NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

MSc PROGRAM
”DATA SCIENCE AND INFORMATION TECHNOLOGIES”

MSc THESIS

Data Democratisation with Deep Learning: Structured Query Translation from and to Natural Language

Georgios G. Katsogiannis-Meimarakis

Supervisor: **Georgia Koutrika,** Research Director, Athena R.C.

ATHENS

APRIL 2023

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΔΙΑΤΜΗΜΑΤΙΚΟ ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
”ΕΠΙΣΤΗΜΗ ΔΕΔΟΜΕΝΩΝ ΚΑΙ ΤΕΧΝΟΛΟΓΙΕΣ ΠΛΗΡΟΦΟΡΙΑΣ”

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Δημοκρατικοποίηση Δεδομένων με Βαθιά Μάθηση: Μετάφραση Δομημένων Ερωτημάτων από και σε Φυσική Γλώσσα

Γεώργιος Γ. Κατσογιάννης-Μεϊμαράκης

Επιβλέπουσα: **Γεωργία Κούτρικα,** Διευθύντρια Ερευνών, Ε.Κ. Αθηνά

ΑΘΗΝΑ

ΑΠΡΙΛΙΟΣ 2023

MSc THESIS

Data Democratisation with Deep Learning: Structured Query Translation from and to Natural Language

Georgios G. Katsogiannis-Meimarakis

S.N.: DS1200011

SUPERVISOR: **Georgia Koutrika,** Research Director, Athena R.C.

EXAMINATION COMMITEE: **Georgia Koutrika,** Research Director, Athena R.C.
Yannis Ioannidis, Professor, NKUA
Ion Androutsopoulos, Professor, AUEB

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Δημοκρατικοποίηση Δεδομένων με Βαθιά Μάθηση: Μετάφραση Δομημένων Ερωτημάτων από και σε Φυσική Γλώσσα

Γεώργιος Γ. Κατσογιάννης-Μεϊμαράκης
Α.Μ.: DS1200011

ΕΠΙΒΛΕΠΟΥΣΑ: **Γεωργία Κούτρικα,** Διευθύντρια Ερευνών, Ε.Κ. Αθηνά

ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ: **Γεωργία Κούτρικα,** Διευθύντρια Ερευνών, Ε.Κ. Αθηνά
Γιάννης Ιωαννίδης, Καθηγητής, ΕΚΠΑ
Ίων Ανδρουτσόπουλος, Καθηγητής, ΟΠΑ

ABSTRACT

While data guides and influences many human activities, the barriers posed by the tools that are needed to retrieve it, such as Structured Query Language (SQL), make data inaccessible for many users. To lift these barriers, researchers have been working on creating natural language interfaces that would allow users to access databases solely through natural language.

Natural language interfaces employ Text-to-SQL systems that can translate a natural language question from the user to an SQL query that can retrieve the data they need. Recently, novel Text-to-SQL systems are adopting deep learning methods with very promising results. At the same time, several challenges remain open, making this area an active and flourishing field of research and development. To make real progress in building Text-to-SQL systems, we need to de-mystify what has been done, understand how and when each approach can be used, and, finally, identify the research challenges ahead of us. We present a detailed taxonomy of neural Text-to-SQL systems that will enable a deeper study of all the parts of such a system. This taxonomy will allow us to make a better comparison between different approaches, as well as highlight specific challenges in each step of the process, thus enabling researchers to better strategize their quest towards the "holy grail" of database accessibility.

However, how can the user verify that the generated SQL query matches their intent if they are not familiar with SQL? To tackle this problem, a system that can translate the SQL query back to natural language is needed (also known as an SQL-to-Text system). We explore the SQL-to-Text problem, we examine its challenges and peculiarities, and present a Transformer-based model that can generate fluent query explanations. Additionally, we look into the difficulties of automatically evaluating the performance of such a system and we examine how different metrics behave in the SQL-to-Text setting.

SUBJECT AREA: Data Democratisation

KEYWORDS: Semantic Parsing, Natural Language Generation, Databases, Deep Learning, Metric Learning, Machine Translation

ΠΕΡΙΛΗΨΗ

Ενώ τα δεδομένα οδηγούν και επηρεάζουν πολλές ανθρώπινες δραστηριότητες, τα εμπόδια που τίθενται από τα εργαλεία που χρειάζονται για να ανακτηθούν, όπως η γλώσσα δομημένων ερωτημάτων (SQL), κάνουν τα δεδομένα μη προσβάσιμα για πολλούς χρήστες. Για να εξαλείψουν αυτά τα εμπόδια, οι ερευνητές έχουν στραφεί προς τη δημιουργία διεπαφών φυσικής γλώσσας που θα επιτρέπουν την πρόσβαση σε βάσεις δεδομένων αποκλειστικά μέσω φυσικής γλώσσας.

Οι διεπαφές φυσικής γλώσσας χρησιμοποιούν συστήματα κειμένου-σε-SQL τα οποία μεταφράζουν τη φυσική γλώσσα από το χρήστη σε ερωτήματα SQL τα οποία ανακτούν τα δεδομένα που ζητάει. Πρόσφατα, νέα συστήματα κειμένου-σε-SQL υιοθετούν τεχνικές βαθιάς μάθησης, δείχνοντας πολύ υποσχόμενα αποτελέσματα. Την ίδια στιγμή, πολλές προκλήσεις παραμένουν ανοιχτές, καθιστώντας αυτήν την περιοχή ένα ενεργό και ανθηρό πεδίο για έρευνα και ανάπτυξη. Για να πετύχουμε αληθινή πρόοδο στη δημιουργία συστημάτων κειμένου-σε-SQL, πρέπει να διαλευκάνουμε όσα έχουν προταθεί, να καταλάβουμε πώς και πότε μπορούμε να χρησιμοποιήσουμε την κάθε μέθοδο, και, τελικά, να αναγνωρίσουμε τις ερευνητικές προκλήσεις που παραμένουν μπροστά μας. Παρουσιάζουμε μια αναλυτική ταξινομία νευρωνικών συστημάτων κειμένου-σε-SQL που θα βοηθήσει στην βαθύτερη μελέτη όλων των μερών ενός τέτοιου συστήματος. Αυτή η ταξινομία θα μας επιτρέψει να κάνουμε καλύτερες συγκρίσεις μεταξύ διαφορετικών προσεγγίσεων, αλλά και να εντοπίσουμε συγκεκριμένες προκλήσεις σε κάθε βήμα της διαδικασίας, βοηθώντας τους ερευνητές να σχεδιάσουν καλύτερα την αναζήτησή τους προς το «ιερό δισκοπότηρο» της προσβασιμότητας στις βάσεις δεδομένων.

Ωστόσο, πώς μπορεί ο χρήστης να επαληθεύσει ότι το ερώτημα SQL που δημιουργήθηκε ταιριάζει με την πρόθεσή του εάν δεν είναι εξοικειωμένος με την SQL; Για την αντιμετώπιση αυτού του προβλήματος, απαιτείται ένα σύστημα που μπορεί να μεταφράσει το ερώτημα SQL στη φυσική γλώσσα (γνωστό και ως σύστημα SQL-σε-κείμενο). Εξερευνούμε το πρόβλημα της μετάφρασης SQL σε κείμενο, εξετάζουμε τις προκλήσεις και τις ιδιαιτερότητές του και παρουσιάζουμε ένα μοντέλο που βασίζεται σε δίκτυα Transformer που μπορεί να δημιουργήσει εύγλωττες επεξηγήσεις ερωτημάτων. Επιπλέον, εξετάζουμε τις δυσκολίες της αυτόματης αξιολόγησης της απόδοσης ενός τέτοιου συστήματος και εξετάζουμε πώς συμπεριφέρονται διαφορετικές αυτόματες μετρικές στα πλαίσια του προβλήματος μετάφρασης SQL σε κείμενο.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Δημοκρατικοποίηση Δεδομένων

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Σημασιολογική Ανάλυση, Παραγωγή Φυσικής Γλώσσας, Βάσεις Δεδομένων, Βαθιά Μάθηση, Μάθηση Μετρικών, Μηχανική Μετάφραση

ACKNOWLEDGEMENTS

This work has been partially funded by the European Union’s Horizon 2020 research and innovation program (grant agreement No 863410). Additionally, this work was supported by computational time granted from the National Infrastructures for Research and Technology S.A. (GRNET) in the National HPC facility - ARIS - under project ID *LMDD*.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

I am very grateful for all the people that played a role, larger or smaller, in this journey: my friends and family for their love, support, and encouragement, as well as my colleagues and collaborators who keep inspiring me to go further, while making work a pleasure. A special thank you to Panorea, for being there for me during the hardest times. My hope is that this pleasant stop is only a part of a larger journey, and the best things are yet to come.

1. INTRODUCTION

In the age of the Digital Revolution, data is now an indispensable commodity that drives almost all human activities, from business operations to scientific research. Nevertheless, its explosive volume and increasing complexity make data querying and exploration challenging even for experts. Existing data query interfaces are either form-based, which are easy to use but offer limited query capabilities, or low-level tools that allow users to synthesize queries in the underlying database query language (e.g., SQL) but are intended for the few (e.g., SQL experts). To empower everyone to access, use, understand, and derive value from data, we need to lift the technical barriers that impede access to data and eliminate dependency to IT experts. Expressing queries in natural language can open up data access to everyone. In the words of E. F. Codd: “If we are to satisfy the needs of casual users of databases we must break the barriers that presently prevent these users from freely employing their native language” [16].

Towards this direction, there has been an increasing research focus on Natural Language (NL) Interfaces for Databases (NLIDBs) that allow users to pose queries in natural language and translate these queries to the underlying database query language. In particular, Text-to-SQL (or NL-to-SQL) systems translate queries from NL to SQL. This is a problem that has troubled researchers for decades [6] and has proven to be notoriously difficult. However, during the past years the introduction of two large Text-to-SQL datasets [136, 131] has opened the door to deep learning techniques, giving new life to this problem. Since then novel neural Text-to-SQL systems are being proposed at a high rate, creating an exciting and competitive research field. In order to understand the current state of the art, its capabilities, and its drawbacks, we believe that a systematic study is needed. For this reason we will present a fine-grained taxonomy of neural Text-to-SQL systems, that will allow us to analyse each system on its own and make better comparisons between systems and design choices. In particular, our contributions on the Text-to-SQL problem are following:

- We present the current state of the deep learning Text-to-SQL landscape, the particularities of the problem, the benchmarks and evaluation methods that are most commonly used, and a wide spectrum of the most recent efforts that leverage the latest and most sophisticated deep learning approaches
- We provide a taxonomy that not only enables a side-by-side comparison of the systems but also allows decomposing the Text-to-SQL problem in a number of subproblems and categorizing existing techniques accordingly
- We provide a detailed discussion of methods used in these systems, taking advantage of our taxonomy to highlight the advantages and shortcomings of different design choices
- We discuss in detail open challenges that are highlighted from our study and provide directions for critical future research

Nevertheless, solving the Text-to-SQL problem is not enough to achieve data democratisation. This is evident if we consider that a non-technical user is not familiar with SQL and as such will not be able to understand and validate the predictions of a Text-to-SQL system. In order for the user to be confident about the queries they run on a NLIDB, it is equally important to explain each query in NL with the use of a SQL-to-Text system.

However, the problem of generating query explanations has received significantly less attention compared to the Text-to-SQL problem. For this reason, our work on this part is directed towards creating a neural system that can generate query explanations, taking advantage of the power of Pre-trained Language Models [22, 92]. Additionally, we discuss the challenges of automatically evaluating query explanations and perform of an evaluation of current metrics, while also proposing a learned metric for this problem. More specifically, on the SQL-to-Text problem, our contributions are the following:

- We propose the use of Pre-trained Language Model for generating SQL explanations, and investigate the use of auxiliary training tasks for improving performance, as well as the use of two adaptation techniques for highly domain-specific databases
- We look into the problem of automatically evaluating explanations of SQL queries, highlighting the drawbacks of available automatic metrics
- We create the *Qx-Paraphrase* and *Qx-Annotate* query explanation datasets, which we use to compare the already available metrics and to fine-tune a learned metric for the SQL-to-Text task
- We perform a user evaluation on our system's predictions so as to have a clear performance indicator both for evaluating our system, but also for evaluating automatic metrics
- We provide insights on our model's performance and common errors by examining predictions from our proposed model

The rest of this thesis is organised as follows: Section 2.1 introduces the Text-to-SQL problem. Section 2.2 provides a definition and explanation of the Text-to-SQL problem, including an analysis of the challenges that make the problem so hard. In Section 2.3, we present the datasets that are currently fueling the creation of deep learning systems. We also touch on the problem of evaluating system performance based on these benchmarks. Section 2.4 presents a fine-grained taxonomy for deep learning Text-to-SQL systems, analysing the most important steps followed by all systems and presenting current work, open problems and hints for future research for each step. Section 2.5 gives an overview of the main neural building blocks used for Text-to-SQL systems, as well as their most common usage. Having established a concrete set of axes for comparing and classifying Text-to-SQL systems, in Section 2.6, a multitude of neural systems are presented and compared based on the aforementioned taxonomy, allowing the reader to grasp the progress that has been made in this domain and the differences between key approaches. In Section 2.7, we take advantage of the taxonomy, to compare different design choices and provide useful insights for researchers and practitioners that are interested in implementing a novel Text-to-SQL system. Section 2.8 aims at inspiring practitioners and researchers in the fields of database systems, natural language processing and deep learning, by shedding light on open problems that need to be addressed, as well as closely related areas that could both give and receive benefit from research done in the Text-to-SQL problem.

Moving on to the SQL-to-Text problem, Section 3.1 introduces the SQL-to-Text problem and Section 3.2 presents related works and compares them to our work. Furthermore, Section 3.3 presents the problem at hand, highlighting its challenges and Section 3.4 presents our SQL-to-Text system. Section 3.5 presents the benchmarks used and created in this work and Section 3.6 discusses the available metrics for the SQL-to-Text problem and their drawbacks, while also proposing a new fine-tuned learned metric for the problem.

Additionally, Section 3.7 contains the experiments that were performed to evaluate our system. Finally, Section 4 concludes the thesis while also providing some directions for future work.

2. THE TEXT-TO-SQL PROBLEM: A SURVEY OF DEEP LEARNING SYSTEMS

2.1 Introduction

The text-to-SQL problem has been the holy grail of the database community for several decades [6]. Early efforts [43, 44, 72, 134] rely primarily on the database schema and data indexes to build the respective SQL query from a NL query. A query answer is defined as a graph where nodes are the relations that contain the query keywords and edges represent the joins between them. *Parsing-based approaches* parse the input question to understand its grammatical structure, which is then mapped to the structure of the desired SQL query [64, 50, 84, 111, 122]. Recently, there has been a growing interest in *neural machine translation (NMT) approaches* [136, 110, 40] that formulate the text-to-SQL problem as a language translation problem, and train a neural network on a large amount of {NL query/SQL} pairs. These approaches have bloomed due to the recent advances in Deep Learning and Natural Language Processing (NLP), along with the creation of two large datasets (WikiSQL [136] and Spider [131]) for training text-to-SQL systems.

As neural text-to-SQL systems are popping up "like mushrooms after a rain" with promising results, an exciting, but, at the same time, highly competitive and fast-paced research field is opening up. While a growing interest on the subject is shown by various tutorials [65, 54, 55] and literature reviews [2, 6, 66, 57, 90, 1, 47, 20] presented at top conferences and journals, an in-depth, systematic study and taxonomy of neural approaches for text-to-SQL is missing. We believe that in order to make real progress in building text-to-SQL systems, we need to de-mystify what has been done, understand how and when each model and approach can be used, and recognize the research challenges ahead of us. Two earlier works [2, 66] study rule-based approaches that originated from the database community; our work has a different scope, focusing entirely on deep learning systems. Additionally, two studies consider both rule-based and neural text-to-SQL systems: [57] provides a taxonomy of both types of systems and an experimental evaluation based on a new accuracy metric proposed by the authors, while [90] provides a large-scale overview of rule-based, neural and conversational NLIDBs. The biggest difference with these works is that we present an in-depth taxonomy tailored to neural systems and their peculiarities (while also covering more and newer efforts). Finally, three studies focus on neural text-to-SQL systems: [1] provides an overview of the neural text-to-SQL landscape, but in a more bare-bones manner compared to our work, and [47, 20], which are the closest to our work, since they both attempt to organise the existing neural text-to-SQL approaches. However, our work goes in greater depth than these works, both by presenting a taxonomy with additional dimensions, but also by using this taxonomy to analyse and compare different systems and design choices. We also point the interested reader to recent surveys on semantic parsing [53] and context-dependent semantic parsing [67], two broader domains that the text-to-SQL problem is a part of.

In a nutshell, this survey aims at catching up with recent advances in deep learning text-to-SQL systems and systematically organising all the different techniques that have been proposed for each step of the translation process. Our objective is to (*a*) put different neural text-to-SQL works in perspective, (*b*) create a fine-grained taxonomy that covers each step of the neural text-to-SQL pipeline, (*c*) explain and organize all the techniques used for each dimension of the taxonomy, (*d*) use the taxonomy to compare and highlight the strengths and weaknesses of different systems and techniques, and (*e*) highlight

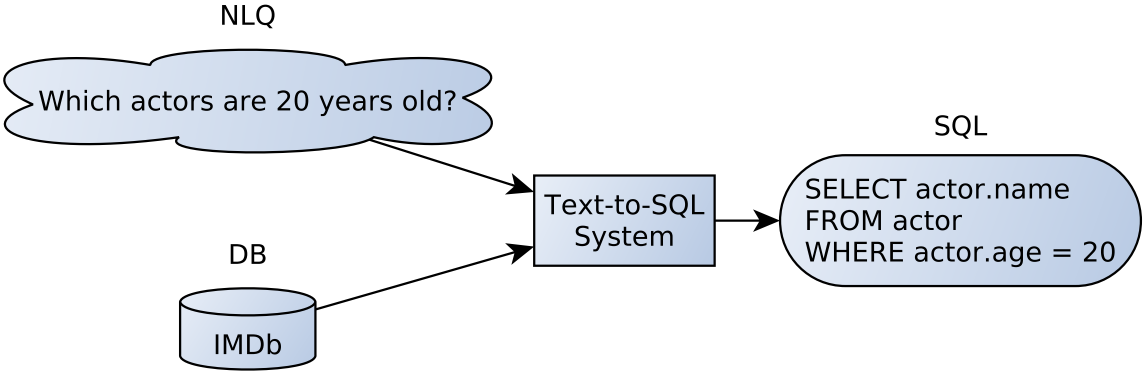


Figure 2.1: The text-to-SQL problem

open challenges and research opportunities for the database and the machine learning communities. Our study is also relevant to other areas, including the broader area of data exploration (e.g., natural language explanations, recommendations), entity resolution, and query optimization, where the methods presented here may be transferred to or inspire the development of new methods.

2.2 The Text-to-SQL Problem

The text-to-SQL problem can be described as follows:

Given a Natural Language Query (NLQ) on a Relational Database (RDB) with a specific schema, produce a SQL query equivalent in meaning, which is valid for the said RDB and that when executed will return results that match the user's intent.

A NLQ may be expressed as a complete and fluent utterance (e.g., *"What movies has Spielberg directed since 2012?"*) or it may be just a few keywords (e.g., *"Italian Restaurants in Vienna"*). A text-to-SQL example can be seen in Figure 2.1. Translating a NLQ to SQL hides challenges related to the understanding of the input NL query as well as related to building the correct (syntactically and semantically) SQL query based on the underlying database schema.

2.2.1 NL Challenges

Ambiguity. Natural language is inherently ambiguous, which means that it allows the formulation of expressions that are open to more than one interpretation. There are several types of ambiguity [4, 80]. We describe the most common ones below.

Lexical ambiguity (or *polysemy*) refers to a single word having multiple meanings. For example, *"Paris"* can be a city or a person.

Syntactic ambiguity refers to a sentence having multiple interpretations based on its syntactic structure. For example, the question *"Find all German movie directors"* can be parsed into *"directors that have directed German movies"* or *"directors from Germany that have directed a movie"*.

Semantic ambiguity refers to a sentence with multiple semantic interpretations. For instance, *"Are Brad and Angelina married?"* may mean they are married to each other or separately.

Context-dependent ambiguity refers to a term having different meanings depending on the query context, the data domain, and the user goals. The most common example terms are *"top"* and *"best"*. Based on *the query context*, for the query *"Who was the best runner of the marathon?"*, the one who completed the race faster (*min* operation) should be returned, but when asking *"Which was the best nation of the 2004 Olympics?"* the one with the most medals (*max* operation) is expected. Based on *the domain*, for the query *"Return the top movie"* on a movie database, *"top"* may mean based on the number of ratings collected. On the other hand, for the query *"Return the top scorer"* on a football database, *"top"* refers to the number of goals scored. Based on *the user*, for a business analyst, the query *"Return the top product"* should return the most profitable products, whereas for a consumer it should return the top-rated products.

Paraphrasing. In natural language, two sentences can have the exact same meaning but be expressed in two completely different ways. For instance, *"How many people live in Texas?"* and *"What is the population of Texas?"*. Both translate to the same SQL query, but the second one may actually be easier for a system because it is likely that a "population" attribute exists in the database schema, and thus, the user intent can be inferred with high confidence. Paraphrasing includes *synonymy* where multiple words have the same meaning (e.g. *"movies"* and *"films"*).

Inference. A query may not contain all information needed for a system to fully understand it. The system has to *infer* the missing information based on the given context. We distinguish two main types of inference:

Elliptical queries are sentences from which one or more words are omitted but can still be understood in the context of the sentence[1]. An example is *"Who was the president before Obama"*. The fact that the query refers to US presidents needs to be inferred.

Follow-up questions are common in conversations between humans. We ask a question, receive an answer, and then ask a follow-up question assuming that the context of the first question is known. For example, *"Q: Which is the capital of Germany?"*, *"A: Berlin"*, *"Q: What about France?"*. In the absence of the first question, the second one does not make sense, but given the query context, it is obvious that it is asking about the capital city of France.

User mistakes. Spelling errors as well as syntactical or grammatical errors make the translation problem even more challenging.

2.2.2 SQL Challenges

SQL Syntax. SQL has a strict syntax, which leads to limited expressivity compared to natural language. There are queries that are easy to express in natural language, but the respective SQL query may be complex. For example, the query *"Return the movie with the best rating"* maps to a nested SQL query.

Furthermore, while a sentence in natural language may contain some mistakes, and still be understood by a human, SQL is not that forgiving. An SQL query translated from a NL query needs to be syntactically and semantically correct in order to be executable over the underlying data.

Database Structure. The user's conceptual model of the data, i.e., the entities, their

[1] `https://en.wikipedia.org/wiki/Ellipsis_(linguistics)`

attributes and relationships that are described in the data, may not match the database schema, and that poses several challenges.

The *vocabulary gap* refers to the differences between the vocabulary used by the database and the one used by the user. For example, in the query *"Who was the best actress in 2011?"*, *"actress"* should map to the *Actor.name* attribute in the database).

Schema ambiguity is when a part of the query may map to more than one database element. For example, *"model"* could refer to *car.model* or *engine.model*.

Implicit join operations occur when parts of a query are translated into joins across multiple relations. For example, *"Find the director of the movie "A Beautiful Mind""* entails joins due to database normalization.

Entity modelling is the problem where a set of entities may be modeled differently, e.g., as different tables or as rows (or values) in a single table. For example, in a university database, every person is either a *Student* or a *Faculty* member, so these two relations suffice. On the other hand, movies have several genres that cannot be stored as different tables. They are stored in a *Genre* relation and are connected with movies through a many-to-many relationship. As a result, similar queries, such as *"Find comedies released in 2018"* and *"Find students enrolled in 2018"* need in fact to be handled differently. The system maps *"comedies"* to a value in the *Genre* table and joins it with the *Movie* table whereas it maps *"students"* to the *Student* relation.

2.3 Datasets & Evaluation

Table 2.1: An overview of Text-to-SQL Benchmarks and their size in queries and databases

Year	Dataset	Queries	Databases
1994	ATIS [89, 18]	275	1
1996	GeoQuery [133]	525	1
2003	Restaurants [105, 85]	39	1
2014	Academic [64]	179	1
2017	IMDb [122]	111	1
	Yelp [122]	68	1
	Scholar [50]	396	1
	WikiSQL [136]	80,654	24,241
2018	Advising [33]	281	1
	Spider [131]	10,181	200
2020	MIMICSQL [113]	10,000	1
	SQUALL [100]	11,276	1,679
	FIBEN [97]	300	1
2021	Spider-Syn [34]	8,034	160
	Spider-DK [35]	535	10
	KaggleDBQA [62]	272	8
	SEDE [41]	12,023	1

To build a neural text-to-SQL system, it is necessary to consider the available datasets for training and evaluation, as well as the evaluation methodology for testing and comparing its performance to other systems. A *text-to-SQL dataset* (or *benchmark*) refers to a set of NL/SQL query pairs defined over one or more databases.

Early system evaluations *did not rely on common datasets*, they rather employed a variety of datasets that combined different databases and query sets of varying size and complexity. In general, the *query sets were small and designed in an ad-hoc way* by the system developers, and as a result it was hard to reach meaningful conclusions about the translation capabilities of a system. Often, the *query sets were proprietary* and hence not available to reproduce the experiments. The *lack of a common dataset* to be used by different system evaluations and the *poor cross-system evaluations* impeded a fair system comparison and a clear view of the text-to-SQL landscape. In addition to these shortcomings, training deep learning text-to-SQL systems requires a substantial query set. As a result, for a long time, the lack of appropriate datasets delayed the adoption of deep learning techniques for the text-to-SQL problem.

This situation drastically changes with the emergence of WikiSQL [136] and Spider [131], in 2017 and 2018 respectively. These are the first large-scale, multi-domain benchmarks that made it possible to train and evaluate neural text-to-SQL systems and provided a common tool to compare different systems easily. While other benchmarks have followed, these two remain the most popular ones. Table 2.2 summarizes and compares the two benchmarks.

This section provides an overview of various text-to-SQL datasets (summarized in Table 2.1), covering either a single or multiple domains, as well as the evaluation methodologies for comparing the system predictions to the ground truth.

Table 2.2: A comparison of the two most popular text-to-SQL benchmarks: WikiSQL and Spider

WikiSQL	Spider
crowd-sourced	created by experts
25K Wikipedia tables	200 databases, 138 domains
80K NL questions	10K NL questions
single-table, simple queries	complex queries
contains errors	higher quality
no query categorization	4 hardness categories

2.3.1 Domain-Specific Text-to-SQL Datasets

Domain-Specific text-to-SQL datasets focus on one domain and typically include a single database, such as: movies and television series (IMDb [122]), restaurant and shop reviews (Yelp [122] and Restaurants [105, 85]), academic research (Scholar [50] and Academic [64]), financial data (Advising [33] and FIBEN [97]), medical data (MIMICSQL [113]), and questions and answers from Stack Exchange (SEDE [41]).

Interestingly, these datasets have not seen the same widespread use as WikiSQL or Spider for a number of reasons. Since they focus on a single domain, it is not possible to argue that a proposed system can be considered a “universal solution” even if it performs well on a specific domain. Second, their size is relatively small compared to Spider and WikiSQL, usually not surpassing a thousand examples. Third, most of these datasets do not have a pre-defined train/dev/test split so that systems trained and evaluated on them would be compared fairly to one another.

Player	No.	Nationality	Position	Years in Toronto	School/Club Team
Leandro Barbosa	20	Brazil	Guard	2010-2012	Tilibra
Muggsy Bogues	14	USA	Guard	1999-2001	Wake Forest
Jerryd Bayless	5	USA	Guard	2010-2012	Arizona
⋮	⋮	⋮	⋮	⋮	⋮

NLQ: What nationality is the player Muggsy Bogues?

SQL: SELECT nationality WHERE player = muggsy bogues

Figure 2.2: An example from the WikiSQL dataset.

! Late 1941	Late 1942	Sept. 1943	Late 1943	Late 1944	1978 Veteran membership
Croatia	7000	48000	78000	122000	150000
Slovenia	2000	4000	6000	34000	38000
Serbia	23000	8000	13000	22000	204000
⋮	⋮	⋮	⋮	⋮	⋮

NLQ: Name the most late 1943 with late 194 in slovenia

SQL: SELECT max(late 1943) WHERE ! late 1941 = slovenia

Figure 2.3: An incoherent example from the WikiSQL dataset.

Even though the generalisation capability of a text-to-SQL model is an important challenge, a realistic application would most likely require a text-to-SQL system to work with a single database of a specific domain, or with a few related databases. In such a scenario, a high performance on a single domain may be even more important than a cross-domain generalisation capability, and achieving it is very challenging [41].

Furthermore, datasets such as SEDE[41], are made specifically to reflect that SQL queries in real-life scenarios can be very complex and long; having numerical computations, variable declarations, date manipulations, and other elements that are not present in the Spider and WikiSQL datasets. SEDE's authors demonstrate that the state-of-the-art systems which achieve high scores on Spider, do not perform as well on SEDE, proving the necessity for new and more advanced benchmarks.

2.3.2 Cross-Domain Text-to-SQL Datasets

WikiSQL. WikiSQL [136] is a large crowd-sourced dataset for developing natural language interfaces for relational databases, released along with the Seq2SQL text-to-SQL system. It contains over 25,000 Wikipedia tables and over 80,000 natural language and SQL question pairs created by crowd-sourcing. Each entry in the dataset consists of a table with its columns, a Natural Language Question (NLQ) and a SQL query. Figure 2.2 shows an example from the dataset.

The complexity of the SQL queries found in WikiSQL is low because each query is directed to a single table and not to a relational database and they are do not use any complex SQL clause such as JOIN, GROUP BY, ORDER BY, UNION, and INTERSECTION. Additionally, WikiSQL does not allow the selection of multiple columns in a single query or the use of the asterisk (*) operator. Consequently, the proposed task is much simpler than the ultimate goal of creating a natural language interface for relational databases.

We must also note that WikiSQL contains multiple errors and ambiguities, which might hinder the performance of a model trained on it. Figure 2.3 demonstrates an example of a table incorrectly copied from Wikipedia that was nevertheless used to generate a pair of a NLQ and a SQL query that, ultimately, make no sense. Research even suggests that the state-of-the-art systems have reached the upper barrier of accuracy on the task [46]. This is also demonstrated by evaluating human performance on a small proportion of the dataset.

Spider. Spider [131] is a large-scale complex and cross-domain semantic parsing and text-to-SQL dataset annotated by 11 Yale students. It contains 200 relational databases from 138 different domains along with over 10,000 natural language questions and over 5,000 SQL queries. Its queries range from simple to hard, using all the common SQL elements, including nesting. These characteristics of the dataset along with its high quality, since it was hand-crafted and re-checked, have led researchers to widely rely on it for building systems that can generate quite complex SQL queries.

Other Cross-Domain Datasets. Recent cross-domain datasets focus on particular aspects of the text-to-SQL problem. Spider-DK [34] extends Spider to explore system capabilities at cross-domain generalization (i.e., robustness to domain-specific vocabulary across different domains), while Spider-Syn [34] focuses on robustness to synonyms and different vocabulary. Both datasets highlight very interesting and important requirements for a text-to-SQL system, and can be used as supplementary benchmarks.

SQUALL [100] is based on a previous dataset named WikiTableQuestions [82], consisting of NL Questions posed on Wikipedia tables along with the expected answers. In contrast to WikiSQL, there are no structured queries in the WikiTableQuestions dataset. The authors of SQUALL have created the corresponding SQL queries for most of the examples in the WikiTableQuestions dataset, while also providing an alignment between words in the NLQ and the parts of the SQL query that they refer to. This additional feature could steer more thorough research on the schema linking and schema ambiguity problems (briefly mentioned in Section 2.2 and more thoroughly examined in Section 2.4).

Finally, KaggleDBQA [62] is another cross-domain dataset, although of much smaller size, that has been extracted from Kaggle and features real-world databases taken from the Web, having all the peculiarities of a DB that are missing from Spider, whose DBs were created specifically for benchmarking text-to-SQL systems. KaggleDBQA also includes documentation and metadata for its DBs, posing an interesting research question of how this additional information could be used to improve the system performance.

2.3.3 Evaluation Metrics

Having a ground truth SQL query for each NLQ enables us to train and evaluate a deep learning text-to-SQL system on it. In this section, we will present metrics used to evaluate a text-to-SQL system's predictions.

String Matching (introduced as Logical Form Accuracy [136]) is the simplest accuracy metric for text-to-SQL. It considers the ground truth and predicted queries as simple strings and checks whether they are identical. A match is only found when the predicted query is written exactly as the ground truth, without taking into account that many parts of a SQL query can be written in a different order or even in a different but still equivalent way.

Execution Accuracy [136, 131] (or Query Accuracy [13]) is another simple approach for comparing SQL queries. For each NLQ, both the ground truth and the predicted queries are executed against the corresponding database (or table) and their results are compared. If the results are the same, then the prediction is considered correct. False positives can occur when both queries return the same results, but are different on a semantic level (e.g., when they return empty results or when an aggregation function is applied to different columns that happen to return the same result).

Component Matching [131] is proposed in order to obtain a better understanding of which parts of the SQL query are predicted correctly. For example, we might consider the SELECT column accuracy, i.e., the percentage of the predicted queries that have the same columns in the SELECT clause as the corresponding ground truth queries. For some parts, a more sophisticated approach might be necessary to avoid incorrect classifications. For instance, when comparing the conditions of the WHERE clause, their order should not be taken into account.

Exact Set Matching [131] (or Query Match Accuracy [120]) considers all the possible component matches and classifies a prediction as correct if all component matches are correct (e.g., aggregation function, condition operators, SELECT columns, etc.).

Exact Set Match Without Values is a category in the Spider [131] dataset, that works in the same way as exact set matching, but does not take into account if the values that appear in the predicted query are the same as the ones that appear in the gold query. The reason for this simplification is that predicting the correct values can be very challenging, especially when these values appear in the NLQ differently to the way they are stored in the DB (e.g., the word "Greek" might imply a condition such as country="Greece"). Although this metric might be considered as common practice in the Spider benchmark, as research shows [33], disregarding values during evaluation removes an important challenge of the text-to-SQL problem.

Sub-tree Elements Matching (or Partial Component Match F1 - PCMF1) [41] is a metric proposed to avoid a score of zero by the exact set match metric, when some parts of the predicted query are correct. It considers parts of the query such as the SELECT, WHERE, FROM, etc. clauses and it calculates the F1 score of each clause based on the precision and recall of the predicted attributes in the clause. The final PCMF1 score of a predicted query is the average F1 score of all the considered query parts. For example, in large queries, the system might predict a large part of the query correctly and make some errors in the WHERE clause. While the exact match metric would assign a score of zero even for a small mistake, the PCMF1 metric would assign a score relatively close to one, thus providing a better assessment of the system performance.

A more thorough methodology for evaluating the semantic equivalence of two SQL queries has been proposed by [57], but has yet to be adopted by any deep learning systems. This approach starts by comparing the execution result of the two queries, as well as their results on additional generated data, in case the original database contains a small amount of data. Furthermore, a prover is used to provide a proof of equivalence between the queries or a counter example in the case of non-equivalence. If the prover cannot work for the given queries, then a query re-writer is applied on both queries and the re-written queries' parse trees are compared. If the re-written parse trees are structurally identical then the queries are semantically equivalent, otherwise the queries are manually evaluated by an expert. While this approach could detect matches even if queries are expressed in fundamentally different ways, the requirement of manual labor as well as the extra processing requirements it presents, are some of the reasons why it has not seen widespread use yet.

What metric each system is using greatly depends on the dataset that each system is created for and aims at entering its leaderboard[2,3]. Specifically, systems that are built for

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

[3] https://github.com/salesforce/WikiSQL

the WikiSQL dataset, use Logical Form Accuracy and Execution Accuracy, while systems built for the Spider dataset use Exact Set Matching without Values and Execution Accuracy. This strongly indicates the influence that benchmark creators have on the evaluation strategy of text-to-SQL systems. It also highlights the responsibility of the next benchmark creators to address the problems of current metrics and include more thorough evaluation metrics.

2.4 Taxonomy

Despite the fact that deep learning approaches have only recently become popular for the text-to-SQL problem, numerous systems have already been proposed, that bring a wide variety of novelties and employ different approaches. Nevertheless, there are key parts that serve common purposes across almost all systems, which allow us to build a general model that can help us better understand them. Hence, the goal of this section is to present an overview of the most important parts of neural text-to-SQL systems as well as a taxonomy of the possible choices in each part.

Figure 2.4 shows an overview of a neural text-to-SQL system. The main input of a text-to-SQL system is a NL query (NLQ) and the database (DB) that the NLQ is posed on. The first step, whenever employed, is *Schema Linking*, which aims at the discovery of possible mentions of database elements (tables, columns and values) in the NLQ. These discovered schema links, along with the rest of the inputs, will be fed into the neural network that is responsible for the translation.

The core of this neural network consists of two main parts: the *encoder* and the *decoder*. The encoder takes one or more inputs of variable shapes and transforms them into one or more internal representations with fixed shapes that are consumed by the decoder. Additionally, the encoder usually infuses the representation of each input with information from the rest of the inputs, so as to create a more informed representation that better captures the instance of the problem at hand. The decoder uses the representations calculated by the encoder and makes predictions on the most probable SQL query (or parts of it).

Given that the inputs (NLQ, DB, schema links) are mainly textual, *Natural Language Representation* is responsible for creating an efficient numerical representation that can be accepted by the encoder. *Input Encoding* is the process of further structuring the inputs in a format that can be accepted by the encoder, as well as the choice of an appropriate encoder network for processing them and producing an internal hidden representation. Finally, *Output Decoding* consists of designing the structure of the predictions that the network will make, as well as choosing the appropriate network for making such predictions (e.g., a SQL query can be viewed as a simple string, or as a structured program which follows a certain grammar). While some systems perform the *NL Representation* and *Encoding* steps separately (e.g., a representation based on word embeddings which is then encoded by a LSTM), in some cases, they can be almost indistinguishable (e.g., when using BERT [23]). It is even possible for all three steps to be merged into one (e.g., when using the T5 encoder-decoder pre-trained language model [92]). Finally, the *neural training* refers to the procedure followed for training the neural network.

The last dimension of the taxonomy is the *Output Refinement*, which can be applied during the decoding phase in order to reduce the possibility of errors and to achieve better results. Note that even though Output Refinement is closely related to Output Decoding and even

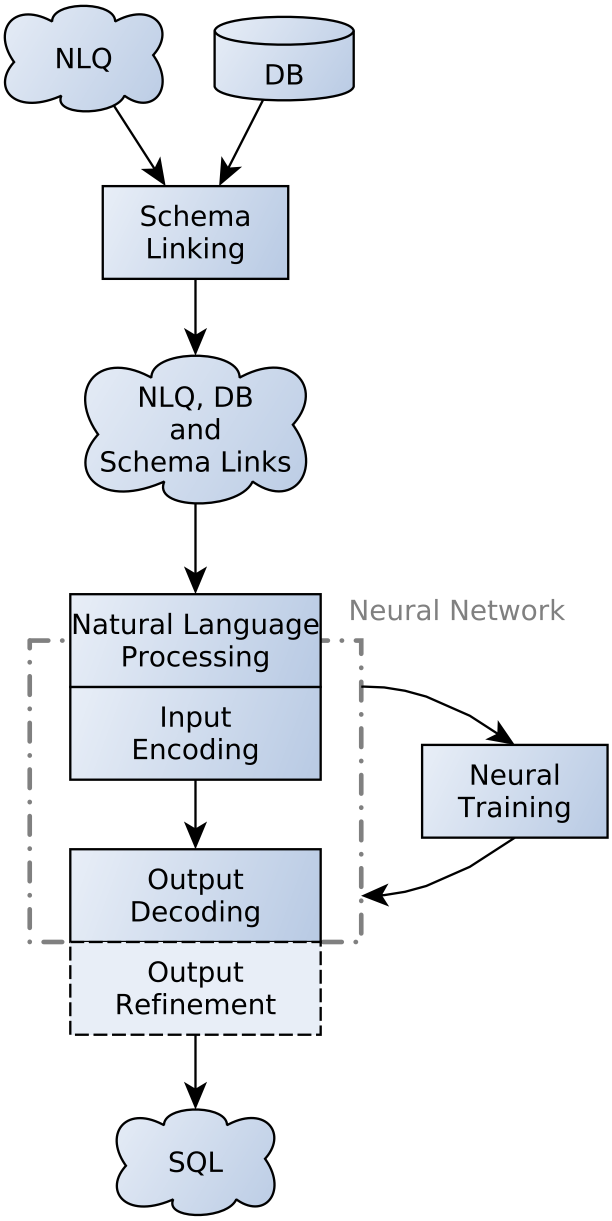


Figure 2.4: Overview of a neural text-to-SQL system, based on the proposed taxonomy

interacts with the decoder, it is not a part of the neural network. As such, in most cases, it is possible to add or remove an output refinement technique once the system has been created and trained.

2.4.1 Schema Linking

To better grasp the concept of schema linking, let us think of how a human, asked to write a SQL query from a NLQ, would start by looking at the underlying database and by trying to identify how the entities mentioned in the NL are stored in the database. In other words, they would attempt to *link* parts of the NLQ to the database elements they are referring to. Intuitively, a text-to-SQL system could benefit by doing the same when translating a NLQ.

More formally, schema linking is the process of discovering which parts of the NLQ refer to

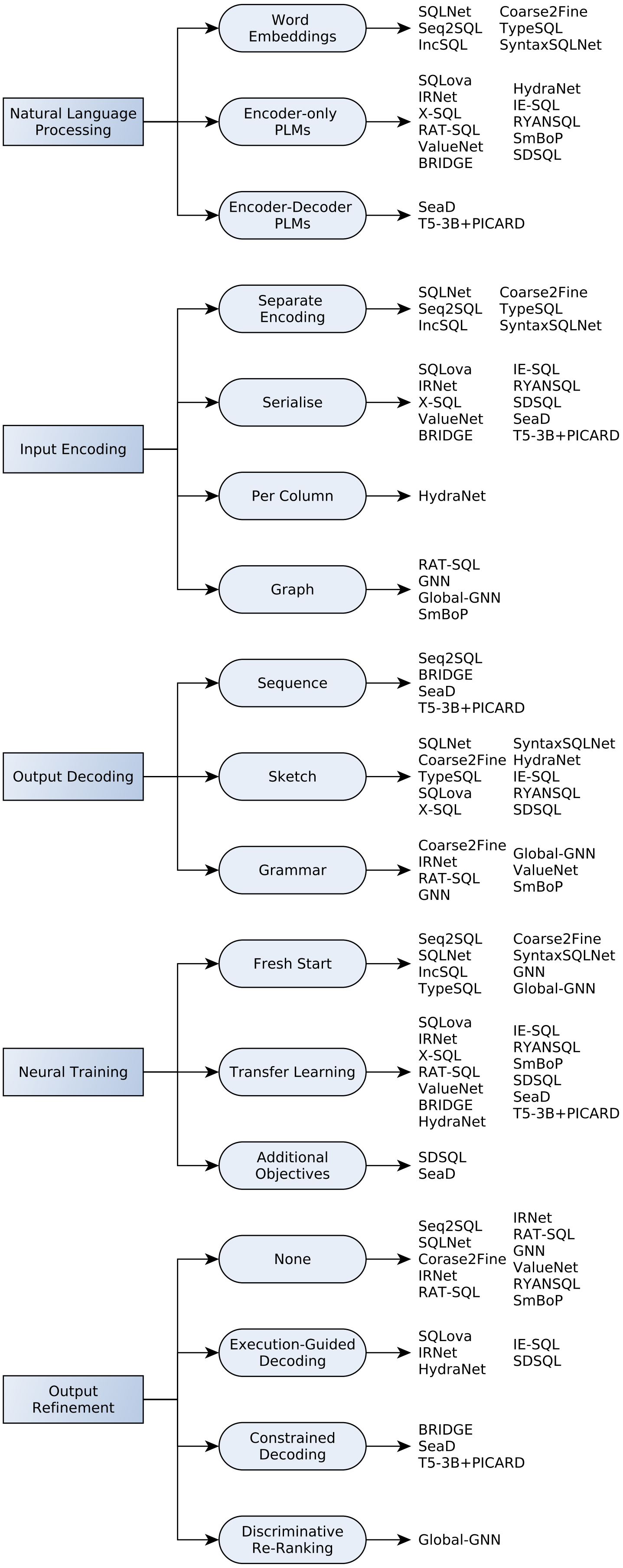


Figure 2.5: System Categorisation on the taxonomy dimensions of Natural Language Representation, Input Encoding, Output Decoding, Neural Training and Output Refinement

which database elements. The NLQ parts that could possibly refer to a database element are called *query candidates*, while the database elements that could occur in the NLQ are called *database candidates*. Query candidates can be words or phrases, while database candidates can be tables, columns, and values in the database. A connection between a query candidate and a database candidate is called a *schema link*, which can be further categorized as a *table link* or *column link*, when the query candidate maps to a table name or column name, respectively, and *value link*, when it matches a value of a column.

Schema linking is very challenging for a variety of reasons. Query and database candidates may not use the same vocabulary nor appear in the exact same phrasing. For example, the phrase *"sang by"* in the NLQ might refer to the database column *"singer"* (same word stem, phrased differently) or *"artist"* (vocabulary mismatch). This problem is even more challenging when the NLQ expresses a condition (i.e., a reference to a DB value) in a different way than how the value is stored in the DB. This is an issue because in contrast to the table and column names of the DB, the sheer volume of data stored in a DB prohibits using all DB values as inputs to the system, making it very challenging for the system to build the correct SQL condition. For example, the word *"female"* might imply a condition such as *"gender=F"*. In this case, besides a schema link between *"female"* and the column *"gender"*, the system must also be given the value as it is stored in the DB (*"F"*) as part of the input, in order to use it when constructing the SQL prediction. Otherwise, it will most likely produce a condition like *"gender=female"*, which would return no rows. Due to the volume of a DB, finding value links is not only hard but can be very computation-expensive.

The schema linking process has two parts. *Candidate discovery* is the process of extracting query candidates from the NLQ and database candidates from the underlying database. *Candidate matching* is the process of comparing a set of query candidates and a set of database candidates and establishing the links.

Schema linking enhances the input, and a system can operate without it. Hence, performing no schema linking is possible too. In fact, while most recent systems incorporate some form of schema linking in their workflow, earlier ones (e.g., Seq2SQL [136], SQLNet [120]) and even some recent ones (e.g., HydraNet [74], T5+PICARD [95], SeaD [121]) simply rely on their neural components to make predictions.

2.4.1.1 Query Candidate Discovery

We first walk through the techniques used for discovering query candidates.

Single Tokens. A simple approach for finding query candidates is to consider all the single words of the NLQ as query candidates. This is obviously prone to errors as it is likely that a query candidate spans over multiple tokens (e.g., *"New York"*, *"Iggy Pop and the Stooges"*).

Multi-word Candidates. To find all possible query candidates, even multi-word ones, it is necessary to consider n-grams of varying length. For example, IRNet [40] uses all n-grams of length from 1 to 6 in the user question as query candidates. It processes them in descending order of length and if a n-gram is marked as a schema link, the system discards all the smaller n-grams that are contained in it, to avoid generating duplicate links. Furthermore, IRNet [40] assumes that any phrase (n-gram) appearing inside quotes must be a reference to a value stored inside the database. Note that in this case, the system not only discovers a query candidate, but also asserts that the database candidate that will be linked to it must be a value.

Named Entities. ValueNet [12] adds an extra step for intelligent candidate discovery, by performing Named Entity Recognition (NER) on the user's NLQ to discover possible query candidates. This technique is very effective in discovering candidates that refer to a widely known entity such as a place or a person but might not generalize to entities that are specific to a certain domain. ValueNet asserts that candidates discovered through NER refer to a DB value, i.e., the DB candidate they will be matched to, must be a value. TypeSQL [127] uses the Freebase[4] Knowledge Graph to perform NER. It searches for five types of entities, namely: *Person, Place, Country, Organization and Sport*. However, the query candidates that are found to be Named Entities are not matched to a DB candidate, but simply marked with the entity type that describes them.

Additional Candidates. As mentioned earlier, creating correct conditions can be even more challenging when the value is not expressed in the NLQ exactly as it is stored in the DB. ValueNet [12] proposes an improved pipeline for generating additional candidates for value links that consists of: (*a*) identifying possible query candidates using NER, (*b*) generating additional candidates by looking up similar values in the database and by using string manipulation, and (*c*) validating all the generated candidates by confirming they appear in the database. The validated candidates are then given to the system, to aid it in generating correct conditions. Let us consider the following example, where the NLQ contains the phrase *"New York"*, but the DB contains the value *"NY"*. ValueNet would recognize *"New York"* as a named entity, it would generate additional similar candidates (e.g., *"N. York"*, *"N.Y."* and *"NY"*) and it would look them up in the DB. Doing so, it would discover that only *"NY"* appears in the DB, and would only add this value in the input to help the system create a correct condition (e.g., *"state=NY"*).

2.4.1.2 Database Candidate Discovery

Table and Column Names. The first and most obvious source for database candidates are the names of the tables and columns of the database. Given that most databases contain a relatively small number of tables and columns, all of them can be database candidates.

Values via Lookup. Values stored in the database comprise another large pool for database candidates. However, due to the volume of data, iterating over all the DB values is not performance-wise. Indexes have been widely used in earlier text-to-SQL systems, which do not rely on deep learning [43, 64], to accelerate the search. ValueNet [12] also uses indexes and computationally cheap methods for retrieving values from the DB. It is necessary to note that a database lookup requires the use of an already discovered query candidate. In order to avoid greedily looking up all the query candidates, the system might only look up certain query candidates that seem more likely to refer to a value (e.g., because the are found inside quotes or based on heuristics).

Values via Knowledge Graphs. IRNet [40] assumes that access to the database contents is not possible and employs the knowledge graph ConceptNet [103] for recognizing value links. As a first step, IRNet considers that all n-grams beginning and ending with single quotes are query candidates referring to values. In order to discover the DB column or table that could contain a value such as the discovered query candidate, the system searches each candidate in the knowledge graph and only keeps two types of results: *is-*

[4] https://developers.google.com/freebase

type-of and *related-terms*. For example, when searching for *“New York”* in ConceptNet, one of the returned results is *is-type-of “state”*. This result helps IRNet link *“New York”* to a column named *“state”* or similarly. Note that this approach stands out from what has been discussed so far, in the way that a value link is discovered using an intermediate candidate (knowledge graph result) and the column names.

2.4.1.3 Candidate Matching

Having discovered the query and database candidates, an efficient method is needed for comparing them to identify possible links. As discussed earlier, candidates are not always expressed in the same way in both sides, so identifying links is not straightforward. Techniques that can recognize semantic similarities between candidates are required.

Exact and Partial Matching. The simplest approach is to look for exact and partial matches, as it is done by IRNet [40]. An exact match requires that the candidates are identical, while a partial match occurs when one candidate is a substring of the other. Admittedly, this approach is bare-bone and while it can discover more obvious links, it can also result in false positive matches when candidates share the same words (e.g., *“residence”* would be considered a partial match with *“former residence”*).

Fuzzy/Approximate String Matching. Another useful technique for identifying matches when the link in the candidates are written differently is approximate string matching. An example of such an approach is the Damerau-Levenshtein distance [19], used by ValueNet [12]. While such techniques aid at identifying matches with different spelling or spelling mistakes (e.g., *“color”*-*“colour”*), they cannot handle synonyms and thus are not robust to the use of different vocabulary.

Learned Embeddings. To calculate the similarity between words of the NLQ and schema entities, an earlier work in the area of semantic parsing [60] proposes the use of learned word embeddings. The system learns word embeddings using the words of the text-to-SQL training corpus and combines them with additional features that are calculated using NER, edit distance and indicators for exact token and lemma match. These embeddings are then used to calculate the similarity of query candidates to DB candidates. While this approach is more expensive than previous matching techniques, it allows for much more flexible and intelligent matching. This approach was also adopted by text-to-SQL systems [10, 11] as well.

Classifiers. Given the complexity of schema linking, it may be possible to achieve better results by training a model to perform schema linking.

A Conditional Random Field (CRF) model [61] can be trained on a small group of hand-labelled samples to recognize column links, table links and value links for numerical and textual values [13]. The predictions of this model can then be passed to the main neural network of the text-to-SQL system along with the rest of the inputs. DBTagger [107] uses a similar approach to solve the schema linking problem as a sequence tagging problem. It employs CRFs on every token of the NLQ to identify: (*a*) its Part of Speech (POS), (*b*) schema link type (e.g., table link, value link, etc.), and (*c*) the specific schema element that it refers to. The authors argue that learning these three tasks in a multi-learning paradigm helps the system achieve better performance than it would if it only learned to identify the schema element each token refers to.

The SDSQL [45] system is simultaneously trained on two tasks: (*a*) the text-to-SQL task, similarly to all systems, and (*b*) the *Schema Dependency Learning* task. For this additional learning task, the system is essentially trained to discover schema links in the form of dependencies between the words of the NLQ and the parts of the SQL query. Namely, the possible dependencies are: select-column (S-Col), select-aggregation (S-Agg), where-column (W-Col), where-operator (W-Op) and where-value (W-Val). For example, a select-column (S-Col) label is assigned to the dependency between the column appearing in the SELECT clause and the word of the NLQ that refers to it. A deep biaffine network [27, 29] is trained along the rest of the system to detect the existence and type of these dependencies. Training data for this task is created from the already available NL and SQL pairs, by assigning dependency labels between the NLQ tokens and table columns. Although the schema links discovered by the system are not directly used for predicting the SQL query, training for both tasks simultaneously has a positive effect on the system performance. This task goes beyond the schema linking task, as some of the aforementioned dependencies include query candidates that might refer to query parts (e.g., aggregation functions and condition operations). It should also be noted that this approach has been applied to WikiSQL, but it has not yet been extended to the more challenging Spider dataset.

Neural Attention. While attention layers do not directly determine a match, we mention them briefly because of their capability to highlight connections between query and DB candidates, which can improve the system's internal representation and boost its performance. SQLNet [120] was the first system to introduce such a mechanism, named *Column Attention*, that processes the NLQ and column names and finds relevant columns for each word of the NLQ. The Transformer [108] neural architecture, which is based on an attention mechanism, has been instrumental to the widespread use of PLMs that have become the go-to solution for input encoding, greatly benefiting the accuracy of text-to-SQL systems. Finally, RAT-SQL [110] proposed a modified Transformer layer, called Relation-Aware Transformer (RAT), that biases the attention mechanism of the Transformer towards already-known relations from the DB schema and discovered schema links.

2.4.2 Natural Language Representation

An essential step for text-to-SQL systems is creating and processing numerical representations of their NL inputs. Until recently, the most popular technique for NL representation has been pre-trained word embeddings. Recent advances in NLP, such as the introduction of the Transformer architecture [108] followed by its use to create large Pre-trained Language Models (PLMs), has tipped the scales greatly to its favour. Additionally, as new PLMs are emerging, a new research path is being paved focusing on the design of better PLMs or PLMs created specifically for certain problems (such as the text-to-SQL problem).

2.4.2.1 Word Embeddings

Word embeddings aim at mapping each word to a unique numerical vector. While there are simplistic approaches for creating such vectors (e.g., one-hot embeddings), more advanced algorithms [78, 83] aim at making the value of each vector meaningful. These vectors are usually trained from a large text corpus (e.g., Wikipedia or Twitter) using a self-supervised algorithm that is mainly based on word co-occurrences. The set of pre-trained vectors can then be used to build a model that benefits from the inherent knowledge that is present in the vectors due to their training.

For example, the GloVe [83] embeddings, which capture interesting word relationships, were frequently used by the first text-to-SQL systems. Such word relationships include words with similar meaning being near neighbors and linear substructures that indicate similar relationships between words (e.g., the distances between the word pairs Paris-France and Athens-Greece will be similar because these words share a capital-country relation). A pre-trained set of GloVe embeddings can be used to create numerical representations for NL inputs of a model, which can then be encoded using a RNN (such as a LSTM).

2.4.2.2 Pre-trained Language Models

The introduction of the Transformer architecture [108] and its use in PLMs such as BERT [23] has led to a great performance boost in many NLP problems. The text-to-SQL problem is no exception, as the use of PLMs has quickly become the go-to solution for NL representation. In order to understand how a PLM can be used in a text-to-SQL system, it is first necessary to highlight the difference between two main categories of PLMs: (*a*) encoder-only and (*b*) encoder-decoder models.

Encoder-only models, like BERT [23], RoBERTa [71], and TaBERT [125], take a sequential input and produce a contextualized numerical representation for each input token. The term “contextualized” marks a notable difference to word embedding techniques, which map each word to a fixed vector, while the representations given by PLMs are computed taking all tokens of the input into account. This representation can then be used by additional neural layers to make a prediction for the downstream task at hand. While GloVe representations can be seen as improved word embeddings and can be used in similar fashion (e.g., using an LSTM), this is not necessary. In fact, due to the robustness of PLMs, it is possible to process their outputs using very simple and small neural networks and still achieve better results than complex networks using word embeddings.

Encoder-decoder models, like T5 [92] and BART [63], are full end-to-end models that take a sequential text input and return a sequential text output (seq-to-seq). These models produce the final output on their own, without the need for any extra neural layers, and can be used on any downstream task as long as the expected output can be modeled as a text sequence.

Furthermore, as such models are gaining more attention, the creation of task-specific PLMs is becoming a new research area of its own. Such models can be customized to work with different types of inputs and perform better on less generic tasks, such as the text-to-SQL task. There are multiple PLMs, such as GraPPa [128] and TaBERT [125], that have been designed to work with structured and tabular data as well as to better generalize in tasks that use SQL, and they can improve the performance of a text-to-SQL system when used in place of a generic PLM. It must also be noted that while most text-to-SQL systems are originally proposed with BERT [23] or another general-purpose PLM, they often manage to achieve higher scores by replacing it with a PLM, such as TaBERT [125], that was specifically pre-trained for a task that uses structured data, like the text-to-SQL task.

2.4.3 Input Encoding

The dimension of input encoding examines how the input is structured and fed to the neural encoder of the system, so that it can be processed effectively. There are dif-

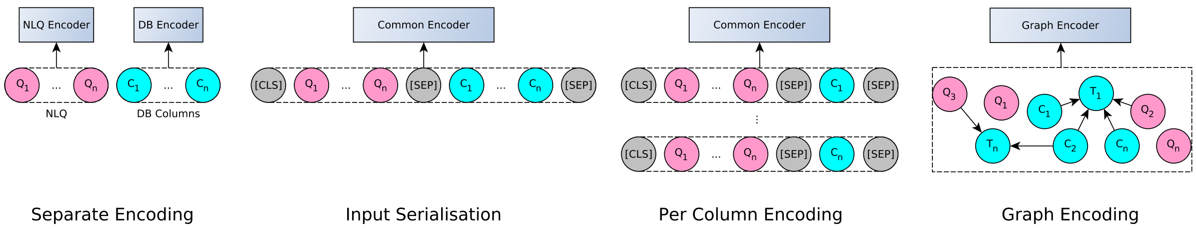

NLQ Encoder
DB Encoder
Q_1 ... Q_n
C_1 ... C_n
NLQ
DB Columns
Separate Encoding
Common Encoder
[CLS] Q_1 ... Q_n [SEP] C_1 ... C_n [SEP]
Input Serialisation
Common Encoder
[CLS] Q_1 ... Q_n [SEP] C_1 [SEP]
[CLS] Q_1 ... Q_n [SEP] C_n [SEP]
Per Column Encoding
Graph Encoder
Q_3 Q_1 C_1 T_1 Q_2 T_n C_2 C_n Q_n
Graph Encoding

Figure 2.6: An overview of the possible encoding choices. Pink tokens represent words of the NLQ, blue tokens represent database elements and grey tokens are auxiliary tokens.

ferent inputs that are useful for translating a NLQ to SQL. The NLQ and the names of the DB columns and tables could be considered the minimum required input. Other features that could improve the network performance include: (*a*) the relationships present in the DB schema, including primary-to-foreign key relationships and relationships between columns and tables, and (*b*) links and additional values that have been discovered during the schema linking process.

The use of neural networks mandates the transformation of all inputs into a form that can be accepted by the network. This can be very restrictive, given how heterogeneous these types of inputs are and how difficult it is to represent them all in a single type of input. In this section, we examine the most representative choices for input encoding, while also taking into account the additional features that each choice can incorporate. We distinguish four encoding schemes: (*a*) separate NLQ and column encodings (*b*) input serialization (*c*) encoding NLQ with each column separately, and (*d*) schema graph encoding. A schematic overview of the possible encoding choices can be seen in Figure 2.6.

2.4.3.1 Separate NLQ and Column Encodings

A first approach, used mostly by earlier systems (e.g., Seq2SQL [136], SQLNet [120]), is to encode the NLQ separately from the table columns. The main reason for encoding the two inputs separately is the shape mismatch between them; while the NLQ is a simple sentence (i.e., a sequence of words), the table header is a list of column names, where each name can contain multiple words, i.e., it is a sequence of sequences of words.

In Seq2SQL [136], SQLNet [120] and IncSQL [99], each word (embedding) of the NLQ is fed into a bi-directional LSTM (bi-LSTM) that produces a hidden state representation for each word. For column headers, since each column name can have multiple words, a bi-LSTM is used for each column name, and the final hidden state of each column is used as the initial representation for the column. Notice that by keeping only the last state of each column name, the representation of the header becomes a simple sequence and not a nested sequence. Since the two inputs are encoded separately, they must be combined at some point so that the output is influenced by both of them. This can be done by using cross-serial dot-product attention [73], concatenating the two representations, summing them or using a combination of the above.

None of the studied systems that follow this encoding approach use any extra features besides the NLQ and DB columns. This may be attributed to the fact that these are some of the earliest neural text-to-SQL proposals, which did not perform schema linking and focused on the simpler WikiSQL dataset.

2.4.3.2 Input Serialisation

A different approach is to serialise all the inputs into a single sequence and encode it all at once. This is a very common practice when using PLMs (e.g. BERT [23], T5 [92]) that create a contextualized representation of their input, because if each input were to be encoded separately, the system would not benefit from the PLM's contextualization ability. This approach simplifies the encoding process and benefits from the robustness of PLMs. However, it also carries disadvantages, such as losing schema structure information and being unable to easily represent relationships between the inputs (e.g., primary-foreign key relationships, schema links, etc.). As we go through some different serialisation approaches, we will also examine how much information can be retained in each case.

It should also be noted that PLMs usually employ a few special tokens that are added to the serialised sequence. For example, BERT [23] uses the classification [CLS] and the separating [SEP] special tokens. The [CLS] token is added at the start of the sequence. Its contextualized output, which gathers information from all the tokens in the sequence thanks to the underlying attention networks, can be used to make classification predictions that concern the entire sequence. The [SEP] special token can be used to separate different sentences in the same sequence. These tokens are also useful for the text-to-SQL problem.

The simplest serialisation technique, used by several systems [42, 46, 76] that work on the WikiSQL dataset, creates a single input sequence that only contains the NLQ and all the table headers. The serialised sequence starts with the [CLS] token, as is common for BERT, then the NLQ tokens are appended, followed by a [SEP] token marking the end of the NLQ and then each column name is added followed by a [SEP] token. This input is processed by BERT, which creates a contextualized representation that has the same length as the input, and that can be processed by the rest of the network to make predictions. Since these systems only work with single tables, there is not a lot of information that needs to be preserved, but it could be argued that this approach separates the column names much less strictly compared to the separate encoding approach.

IRNet [40] (when using BERT) creates an input that starts with a [CLS] token, then continues with the NLQ's tokens followed by a [SEP] token, the name of each column of the database followed by a [SEP] token, and finally the table names of the schema, each separated with a [SEP] token as well. In order to encode discovered schema links along with the rest of the input, IRNet uses three extra tokens, namely [Column], [Table], [Value], that can be appended before a NLQ token or phrase, to mark that it was linked to a database candidate. Still, using this serialisation format, there is a lot of schema information not captured. For example, it is not possible to extract any primary-foreign key relationships, or to which table each column belongs.

Finally, BRIDGE [69] constructs an input for a PLM that starts with a [CLS] token, followed by the NLQ and a [SEP] token, as well as the tables and column of the DB, where a [T] and [C] token is added before each table and column name, respectively, so as to better preserve each attribute's role. The difference between IRNet's and BRIDGE's use of the special [C]/[Column] and [T]/[Table] tokens is that the former uses them in the NLQ part to indicate a schema link to a column or table, while the latter uses them to indicate that the tokens after a [C] or [T] token are a column or table name, respectively. BRIDGE also uses an extra third token [V] along with a value, after a column name, to mark that this value appears under the column at hand and was discovered as a possible value link to some NLQ candidate. In this case, BRIDGE uses the [V] token in the DB schema part of the input while also appending a value after it, while IRNet uses the [Value] token in the NLQ part, without providing the actual value. Additionally, all the columns belonging to a certain table are added right after the table's name in the sequence so as to better preserve the schema structure in this serialised representation. Nevertheless, all relationships between attributes (e.g., primary/foreign keys) are still lost when following this representation.

2.4.3.3 Encoding NLQ with each Column Separately

HydraNet [74] employs a unique approach: it processes the NLQ with each column separately and makes predictions for each column independently. For each table column, a different input is constructed by concatenating the NLQ with the column name and type and the table name. Using this input, the system predicts the probability of the column at hand appearing in the SELECT clause, the probability of the column appearing in the WHERE clause, the operation that will be used if this column appears in the WHERE clause, and so on. It could be argued that this approach does not allow the system to have a complete view of the problem instance, because the neural network makes predictions for each column separately, without being aware of the rest of the table columns. Nevertheless, HydraNet achieves exceptional performance on the WikiSQL benchmark.

This approach does not utilize any additional features (e.g., schema links). However, given that it also serialises its inputs (albeit, only keeping a single column each time), it could draw inspiration from the serialisation techniques described in Section 2.4.3.2 to encode information about schema links. For example, it could append values similarly to BRIDGE [69], or use [Table] and [Column] tokens to explicitly mark column and table names in the input NLQ.

It should be noted, however, that generalizing this approach to a complete relational DB would not be an easy task. First of all, a DB usually has multiple tables, each containing multiple columns, which means that the network would have to make predictions for a much larger number of columns, greatly increasing time complexity for predicting a single SQL query. Furthermore, queries posed on complete DBs often contain JOIN clauses and other operations that depend on more than one entity; as such, processing each column separately becomes very counter-intuitive. Finally, this approach is based on a sketch-based decoder (more in Section 2.4.4), which is hard to extend for complete DBs.

2.4.3.4 Schema Graph Encoding

A graph is the most effective way for representing the DB elements and their relationships. Representing and encoding the input using a graph is used only by a handful of systems [10, 11, 110]. Each node in the graph represents a database table or a column, while their relationships can be represented by edges that connect the respective nodes. It is also possible to add the NLQ words as nodes in the graph, and add edges that connect the query candidates with their equivalent database candidates for representing all the discovered schema links. Additionally, the used graph representation may allow for different classes of nodes and edges leading to even higher expressivity. There can be different classes of nodes to distinguish between tables, columns and NLQ words and different classes of edges to distinguish between edges that represent foreign-primary key relations, edges that indicate a column belonging to a table and edges that represent schema links.

Even though representing the system input as a graph allows for minimal loss of information and can include many types of additional inputs, processing a graph with a neural network is far more difficult than processing a sequence. This is the main reason why graphs have yet to see widespread use in the text-to-SQL problem. However, recent advances in graph neural networks and the clever use of Transformers [108] proposed by RAT-SQL [110] and [98], are showing very promising and might be a good choice for future research.

2.4.4 Output Decoding

Text-to-SQL systems following the encoder-decoder architecture can be divided into three categories based on how their decoder generates the output [15]: (*a*) sequence-based, (*b*) grammar-based, and (*c*) sketch-based slot-filling approaches.

2.4.4.1 Sequence-based approaches

This category includes systems that generate the predicted SQL, or a large part of it, as a sequence of words (comprising SQL tokens and schema elements) [13, 69, 136]. This decoding technique is the simplest, and was adopted by Seq2SQL [136], which is one of the first deep-learning text-to-SQL systems. Later systems steered away from sequence decoding because it is prone to errors.

The main drawback of sequence decoding is that it treats the SQL query as a sequence that needs to be learnt, and at prediction time, there are no measures to safeguard from producing syntactically incorrect queries. When generating a query, it does not take into account the strict SQL grammatical rules, nor does it actively prevent generating incorrect column and table names that do not exist in the DB.

Nevertheless, sequence-based approaches are starting to be used again and are proving to be very efficient thanks to two advances: (*a*) the introduction of large pre-trained seq-to-seq Transformer [108] models (e.g., T5 [92], BART [63]) and (*b*) the use of smarter decoding techniques that constrain the predictions of the decoder and prevent it from producing invalid queries (e.g., PICARD [95]).

2.4.4.2 Sketch-based slot-filling approaches

Systems in this category [42, 46, 74, 76, 120, 127] aim at simplifying the difficult task of generating a SQL query to the easier task of predicting certain parts of the query, such as predicting the table columns that appear in the SELECT clause. In this way, the SQL generation task is transformed into a classification task. In particular, we consider a query sketch with a number of empty slots that must be filled in, and develop neural networks that predict the most probable elements for each slot. A basic prerequisite for such approaches is to have a query sketch that, when completed, will be able to capture the NLQ's intention.

While dividing the text-to-SQL problem into small sub-tasks makes it easier to generate syntactically correct queries, sketch-based approaches may have two drawbacks. Firstly, the resulting neural network architecture may end up being quite complex since dedicated networks may be used for each slot or part of the query. Furthermore, it is hard to extend to complex SQL queries, because generating sketches for any type of SQL query is not trivial.

2.4.4.3 Grammar-based approaches

Systems using a grammar-based decoder [15, 26, 40, 99, 110] are an evolution of sequence-to-sequence approaches, and produce a sequence of grammar rules instead of simple tokens in their output. These grammar rules are instructions that, when applied, can create a structured query.

The most often used grammar-based decoders by text-to-SQL systems have been previously proposed for code generation as an Abstract Syntax Tree (AST) [123, 124]. These models take into account the grammar of the target code language (in our case, the SQL grammar) and consider the target program to be an AST, whose nodes are expanded at every tree level using the grammar rules, until all branches reach a terminal rule. When it reaches a terminal rule, the model might generate a token, for example, a table name, an operator or a condition value, in the case of text-to-SQL. The decoder uses a LSTM-based architecture that predicts a sequence of actions, where each action is the next rule to apply to the program AST. Because the available predictions are based both on the given grammar and the current state of the AST, the possibility of generating a grammatically incorrect query is greatly reduced.

Grammar-based approaches are considered the most advantageous option for generating complex SQL queries, as sequence-based approaches were too prone to errors and sketch-based approaches are difficult to be extended to complex queries. While their status is recently being challenged by the advances of sequence-based decoders discussed earlier, the quest for the most effective decoding technique is far from over.

2.4.5 Neural Training

Another dimension that must be examined when considering a neural text-to-SQL system is the methodology that is followed to train it. Even though the description of a system is usually focused around its architecture and neural layers as well as the way it encodes the inputs and decodes the output, the dimension of neural training is important, because it is the process that enables the neural network to learn how to perform the task at hand.

Earlier systems adopted the simple paradigm of training the network exclusively on a text-to-SQL dataset, however, recent systems have proposed more sophisticated approaches that can greatly benefit the network performance and its generalisation capabilities.

Fresh Start. The most common approach is to train the network from scratch, i.e., initialize all the weights with a random initialization algorithm and train them on a downstream task. However, recent developments in the domain of NLP are showing that pre-trained networks and self-supervised learning are able to achieve much better performance.

Transfer Learning. The use of transfer learning is quickly gaining ground in the NLP community, due to the introduction of Transformers [108], which greatly reduce training time compared to RNNs. Transfer learning refers to when a model trained on a different, usually more generic task, and a different dataset, is incorporated to a new model and further trained on a downstream task (e.g., text-to-SQL). Language models, i.e., networks that have been trained to predict missing words or phrases on huge text corpora, are becoming the standard approach for most NLP tasks, given the performance boost they provide in almost all cases.

Some systems, such as HydraNet [74], rely on language models almost completely, only using linear output layers to produce predictions. Most systems however, incorporate language models as an alternative or an enhancement for word embeddings and RNNs.

Additional Objectives. Another interesting approach that follows the success of language models and self-supervised learning is that of using additional self-supervised tasks while training for the text-to-SQL problem. Recent research [121, 45, 14] suggests that training neural models for more generic tasks besides the downstream task of text-to-SQL that the model is designed to solve, can improve performance on the downstream task. When using additional objectives, one must decide whether the model should be trained on all the auxiliary objectives along the downstream task or whether it should be first trained on the auxiliary tasks and then fine-tuned on the downstream task.

- **Erosion**: The erosion task, proposed by [121], consists of randomly permuting, removing and adding columns to the input schema and training the model to produce the correct SQL query using the eroded schema. Additionally, the system must learn to produce an unknown token when it has to use a column that has been removed from the given schema.
- **Shuffling**: The shuffle task, proposed by [121], randomly changes the order of schema entities and condition values in the input SQL query and NLQ, training the model to correctly re-order them.
- **Graph Pruning**: The graph pruning task, proposed by [14], trains the model to prune all the nodes of the input graph representation that are irrelevant to the given NLQ.
- **Schema Dependency Learning**: SDSQL [45] proposes an additional task to the text-to-SQL task, that closely resonates to the schema linking problem. SDSQL is designed for the WikiSQL dataset. Schema Dependency Learning consists of predicting which words or phrases of the NLQ have a dependency to which columns of the table and the type of the dependency that connects them. The goal is to learn which parts of the NLQ signify that a specific column will appear in the SQL query and the role that the column will have in it (e.g., if it appears in the SELECT clause, if it implies the use of the MAX aggregation function, etc.).

Pre-training Specific Components. Another approach is to train specific parts of our network so that they can better adjust to the peculiarities of the task. For example, GP [135] proposes a framework that pre-trains the system decoder, before training the entire system, in order to better train it on the context-free parts of the SQL grammar, e.g., SQL queries always start with SELECT, the FROM clause is second, and so forth. For this purpose, the encoder's semantic information is replaced by zero vectors so that the decoder is pre-trained without any information about the particular NLQ.

2.4.6 Output Refinement

Once trained, a neural model can be used for inference. There is one last dimension to consider; that of output refinement, i.e., additional techniques that can be applied on a trained model to produce even better results, or to avoid producing incorrect SQL queries.

None. An obvious approach is to use the trained model as is, without output refinement. The most important reason for this approach concerns time and resource availability; in some applications, it might be crucial to achieve low latency responses or to run on everyday machines. For example, PICARD [95], increases inference time by 0.6s when running on a machine with very high-end GPU and arguably even more so on a personal computer. It must be noted however that almost all leader-board entries that achieve high results, use some refinement technique.

Execution-guided Decoding. This is a mechanism [112] that helps prevent text-to-SQL systems from predicting SQL queries that return execution errors. Even though sketch-based approaches are designed to avoid syntactical errors, the possibility for semantical errors is ever-present. Some examples of such errors include aggregation functions mismatches (e.g., using AVERAGE on a string type column), condition type mismatches (e.g., comparing a float type column with a string type value), and so forth. To avoid these type of errors, execution-guided decoding can execute partially complete SQL queries at prediction time and decide to avoid a certain prediction if the execution fails or if it returns an empty output. Execution-guided decoding is system-agnostic and can be applied to most sketch-based systems (e.g., HydraNet, IE-SQL), increasing their accuracy in almost all cases. Let us note that even though some systems presented in this work might not be proposed using execution-guided decoding in their original paper, they are subsequently shown to perform better in the WikiSQL leaderboard when using it. For this reason, they are shown to use execution-guided decoding in Figure 2.5 and Table 2.3.

Constrained Decoding. While generative models with sequence-based outputs are becoming more powerful for NL generation, they are clearly prone to errors when it comes to generating structured language like SQL. PICARD [95] proposes a novel method for incrementally parsing and constraining auto-regressive decoders, to prevent them from producing grammatical or syntactical errors. For each token prediction, PICARD examines the generated sequence so far along with the k most probable next tokens and discards all tokens that would produce a grammatically incorrect SQL query, use an attribute that is not present in the DB at hand, or use a table column without having its table in the query scope (i.e., not having the appropriate table in the FROM clause). Using PICARD, a seq-to-seq pre-trained transformer model (T5-3B [92]) has managed to reach the top of the SPIDER leader-board, lifting the barriers of using sequence-based decoders for text-to-SQL. It should be noted that while PICARD could be considered as the most sophisticated constrained decoding technique, other systems with sequence-based decoders have proposed similar decoding techniques to avoid errors. Some examples of such systems are SeaD [121] and BRIDGE [69].

Discriminative Re-ranking. The Global-GNN
parser [11] proposes an additional network that re-ranks the top-k predictions of the main text-to-SQL network and is trained separately from it. The discriminative re-ranker network takes into account the words of the NLQ and the database elements used by each of the k highest-confidence SQL predictions, by the text-to-SQL network, and re-ranks them based on how relevant it believes they are. Its authors argue that while the text-to-SQL network usually predicts the correct structure for the target SQL query, it might not always predict the correct columns, tables and aggregation functions, because each of them is predicted only knowing already predicted elements and not future predictions. On the other hand, the re-ranker can look at the completed predictions and judge the use of each database element in hindsight, thus improving the prediction quality.

2.5 Neural Architecture

Neural architecture refers to the building blocks used to create all neural parts of the system. This section examines the types of neural layers used by text-to-SQL systems, and analyzes the roles and functions that each one of them is often used for.

Linear Networks. Linear (or Dense) Neural Networks are often used as output layers for sketch-based decoders or to process an internal representation. Given that this type of neural layer is not suited for processing data in a sequence format, they are not effective at processing input such as a NLQ, or producing output in a sequence format (e.g., in a sequence or grammar-based decoder). In sketch-based decoders, however, where the network must predict the correct choice for a certain slot, linear layers are the best suited option to perform this classification task (i.e., choose the best option for filling a slot out of all the available options).

Recurrent Neural Networks. Recurrent Neural Networks (RNNs) have long been considered the go-to solution for NLP, only to be recently dethroned by the powerful Transformers. The main advantage of RNNs is their ability to (*a*) effectively process series inputs, such as a NLQ, which is a series of words, and (*b*) to generate a series output, such as the condition value of a WHERE clause, or a series of grammar rules that can generate a SQL query. Well-known RNN architectures include the LSTM (Long Short-Term Memory) and the GRU (Gated Recurrent Unit). The LSTM is popular for NLP tasks and most often used in text-to-SQL systems.

Early systems, such as Seq2SQL [136] and SQLNet [120], relied on LSTMs for input encoding (along with pre-trained word embeddings), but this type of use is now outperformed by pre-trained Language Models. Even though the recent success of Transformers and Language Models has greatly reduced the use of RNNs in the input encoding phase, RNNs are still being used to assist LMs in input encoding and to generate non-NL series outputs. For example, IRNet [40] uses BERT to encode the input NLQ and schema but also employs LSTMs to create single-token representations for columns and tables with more than one word in their name (and more than one token to represent them).

RNNs are also often used for generating a series output. For example, Seq2SQL [136] and SQLNet [120] employ pointer networks [109] comprised of LSTM layers that generate the entire WHERE clause or the condition value of the WHERE clause, respectively. Another case of RNNs for output generation is seen in systems (e.g., IRNet [40], RAT-SQL [110]) that employ a grammar-based decoder that generates an SQL query as an abstract syntax tree, leveraging work in semantic parsing [123] that uses LSTMs.

Transformers. In text-to-SQL systems, Transformers are commonly used in Transformer-based Pre-trained Language Models for input encoding, to create a contextualized representation of the input text. Pre-trained Language Models offer more robust representations and greatly improve the model performance almost all of the times, making them more preferable than pre-trained word embeddings. To use them for input encoding, one can simply replace the input encoder (e.g., word embeddings and LSTM) with a model like BERT.

There have been also other, rarer uses of Transformers in text-to-SQL systems. For example, HydraNet is a system completely reliant on a pre-trained language model. In this case, the text-to-SQL problem is formulated so that it matches the pre-training logic of a language model and only very simple linear networks are used to make predictions using the contextualized representations created by the Language Model.

Another unique example is RAT-SQL [110], which uses specifically modified Relation Aware Transformers (RAT) to encode its input. What is special about RAT is that they also accept pre-defined relations about the elements of input series, which essentially allows to bias the encoder towards already known relations in the database schema and

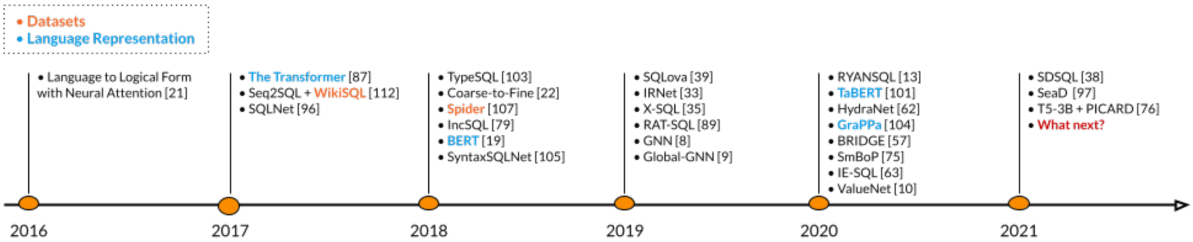


Figure 2.7: A timeline of deep learning text-to-sql systems, datasets and language representation techniques

the user question. A similar approach is used by [98] in order to extend the Transformer architecture to support relations between elements of the inputs, in the form of a GNN Sub-layer. This extension of the Transformer allows to encode the input as a graph, where the edges can have different layers, similarly to RAT-SQL; however, its performance is much lower on the Spider benchmark.

Conditional Random Fields (CRFs). CRFs [61] are a type of discriminative machine learning model that excels at modelling relations and dependencies. Because of this capability, CRFs are often used in NLP for labelling tasks such as Part-of-Speech (POS) tagging and Named Entity Recognition (NER). Even though CRFs are rarely used in text-to-SQL, there is a notable mention of a system integrating them in its neural architecture for a specific sub-task. Namely, IE-SQL [76] employs CRFs tasked with two schema-linking tasks of recognising: (*a*) which words in the NLQ are *slot mentions* to SQL elements, such as the SELECT column and the WHERE columns, and (*b*) finding *slot relations*, i.e., grouping each of the WHERE column mentions with the mentions of operations and values that correspond to them. Both tasks are modelled as labeling tasks, which is why CRFs are a good choice.

Convolutional Neural Networks (CNNs). Convolutional networks are very rarely used for the text-to-SQL task, since they are best suited for processing visual data. One example of a system using CNNs is RYANSQL [15], which uses CNNs with Dense Connections [126], in order to encode the inputs. However, the authors of RYANSQL demonstrate that replacing this CNN-based encoder with a PLM can greatly improve the model's performance, making the choice to steer away from CNNs all the more obvious.

2.6 Systems

Having established a taxonomy for deep learning text-to-SQL systems, let us now zoom in on key systems that have introduced novel and interesting ideas and have shaped the area. This section provides insights and explanations on these systems while also grouping them based on important milestones of this research area. Figure 2.7 presents a chronological view on deep learning text-to-SQL systems, along with important datasets and language representation advancements that have had a great impact on the domain. While certain systems could obviously fit in multiple sections, this specific categorization is based on the novelty introduced by each system at the time of its publishing, its influence on later systems, as well as the possible importance of each novelty given its capability to address future and open research problems.

```
SELECT $AGG $SEL_COL
(WHERE $COND_COL $COND_OP $COND_VAL
(AND $COND_COL $COND_OP $COND_VAL)* )?
```

Figure 2.8: Query Sketch proposed by SQLNet

2.6.1 The Dawn of an Era

As mentioned before, the era of deep learning text-to-SQL systems essentially starts with the release of the first large annotated text-to-SQL dataset. WikiSQL was released along with *Seq2SQL* [136], which was one of the first neural networks for the text-to-SQL task and was based on previous work focusing on generating logical forms using neural networks [25]. The system predicts the aggregation function and the column for the SELECT clause as classification tasks and generates the WHERE clause using a seq-to-seq pointer network. The latter part of the system is burdened with generating parts of the query that can lead to syntactic errors, which is its major drawback.

A big difference from almost all other systems is that Seq2SQL is partly trained using reinforcement learning. While the aggregation function and SELECT column predictors are trained using cross entropy loss, the WHERE clause predictor is trained using a reward function that returns a positive reward if the produced query returns the same results as the ground truth query and a negative reward if the query returns different results or if it cannot be executed due to errors. The reasoning behind using reinforcement learning, even though it generally performs worse than supervised learning, is that the WHERE clause can be expressed in multiple ways and still be correct.

To address these problems, i.e., that sequence decoders can produce errors and that reinforcement learning is not ideal, *SQLNet* [120] proposed using a query sketch with fixed slots that, when filled, form a SQL query. This sketch can be seen in Figure 2.8, and it covers all the queries present in the WikiSQL dataset. Using a sketch allowed the problem to be formulated almost entirely as a classification problem, since the network has to predict: (*a*) the aggregation function between a fixed number of choices, (*b*) the SELECT column among a number of columns present in the table, (*c*) the number of conditions (between 0 and 4 in the WikiSQL dataset), (*d*) the columns present in the WHERE clause (as multi-label classification, since they can be more than one), (*e*) the operation of each condition among a fixed number of operations ($\leq, =, \geq$) and (*f*) the value of each condition. Predicting the value is achieved using a sequence generator network, which in this case is only responsible for the value and not for the SQL syntax or grammar, so syntactic mistakes are avoided.

Another improvement introduced in SQLNet is the introduction of a *column attention neural architecture* to the network. Given that SQLNet encodes the NLQ and table columns separately, the encoded representation of the NLQ does not have any information on the available columns and thus cannot inform the system on which words in the NLQ are important for generating the correct SQL query. Column attention is an attention mechanism that infuses the NLQ representation with information about the table columns, so as to emphasize the words that might be more related to the table. Other than that, both systems are similar to each other, using GloVe [83] embeddings for text representation and LSTM networks for encoding them.

2.6.2 Sketch Generation

While the use of a sketch greatly simplifies the text-to-SQL problem and makes predictions simpler for neural networks, the complexity of SQL queries the system can generate using a single sketch is restricted. Systems such as Coarse2Fine [26] and RYANSQL [15] have tried to generalise sketch-based decoding, by attempting to not only fill in the slots of a sketch but also generate the appropriate sketch for a given NLQ.

Coarse2Fine [26] is a semantic parser that can generate various types of programs, one of which is SQL. Its main highlight is that it decomposes the decoding process into two steps: first, it generates a rough (coarse) sketch of the target program without low-level details, and then it fills this sketch with the missing (fine) details. Its authors argue that a great advantage of this approach is that the network can disentangle high-level from low-level knowledge and learn each one of them more effectively. Unfortunately, this system is only used on the WikiSQL dataset and is not extended to more complex SQL queries, which is not trivial work. In fact, because Coarse2Fine is designed for the WikiSQL dataset, the sketches it generates only differ between them in the number of conditions that appear in the WHERE clause and the operations in each condition. As such, while the idea it proposes might be very interesting, in practice, it essentially achieves generating SQL queries of no greater complexity than what simple sketch-based systems do.

RYANSQL [15] is another system that generates the appropriate sketch before filling it, but in contrast to the previous, it manages to produce much more complex SQL queries such as the ones present in the Spider dataset. This is achieved by breaking down each SQL query into a non-nested form that consists of multiple, simpler, sub-queries. The authors propose 7 types of sub-queries, each with its own sketch, that can be combined to produce more complex queries. The network then learns to recursively predict the type of each sub-query and to subsequently fill in its sketch. RYANSQL achieved the first position in the Spider benchmark at the time of its publication, but has since been surpassed by other systems, while no other similar approach has been able to achieve comparable performance.

SyntaxSQLNet [129] follows a similar approach, but instead of generating the query sketch, it follows a pre-defined SQL grammar that determines which of its 9 slot-filling modules needs to be called to make a prediction. This allow the system to produce grammatically correct complex queries while enjoying the benefits of a sketch-based decoder. At each prediction step, the grammar and the prediction history from the previous steps are used to determine the module (e.g., COLUMN module, AGGREGATOR module, OPERATOR module, HAVING module, etc.) that needs to make a prediction in order to build the SQL query. Although this is a hybrid approach, the architecture of the decoder modules classifies SyntaxSQLNet as a sketch-based decoding system. The main difference is that most sketch-based decoders call all their slot-filling modules simultaneously to fill the sketch, whereas SyntaxSQLNet calls specific modules recursively because the grammar defines what needs to be filled in at each prediction step. SyntaxSQLNet was one of the first systems proposed for Spider. Since then, many systems have achieved better performance scores while steering away from this methodology, hinting at its weaknesses. For example, one of the main challenges is to effectively pass all the information of the prediction history and the current state of the generated SQL to each module, at every prediction step.

2.6.3 Graph Representations

The use of graphs for input encoding has only recently seen increased use, despite its powerful capability to represent the DB schema. This section explores key systems that have shown new perspectives on how graphs can be represented and used in the text-to-SQL task.

A natural option for processing graphs are Graph Neural Networks (GNNs). However, while being a good option for tasks such as node classification, node clustering and edge prediction, they are not as suitable for generative tasks like the text-to-SQL problem. Two systems manage to leverage GNNs to encode the database schema and its elements: the GNN parser [10] and its successor Global-GNN parser [11]. To achieve this, the database schema is represented as a graph, where tables and columns are represented as nodes, and different types of edges represent the relationships between them (e.g., which columns appear in which table and which columns and tables are connected with a primary-foreign key relationship). For NLQ encoding, both systems use word embeddings and LSTM networks, while node encodings calculated by the GNNs are concatenated to each word embedding, based on the discovered schema links. For decoding, both systems use a grammar-based decoder [123] that generates a SQL query as an Abstract Syntax Tree (AST), which is often used by grammar-based systems [12, 40, 110]. Global-GNN [11] introduces the use of a re-ranker that, given k SQL predictions from the network, chooses the best interpretation based on the database elements used and the graph representation calculated.

In order to avoid the disadvantages of GNNs, other efforts modify architectures that have already shown their power in the text-to-SQL task, such as the Transformer [108], so that they can accept edge information and process a graph. *RAT-SQL* [110] uses a graph representation of the input, but instead of using GNNs, it proposes a modified Transformer architecture named Relation Aware Transformer (RAT). Firstly, it creates a *question-contextualized schema graph*; i.e., a graph representing the database tables and columns as well as the words of the NLQ as nodes and the relationships between them as edges. An edge can appear either between two database nodes, similarly to the previous systems, or between a database node and a word node. In this graph, schema linking is performed to discover connections between a database node and a word node that might refer to it. The names of all the nodes in the graph are first encoded using BERT [23] and then processed by the RAT network, along with the edge information of each node. The RAT neural block performs *relation aware self-attention* on its inputs, which essentially biases the network towards the given relations (edges). This allows the system to use Transformers and even pre-trained language models to process the graph as a series while also utilising the information present in the graph edges. Finally, it generates a SQL query as an AST using the method mentioned above [123].

All systems discussed in this section have grammar-based decoders. This happens mainly because they aim to produce complex queries such as the ones in the Spider dataset, and at the time of their publication, grammar-based decoders were the most common option. It would be possible for a system using a graph representation of the input to use a different decoder with its own advantages and drawbacks.

2.6.4 Using Intermediate Languages

Following the success of the grammar-based methods in generating complex SQL queries over multi-table DBs, researchers also examined the use of languages during the decoding phase that can better align with NL than SQL making it easier for the system to make predictions, but at the same time they can be deterministically translated into SQL. We examine key systems that use an Intermediate Language, either a pre-existing language or one created specifically for this task, as the target language for the neural decoder.

IRNet [40] is a grammar-based system capable of generating complex SQL queries, such as the ones in the Spider dataset. It uses the same AST decoding method [123] for code generation used in other grammar-based text-to-SQL systems (e.g., RATSQL [110] and the GNN parser [10]). The main difference is that it predicts an AST of a SemQL program, which is an Intermediate Language created specifically for this system. Its authors argue that it is easier to generate queries in this language and then transform them to SQL. Furthermore, IRNet performs schema linking by considering all n-grams of length 1 to 6 as query candidates and all column and table names as DB candidates and uses exact and partial matches to discover links between them. It also searches for all query candidates that appear inside quotes in the ConceptNet knowledge graph [103] in order to link them to a database column or table. Input encoding uses BERT followed by linear and recurrent neural networks.

SmBoP [94] is a grammar-based system that introduces various novelties in the decoding phase. The use of relational algebra as an Intermediate Language is one of them. Its authors argue that, along with being better aligned with NL, relational algebra is a language that is already used by DB engines, unlike SemQL. Additionally, in order to decode ASTs of queries in relational algebra, SmBoP uses a bottom-up parser, in contrast to the usual approach of generating ASTs by performing top-down depth-first traversal, followed by almost all text-to-SQL systems. The bottom-up decoder generates at time step t, the top-k sub-trees of height $\leq t$, where k is a given parameter that represents the number of beams used during the decoding search. The main advantage of the bottom-up parsing is that at any given time-step, the generated sub-trees are meaningful and executable sub-programs, while in the top-down parsing, intermediate states are partial programs without a clear meaning.

2.6.5 The Age of BERT

Much like in other NLP problems, replacing a conventional encoder with a pre-trained language model such as BERT [23] has been shown to improve performance of a text-to-SQL system.

SQLova [46] is a sketch-based approach focused on the WikiSQL dataset. It employs a large and complex network almost identical to the one used by SQLNet, with its main difference being that instead of GloVe embeddings, it uses BERT to create a contextualized representation of the NLQ and table headers. The representations are then passed to 6 networks, each responsible for a different part of the query sketch, that are very similar to the sub-networks used by SQLNet. The result is a staggering, almost 20%, increase in execution accuracy on the test set of WikiSQL, indicating BERT's power in the text-to-SQL task.

HydraNet [74] is another sketch-based approach on the WikiSQL benchmark taking advantage of the BERT language model. Its main difference from SQLova is that HydraNet aligns itself better to the way that BERT has been pre-trained and only uses a simple linear network after receiving the contextualized representations from BERT, instead of large networks with LSTMs and attention modules like SQLova. Furthermore, HydraNet processes each table header separately instead of jointly encoding them, an approach that is unique to this system. As a result, it can only make predictions for each column on its own, i.e., it decides if the column at hand will appear in the SELECT clause, if it will appear in the WHERE clause, what its operation will be if it appears in the WHERE clause and so on. HydraNet, with its simpler architecture leveraging BERT, achieves better accuracy on WikiSQL than SQLova, which employs a larger and more complex network.

X-SQL [42] is a sketch-based system using the MT-DNN pre-trained language model [70], that was built for the WikiSQL benchmark. Similarly to HydraNet, it uses much simpler networks than SQLova for filling the slots of the query sketch. However, it encodes all table headers simultaneously, along with the user question. Additionally, instead of using segment embeddings that originally indicate the span of different sentences in the language model's input, X-SQL uses *type embeddings*. These embeddings differentiate between the different types of elements in the input, such as the user's question, categorical columns and numerical columns. Furthermore, it uses an attention layer to create a single token representation for columns that have more than one token (i.e., more than one word in their name). X-SQL also outperforms the much more complex SQLova, achieving slightly lower scores than HydraNet.

2.6.6 Schema Linking Focus

As discussed earlier, schema linking is a major part of creating a SQL query from a NLQ. This section looks into systems that have put extra effort on schema linking, or even based their entire workflow on this process.

TypeSQL [127] is one of the first systems to introduce a process similar to schema linking in its workflow, and one of the few systems working on WikiSQL that uses schema linking. Its methodology is described as *Type Recognition*, but closely resonates to the concept of schema linking. The goal of this methodology is to assign a "type" to every token of the NLQ. It considers all n-grams in the NLQ of length from 2 to 6 and tries to assign them one of the following "types": (*a*) *Column*, if it matches the name of a column or a value that appears under a column, (*b*) *Integer, Float, Date or Year*, if it a numerical n-gram, (*c*) *Person, Place, Country, Organization or Sport* by performing NER using the Freebase knowledge graph. Even though this process is unilateral, as its main goal is to classify the query candidates into a type category and not to explicitly link them to a DB candidate, it is one of the first attempts towards schema linking.

ValueNet [12] builds on the grammar-based system IRNet [40] focusing on schema linking and condition value discovery. The main motivation of the system is that despite the constant improvement of text-to-SQL systems, even the state-of-the-art is falling behind at predicting the correct values in the SQL conditions. Similarly to IRNet, ValueNet decodes a SQL query in a SemQL 2.0 AST. SemQL 2.0 extends the SemQL grammar with values. Additionally, since condition values might not be written by the user in the exact same way they appear in the DB, ValueNet employs an extended value discovery workflow of five steps:

- *value extraction*: to recognize possible value mentions in the NLQ, it uses NER and heuristics;

- *value candidate generation*: to create additional candidate values, it uses string similarity, hand-crafted heuristics and n-grams;
- *value candidate validation*: to reduce the number of candidate values, it keeps only the candidates that appear in the DB;
- *value candidate encoding*: it appends each candidate to the input along with the table and the column it was found under, and
- *neural processing*: the encoded representations are processed by the neural network, which eventually decides if and where they will be used.

The authors also provide a classification of the Spider queries based on the difficulty of discovering the values. This is another important aspect of the text-to-SQL problem usually overlooked by other works.

SDSQL [45] is a sketch-based system designed for the WikiSQL task. What is special about this system is that it can be viewed as two neural networks tackling two tasks at the same time. The first network predicts SQL queries using the same architecture used by SQLova [46], while the second network performs *schema dependency* predictions. The schema dependency network uses bi-affine networks [28] to predict dependencies between the words of the NLQ and the table headers. Such dependencies include: (*a*) the *select-column* dependency that connects a query candidate that maps to a column that will appear in the SELECT clause with the corresponding column of the table, and (*b*) the *where-value* dependency that connects the query candidate that refers to a value that will appear in the WHERE clause to the table column it belongs to. It must be noted that even though the second network performs schema linking, its predictions are not directly used by the first network to construct the SQL query. Instead, a combined loss from the predictions of both tasks is used to train the weights of the networks, which allows the *schema dependency learning* to improve the first network's performance indirectly.

IE-SQL [76] proposes a unique approach to the text-to-SQL problem almost completely based on schema linking. It uses two instances of BERT [23] to perform two different tasks: a *mention extractor* and a *linker*. The mention extractor recognizes which query candidates are mentions of columns that will be used in the SELECT and WHERE clauses of the SQL query, mentions of aggregation functions, condition operators and condition values. Additionally, the mention extractor recognizes mentions that should be grouped together. For example, the mentions of the column, the operator and the value that belong to the same condition are grouped together. Having extracted the mentions, the linker maps the mentions of column names to the actual columns of the table they are referring to. The linker also maps value mentions without a grouped column to the appropriate table column. By using the predictions of the mention extractor and the linker, IE-SQL can predict an SQL query, without any additional neural component. Even though this approach may not be a clear match with any of the three decoding categories, we classify it as a sketch-based system because its methodology is heavily based on the existence of a query sketch similar to the one used by SQLNet [120]. IE-SQL can better learn the dependencies between the slots and uses a more robust approach. Still, the mention types it recognizes are a direct match to the slots of the query sketch. Therefore, extending it to queries beyond the sketch is not trivial.

2.6.7 The Return of the Sequence

Generating SQL queries using a sequence-based decoder was initially avoided as it could produce syntax and grammar errors, as discussed in Section 2.4.4. Grammar-based decoders were instead regarded as the best choice for a system to effectively generate complex SQL queries. However, recent works [69, 95, 121] have changed the landscape by introducing a series of techniques that minimize the possibility of errors by sequence-based decoders. These techniques have made the use of very powerful pre-trained encoder-decoder models [63, 92] a viable and high-performing option, allowing the systems that use them to achieve top performance in both the Spider and WikiSQL benchmarks.

SeaD [121] is a system based on the BART [63] encoder-decoder pre-trained language model designed on WikiSQL. To overcome the drawbacks of its sequence-based decoder, SeaD employs two techniques: (*a*) it introduces two additional tasks on which the model is trained at the same time with the text-to-SQL task, and (*b*) it uses execution-guided decoding [112], slightly modified to work with its sequence-based decoder. Its main contribution is the use of the two additional training tasks named *erosion* and *shuffle* (see Section 2.4.5), which are designed specifically to help the model better understand the nature of the text-to-SQL problem and the tables used by the WikiSQL dataset. The use of additional training tasks is also closely aligned with how language models are pre-trained to understand the more general notion of natural language before being fine-tuned to a specific task. Nevertheless, while SeaD has managed to overcome the limitations of sequence-based decoders and achieve the best performance on the WikiSQL benchmark, both the decoding technique and the additional objectives it employs are designed with the WikiSQL dataset in mind. Extending them to full relational databases would not be a trivial matter.

BRIDGE [69] is another recent system with a sequence-based decoder that works on Spider, although it does not use an encoder-decoder language model. Instead, it uses BERT [23] and LSTM networks for input encoding and enriches the input representation using linear networks that use metadata such as foreign and primary key relationships, as well as column type information. Additionally, the system performs schema linking using fuzzy string matching between query candidates and the values of columns that only take values from a pre-defined list (i.e., picklist attributes). The discovered values are added in the input sequence to help the network create better SQL queries. Finally, the sequence-based decoder used by BRIDGE is a pointer generator network using *Schema-consistency Guided Decoding*, a constraining strategy to avoid the aforementioned drawbacks of sequence-based decoders. In order to use schema-consistency guided decoding, BRIDGE is trained (and makes predictions) on SQL queries written in *execution order*, i.e. all queries start with the FROM clause, followed by the WHERE, GROUP BY, HAVING, SELECT, ORDER BY and LIMIT clauses, strictly in that order. This means that all columns that appear in the query, must appear after the table that they belong to has been generated. Based on this, BRIDGE can limit the search space of columns and avoid using columns that will produce invalid SQL queries.

PICARD [95] is a constraining technique for auto-regressive decoders of language models, that is specifically created to improve their performance on the text-to-SQL task. Essentially, at each prediction step, it constrains the model's set of possible predictions by removing tokens that could produce syntactically and grammatically incorrect SQL queries.

It is used at inference time, by looking at the confidence scores of the model's prediction and the schema of the underlying DB, and it operates at three levels:

- it rejects misspelled attributes and keywords, as well as tables and columns that are invalid for the given schema,
- it parses the output as an AST to reject grammatical errors, such as an incorrect order of keywords and clauses or an incorrect query structure,
- it checks that all used tables have been brought into scope by being included in the FROM clause and that all used columns belong to exactly one table that has been brought into scope.

When PICARD is used with the T5 [92] pre-trained language model (the 3B parameters version), it ranked first on the Spider leaderboard for execution with values. This of course does not come without any drawbacks, such as the increased prediction time due to the constrained decoding, as well as the tremendous computational and memory requirements for training and running such a large model as T5-3B.

Table 2.3: Systems examined in this work. In the Natural Language column, WE, E-PLM and ED-PLM stand for Word Embeddings, Encoder-only PLM and Encoder-Decoder PLM accordingly. In the Neural Training column, FS, TL and AO stand for Fresh Start, Transfer Learning and Additional Objective accordingly. In the Output Refinement column, EG Decoding and Constr. Decoding stand for Execution-Guided and Constrained Decoding accordingly.

Year	System	Benchmark	Schema Linking	Natural Language	Input Encoding	Output Decoding	Neural Training	Output Refinement
2017	Seq2SQL	WikiSQL	✗	WE	Separate	Sequence	FS	✗
	SQLNet	WikiSQL	✗	WE	Separate	Sketch	FS	✗
2018	IncSQL	WikiSQL	✗	WE	Separate	Grammar	FS	✗
	TypeSQL	WikiSQL	✓	WE	Separate	Sketch	FS	✗
	Coarse2Fine	WikiSQL	✗	WE	Separate	Sketch	FS	✗
	SyntaxSQLNet	Spider	✗	WE	Separate	Sketch	FS	✗
2019	SQLova	WikiSQL	✗	E-PLM	Serialise	Sketch	TL	EG Decoding
	IRNet	Spider	✓	WE or E-PLM	Serialise	Grammar	TL	✗
	X-SQL	WikiSQL	✗	E-PLM	Serialise	Sketch	TL	EG Decoding
	RAT-SQL	Spider	✓	WE or E-PLM	Graph	Grammar	TL	✗
	GNN	Spider	✓	WE	Graph	Grammar	FS	✗
	Global-GNN	Spider	✓	WE	Graph	Grammar	FS	Re-ranking
2020	ValueNet	Spider	✓	E-PLM	Serialise	Grammar	TL	✗
	BRIDGE	Spider	✓	E-PLM	Serialise	Sequence	TL	Constr. Decoding
	HydraNet	WikiSQL	✗	E-PLM	Per column	Sketch	TL	EG Decoding
	IE-SQL	WikiSQL	✓	E-PLM	Serialise	Sketch	TL	EG Decoding
	RYANSQL	Spider	✗	WE or E-PLM	Serialise	Sketch	TL	✗
	SmBoP	Spider	✓	E-PLM	Graph	Grammar	TL	✗
2021	SDSQL	WikiSQL	✓	E-PLM	Serialise	Sketch	TL + AO	EG Decoding
	SeaD	WikiSQL	✗	ED-PLM	Serialise	Sequence	TL + AO	Constr. Decoding
	T5-3B+PICARD	Spider	✗	ED-PLM	Serialise	Sequence	TL	Constr. Decoding

2.7 Discussion and and Higher-level Comparison

In what follows, we make several observations regarding how the landscape is shaped along the dimensions of our taxonomy, presented in Section 2.4. Table 2.3 provides an overview of the design choices of each system studied in this survey. Additionally, we provide some higher-level insights that can be useful for practitioners interested in introducing a deep learning text-to-SQL system in a real-world use case. These insights include remarks concerning: adaptability to new databases, difficulty of implementation,

Table 2.4: Higher-level comparison of taxonomy dimensions on various practical dimensions (↗ signifies good performance, ↘ signifies poor performance, and → signifies average performance).

	Natural Language			Input Encoding				Output Decoding			Neural Training			Output Refinement		
	WE	E-PLM	ED-PLM	Separate	Serialise	Per Column	Graph	Sketch	Grammar	Sequence	Fresh Start	Tr. Learning	Add. Objectives	EG-Decoding	Constr. Decoding	Re-ranking
Ease of Implementation	→	↗	↗	→	↗	→	↘	→	↘	↗	↗	→	→	→	↘	→
Use with Full RDBs	▨	▨	▨	→	↗	↘	↗	↘	↗	→	▨	▨	▨	▨	▨	▨
Extend to New SQL Types	▨	▨	▨	▨	▨	▨	▨	→	↘	↗	↗	→	→	→	↘	→
Computational Costs	→	↘	↘	↘	→	↘	↗	↗	→	→	↗	↘	→	→	↘	→
Handling Large Schemas	→	↘	↘	→	↘	↘	↗	▨	▨	▨	▨	▨	▨	▨	▨	▨

technical demands, and other advantages and drawbacks of certain design choices. A summary of these insights can be seen in Table 2.4.

Output Decoding. There is a connection between the decoding approach used by a system and the benchmark on which it operates. Systems that operate on Spider do not use a sketch-based decoder. This is due to the fact that sketch-based approaches are more cumbersome to be adapted for generating complex SQL queries. RYANSQL [15] attempted extending the sketch-based approach to Spider, but later systems steered away from this choice. Furthermore, while until recently grammar-based decoders dominated the Spider benchmark and sketch-based decoders dominated WikiSQL, recent improvements in sequenced-based decoders have turned the tables, bringing sequence-based decoders on the top of both benchmarks (i.e., T5-3B+PICARD [95] for Spider and SeaD [121] for WikiSQL).

The output decoder is what defines the system's SQL expressiveness and the effort needed to implement and extend the system to new types of SQL queries. For example, grammar-based decoders are harder to implement, since an extensive grammar is required in order for the system to cover all the possible SQL queries that the use case in question might require. Additionally, extending a system to use mathematical operations (e.g., `WHERE end_year - start_year < 4`) will require varying degrees of effort depending on the type of decoder. In the case of a sketch-based or grammar-based decoder, an extension of the sketch or grammar is necessary to cover the new query type. On the other hand, sequence-based decoders can effectively generate everything (which is usually a drawback), as long as there are training examples to learn from.

NL representation. There is a clear tendency by the latest models to use PLMs for NL representation. Besides the systems that use GNNs for input encoding [10, 11], the only systems that use word embeddings for NL representation, were published before PLMs were widely available. In almost all cases, the use of a PLM instead of word embeddings leads to a boost in performance. This is also shown in some systems that were originally designed to work with word embeddings, but are also tested with a PLM during ablation studies (e.g., RAT-SQL [110], RYANSQL [15]). In fact, with the constant introduction of new PLMs, the question of which PLMs is more suitable becomes all the more relevant. However, a major, typically overlooked, drawback of PLMs is their computational cost and hardware requirements. Even though the cost of pre-training can be alleviated because it is very easy to find a pre-trained model online, there is still the cost of training for the text-to-SQL downstream task, as well as during inference. Running a model with a PLM will also require an additional amount of computational resources (usually memory and/or a GPU) due to the size of these models. For example, BERT-base [23] has 110M parameters, BERT-large has 340M parameters, and T5 [92] has variations of similar sizes that reach up to 11B parameters, while the one presented with PICARD [95] has 3B parameters. This must be considered, especially when building applications that must support heavy workloads or have low latency requirements.

Input Encoding. Regarding input encoding, there are two main observations to point out: (*a*) while earlier systems performed separate encoding, later systems use serialised or graph encoding, and (*b*) newer systems working on the WikiSQL, all use serialised encoding. The clear tendency to use serialised encoding can be easily attributed to the extensive use of PLMs, which offer much better performance with a serialised input. This is even more true in the case of WikiSQL, because single tables can easily be serialised along with the NLQ making the combination of PLMs and serialised encoding an easy and powerful choice. However, when it comes to DBs with several tables and relationships among them and their columns, a more flexible and informative representation is required. Some systems have examined the more innovative approach of graph encoding, which so far seems promising, offering a lot of ground for future research. Another practical limitation that must be taken into account is how flexible each encoding option is when it comes to DBs with large schemas. For example, the SDSS database, which stores data from astronomical surveys, has 87 tables with some tables containing up to a hundred columns. Serialising such a schema would result in a very long sequence that can not be processed by a PLM due to their limitation in input length. Similarly, performing separate encoding might create a bottleneck in the schema encoder side. Graph encoding might be more efficient for handling larger schemas, since GNNs can encode each schema element as a single node. However, this approach is also prone to poorer performance as the schema gets larger.

Schema Linking. Table 2.5 displays the schema linking techniques used by each system studied in this survey. While the first text-to-SQL systems did not perform any kind of schema linking, later systems have proposed various intricate schema linking pipelines. On the query side, we observe that almost all systems consider single-word and multi-word tokens, while ValueNet [12] also performs NER to find possible candidates. On the DB side, using the table and column names is the baseline for most systems, while some systems also lookup the values that are present in the DB. Finally, to match the candidates, some systems use simple text matching (either exact or partial), while newer systems have experimented with the use of classifiers instead of string operations to find matches. It becomes quickly apparent that schema linking is mostly explored by systems operating on the Spider dataset, accompanied by very few systems using the WikiSQL benchmark. This is somewhat expected, given that as the SQL complexity and the volume of tables, columns and data increase, researchers seek to aid the neural network by providing auxiliary information. However, what is very peculiar is that some high performing recent systems (i.e., T5-3B+PICARD [95] and SeaD [121]) do not perform any schema linking at all. This is an open research question. Can powerful neural architectures, pre-trained on vast amounts of data, defy the need for schema linking? Or, can they achieve even higher scores if combined with schema linking? One important observation is that very little effort has been put into testing how fast and scalable these approaches are, especially for very large databases. In fact, to the best of our knowledge, only a single work [107] provides experimental evaluations concerning the time and memory used for schema linking. Hence, extra caution is necessary when using these methods in a real-world system, as most of them are not adequately optimised.

Neural Training. The neural training dimension is closely connected to the NL representation adopted by each system. This happens because using a PLM means that the model adopts the Transfer Learning paradigm, because it further trains an already pre-trained neural component on a new downstream task. There are no cases of systems performing transfer learning on other parts of the model besides the NL representation part. This is mostly due to the fact that PLMs perform exceptionally well, and making an improvement through a different transfer learning technique would be very difficult. Furthermore, there are only two models that use additional objectives during training [69, 121]. This relatively novel approach follows the success of PLMs using various auxiliary tasks during pre-training and seems to be very promising in training a model that achieves better generalization. It must be noted that the time and computing resources needed to train a model using each training approach are usually not taken into account when presenting new models, in favour of better performance metrics. It is however necessary to address them in order to make the use of such models feasible in a real-world application. For example, the pre-training part of transfer learning is very costly, unless the pre-trained model is made available by its creators. Similarly, using additional objectives will greatly increase the computations that must be performed, thus increasing the cost of training.

Output Refinement. The output refinement heavily depends on the approach used for output decoding, as well as the dataset that the system operates on. A system designed for WikiSQL can use execution-guided decoding [112], no matter the type of its decoder because of the simplicity of the WikiSQL queries. Systems with sequence-based decoders can use constrained decoding techniques to improve their predictions and reduce the possibilities of errors. In fact, this output refinement technique is one of the main reasons why they can be so effective. The re-ranking technique could be used by any system that can produce more than one predictions for a single input, but in practice it has not been adopted by any other system after being proposed by Global-GNN [11]. Furthermore, each refinement technique adds an additional burden to the system that translates to extra computational cost and more time needed to make a prediction. When used in a real-time application, it is necessary to consider if the performance boost gained from the refinement step, is worth the extra time and resources required.

2.8 Research Challenges

While a lot of progress has been made on the text-to-SQL problem, several important issues need to be tackled. In this section, we outline some of the most challenging problems and highlight interesting research opportunities for the database and the machine learning communities that could greatly impact the state of the art in text-to-SQL research and beyond.

2.8.1 Benchmarks

As mentioned earlier, WikiSQL and Spider are large-scale query benchmarks that provide a common way to evaluate and compare different systems. They have simplified system evaluation, and they are often seen as the panacea for text-to-SQL evaluation. Researchers tend to over-rely on these benchmarks to argue that their systems are advancing the state of the art, and they do not spend time performing additional experiments on other benchmarks. However, given the progress in system-building, new standards are necessary for benchmarking text-to-SQL systems, in order to make these systems applicable to real-world scenarios, and to continue pushing the state of the art.

Table 2.5: A comparison of schema linking techniques used by the examined systems; the schema linking process is divided in the query candidate discovery, DB candidate discovery and candidate matching phases, as described in our taxonomy

Year	System	Benchmark	Query				DB			Matching					
			Single Tokens	Multi-word Tokens	Named Entities	Additional Candidates	Table and Column Names	Values via Lookup	Values via KGs	Exact Matching	Partial Matching	Approximate Matching	Learned Embeddings	Classifiers
2017	Seq2SQL	WikiSQL												
	SQLNet	WikiSQL												
2018	IncSQL	WikiSQL												
	TypeSQL	WikiSQL	✓	✓	✓		✓	✓		✓				
	Coarse2Fine	WikiSQL												
	SyntaxSQLNet	Spider												
2019	SQLova	WikiSQL												
	IRNet	Spider	✓	✓			✓		✓	✓	✓			
	X-SQL	WikiSQL												
	RAT-SQL	Spider	✓	✓			✓	✓		✓	✓			
	GNN	Spider	✓				✓						✓	
	Global-GNN	Spider	✓				✓						✓	
2020	ValueNet	Spider	✓	✓	✓	✓	✓	✓		✓	✓			
	BRIDGE	Spider	✓					✓				✓		
	HydraNet	WikiSQL												
	IE-SQL	WikiSQL	✓	✓			✓							✓
	RYANSQL	Spider												
	SmBoP	Spider	✓	✓			✓	✓		✓	✓			
2021	DBTagger	-	✓	✓			✓							✓
	SDSQL	WikiSQL	✓	✓			✓							✓
	SeaD	WikiSQL												
	T5-3B+PICARD	Spider												

First of all, datasets such as WikiSQL, that contain single-table databases and very simple SQL queries, can not be seen as realistic benchmarks for real-world applications. Given that the SQL queries in WikiSQL can be covered by a very simple sketch, as the one shown in Figure 2.8, and current systems have reached very high accuracy scores on this dataset, there is a need for more challenging benchmarks. These benchmarks were a good start for the neural text-to-SQL field and have allowed a lot of novel ideas to be implemented in a “sandbox” environment, but the state of the art is now able to achieve much more.

Similarly, Spider contains DBs and queries that were specifically created for text-to-SQL evaluation but they are rather simplistic and do not reflect the characteristics of real-world DBs. For example, the Spider DBs have a simple schema or too little data stored. In fact, the 166 DBs of Spider that are available to the public (i.e. train and dev set, since the test set is held-out by the authors) sum up to less than 1GB. Ideally, new benchmarks should aspire to introduce real cases of DBs taken from the industry and academia, accompanied with real logs of SQL queries performed on them by their users. The NLQ part could be obtained either by asking the users to specify what their intention was when running these queries, by asking SQL experts to explain them, or by employing a SQL-to-text system.

Another important drawback of current benchmarks is their relatively small number of examples (i.e., NL-SQL pairs), especially compared to datasets used by deep neural networks in other problems (e.g., the SQuAD Question Answering dataset contains more than 100K examples). Besides the obvious contribution of creating a new large-scale dataset from scratch, there are a few other paths that could be considered. For example, it would be possible to create a novel benchmark suite containing multiple previous benchmarks.

This would not be a trivial work, since a lot of consideration is needed concerning how to split the datasets in a way that the train and test sets could help developers understand if their system can successfully generalise to unseen DBs, domains, SQL query patterns, NLQ vocabulary, etc. Another way to create a new benchmark could be by transforming similar benchmarks used in slightly different tasks, or different query languages. For example, a text-to-SPARQL dataset, such as the CFQ dataset [56] that contains more than 200 thousand NLQ-to-SPARQL examples or the LC-QuAD dataset [106] that contains 5 thousand examples, would be very beneficial if converted to SQL (similarly to how the WikiTableQuestions [82] was used to create the SQUALL [100] text-to-SQL dataset).

Another critical limitation of existing benchmarks is that they fail to address the question of what type of NL and SQL queries a system can understand and build, respectively. This is due to the lack of a clear query categorization. For instance, Spider has four very coarse-grained classes of queries. This highlights the need for new benchmarks and in-depth system evaluations, in the spirit of [36, 8], that provide fine-grained query categories and allow researchers to understand the strengths and weaknesses of a system.

Furthermore, existing benchmarks assume that for each NL query, there is only one correct SQL query. This may be restricting. First, there are NL queries that may have more than one correct translations over the data. Second, equivalent SQL queries are written in a different way but return the same results.

Finally, while the state-of-the-art systems are still dealing with ‘getting the answer right’, they are mostly overlooking the ‘getting the answer fast’. The database community could come up with benchmarks that focus on efficiency (not just effectiveness) and allow evaluating systems based on execution time and resource consumption in addition to translation accuracy.

2.8.2 System Efficiency & Technical Feasibility

Focusing on the translation accuracy of the system is only one side of the coin. Evaluating system efficiency is important in order to understand the viability of a solution and pinpoint the pain points that need to be addressed. Deep learning text-to-SQL systems are typically relying on very complex models, which have been trained and evaluated in toy databases (like the ones contained in existing benchmarks). Hence, it comes to no surprise that they have not yet seen practical applications in real-life use-cases and domains, and their usefulness is to be proved. Several important challenges need to be tackled first.

Firstly, while the use of PLMs for NL Representation is highly favored by newer systems, these models introduce a large overhead at inference time, and while using larger PLMs usually translates to higher accuracy, it also translates to higher inference times. Output refinement techniques are also adding extra overhead that might make a system impractical to use in a real-world scenario. For example, one of the best-performing models on the Spider dataset, T5-3B+PICARD, uses a large PLM along with a computationally-intensive output refinement technique. Adapting such a model to work with fewer resources, reducing its training time, or optimising its output refinement would be a significant scientific and engineering achievement.

Furthermore, input encoding techniques such as serialisation combined with PLMs have an input size limit (usually of 512 tokens), which poses no problem for the DBs in Spider, but is restricting when working with real-world database schemas. The challenge of creating a robust input encoding technique that can efficiently work with larger schemas, must also be tackled in order to make text-to-SQL systems technically feasible.

Additionally, schema linking techniques have been shown to work and be beneficial for systems working on the Spider dataset, but they have yet to be tested on an real, large-scale DB. Even though using indices and other DB lookup techniques might speed up schema linking, it is still questionable if looking up multiple words or n-grams for every NLQ, is efficient in a real application. Advanced matching techniques, such as classifiers, also introduce additional overhead. There is a lot of room for contributions in optimising schema linking, and this could be the area where the DB community has the most to offer in order to make the breakthroughs of the NLP world usable in practice.

In a nutshell, improving translation speed by building efficient methods is necessary. But this may not be enough. Text-to-SQL translation creates overhead to the overall query execution time that the user will experience, and hence needs to be weighted in. Early text-to-SQL systems originating from the DB community [43, 44, 72, 134, 64] not only tried to generate correct SQL queries but also optimal in terms of execution speed. Hence, many of them contained logic for generating code that would return the desired results fast. Ultimately, allowing the user to express questions in natural language should free them from the technical details of how this query should be expressed in the underlying system language and and how it should be executed efficiently.

2.8.3 Universality of the Solution

Another challenge is the universality of the solution, i.e., performing equally well for different databases. This problem becomes highly relevant when applying a text-to-SQL system to an actual database [41] that is used in a business, research or any other real-world use case. Apart from the large number of tables and attributes that we have already discussed, such databases may contain table and column names that use domain-specific terminology. For example, the SDSS [104] database has attributes such as “speccobj” (spectroscopic object) and “photoobj” (photometric object), that are unknown to and hence cannot be translated by any of the available text-to-SQL systems. That is why in real-life applications, ontologies and domain knowledge are used to enable reliable text-to-SQL translations [91, 5].

It is also important to enable natural language queries in languages other than English, which is the main focus of current efforts. Due to the problem’s multidisciplinarity, database, ML, and NLP approaches can join forces to push the barrier further.

2.8.4 Data Augmentation

The need of deep learning models to train on a high volume of training examples, combined with the relatively small size of available benchmarks and the cost of manually creating new examples, has elevated data augmentation to an important problem.

DBPal [115] is a template-based approach that uses manually-crafted templates of NL/SQL-pairs, which can be filled with the names of tables, columns and values in order to create training instances. The NLQs can be further augmented, with the use of NL techniques such as paraphrasing, random deletions and synonym substitutions. Nevertheless, such templates and NL techniques can not work consistently across all new DBs an might often result to “robotic” or unnatural NLQs. Another approach [39] uses a similar template-based approach to create SQL queries by sampling column names and values from a given table and then applies Recurrent Neural Networks (RNNs) to generate the equivalent NLQ. A more recent work [116] proposes a pipeline that can generate examples spanning over multiple tables of a relational database. SQL queries are created using an abstract syntax tree grammar and filling them with attributes from the database. The NLQs are then generated using a hierarchical, RNN-based neural model, that recursively generates explanations for all parts of the queries and then concatenates them.

However, even though some initial efforts have been made, a systematic evaluation of how each approach affects different systems, as well as the quality of generated data in each case, is still missing. Additionally, another research question that arises is how to train a system using domain-specific or augmented data, along with a general-domain dataset such as Spider. For example, should the system be trained simultaneously on domain-specific as well as general-use data, or only on domain-specific data, or should a more advanced sampling method be used [116]?

2.8.5 The Path to Data Democratisation

While the text-to-SQL problem is a major research challenge, it is also important to understand that it is a piece of the greater puzzle of data democratisation. In order to allow all users, no matter their technical knowledge, to easily access data and to derive value from it, we must consider complementary problems, such as query explanations, query result explanations, and query recommendations. These problems can also benefit from and be inspired by the models and methods presented in our study.

Query Result Explanations. The results of a query are typically presented in a tabular form that is not self-explanatory. Generating NL explanations for query results is another open research area [102, 21]. Interestingly, while there has been considerable work on the "sibling" area of data-to-text generation [9], the problem of query result explanations (or QR-to-text) has several intricacies that do not allow directly adapting methods from the data-to-text generation domain. The need to capture query semantics (that are implied by the results), the lack of appropriate benchmarks, and the fact that query results may contain several rows from different tables that are joined are just a few of the open issues.

Query Recommendations. Even when the user understands the data that is kept in the database, it might not always be clear what kind of queries can be asked and what kind of knowledge can be extracted. For this reason, query recommendations can help a user find interesting queries to ask the database, either based on the user preferences and history, or on queries that are frequently asked by other users of the same database [48] or by analyzing the data [37]. In this context, adapting deep-learning models for query recommendations offers numerous challenges and opportunities.

Conversational Text-to-SQL. Developing a conversational DB interface is another promising task, very similar to earlier non-DL approaches such as Analyza [24], which heavily involves the user in the translation process. Since our ultimate goal is creating a user-friendly and seamless experience, it would be very interesting to allow the user to access and query data solely through the power of natural language and conversation. The release of a conversational (CoSQL [130]), and a context-dependent (SParC [132]) text-to-SQL dataset, based on the Spider [131] dataset, has allowed for more focused progress in this domain. The conversational version of the problem carries new aspects and difficulties that candidate systems must tackle. First and foremost, for each prediction, the system must take into account all previous interactions with the user (i.e., all previous NLQs and the predicted SQL queries). Additionally, it is often necessary to ask the user for clarifications when facing vague questions, or ask the user to chose between possible interpretations of an utterance in the conversation. While some of the systems presented in this work can be adapted to work in a conversational setting, heavier modifications are often necessary in order for the model to effectively encode the conversation history and the previous SQL predictions (note that we have only discussed about encoding NL and DB schemas). Ultimately, this aspect of the problem opens the path towards “intelligent data assistants“ [77], similar to but extremely more powerful than the intelligent personal assistants that are gaining more and more popularity and use through our smartphones and dedicated speakers devices.

3. THE SQL-TO-TEXT PROBLEM: A NOVEL DEEP LEARNING MODEL

3.1 Introduction

Query Languages, such as Structured Query Language (SQL), are a necessary tool in order to access, navigate and obtain data that is stored in a variety of data stores, such as relational databases. However, these query languages often impede non-technical users from accessing valuable data, that is crucial for their work. In an effort to allow casual users (i.e., users without technical expertise) to freely access databases, researchers have been working towards creating Natural Language Interfaces for Databases (NLIDBs) [17]. These interfaces allow users to query databases using only Natural Language (NL) instead of a Query Language (QL). As we discussed in the previous chapter, a lot of work has been put towards creating Text-to-SQL systems that can translate a NL question from the user to a SQL query that will retrieve the desired data. However, given that the user is not familiar with SQL, how can they validate that the translated SQL query actually matches their intent? Providing NL explanations for these queries would allow for a complete NL experience, while also increasing the confidence of casual users that the results they are getting are what they were looking for. Our work focuses on this exact problem of generating NL explanations for SQL queries, also called the SQL-to-Text problem.

Furthermore, SQL-to-Text can also be beneficial for other use cases, both inside and outside the scope of a NLIDB. A more advanced NLIDB might also offer recommendations of SQL queries, in order to help the user discover interesting data stored in the database. Once again, in order for the user to understand the recommendations it is important to provide a NL description of the proposed queries. SQL-to-Text systems can also help experienced users speed up their workflow. Software developers and DB administrators often come across complicated queries that are directed to DBs with schemata they are not familiar with. Getting a quick NL explanation of such queries would save a lot of time. Finally, SQL-to-Text systems are also essential for data augmentation pipelines for training Text-to-SQL systems [39, 117]. In such a pipeline, SQL queries will be generated automatically, either using a predefined grammar or by modifying a set of given SQL queries, and a SQL-to-Text system will be used to generate their NL counterparts. This process will produce augmented NL/SQL pairs that can be used to boost the performance of Text-to-SQL systems.

When generating a query explanation, the challenges to be considered stem from two directions: the SQL side and the NL side of the problem. On the one side, SQL queries may contain complex elements (e.g., nested queries, unions, etc.) that require a better understanding of their purpose to be explained because they do not serve the same purpose every time. For example, the SELECT clause always determines the attributes that will be returned but a nested query might be used to get the students with the best grades in one case or to get the students that passed a certain class in another case. Additionally, a SQL-to-Text system must take into account the schema and the domain of the database when generating an explanation, because they can heavily influence the meaning of the query. On the other side, the generated natural language must appear as fluent and coherent as possible, in order to be understood by a non-expert user. It should also correctly convey the meaning of the query as simply as possible, without any unnecessary repetitions. Finally, additional challenges arise when evaluating a query explanation system. The lack of a dedicated dataset and, more importantly, a metric specifically designed to evaluate query explanations, prevent researchers from assessing a model's performance quickly and confidently. All studies so far use Text-to-SQL benchmarks in reverse order, which are not ideal for evaluating SQL-to-Text systems because they do not offer any difficulty categorisation of their NL parts and might introduce dataset construction biases because they were created for different purposes. Finally, currently available metrics are not robust for evaluating query explanations, which are often very brief and full of condensed information, and can become erroneous by only changing a single word.

Previous efforts at generating SQL query explanations, can be divided into two categories: systems originating from the DB community [30, 49, 58, 59, 101, 115] that follow rule-based approaches that create explanations using pre-defined templates, and systems originating from the NLP community [39, 75, 117, 119] that employ deep learning techniques to solve the problem. Both categories of systems present different drawbacks: the former approaches tend to create explanations that appear “robotic” and need human effort to be applied to an unseen DB (e.g., to construct new rules or templates), while the latter approaches can not guarantee to produce correct explanations every time. Additionally, no system thus far has taken advantage of the latest advances in the field of NLP, such as Transformer-based [108] Pre-trained Language Models (PLMs) (e.g., BERT [23], T5 [92]), which show state of the art performance at text generation and understanding tasks. Finally, to the best of our knowledge, no work thus far has provided a deeper look into the difficulty of evaluating query explanations.

Moving on, we focus on the SQL-to-Text problem in two directions. On the one hand, we discuss the difficulties of generating query explanations and we propose a new model for generating SQL queries that leverages the power of PLMs. On the other hand, we investigate the problem of evaluating query explanations and we create two datasets of query explanations which we use to evaluate existing metrics and fine-tune a new learned metric in order to tackle the drawbacks of current metrics.

3.2 Related Work

SQL-to-Text. The SQL-to-text problem has seen relatively little attention from the research community, and the few available approaches can be classified into two categories: template-based and neural-based. Template-based [30, 49, 58, 59, 101, 115] systems work by constructing a structured representation of the query (e.g., a query graph) and producing a NL explanation for each part of the representation, using templates that are provided by the user. As such, template-based systems can produce very accurate explanations of SQL queries, because they are designed to produce an NL explanation of every part of the query, but require a lot of manual effort in order to create new templates for a new DB that the system must work on. The biggest caveat of template-based systems is that they often generate ”robotic” and unnatural explanations, because they translate every single part of the query, which can lead to repetitions and unnecessary information. A simple example can be seen in the following SQL query:

```
SELECT p.title FROM projects p
WHERE p.start_year >= 2014
    AND p.start_year <= 2018
```

Table 3.1: BLEU scores of neural SQL-to-Text systems on the WikiSQL and Spider datasets

Model	WikiSQL	Spider
Seq-to-Seq [51]	18.40	-
Graph-to-Seq (GNN) [119]	28.70	-
Graph-to-Seq (RGT) [75]	31.20	28.84

Logos [58], a template-based approach, creates a *query-graph*, that contains all the query elements and DB tables that are used in the query, and provides a verbalisation of each node of the graph and paths between specific nodes. At the end, a template-based approach would produce the following explanation: *“Find projects whose start year is greater than or equal to 2014 and start year is less than or equal to 2018.”*, but the query could be explained much more fluently as *“Get the names of projects started between 2014 and 2018.”*.

On the other hand, deep learning solutions offer better generalisation to unseen databases and more fluent explanations, but are not guaranteed to generate accurate explanations every time. These systems consider the SQL-to-Text problem as a text generation task and learn to generate query explanations by being trained on thousands of SQL/NL pairs. Deep learning SQL-to-Text models fall under two categories, based on how they process the input query: sequence-to-sequence models and graph-to-sequence models. Sequence-to-sequence models [39, 117] process the input as a text sequence, either in its entirety [39] or in multiple runs [117], where each run creates an explanation for a different clause of the query, and all runs are subsequently merged, to form the final explanation. The aforementioned sequence-to-sequence models, also incorporate a copy-mechanism [38] in their decoder, which allows them to directly copy tokens from their inputs to the generated explanation (e.g., to use a table or column name in the explanation).

Graph-to-Sequence (or Tree-to-Sequence) approaches [119, 75] create an abstract-sytax-tree of the SQL query, and process it as a structured input. An earlier approach [119] uses an existing graph-to-seq framework [118] that employs GNNs to process the input SQL graph and RNNs to generate the output explanation. However this approach is only tested on queries taken from the WikiSQL [136] dataset, which are written for single tables and not entire databases, and thus are of relatively low complexity. A more recent system [75] takes advantage of Transformer [108] networks to create a *Relation-Aware Graph Transformer* (RGT) encoder, that is more robust at encoding large query graphs, such as the ones resulting from complex SQL queries with multiple clauses. The RGT can encode both the syntax and the relations between the nodes of the tree, in order to provide a more informative representation of the query. However, using a graph encoder rejects the opportunity to use a sequence-to-sequence PLM, and leverage its power at understanding and generating NL, since it is not efficient to tweak the input types of a already pre-trained model.

Table 3.1 displays the scores achieved by previous SQL-to-Text systems on two popular Text-to-SQL benchmarks, the WikiSQL [136] and Spider [131] benchmarks. Despite the small number of proposed systems, we can also note the lack of a dedicated benchmark and metric for the problem. A benchmark created specifically for SQL-to-Text could provide better insights on the categories that a system can accurately explain, and alternative ground truths to help evaluate predictions more accurately. So far, systems have relied on Text-to-SQL benchmarks and the BLEU [81] metric for evaluation.

SPARQL-to-Text. The SPARQL-to-Text [31, 79] problem is also closely related to the problem at hand, but has seen even less attention than SQL-to-Text. These approaches are closely related to the template-based approaches for SQL-to-Text, which create a representation of the query (e.g., a query graph), and apply a pre-defined set of rules to simplify the representation, and eventually verbalise each of its components.

Code summarisation. Finally, the SQL-to-Text problem, can be seen as an instance of the larger code summarisation problem, which aims at verbalising fragments of code from any programming language to NL. This is a highly complicated task, given the intricacies, expressiveness and levels of abstractions of modern programming languages. To the best of our knowledge, researchers have only recently focused on this task, taking advantage of the power of deep learning models, as well as online code repositories that contain vast ammounts of code snippets along with comments and NL descriptions of their use. The first approaches [51] used RNN-based sequence-to-sequence models, which also inspired the SQL-to-Text work mentioned earlier. However, the latest approaches are taking advantage of PLMs [3, 32, 114] to develop models that can summarise code snippets, translate to different programming languages, detect bugs and security issues, and other code understanding and generation tasks. In this area, research is mainly focused on designing the most adequate pre-training tasks and datasets that will give a model all the necessary knowledge to be fine-tuned on many different tasks.

Comparison to our work. Our approach falls into the category of sequence-to-sequence deep learning SQL-to-Text models. To the best of our knowledge, it is the first system to use a Pre-trained Language Model (PLM) based on Transformer [108] networks, instead of the less robust Recurrent Neural Networks (RNNs) which are proven to face difficulties at discovering connections between input elements that do not appear near one another (e.g., an attribute appearing early, in the SELECT clause as well as later on, in the WHERE clause). Additionally, we are the first to explore the use of adaptation techniques and their benefit when adapting our system to a DB taken from the scientific domain. This is a challenge that has been overlooked by previous systems which only consider simpler benchmarks such as Spider and WikiSQL. Finally, we also investigate the performance of various metrics for the SQL-to-Text problem, which previous works have overlooked by relying on n-gram based metrics such as the BLEU [81] score.

3.3 The SQL-to-Text Problem

The Query-to-Text task aims at providing a NL explanation of a structured query, that describes its intent and is understandable by a non-expert. This is a task that falls under the broader research area of code summarisation that aims to provide NL summaries for code in various programming languages. Instances of the Query-to-Text task include the SQL-to-Text task, SPARQL-to-Text task, the Cypher-to-Text task, and many other tasks based on different query languages, used for different types of data stores.

3.3.1 Problem Formulation

The goal of the *SQL-to-Text problem* is the following: Given a SQL query q, directed to a Relational Database with a given schema s, generate an explanation n in Natural Language that accurately describes its intent.

There are multiple desired properties to consider, concerning the explanation such as fluency, brevity and semantic accuracy. More specifically, the generated explanation must: (i) convey the same semantic meaning as the one expressed in the formal query language, (ii) be fluent and human-like, mimicking a human speaker so as not to seem robotic, and (iii) be brief and avoid unnecessary repetitions. However, generating NL with such high standards is far from trivial, especially when it must accurately describe a SQL query that can convey a lot of information.

3.3.2 Challenges

The SQL-to-Text problem hides several challenges, that stem from two parts: (i) the complexity of SQL and DBs, and (ii) the difficulty of generating fluent and accurate NL utterances. First and foremost, such a system should generate fluent and human-like explanations of SQL queries. This requires attention in multiple directions: There must be no unnecessary repetitions (e.g., "projects starting after 2014 and projects starting before 2020" should be "projects starting between 2014 and 2020"). Additionally, there must not be unnecessary over-complications (e.g., "show me actors that play in a movie and other actors who play in the same movie" should be "show me actors who have played in the same movie") Finally, the correct vocabulary should be used according to the DB domain (e.g. "Which artist created the soundtrack of each movie" should be "Which artist composed the soundtrack of each movie"). Similarly, another challenge is correctly identifying the DB domain and using the appropriate vocabulary. For example, the `MAX` aggregation function must be translated in a different way, depending on the context and the attribute on which it is applied. In a DB containing sport data, the `MAX(lap_time)` refers to the *"slowest lap time"*, while in a database containing products, the `MAX(price)` refers to the *"highest product price"*. Finally, the complexity of SQL poses additional challenges in tackling this problem, which are closely related to the ones described above. A SQL-to-Text system might encounter very complex queries that join multiple tables, use multiple attributes, have nested queries, and other complicated clauses. In order to explain complicated queries, the system must be able to understand what needs to be verbalised and in which cases. For example, most joins may not need an explicit verbalisation, but some queries might perform a join over a certain key that has a large impact on the query meaning.

Finally, efficiently and quickly evaluating query explanations is another important challenge. Currently, there is no dedicated metric for the SQL-to-Text problem and almost all previous works either rely on human evaluations, or on automatic metrics created for translation problems (e.g., BLEU [81]). Unfortunately, both choices come with disadvantages: human evaluations require a lot of time and effort, and on the other hand, automatic translation metrics are not ideal for the problem at hand. In fact, metrics such as BLEU rely on n-gram matching to compare two given texts, that is if the same phrases appear in both texts, then a high score is given. However, this approach does not consider the actual meaning of the texts and is not robust when a synonym is used instead of the expected word, as we are going to show.

3.3.3 What is truly a Query Explanation?

A previous study [31] notes three basic types of linguistic expressions that can be employed to express a query in NL: (i) a statement describing the expected data results (e.g., "Actors from Greece."), (ii) a question about the existence of the queried data (e.g., "Which actors are from Greece?"), and (iii) a command requesting the queried data (e.g., "Show all actors from Greece."). However, a query explanation can have additional levels of variation that need to be considered, in order to truly achieve progress. Let us consider the following query, taken from the Spider dataset's dev set, as a running example to demonstrate our point:

```
SELECT name, location, district
FROM shop
ORDER BY number_products DESC
```

If we were to closely follow each element of the query, similarly to how template-based approaches generate explanations, we could produce the following query explanation: *"Return the shop name, shop location and shop district of all shops. Order the results by shop number of products, in descending order"*. While this explanation includes every single element of the query, it is unlikely that a human would explain it in this way, as it includes many repetitions, it is unnecessarily long, and could be expressed much more fluently. A more "relaxed" take on explaining the query could produce the following utterance: *"Show me the shops, ordered by their number of products"*. One could argue, however, that this explanation does not describe the query in an exact manner, and could also cover other similar queries. For example, it does not specify exactly which attributes must be presented (i.e., name, location and district), nor how the results will be ordered (i.e., in descending order of product number), both of which are explicitly specified in the SQL query. On the other hand, it is also valid to argue that this additional information might be redundant for certain queries and omiting it increases the quality of the explanation. A more generally accepted approach would be the following explanation: *"Show me the name, location and district of all shops, in descending order of number of products"*.

Let us now consider a query that contains a GROUP BY clause with a COUNT aggregation function. The following example is also taken from the Spider dev set and is posed on a database containing information about concerts and stadiums where the concerts took place:

```
SELECT T2.name , COUNT(*)
FROM concert AS T1
JOIN stadium AS T2
    ON T1.stadium_id = T2.stadium_id
GROUP BY T1.stadium_id
```

A simple and fluent explanation could be *"How many concerts took place in each stadium?"*. However, in the same spirit as previously, one could say that this explanation does not complete encapsulate the intent of the SELECT clause. Instead, it would be valid to say that the correct interpretation should also include *"Show me the number of concerts per stadium along with the name of the stadium"*. This might seem excessive to some, given that a query that would return a list of numbers without the stadium that corresponds to each number of concerts. However, this depends on the precision and strictness required for describing the SQL query.

Finally, the proper explanation of the asterisk operator and the level of detail it requires, are still debatable. We use the following SQL query as a motivating example:

```
SELECT *
FROM project
WHERE start_year > 2014
```

Let us discuss possible explanations for this query. First of all, the explanation *"Show me projects that started after 2014"* is simple and straight-forward, but one could point out that a query with a clause such as `SELECT project.title` would have the same explanation, and as such more detail is needed. Something along the lines of *"Show information about projects starting after 2014"* or *"Show everything about projects that start after 2014"*, might be more precise, but could still be vague to a non-technical user (e.g., what is "everything"?). A more strict explanation, that would also be clear for non-technical users, could be *"Show me all information stored in the database about projects starting after 2019."*. But it might be possible that there are other tables in the DB that contain information related to projects. Maybe *"Show me all information about projects starting after 2019, that is stored in the table named projects"* is the strictest explanation, but at this point we are clearly drifting away from our main goal of providing fluent and user-friendly explanations.

3.4 A SQL-to-Text System

3.4.1 The Model

We propose the use of a Transformer-based [108] sequence-to-sequence model, following an encoder-decoder architecture. The sequence-to-sequence architecture allows us to parse the input SQL queries with great ease, and at the same time opens up the possibilities to apply our model on different inputs and tasks without the need for changes to the architecture. For example, the model can as easily be trained for SPARQL-to-Text, by only changing the dataset it is trained on, or it can be trained for Text-to-SQL by reversing the inputs and outputs of the dataset. We initialise the model with the T5-base Pre-trained Language Model (PLM), which has been shown to perform very well on various NL Understanding and Generation tasks [92]. We follow the same input format as the one used by T5's authors: $X = p : i$, where p is the task prefix (e.g., "translate SQL to English") and i is the actual input (e.g., the SQL query).

3.4.2 Training Tasks

Given the multi-learning capabilities of the T5 model [92], as well as the benefits that auxiliary learning tasks can have towards the downstream task, we also experiment with additional tasks that may improve the models performance. We propose four configurations for training our model using SQL and SPARQL:

1. SQL-to-Text only (no additional tasks)
2. {SQL, SPARQL}-to-Text (generating explanations for both SQL and SPARQL)
3. SQL-to-Text and Text-to-SQL (generating SQL explanations and queries)

4. {SQL, SPARQL}-to-Text and Text-to-{SQL, SPARQL} (generating SQL and SPARQL explanations and queries)

Using the above configurations, we investigate the effects of training the model on similar tasks (query generation) and languages (SPARQL), on its SQL explanation capabilities. Other possible choices could include code summarisation and generation tasks for other programming languages (e.g., Python, C, etc.), bug and threat classification, or even table understanding tasks. However, we choose to only experiment with the most closely related tasks.

3.4.3 Adapting to Scientific Databases

When applying our model on a database from the scientific domain there are several difficulties that arise, mainly from the use of domain-specific and scientific vocabulary. This is a challenge that is often disregarded, but is very relevant when applying a system in a real-world use case that contains domain-specific vocabulary and knowledge, that the model has never seen during its training. For example, a database containing astronomical data will contain attributes such as `specobj`, `photobj`, and `ra`, that require words such as “spectroscopic”, “photometric”, and “right ascension” in order to explain them. However, our model has never seen this kind of attribute names and vocabulary during its training nor its pre-training. In order to improve performance on such databases we apply two adaptation techniques: (i) descriptive attributes, and (ii) DB-specific training.

Descriptive Attributes. In order to help the model what attributes such as `specobj` refer to, we use descriptive attribute names to make the SQL query easier to explain. Specifically, a domain expert must first provide a list of descriptive and understandable names for each column and table name in the DB. Then an extra pre-processing step is added to the pipeline, where for each query that needs to be explained, its attributes will be automatically substituted by the equivalent descriptive attributes. The neural model will then be given the pre-processed query with the descriptive attributes, instead of the cryptic attributes that are understandable only by domain experts. For example, the query:

```
SELECT COUNT(specobjid)
FROM specobj
```

which is explained as “Count the number of specobjs.”, by the model, will be transformed to:

```
SELECT COUNT(spectroscopic_object_id)
FROM spectroscopic_object
```

This query will be much easier to explain by our model, which in fact generates the following, much more fluent, explanation: “How many spectroscopic objects are there?”.

DB-Specific Training. Another method for adapting the model to a scientific database is by specifically training it on examples of queries and query explanations take from the given database. This approach also requires manual work from a domain expert, in order to produce high quality SQL and explanation pairs. In fact this approach will require a lot more effort, because in order to have a considerable impact, the DB-specific training examples must be of a considerable size and quality. This also requires the domain expert to have some technical knowledge in using relational databases and writing SQL queries, as well as to be familiar with the schema of the DB at hand. However, despite the additional effort required, this approach can help the system learn how to verbalise the queries posed on the DB, like a domain expert would do.

In summary, the two aforementioned adaptation techniques allow for three different approaches when applying our model to a database with domain-specific vocabulary: (i) train the model on a general knowledge dataset and use descriptive attributes when explaining queries for the given database, (ii) train the model on both a general knowledge dataset and a dataset created for the given database, and (iii) use both techniques at the same time. In the third case, the model is trained on both a general knowledge dataset and a dataset from the given database, and the descriptive attributes are used both during training and inference. Experiments for all these cases are shown in Section 3.7.

3.5 Benchmarks

3.5.1 (No) SQL-to-Text Benchmarks

Currently, there are no benchmarks created specifically for the SQL-to-Text task. This is because the problem remains relatively unexplored, compared to other similar problems in the area. Nevertheless, there are multiple available benchmarks for the reverse problem (i.e., Text-to-SQL). Even though these dataset were not created to be used for the task we aim to tackle, they can be adapted to SQL-to-Text-Problem by using their inputs (i.e. natural language queries) as targets and their outputs (i.e. SQL queries) as inputs. However, this approach should not be seen as a panacea, but rather as a temporary solution. Firstly, these datasets are created specifically for the Text-to-SQL task and can carry dataset construction biases. For example, the NL parts of some examples might be intentionally malformed or incomplete in order to make them more challenging for Text-to-SQL systems. However, when these examples are used as ground truths for SQL-to-Text systems, we are essentially teaching our systems to generate low quality explanations. Additionally, a benchmark should provide fine-grained categories of explanations and queries that help developers evaluate the capabilities of their systems and to uncover their drawbacks.

Spider [131] is one of the most influential and widely-used dataset in the Text-to-SQL domain. This will be our main dataset used for training and evaluating our model, due to its widespread use and proven quality. The Spider dataset contains 10,181 NL questions and 5,693 SQL queries, posed over 200 relational database coming from 138 different domains. For this work we only use the train and dev splits of the dataset, as the test set is held out by the authors for evaluation purposes.

We also use two additional, smaller-scale, benchmarks coming from scientific use cases:

SDSS[1] (Sloan Digital Sky Survey) is a very large astrophysics database, containing the most detailed three-dimensional map of the universe ever made. It contains data collected from a 5-m wide-angle optical telescope at Apache Point Observatory in New Mexico, United States, that has been collecting observations from 2000 and keeps going on until today. The original SDSS database contains 10 tables each of which contains up to many hundreds of columns. Fir this work, we use a subset of the database comprised of 5

[1] https://www.sdss.org/

original tables from SDSS and 1 additional table for photometrically observed astronomical objects. There are 61 columns, averaging about 10 columns per table.

OncoMX is a database used for cancer research and contains information about cancer mutation and expression, and cancer biomarkers. The version of OncoMX we use contains information from cancer biomarker databases (EDRN6, FDA7), gene expression in healthy anatomical entities (Bgee8), differential gene expression between healthy and cancerous samples (BioX-press9) and cancer mutations (BioMuta10). The database comprises 25 tables that have 2 to 14 columns each, for a total of 111 columns.

Table 3.2 provides additional information about the size and SQL hardness[2] of all the datasets used in this work. The hardness metrics for Spider were calculated by its authors, while the hardness for the SDSS and OncoMx datasets were calculated by us, using the tools provided by Spider's authors.

Table 3.2: SQL hardness statistics of the datasets used in this work. The SQL hardness classification is based on the criterion proposed by the Spider dataset

Dataset	Easy		Medium		Hard		Extra Hard		Total
Spider Train	1944	22.4%	2831	32.7%	1758	20.3%	2126	24.5%	8659
Spider Dev	248	23.9%	446	43.1%	174	16.8%	166	16.0%	1034
SDSS Train	20	20%	54	54%	2	2%	24	24%	100
SDSS Test	12	12%	28	28%	20	20%	40	40%	100
OncoMX Train	21	42%	20	40%	7	14%	2	4%	50
OncoMX Test	39	37.8%	49	47.5%	11	10.6%	4	3.8%	103

3.5.2 Two New Benchmarks for Metric Evaluation

Another challenge for SQL-to-Text systems is the lack of a dedicated metric for evaluating them. Until now, researchers have evaluated their system either with user evaluations [119], which are costly, or with automatic translation metrics (e.g., BLEU [81]), which are not ideal for this problem (more in Section 3.6). For this reason, we create two benchmarks that will help us evaluate the performance of different metrics, so as to better understand how reliable they are for the SQL-to-Text problem. Both datasets contain pairs of query explanations, along with a label that indicates whether the two explanations are semantically equivalent (i.e., if they describe the same query). This format helps us evaluate the currently available metrics, that provide a score for a generated explanation given a reference (i.e., ground truth) explanation. An ideal metric would provide a very high score if the label indicates that the two explanation are equivalent and a very low score if the label indicates that they are not equivalent. We will now describe how both datasets were created.

3.5.2.1 Qx-Paraphrase: A Metric Benchmark by Paraphrasing

We create a benchmark by paraphrasing query explanations from the Spider [131] dataset, which we refer to as *Qx-Paraphrase*. To do so, we begin by randomly extracting 100

[2] More information on the harness criteria can be found in the Spider github repo (`https://github.com/taoyds/spider`).

examples from the Spider dev set. For each example, we generate a semantically equivalent explanation with different vocabulary, using publicly available paraphrasing tools[3][4], and manually validate them afterwards to ensure their correctness and semantic equivalence. We also manually create an incorrect paraphrase of the original example, that uses very similar vocabulary, by changing a small number of words that drastically change the explanation's meaning (e.g., older instead of younger, more instead of less, etc.). By doing so we create a small corpus containing 100 correct explanation pairs, and 100 incorrect explanation pairs. Some examples of such pairs can be seen in Table 3.3. The Qx-Paraphrase benchmark can help us evaluate the robustness of metrics to synonyms and paraphrases, as well as predictions that use very similar vocabulary but are incorrect due to small differences.

Table 3.3: Examples from the Qx-Paraphrase benchmark: Each original query is paraphrased twice, once with a semantically equivalent (correct) paraphrase, and once with a different (incorrect) meaning.

Original NLQ		**Correct/Incorrect Paraphrases**
How many states are there?	✓	What is the total number of states?
	✗	How many **counties** are there?
Which year has most number of concerts?	✓	Which year is the busiest in terms of concerts?
	✗	Which year has **smallest** number of concerts?
What is the average weight for each type of pet?	✓	How much does each type of pet weigh on average?
	✗	What is the average **age** for each type of pet?

3.5.2.2 Qx-Annotate: A Metric Benchmark by Annotating

We create the *Qx-Annotate* benchmark by performing a human annotation of predicted query explanations. More specifically, we use all models described in Sections 3.4.2 and 3.4.3 (i.e., 4 alternatives for training tasks and 4 alternatives for adapting to scientific databases) and apply them to the aforementioned datasets. For each example, the experts were given the input SQL query along with the ground truth NL explanation and the prediction of a model, and were asked to classify the prediction as correct or incorrect. The resulting benchmark consists of 1800 pairs of ground truth and predicted explanations, along with a label indicating whether the predicted explanation is correct. Our expert group consists of a combination of 7 MSc and PhD students, whose work is mainly focused on the DB field. For the SDSS and OncoMX datasets, we ask human experts to evaluate all examples of their test sets, while for the Spider dataset they evaluate a random sample of 350 examples out of the 1034 examples of the dev set. Because the Qx-Annotate benchmark is created using real predictions of SQL-to-Text models, it can provide better insights on how a metric performs in a realistic use case.

3.6 (No) SQL-to-Text Metrics

Let us now consider an unaddressed challenge closely realated to the SQL-to-Text problem: How can we measure the correctness of a candidate (predicted) NL explanation of a query against the reference (ground truth) explanation provided by a benchmark? Most works so far have relied on automatic translation metrics such as the BLEU score

[3] `https://quillbot.com/`

[4] `https://www.prepostseo.com/paraphrasing-tool`

[81]. However, are automatic translation metrics, designed for evaluating abstract utterances, sufficient for evaluating NL explanations of code snippets like SQL queries, where a change of column name can completely change its semantics? As shown in Table 3.4, a candidate explanation that has similar vocabulary to the reference but refers to the wrong column is rated much higher than a candidate that is slightly paraphrased but semantically equivalent.

In this section, we present various automatic metrics that are available and could be suitable for the problem at hand, grouping them in two large categories based on how they work: (i) n-gram based metrics and (ii) learned metrics. We also talk about the limitations and drawbacks of these metrics and fine-tune a learned metric specifically for the SQL-to-Text problem. Finally, we use the, previously described, Qx-Paraphrase and Qx-Annotate benchmarks, to evaluate the performance of all the metrics at hand and present our findings.

3.6.1 Automatic Metrics

3.6.1.1 N-Gram-Based Metrics

BLEU (BiLingual Evaluation Understudy) score [81] is an automatic evaluation metric that was originally proposed for machine translation. Besides machine translation, this is a very frequently-used evaluation metric for text generation tasks such as text summarisation and image captioning. It evaluates the quality of a candidate translation against one or more reference translations based on n-gram precision. More specifically, n-gram precision is the number of n-grams of the candidate translation that are also present in the reference translation. Since its introduction, many different implementations have been published, which make it difficult to have a consistent comparison between different works. This problem is described in [88].

ROUGE (Recall-Oriented Understudy for Gisting Evaluation) [68] score is an automatic evaluation metric proposed for text summarisation. Similarly to n-gram precision, n-gram recall refers to the number of n-grams in the reference that also appear in the candidate translation.

chrF [86] and **chrF++** [87] automatic translation metrics that take into account character n-grams along with word n-grams. More specifically, the initial chrF metric is based completely on character n-gram matching while the later chrF++ combines both character and word n-grams.

METEOR (Metric for Evaluation of Translation with Explicit Ordering) score [7] is another automatic metric for machine translation based on n-gram matching. It combines both n-gram precision and recall and also takes into account the ordering of the candidate translation compared to the reference.

3.6.1.2 Learned Metrics

Sentence Transformers [93] is a state of the art technique for creating sentence embeddings using siamese networks of BERT-like models. Essentially, a BERT-like model is used to create a sentence embedding for the candidate and reference query explanations and the cosine similarity of their embeddings is used as a metric of their semantic similarity. For the purposes of this work, we use the `all-mpnet-base-v2` model due to its exceptional performance on semantic search tasks, compared to all available models.

BLEURT [96] use the BERT's [CLS] token and a linear layer to predict a score. The main difference to Sentence Transformers is that BLEURT, processes the two inputs (i.e., reference and candidate) at the same time, and not separately. The output [CLS] token contains a contextualised representation based on both inputs, which is then used to predict a score through a linear layer. Although the model is trained to produce outputs in the range of $[-1, 1]$, it is not possible to guarantee that its output will always be in this range, due to the nature of the model. For this reason, we clip its output between $[-1, 1]$ before, normalising it to the common range.

3.6.2 Limitations of Current Metrics

Even though all prior works rely on the BLEU score to evaluate their systems, this metric is not ideal for the SQL-to-Text problem. Given that such metrics rely on n-gram matching, they are not robust to the use of different vocabulary, expressions, and syntax that could convey the same meaning. For example, the explanations "How many singers do we have?" and "What is the number of singers?" are semantically equivalent, despite only sharing one common word. Additionally, as the authors of BLEU state: *"[...] quantity leads to quality"*, that is, BLEU is most efficient when applying it on large corpora. However, query explanations are usually short texts loaded with very specific information, and changing even a single word can completely alter their meaning, rendering them incorrect. For example "How many singers do we have?" describes a different query than "How many songs do we have?". The scores given to the aforementioned examples can be seen in Table 3.4. It is clear that n-gram based metrics will favor the wrong explanation, because it uses similar vocabulary and syntax.

On the other hand, learned metrics are more robust at capturing the semantics of their inputs due to their training. As such, they could be more promising for the SQL-to-Text task, but can not be seen as a panacea. For this category we can identify two drawbacks: Firstly, these metrics usually rely on deep learning models and the scores they produce are not explainable and occasionally might not be completely reproducible. Additionally, even though these metrics can provide a semantic comparison between texts, they are not guaranteed to rate candidate explanations, on the exact criteria of the SQL-to-Text problem. For example, the ground truth "Tell me the age of the oldest dog." and the prediction "Tell me the age of the youngest dog.", are arguably semantically similar, but do not describe the same query. However the Sentence Transformer metric gives a higher score to this prediction than the correct prediction "How old is the eldest dog?", as shown in Table 3.4.

Table 3.4: Examples of NL explanations of SQL queries and their respective scores. Notice how classic automatic translation metrics can favor the wrong explanation.

Reference	Candidates	BLEU	chrF++	METEOR	s-Trans
How many singers do we have?	How many **songs** do we have?	48.89	68.49	80.66	61.07
	What is the number of singers?	7.80	24.15	0.00	89.09
Tell me the age of the oldest dog.	Tell me the age of the **youngest** dog.	66.06	72.83	86.47	90.28
	How old is the eldest dog?	5.66	37.42	16.12	81.16

3.6.3 BLEURT-Q: Fine-tuning BLEURT for Query Explanations

Based on the aforementioned observations about the limitations of the available metrics for SQL-to-Text, we decide to fine-tune a learned metric specifically for this task. Towards this goal, we use the Qx-Paraphrase dataset (a 80% train split) and the BLEURT [96] model. The authors of BLEURT claim that their model can be fine-tuned to provide better evaluations for a given task or domain, even with a small amount of examples. To do so, it is necessary to provide a training set that is comprised of candidate and reference translation pairs along with a rating r of the candidate, where $r \in [-1, 1]$. In our case, a rating of $r = 1$ will be assigned to pairs where the candidate explanation is semantically equivalent to the reference (despite the use of different vocabulary), and a rating of $r = -1$ will be assigned to pairs where the candidate explanation describes a different query than the reference (despite the use of the same vocabulary). We use this fine-tuned BLEURT model (denoted as *BLEURT-Q*(uery) for the rest of the paper), for our experiments in Sections 3.6.4 and 3.7.

3.6.4 Evaluating Metric Performance

Moving on, we will try to get some experimental insights on how well the aforementioned metrics perform on query explanations. We will use the Qx-Annotate and the Qx-Paraphrase (a 20% test split) datasets to evaluate 4 n-gram based metrics (BLEU, Rouge, METEOR, and chrF) and three learned metrics (Sentence Transformers, BLEURT, and BLEURT-Q). For each dataset, we calculate the scores given by each metric on the correct and the incorrect pairs, and display the average score for each in Figure 3.1.

All the results that we present have been normalized in the common range of $[0, 100]$, with 0 being the worse and 100 being the best score. This is because different metrics might return scores in different ranges (e.g., $[0, 1]$ or $[-1, 1]$).

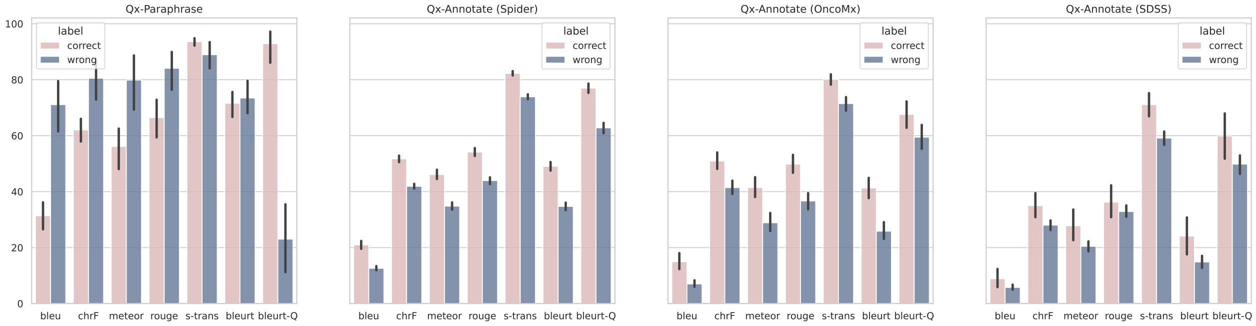


Figure 3.1: A comparison of scores given by automatic metrics on the Qx-Paraphrase dataset and the Qx-Annotate dataset (Spider/SDSS/OncoMx sub-sets shown separately). An ideal metric would assign high scores to correct explanations and low scores to incorrect explanations.

Looking at the results we can make some interesting observations. First of all, on the handmade dataset, we observe that all n-gram-based metrics are struggling, and are consistently ranking wrong explanations with higher scores compared to correct explanations. The learned metrics perform better as the Sentence Transformers and BLEURT metrics provide similar scores to both correct and incorrect metrics, and the fine-tuned BLEURT metric performs exceptionally giving very high scores to correct explanations and low scores to wrong explanations. However, this ideal performance does not transfer to the rest of the datasets. On the three other datasets, we observe that all metrics, no matter if they are trained, fine-tuned or n-gram-based perform very similarly. In fact, in almost all cases, all metrics rank the correct explanations higher than the wrong explanations, with each metric having a different scale and difference between the two scores. Furthermore, the fine-tuned BLEURT metric seems to perform worse than the rest of the metrics, having the least difference between correct and incorrect explanations and in one case, even giving higher scores to incorrect explanations.

These observations lead us to some conclusions. First of all, we should not be too quick to dismiss n-gram-based metrics completely. Even though experiments on the Qx-Paraphrase (which contains difficult examples) show very bad performance for n-gram metrics, experiments on the Qx-Annotate dataset (which contains examples from real model predictions) show that n-gram metrics performance is very close to the performance of learned metrics. Second, there is no metric that can make a clear distinction between correct and incorrect explanations, as all metrics give relatively similar scores to all examples. Finally, while the BLEURT-Q metric beats all other metrics on the Qx-Paraphrase dataset, the fact that its performance is similar to all other metrics on the Qx-Annotate dataset leads us to believe that we should use a larger variety of examples to fine-tune it.

3.7 System Evaluation

We move on with the evaluation of our proposed model, which aims to answer two questions: (i) Which training tasks (discussed in Section 3.4.2) help the model achieve the best performance at the SQL-to-Text task, and (ii) Which adaptation techniques (discussed in Section 3.4.3) help the model achieve the best performance on new DB taken from the scientific use cases. For the first set of experiments we use the Spider dataset and for the second set of experiemts we use the SDSS and OncoMx scientific datasets. For all experiments we use two n-gram based metrics (BLEU and chrF), three learned metrics (Sentence Transformers, BLEURT, and BLEURT-Q), as well as an user evaluation performed by experts of the DB domain. For each example, the experts were given the input SQL query along with the ground truth NL explanation and the prediction of a model, and were asked to classify the prediction as correct or incorrect. Our expert group consists of a combination of 7 MSc and PhD students, whose work is mainly focused on the DB field. For the SDSS and OncoMX datasets, we ask human experts to evaluate all examples of their test sets, while for the Spider dataset they evaluate a random sample of 350 examples out of the 1034 examples of the dev set.

All the results that we present have been normalized in the common range of $[0, 100]$, with 0 being the worse and 100 being the best score. This is because different metrics might return scores in different ranges (e.g., $[0, 1]$ or $[-1, 1]$). The experts' score refers to the percentage of predictions the experts labeled as correct.

3.7.1 Experiment I: Learning Tasks

Our first experiment is centered around the use of additional learning tasks, that might aid the model at achieving higher performance in the SQL-to-Text task. We examine four configurations: (i) Query-to-Text for SQL only, (ii) Query-to-Text for SQL and SPARQL, (iii) Query-to-Text and Text-to-Query for SQL, and (iv) Query-to-Text and Text-to-Query for SQL and SPARQL. Table 3.5 displays the performance on the SQL-to-Text task of each one of the aforementioned models, evaluated on the Spider dev set. We also include the

Table 3.5: SQL-to-Text performance scores on the Spider Dev Set for different training configurations

Training Tasks	BLEU	chrF	s-Trans	BLEURT	BLEURT-Q	Experts
SQL-to-Text	18.16	48.46	**79.70**	44.04	**81.89**	**58.57**
{SQL, SPARQL}-to-Text	**18.63**	**48.61**	79.48	**44.24**	80.67	55.71
SQL-to-Text, Text-to-SQL	16.41	46.46	78.03	40.70	79.80	49.14
{SQL, SPARQL}-to-Text, Text-to-{SQL, SPARQL}	13.57	43.80	75.62	37.10	79.64	31.71
RGT [75]	28.84	-	-	-	-	-

performance of RGT [75], which to our knowledge is the only system to be evaluated on Spider. RGT is reported to achieve a higher BLEU score compared to our model, which could be an indicator that the graph representation of the input SQL query helps the system capture the semantics of the query more efficiently than the sequence representation we use.

Looking at the experts' evaluation, we observe that the first model performs the best, producing a correct explanation about 58% of the time, while the second model achieves a similar performance, being correct about 55% of the time. Adding the Query-to-Text tasks in the training procedure (lines 3 and 4), we observe a large loss in performance, with a 10% and 27% decrease in correct explanations when training only for SQL and for both SQL and SPARQL respectively. This could be explained due to the fact that when the model learns to perform additional tasks, its performance on one of them at a time drops. However, the {SQL, SPARQL}-to-Text model seems very promising, as it only sacrifices 3% of accuracy in order to learn an additional query language.

Moreover, we observe that all metrics tend to rank the models similarly, or even in the same way as the experts' evaluation does. The fine-tuned BLEURT metric ranks the first model as the best, with the second one being closely behind, but does not show a big decrease for the latter two models in the same way that the experts did. The same behavior is seen by the Sentence-Transformers metric. The rest of the metrics give a slight advantage to the second model, with the first one being slightly behind, while the latter two models achieve lower scores, with the last one having the worst scores, similarly to the experts' evaluation.

3.7.2 Experiment II: New Domains

Table 3.6: Performance scores on new domain-specific DBs, using different adaptation techniques

Database	Adaptation Technique	BLEU	chrF	s-Trans	BLEURT	BLEURT-Q	Experts
SDSS	None	3.40	21.11	52.39	6.61	55.80	8.00
	Descriptive Attributes	**8.61**	**34.57**	65.34	20.57	75.54	**22.00**
	DB-specific Training	6.88	27.07	59.91	12.22	64.96	18.00
	Both	6.43	34.34	**67.37**	**26.57**	**78.44**	**22.00**
OncoMX	None	9.71	44.62	76.02	30.71	75.94	42.00
	Descriptive Attributes	8.03	42.34	70.03	29.89	77.68	34.00
	DB-specific Training	**13.95**	**50.50**	**80.48**	**40.80**	**78.46**	**72.00**
	Both	13.49	48.65	77.87	35.17	75.12	66.00

Moving on, we only train the model for the SQL-to-Text task, and evaluate its performance on the new scientific domains presented by the SDSS and OncoMX databases. Additionally, we evaluate the best technique for adapting it to these difficult and unseen domains. We examine the aforementioned adaptation techniques: using descriptive attributes, DB-specific training, and a combination of both:

Table 3.6 displays the performance on each database, following different adaptation techniques. For each database, we evaluate our base model without any adaptation technique, then with the descriptive attributes, with DB-specific training, and finally with both descriptive attributes and DB-specific training. For the SDSS database, we observe that initially the model's performance is quite low, with experts indicating that it produces a correct answer only 8% of the time. We also observe that all adaptation techniques improve the model's performance, with the descriptive attributes achieving the best performance, of being correct 22% of the time. The same performance can be achieved by combining both techniques, although given the cost of additional training, using only descriptive attributes can be considered the best choice. Another observation is that all metrics tend to agree with the experts' evaluation, with the n-gram-based metrics showing a preference to the model that uses only the descriptive attributes, while neural-based metrics favor the model using both descriptive attributes and DB-specific training. For most metrics these two scores are relatively close to each other, and clearly higher than the first model (i.e., no adaptation technique). Additionally, all metrics except BLEU show that the two best-performing models are better than the model using only DB-training.

On the OncoMx dataset, we observe a different behavior between different adaptation techniques. All metrics, except for the fine-tuned BLEURT, agree with the experts' evaluation on the ranking of different approaches: Using descriptive attributes produces the worst performance, even worse than using no adaptation technique at all, and using DB-specific training only achieves the best performance, surpassing the combination of both adaptation techniques. Additionally, the experts' evaluation indicates a much higher percentage of correct explanations on this dataset, compared to the SDSS dataset, where in the best case we achieve 72% of correct explanations compared to only 22% for the best model in SDSS. This increased performance is also mirrored on the metrics which also show higher scores across all techniques, compared to the scores on the SDSS database. The only exception to this is the fine-tuned BLEURT metric, whose scores are not that different between the two databases (e.g., 78.44 when the experts indicate a 22% correctness on SDSS, and 78.46 when the experts indicate a 72% correctness on OncoMx).

3.7.3 Example predictions

Let us now examine some predictions from the aforementioned experiments, in order to better understand how different models perform with different inputs, and how the experts' evaluations match the scores given by the metrics. Figure 3.2 contains examples taken from the first experiment on the Spider dataset, and Figure 3.3 contains examples from the second experiment, on the SDSS and OncoMX databases.

3.7.3.1 Examples from Experiment I

Let us first consider the predictions on the Spider dataset, from Figure 3.2. The example shown in Figure 3.2a displays 4 explanations that are all very close to being correct, but have minor errors. The first explanation contains the phrase "in descending **alphabetical** order of number of products", which is incorrect since the number of products is a number and is not ordered alphabetically. The second explanation is also very close to being correct, but does not specify if the ordering should be in ascending or descending order. Admittedly, one could argue that this omission is not that big of a mistake to render the prediction incorrect, or even that this makes the explanation more fluent and is preferable to omit the order type. The third explanation is the only one that the experts marked as correct. This explanation does not omit any part of the SQL query, but one could argue that it is incorrect due to the phrase “for all shops **and items**”, since the attributes “name, location, and district” only refer to shops and not to items. The fourth explanation does not include the “district” attribute, which is why it has been marked as incorrect by the experts. What is also interesting that this explanation uses the indicative mood, which gives a very different feel to most explanations which are posed as questions or orders. Additionally, we observe that all metrics favor the explanation that was labeled as correct by the experts.

Moving on to the example in Figure 3.2b, we observe that no model managed to generate a correct explanation. This is in fact a more complex SQL query, containing many different clauses such as JOIN, GROUP BY, ORDER BY, etc. The first prediction is mostly correct but does not mention the attribute `Level_of_membership` that appears in the SELECT clause. The second prediction, is also very close to being correct but refers to “spent the most **time**”, when the attribute `Total_spent` that appears in the query is actually referring to money instead of time. Admittedly, this is an error that could have been made by an expert as well, given that the model does not have access to any information besides the query, such as the data inside the table, or additional information about the attributes. The third prediction, is the most bizarre of the four, where the model generated the phrase “who just stayed there”. This prediction is probably related to the model’s pre-training for generating generic natural language, as such a phrase is unlikely to occur in a query explanation. The fourth prediction, is also very close to being correct but was probably marked as incorrect because the phrase “who had the most spent” does not make a lot of sense syntactically. However, a phrasing such as “who pent the most” would probably have been correct, while also avoiding to mention what was being spent, which was ambiguous from just reading the query. Finally, all queries rank the second prediction as the one closest to the ground truth, besides incorrectly referring to spending time instead of money.

Figure 3.2c shows another example with a query with many clauses but not that many attributes. The first prediction is mostly correct but makes one important mistake by asking the year with the “**least** matches”, instead of the “most matches”. The second prediction is probably the most incomprehensible, as it is not clear what the model’s intention was. The third explanation is the only one that was labeled as correct by the experts, and it is completely on point on conveying the query’s semantics. The fourth explanation, is not correct for two reasons: the query does not “find **matches**”, it finds the year, and it has missed the LIMIT clause, which combined with the ORDER BY clause returns the maximum or the minimum depending on the ordering type. Additionally, we observe that most metrics incorrectly rank the first prediction as the best one, except for the fine-tuned BLEURT metric which manages to rank the correct explanation as the best one.

3.7.3.2 Examples from Experiment II

Let us now consider some examples from the SDSS and OncoMX datasets, taken from experiment II, shown in Figure 3.3.

The example in Figure 3.3a is taken from the SDSS dataset. The first prediction, does not correctly translate the attribute `photobj`, because the model does not have such knowledge. Instead it provides an incorrect explanation of "photo types" instead of "photometric objects". The second prediction however is correct. Thanks to the descriptive attributes that were added, the model was able to correctly use the phrase "photometric objects" and even translated `photo_type` as "classification". The third prediction was marked as correct by the experts, but could also have been labeled as incorrect given that it has the same mistake as the first prediction, with the generated phrase "photo type" not being exactly equivalent to "photometric object". The fourth prediction is incorrect, as it does not grasp the meaning of the WHERE clause and produces a rather incoherent phrase such as "the star that is a STAR". Additionally, we observe that the BLEU, chrF and Sentence-Transformers metrics favor the fourth predictions which is incorrect, while BLEURT ranks the third prediction as the best, even though it is arguably not the best. The fine-tuned BLEURT model is the only metric that ranks the most precise prediction (i.e., the second one) with the highest score.

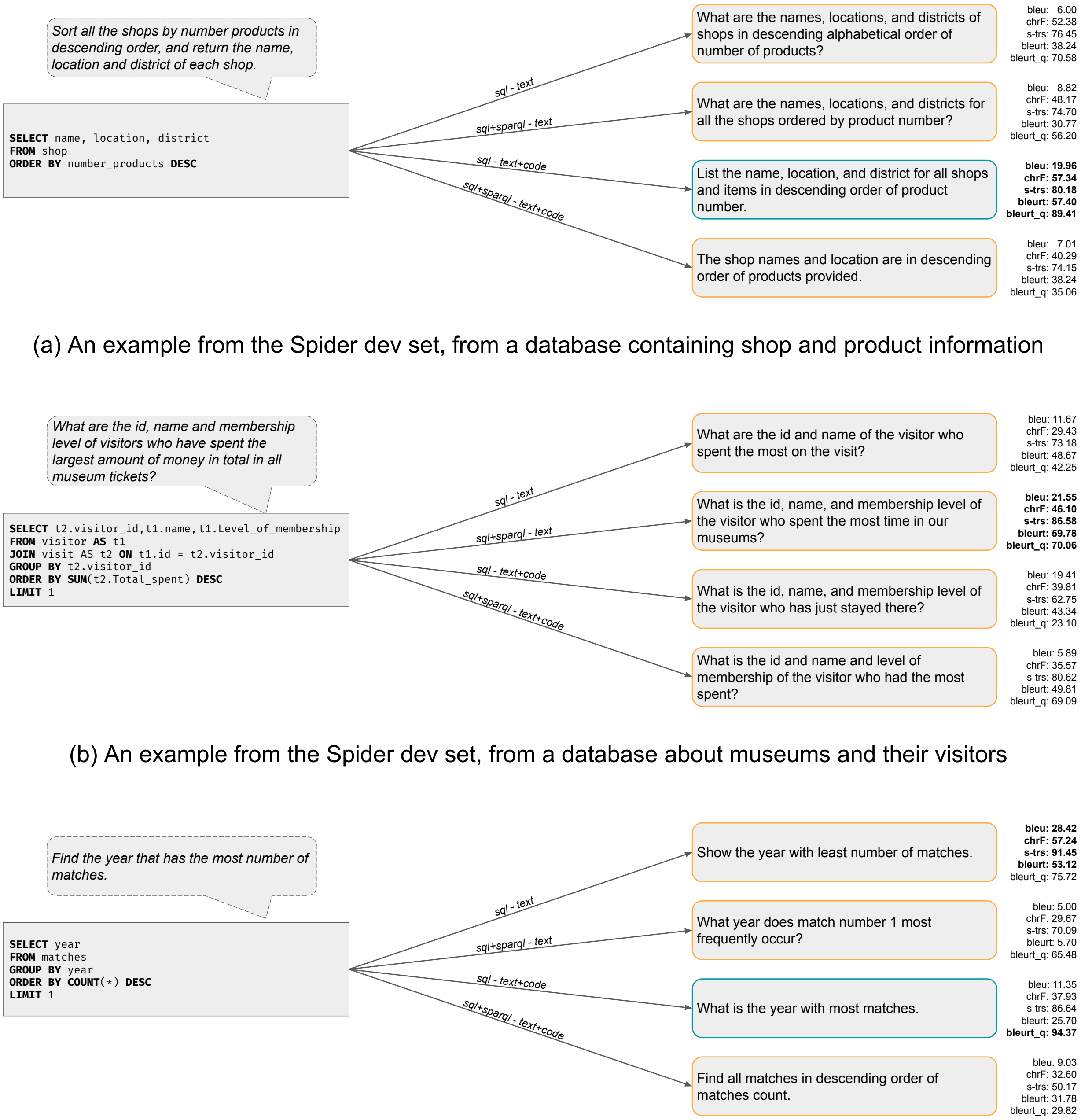
The examples show in Figure 3.3b is taken from the OncoMx dataset and contains a long SQL query that joins three different tables. Only the third prediction was marked as correct and it is in fact the most comprehensive and accurately describes the SQL query. The rest of the predictions are incorrect either because they fail to capture that the query returns all information (asterisk operator) about disease mutations, or they miss the condition of the query which limits the results to mutations concerning the liver. All of the automatic metrics rank the third prediction as the best, with the only exception being the fine-tuned BLEURT metric which favors the fourth prediction.

Finally, the example in Figure 3.3c shows another examples from the OncoMx dataset, containing a COUNT aggregation function. In this case we have three predictions marked as correct by the experts and only the first prediction being marked as incorrect. The first prediction is incorrect because it refers to "medicines" instead of tests or test instruments. The three other predictions all correctly capture the intention of the SQL query, eventhough they phrase the NL explanation with slight differences. The automatic metrics distribute their highest ratings between the correct explanations, with the second prediction being favored by BLEURT, the third explanation being favored by BLEU, and the fourth explanation being favored by chrF, sentence transformers and fine-tuned BLEURT.

3.7.4 Summary of Observations

To summarise, we identify five categories of errors from our observations: (i) omitting DB elements, (ii) reversing conditions, (iii) misunderstanding inferred details, and (iv) hallucinations. The first two categories are related to the system's understanding of the SQL query; we observe that in some cases the system will not verbalise all columns that appear in the SELECT clause, or that it will incorrectly reverse conditions (e.g., "**least**" instead of "most"). The third category contains errors that occur because the model was asked to infer details that were not avaiable in its input (e.g., explaining the attribute `total_spent` as "total spent **time**" instead of "total spent money"). Finally, the problem of hallucinations (i.e., generating text that is not related to the input) is an already known problem of language models [52] but we also observe types of hallucinations that are specific to the task we study. For example, some explanations might contain extra information that does not appear in the query (e.g., "**alphabetical** order" when the ordered attribute is not alphanumerical), or additional attributes that do not appear in the query (e.g., "shops and

items” when the query only refers to stores).



Sort all the shops by number products in descending order, and return the name, location and district of each shop.

```
SELECT name, location, district
FROM shop
ORDER BY number_products DESC
```

What are the names, locations, and districts of shops in descending alphabetical order of number of products?

bleu: 6.00
chrF: 52.38
s-trs: 76.45
bleurt: 38.24
bleurt_q: 70.58

What are the names, locations, and districts for all the shops ordered by product number?

bleu: 8.82
chrF: 48.17
s-trs: 74.70
bleurt: 30.77
bleurt_q: 56.20

List the name, location, and district for all shops and items in descending order of product number.

bleu: 19.96
chrF: 57.34
s-trs: 80.18
bleurt: 57.40
bleurt_q: 89.41

The shop names and location are in descending order of products provided.

bleu: 7.01
chrF: 40.29
s-trs: 74.15
bleurt: 38.24
bleurt_q: 35.06

(a) An example from the Spider dev set, from a database containing shop and product information

What are the id, name and membership level of visitors who have spent the largest amount of money in total in all museum tickets?

```
SELECT t2.visitor_id,t1.name,t1.Level_of_membership
FROM visitor AS t1
JOIN visit AS t2 ON t1.id = t2.visitor_id
GROUP BY t2.visitor_id
ORDER BY SUM(t2.Total_spent) DESC
LIMIT 1
```

What are the id and name of the visitor who spent the most on the visit?

bleu: 11.67
chrF: 29.43
s-trs: 73.18
bleurt: 48.67
bleurt_q: 42.25

What is the id, name, and membership level of the visitor who spent the most time in our museums?

bleu: 21.55
chrF: 46.10
s-trs: 86.58
bleurt: 59.78
bleurt_q: 70.06

What is the id, name, and membership level of the visitor who has just stayed there?

bleu: 19.41
chrF: 39.81
s-trs: 62.75
bleurt: 43.34
bleurt_q: 23.10

What is the id and name and level of membership of the visitor who had the most spent?

bleu: 5.89
chrF: 35.57
s-trs: 80.62
bleurt: 49.81
bleurt_q: 69.09

(b) An example from the Spider dev set, from a database about museums and their visitors

Find the year that has the most number of matches.

```
SELECT year
FROM matches
GROUP BY year
ORDER BY COUNT(*) DESC
LIMIT 1
```

Show the year with least number of matches.

bleu: 28.42
chrF: 57.24
s-trs: 91.45
bleurt: 53.12
bleurt_q: 75.72

What year does match number 1 most frequently occur?

bleu: 5.00
chrF: 29.67
s-trs: 70.09
bleurt: 5.70
bleurt_q: 65.48

What is the year with most matches.

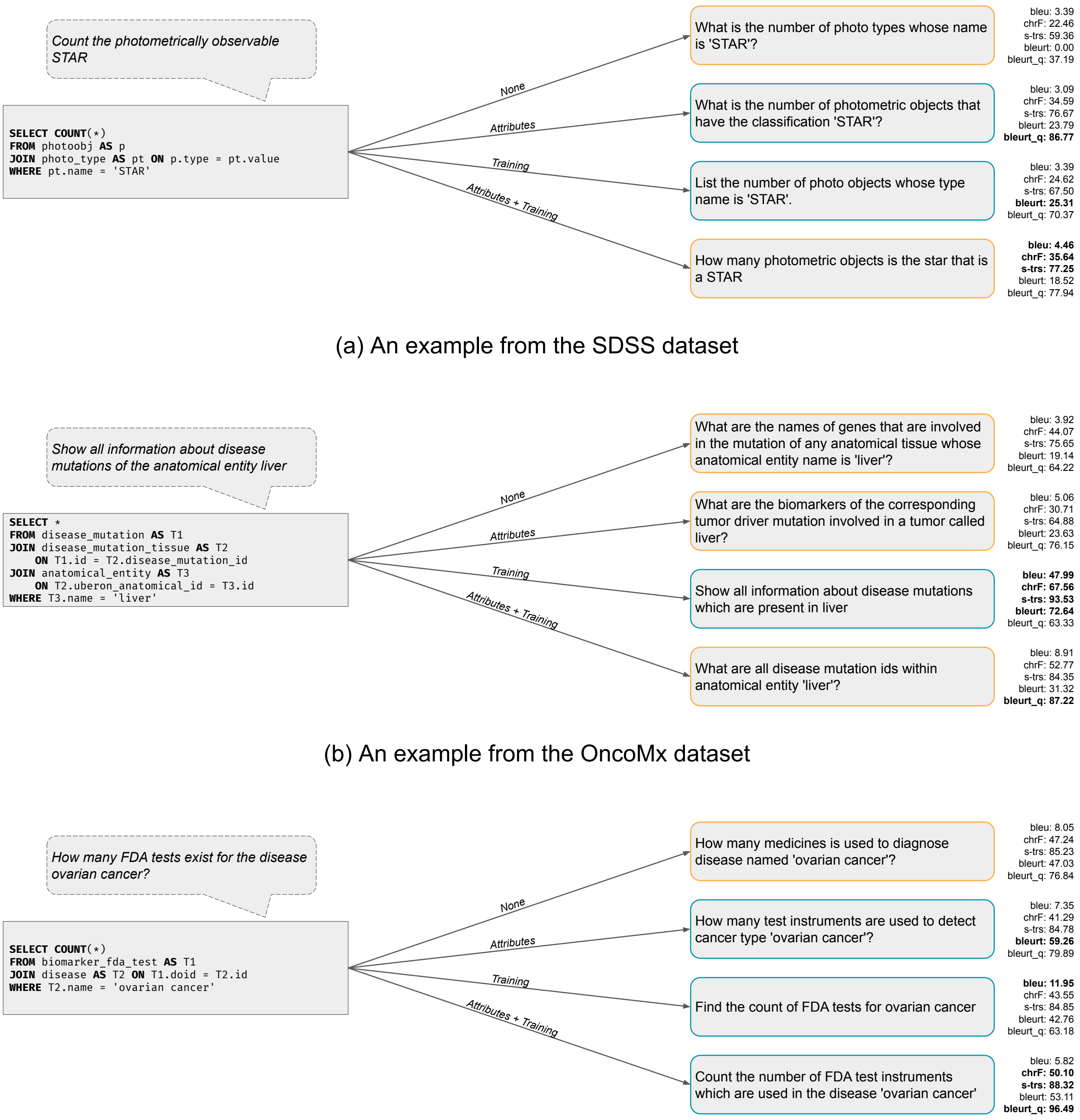
bleu: 11.35
chrF: 37.93
s-trs: 86.64
bleurt: 25.70
bleurt_q: 94.37

Find all matches in descending order of matches count.

bleu: 9.03
chrF: 32.60
s-trs: 50.17
bleurt: 31.78
bleurt_q: 29.82

(c) An example from the Spider dev set, from a database containing match and stadium information

Figure 3.2: Examples of predictions on the Spider dev set, for each input we provide the output from the four different models trained with the previously discussed configuration. The scores of each metric are included next to each prediction, with the highest score of each metric being in bold.



Count the photometrically observable STAR

```
SELECT COUNT(*)
FROM photoobj AS p
JOIN photo_type AS pt ON p.type = pt.value
WHERE pt.name = 'STAR'
```

What is the number of photo types whose name is 'STAR'?

bleu: 3.39
chrF: 22.46
s-trs: 59.36
bleurt: 0.00
bleurt_q: 37.19

What is the number of photometric objects that have the classification 'STAR'?

bleu: 3.09
chrF: 34.59
s-trs: 76.67
bleurt: 23.79
bleurt_q: 86.77

List the number of photo objects whose type name is 'STAR'.

bleu: 3.39
chrF: 24.62
s-trs: 67.50
bleurt: 25.31
bleurt_q: 70.37

How many photometric objects is the star that is a STAR

bleu: 4.46
chrF: 35.64
s-trs: 77.25
bleurt: 18.52
bleurt_q: 77.94

(a) An example from the SDSS dataset

Show all information about disease mutations of the anatomical entity liver

```
SELECT *
FROM disease_mutation AS T1
JOIN disease_mutation_tissue AS T2
    ON T1.id = T2.disease_mutation_id
JOIN anatomical_entity AS T3
    ON T2.uberon_anatomical_id = T3.id
WHERE T3.name = 'liver'
```

What are the names of genes that are involved in the mutation of any anatomical tissue whose anatomical entity name is 'liver'?

bleu: 3.92
chrF: 44.07
s-trs: 75.65
bleurt: 19.14
bleurt_q: 64.22

What are the biomarkers of the corresponding tumor driver mutation involved in a tumor called liver?

bleu: 5.06
chrF: 30.71
s-trs: 64.88
bleurt: 23.63
bleurt_q: 76.15

Show all information about disease mutations which are present in liver

bleu: 47.99
chrF: 67.56
s-trs: 93.53
bleurt: 72.64
bleurt_q: 63.33

What are all disease mutation ids within anatomical entity 'liver'?

bleu: 8.91
chrF: 52.77
s-trs: 84.35
bleurt: 31.32
bleurt_q: 87.22

(b) An example from the OncoMx dataset

How many FDA tests exist for the disease ovarian cancer?

```
SELECT COUNT(*)
FROM biomarker_fda_test AS T1
JOIN disease AS T2 ON T1.doid = T2.id
WHERE T2.name = 'ovarian cancer'
```

How many medicines is used to diagnose disease named 'ovarian cancer'?

bleu: 8.05
chrF: 47.24
s-trs: 85.23
bleurt: 47.03
bleurt_q: 76.84

How many test instruments are used to detect cancer type 'ovarian cancer'?

bleu: 7.35
chrF: 41.29
s-trs: 84.78
bleurt: 59.26
bleurt_q: 79.89

Find the count of FDA tests for ovarian cancer

bleu: 11.95
chrF: 43.55
s-trs: 84.85
bleurt: 42.76
bleurt_q: 63.18

Count the number of FDA test instruments which are used in the disease 'ovarian cancer'

bleu: 5.82
chrF: 50.10
s-trs: 88.32
bleurt: 53.11
bleurt_q: 96.49

(c) An example from the OncoMx dataset

Figure 3.3: Examples of predictions on the SDSS and OncoMx datasets, for each input we provide the output from the four different models trained with the previously discussed adaptation techniques. The scores of each metric are included next to each prediction, with the highest score of each metric being in bold.

4. CONCLUSION AND FUTURE WORK

In this thesis, we studied two sides of enabling users to query databases with NL: the Text-to-SQL and SQL-to-Text problems. We believe that both problems are of equal importance towards the path to data democratisation. However, not both problems have been studied to the same extent. For this reason, our work on each problem was split to two different directions.

For the Text-to-SQL problem, we provided a fine-grained taxonomy of deep learning Text-to-SQL systems, based on six axes: (*a*) Schema Linking, (*b*) Natural Language Representation, (*c*) Input Encoding, (*d*) Output Decoding, (*e*) Neural Training, and (*f*) Output Refinement. For each axis of our taxonomy, we analysed all the approaches that have been presented so far and explained their strengths and weaknesses. We relied on this taxonomy to present some of the most important systems that have been proposed, grouping them together, in order to highlight their similarities, differences and innovations. Finally, having presented the current state of the art, we discussed open challenges and research opportunities that must be tackled in order to truly advance the field of Text-to-SQL, as well as broader challenges that are closely related to it. It is important to keep in mind, that the ultimate goal of Text-to-SQL research is to empower the casual user to access and derive value from data. This is a goal that requires the combined effort of multiple disciplines and can not be measured by a single performance metric.

For the SQL-to-Text problem, we proposed the use of Transformer-based PLM for generating fluent and human-like explanations, along with three additional training tasks to improve the model's performance and two adaptation techniques to tackle the challenge of applying a SQL-to-Text model on a new DB from the scientific domain. We also created two benchmarks for evaluating SQL-to-Text metrics and used them to evaluate currently available metrics and to fine-tune a learned metric specifically for this task. Finally, we presented our evaluation on SQL-to-Text metrics and on our model's performance using existing metrics, our fine-tuned metric, and a user evaluation.

Moving forward we identify several challenges that remain open. For the Text-to-SQL problem, we identify a large need to optimise the approaches proposed by the NLP community and make their use more feasible alongside a RDBMS. While a lot of innovative techniques have been proposed, they mostly rely on very large models that require a lot of processing power. In order to make NLIDBs a reality we need to make Text-to-SQL systems faster, more efficient, and easier to deploy. This also dictates the creation of larger and more realistic benchmarks, that more closely resemble the DBs that are use in real life applications. For the SQL-to-Text problem, we identify two points for future research. Firstly, we believe that additional light should be shed on efficiently evaluating SQL-to-Text systems, in two directions: (i) with the creation of a dedicated metric that will take the requirements of the problem and will also use the given SQL query besides just the ground truth and predicted explanation, and (ii) with the creation of a dedicated dataset both for training and evaluating SQL-to-Text systems. Additionally, further research is due on the use of PLMs for the SQL-to-Text task either by experimenting with newer and larger models, or by the combining graph-to-seq architecture, proposed by previous models, with the power of NL pre-training.

ABBREVIATIONS - ACRONYMS

SQL	Structured Query Language
NL	Natural Language
DB	Database
RDBMS	Relational Database Management Systems
ML	Machine Learning
DL	Deep Learning
PLM	Pre-trained Language Model

BIBLIOGRAPHY

[1] Shanza Abbas, Muhammad Umair Khan, Scott Uk-Jin Lee, Asad Abbas, and Ali Kashif Bashir. A review of NLIDB with deep learning: Findings, challenges and open issues. *IEEE Access*, 2022.

[2] Katrin Affolter, Kurt Stockinger, and Abraham Bernstein. A comparative survey of recent natural language interfaces for databases. *The VLDB Journal*, 28(5):793–819, Aug 2019.

[3] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668, Online, June 2021. Association for Computational Linguistics.

[4] Ambiguity. https://stanford.io/2YXcECi.

[5] Sihem Amer-Yahia, Georgia Koutrika, Martin Braschler, Diego Calvanese, Davide Lanti, Hendrik Lücke-Tieke, Alessandro Mosca, Tarcisio Mendes de Farias, Dimitris Papadopoulos, Yogendra Patil, Guillem Rull, Ellery Smith, Dimitrios Skoutas, Srividya Subramanian, and Kurt Stockinger. Inode: Building an end-to-end data exploration system in practice. *SIGMOD Rec.*, 50(4):23–29, jan 2022.

[6] Ion Androutsopoulos, Graeme D. Ritchie, and Peter Thanisch. Natural language interfaces to databases - an introduction. *Natural Language Engineering*, 1(1):29–81, 1995.

[7] Satanjeev Banerjee and Alon Lavie. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*, pages 65–72, Ann Arbor, Michigan, June 2005. Association for Computational Linguistics.

[8] Theofilos Belmpas, Orest Gkini, and Georgia Koutrika. Analysis of database search systems with THOR. In David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 2681–2684. ACM, 2020.

[9] Jonathan Berant, Daniel Deutch, Amir Globerson, Tova Milo, and Tomer Wolfson. Explaining queries over web tables to non-experts. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1570–1573, 2019.

[10] Ben Bogin, Jonathan Berant, and Matt Gardner. Representing schema structure with graph neural networks for text-to-SQL parsing. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4560–4565, Florence, Italy, July 2019. Association for Computational Linguistics.

[11] Ben Bogin, Matt Gardner, and Jonathan Berant. Global reasoning over database structures for text-to-SQL parsing. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3659–3664, Hong Kong, China, November 2019. Association for Computational Linguistics.

[12] Ursin Brunner and Kurt Stockinger. Valuenet: A neural text-to-SQL architecture incorporating values. *ArXiv*, abs/2006.00888, 2020.

[13] Ruichu Cai, Boyan Xu, Zhenjie Zhang, Xiaoyan Yang, Zijian Li, and Zhihao Liang. An encoder-decoder framework translating natural language to database queries. In Jérôme Lang, editor, *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pages 3977–3983. ijcai.org, 2018.

[14] Ruisheng Cao, Lu Chen, Zhi Chen, Yanbin Zhao, Su Zhu, and Kai Yu. LGESQL: Line graph enhanced text-to-SQL model with mixed local and non-local relations. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 2541–2555, Online, August 2021. Association for Computational Linguistics.

[15] DongHyun Choi, Myeong Cheol Shin, EungGyun Kim, and Dong Ryeol Shin. RYANSQL: Recursively applying sketch-based slot fillings for complex text-to-sql in cross-domain databases, 2020.

[16] E. F. Codd. Seven steps to rendezvous with the casual user. In J. W. Klimbie and K. L. Koffeman, editors, *Data Base Management, Proceeding of the IFIP Working Conference Data Base Management, Cargèse, Corsica, France, April 1-5, 1974*, pages 179–200. North-Holland, January 1974.

[17] E. F. Codd. Seven steps to rendezvous with the casual user. In *IFIP Working Conference Data Base Management*, 1974.

[18] Deborah A. Dahl, Madeleine Bates, Michael Brown, William Fisher, Kate Hunicke-Smith, David Pallett, Christine Pao, Alexander Rudnicky, and Elizabeth Shriberg. Expanding the scope of the atis task: The atis-3 corpus. In *Proceedings of the Workshop on Human Language Technology*, HLT '94, page 43–48, USA, 1994. Association for Computational Linguistics.

[19] Fred J. Damerau. A technique for computer detection and correction of spelling errors. *Commun. ACM*, 7(3):171–176, March 1964.

[20] Naihao Deng, Yulong Chen, and Yue Zhang. Recent advances in text-to-SQL: A survey of what we have and what we expect. In *Proceedings of the 29th International Conference on Computational Linguistics*, pages 2166–2187, Gyeongju, Republic of Korea, October 2022. International Committee on Computational Linguistics.

[21] Daniel Deutch, Nave Frost, and Amir Gilad. Explaining natural language query results. *VLDB J.*, 29(1):485–508, 2020.

[22] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding, 2019.

[23] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[24] Kedar Dhamdhere, Kevin S. McCurley, Ralfi Nahmias, Mukund Sundararajan, and Qiqi Yan. *Analyza: Exploring Data with Conversation*. ACM, 2017.

[25] Li Dong and Mirella Lapata. Language to logical form with neural attention, 2016.

[26] Li Dong and Mirella Lapata. Coarse-to-fine decoding for neural semantic parsing, 2018.

[27] Timothy Dozat and Christopher D. Manning. Deep biaffine attention for neural dependency parsing, 2017.

[28] Timothy Dozat and Christopher D. Manning. Deep biaffine attention for neural dependency parsing. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[29] Timothy Dozat and Christopher D. Manning. Simpler but more accurate semantic dependency parsing, 2018.

[30] Stavroula Eleftherakis, Orest Gkini, and Georgia Koutrika. Let the database talk back: Natural language explanations for sql. *Proceedings of the 2nd Workshop on Search, Exploration, and Analysis in Heterogeneous Datastores, co-located with VLDB 2021*, 2021.

[31] Basil Ell, Denny Vrandečić, and Elena Simperl. Spartiqulation: Verbalizing sparql queries. In Elena Simperl, Barry Norton, Dunja Mladenic, Emanuele Della Valle, Irini Fundulaki, Alexandre Passant, and Raphaël Troncy, editors, *The Semantic Web: ESWC 2012 Satellite Events*, pages 117–131, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[32] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online, November 2020. Association for Computational Linguistics.

[33] Catherine Finegan-Dollak, Jonathan K. Kummerfeld, Li Zhang, Karthik Ramanathan, Sesh Sadasivam, Rui Zhang, and Dragomir Radev. Improving text-to-SQL evaluation methodology. In *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.

[34] Yujian Gan, Xinyun Chen, Qiuping Huang, Matthew Purver, John R. Woodward, Jinxia Xie, and Pengsheng Huang. Towards robustness of text-to-SQL models against synonym substitution. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 2505–2515, Online, August 2021. Association for Computational Linguistics.

[35] Yujian Gan, Xinyun Chen, and Matthew Purver. Exploring underexplored limitations of cross-domain text-to-SQL generalization. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8926–8931, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics.

[36] Orest Gkini, Theofilos Belmpas, Yannis Ioannidis, and Georgia Koutrika. An in-depth benchmarking of text-to-sql systems. In *SIGMOD Conference*. ACM, 2021.

[37] Apostolos Glenis and Georgia Koutrika. Pyexplore: Query recommendations for data exploration without query logs. In Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 2731–2735. ACM, 2021.

[38] Jiatao Gu, Zhengdong Lu, Hang Li, and Victor O.K. Li. Incorporating copying mechanism in sequence-to-sequence learning. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1631–1640, Berlin, Germany, August 2016. Association for Computational Linguistics.

[39] Daya Guo, Yibo Sun, Duyu Tang, Nan Duan, Jian Yin, Hong Chi, James Cao, Peng Chen, and Ming Zhou. Question generation from SQL queries improves neural semantic parsing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1597–1607, Brussels, Belgium, October-November 2018. Association for Computational Linguistics.

[40] 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, 2019.

[41] Moshe Hazoom, Vibhor Malik, and Ben Bogin. Text-to-SQL in the wild: A naturally-occurring dataset based on stack exchange data. In *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*, pages 77–87, Online, August 2021. Association for Computational Linguistics.

[42] Pengcheng He, Yi Mao, Kaushik Chakrabarti, and Weizhu Chen. X-sql: reinforce schema representation with context, 2019.

[43] Vagelis Hristidis, Luis Gravano, and Yannis Papakonstantinou. Efficient IR-style keyword search over relational databases. In *VLDB*, pages 850–861, 2003.

[44] Vagelis Hristidis and Yannis Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, pages 670–681, 2002.

[45] Binyuan Hui, Xiang Shi, Ruiying Geng, Binhua Li, Yongbin Li, Jian Sun, and Xiaodan Zhu. Improving text-to-SQL with schema dependency learning, 2021.

[46] Wonseok Hwang, Jinyeong Yim, Seunghyun Park, and Minjoon Seo. A comprehensive exploration on wikisql with table-aware word contextualization, 2019.

[47] Radu Cristian Alexandru Iacob, Florin Brad, Elena-Simona Apostol, Ciprian-Octavian Truică, Ionel Alexandru Hosu, and Traian Rebedea. Neural approaches for natural language interfaces to databases: A survey. In *Proceedings of the 28th International Conference on Computational Linguistics*, pages 381–395, Barcelona, Spain (Online), December 2020. International Committee on Computational Linguistics.

[48] Stratos Idreos, Olga Papaemmanouil, and Surajit Chaudhuri. Overview of data exploration techniques. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 277–281. ACM, 2015.

[49] Yannis Ioannidis. From databases to natural language: The unusual direction. In Epaminondas Kapetanios, Vijayan Sugumaran, and Myra Spiliopoulou, editors, *Natural Language and Information Systems*, pages 12–16, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[50] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy, and Luke Zettlemoyer. Learning a neural semantic parser from user feedback. In Regina Barzilay and Min-Yen Kan, editors, *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, pages 963–973. Association for Computational Linguistics, 2017.

[51] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083, Berlin, Germany, August 2016. Association for Computational Linguistics.

[52] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Yejin Bang, Wenliang Dai, Andrea Madotto, and Pascale Fung. Survey of hallucination in natural language generation. *ACM Computing Surveys*, 2022.

[53] Aishwarya Kamath and Rajarshi Das. A survey on semantic parsing. In *1st Conference on Automated Knowledge Base Construction, AKBC 2019, Amherst, MA, USA, May 20-22, 2019*, 2019.

[54] George Katsogiannis-Meimarakis and Georgia Koutrika. A deep dive into deep learning approaches for text-to-SQL systems. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD/PODS '21, page 2846–2851, New York, NY, USA, 2021. Association for Computing Machinery.

[55] George Katsogiannis-Meimarakis and Georgia Koutrika. Deep learning approaches for text-to-SQL systems. In *EDBT*, pages 710–713, 2021.

[56] Daniel Keysers, Nathanael Schärli, Nathan Scales, Hylke Buisman, Daniel Furrer, Sergii Kashubin, Nikola Momchev, Danila Sinopalnikov, Lukasz Stafiniak, Tibor Tihon, et al. Measuring compositional generalization: A comprehensive method on realistic data. *arXiv preprint arXiv:1912.09713*, 2019.

[57] Hyeonji Kim, Byeong-Hoon So, Wook-Shin Han, and Hongrae Lee. Natural language to sql: Where are we today? *Proc. VLDB Endow.*, 13(10):1737–1750, June 2020.

[58] Andreas Kokkalis, Panagiotis Vagenas, Alexandros Zervakis, Alkis Simitsis, Georgia Koutrika, and Yannis Ioannidis. Logos: A system for translating queries into narratives. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, page 673–676, New York, NY, USA, 2012. Association for Computing Machinery.

[59] Georgia Koutrika, Alkis Simitsis, and Yannis E. Ioannidis. Explaining structured queries in natural language. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 333–344, 2010.

[60] Jayant Krishnamurthy, Pradeep Dasigi, and Matt Gardner. Neural semantic parsing with type constraints for semi-structured tables. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 1516–1526, Copenhagen, Denmark, September 2017. Association for Computational Linguistics.

[61] John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of the Eighteenth International Conference on Machine Learning*, ICML '01, page 282–289, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[62] Chia-Hsuan Lee, Oleksandr Polozov, and Matthew Richardson. KaggleDBQA: Realistic evaluation of text-to-SQL parsers. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 2261–2273, Online, August 2021. Association for Computational Linguistics.

[63] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension, 2019.

[64] Fei Li and H. V. Jagadish. Constructing an interactive natural language interface for relational databases. *PVLDB*, 8(1):73–84, September 2014.

[65] Yunyao Li and Davood Rafiei. Natural language data management and interfaces: Recent development and open challenges. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 1765–1770, New York, NY, USA, 2017. Association for Computing Machinery.

[66] Yunyao Li and Davood Rafiei. *Natural Language Data Management and Interfaces*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2018.

[67] Zhuang Li, Lizhen Qu, and Gholamreza Haffari. Context dependent semantic parsing: A survey. In *Proceedings of the 28th International Conference on Computational Linguistics*, pages 2509–2521, Barcelona, Spain (Online), December 2020. International Committee on Computational Linguistics.

[68] Chin-Yew Lin. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain, July 2004. Association for Computational Linguistics.

[69] Xi Victoria Lin, Richard Socher, and Caiming Xiong. Bridging textual and tabular data for cross-domain text-to-SQL semantic parsing. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 4870–4888, Online, November 2020. Association for Computational Linguistics.

[70] Xiaodong Liu, Pengcheng He, Weizhu Chen, and Jianfeng Gao. Multi-task deep neural networks for natural language understanding, 2019.

[71] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach, 2019.

[72] Yi Luo, Xuemin Lin, Wei Wang, and Xiaofang Zhou. Spark: Top-k keyword query in relational databases. In *ACM SIGMOD*, pages 115–126, 2007.

[73] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation, 2015.

[74] Qin Lyu, Kaushik Chakrabarti, Shobhit Hathi, Souvik Kundu, Jianwen Zhang, and Zheng Chen. Hybrid ranking network for text-to-sql. Technical Report MSR-TR-2020-7, Microsoft Dynamics 365 AI, March 2020.

[75] Da Ma, Xingyu Chen, Ruisheng Cao, Zhi Chen, Lu Chen, and Kai Yu. Relation-aware graph transformer for sql-to-text generation. *Applied Sciences*, 12(1), 2022.

[76] Jianqiang Ma, Zeyu Yan, Shuai Pang, Yang Zhang, and Jianping Shen. Mention extraction and linking for SQL query generation. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6936–6942, Online, November 2020. Association for Computational Linguistics.

[77] Antonis Mandamadiotis, Georgia Koutrika, Stavroula Eleftherakis, Apostolis Glenis, Dimitrios Skoutas, and Yannis Stavrakas. Datagent: The imminent age of intelligent data assistants. *Proc. VLDB Endow.*, 14(12):2815–2818, 2021.

[78] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In Yoshua Bengio and Yann LeCun, editors, *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, 2013.

[79] Axel-Cyrille Ngonga Ngomo, Lorenz Bühmann, Christina Unger, Jens Lehmann, and Daniel Gerber. Sorry, i don't speak sparql: Translating sparql queries into natural language. In *Proceedings of the 22nd International Conference on World Wide Web*, WWW '13, page 977–988, New York, NY, USA, 2013. Association for Computing Machinery.

[80] Notes on ambiguity. http://bit.ly/2YTLFeR.

[81] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA, July 2002. Association for Computational Linguistics.

[82] Panupong Pasupat and Percy Liang. Compositional semantic parsing on semi-structured tables. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1470–1480, Beijing, China, July 2015. Association for Computational Linguistics.

[83] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.

[84] Ana-Maria Popescu, Alex Armanasu, Oren Etzioni, David Ko, and Alexander Yates. Modern natural language interfaces to databases: Composing statistical parsing with semantic tractability. In *COLING*, 2004.

[85] Ana-Maria Popescu, Oren Etzioni, and Henry Kautz. Towards a theory of natural language interfaces to databases. In *Proceedings of the 8th International Conference on Intelligent User Interfaces*, IUI '03, page 149–157, New York, NY, USA, 2003. Association for Computing Machinery.

[86] Maja Popović. chrF: character n-gram F-score for automatic MT evaluation. In *Proceedings of the Tenth Workshop on Statistical Machine Translation*, pages 392–395, Lisbon, Portugal, September 2015. Association for Computational Linguistics.

[87] Maja Popović. chrF++: words helping character n-grams. In *Proceedings of the Second Conference on Machine Translation*, pages 612–618, Copenhagen, Denmark, September 2017. Association for Computational Linguistics.

[88] Matt Post. A call for clarity in reporting BLEU scores. In *Proceedings of the Third Conference on Machine Translation: Research Papers*, pages 186–191, Brussels, Belgium, October 2018. Association for Computational Linguistics.

[89] P. J. Price. Evaluation of spoken language systems: The atis domain. In *Proceedings of the Workshop on Speech and Natural Language*, HLT '90, page 91–95, USA, 1990. Association for Computational Linguistics.

[90] Abdul Quamar, Vasilis Efthymiou, Chuan Lei, and Fatma Özcan. Natural language interfaces to data. *Found. Trends Databases*, 11(4):319–414, may 2022.

[91] Abdul Quamar, Fatma Özcan, Dorian Miller, Robert J Moore, Rebecca Niehus, and Jeffrey Kreulen. Conversational bi: An ontology-driven conversation system for business intelligence applications. *Proc. VLDB Endow.*, 13(12):3369–3381, aug 2020.

[92] 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. *Journal of Machine Learning Research*, 21(140):1–67, 2020.

[93] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 11 2019.

[94] Ohad Rubin and Jonathan Berant. Smbop: Semi-autoregressive bottom-up semantic parsing, 2020.

[95] Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. PICARD: Parsing incrementally for constrained auto-regressive decoding from language models, 2021.

[96] Thibault Sellam, Dipanjan Das, and Ankur Parikh. BLEURT: Learning robust metrics for text generation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7881–7892, Online, July 2020. Association for Computational Linguistics.

[97] Jaydeep Sen, Chuan Lei, Abdul Quamar, Fatma Ozcan, Vasilis Efthymiou, Ayushi Dalmia, Greg Stager, Ashish Mittal, Diptikalyan Saha, and Karthik Sankaranarayanan. ATHENA++: Natural language querying for complex nested sql queries. *Proc. VLDB Endow.*, 13(11):2747–2759, 2020.

[98] Peter Shaw, Philip Massey, Angelica Chen, Francesco Piccinno, and Yasemin Altun. Generating logical forms from graph representations of text and entities. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 95–106, Florence, Italy, July 2019. Association for Computational Linguistics.

[99] Tianze Shi, Kedar Tatwawadi, Kaushik Chakrabarti, Yi Mao, Alex Polozov, and Weizhu Chen. IncSQL: Training incremental text-to-sql parsers with non-deterministic oracles. Technical Report MSR-TR-2018-36, Microsoft, November 2018.

[100] Tianze Shi, Chen Zhao, Jordan Boyd-Graber, Hal Daumé III, and Lillian Lee. On the potential of lexicological alignments for semantic parsing to SQL queries. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1849–1864, Online, November 2020. Association for Computational Linguistics.

[101] Alkis Simitsis and Yannis Ioannidis. DBMSs should talk back too. *arXiv preprint arXiv:0909.1786*, 2009.

[102] Alkis Simitsis, Georgia Koutrika, and Yannis Ioannidis. Précis: from unstructured keywords as queries to structured databases as answers. *The VLDB Journal*, 17(1):117–149, 2008.

[103] R. Speer and Catherine Havasi. Representing general relational knowledge in conceptnet 5. In *LREC*, 2012.

[104] Alexander S Szalay, Jim Gray, Ani R Thakar, Peter Z Kunszt, Tanu Malik, Jordan Raddick, Christopher Stoughton, and Jan vandenBerg. The sdss skyserver: public access to the sloan digital sky server data. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 570–581, 2002.

[105] Lappoon R. Tang and Raymond J. Mooney. Automated construction of database interfaces: Intergrating statistical and relational learning for semantic parsing. In *2000 Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora*, pages 133–141, Hong Kong, China, October 2000. Association for Computational Linguistics.

[106] Priyansh Trivedi, Gaurav Maheshwari, Mohnish Dubey, and Jens Lehmann. Lc-quad: A corpus for complex question answering over knowledge graphs. In *International Semantic Web Conference*, pages 210–218. Springer, 2017.

[107] Arif Usta, Akifhan Karakayali, and Özgür Ulusoy. Dbtagger: Multi-task learning for keyword mapping in NLIDBs using bi-directional recurrent neural networks. *Proc. VLDB Endow.*, 14(5):813–821, jan 2021.

[108] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[109] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks, 2017.

[110] Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. Rat-sql: Relation-aware schema encoding and linking for text-to-sql parsers, 2020.

[111] Chenglong Wang, Alvin Cheung, and Rastislav Bodík. Synthesizing highly expressive SQL queries from input-output examples. In *38th ACM SIGPLAN*, pages 452–466, 2017.

[112] Chenglong Wang, Kedar Tatwawadi, Marc Brockschmidt, Po-Sen Huang, Yi Mao, Oleksandr Polozov, and Rishabh Singh. Robust text-to-sql generation with execution-guided decoding, 2018.

[113] 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*, pages 350–361, 2020.

[114] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics.

[115] Nathaniel Weir, Prasetya Utama, Alex Galakatos, Andrew Crotty, Amir Ilkhechi, Shekar Ramaswamy, Rohin Bhushan, Nadja Geisler, Benjamin Hättasch, Steffen Eger, Ugur Cetintemel, and Carsten Binnig. Dbpal: A fully pluggable nl2sql training pipeline. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 2347–2361, New York, NY, USA, 2020. Association for Computing Machinery.

[116] Kun Wu, Lijie Wang, Zhenghua Li, Ao Zhang, Xinyan Xiao, Hua Wu, Min Zhang, and Haifeng Wang. Data augmentation with hierarchical SQL-to-question generation for cross-domain text-to-SQL parsing. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8974–8983, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics.

[117] Kun Wu, Lijie Wang, Zhenghua Li, Ao Zhang, Xinyan Xiao, Hua Wu, Min Zhang, and Haifeng Wang. Data augmentation with hierarchical SQL-to-question generation for cross-domain text-to-SQL parsing. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8974–8983, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics.

[118] Kun Xu, Lingfei Wu, Zhiguo Wang, Yansong Feng, and Vadim Sheinin. Graph2seq: Graph to sequence learning with attention-based neural networks. *ArXiv*, abs/1804.00823, 2018.

[119] Kun Xu, Lingfei Wu, Zhiguo Wang, Yansong Feng, and Vadim Sheinin. SQL-to-text generation with graph-to-sequence model. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 931–936, Brussels, Belgium, October-November 2018. Association for Computational Linguistics.

[120] Xiaojun Xu, Chang Liu, and Dawn Song. Sqlnet: Generating structured queries from natural language without reinforcement learning, 2017.

[121] Kuan Xuan, Yongbo Wang, Yongliang Wang, Zujie Wen, and Yang Dong. Sead: End-to-end text-to-sql generation with schema-aware denoising, 2021.

[122] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. Sqlizer: Query synthesis from natural language. *PACMPL*, pages 63:1–63:26, 2017.

[123] Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation, 2017.

[124] Pengcheng Yin and Graham Neubig. TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 7–12, Brussels, Belgium, November 2018. Association for Computational Linguistics.

[125] Pengcheng Yin, Graham Neubig, Wen tau Yih, and Sebastian Riedel. Tabert: Pretraining for joint understanding of textual and tabular data, 2020.

[126] Deunsol Yoon, Dongbok Lee, and SangKeun Lee. Dynamic self-attention : Computing attention over words dynamically for sentence embedding, 2018.

[127] Tao Yu, Zifan Li, Zilin Zhang, Rui Zhang, and Dragomir Radev. Typesql: Knowledge-based type-aware neural text-to-sql generation, 2018.

[128] 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.

[129] Tao Yu, Michihiro Yasunaga, Kai Yang, Rui Zhang, Dongxu Wang, Zifan Li, and Dragomir Radev. Syntaxsqlnet: Syntax tree networks for complex and cross-domaintext-to-sql task, 2018.

[130] Tao Yu, Rui Zhang, He Yang Er, Suyi Li, Eric Xue, Bo Pang, Xi Victoria Lin, Yi Chern Tan, Tianze Shi, Zihan Li, Youxuan Jiang, Michihiro Yasunaga, Sungrok Shim, Tao Chen, Alexander Fabbri, Zifan Li, Luyao Chen, Yuwen Zhang, Shreya Dixit, Vincent Zhang, Caiming Xiong, Richard Socher, Walter S Lasecki, and Dragomir Radev. Cosql: A conversational text-to-sql challenge towards cross-domain natural language interfaces to databases, 2019.

[131] 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, 2019.

[132] Tao Yu, Rui Zhang, Michihiro Yasunaga, Yi Chern Tan, Xi Victoria Lin, Suyi Li, Heyang Er, Irene Li, Bo Pang, Tao Chen, Emily Ji, Shreya Dixit, David Proctor, Sungrok Shim, Jonathan Kraft, Vincent Zhang, Caiming Xiong, Richard Socher, and Dragomir Radev. Sparc: Cross-domain semantic parsing in context, 2019.

[133] John M. Zelle and Raymond J. Mooney. Learning to parse database queries using inductive logic programming. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence - Volume 2*, AAAI'96, page 1050–1055. AAAI Press, 1996.

[134] Zhong Zeng, Mong Li Lee, and Tok Wang Ling. Answering keyword queries involving aggregates and groupby on relational databases. *EDBT*, pages 161–172, 2016.

[135] Liang Zhao, Hexin Cao, and Yunsong Zhao. GP: Context-free grammar pre-training for text-to-sql parsers, 2021.

[136] Victor Zhong, Caiming Xiong, and Richard Socher. Seq2sql: Generating structured queries from natural language using reinforcement learning, 2017.