and Richard Socher. Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning. November 2017. URL https://doi.org/10.48550/arXiv.1709.00103.

A Custom Evaluation Script

```
import sqlparse
from sqlparse.sql import Identifier, Token, Where, Comparison, Parenthesis, Function, IdentifierList, Statement
from sqlparse.tokens import Keyword, DML, Name, Punctuation, Whitespace
from collections import Counter

# Define constants for various SQL components
CLAUSE_KEYWORDS = {'SELECT', 'FROM', 'WHERE', 'GROUP BY', 'ORDER BY', 'LIMIT', 'INTERSECT', 'UNION', 'EXCEPT'}
JOIN_KEYWORDS = {'JOIN', 'ON', 'AS'}
AGGREGATE_FUNCTIONS = {'AVG', 'COUNT', 'DISTINCT', 'MAX', 'MIN', 'SUM'}
LOGICAL_OPERATORS = {'ALL', 'AND', 'NOT', 'IN', 'NOT IN', 'ANY', 'BETWEEN', 'OR', 'EXISTS', 'LIKE', 'SOME', '=', '>', '<'}
WHERE_OPS = {'NOT', 'BETWEEN', '=', '>', '<', '>=', '<=', '!=', 'IN', 'LIKE', 'IS', 'EXISTS'}
UNIT_OPS = {'NONE', '-', '+', '*', '/'}
AGG_OPS = {'NONE', 'MAX', 'MIN', 'COUNT', 'SUM', 'AVG'}
COND_OPS = {'AND', 'OR'}
SQL_OPS = {'INTERSECT', 'UNION', 'EXCEPT'}
ORDER_OPS = {'DESC', 'ASC'}

def extract_clause_components(query):
    """
    Extracts the clause keywords and arguments from a SQL query, including the WHERE clause and functions like MAX and COUNT.
    """
    parsed = sqlparse.parse(query)
    if not parsed:
        return [], [], [], [], [], False

    stmt = parsed[0]
    keywords = set()
    arguments = set()
    operators = set()
    where_columns = set()
    where_operators = set()
    where_values = set()
    nested_query_present = False

    def extract_function_arguments(token):
        """
        Extract arguments within SQL functions like MAX and COUNT.
        """
        if isinstance(token, Function):
            function_name = token.get_real_name().upper()
            if function_name in AGGREGATE_FUNCTIONS:
```

```
                keywords.add(function_name)
                for sub_token in token.tokens:
                    if isinstance(sub_token, Parenthesis):
                        for inside_token in sub_token.tokens:
                            if isinstance(inside_token, Identifier) or inside_token.ttype in (Name,)
    :
                                arguments.add(inside_token.get_real_name())
                            elif inside_token.ttype == Punctuation and inside_token.value == '*':
                                arguments.add('*')

    def normalize_column(column):
        """
        Normalizes the column by removing any table aliases or prefixes.
        """
        return column.split('.')[-1] if column else column

    def extract_where_clause(where_token):
        """
        Recursively extract components from the WHERE clause, including nested queries and operators
    .
        """
        nonlocal nested_query_present
        keywords.add('WHERE')  # Add WHERE keyword when encountering a WHERE clause

        for sub_token in where_token.tokens:
            if isinstance(sub_token, Where):
                extract_where_clause(sub_token)
            elif isinstance(sub_token, Comparison):
                left, operator, right = sub_token.left, sub_token.token_next(0)[1], sub_token.right
                if isinstance(left, Identifier):
                    where_columns.add(normalize_column(left.get_real_name()))
                if operator and operator.value.upper() in LOGICAL_OPERATORS:
                    where_operators.add(operator.value.upper())
                if operator and operator.value.upper() == '!=':
                    where_operators.add('!=')
                if isinstance(right, Token):
                    where_values.add(right.value.strip('\'"'))
            elif sub_token.ttype == Keyword and sub_token.value.upper() == 'NOT':
                if sub_token.token_next(1)[1].value.upper() == 'IN':
                    where_operators.add('NOT IN')
            elif isinstance(sub_token, Parenthesis):
                # Check for nested SELECT statements
                nested_query_present = any(isinstance(t, Where) for t in sub_token.tokens)
            elif sub_token.ttype in {Keyword, Punctuation} and sub_token.value.upper() in
    LOGICAL_OPERATORS:
                where_operators.add(sub_token.value.upper())
            elif sub_token.ttype in {Keyword, Punctuation} and sub_token.value.upper() == '!=':
```

```
                where_operators.add('!=')

    def extract_group_by_clause(stmt):
        """
        Extract components from the GROUP BY clause.
        """
        group_by_keywords = {'GROUP BY'}
        for token in stmt.tokens:
            if token.ttype in {Keyword} and token.value.upper() in group_by_keywords:
                keywords.add('GROUP BY')
                _, next_token = stmt.token_next(stmt.token_index(token))
                if isinstance(next_token, IdentifierList):
                    for identifier in next_token.get_identifiers():
                        arguments.add(normalize_column(identifier.get_real_name()))
                elif isinstance(next_token, Identifier):
                    arguments.add(normalize_column(next_token.get_real_name()))

    def extract_join_clause(stmt):
        """
        Extract components from the JOIN clause.
        """
        join_keywords = {'JOIN', 'ON'}
        tokens = stmt.tokens
        for i, token in enumerate(tokens):
            if token.ttype in {Keyword} and token.value.upper() in join_keywords:
                keywords.add(token.value.upper())
                _, next_token = stmt.token_next(i)
                if isinstance(next_token, IdentifierList):
                    for identifier in next_token.get_identifiers():
                        arguments.add(normalize_column(identifier.get_real_name()))
                elif isinstance(next_token, Identifier):
                    arguments.add(normalize_column(next_token.get_real_name()))
                # Capture the columns used in the ON clause
                if token.value.upper() == 'ON':
                    for j in range(i + 1, len(tokens)):
                        sub_token = tokens[j]
                        if isinstance(sub_token, Comparison):
                            left, operator, right = sub_token.left, sub_token.token_next(0)[1],
     sub_token.right
                            if isinstance(left, Identifier):
                                arguments.add(normalize_column(left.get_real_name()))
                            if operator and operator.value.upper() in LOGICAL_OPERATORS:
                                operators.add(operator.value.upper())
                            if isinstance(right, Identifier):
                                arguments.add(normalize_column(right.get_real_name()))
                        elif sub_token.ttype == Keyword and sub_token.value.upper() == 'AND':
                            # Move to the next token to check for additional conditions
```

```
                        continue
                    elif sub_token.ttype in {Keyword, Punctuation} and sub_token.value.upper()
 in LOGICAL_OPERATORS:
                        operators.add(sub_token.value.upper())

def extract_exceptions(stmt):
    """
    Extract components from the EXCEPT clause.
    """
    exception_keywords = {'EXCEPT'}
    for token in stmt.tokens:
        if token.ttype in {Keyword} and token.value.upper() in exception_keywords:
            keywords.add('EXCEPT')
            _, next_token = stmt.token_next(stmt.token_index(token))
            if isinstance(next_token, Parenthesis):
                for sub_token in next_token.tokens:
                    if isinstance(sub_token, Statement):
                        nested_keywords, nested_arguments, nested_where_columns,
 nested_where_operators, nested_where_values, nested_nested_query = extract_clause_components(
 sub_token.value)
                        keywords.update(nested_keywords)
                        arguments.update(nested_arguments)
                        where_columns.update(nested_where_columns)
                        where_operators.update(nested_where_operators)
                        where_values.update(nested_where_values)
                        nested_query_present = nested_query_present or nested_nested_query

for token in stmt.tokens:
    if token.ttype in {Keyword, DML} and token.value.upper() in CLAUSE_KEYWORDS:
        keywords.add(token.value.upper())
    elif isinstance(token, Where):
        # Capture arguments within the WHERE clause
        extract_where_clause(token)
    elif isinstance(token, Identifier) or token.ttype in (Name,):
        normalized_token = normalize_column(token.get_real_name())
        arguments.add(normalized_token)
    elif isinstance(token, IdentifierList):
        for identifier in token.get_identifiers():
            normalized_identifier = normalize_column(identifier.get_real_name())
            arguments.add(normalized_identifier)
    elif isinstance(token, Function):
        extract_function_arguments(token)
    elif token.ttype in {Keyword, Punctuation} and token.value.upper() in LOGICAL_OPERATORS:
        operators.add(token.value.upper())
    elif token.ttype == Keyword and token.value.upper() in SQL_OPS:
        keywords.add(token.value.upper())

```

```
    # Hardcode inclusion of INTERSECT
    if 'INTERSECT' in query.upper():
        keywords.add('INTERSECT')

    # Extract additional clauses
    extract_group_by_clause(stmt)
    extract_join_clause(stmt)
    extract_exceptions(stmt)

    return keywords, arguments, operators, where_columns, where_operators, where_values,
     nested_query_present

def normalize_values(set1, set2):
    """
    Normalizes values by removing single and double quotes.
    """
    def remove_quotes(value):
        return value.replace("'", "").replace('"', "")

    normalized_set1 = {remove_quotes(item) for item in set1}
    normalized_set2 = {remove_quotes(item) for item in set2}

    return normalized_set1, normalized_set2

def calculate_similarity(set1, set2):
    """
    Calculates similarity between two sets as the ratio of the intersection to the union.
    """
    set1, set2 = normalize_values(set1, set2)

    if not set1 and not set2:
        return 1.0  # Both sets are empty, consider them as identical

    intersection = set1.intersection(set2)
    union = set1.union(set2)

    return len(intersection) / len(union)

def calculate_ngram_overlap(set1, set2, n=3):
    """
    Calculates n-gram overlap between two sets.
    """
    set1, set2 = normalize_values(set1, set2)

    if not set1 and not set2:
        return 1.0  # Both sets are empty, consider them as identical

```

```
    def get_ngrams(s, n):
        return [s[i:i+n] for i in range(len(s)-n+1)]

    set1_ngrams = Counter()
    set2_ngrams = Counter()

    for item in set1:
        set1_ngrams.update(get_ngrams(item, n))

    for item in set2:
        set2_ngrams.update(get_ngrams(item, n))

    intersection = set1_ngrams & set2_ngrams
    union = set1_ngrams | set2_ngrams

    if not union:
        return 0.0

    return sum(intersection.values()) / sum(union.values())

def evaluate_similarity(gold_file, predicted_file):
    """
    Evaluates the similarity between gold and predicted SQL queries based on keywords and arguments.
    """
    with open(gold_file, 'r') as gf, open(predicted_file, 'r') as pf:
        gold_queries = gf.readlines()
        predicted_queries = pf.readlines()

    if len(gold_queries) != len(predicted_queries):
        print("Mismatch in number of queries between gold and predicted files.")
        return

    for i, (gold_query, predicted_query) in enumerate(zip(gold_queries, predicted_queries)):
        gold_keywords, gold_arguments, gold_operators, gold_where_columns, gold_where_operators,
     gold_where_values, gold_nested_query = extract_clause_components(gold_query.strip())
        predicted_keywords, predicted_arguments, predicted_operators, predicted_where_columns,
     predicted_where_operators, predicted_where_values, predicted_nested_query =
     extract_clause_components(predicted_query.strip())

        keyword_similarity = calculate_similarity(gold_keywords, predicted_keywords)
        argument_similarity = calculate_similarity(gold_arguments, predicted_arguments)
        operator_similarity = calculate_similarity(gold_operators, predicted_operators)
        column_similarity = calculate_similarity(gold_where_columns, predicted_where_columns)
        where_operator_similarity = calculate_similarity(gold_where_operators,
     predicted_where_operators)
        value_similarity = calculate_similarity(gold_where_values, predicted_where_values)

```

```
        keyword_ngram_overlap = calculate_ngram_overlap(gold_keywords, predicted_keywords)
        argument_ngram_overlap = calculate_ngram_overlap(gold_arguments, predicted_arguments)
        operator_ngram_overlap = calculate_ngram_overlap(gold_operators, predicted_operators)
        column_ngram_overlap = calculate_ngram_overlap(gold_where_columns, predicted_where_columns)
        where_operator_ngram_overlap = calculate_ngram_overlap(gold_where_operators,
    predicted_where_operators)
        value_ngram_overlap = calculate_ngram_overlap(gold_where_values, predicted_where_values)

        print(f"Query {i+1} similarity:")
        print(f"  Keyword similarity: {keyword_similarity:.2f}")
        print(f"  Keyword n-gram overlap: {keyword_ngram_overlap:.2f}")
        print(f"  Argument similarity: {argument_similarity:.2f}")
        print(f"  Argument n-gram overlap: {argument_ngram_overlap:.2f}")
        print(f"  Operator similarity: {operator_similarity:.2f}")
        print(f"  Operator n-gram overlap: {operator_ngram_overlap:.2f}")
        print(f"    Gold keywords: {gold_keywords}")
        print(f"    Predicted keywords: {predicted_keywords}")
        print(f"    Gold arguments: {gold_arguments}")
        print(f"    Predicted arguments: {predicted_arguments}")
        print(f"    Gold operators: {gold_operators if gold_operators else 'None'}")
        print(f"    Predicted operators: {predicted_operators if predicted_operators else 'None'}")
        print(f"  WHERE clause component similarities:")
        print(f"    Column similarity: {column_similarity:.2f}")
        print(f"    Column n-gram overlap: {column_ngram_overlap:.2f}")
        print(f"    Operator similarity: {where_operator_similarity:.2f}")
        print(f"    Operator n-gram overlap: {where_operator_ngram_overlap:.2f}")
        print(f"    Value similarity: {value_similarity:.2f}")
        print(f"    Value n-gram overlap: {value_ngram_overlap:.2f}")
        print(f"      Gold WHERE columns: {gold_where_columns}")
        print(f"      Predicted WHERE columns: {predicted_where_columns}")
        print(f"      Gold WHERE operators: {gold_where_operators if gold_where_operators else 'None
    '}")
        print(f"      Predicted WHERE operators: {predicted_where_operators if
    predicted_where_operators else 'None'}")
        print(f"      Gold WHERE values: {gold_where_values}")
        print(f"      Predicted WHERE values: {predicted_where_values}")

if __name__ == "__main__":
    gold_file = 'gold_queries.sql'
    predicted_file = 'predicted_queries.sql'
    evaluate_similarity(gold_file, predicted_file)
```

custom_eval_script.py

B Custom Evaluation Outputs

```
Query 1 similarity:
  Keyword similarity: 1.00
  Keyword n-gram overlap: 1.00
  Argument similarity: 0.33
  Argument n-gram overlap: 0.21
  Operator similarity: 1.00
  Operator n-gram overlap: 1.00
    Gold keywords: {'SELECT', 'WHERE', 'FROM'}
    Predicted keywords: {'SELECT', 'WHERE', 'FROM'}
    Gold arguments: {'word', 'syntax_terminal_nodes'}
    Predicted arguments: {'word', 'prosody_nodes'}
    Gold operators: None
    Predicted operators: None
  WHERE clause component similarities:
    Column similarity: 0.00
    Column n-gram overlap: 0.07
    Operator similarity: 1.00
    Operator n-gram overlap: 1.00
    Value similarity: 1.00
    Value n-gram overlap: 0.00
      Gold WHERE columns: {'seq', 's_num'}
      Predicted WHERE columns: {'word', 'filesequence'}
      Gold WHERE operators: {'=', 'AND'}
      Predicted WHERE operators: {'=', 'AND'}
      Gold WHERE values: {'3', '1'}
      Predicted WHERE values: {'1', '3'}
Query 2 similarity:
  Keyword similarity: 0.60
  Keyword n-gram overlap: 0.69
  Argument similarity: 0.00
  Argument n-gram overlap: 0.15
  Operator similarity: 1.00
  Operator n-gram overlap: 1.00
    Gold keywords: {'SELECT', 'WHERE', 'MAX', 'FROM'}
    Predicted keywords: {'SELECT', 'WHERE', 'FROM', 'COUNT'}
    Gold arguments: {'seq', 'syntax_terminal_nodes'}
    Predicted arguments: {'prosody_nodes'}
    Gold operators: None
    Predicted operators: None
  WHERE clause component similarities:
    Column similarity: 0.00
    Column n-gram overlap: 0.00
    Operator similarity: 0.00
    Operator n-gram overlap: 0.00
    Value similarity: 0.00
```

```
    Value n-gram overlap: 0.00
      Gold WHERE columns: {'s_num'}
      Predicted WHERE columns: {'word'}
      Gold WHERE operators: {'='}
      Predicted WHERE operators: {'LIKE'}
      Gold WHERE values: {'3'}
      Predicted WHERE values: {'%3%'}
Query 3 similarity:
  Keyword similarity: 0.75
  Keyword n-gram overlap: 0.60
  Argument similarity: 0.00
  Argument n-gram overlap: 0.13
  Operator similarity: 1.00
  Operator n-gram overlap: 1.00
    Gold keywords: {'SELECT', 'WHERE', 'ORDER BY', 'FROM'}
    Predicted keywords: {'SELECT', 'WHERE', 'FROM'}
    Gold arguments: {'seq', 'pos', 'syntax_terminal_nodes'}
    Predicted arguments: {'word', 'prosody_nodes'}
    Gold operators: None
    Predicted operators: None
  WHERE clause component similarities:
    Column similarity: 0.00
    Column n-gram overlap: 0.00
    Operator similarity: 1.00
    Operator n-gram overlap: 0.00
    Value similarity: 1.00
    Value n-gram overlap: 0.00
      Gold WHERE columns: {'s_num'}
      Predicted WHERE columns: {'ts'}
      Gold WHERE operators: {'='}
      Predicted WHERE operators: {'='}
      Gold WHERE values: {'3'}
      Predicted WHERE values: {'3'}
Query 4 similarity:
  Keyword similarity: 1.00
  Keyword n-gram overlap: 1.00
  Argument similarity: 0.33
  Argument n-gram overlap: 0.19
  Operator similarity: 1.00
  Operator n-gram overlap: 1.00
    Gold keywords: {'SELECT', 'WHERE', 'FROM'}
    Predicted keywords: {'SELECT', 'WHERE', 'FROM'}
    Gold arguments: {'pos', 'syntax_terminal_nodes'}
    Predicted arguments: {'pos', 'prosody_nodes'}
    Gold operators: None
    Predicted operators: None
  WHERE clause component similarities:
```

```
    Column similarity: 0.00
    Column n-gram overlap: 0.00
    Operator similarity: 0.50
    Operator n-gram overlap: 0.00
    Value similarity: 0.50
    Value n-gram overlap: 0.00
      Gold WHERE columns: {'seq', 's_num'}
      Predicted WHERE columns: {'word'}
      Gold WHERE operators: {'=', 'AND'}
      Predicted WHERE operators: {'='}
      Gold WHERE values: {'3', '1'}
      Predicted WHERE values: {'3'}
Query 5 similarity:
  Keyword similarity: 1.00
  Keyword n-gram overlap: 1.00
  Argument similarity: 0.33
  Argument n-gram overlap: 0.43
  Operator similarity: 1.00
  Operator n-gram overlap: 1.00
    Gold keywords: {'SELECT', 'WHERE', 'FROM'}
    Predicted keywords: {'SELECT', 'WHERE', 'FROM'}
    Gold arguments: {'syntax_graph', 'graph_id'}
    Predicted arguments: {'prosody_graph', 'graph_id'}
    Gold operators: None
    Predicted operators: None
  WHERE clause component similarities:
    Column similarity: 0.00
    Column n-gram overlap: 0.00
    Operator similarity: 1.00
    Operator n-gram overlap: 0.00
    Value similarity: 1.00
    Value n-gram overlap: 0.00
      Gold WHERE columns: {'s_num'}
      Predicted WHERE columns: {'filesequence'}
      Gold WHERE operators: {'='}
      Predicted WHERE operators: {'='}
      Gold WHERE values: {'3'}
      Predicted WHERE values: {'3'}
Query 6 similarity:
  Keyword similarity: 1.00
  Keyword n-gram overlap: 1.00
  Argument similarity: 1.00
  Argument n-gram overlap: 1.00
  Operator similarity: 1.00
  Operator n-gram overlap: 1.00
    Gold keywords: {'SELECT', 'WHERE', 'FROM'}
    Predicted keywords: {'SELECT', 'WHERE', 'FROM'}
```

```
    Gold arguments: {'word', 'prosody_nodes'}
    Predicted arguments: {'word', 'prosody_nodes'}
    Gold operators: None
    Predicted operators: None
  WHERE clause component similarities:
    Column similarity: 0.00
    Column n-gram overlap: 0.00
    Operator similarity: 0.50
    Operator n-gram overlap: 0.00
    Value similarity: 0.50
    Value n-gram overlap: 0.93
      Gold WHERE columns: {'filesequence', 'tone'}
      Predicted WHERE columns: {'closure_name'}
      Gold WHERE operators: {'=', 'AND'}
      Predicted WHERE operators: {'='}
      Gold WHERE values: {'dlf-nachrichten-200703262000', 'NONE'}
      Predicted WHERE values: {'dlf-nachrichten-200703262000'}
Query 7 similarity:
  Keyword similarity: 1.00
  Keyword n-gram overlap: 1.00
  Argument similarity: 0.33
  Argument n-gram overlap: 0.25
  Operator similarity: 1.00
  Operator n-gram overlap: 1.00
    Gold keywords: {'SELECT', 'WHERE', 'FROM'}
    Predicted keywords: {'SELECT', 'WHERE', 'FROM'}
    Gold arguments: {'is_target_syn_root', 'is_label'}
    Predicted arguments: {'node', 'is_label'}
    Gold operators: None
    Predicted operators: None
  WHERE clause component similarities:
    Column similarity: 0.00
    Column n-gram overlap: 0.08
    Operator similarity: 1.00
    Operator n-gram overlap: 0.00
    Value similarity: 0.00
    Value n-gram overlap: 0.14
      Gold WHERE columns: {'syn_root_id'}
      Predicted WHERE columns: {'node_id'}
      Gold WHERE operators: {'='}
      Predicted WHERE operators: {'='}
      Gold WHERE values: {'900614'}
      Predicted WHERE values: {'900514'}
Query 8 similarity:
  Keyword similarity: 1.00
  Keyword n-gram overlap: 1.00
  Argument similarity: 0.67
```

```
  Argument n-gram overlap: 0.90
  Operator similarity: 1.00
  Operator n-gram overlap: 1.00
    Gold keywords: {'SELECT', 'WHERE', 'FROM'}
    Predicted keywords: {'SELECT', 'WHERE', 'FROM'}
    Gold arguments: {'node_type', 'node', 'grp'}
    Predicted arguments: {'node_type', 'node'}
    Gold operators: None
    Predicted operators: None
  WHERE clause component similarities:
    Column similarity: 1.00
    Column n-gram overlap: 1.00
    Operator similarity: 1.00
    Operator n-gram overlap: 0.00
    Value similarity: 1.00
    Value n-gram overlap: 1.00
      Gold WHERE columns: {'node_id'}
      Predicted WHERE columns: {'node_id'}
      Gold WHERE operators: {'='}
      Predicted WHERE operators: {'='}
      Gold WHERE values: {'900620'}
      Predicted WHERE values: {'900620'}
Query 9 similarity:
  Keyword similarity: 1.00
  Keyword n-gram overlap: 1.00
  Argument similarity: 0.33
  Argument n-gram overlap: 0.20
  Operator similarity: 1.00
  Operator n-gram overlap: 1.00
    Gold keywords: {'SELECT', 'WHERE', 'FROM'}
    Predicted keywords: {'SELECT', 'WHERE', 'FROM'}
    Gold arguments: {'edge', 'grp'}
    Predicted arguments: {'edge', 'edge_type'}
    Gold operators: None
    Predicted operators: None
  WHERE clause component similarities:
    Column similarity: 0.25
    Column n-gram overlap: 0.64
    Operator similarity: 1.00
    Operator n-gram overlap: 1.00
    Value similarity: 0.67
    Value n-gram overlap: 0.75
      Gold WHERE columns: {'target_id', 'source_graph', 'source_id'}
      Predicted WHERE columns: {'target_graph', 'source_id'}
      Gold WHERE operators: {'=', 'AND'}
      Predicted WHERE operators: {'=', 'AND'}
      Gold WHERE values: {'1349', '56666', '56690'}
```

```
      Predicted WHERE values: {'56666', '56690'}
Query 10 similarity:
  Keyword similarity: 0.60
  Keyword n-gram overlap: 0.82
  Argument similarity: 0.40
  Argument n-gram overlap: 0.47
  Operator similarity: 0.00
  Operator n-gram overlap: 0.00
    Gold keywords: {'SELECT', 'WHERE', 'FROM'}
    Predicted keywords: {'FROM', 'SELECT', 'WHERE', 'JOIN', 'ON'}
    Gold arguments: {'name', 'graph_type_definition'}
    Predicted arguments: {'node', 'name', 'node_type', 'graph_type_def_id', 'graph_type_definition'}
    Gold operators: None
    Predicted operators: {'='}
  WHERE clause component similarities:
    Column similarity: 0.00
    Column n-gram overlap: 0.00
    Operator similarity: 1.00
    Operator n-gram overlap: 0.00
    Value similarity: 1.00
    Value n-gram overlap: 1.00
      Gold WHERE columns: {'grp'}
      Predicted WHERE columns: {'tiger_node_id'}
      Gold WHERE operators: {'='}
      Predicted WHERE operators: {'='}
      Gold WHERE values: {'tigerXML_node'}
      Predicted WHERE values: {'tigerXML_node'}
```

Listing 7: Evaluation output for SmBop in Experiment 1.

```
Query 1 similarity:
  Keyword similarity: 0.75
  Keyword n-gram overlap: 0.82
  Argument similarity: 0.20
  Argument n-gram overlap: 0.16
  Operator similarity: 1.00
  Operator n-gram overlap: 1.00
    Gold keywords: {'WHERE', 'SELECT', 'FROM'}
    Predicted keywords: {'WHERE', 'JOIN', 'SELECT', 'FROM'}
    Gold arguments: {'syntax_terminal_nodes', 'word'}
    Predicted arguments: {'node', 'prosody_nodes', 'is_target_syn_root', 'word'}
    Gold operators: None
    Predicted operators: None
  WHERE clause component similarities:
    Column similarity: 0.00
    Column n-gram overlap: 0.00
    Operator similarity: 0.00
    Operator n-gram overlap: 0.00
```

```
    Value similarity: 0.50
    Value n-gram overlap: 0.00
      Gold WHERE columns: {'seq', 's_num'}
      Predicted WHERE columns: {'is_label'}
      Gold WHERE operators: {'=', 'AND'}
      Predicted WHERE operators: {'LIKE'}
      Gold WHERE values: {'1', '3'}
      Predicted WHERE values: {'1'}
Query 2 similarity:
  Keyword similarity: 0.60
  Keyword n-gram overlap: 0.69
  Argument similarity: 0.00
  Argument n-gram overlap: 0.15
  Operator similarity: 1.00
  Operator n-gram overlap: 1.00
    Gold keywords: {'WHERE', 'FROM', 'SELECT', 'MAX'}
    Predicted keywords: {'WHERE', 'COUNT', 'SELECT', 'FROM'}
    Gold arguments: {'syntax_terminal_nodes', 'seq'}
    Predicted arguments: {'prosody_nodes'}
    Gold operators: None
    Predicted operators: None
  WHERE clause component similarities:
    Column similarity: 0.00
    Column n-gram overlap: 0.00
    Operator similarity: 0.00
    Operator n-gram overlap: 0.00
    Value similarity: 0.00
    Value n-gram overlap: 0.00
      Gold WHERE columns: {'s_num'}
      Predicted WHERE columns: {'word'}
      Gold WHERE operators: {'='}
      Predicted WHERE operators: {'LIKE'}
      Gold WHERE values: {'3'}
      Predicted WHERE values: {'1'}
Query 3 similarity:
  Keyword similarity: 0.60
  Keyword n-gram overlap: 0.53
  Argument similarity: 0.40
  Argument n-gram overlap: 0.59
  Operator similarity: 1.00
  Operator n-gram overlap: 1.00
    Gold keywords: {'ORDER BY', 'WHERE', 'SELECT', 'FROM'}
    Predicted keywords: {'WHERE', 'JOIN', 'SELECT', 'FROM'}
    Gold arguments: {'pos', 'syntax_terminal_nodes', 'seq'}
    Predicted arguments: {'pos', 'syntax_terminal_nodes', 'prosody_nodes', 'node'}
    Gold operators: None
    Predicted operators: None
```

```
  WHERE clause component similarities:
    Column similarity: 0.00
    Column n-gram overlap: 0.00
    Operator similarity: 0.00
    Operator n-gram overlap: 0.00
    Value similarity: 0.00
    Value n-gram overlap: 0.00
      Gold WHERE columns: {'s_num'}
      Predicted WHERE columns: {'filesequence'}
      Gold WHERE operators: {'='}
      Predicted WHERE operators: {'LIKE'}
      Gold WHERE values: {'3'}
      Predicted WHERE values: {'1'}
Query 4 similarity:
  Keyword similarity: 0.75
  Keyword n-gram overlap: 0.82
  Argument similarity: 0.50
  Argument n-gram overlap: 0.61
  Operator similarity: 1.00
  Operator n-gram overlap: 1.00
    Gold keywords: {'WHERE', 'SELECT', 'FROM'}
    Predicted keywords: {'WHERE', 'JOIN', 'SELECT', 'FROM'}
    Gold arguments: {'pos', 'syntax_terminal_nodes'}
    Predicted arguments: {'pos', 'syntax_terminal_nodes', 'prosody_nodes', 'node'}
    Gold operators: None
    Predicted operators: None
  WHERE clause component similarities:
    Column similarity: 0.00
    Column n-gram overlap: 0.00
    Operator similarity: 0.00
    Operator n-gram overlap: 0.00
    Value similarity: 0.50
    Value n-gram overlap: 0.00
      Gold WHERE columns: {'seq', 's_num'}
      Predicted WHERE columns: {'word'}
      Gold WHERE operators: {'=', 'AND'}
      Predicted WHERE operators: {'LIKE'}
      Gold WHERE values: {'1', '3'}
      Predicted WHERE values: {'1'}
Query 5 similarity:
  Keyword similarity: 1.00
  Keyword n-gram overlap: 1.00
  Argument similarity: 0.00
  Argument n-gram overlap: 0.24
  Operator similarity: 1.00
  Operator n-gram overlap: 1.00
    Gold keywords: {'WHERE', 'SELECT', 'FROM'}
```

```
    Predicted keywords: {'WHERE', 'SELECT', 'FROM'}
    Gold arguments: {'syntax_graph', 'graph_id'}
    Predicted arguments: {'is_target_syn_root', 'is_graph_id'}
    Gold operators: None
    Predicted operators: None
  WHERE clause component similarities:
    Column similarity: 0.00
    Column n-gram overlap: 0.00
    Operator similarity: 1.00
    Operator n-gram overlap: 0.00
    Value similarity: 0.00
    Value n-gram overlap: 0.00
      Gold WHERE columns: {'s_num'}
      Predicted WHERE columns: {'syn_graph_id'}
      Gold WHERE operators: {'='}
      Predicted WHERE operators: {'='}
      Gold WHERE values: {'3'}
      Predicted WHERE values: {'1'}
Query 6 similarity:
  Keyword similarity: 1.00
  Keyword n-gram overlap: 1.00
  Argument similarity: 0.33
  Argument n-gram overlap: 0.73
  Operator similarity: 1.00
  Operator n-gram overlap: 1.00
    Gold keywords: {'WHERE', 'SELECT', 'FROM'}
    Predicted keywords: {'WHERE', 'SELECT', 'FROM'}
    Gold arguments: {'prosody_nodes', 'word'}
    Predicted arguments: {'tone', 'prosody_nodes'}
    Gold operators: None
    Predicted operators: None
  WHERE clause component similarities:
    Column similarity: 0.50
    Column n-gram overlap: 0.83
    Operator similarity: 0.00
    Operator n-gram overlap: 0.00
    Value similarity: 0.00
    Value n-gram overlap: 0.00
      Gold WHERE columns: {'tone', 'filesequence'}
      Predicted WHERE columns: {'filesequence'}
      Gold WHERE operators: {'=', 'AND'}
      Predicted WHERE operators: {'NOT IN'}
      Gold WHERE values: {'NONE', 'dlf-nachrichten-200703262000'}
      Predicted WHERE values: {'(SELECT T2.node_id FROM node AS T2)'}
Query 7 similarity:
  Keyword similarity: 1.00
  Keyword n-gram overlap: 1.00
```

```
  Argument similarity: 1.00
  Argument n-gram overlap: 1.00
  Operator similarity: 1.00
  Operator n-gram overlap: 1.00
    Gold keywords: {'WHERE', 'SELECT', 'FROM'}
    Predicted keywords: {'WHERE', 'SELECT', 'FROM'}
    Gold arguments: {'is_target_syn_root', 'is_label'}
    Predicted arguments: {'is_target_syn_root', 'is_label'}
    Gold operators: None
    Predicted operators: None
  WHERE clause component similarities:
    Column similarity: 0.00
    Column n-gram overlap: 0.00
    Operator similarity: 1.00
    Operator n-gram overlap: 0.00
    Value similarity: 0.00
    Value n-gram overlap: 0.00
      Gold WHERE columns: {'syn_root_id'}
      Predicted WHERE columns: {'is_node_descr'}
      Gold WHERE operators: {'='}
      Predicted WHERE operators: {'='}
      Gold WHERE values: {'900614'}
      Predicted WHERE values: {'1'}
Query 8 similarity:
  Keyword similarity: 0.75
  Keyword n-gram overlap: 0.82
  Argument similarity: 0.40
  Argument n-gram overlap: 0.24
  Operator similarity: 1.00
  Operator n-gram overlap: 1.00
    Gold keywords: {'WHERE', 'SELECT', 'FROM'}
    Predicted keywords: {'WHERE', 'JOIN', 'SELECT', 'FROM'}
    Gold arguments: {'node', 'grp', 'node_type'}
    Predicted arguments: {'node', 'is_target_syn_root', 'is_node_descr', 'node_type'}
    Gold operators: None
    Predicted operators: None
  WHERE clause component similarities:
    Column similarity: 0.00
    Column n-gram overlap: 0.62
    Operator similarity: 1.00
    Operator n-gram overlap: 0.00
    Value similarity: 0.00
    Value n-gram overlap: 0.00
      Gold WHERE columns: {'node_id'}
      Predicted WHERE columns: {'is_node_id'}
      Gold WHERE operators: {'='}
      Predicted WHERE operators: {'='}
```

```
      Gold WHERE values: {'900620'}
      Predicted WHERE values: {'1'}
Query 9 similarity:
  Keyword similarity: 0.75
  Keyword n-gram overlap: 0.82
  Argument similarity: 0.00
  Argument n-gram overlap: 0.00
  Operator similarity: 1.00
  Operator n-gram overlap: 1.00
    Gold keywords: {'WHERE', 'SELECT', 'FROM'}
    Predicted keywords: {'WHERE', 'JOIN', 'SELECT', 'FROM'}
    Gold arguments: {'edge', 'grp'}
    Predicted arguments: {'graph_type_definition', 'name', 'prosody_nodes', 'syntax_terminal_nodes'}
    Gold operators: None
    Predicted operators: None
  WHERE clause component similarities:
    Column similarity: 0.00
    Column n-gram overlap: 0.05
    Operator similarity: 0.67
    Operator n-gram overlap: 1.00
    Value similarity: 0.00
    Value n-gram overlap: 0.00
      Gold WHERE columns: {'target_id', 'source_graph', 'source_id'}
      Predicted WHERE columns: {'filesequence', 'node_id'}
      Gold WHERE operators: {'=', 'AND'}
      Predicted WHERE operators: {'=', 'AND', '>'}
      Gold WHERE values: {'56690', '1349', '56666'}
      Predicted WHERE values: {'1'}
Query 10 similarity:
  Keyword similarity: 0.75
  Keyword n-gram overlap: 0.82
  Argument similarity: 0.50
  Argument n-gram overlap: 0.75
  Operator similarity: 1.00
  Operator n-gram overlap: 1.00
    Gold keywords: {'WHERE', 'SELECT', 'FROM'}
    Predicted keywords: {'WHERE', 'JOIN', 'SELECT', 'FROM'}
    Gold arguments: {'graph_type_definition', 'name'}
    Predicted arguments: {'graph_type_definition', 'name', 'closure', 'node'}
    Gold operators: None
    Predicted operators: None
  WHERE clause component similarities:
    Column similarity: 0.00
    Column n-gram overlap: 0.00
    Operator similarity: 1.00
    Operator n-gram overlap: 0.00
    Value similarity: 0.00
```

```
    Value n-gram overlap: 0.00
      Gold WHERE columns: {'grp'}
      Predicted WHERE columns: {'closure_name'}
      Gold WHERE operators: {'='}
      Predicted WHERE operators: {'='}
      Gold WHERE values: {'tigerXML_node'}
      Predicted WHERE values: {'1'}
```

Listing 8: Evaluation output for IRNet in Experiment 1.

```
Query 1 similarity:
  Keyword similarity: 0.50
  Keyword n-gram overlap: 0.53
  Argument similarity: 0.40
  Argument n-gram overlap: 0.32
  Operator similarity: 0.00
  Operator n-gram overlap: 0.00
    Gold keywords: {'FROM', 'SELECT', 'WHERE'}
    Predicted keywords: {'FROM', 'GROUP BY', 'SELECT', 'WHERE', 'ON', 'JOIN'}
    Gold arguments: {'syntax_terminal_nodes', 'word'}
    Predicted arguments: {'syntax_terminal_nodes', 'word', 'syntax_terminal_graph_id', 'syntax_graph
     ', 'sentence_number'}
    Gold operators: None
    Predicted operators: {'='}
  WHERE clause component similarities:
    Column similarity: 0.50
    Column n-gram overlap: 0.62
    Operator similarity: 0.50
    Operator n-gram overlap: 0.00
    Value similarity: 0.50
    Value n-gram overlap: 0.00
      Gold WHERE columns: {'sentence_number', ''sequence''}
      Predicted WHERE columns: {'sentence_number'}
      Gold WHERE operators: {'AND', '='}
      Predicted WHERE operators: {'='}
      Gold WHERE values: {'3', '1'}
      Predicted WHERE values: {'3'}
Query 2 similarity:
  Keyword similarity: 0.60
  Keyword n-gram overlap: 0.69
  Argument similarity: 0.00
  Argument n-gram overlap: 0.16
  Operator similarity: 1.00
  Operator n-gram overlap: 1.00
    Gold keywords: {'FROM', 'SELECT', 'WHERE', 'MAX'}
    Predicted keywords: {'FROM', 'SELECT', 'WHERE', 'COUNT'}
    Gold arguments: {'syntax_terminal_nodes', ''sequence''}
    Predicted arguments: {'syntax_graph'}
```

```
    Gold operators: None
    Predicted operators: None
  WHERE clause component similarities:
    Column similarity: 1.00
    Column n-gram overlap: 1.00
    Operator similarity: 1.00
    Operator n-gram overlap: 0.00
    Value similarity: 1.00
    Value n-gram overlap: 0.00
      Gold WHERE columns: {'sentence_number'}
      Predicted WHERE columns: {'sentence_number'}
      Gold WHERE operators: {'='}
      Predicted WHERE operators: {'='}
      Gold WHERE values: {'3'}
      Predicted WHERE values: {'3'}
Query 3 similarity:
  Keyword similarity: 0.75
  Keyword n-gram overlap: 0.60
  Argument similarity: 0.67
  Argument n-gram overlap: 0.71
  Operator similarity: 1.00
  Operator n-gram overlap: 1.00
    Gold keywords: {'FROM', 'SELECT', 'WHERE', 'ORDER BY'}
    Predicted keywords: {'FROM', 'SELECT', 'WHERE'}
    Gold arguments: {'syntax_terminal_nodes', '`sequence`', 'pos'}
    Predicted arguments: {'syntax_terminal_nodes', 'pos'}
    Gold operators: None
    Predicted operators: None
  WHERE clause component similarities:
    Column similarity: 0.00
    Column n-gram overlap: 0.00
    Operator similarity: 1.00
    Operator n-gram overlap: 0.00
    Value similarity: 1.00
    Value n-gram overlap: 0.00
      Gold WHERE columns: {'sentence_number'}
      Predicted WHERE columns: {'word'}
      Gold WHERE operators: {'='}
      Predicted WHERE operators: {'='}
      Gold WHERE values: {'3'}
      Predicted WHERE values: {'3'}
Query 4 similarity:
  Keyword similarity: 1.00
  Keyword n-gram overlap: 1.00
  Argument similarity: 1.00
  Argument n-gram overlap: 1.00
  Operator similarity: 1.00
```

```
  Operator n-gram overlap: 1.00
    Gold keywords: {'FROM', 'SELECT', 'WHERE'}
    Predicted keywords: {'FROM', 'SELECT', 'WHERE'}
    Gold arguments: {'syntax_terminal_nodes', 'pos'}
    Predicted arguments: {'syntax_terminal_nodes', 'pos'}
    Gold operators: None
    Predicted operators: None
  WHERE clause component similarities:
    Column similarity: 0.00
    Column n-gram overlap: 0.00
    Operator similarity: 0.50
    Operator n-gram overlap: 0.00
    Value similarity: 0.50
    Value n-gram overlap: 0.00
      Gold WHERE columns: {'sentence_number', ''sequence''}
      Predicted WHERE columns: {'word'}
      Gold WHERE operators: {'AND', '='}
      Predicted WHERE operators: {'='}
      Gold WHERE values: {'3', '1'}
      Predicted WHERE values: {'3'}
Query 5 similarity:
  Keyword similarity: 0.60
  Keyword n-gram overlap: 0.82
  Argument similarity: 0.00
  Argument n-gram overlap: 0.24
  Operator similarity: 0.00
  Operator n-gram overlap: 0.00
    Gold keywords: {'FROM', 'SELECT', 'WHERE'}
    Predicted keywords: {'FROM', 'SELECT', 'WHERE', 'ON', 'JOIN'}
    Gold arguments: {'syntax_graph_id', 'syntax_graph'}
    Predicted arguments: {'syntax_nonterminal_node_id', 'graph_id', 'syntax_nonterminal_nodes', '
     node', 'node_id'}
    Gold operators: None
    Predicted operators: {'='}
  WHERE clause component similarities:
    Column similarity: 1.00
    Column n-gram overlap: 1.00
    Operator similarity: 1.00
    Operator n-gram overlap: 0.00
    Value similarity: 1.00
    Value n-gram overlap: 0.00
      Gold WHERE columns: {'sentence_number'}
      Predicted WHERE columns: {'sentence_number'}
      Gold WHERE operators: {'='}
      Predicted WHERE operators: {'='}
      Gold WHERE values: {'3'}
      Predicted WHERE values: {'3'}
```

```
Query 6 similarity:
  Keyword similarity: 0.75
  Keyword n-gram overlap: 0.69
  Argument similarity: 0.33
  Argument n-gram overlap: 0.61
  Operator similarity: 1.00
  Operator n-gram overlap: 1.00
    Gold keywords: {'FROM', 'SELECT', 'WHERE'}
    Predicted keywords: {'FROM', 'SELECT', 'WHERE', 'EXCEPT'}
    Gold arguments: {'word', 'prosody_nodes'}
    Predicted arguments: {'prosody_nodes', 'accents'}
    Gold operators: None
    Predicted operators: None
  WHERE clause component similarities:
    Column similarity: 1.00
    Column n-gram overlap: 1.00
    Operator similarity: 0.50
    Operator n-gram overlap: 0.00
    Value similarity: 0.50
    Value n-gram overlap: 0.93
      Gold WHERE columns: {'file_sequence', 'tone'}
      Predicted WHERE columns: {'file_sequence', 'tone'}
      Gold WHERE operators: {'AND', '='}
      Predicted WHERE operators: {'='}
      Gold WHERE values: {'NONE', 'dlf-nachrichten-200703262000'}
      Predicted WHERE values: {'dlf-nachrichten-200703262000'}
Query 7 similarity:
  Keyword similarity: 0.60
  Keyword n-gram overlap: 0.82
  Argument similarity: 0.40
  Argument n-gram overlap: 0.69
  Operator similarity: 0.00
  Operator n-gram overlap: 0.00
    Gold keywords: {'FROM', 'SELECT', 'WHERE'}
    Predicted keywords: {'FROM', 'SELECT', 'WHERE', 'ON', 'JOIN'}
    Gold arguments: {'info_status_target_syn_root', 'info_status_label'}
    Predicted arguments: {'graph_id', 'info_status_target_syn_root', 'node', 'syn_graph_id', '
     info_status_label'}
    Gold operators: None
    Predicted operators: {'='}
  WHERE clause component similarities:
    Column similarity: 0.00
    Column n-gram overlap: 0.08
    Operator similarity: 1.00
    Operator n-gram overlap: 0.00
    Value similarity: 0.00
    Value n-gram overlap: 0.14
```

```
      Gold WHERE columns: {'syn_root_id'}
      Predicted WHERE columns: {'node_id'}
      Gold WHERE operators: {'='}
      Predicted WHERE operators: {'='}
      Gold WHERE values: {'900614'}
      Predicted WHERE values: {'900514'}
Query 8 similarity:
  Keyword similarity: 1.00
  Keyword n-gram overlap: 1.00
  Argument similarity: 1.00
  Argument n-gram overlap: 1.00
  Operator similarity: 1.00
  Operator n-gram overlap: 1.00
    Gold keywords: {'FROM', 'SELECT', 'WHERE'}
    Predicted keywords: {'FROM', 'SELECT', 'WHERE'}
    Gold arguments: {'node_group', 'node_type', 'node'}
    Predicted arguments: {'node_group', 'node_type', 'node'}
    Gold operators: None
    Predicted operators: None
  WHERE clause component similarities:
    Column similarity: 1.00
    Column n-gram overlap: 1.00
    Operator similarity: 1.00
    Operator n-gram overlap: 0.00
    Value similarity: 1.00
    Value n-gram overlap: 1.00
      Gold WHERE columns: {'node_id'}
      Predicted WHERE columns: {'node_id'}
      Gold WHERE operators: {'='}
      Predicted WHERE operators: {'='}
      Gold WHERE values: {'900620'}
      Predicted WHERE values: {'900620'}
Query 9 similarity:
  Keyword similarity: 1.00
  Keyword n-gram overlap: 1.00
  Argument similarity: 1.00
  Argument n-gram overlap: 1.00
  Operator similarity: 1.00
  Operator n-gram overlap: 1.00
    Gold keywords: {'FROM', 'SELECT', 'WHERE'}
    Predicted keywords: {'FROM', 'SELECT', 'WHERE'}
    Gold arguments: {'edge_group', 'edge'}
    Predicted arguments: {'edge_group', 'edge'}
    Gold operators: None
    Predicted operators: None
  WHERE clause component similarities:
    Column similarity: 0.25
```

```
    Column n-gram overlap: 0.45
    Operator similarity: 1.00
    Operator n-gram overlap: 1.00
    Value similarity: 0.67
    Value n-gram overlap: 0.75
      Gold WHERE columns: {'source_graph', 'source_node', 'target_node'}
      Predicted WHERE columns: {'source_graph', 'target_graph'}
      Gold WHERE operators: {'AND', '='}
      Predicted WHERE operators: {'AND', '='}
      Gold WHERE values: {'1349', '56666', '56690'}
      Predicted WHERE values: {'56666', '56690'}
Query 10 similarity:
  Keyword similarity: 0.60
  Keyword n-gram overlap: 0.82
  Argument similarity: 0.40
  Argument n-gram overlap: 0.47
  Operator similarity: 0.00
  Operator n-gram overlap: 0.00
    Gold keywords: {'FROM', 'SELECT', 'WHERE'}
    Predicted keywords: {'FROM', 'SELECT', 'WHERE', 'ON', 'JOIN'}
    Gold arguments: {'name', 'graph_type_definition'}
    Predicted arguments: {'node_type', 'graph_type_definition', 'node', 'graph_type_def_id', 'name'}
    Gold operators: None
    Predicted operators: {'='}
  WHERE clause component similarities:
    Column similarity: 0.00
    Column n-gram overlap: 0.29
    Operator similarity: 1.00
    Operator n-gram overlap: 0.00
    Value similarity: 1.00
    Value n-gram overlap: 1.00
      Gold WHERE columns: {'graph_type_group'}
      Predicted WHERE columns: {'node_group'}
      Gold WHERE operators: {'='}
      Predicted WHERE operators: {'='}
      Gold WHERE values: {'tigerXML_node'}
      Predicted WHERE values: {'tigerXML_node'}
```

Listing 9: Evaluation output for SmBop in Experiment 2.

```
Query 1 similarity:
  Keyword similarity: 0.75
  Keyword n-gram overlap: 0.82
  Argument similarity: 0.20
  Argument n-gram overlap: 0.40
  Operator similarity: 1.00
  Operator n-gram overlap: 1.00
    Gold keywords: {'FROM', 'SELECT', 'WHERE'}
```

```
    Predicted keywords: {'FROM', 'SELECT', 'JOIN', 'WHERE'}
    Gold arguments: {'syntax_terminal_nodes', 'word'}
    Predicted arguments: {'sentence_number', 'syntax_terminal_nodes', 'node', 'prosody_nodes'}
    Gold operators: None
    Predicted operators: None
  WHERE clause component similarities:
    Column similarity: 0.00
    Column n-gram overlap: 0.00
    Operator similarity: 0.50
    Operator n-gram overlap: 0.00
    Value similarity: 0.50
    Value n-gram overlap: 0.00
      Gold WHERE columns: {'sentence_number', ''sequence''}
      Predicted WHERE columns: {'word'}
      Gold WHERE operators: {'=', 'AND'}
      Predicted WHERE operators: {'='}
      Gold WHERE values: {'3', '1'}
      Predicted WHERE values: {'1'}
Query 2 similarity:
  Keyword similarity: 0.60
  Keyword n-gram overlap: 0.69
  Argument similarity: 0.00
  Argument n-gram overlap: 0.12
  Operator similarity: 1.00
  Operator n-gram overlap: 1.00
    Gold keywords: {'MAX', 'SELECT', 'FROM', 'WHERE'}
    Predicted keywords: {'FROM', 'SELECT', 'COUNT', 'WHERE'}
    Gold arguments: {'syntax_terminal_nodes', ''sequence''}
    Predicted arguments: {'prosody_nodes'}
    Gold operators: None
    Predicted operators: None
  WHERE clause component similarities:
    Column similarity: 0.00
    Column n-gram overlap: 0.00
    Operator similarity: 0.00
    Operator n-gram overlap: 0.00
    Value similarity: 0.00
    Value n-gram overlap: 0.00
      Gold WHERE columns: {'sentence_number'}
      Predicted WHERE columns: {'word'}
      Gold WHERE operators: {'='}
      Predicted WHERE operators: {'LIKE'}
      Gold WHERE values: {'3'}
      Predicted WHERE values: {'1'}
Query 3 similarity:
  Keyword similarity: 0.60
  Keyword n-gram overlap: 0.53
```

```
  Argument similarity: 0.40
  Argument n-gram overlap: 0.49
  Operator similarity: 1.00
  Operator n-gram overlap: 1.00
    Gold keywords: {'FROM', 'SELECT', 'ORDER BY', 'WHERE'}
    Predicted keywords: {'FROM', 'SELECT', 'JOIN', 'WHERE'}
    Gold arguments: {'syntax_terminal_nodes', 'pos', ''sequence''}
    Predicted arguments: {'node', 'syntax_terminal_nodes', 'pos', 'prosody_nodes'}
    Gold operators: None
    Predicted operators: None
  WHERE clause component similarities:
    Column similarity: 0.00
    Column n-gram overlap: 0.00
    Operator similarity: 0.00
    Operator n-gram overlap: 0.00
    Value similarity: 0.00
    Value n-gram overlap: 0.00
      Gold WHERE columns: {'sentence_number'}
      Predicted WHERE columns: {'word'}
      Gold WHERE operators: {'='}
      Predicted WHERE operators: {'LIKE'}
      Gold WHERE values: {'3'}
      Predicted WHERE values: {'1'}
Query 4 similarity:
  Keyword similarity: 0.75
  Keyword n-gram overlap: 0.82
  Argument similarity: 0.50
  Argument n-gram overlap: 0.61
  Operator similarity: 1.00
  Operator n-gram overlap: 1.00
    Gold keywords: {'FROM', 'SELECT', 'WHERE'}
    Predicted keywords: {'FROM', 'SELECT', 'JOIN', 'WHERE'}
    Gold arguments: {'syntax_terminal_nodes', 'pos'}
    Predicted arguments: {'node', 'syntax_terminal_nodes', 'pos', 'prosody_nodes'}
    Gold operators: None
    Predicted operators: None
  WHERE clause component similarities:
    Column similarity: 0.00
    Column n-gram overlap: 0.00
    Operator similarity: 0.00
    Operator n-gram overlap: 0.00
    Value similarity: 0.50
    Value n-gram overlap: 0.00
      Gold WHERE columns: {'sentence_number', ''sequence''}
      Predicted WHERE columns: {'word'}
      Gold WHERE operators: {'=', 'AND'}
      Predicted WHERE operators: {'LIKE'}
```

```
      Gold WHERE values: {'3', '1'}
      Predicted WHERE values: {'1'}
Query 5 similarity:
  Keyword similarity: 0.75
  Keyword n-gram overlap: 0.82
  Argument similarity: 0.00
  Argument n-gram overlap: 0.28
  Operator similarity: 1.00
  Operator n-gram overlap: 1.00
    Gold keywords: {'FROM', 'SELECT', 'WHERE'}
    Predicted keywords: {'FROM', 'SELECT', 'JOIN', 'WHERE'}
    Gold arguments: {'syntax_graph_id', 'syntax_graph'}
    Predicted arguments: {'node', 'syntax_terminal_nodes', 'graph_id'}
    Gold operators: None
    Predicted operators: None
  WHERE clause component similarities:
    Column similarity: 1.00
    Column n-gram overlap: 1.00
    Operator similarity: 1.00
    Operator n-gram overlap: 0.00
    Value similarity: 0.00
    Value n-gram overlap: 0.00
      Gold WHERE columns: {'sentence_number'}
      Predicted WHERE columns: {'sentence_number'}
      Gold WHERE operators: {'='}
      Predicted WHERE operators: {'='}
      Gold WHERE values: {'3'}
      Predicted WHERE values: {'1'}
Query 6 similarity:
  Keyword similarity: 1.00
  Keyword n-gram overlap: 1.00
  Argument similarity: 0.33
  Argument n-gram overlap: 0.73
  Operator similarity: 1.00
  Operator n-gram overlap: 1.00
    Gold keywords: {'FROM', 'SELECT', 'WHERE'}
    Predicted keywords: {'FROM', 'SELECT', 'WHERE'}
    Gold arguments: {'prosody_nodes', 'word'}
    Predicted arguments: {'tone', 'prosody_nodes'}
    Gold operators: None
    Predicted operators: None
  WHERE clause component similarities:
    Column similarity: 0.50
    Column n-gram overlap: 0.85
    Operator similarity: 0.00
    Operator n-gram overlap: 0.00
    Value similarity: 0.00
```

```
    Value n-gram overlap: 0.00
      Gold WHERE columns: {'tone', 'file_sequence'}
      Predicted WHERE columns: {'file_sequence'}
      Gold WHERE operators: {'=', 'AND'}
      Predicted WHERE operators: {'NOT IN'}
      Gold WHERE values: {'dlf-nachrichten-200703262000', 'NONE'}
      Predicted WHERE values: {'(SELECT T2.node_id FROM node AS T2)'}
Query 7 similarity:
  Keyword similarity: 1.00
  Keyword n-gram overlap: 1.00
  Argument similarity: 1.00
  Argument n-gram overlap: 1.00
  Operator similarity: 1.00
  Operator n-gram overlap: 1.00
    Gold keywords: {'FROM', 'SELECT', 'WHERE'}
    Predicted keywords: {'FROM', 'SELECT', 'WHERE'}
    Gold arguments: {'info_status_label', 'info_status_target_syn_root'}
    Predicted arguments: {'info_status_label', 'info_status_target_syn_root'}
    Gold operators: None
    Predicted operators: None
  WHERE clause component similarities:
    Column similarity: 0.00
    Column n-gram overlap: 0.00
    Operator similarity: 1.00
    Operator n-gram overlap: 0.00
    Value similarity: 0.00
    Value n-gram overlap: 0.00
      Gold WHERE columns: {'syn_root_id'}
      Predicted WHERE columns: {'info_status_node_descr'}
      Gold WHERE operators: {'='}
      Predicted WHERE operators: {'='}
      Gold WHERE values: {'900614'}
      Predicted WHERE values: {'1'}
Query 8 similarity:
  Keyword similarity: 0.75
  Keyword n-gram overlap: 0.82
  Argument similarity: 0.75
  Argument n-gram overlap: 0.89
  Operator similarity: 1.00
  Operator n-gram overlap: 1.00
    Gold keywords: {'FROM', 'SELECT', 'WHERE'}
    Predicted keywords: {'FROM', 'SELECT', 'JOIN', 'WHERE'}
    Gold arguments: {'node', 'node_type', 'node_group'}
    Predicted arguments: {'node', 'node_type', 'edge', 'node_group'}
    Gold operators: None
    Predicted operators: None
  WHERE clause component similarities:
```

```
    Column similarity: 0.00
    Column n-gram overlap: 0.17
    Operator similarity: 1.00
    Operator n-gram overlap: 0.00
    Value similarity: 0.00
    Value n-gram overlap: 0.00
      Gold WHERE columns: {'node_id'}
      Predicted WHERE columns: {'source_node'}
      Gold WHERE operators: {'='}
      Predicted WHERE operators: {'='}
      Gold WHERE values: {'900620'}
      Predicted WHERE values: {'1'}
Query 9 similarity:
  Keyword similarity: 0.75
  Keyword n-gram overlap: 0.82
  Argument similarity: 0.67
  Argument n-gram overlap: 0.83
  Operator similarity: 1.00
  Operator n-gram overlap: 1.00
    Gold keywords: {'FROM', 'SELECT', 'WHERE'}
    Predicted keywords: {'FROM', 'SELECT', 'JOIN', 'WHERE'}
    Gold arguments: {'edge_group', 'edge'}
    Predicted arguments: {'node', 'edge_group', 'edge'}
    Gold operators: None
    Predicted operators: None
  WHERE clause component similarities:
    Column similarity: 0.50
    Column n-gram overlap: 0.55
    Operator similarity: 1.00
    Operator n-gram overlap: 1.00
    Value similarity: 0.00
    Value n-gram overlap: 0.00
      Gold WHERE columns: {'source_graph', 'source_node', 'target_node'}
      Predicted WHERE columns: {'target_node', 'source_node', 'node_id'}
      Gold WHERE operators: {'=', 'AND'}
      Predicted WHERE operators: {'=', 'AND'}
      Gold WHERE values: {'56666', '56690', '1349'}
      Predicted WHERE values: {'1'}
Query 10 similarity:
  Keyword similarity: 0.75
  Keyword n-gram overlap: 0.82
  Argument similarity: 0.50
  Argument n-gram overlap: 0.75
  Operator similarity: 1.00
  Operator n-gram overlap: 1.00
    Gold keywords: {'FROM', 'SELECT', 'WHERE'}
    Predicted keywords: {'FROM', 'SELECT', 'JOIN', 'WHERE'}
```

```
    Gold arguments: {'name', 'graph_type_definition'}
    Predicted arguments: {'node', 'name', 'graph_type_definition', 'closure'}
    Gold operators: None
    Predicted operators: None
  WHERE clause component similarities:
    Column similarity: 0.00
    Column n-gram overlap: 0.00
    Operator similarity: 1.00
    Operator n-gram overlap: 0.00
    Value similarity: 0.00
    Value n-gram overlap: 0.00
      Gold WHERE columns: {'graph_type_group'}
      Predicted WHERE columns: {'closure_name'}
      Gold WHERE operators: {'='}
      Predicted WHERE operators: {'='}
      Gold WHERE values: {'tigerXML_node'}
      Predicted WHERE values: {'1'}
```

Listing 10: Evaluation output for IRNet in Experiment 2.

```
Query 1 similarity:
  Keyword similarity: 0.83
  Keyword n-gram overlap: 0.61
  Argument similarity: 0.21
  Argument n-gram overlap: 0.23
  Operator similarity: 0.50
  Operator n-gram overlap: 0.00
    Gold keywords: {'WHERE', 'JOIN', 'ON', 'SELECT', 'FROM'}
    Predicted keywords: {'INTERSECT', 'WHERE', 'JOIN', 'ON', 'SELECT', 'FROM'}
    Gold arguments: {'graph_id', 'prosody_nodes', 'accents', 'syntax_terminal_nodes', 'word', '
     source_id', 'node_id', 'target_id', 'target_graph', 'ts', 'source_graph', 'tone', 'edge'}
    Predicted arguments: {'node_id', 'prosody_nodes', 'word', 'node'}
    Gold operators: {'AND', '='}
    Predicted operators: {'='}
  WHERE clause component similarities:
    Column similarity: 0.00
    Column n-gram overlap: 0.00
    Operator similarity: 1.00
    Operator n-gram overlap: 0.00
    Value similarity: 0.00
    Value n-gram overlap: 0.00
      Gold WHERE columns: {'grp'}
      Predicted WHERE columns: {'node_type', 'node_id'}
      Gold WHERE operators: {'='}
      Predicted WHERE operators: {'='}
      Gold WHERE values: {'link_edge'}
      Predicted WHERE values: {'syntactic leaf', 'node.node_id'}
Query 2 similarity:
```

```
  Keyword similarity: 0.50
  Keyword n-gram overlap: 0.53
  Argument similarity: 0.07
  Argument n-gram overlap: 0.10
  Operator similarity: 0.00
  Operator n-gram overlap: 0.00
    Gold keywords: {'WHERE', 'JOIN', 'ON', 'SELECT', 'ORDER BY', 'FROM'}
    Predicted keywords: {'FROM', 'WHERE', 'SELECT'}
    Gold arguments: {'sentences', 'sentence_array', 'graph_id', 'tone', 'terminal_and_prosody', '
     word', 's_num', 'seq', 'node_id', 'pos', 'sentence_length', 'ts', 'accents', 'filesequence'}
    Predicted arguments: {'prosody_nodes', 'word'}
    Gold operators: {'='}
    Predicted operators: None
  WHERE clause component similarities:
    Column similarity: 0.00
    Column n-gram overlap: 0.00
    Operator similarity: 0.33
    Operator n-gram overlap: 0.67
    Value similarity: 0.00
    Value n-gram overlap: 0.00
      Gold WHERE columns: {'accents', 'tone'}
      Predicted WHERE columns: {'word'}
      Gold WHERE operators: {'AND', '!=', 'LIKE'}
      Predicted WHERE operators: {'LIKE'}
      Gold WHERE values: {'|NONE|', '%\\\\%%'}
      Predicted WHERE values: {'%nuclear%'}
Query 3 similarity:
  Keyword similarity: 0.50
  Keyword n-gram overlap: 0.60
  Argument similarity: 0.00
  Argument n-gram overlap: 0.01
  Operator similarity: 0.00
  Operator n-gram overlap: 0.00
    Gold keywords: {'EXCEPT', 'WHERE', 'JOIN', 'ON', 'SELECT', 'FROM'}
    Predicted keywords: {'FROM', 'WHERE', 'SELECT'}
    Gold arguments: {'graph_id', 'node_graph', 'syntax_terminal_nodes', 'subnode_id', 'node_id', '
     closure', 'syntax_nonterminal_nodes'}
    Predicted arguments: {'sentences', 'sentence_array'}
    Gold operators: {'AND', '='}
    Predicted operators: None
  WHERE clause component similarities:
    Column similarity: 0.00
    Column n-gram overlap: 0.00
    Operator similarity: 0.00
    Operator n-gram overlap: 0.00
    Value similarity: 0.00
    Value n-gram overlap: 0.00
```

```
      Gold WHERE columns: {'pos', 'cat'}
      Predicted WHERE columns: {'sentence_length'}
      Gold WHERE operators: {'LIKE'}
      Predicted WHERE operators: {'<'}
      Gold WHERE values: {'DP[%', 'A[%'}
      Predicted WHERE values: {"(SELECT MAX( sentences.sentence_length ) FROM sentences WHERE
    sentences.sentence_length LIKE '%DPs%')"}
Query 4 similarity:
  Keyword similarity: 0.80
  Keyword n-gram overlap: 0.73
  Argument similarity: 0.20
  Argument n-gram overlap: 0.29
  Operator similarity: 0.50
  Operator n-gram overlap: 0.00
    Gold keywords: {'WHERE', 'JOIN', 'ON', 'SELECT', 'FROM'}
    Predicted keywords: {'FROM', 'JOIN', 'ON', 'SELECT'}
    Gold arguments: {'graph_id', 'prosody_nodes', 'accents', 'syntax_terminal_nodes', 'word', '
     source_id', 'node_id', 'target_id', 'target_graph', 'ts', 'source_graph', 'tone', 'edge', '
     filesequence'}
    Predicted arguments: {'prosody_nodes', 'syntax_graph', 'graph_id', 'word'}
    Gold operators: {'AND', '='}
    Predicted operators: {'='}
  WHERE clause component similarities:
    Column similarity: 0.00
    Column n-gram overlap: 0.00
    Operator similarity: 0.00
    Operator n-gram overlap: 0.00
    Value similarity: 0.00
    Value n-gram overlap: 0.00
      Gold WHERE columns: {'grp'}
      Predicted WHERE columns: set()
      Gold WHERE operators: {'='}
      Predicted WHERE operators: None
      Gold WHERE values: {'link_edge'}
      Predicted WHERE values: set()
```

Listing 11: Evaluation output for SmBop in Experiment 3.

```
Query 1 similarity:
  Keyword similarity: 0.50
  Keyword n-gram overlap: 0.60
  Argument similarity: 0.00
  Argument n-gram overlap: 0.04
  Operator similarity: 0.00
  Operator n-gram overlap: 0.00
    Gold keywords: {'SELECT', 'FROM', 'JOIN', 'EXCEPT', 'ON', 'WHERE'}
    Predicted keywords: {'FROM', 'WHERE', 'SELECT'}
```

```
    Gold arguments: {'closure', 'subnode_id', 'graph_id', 'syntax_terminal_nodes', 'node_graph', '
     syntax_nonterminal_nodes', 'node_id'}
    Predicted arguments: {'created', 'graph_type_definition'}
    Gold operators: {'AND', '='}
    Predicted operators: None
  WHERE clause component similarities:
    Column similarity: 0.00
    Column n-gram overlap: 0.00
    Operator similarity: 0.00
    Operator n-gram overlap: 0.00
    Value similarity: 0.00
    Value n-gram overlap: 0.00
      Gold WHERE columns: {'cat', 'pos'}
      Predicted WHERE columns: {'graph_type_def_id'}
      Gold WHERE operators: {'LIKE'}
      Predicted WHERE operators: {'NOT IN'}
      Gold WHERE values: {'DP[%', 'A[%'}
      Predicted WHERE values: {'(SELECT T2.source_id FROM edge AS T2 JOIN node AS T4 JOIN
     prosody_nodes AS T3 WHERE T3.word like 1       )'}
```

Listing 12: Evaluation output for IRNet in Experiment 3.