Text-to-SQL Parsing: Using Execution Plans as an Intermediate Language

Thesis submitted in partial fulfillment
of the requirements for the degree of
“DOCTOR OF PHILOSOPHY”

by

Ben Eyal

Submitted to the Senate of Ben-Gurion University
of the Negev

January 31, 2024

Beer-Sheva

This work was carried out under the supervision of Prof. Michael Elhadad

In the Department of Computer Science

Faculty of Natural Sciences

Acknowledgements

I would like to thank the people without whom this thesis would not have been possible because, in the end, it's not the thesis; it's the friends we made along the way.

First and foremost, I want to thank my advisor and friend, Prof. Michael Elhadad, whose guidance was a beacon of light in this journey called a Ph.D. Always calm, always caring, always there when it means the most, with the smile that says "Don't worry, you got this." I am glad I had the opportunity to teach alongside you and put the "fun" back into "functional programming", for every Zoom session or phone call, which I always enjoyed.

A huge "thank you" goes to Meni Adler and Yael Netzer, my eternal lab mates, and to Yaron Gonen, an honorary lab mate, who supported me at every step and listened to my ramblings. *Shnaim Ohazin* was (and still is) one of the best hours of the week.

Thank you to my research buddies, Moran Mahabi, Ophir Haroche, and Amir Bachar, who have seen so much QPL by now that they can write a query in the middle of the night. I wish them good luck in their Master's degrees and, hopefully, future Ph.D.s.

To my friends and co-workers at AI2 Israel: Yoav, Ron, Smadar, Micah, Hillel, Menny, Aryeh, Sigal, Mark, and Dan, thank you for being there for me through all the deadlines and the time I needed to see this journey through, and for letting me enjoy waking up every day for work.

To my parents, Carmi and Ori, who kept believing in me even when I lost belief in myself, pushing me to finish what I started, thank you for the kind and uplifting words when I needed them most. You're the best.

Last but not least, I want to thank my life partner and one true love, Gali, for staying with me through thick and thin, for long days and sleepless nights of writing, experiments, successes, breakdowns, and everything in between. I could not have done this without your support, words, and deeds.

Abstract

The Text-to-SQL task constitutes a significant domain within natural language processing (NLP) research and has garnered considerable scholarly attention due to its profound implications for human-computer interaction and database management. This task entails the conversion of natural language questions into structured SQL (Structured Query Language) queries, thereby facilitating seamless communication between non-technical users and relational databases. This task is important because databases are ubiquitous in modern applications, and text-to-SQL models democratize access to data repositories.

Beyond its practical importance, text-to-SQL is also a useful task theoretically as it allows probing the performance of NLP models in language understanding. The user of a text-to-SQL model can analyze the resulting SQL and verify the expected properties of the query given the question, such as the presence of certain schema items or joins, in addition to checking whether the denotation (the result of running the query) is correct.

In this work, we answer the following research questions: (1) how to make semantic parsing, specifically text-to-SQL, more compositional; (2) how to make the output of a semantic parser more interpretable to its users; (3) how to make a semantic parser more robust to linguistic variability in the input questions.

Our first contribution introduces Query Plan Language (QPL), a novel methodology in text-to-SQL parsing by employing execution plans as an intermediary language to test the hypothesis that a modular target language leads to more compositional learning. The implementation of QPL as an intermediate step offers an enhanced method for interpreting user queries and generating corresponding SQL commands, thereby improving the precision and efficiency of database query processing.

We also define the Spider-QPL dataset and experiment using the same neural architecture for two different tasks, text-to-SQL, and text-to-QPL, and observe that pre-trained large language models can be fine-tuned for better compositional generalization for text-to-QPL than text-to-SQL.

In our second contribution, we show that users with limited programming expertise find QPL more comprehensible than SQL, particularly in the context of complex queries. This characteristic of QPL is crucial as it facilitates broader user engagement with database systems. By making complex query structures more accessible to non-expert users, QPL enhances the usability of database interaction, allowing a diverse user base to perform advanced data retrieval tasks.

The prevalence of English as the de facto language employed in the design and structure of database schemas, where table and column names are typically expressed in English, presents a notable challenge for non-expert users who are not proficient in English. Our third contribution explores the domain of cross-lingual text-to-SQL: we explore the task of answering natural language questions in Hebrew, referring to a database whose schema and content are in English. Our experiments indicate that recent progress in large language models enables the construction of cross-lingual semantic parsers by exploiting automatic machine translation of the question. The variability in the vocabulary used to refer to database schema items in the questions after they have been translated indicates that more work is needed to address the problem of robust schema linking in current text-to-SQL models.

In a series of experiments, we verify that state-of-the-art architectures still exhibit low robustness in text-to-SQL and show that round-trip translation attacks on questions remain challenging.

Overall, this research advances our understanding of semantic parsing by stressing the importance of targeting modular formal languages with simple operational semantics to support compositional generalization. Our experimental results indicate that even when trained models reach very high performance (over 90%) on standard text-to-SQL benchmarks, we should keep low confidence that the predicted queries match the actual intent of the users because users have a limited understanding of complex SQL queries. Finally, cross-linguistic queries and round-trip translation attacks demonstrate the importance of improving ways models recognize linguistic variability, accounting for the many distinct creative ways people use to refer to the same content.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

1.1 Do Language Models Understand?

Natural Language Processing has witnessed a paradigm shift in recent years with the rise of large language models (LLMs) such as OpenAI's ChatGPT, Google's Bard, and Anthropic's Claude. These models have transformed our understanding of machine learning and natural language processing. Among the most pressing questions in this domain is whether these models truly "understand" language. On the one hand, in its most basic form, a language model is just a piece of code that samples a new word based on the words it produced previously, so it cannot possibly "understand". On the other hand, there's a reason why these LLMs have grown in popularity, as they do give some semblance of understanding when interacted with, despite "hallucinating" at times. The notion of understanding, however, is not a binary attribute but a spectrum of capabilities that can be measured in various ways.

One way to assess the capability of a language model is through task-oriented benchmarks, i.e., a model's "understanding" is evaluated based on a single domain and task. For example, a model can be measured by its success in classifying movie reviews' sentiments or answering 8th-grade math questions. These benchmarks are similar to unit tests in software development. We also want to verify that the model's behavior satisfies the expected properties of understanding (continuing the software development metaphor; this is similar to property-based testing), specifically two properties have been investigated: consistency testing (a model should not say p and $\neg p$) and *compositionality.*

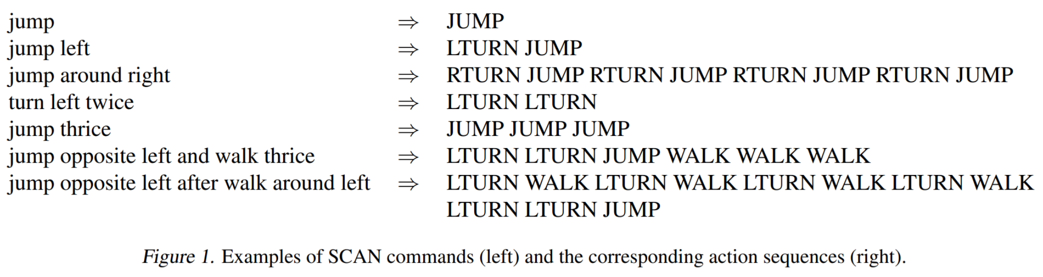
1.2 Compositionality

In semantics, the principle of compositionality [57] states that the meaning of a complex expression is determined by the meanings of its constituent expressions and the syntactic rules used to combine them. For our purposes, we say that if a model can interpret simple expressions, it should be able to interpret their syntactic combination.

Two main approaches to assessing a model's compositional generalization ability are specialized benchmarks and compositional splits.

Specialized benchmarks for compositional generalization, such as SCAN [38] and COGS [37], measure a model's capability to parse complex expressions. During training, the model is shown a mapping between natural language and a logical form up to a certain depth. The model is then expected to generalize to unseen expression depths solely based on the training data and structure. Examples of the SCAN and COGS benchmarks are shown in Figure 1.1 and Figure 1.2, respectively.

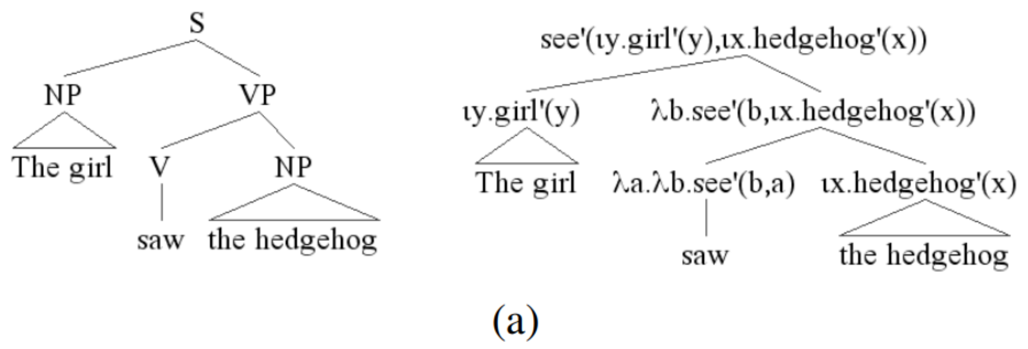
The second approach to measure compositional generalization is using compositional splits, as proposed by Keysers, Schärli, Scales, Buisman, Furrer, Kashubin, Momchev, Sinopalnikov, Stafiniak, Tihon, Tsarkov, Wang, van Zee, and Bousquet [36]. Using a general formula, a dataset can be engineered to be split into training and test sets with the property that the



jump	⇒	JUMP
jump left	⇒	LTURN JUMP
jump around right	⇒	RTURN JUMP RTURN JUMP RTURN JUMP RTURN JUMP
turn left twice	⇒	LTURN LTURN
jump thrice	⇒	JUMP JUMP JUMP
jump opposite left and walk thrice	⇒	LTURN LTURN JUMP WALK WALK WALK
jump opposite left after walk around left	⇒	LTURN WALK LTURN WALK LTURN WALK LTURN WALK LTURN LTURN JUMP

Figure 1. Examples of SCAN commands (left) and the corresponding action sequences (right).

Figure 1.1: Example from the SCAN Benchmark



TRAINING

[[The girl]] = $\iota x.\,girl'(x)$, [[The cat]] = $\boldsymbol{\iota x.\,cat'(x)}$, [[The boy]] = $\iota x.\,boy'(x)$

[[The cat loves the girl]] = $love'(\boldsymbol{\iota x.\,cat(x)}, \iota x.\,girl'(x))$

[[The hedgehog sees the cat]] = $see'(\iota x.\,hedgehog'(x), \boldsymbol{\iota x.\,cat'(x)})$

GENERALIZATION

[[The boy loves the hedgehog]] = $love'(\iota x.\,boy'(x), \boldsymbol{\iota x.\,hedgehog(x)})$

Figure 1.2: Example from the COGS Benchmark

Sentence	Atoms	Compounds
"Our vigilance is not partisan."	`nsubj, poss, our, vigilance, partisan`	`(vigilance, poss, our), (partisan, nsubj, vigilance)`
"We shall now hear Mr Wurtz speaking against this request."	`hear, aux, shall, speak, nsubj, wurtz, hear, ccomp, speak`	`(hear, aux, shall), (speak, nsubj, wurtz), (hear, ccomp, speak)`
"This seems to me to be a workable solution."	`solution, amod, workable, seem, xcomp, solution`	`(solution, amod, workable), (seem, xcomp, solution)`

Figure 1.3: Example of Compositional Split (Source: Moisio et al. [53])

simplest expressions (atoms) can be part of both sets, but complex expressions (compounds) are found in only one of the sets. In Figure 1.3, we see an example of a compositional split in the Machine Translation domain where the atoms are parts of a dependency graph, i.e., words and edge labels, and the compounds are the connections of the dependency graph.

1.3 Semantic Parsing

A complex and realistic task that allows evaluating a model's compositional generalization is Semantic Parsing (SP), the task of translating a natural language utterance to a logical form. A logical form can be executed to produce a denotation. Some examples of logical forms are λ-calculus, SQL, and Python.

A question that is relevant today more than ever is why we should "bother" with semantic parsing when question-answering (QA) is a uniform format to interact with an NLP system. While QA goes from a question to an answer, SP needs to go through an intermediate logical form before producing an answer. QA is also universal because it allows us to test a model's linguistic capabilities and world knowledge in any domain. It is also possible to "reduce" any task to QA. With that in mind, it would seem as if semantic parsing is redundant.

Despite the above desirable properties of QA as a task, semantic parsing is still hugely impactful in natural language processing, especially in light of ever-present “hallucinations” by LLMs. A first reason is that QA is challenging to assess, as there can be many questions with the same answer (variability) and many valid answers to a single question (ambiguity). This makes it hard to compare predicted answers to the expected ones. A second reason is that QA models learn **what** to answer while SP models learn **how** to answer, which allows a semantic parser to use skills that are hard to learn (e.g., arithmetic). In semantic parsing, we have known semantics for the logical form so that we can probe the semantic properties of the predicted logical form. In the text-to-code domain, we can execute the predicted and expected code snippets and compare the results. Seeing the predicted logical form gives the user a glimpse of what the model “understood”. In text-to-SQL for example, the availability of the query is a form of explanation of what was understood by the parser.

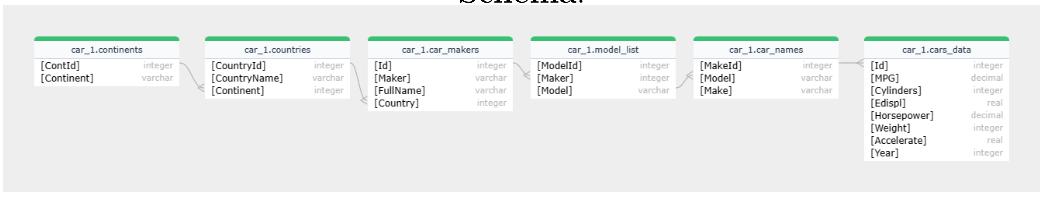
1.4 Text-to-SQL

Structured Query Language (SQL) is widely recognized as the standard language for database management and querying in relational database systems. Despite its ubiquity and utility in database management, SQL’s complexity can pose a significant challenge to non-expert users, particularly when constructing complex queries. SQL’s specificity and syntactic rigidity often necessitate a substantial understanding of database schema and query syntax, which can deter users without a technical background in computer science or database management.

In response to these challenges, the development of text-to-SQL models represents a significant area of research in natural language processing. These models are designed to facilitate database interaction by enabling users to input queries in natural language. The primary function of these models is to convert a user’s natural language query into a valid SQL query, which can then be executed against a database. This approach aims to simplify the process of database querying, making it more accessible to users who lack proficiency in SQL. The effectiveness of text-to-SQL models depends on

Question:
What is the number of cars with more than 4 cylinders?

Schema:



SQL:

```
SELECT COUNT(*)
FROM cars_data
WHERE cylinders > 4
```

Figure 1.4: Spider Example

their ability to accurately interpret natural language semantics and translate them into the structured format required by SQL.

The advancement of text-to-SQL technology has been accompanied by the development of various datasets, which are essential for training and evaluating the performance of these models. Among these datasets, the “Spider” dataset [88] has emerged as a notable resource in recent years. Characterized by its cross-domain scope, the Spider dataset encompasses various natural language questions, database schemas, and corresponding SQL queries. The diversity of the dataset ensures that models trained on it can adapt to multiple query types and database structures. Figure 1.4 shows an example from the development set of Spider.

1.5 Outline

In this thesis, we explore three research questions. Chapter 3 investigates how we assess and improve the compositional generalization of text-to-SQL models. Chapter 4 studies whether people benefit from code generated by a text-to-SQL system. Finally, Chapter 5 seeks to assess how robust models are to question formulation variations, stressing the importance of *schema linking* in a cross-lingual setting. We start in Chapter 2 with a review of previous work in the field of text-to-SQL.

CHAPTER 2

Previous Work

This chapter surveys the current state-of-the-art in the field of text-to-SQL. We start with a description of the most widely used datasets and benchmarks in the field with examples and then present the methods with the key ideas that brought progress on this task.

2.1 Datasets

Before the introduction of the Spider dataset mentioned above, research and evaluation in the text-to-SQL domain primarily utilized nine datasets, each focusing on distinct query domains and database structures.

The ATIS dataset [11], a pioneer in text-to-SQL conversion, was anchored in air travel queries from the Air Travel Information System, mapping these to SQL queries. Similarly, GeoQuery [89] encompassed questions about U.S. geography, initially linking utterances to logical forms before being adapted by Iyer, Konstas, Cheung, Krishnamurthy, and Zettlemoyer [35] to include

SQL pairings.

The Restaurants dataset [59, 70] provided a basis for queries about dining establishments, encompassing aspects like names, specialties, locations, and ratings. Scholar [35] and Academic [42] datasets concentrated on academic-related inquiries; the former targeting academic publications and the latter focusing on queries used on the Microsoft Academic Search website, encompassing a comprehensive range of query logics supported by the platform.

Yelp and IMDB datasets [83] served queries related to customer reviews and movie databases, respectively. WikiSQL [94], notable for its scale, was the largest among these datasets, containing a rich collection of natural language questions, SQL queries, and corresponding tables extracted from diverse Wikipedia tables.

Lastly, the Advising dataset [20] addressed queries about university course information, further diversifying the range of domains covered by text-to-SQL datasets prior to Spider's creation.

Recent datasets have increased the scale to more samples and more domains [39, 43]. In this work, we focus on the *Spider* dataset for our experiments, as it enables comparison with many previous methods.

Table 2.1 presents the size of each dataset, along with a sample question and its respective query.

Dataset	Sample Question	SQL Query	Size
ATIS	Show me airline abbreviations.	`SELECT DISTINCT airline_code FROM airline`	5,280
GeoQuery	What is the biggest city in Wyoming?	`SELECT city_name FROM city WHERE population = (SELECT MAX(population) FROM city WHERE state_name = 'Wyoming') AND state_name = 'Wyoming'`	877
Restaurants	How many "Buttercup Kitchen" are there in San Francisco?	`SELECT COUNT(*) FROM restaurant AS t1 JOIN LOCATION AS t2 ON t1.id = t2.restaurant_id WHERE t2.city_name = 'San Francisco' AND t1.name = 'Buttercup Kitchen'`	378
Scholar	Papers that were not published in the last year.	`SELECT DISTINCT paperid FROM paper WHERE YEAR <> 2015`	817
Academic	Return me the homepage of PVLDB.	`SELECT homepage FROM journal WHERE name = 'PVLDB'`	196
Yelp	List all the businesses with more than 4.5 stars.	`SELECT name FROM business WHERE rating > 4.5`	128
IMDB	What year is the movie "The Imitation Game" from?	`SELECT release_year FROM movie WHERE title = 'The Imitation Game'`	131
WikiSQL	What is the name of the license created by IETF?	`SELECT License FROM table WHERE Creator = 'ietf'`	80,654
Advising	Can undergrads take 550?	`SELECT DISTINCT ADVISORY_REQUIREMENT, ENFORCED_REQUIREMENT, NAME FROM COURSE WHERE DEPARTMENT = 'EECS' AND NUMBER = 550`	4,570

Table 2.1: Summary of Text-to-SQL Datasets

2.2 Methods

2.2.1 Rule-based

Even before the inception of SQL, we already see a need for a natural language interface to a database in the LUNAR system [80]. The primary objective of LUNAR was to allow scientists to retrieve information from a database using natural language. It aimed to interpret queries in English and translate them into a formal language that can be executed against a database.

LUNAR utilized a grammar-based approach to interpret user queries, wherein grammar rules were crafted to handle the specific syntax and semantics of the domain language. It used semantic primitives to represent the meanings of words and phrases, combining them based on the syntactic structure to construct a semantic representation of the user's query. LUNAR translated this semantic representation into a formal query language that could retrieve the relevant information from the database.

Although LUNAR did its task well, the methodology cannot be easily translated to other domains. The manual crafting of rules and the embedding of domain knowledge is labor-intensive, and handling the wide range of natural language variability and ambiguity poses significant challenges.

Following the development of LUNAR, CHAT-80 [75] emerged as a seminal system that advanced the concept of transforming natural language inquiries into formal queries to retrieve information from databases. CHAT-80 was designed to convert English language queries into PROLOG, thereby serving as an intuitive interface to a database consisting of world facts.

CHAT-80 significantly enhanced the flexibility of natural language processing for database inquiries by introducing a comprehensive syntactic and semantic analysis approach. This system was proficient in conducting a rigorous syntactic analysis to construct logical forms of input queries, which were then interpreted into PROLOG expressions.

Unlike LUNAR, CHAT-80 was recognized for its extensive grammatical framework, which allowed for a more inclusive and elaborate representation of natural language semantics. This framework facilitated the efficient handling of diverse syntactic constructions and complex semantic interpretations, enabling a wider coverage of natural language expressions and formulations.

However, similar to LUNAR, CHAT-80 also faced substantial challenges in scaling and adapting to varied domains and databases, primarily due to its reliance on meticulously designed grammar rules and semantic interpretations. The manual formulation of these components posed limitations in extending the system's capability to comprehend and interpret the vast and intricate spectrum of natural language expressions and constructs across different domains.

2.2.2 Machine Learning

The work by Zelle and Mooney [89] is a pivotal work in the area of natural language processing and represents a significant step in the development of Text-to-SQL systems. The main objective of the paper was to demonstrate the possibility of using inductive logic programming (ILP) to learn to parse natural language database queries. The approach is particularly important as it utilizes examples to learn the translation of natural language queries into a logical form that can be subsequently converted into SQL.

The work by Zettlemoyer and Collins [90] proposed a novel approach to semantic parsing that employs a probabilistic categorial grammar as the underlying formalism. Their method involves learning mappings from sentences to logical form representations, effectively transforming natural language into a formal language. The key innovation is the use of a structured classification approach, which enables the learning of complex mappings in a supervised learning setting.

Both contributed seminal work in the realm of text-to-SQL through machine learning methodologies. They share two significant shortcomings:

- **Domain Dependency**: Both models are heavily reliant on domain-specific data, making them less adaptable to other, especially new, domains without extensive, domain-specific annotated data.
- **Data Annotation**: The necessity for extensively annotated training data in both approaches is a substantial barrier, as creating such datasets is often expensive, laborious, and unscalable.

2.2.3 Neural Architectures

Dong and Lapata [15] were the first to use an attention-based sequence-to-sequence model, previously shown to excel at the machine translation task, for the text-to-SQL task. The method was tested on two single-domain datasets (GeoQuery and ATIS) and achieved ~84% exact match accuracy. However, the method fails to generalize to different domains, achieving only 4.8% exact match accuracy on the Spider dataset.

Bogin, Berant, and Gardner [4] presented a method based on Graph Neural Networks (GNNs) [46] which uses the connections, i.e., primary and foreign keys, between database schema items (columns and tables) as edges in the GNN, thus greatly enhancing the generalization capability of the model to unseen schemas. This approach raised the bar of text-to-SQL models evaluated on the Spider dataset, improving the previous state-of-the-art from 19.7% to 39.4% exact match accuracy.

Another notable work from the same time as the GNN method is due to Guo, Zhan, Gao, Xiao, Lou, Liu, and Zhang [30], who introduced IRNet. IRNet works in two phases:

1. **Schema Linking**: find phrases in the input utterance that relate to the database schema items.
2. **SemQL Synthesis**: the model outputs a more simple and regular language than SQL called SemQL, which is based on relational algebra [10]. This language is then translated automatically to SQL.

This approach improved on the GNN method above, going up to 46.7% exact match accuracy.

RAT-SQL [73] introduced a unified framework for addressing the challenges of schema encoding and linking in the context of text-to-SQL parsing. It leverages a relation-aware self-attention mechanism to effectively encode the relational structure of database schemas and align them with natural language queries. This approach significantly improves the parsing model's ability to handle unseen database schemas, a common challenge in semantic parsing tasks. The framework's effectiveness is demonstrated through its superior performance on the Spider dataset (57.2% exact match), achieving state-of-the-art results, particularly when augmented with BERT embeddings (65.6% exact match).

Central to the RAT-SQL framework are its schema linking techniques, including both name-based and value-based linking. Name-based linking focuses on matching column and table names in questions, while value-based linking aligns question mentions with database values. This dual approach allows for a more nuanced and accurate interpretation of natural language queries, facilitating their conversion into SQL queries. Additionally, the decoder in RAT-SQL plays a crucial role by generating SQL actions, such as expanding grammar rules and selecting appropriate schema elements, further enhancing the framework's query generation capabilities.

Gan, Chen, Xie, Purver, Woodward, Drake, and Zhang [26] introduced Natural SQL (NatSQL), a novel intermediate representation (IR) designed to simplify the translation of natural language descriptions into SQL queries. NatSQL addresses the inherent mismatch between natural language and the structured format of SQL, which has been a significant challenge in text-to-SQL conversion. It accomplishes this by simplifying SQL queries, dispensing with complex operators and keywords, and reducing the need for nested subqueries and a large number of schema items. This simplification makes it easier to generate executable SQL queries that closely align with natural language descriptions.

By incorporating NatSQL into existing neural network models, the paper demonstrates significant improvements in model performance. Particularly notable is the achievement of state-of-the-art execution accuracy on the Spider benchmark when NatSQL is used in conjunction with the RAT-SQL+GAP model, achieving 73.3% exact match accuracy. This suggests that optimizing the correspondence between natural language and query languages through specialized IRs like NatSQL is a promising direction for enhancing the capabilities of text-to-SQL models.

Furthermore, NatSQL specifically improves the generation of SQL keywords and schema items, areas where traditional text-to-SQL models often struggle. This improvement stems from NatSQL's design, which more effectively captures the intents expressed in natural language and translates them into the appropriate SQL structure. In essence, NatSQL serves as a bridge, reducing the gap between the expressive natural language queries and the technical specifics of SQL code, thus facilitating more accurate and efficient text-to-SQL translation.

2.2.4 Large Language Models (LLMs)

In 2017, Vaswani, Shazeer, Parmar, Uszkoreit, Jones, Gomez, Kaiser, and Polosukhin [72] presented the "Transformer" architecture, which initially has been used mainly for machine translation. However, due to T5 [61] in 2020, the text-to-SQL task has entered its age of LLMs. Shaw, Chang, Pasupat, and Toutanova [66] have shown that given an input that includes the natural language question together with basic schema information (table and column names), the largest T5 model (3B parameters) fine-tuned on the Spider training set achieves 70% exact match accuracy, competitive with the state-of-the-art at the time, 70.6%.

Aside from the above approach, all other methods use a custom architecture to achieve their results, meaning that large pre-trained language models cannot be used for the task. On the other hand, using only a fine-tuned pre-trained model to generate a SQL query can yield invalid SQL, either because it's malformed, i.e., not adhering to the SQL grammar, or has semantic problems, such as using a column not belonging to a certain table.

Enter PICARD [64], which presented a novel method for enhancing text-to-SQL translation by constraining the output of pre-trained language models. Large language models, while effective in various tasks, often struggle with generating valid SQL queries due to their unconstrained output space. PICARD addresses this by applying incremental parsing to constrain auto-regressive decoders, ensuring that only valid SQL tokens are produced at each decoding step. This method significantly improves the accuracy of models on Spider, transforming them into state-of-the-art solutions without the need for extensive modifications.

PICARD's design is both versatile and efficient. It is compatible with any auto-regressive language model decoder and does not require substantial changes to pre-existing model architectures or training processes. The incremental parsing technique operates directly on the model's output, in this case, SQL code, and is easily integrated at the inference stage. This flexibility allows PICARD to be applied broadly across different models and tasks.

The framework operates in multiple modes, each providing different levels of parsing and validation. The simplest mode, lexing, checks the output at a lexical level, ensuring the validity of SQL tokens regardless of their order. This mode is crucial for detecting errors like misspelled keywords or incorrect table and column names. More advanced modes involve parsing the model's output to an abstract syntax tree (AST), considering the grammatical structure of the SQL query, and enforcing constraints on query composition and structure.

In its most comprehensive mode, PICARD includes additional analytical processes known as guards. These guards impose strict constraints on the relationships between tables, columns, and aliases within the SQL queries. For instance, they ensure that referenced tables and columns are correctly brought into scope and prevent the misuse of aliases. This level of parsing significantly reduces the likelihood of generating syntactically incorrect or semantically nonsensical queries.

While the original method uses the SQLite grammar to work with the Spider dataset and the T5 family of models for their experiments, this method can be used with any fine-tuned pre-trained language model, and different grammars can be used, so long as one writes a parser. PICARD took the pre-trained T5-3B model from 70% exact match accuracy to 75.5% and 79.3% execution accuracy. Today, most methods that rely on a pre-trained language model use PICARD by default, as it is an "easy win": PICARD does not incur any time or memory costs during fine-tuning (as it is not used at that point) and works seamlessly with many HuggingFace Transformers [77] models.

With the emergence of GPT-3 [5] and GPT-4 [56], fine-tuned models were surpassed in the Spider leaderboard in terms of *execution* accuracy, rather than *exact match* accuracy.

Pourreza and Rafiei [60] introduced a method called DIN-SQL to enhance the performance of LLMs in text-to-SQL tasks. It addresses the gap between fine-tuned models and LLMs in handling complex text-to-SQL datasets like Spider. The approach involves decomposing text-to-SQL tasks into smaller sub-tasks, significantly improving the LLMs' performance. The paper demonstrates that this method enhances the LLMs' few-shot performance by approximately 10%, achieving new state-of-the-art execution accuracy on the Spider test set.

This approach uses few-shot prompting to decompose the text-to-SQL task into multiple steps. While zero-shot prompting provides a baseline, it often falls short, especially for medium and complex queries. The proposed method outperforms the few-shot prompting method by a large margin, using two versions of the CodeX model and GPT-4 for prompting. The method sets new grounds in execution accuracy (85.3%), surpassing many heavily fine-tuned models.

DIN-SQL comprises four modules: schema linking, query classification and decomposition, SQL generation, and self-correction. These modules are implemented using prompting techniques, showing that LLMs can solve the decomposed problems effectively. The self-correction module specifically addresses minor errors in the generated SQL queries, such as missing or redundant keywords, enhancing the overall accuracy and reliability of the models.

Gao, Wang, Li, Sun, Qian, Ding, and Zhou [28] highlights the absence of systematic benchmarks for developing LLM-based Text-to-SQL solutions and addresses this gap by conducting an extensive comparison of existing prompt engineering methods. This comparison includes various aspects such as question representation, example selection, and organization. Based on these findings, the authors propose an integrated solution, DAIL-SQL, which significantly improves performance on the Spider leaderboard (86.6% execution accuracy).

To summarize, the key ideas presented in the above works are mainly the importance of schema linking, the use of an intermediate language to bridge the gap between natural language questions and SQL, and the PICARD constrained decoding method, which guarantees valid output from a pre-trained language model. As both SemQL and NatSQL are intermediate languages that are based on the syntax of SQL rather than its semantics, in Chapter 3, we will explore the other direction: an intermediate language that is semantics-based. In Chapter 5, we will check the robustness of schema linking to perturbations in the question. Although we can see an upward trend in the execution accuracy of text-to-SQL models on the Spider dataset, careful inspection of their results shows that they succeed in generating simple queries and fall short when generating complex SQL, which is a current limitation in text-to-SQL models (even those based on LLMs and achieve more than 80% execution accuracy). This topic is discussed in-depth in Chapter 3.

CHAPTER 3

Query Decomposition for Compositional Generalization

3.1 Introduction

Querying and exploring complex relational data stores necessitate programming skills and domain-specific knowledge of the data. Text-to-SQL semantic parsing allows non-expert programmers to formulate questions in natural language, convert the questions into SQL, and inspect the execution results. While recent progress on this task has been remarkable, general cross-domain text-to-SQL models still face challenges on complex schemas and queries. State-of-the-art text-to-SQL models show performance above 90% for easy queries but fall to about 50% on complex ones (see Table 3.1). This accuracy drop is particularly bothersome for non-experts because they also find it difficult to verify whether a complex SQL query corresponds to the intent behind the question they asked. In a user study we performed, we found that software engineers who are not experts in SQL fail to determine whether a complex SQL query corresponds to a question in about 66% of the cases (see Chapter 4). The risk of text-to-code models producing incorrect results with confidence is thus acute: complex SQL queries non-aligned with users' intent will be hard to detect.

Question

```
What is the official language spoken in the country whose head of state
is Beatrix?
```

Gold SQL

```
SELECT T2.Language
FROM country AS T1
JOIN countrylanguage AS T2 ON T1.Code  =  T2.CountryCode
WHERE T1.HeadOfState  =  'Beatrix' AND T2.IsOfficial  =  'T'
```

Gold QPL

```
#1 = Scan Table [country] Predicate [HeadOfState = 'Beatrix']
     Output [Code, HeadOfState]
#2 = Scan Table [countrylanguage] Output [CountryCode, Language, IsOfficial]
#3 = Filter [#2] Predicate [IsOfficial = 'T'] Output [CountryCode, Language]
#4 = Join [#1, #3] Predicate [#3.CountryCode = #1.Code] Output [#3.Language]
```

Computed Question Decomposition (QD)

```
#1 = Scan the table country and retrieve the code and
     head of state of the country whose head of state is Beatrix
#2 = Scan the table countrylanguage and retrieve the country codes,
     languages and if they're official
#3 = Filter from #2 all the official languages and
     retrieve the country codes and languages
#4 = Join #1 and #3 based on the matching country codes and retrieve
     the language spoken in the country whose head of state is Beatrix
```

Predicted QDMR

```
#1 = return countries whose head of state is Beatrix ;
#2 = return the official language spoken in the official language of #1
```

Figure 3.1: Example QPL and Question Decomposition compared to the original SQL query from *Spider* and to the predicted QDMR question decomposition from [78].

This chapter addresses the challenge of dealing with complex data retrieval questions through a compositional approach. Based on the success of the question decomposition approach for multi-hop question answering, recent work in semantic parsing has also investigated ways to deal with complex SQL queries with a Question Decomposition (QD) strategy. In another direction, previous attempts have focused on splitting complex SQL queries into spans (e.g., aggregation operators, join criteria, column selection) and generating each span separately.

Our approach starts with a semantic analysis of the SQL query. We introduce a new intermediary language, which we call Query Plan Language (QPL), that is modular and decomposable. QPL can be directly executed on SQL databases through direct translation to modular SQL Common Table Expressions (CTEs). We design QPL to be both easier to learn with modern neural architectures than SQL and easier to interpret by non-experts. The overall approach is illustrated in Fig. 3.1. We develop an automatic translation method from SQL to QPL. On the basis of the modular QPL program, we also learn how to generate a natural language *decomposition* of the original question. In contrast to generic QD methods such as QDMR [78], our decomposition takes into account the database schema, which is referenced by the question, and the semantics of the QPL operations.

Previous research in semantic parsing has shown that the choice of the target language impacts a model's ability to learn to parse text into an accurate semantic representation. For instance, Guo, Liu, Lou, Li, Liu, Xie, and Liu [31] compared the performance of various architectures on three question-answering datasets with targets converted to Prolog, Lambda Calculus, FunQL, and SQL. They discovered that the same architectures produce lower accuracy (up to a 10% difference) when generating SQL, indicating that SQL is a challenging target language for neural models. The search for a target language that is easier to learn has been pursued in Text-to-SQL as well [26, 30, 87]. We can view QPL as another candidate intermediary language, which, in contrast to previous attempts, does not rely on a syntactic analysis of the SQL queries but rather on a semantic transformation into a simpler, more regular query language.

The rest of the chapter reviews recent work in text-to-SQL models that investigates intermediary representations and question decomposition. We then present the Query Plan Language (QPL) we have designed and the conversion procedure we have implemented to translate the existing large-scale *Spider* dataset into QPL. We then describe how we exploit the semantic transformation of SQL to QPL to derive a dataset of schema-dependent Question Decomposition. We finally present strategies that exploit the compositional nature of QPL to train models capable of predicting complex QPL query plans from natural language questions and to decompose questions into data-retrieval-oriented decompositions.

We investigate the following research question introduced in Chapter 1:
(RQ1) Is it easier to learn Text-to-QPL – a modular, decomposable query language – than to learn Text-to-SQL using Language Model based architectures;

Our main contributions are (1) the definition of the QPL language together with automatic translation from SQL to QPL and execution of QPL on standard SQL servers; (2) the construction of the Spider-QPL dataset which enriches the Spider samples with validated QPL programs for Spider's dataset together with Question Decompositions based on the QPL structure; (3) Text-to-QPL models to predict QPL from a (Schema + Question) input that are competitive with state of the art Text-to-SQL models and perform better on complex queries.[1]

3.2 Starting Points

Text-to-SQL parsing consists of mapping a question $Q = (x_1, \dots, x_n)$ and a database schema $S = [table_1(col_1^1 \dots col_{c_1}^1), \dots, table_T(col_1^T \dots col_{c_T}^T)]$ into a valid SQL query $Y = (y_1, \dots, y_q)$. Performance metrics include exact match (where the predicted query is compared to the expected one according to the

[1]All of the datasets and code are available on `https://github.com/bgunlp/qpl`.

Difficulty	NatSQL + RAT-SQL	Din-SQL GPT-4	GPT3.5-turbo
Easy	**91.6%**	91.1%	87.7%
Medium	75.2%	**79.8%**	75.1%
Hard	65.5%	64.9%	**72.5%**
Extra Hard	51.8%	43.4%	**53.9%**
Overall	73.7%	74.2%	**74.3%**

Table 3.1: *Spider* Development Set baseline execution accuracy by difficulty level

overall SQL structure and within each field token by token) and execution match (where the predicted query is executed on a database and results are compared).

3.2.1 Architectures for Text-to-SQL

Since the work of Dong and Lapata [15], leading text-to-SQL models have adopted attention-based sequence-to-sequence architectures, translating the question and schema into a well-formed SQL query. Pre-trained transformer models have improved performance as in many other NLP tasks, starting with BERT-based models [34, 47] and up to larger LLMs, such as T5 [61] in [64], OpenAI *CodeX* [7] and GPT variants [49, 60, 62].

In addition to pre-trained transformer models, several task-specific improvements have been introduced: the encoding of the schema can be improved through effective representation learning [4], and the attention mechanism of the sequence-to-sequence model can be fine-tuned [73]. On the decoding side, techniques that incorporate the syntactic structure of the SQL output have been proposed.

To make sure that models generate a sequence of tokens that obey SQL syntax, different approaches have been proposed: in Yin and Neubig [86], instead of generating a sequence of tokens, code-oriented models generate the abstract syntax tree (AST) of expressions of the target program. Scholak et al. [64] defined the *constrained decoding method* with PICARD. PICARD is an independent module on top of a text-to-text auto-regressive model that uses an incremental parser to constrain the generated output to adhere to the target SQL grammar. Not only does this eliminate almost entirely invalid SQL queries, but the parser is also schema-aware, thus reducing the number of semantically incorrect queries, e.g., selecting a non-existent column from a specific table. We have adopted constrained decoding in our approach by designing an incremental parser for QPL and enforcing the generation of syntactically valid plans.

3.2.2 Zero-shot and Few-shot LLM Methods

With recent LLM progress, the multi-task capabilities of LLMs have been tested on text-to-SQL. In zero-shot mode, a task-specific prompt is prefixed to a textual encoding of the schema and the question, and the LLM outputs an SQL query. Rajkumar et al. [62] and Liu, Hu, Wen, and Yu [48] showed that OpenAI Codex achieves 67% execution accuracy. In our own evaluation, GPT-4 (as of May 2023) achieves about 74% execution accuracy under the same zero-shot prompting conditions.

Few-shot LLM prompting strategies have also been investigated: example selection strategies are reviewed in Guo, Tian, Tang, Wang, Wen, Yang, and Wang [29] and Nan, Zhao, Zou, Ri, Tae, Zhang, Cohan, and Radev [54] and report about 85% execution accuracy when tested on the Spider development set or the Spider training set. Pourreza and Rafiei [60] and Liu and Tan [49] are top performers on *Spider* with the GPT4-based DIN-SQL. They use multi-step prompting strategies with query decomposition.

Few-shot LLM prompting methods close the gap and even outperform specialized Text-to-SQL models with about 85% execution match vs. 80% for 3B parameters specialized models on the *Spider* test set, without requiring any fine-tuning or training. In this paper, we focus on the hardest cases of queries, which remain challenging both in SQL and in QPL (with execution accuracy at about 60% in the best cases). We also note that OpenAI-based models are problematic as baselines since they cannot be reproduced reliably.[2]

3.2.3 Intermediary Target Representations

Most text-to-SQL systems suffer from a severe drop in performance for complex queries, as reported, for example, in DIN-SQL results where the drop in execution accuracy between simple queries and hard queries is from about 85% to 55% (see also Lee [40]). We demonstrate this drop in Table 3.1, which shows execution accuracy of leading baseline models [26, 60] per Spider difficulty level on the development set. The gpt-3.5-turbo results correspond to our own experiment using zero-shot prompting. Other methods have been used to demonstrate that current sequence-to-sequence methods suffer at *compositional generalization*, that is, systems trained on simple queries fail to generate complex queries, even though they know how to generate the components of the complex query. This weakness is diagnosed by using challenging *compositional splits* [27, 36, 66] over the training data.

One of the reasons for such failure to generalize to complex queries relates to the gap between the syntactic structure of natural language questions and the target SQL queries. This has motivated a thread of work attempting to generate simplified or more generalizable logical forms than executable SQL queries. These attempts are motivated by empirical results on other semantic parsing formalisms that showed that adequate syntax of the logical form can make learning more successful [31, 32].

Most notable attempts include SyntaxSQLNet [87], SemQL [30] and NatSQL [26].

NatSQL aims at reducing the gap between questions and queries. It introduces a simplified syntax for SQL from which the original SQL can be recovered. Figure 3.2 illustrates how this simplified syntax is aligned with the spans of the question.

[2]It is most likely that the Spider dataset was part of the training material processed by GPT models.

Question

```
What type of pet is the youngest animal,
and how much does it weigh?
```

SQL

```
SELECT PetType , Weight FROM Pets
ORDER BY Pet_Age LIMIT 1
```

Spider-SS Decomposition

SubSentence: `What type of pet`
NatSQL: `SELECT Pets.Pettype`
SubSentence: `is the youngest animal`
NatSQL: `ORDER BY Pets.Pet_Age LIMIT 1`
SubSentence: `and how much does it weigh?`
NatSQL: `SELECT Pets.Weight`

Figure 3.2: NatSQL and Question Decomposition in *Spider-SS* [27]

Our work is directly related to this thread. Our approach in designing *QPL* is different from NatSQL in that we do not follow SQL syntax nor attempt to mimic the syntax of natural language. Instead, we apply a semantic transformation on the SQL query and obtain a compositional regular query language where all the nodes are simple executable operators that feed into other nodes in a data-flow graph according to the execution plan of the SQL query. Our method does not aim to simplify the mapping of a single question to a whole query but instead to decompose a question into a tree of simpler questions, which can then be mapped to simple queries. The design of QPL vs. SQL adopts the same objectives as those defined in KoPL vs. SparQL in Cao, Shi, Pan, Nie, Xiang, Hou, Li, He, and Zhang [6] in the setting of QA over Knowledge Graphs.

3.2.4 Question Decomposition Approaches

Our approach is also inspired by work attempting to solve complex QA and semantic parsing using a *question decomposition strategy* [14, 23, 55, 58, 63, 79, 84, 93]. In this approach, the natural language question is decomposed into a chain of sub-steps, which has been popular in the context of Knowledge-Graph-based QA with multi-hop questions [52, 91]. Recent work attempts to decompose the questions into trees [33], which yields explainable answers [92].

In this approach, the question decomposer is sometimes learned in a joint manner to optimize the performance of an end-to-end model [85]; it can also be derived from a syntactic analysis of complex questions [13], or specialized pre-training of decompositions using distant supervision from comparable texts [95], or weak supervision from execution values [79]. LLMs have also been found effective as generic question decomposers in Chain of Thought (CoT) methods [8, 74, 76]. In this work, we compare our own Question Decomposition method with the QDMR model [79].

3.3 Query Plan Language (QPL) Definition and Dataset Conversion

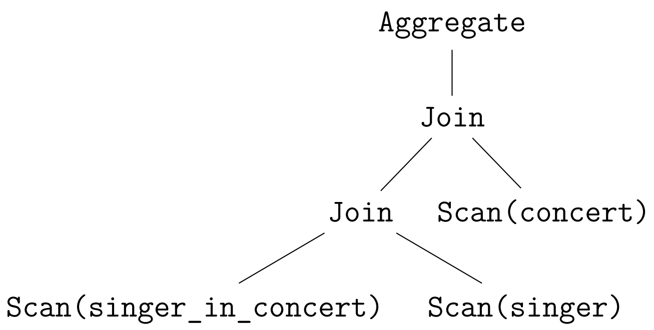
In order to investigate methods that analyze queries of different complexity and ways to decompose queries into sub-queries in a productive manner, we design a new intermediary language that we call *Query Plan Language* (QPL). QPL is intended to express the same queries as SQL; we also want QPL queries to be easy to decompose or recompose. Finally, we want QPL queries to be syntactically organized in a bottom-up manner so that queries are organized starting from simple steps, that we incrementally combine into more complex computations. Naturally, we want to be able to execute QPL queries and obtain the same result as their equivalent SQL queries.

We design QPL as a modular dataflow language that encodes the semantics of SQL queries. We take inspiration in our semantic transformation from SQL to QPL from the definition of the **execution plans** used internally by SQL optimizers, e.g., [65]. We automatically convert the original *Spider* dataset into a version that includes QPL expressions for all the training and development parts of *Spider*, and verify that in each case, the QPL query returns exactly the same result as the original SQL query. The detailed syntax of QPL is shown in Appendix B.

QPL abstract syntax is a hierarchical representation of execution plans. It is a tree of operations in which the leaves are *table reading* nodes (`Scan` nodes), and the inner nodes are either unary operations (such as `Aggregate` and `Filter`) or binary operations (such as `Join` and `Intersect`). Nodes have arguments, such as the table to scan in a `Scan` node or the join predicate of a `Join` node. For example, the SQL query:

```
SELECT Theme, AVG(Age)
FROM singer AS T1
    JOIN singer_in_concert AS T2
        ON T1.Singer_ID = T2.Singer_ID
    JOIN concert AS T3
        ON T2.concert_ID = T3.concert_ID
GROUP BY Theme
```

Results in the following execution tree (outputs and parameters omitted for brevity):



We write QPL's concrete syntax bottom-up, and from the above SQL query we get the following QPL expression:

```
#1 = Scan Table [ singer_in_concert ] Output [ concert_ID , Singer_ID ]
#2 = Scan Table [ singer ] Output [ Singer_ID , Age ]
#3 = Join [ #1 , #2 ] Predicate [ #1.Singer_ID = #2.Singer_ID ]
                      Output [ #2.Age , #1.concert_ID ]
#4 = Scan Table [ concert ] Output [ concert_ID , Theme ]
#5 = Join [ #3 , #4 ] Predicate [ #3.concert_ID = #4.concert_ID ]
                      Output [ #3.Age , #4.Theme ]
#6 = Aggregate [ #5 ] GroupBy [ Theme ]
                      Output [ AVG(Age) AS Avg_Age , Theme ]
```

An important distinction between QPL plans and SQL queries is that every QPL sub-plan is a valid executable operator, which returns a stream of data tuples. For example, Fig. 3.1 shows an execution plan with 4 steps and depth 2. The 4 steps are the two `Scan` leaves, the `Filter` sub-plan, and the `Join` sub-plan, which is the root of the overall plan.

We automatically convert SQL queries into semantically equivalent QPL plans by reusing the execution plans produced by Microsoft SQL Server 2019 query optimizer [22]. QPL is a high-level abstraction of the physical execution plan produced (which includes data and index statistics). In QPL syntax, we reduced the number of operators to the 10 operators listed in Table 3.2. We also design the operators to be *context free*, i.e., all operators take as input streams of tuples and output a stream of tuples, and the output of an operator only depends on its inputs.[3] We experiment with different syntactic realizations of QPL expressions and elected the version where steps are numbered and ordered bottom-up, corresponding roughly to the execution order of the steps. We validate that the result sets returned by the converted QPL plans are equivalent to those of the original Spider SQL queries. We thus enrich the Spider training and development sets with

[3]This is in contrast to SQL execution plan operators such as *Nested-Loops* where the two children nodes tightly depend on each other.

Operator	Description
Scan	Scan all rows in a table with optional filtering predicate
Aggregate	Aggregate a stream of tuples using a grouping criterion into a stream of groups
Filter	Remove tuples from a stream that do not match a predicate
Top	Select top-K tuples from a stream
Sort	Sort a stream according to a sorting expression
TopSort	Select the top-K tuples from a stream according to a sorting expression
Join	Perform a logical join operation between two streams based on a join condition
Except	Compute the set difference between two streams of tuples
Intersect	Compute the set intersection between two streams of tuples
Union	Compute the set union between two streams of tuples

Table 3.2: Description of QPL Operators

semantically equivalent QPL programs as shown in Fig. 3.3. Further details on the conversion from SQL to CTE are available in Appendix D.

3.4 Translating Questions into QPL

3.4.1 Text-to-QPL Model

In order to train a text-to-QPL model, we fine-tune Flan-T5-XL [9] (3B parameters) on 6,509 training examples. Each example contains a question, schema information, and the gold computed QPL. The input to the model is the same as in Shaw et al. [66], i.e., `Question | Schema Name | Table1 : Col11, Col12, ... | Table2 : Col21, Col22, ...` We also experiment with rich schema encoding, adding type, key, and value information as described below. We train the model for 15 full epochs and choose the model with the best execution accuracy on the development set. Execution accu-

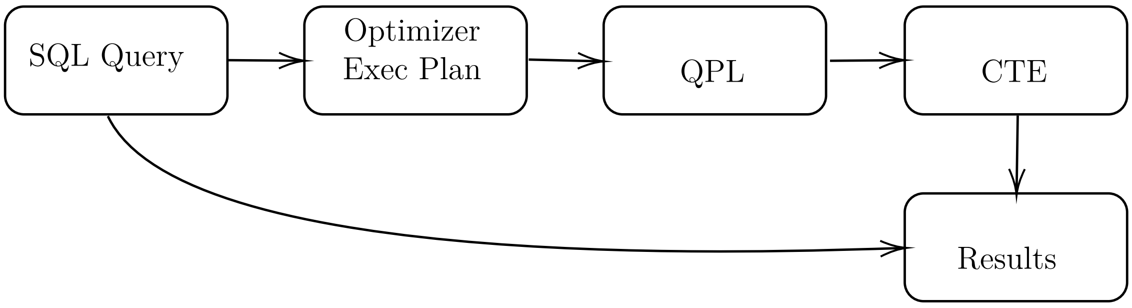


Figure 3.3: QPL generation process: the dataset SQL expressions are run through the query optimizer, which is then converted into QPL. QPL expressions are converted into modular CTE SQL programs, which can be executed. We verify that the execution results match those of the original SQL queries.

Original SQL

```
SELECT template_id, count(*)
FROM Documents
GROUP BY template_id
```

CTE

```
WITH
Scan_1 AS (
    SELECT Template_ID FROM Documents
),
Aggregate_2 AS (
    SELECT COUNT(*) AS Count, Template_ID
    FROM Scan_1
    GROUP BY Template_ID
)
SELECT * FROM Aggregate_2
```

Figure 3.4: SQL query and its equivalent CTE

racy is calculated by generating a QPL prediction, converting it to Common Table Expression format (CTE) (see example in Fig. 3.4), running the CTE in the database, and comparing the predicted result sets of the predicted and gold CTEs. The final evaluation of the model uses the PICARD [64] decoder with a parser we developed for QPL syntax. This constrained decoding method ensures that the generated QPL programs are syntactically valid. Details on this implementation are available in Appendix C.

Schema Encoding

We compare two methods to describe the database schema when we prompt our models:

1. Simple Schema Encoding: this is similar to Shaw et al. [66], *i.e.*, `Question | Schema Name | Table1 : Col11, Col12, ... | Table2 : Col21, Col22, ...`
2. Rich Schema Encoding: this encoding provides for each column a simplified type (as in Spider - text, number, date, or others); primary and foreign keys; and values.

For example:

Simple Schema Encoding: pets_1

```
Table Student (StuID, LName, Fname, Age, Sex, Major, Advisor, city_code)
Table Pets (PetID, PetType, pet_age, weight)
Table Has_Pet (StuID, PetID)
```

Rich Schema Encoding: pets_1

```
CREATE TABLE Student (
    StuID number,
    LName text,
    Fname text,
    Age number,
    Sex text,
    Major number,
    Advisor number,
    city_code text,
    primary key ( StuID )
)

CREATE TABLE Pets (
    PetID number,
    PetType text ( dog ),
    pet_age number,
    weight number,
    primary key ( PetID )
)

CREATE TABLE Has_Pet (
    StuID number,
    PetID number,
    foreign key ( StuID ) references Student ( StuID ),
    foreign key ( PetID ) references Pets ( PetID )
)
```

Values are added after each column when an n-gram from the question is found as one of the values in one of the rows of the table. For example, for the question *"How much does the youngest dog weigh?"*, the n-gram *"dog"* is found in the values of the column PetType. In this case, the value annotation `PetType text ( dog )` is encoded.

3.5 Question Decomposition

We use the QPL plans automatically computed from SQL queries in the dataset to derive a set of question decompositions (QD) that are grounded in the QPL steps, as shown in Fig. 3.1. We investigate three usages of this QD method: (1) $QPL \to QD$: we learn how to generate a QD given a QPL plan; this is useful at inference time, to present the predicted QPL to non-expert users, as a more readable form of the plan; (2) $Q \to QD$ we train a question decomposer on the Spider-QPL dataset for which we collect a set of validated automatically generated QDs; (3) $Q + QD \to QPL$ we finally investigate a Text-to-QPL predictor model which given a question, firstly generates a corresponding QD, and then predicts a QPL plan based on (Q+QD).

3.5.1 QPL to QD

We use the OpenAI `gpt-3.5-turbo-0301` model to generate a QD given a QPL plan. We prepared a few-shot prompt that includes a detailed description of the QPL language syntax and six examples that cover all the QPL operators that we prepared manually (see Appendix F).

We manually validated 50 pairs (QPL, QD) generated using this method and found them to be reliable, varied, and fluent. In addition, we designed an automatic metric to verify that the generated QDs are well aligned with the source QPL plan: (1) we verify that the number of steps in QD is the same as that in the source QPL; (2) we identify the leaf *Scan* instructions in the QD and verify that they are aligned with the corresponding QPL Scan operations. To this end, we use a fuzzy string matching method to identify the name of the table to be scanned in the QD instruction. The QPL-QD alignment score combines the distance between the length of QD and QPL and the IoU (intersection over union) measure of the set of Scan operations.

3.5.2 Dataset Preparation

Using the QPL $\rightarrow$ QD generator, we further enrich the Spider-QPL dataset with a computed QD field for each sample. For the sake of comparison, we also compute the predicted QDMR decomposition of the question [78] using the Question Decomposer from [79][4]

We obtain for each example a tuple: <Schema, Question, SQL, QPL, QD, QDMR>. We obtained 1,007 valid QDs (QPL-QD alignment score of 1.0) in the Spider Dev Set (out of 1,034) and 6,285 out of 6,509 in the Training Set with a valid QPL.

[4]We used the decomposer model published in `https://github.com/tomerwolgithub/question-decomposition-to-sql`.

3.5.3 Question Decomposer Model

Given the dataset of <Q, QD> obtained above, we train a QPL Question Decomposer, which learns to predict a QD in our format given a question and a schema description: $Q + Schema \rightarrow QD$. We fine-tune a Flan-T5-XL (3B parameters) model for this task, using the same schema encoding as for the $Q + Schema \rightarrow QPL$ model shown in §3.4.1.

Spider Difficulty	Q → QPL	Q+QD → QPL	Q → SQL	Support
Simple Schema				
Easy	87.5%	84.7%	91.9%	248
Medium	84.3%	72.2%	76.9%	446
Hard	66.7%	62.0%	64.9%	174
Extra Hard	54.8%	45.1%	44.6%	166
Overall	**77.4%**	69.1%	73.3%	1034
Rich Schema				
Easy	93.5%		91.5%	248
Medium	89.0%		88.3%	446
Hard	74.7%		69.5%	174
Extra Hard	65.1%		63.9%	166
Overall	**83.8%**		82.0%	1034

Table 3.3: Accuracy on Spider Development Set by difficulty level with *Simple Schema Encoding* (table names and column names) and *Rich Schema Encoding* (column types, keys, values).

3.5.4 Q+QD to QPL Prediction

We train a Flan-T5-XL (3B parameters) model under the same conditions as previous models on $\langle Q, QD, QPL \rangle$ to predict QPL given the QD computed by our question decomposer.

3.6 Experiments and Results

3.6.1 Text-to-QPL Prediction

We present our results on the Spider development set in Table 3.3. We compare our models to T5-3B with PICARD [64], as it is the closest model to ours in terms of number of parameters, architecture, and decoding strategy. To make the comparison as close as possible, we retrain a model <Q $\rightarrow$ SQL> using the same base model Flan-T5-XL (3B parameters) as we use for our <Q $\rightarrow$ QPL> model. We also compare two schema encoding methods: *Simple Schema Encoding* only provides the list of table names and column names for each table; *Rich Schema Encoding* provides for each column additional information: simplified type (same types as used in Spider's dataset - text, number, date, other), keys (primary and foreign keys) and values (see §3.4.1 for details). We see that at every difficulty level (except "Easy" for Simple Schema Encoding), our Q $\rightarrow$ QPL model improves on the baseline. The same is true compared to the other models in Table 3.1.[5] All other things being equal, this experiment indicates that it is easier to learn QPL as a target language than SQL **(RQ1)**.

On overall accuracy, the direct Q $\rightarrow$ QPL model achieves a respectable 77.4% without database content and 83.8% with database content. Our model notably achieves the highest execution accuracy on Hard and Extra-Hard queries across existing fine-tuned and LLM-based models (70.0% across Hard+Extra-Hard with database content).

The <Q + QD $\rightarrow$ QPL> model is inferior to the direct Q $\rightarrow$ QPL model (69.1% to 77.4% with simple schema encoding). We verified that this is due to the lower quality of the QD produced by our question decomposer. On Oracle QD, the model produces about 83% accuracy without database content. Table 3.4 confirms that the model accuracy level increases when

[5]The Q $\rightarrow$ SQL baseline we show reaches 82% vs. 79% reported in Scholak et al. [64] as we used a more complete schema encoding and the instruction fine-tuned Flan-T5-XL (3B parameters) model as opposed to the base T5-3B model.

the QD-QPL alignment score increases. This indicates that the development of a more robust question decomposer grounded in query decomposition for training signal has the potential to improve semantic parsing performance.

QD-QPL Alignment	Support	Correct	Exec Acc	Avg QPL Gold Len.
$[0.0, 0.4]$	1	0	0	5.0
$(0.4, 0.5]$	19	4	21.1	4.6
$(0.5, 0.6]$	9	2	22.2	6.3
$(0.6, 0.7]$	43	9	20.9	5.0
$(0.7, 0.8]$	63	21	33.3	4.2
$(0.8, 0.9]$	93	33	35.5	4.6
$(0.9, 1]$	779	624	80.1	2.8

Table 3.4: Q+QD $\rightarrow$ QPL model trained with QDs predicted by Trained Question Decomposer

In addition, we find that the <Q+QD $\rightarrow$ QPL> model produces correct answers that were not computed by the direct <Q $\rightarrow$ QPL> model in 50 cases (6 easy, 18 medium, 15 hard, 11 extra hard). This diversity is interesting because, for hard and extra-hard cases, execution accuracy remains low (55%-74%). Showing multiple candidates from different models may be a viable strategy to indicate the lack of confidence the model has on these queries.

3.7 Error Analysis

We manually analyzed errors on the QPL programs predicted by the best running model we have trained (see Table 3.3), which achieved an execution match of 83.8%. We analyzed all the cases where the automatic execution match procedure did not find an exact match between the result set returned by the predicted QPL and the gold QPL (which itself was found to be an exact match for the original Spider SQL statement).

In this manual analysis, we found a few cases where the result set was actually a correct execution match given the question. The reasons why these cases were not identified by the automatic result set comparison procedure are: (a) one of the queries included a distinct clause that prevented duplicate rows in the resultset while the other resultset did not; (b) the name of one of the computed columns was different across the two resultsets (e.g., the column corresponding to an aggregate like `max` was named in different ways); (c) the query made different assumptions about ties for aggregate values. For example, the question *"how many courses are taught by the youngest teacher"* assumes a single teacher is ranked as the youngest. When age is recorded in years, the query can find multiple teachers that are ranked as "youngest." In most cases of this type, the Spider query returns a single row even though ties can be found. In our assessment, we assumed this type of query is ambiguous and accepted either the single row or multiple rows resultsets as correct. The result of this manual classification is listed in the GitHub repository associated with this work `https://github.com/bgunlp/qpl`.

For each case that we classified as an error, we manually classified the error by first identifying the first line in the QPL program, which we identified as a mistake. We then classified the reason why this line is wrong. This analysis identified that the most challenging error type relates to queries involving aggregate operations (*group by* and aggregated values such as `count`, `sum` or `max`).

Table 3.5 shows the breakdown of errors by error types for the Q $\rightarrow$ QPL model with Rich Schema Encoding. We hypothesize that errors related to Join operations could be reduced by exploiting a more expressive description of the schema structure and applying either a post-processing critique of the generated QPL or enforcing stricter constraints in the QPL PICARD parser. Similarly, a more detailed description of the content of encoded columns could improve the *Wrong Constant* type.

Error Type	**Error**	**Err.%**	**Explanation**
Wrong aggregate	35	21%	Error in Aggregate (sum, avg, count, max, min)
Join	31	19%	Wrong join (e.g., not on Primary/Foreign key)
Wrong column	17	12%	Output does not include the right columns
Missing filter	17	10%	Filter stage is missing
Wrong constant	15	9%	Compare with wrong constant (e.g., IsOfficial = 'Y' vs. 'T')
Wrong predicate	12	7%	Error in selection predicate (e.g., > instead of <)
Lost	12	7%	Predicted QPL is completely wrong
Typing	7	4%	Compare with constant of wrong type (e.g., age = 'old' vs. 20)
Extremum	4	2%	Error in selecting top value (min vs. max, for example)
Intersect	4	2%	Error in Intersect operation
Wrong structure	3	2%	QPL plan is not a connected tree
Syntax issue	3	2%	Predicted QPL is not syntactically valid
Except	2	1%	Error in Except operation
Distinct	1	1%	Missing distinct flag
Wrong table	1	1%	Refers to the wrong table in the schema
Grand Total	167		

Table 3.5: Error Types: Breakdown of errors by error types

The Spider development set includes 20 different schemas. Table 3.6 shows the breakdown of errors per schema. We observe that 5 schemas account for over 70% of the errors. These schemas do not follow best practices in data modeling: keys are not declared, column naming is inconsistent, and value encoding in some columns is non-standard (e.g., use of 'T'/'F' for boolean values). This finding again indicates that more expressive encoding of schema information could improve performance on some of the more challenging samples.

Schema ID	**Errors**	**Samples**	**Error Rate**
battle_death	1	16	6%
car_1	**27**	92	29%
concert_singer	5	45	11%
course_teach	0	30	0%
cre_Doc_Template_Mgt	8	84	10%
dog_kennels	12	82	15%
employee_hire_evaluation	1	38	3%
flight_2	**15**	80	19%
museum_visit	2	18	11%
network_1	9	56	16%
orchestra	0	40	0%
pets_1	2	42	5%
poker_player	0	40	0%
real_estate_properties	1	4	25%
singer	1	30	3%
student_transcripts_tracking	**16**	78	21%
tvshow	5	62	8%
voter_1	3	15	20%
world_1	**44**	120	37%
wta_1	**15**	62	24%
Grand Total	**167**	**1034**	**16%**

Table 3.6: Breakdown of errors by Schema ID: 5 schemas out of the 20 present in Spider's development set account for 70% of the errors. These schemas do not follow best practices in data modeling and lack proper foreign key declarations.

3.8 Conclusion

We presented a method to improve compositional learning of complex text-to-SQL models based on QPL, a new executable and modular intermediary language that is derived from SQL through semantic transformation. We provide software tools to automatically translate SQL queries into QPL and to execute QPL plans by translating them to CTE SQL statements. We also compiled Spider-QPL, a version of Spider that includes QPL and Question Decomposition for all examples.

Our experiments indicate that QPL is **easier to learn** using fine-tuned LLMs (our text-to-QPL model achieves SOTA results on fine-tuned models on Spider dev set without db values, especially on hard and extra-hard queries). On the basis of the computed QPL plans, we derived a new form of Question Decomposition and trained a question decomposer that is sensitive to the target database schema, in contrast to existing generic question decomposers. Given a predicted QPL plan, we can derive a readable QD that increases the interpretability of the predicted plan.

In future work, we plan to exploit the modularity of QPL plans for data augmentation and explore multi-step inference techniques. We have started experimenting with an auto-regressive model that predicts QPL plans line by line. Our error analysis indicates that enhancing the model to further take advantage of the database values and foreign keys has the potential to increase the robustness of this multi-step approach. We are also exploring whether users can provide interactive feedback on predicted QD as a way to guide QPL prediction.

CHAPTER 4

SQL vs. QPL Interpretability

One of the key differences between semantic parsing and QA is that semantic parsing produces an explicit executable logical form, which can be analyzed and executed. The execution returns the eventual answer to the user's question. In QA, the results are directly computed by the QA model. This difference has an implication on the interpretability of the answers produced by the models: one can analyze the code produced by a semantic parser (in our case, the SQL or QPL query) and point to the places where it is different from the user intent in the question. One can also demonstrate the difference by executing the same query on databases containing different contents to highlight the difference in meaning between the user intent and the predicted query.

Using such techniques, it is possible to explain why a predicted query differs from the desired one, sometimes in a subtle manner. For example, consider the question: "find the name of the employee who was awarded the most times in the best-employee award." The question involves counting the number of times each employee received the award, sorting the list of employees based on this number, and then selecting the name of the employee that has the highest number of awards. When there are ties (multiple employees got the award the same number of times, for example, John and Mary), the answer should list multiple names. An answer with a single name would probably also be acceptable if it is augmented with an explanation (i.e., “John received the award 5 times, Mary did as well”). If a predicted query to this question returns “Mary” and the gold answer lists “John,” then the execution match procedure will fail without noting that both answers are equivalent.

More generally, when we analyze and compare queries written in a formal language (either SQL or QPL in our case), we can pinpoint differences that are easier to explain than when comparing the computed result set denotations. In this chapter, we present a user experiment that aims to assess whether programmers find it easier to perform such query comparisons on QPL than on SQL.

In order to probe whether QPL is easier to interpret by non-expert SQL users, we organized a user experiment. We selected a sample of 22 queries of complexity Hard and Extra-Hard. We collected predicted SQL queries and QPL plans for the queries, with half correct (producing the expected output) and half incorrect. We sampled the incorrect queries from the predictions of our text-to-QPL model so that they would be realistic queries but still incorrect queries. In other words, this is an adversarial contrastive test where the incorrect queries look very similar to the intended queries and are complex queries.

Four volunteer software engineers with over five years of experience participated in the experiment. We asked them to determine whether a query in either SQL or QPL corresponded to the intent of the natural language question. The participants were each exposed to half cases in QPL and half in SQL, half correct and half incorrect. We measured the time it took for each participant to make a decision for each case. Results are reported in Table 4.1.

The participants all know SQL (and use it in their work) but have never heard of QPL (which was introduced in this work and to which they are not related). We introduced the participants to QPL through a series of examples of increasing complexity and with a short explanation (see Appendix E).

Type	**Gold label**	**Time**	**Correct**
QPL	Incorrect query	89%	53%
	Correct query	100%	79%
			67%
SQL	Incorrect query	102%	50%
	Correct query	101%	25%
			34%
Avg time	QPL	123s	
	SQL	132s	

Table 4.1: User experiment: 20 (question, query) pairs are shown to 4 users - half in QPL and half in SQL. Half are correct, and half are incorrect. The table reports how long users assessed, on average, each query and how often they were right in assessing the correctness of the query.

The results in Table 4.1 indicate that the participants were correct about QPL in 67% of the cases vs. 34% of the SQL cases, supporting the hypothesis that validating the alignment between a question and query is easier with QPL than with SQL ($p < 0.06$).

We did not observe significant variation in the time it took for participants to assess QPL vs. SQL queries (on average, 132 seconds for SQL vs. 123 seconds for QPL) and no difference in time for prediction of correct vs. incorrect queries and correct vs. incorrect judgment.

Two significant results can be observed:

- The judgment by programmers on SQL queries is extremely low: in only one-third of the cases, the programmers correctly assess that an SQL query matches the intent of a natural language question. Consider that the best available models for text-to-SQL reach about 90%

accuracy on all queries but only about 75% on complex queries. Combining these two factors, it is clear that text-to-SQL models should *not* be put in production until serious work is performed on explainability and verification of the predicted queries.

There are well-known reasons why complex SQL queries are hard to interpret for programmers. The most significant one is that the syntactic order of the components within SQL queries does not match the logical order. For example, SELECT is (usually) the last operation that happens on the result, yet it is the first token in an SQL query. The order of GROUP BY aggregations is confusing with respect to the order of other operations, i.e., whether an aggregation has already happened or not is not always clear syntactically. The blog post "A Beginner's Guide to the True Order of SQL Operations" by Lukas Eder[1] identifies many complexities associated to the interplay between different SQL syntactic devices, in particular, group-by, order-by, distinct and computed expressions.

This accidental complexity is avoided in QPL because the different stages of the computation are split into separate QPL sentences (aggregation, ordering, filtering, join).

- Programmers performed better at aligning user intent of questions with QPL queries even though they have not learned or used QPL earlier. This desirable benefit is a result of the design decision to align QPL with SQL for primitive operations. Yet, further research is required to improve the overall usability of QPL by programmers by studying how programmers actually produce QPL programs or edit them to fix faulty queries.

In conclusion, this small-scale user experiment points to the fragility of existing text-to-SQL models in terms of interpretability and explainability. QPL improves the situation on this front, but the results remain low (only two-thirds of the queries are correctly assessed by programmers). In the longer

[1]See `https://blog.jooq.org/a-beginners-guide-to-the-true-order-of-sql-operations/`

term, these results point to the need to help programmers analyze complex data retrieval needs. We suspect that the natural language questions are also not understood in a uniform manner by different programmers and that many forms of ambiguity (similar to the tie case discussed above or to the operational interpretation of complex SQL schemas in terms of cardinality or key relations) hide additional complexity when interpreting complex questions. Future work should focus on the explainability of text-to-SQL to a larger extent.

Below is an example of a question, schema, and SQL query triplet in which the SQL query is correct given the question and schema. All participants have labeled this triplet as wrong, i.e., the SQL query does not match the question and schema.

```
Schema: world_1
city : ID , Name , CountryCode , District , Population
sqlite_sequence : name , seq
country : Code , Name , Continent , Region , SurfaceArea , IndepYear ,
          Population , LifeExpectancy , GNP , GNPOld , LocalName ,
          GovernmentForm , HeadOfState , Capital , Code2
countrylanguage : CountryCode , Language , IsOfficial , Percentage

Question:
What are the countries that have greater
surface area than any country in Europe?

SQL Query:
SELECT Name FROM country
WHERE SurfaceArea > (
  SELECT MIN(SurfaceArea) AS Min_SurfaceArea
  FROM country
  WHERE Continent = 'Europe'
)
```

The equivalent QPL to this query is:

```
#1 = Scan Table [ country ] Output [ SurfaceArea , Continent ]
#2 = Filter [ #1 ] Predicate [ Continent = 'Europe' ]
     Output [ SurfaceArea ]
#3 = Aggregate [ #2 ] Output [ MIN(SurfaceArea) AS Min_SurfaceArea ]
#4 = Scan Table [ country ] Output [ Name , SurfaceArea ]
#5 = Join [ #3 , #4 ] Predicate [ #4.SurfaceArea > #3.Min_SurfaceArea ]
     Output [ #4.Name ]
```

CHAPTER 5

Cross-Lingual Semantic Parsing and Schema Linking Robustness

5.1 Motivation

When analyzing the types of errors performed by semantic parsers, one can distinguish two sources of complexity: (a) mapping elements in the question to specific schema items in the target query (b) building a query with structure and semantics appropriate to the intent of the question. These two aspects correspond to the lexical vs. syntax-semantics aspects of the task as understood in machine translation. In the case of Text-to-SQL, these two aspects are usually described as *schema linking* and *structure mapping*.

The previous chapters focused on the *structure mapping* side of the task: the design of QPL aimed at improving compositional generalization to queries of increasing structural complexity.

This chapter focuses on the *schema linking* aspect. Schema linking consists of recognizing references to specific schema items (table and column names) and database values within the natural language question. For example, consider the question in Fig. 5.1 and the corresponding SQL query. Schema linking consists of identifying that the question word *country* refers to the column schema item `Airlines.country`, the word Airline to the table name `Airlines`, and the span *"JetBlue Airways"* to a value.

Cross-lingual semantic parsing consists of asking a question in a language that is different from the one used in the query language or in the database values. For example, given the same database, consider the case where the same question is asked in Hebrew. In this case, the lexical distance between the words in the question and the schema items or database values is increased, making the task of schema linking much more challenging.

In this chapter, we explore the impact of the cross-lingual setting on the performance of the Text-to-QPL task. We compare different strategies to cope with cross-lingual questions: training a Hebrew-to-SQL model on small quantities of manually translated questions or using machine translation in different configurations at fine-tuning or inference time. We finally empirically evaluate the robustness of the models to perturbations in the question formulation induced by roundtrip translation.

English Question
Which country does Airline "JetBlue Airways" belong to?
Which country[`Airlines.country`] *does Airline*[`Airlines`] *"JetBlue Airways"*[`value`] *belong to?*

Hebrew Question
לאיזו מדינה שייכת חברת התעופה ג'ט בלו איירווייז?
le-eizo medina shayekhet hevrat ha-te'ufa jet blu airways?
To-which nation belongs company the-flight jet blu airways?
To-which nation[`Airlines.country`] *belongs* [company the-flight][`Airlines`] [*jet blu airways*][`value`]?

SQL

```
SELECT Country
FROM Airlines
WHERE Airline = "JetBlue Airways"
```

Figure 5.1: Schema Linking Example

Most semantic parsing research operates over English datasets. Cross-lingual semantic parsing refers to the task of translating a question in source language L_1 to a target semantic formalism, which is based on a different language L_2.

In the specific case of cross-lingual text-to-SQL, three components can be in different languages: the question, the schema, and the values stored in the database. In cross-lingual settings, we will see the question in the source language of interest, but when it comes to the schema and SQL, there are three approaches: one option is to translate everything, having all schema items and SQL values be in the target language; another approach is to keep the schema intact, but have SQL values be in the target language; finally, we can keep both schema and SQL completely untouched, thus having only the question in a language other than English, with schema and SQL values kept in English. These options are summarized in Table 5.1.

5.2 Starting Points

Dataset	**Question**	**Schema Items**	**DB Values**
Spider	English	English	English
CSpider	Chinese	English	Chinese
VSpider	Vietnamese	Vietnamese	Vietnamese
MultiSpider	EN, DE, FR, ES, JA, ZH, VI	Same	Same
HSpider	Hebrew	English	English

Table 5.1: Cross-lingual Text-to-SQL Variants

In this chapter, we operate under the third setting, the most challenging in terms of *cross-lingual schema linking*, and we run experiments with Hebrew, which also poses acute problems for value recognition because Hebrew uses a different script than English and proper names must be transliterated using Hebrew letters, which introduces wide variability. We explore three main questions: (1) can we learn a cross-lingual semantic parser with limited resources? (2) in a very low resource setting, are manually translated examples better for training than machine-translated examples? (3) how robust is schema linking to round-trip translation noise?

5.2.1 Cross and Multilingual Datasets

Shortly after the release of the Spider dataset [88], Chinese and Vietnamese versions of Spider (CSpider [51] and VSpider [71]) were created. CSpider was manually translated and is roughly the same size as the original Spider. An important choice made during the translation of CSpider was only to translate questions and SQL values, as English schemas with Chinese contents are prevalent in the industry. An example from CSpider looks like:

`SELECT Country FROM AIRLINES WHERE Airline = " 深圳航空公司"`.

VSpider is also completely manually translated, and all questions and schema items are translated into Vietnamese. An example from VSpider looks like:

`select count ( * ) from ca sĩ`.

Recently, the MultiSpider dataset [16] introduced versions of Spider translated into six additional languages: German, French, Spanish, Japanese, Chinese, and Vietnamese. The data was obtained by first using automatic machine translation and manual post-editing. The MultiSpider dataset contains translated schema items and database values. In that sense, MultiSpider is a multi-lingual dataset but not a cross-lingual one (leading to less acute complexity in schema linking).

5.2.2 Schema-linking Robustness Datasets

In addition to the two cross-lingual Spider-derived datasets, three new datasets were designed to test the robustness of text-to-SQL models to question perturbation: Spider-Syn [24], Spider-DK [25], and Spider-Realistic [12].

Spider-Syn provides both a training set and a development set with SQL queries identical to the original Spider, where the questions use synonyms for schema items. For example, the original Spider question *"How many singers do we have?"* is transformed into *"How many vocalists do we have?"*.

Spider-DK provides only a development set with new database schemas and questions requiring domain knowledge. For example, the utterance "Show name, country for all singers ordered by age from the oldest to the youngest." is mapped to the query `SELECT name , country FROM singer ORDER BY birthday ASC` and a text-to-SQL will need to know that `birthday` is related to age.

Based on the observation that many original Spider questions are biased and often use schema items in a non-natural manner, Spider-Realistic manually modifies the original Spider development set questions to remove explicit mention of column names while keeping the SQL queries unchanged.

All three papers observed sharp drops in performance when running existing Spider models on the perturbed datasets. These experiments, however, date from 2020-2021 and predate more robust models built on top of LLMs. We rerun some of these experiments in this Chapter to update these observations.

The papers also experimented with mitigation methods to overcome the performance drop induced by question perturbation. Spider-Syn proposed two approaches: Multi-Annotation Selection (MAS) and adversarial training. In Multi-Annotation Selection, the schemas are annotated with synonyms for each schema item. During inference, if the question contains a word or phrase that is a known synonym, that synonym will be replaced in the question before being input into the model. The main advantage of this approach is that it does not require further training. MAS can be done either manually or automatically, and Gan et al. [24] show that manual MAS leads to better performance. The other approach, adversarial training, uses BERT-Attack [45] to create adversarial examples and then train a model on the adversarial examples. In their experiments, Gan et al. [24] show that training on adversarial examples is not as beneficial as manual MAS and is comparable to training the model on the Spider-Syn dataset.

5.2.3 Models for Cross-lingual Semantic Parsing

The XRICL framework [69] uses in-context learning (ICL) [5] for the cross-lingual Text-to-SQL task. In-context learning is a prompting technique for LLMs whereby the user provides the LLM via a prompt with a list of question-and-answer pairs and finishes the prompt with a question of the same nature as the examples. Given the context in the prompt, the completion of the LLM should then answer the question. XRICL first uses mT5 [82], a multi-lingual pre-trained language model, and fine-tunes it to select N English question-and-SQL pairs to be used for ICL given a non-English question. The N examples are then reranked, and the top K examples are selected for the prompt. To these top K examples, XRICL adds a translation example as a Chain-of-Thought [76] and the final question. The LLM should then translate the question in the target language to English and produce the correct SQL using the K examples. In addition to the XRICL framework, the paper also translates Spider into Farsi and Hindi, along with using CSpider and VSpider for further evaluation. Experimental results indicate the ICL strategy improves over zero-shot baselines for cross-lingual results, but the gap in performance with mono-lingual (English) results is notable (more than 20% delta for cross-lingual questions).

Sherborne and Lapata [67] propose the "Cross-Lingual Generalization Reptile" (XG-Reptile) meta-learning algorithm for cross-lingual semantic parsing. The main goal of this algorithm is to optimize for cross-lingual generalization using less training data per new language without hurting the performance of other languages. The approach demonstrates the potential to indeed reduce the size of the training set for adding new languages to an existing semantic parser by generalizing from one language to another, but absolute performance on the low-resource language remains low. In our experiments, we find that using automatic language translation achieves higher performance on the text-to-SQL task than the low-resource adaptation approach.

The "Representation Mixup Framework" (REX) [68] is introduced to bridge the gap between, on the one hand, using automatic translation to translate a target language to English and then using an English text-to-SQL model and, on the other hand, training a model "from scratch" on manually translated training data. It uses a general encoding layer, a transition layer, and a target-centric layer to best use English translations in the model. For every target language, the model is trained by first initializing the weights with the weights of an English text-to-SQL model, then training the model using manually annotated data in the target language, in addition to English translations, which can be used by the pre-trained English model. The framework achieves competitive results with state-of-the-art on both CSpider and VSpider. Our experiments on Hebrew with HSpider confirm that the combination of low-resource training and automatic translation brings good performance.

5.3 Methods

We investigate the overall question of robustness to question variability in text-to-SQL through two sets of experiments: (a) we assess the performance of different strategies on cross-lingual text-to-SQL and text-to-QPL on a Hebrew Spider translation dataset which we constructed; (b) we measure the robustness of the best-trained models on text-to-QPL to perturbations obtained by roundtrip translation (EN $\rightarrow$ HE $\rightarrow$ EN).

5.3.1 Cross-lingual Text-to-QPL

Low-resource Training

To answer whether we can learn a cross-lingual semantic parser with few resources, we use mT5-XL [82], a multilingual, 3 billion-parameter model that was trained, among other languages, on both English and Hebrew. To obtain Hebrew training examples parallel to our Spider-QPL dataset, we manually translate a small subset of 500 examples and also automatically translate the entire Spider training dataset. The 500 examples were chosen randomly from the training set, keeping the same ratio of easy/medium/hard/extra-hard examples as in the full training set. The translation was done by myself and my advisor. We fine-tune the model on both languages, varying the number of training examples between every training. In total, we fine-tune nine different mT5 models: using 500, 1,000, 3,000, and ~6,500 training examples on both English and Hebrew and another model trained on the 500 manually translated examples. We then compare the execution accuracy between those nine models. The models are summarized in Fig. 5.2.

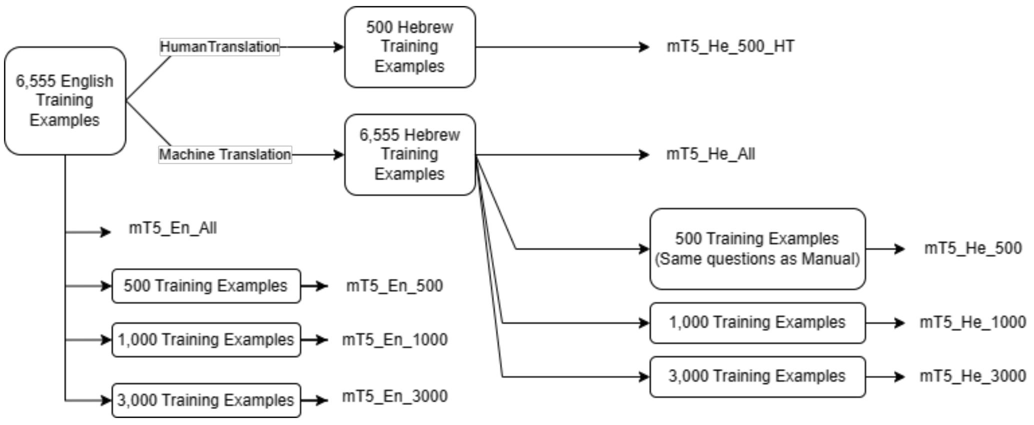


Figure 5.2: mT5 Fine-tuned Models

Inference Time Translation

In order to evaluate model performance given Hebrew questions and English schema items and values, we use three methods: (1) use an mT5-XL-based model fine-tuned directly on Hebrew questions; (2) translate the Hebrew question to English using GPT-4 [56] and the prompt shown in Fig. 5.3, which aims at reducing the distance of the English question from the terms used in the database schema, and use the best English $Q \to QPL$ model (cost of translation was \$0.69 in January 2024); (3) translate the Hebrew question using Google Translate [81], i.e., without a constraining prompt, and use the best English $Q \to QPL$ model. A graphical summary of these methods is shown in Fig. 5.4.

```
I need to translate questions asked in Hebrew into English
about a specific SQL database schema where the tables and columns
are specified in English.

Make sure to use as often as possible English words that appear
within the SQL schema.

### SQL Schema

{sql_schema}

### Hebrew Question

{hebrew_question}

### English Question
```

Figure 5.3: GPT-4 Translation Prompt

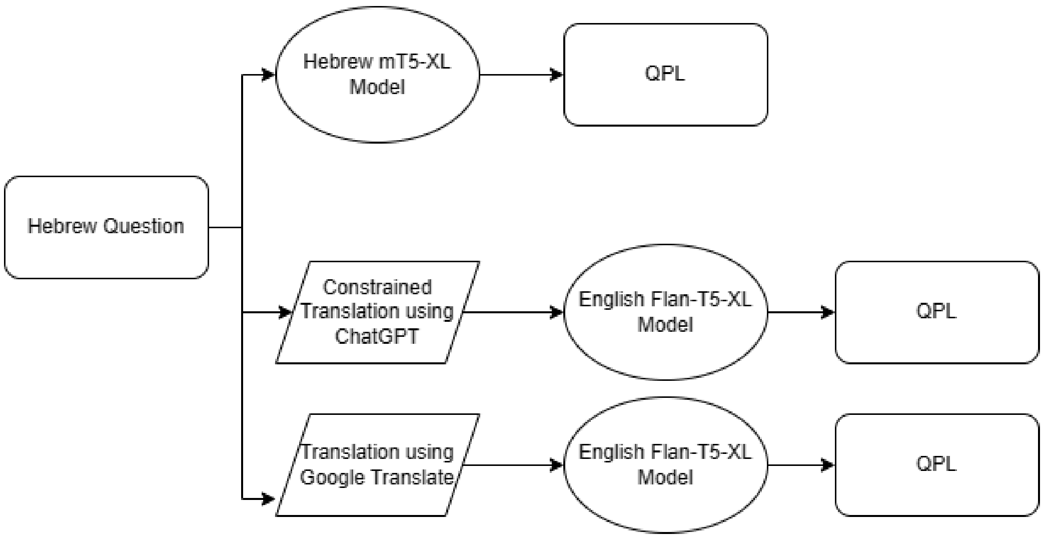


Figure 5.4: Cross-lingual QPL Generation Strategies

5.3.2 Schema Linking Robustness

For robustness checking, we try three combinations of evaluation datasets to check how robust a model is to small changes in the input questions. More specifically, we test the schema linking capabilities of the model by changing schema items in the question. The three datasets we use are Spider-Syn [24], Spider-RT, and Spider-Syn-RT. Spider-RT and Spider-Syn-RT are datasets we made by translating the English questions to Hebrew and back to English using machine translation. Spider-RT uses the original Spider questions, and Spider-Syn-RT uses the Spider-Syn English questions (Fig. 5.5).

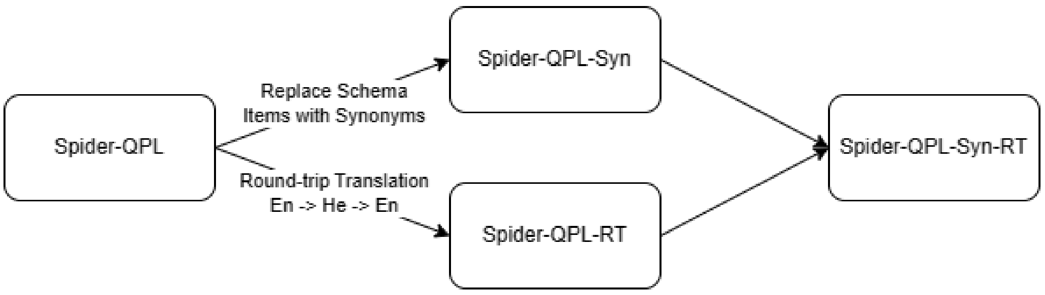

Spider-QPL
Replace Schema Items with Synonyms
Round-trip Translation En -> He -> En
Spider-QPL-Syn
Spider-QPL-RT
Spider-QPL-Syn-RT

Figure 5.5: Robustness Datasets

5.4 Experiments and Results

5.4.1 Cross-lingual Text-to-QPL

Table 5.2 summarizes the results of the nine models described by Fig. 5.2. There's no difference between manual and automatic translation in the very low-resource setting of only 500 training examples (mT5_He_500_HT and mT5_He_500). This means that, at least for Hebrew, manually translating the entire Spider dataset might not be beneficial (as done in CSpider). Instead, it seems effective to automatically translate the dataset and then have proficient Hebrew speakers verify the translations (as done in MultiSpider).

All training sets of each language were chosen to have the same difficulty distribution as in the entire dataset to ensure that the models have "equal opportunity" in learning both simple and complex QPL. And yet, we see that the absolute performance increases as the amount of training data increases (Fig. 5.6 and Fig. 5.7), both for English and for Hebrew. We observe that for very low training rates (500, 1000 samples), PICARD improves performance by large gaps with respect to the corresponding non-PICARD inference mode (as much as 9.4%). This gap indicates that with few training samples, the model does not learn the syntax of the target language, but as more training samples are available (reaching 6,500), the gap reduces to about 3% (comparable to the gap of about 2% observed on the Spider-QPL model with Flan-T5).

Finally, we observe that both for English and Hebrew, the mT5-based models improve in a similar manner as more training is available (see Figure 5.6) and that the gap between English and Hebrew reduces from 8.5% to 3.7%. This indicates that a multi-lingual base model like mT5 can learn a cross-lingual model based on automatically translated data with an acceptable gap in performance compared to the source English. Yet, the overall performance lags below the top model that was learned with a mono-lingual base model of the same size (3 billion parameters, Flan-T5-XL) with 72% for English / 68.3% for Hebrew vs. 80% for English with Flan-T5-XL (3B parameters).

Figure 5.8, Table 5.3, and Table 5.4 show error distribution across difficulties and the development schemas for the English models, respectively. Figure 5.9, Table 5.5, and Table 5.6 show the same for the Hebrew models. Lines marked as bold are "problematic" schemas, i.e., with an error rate greater than 40%. We identify the same problematic schemas that were observed in the original Spider-QPL experiments, with error rates worsened in the cross-lingual setting.

Model	Execution Accuracy		
	w/o PICARD	w/ PICARD	
mT5_En_500	36.65%	44.78%	(+ 8.13%)
mT5_En_1000	51.35%	60.74%	(+ 9.39%)
mT5_En_3000	62.86%	69.05%	(+ 6.19%)
mT5_En_All	68.09%	71.95%	(+ 3.86%)
mT5_He_500_HT	31.33%	36.27%	(+ 4.94%)
mT5_He_500	31.53%	36.27%	(+ 4.69%)
mT5_He_1000	43.81%	52.32%	(+ 8.51%)
mT5_He_3000	60.35%	65.67%	(+ 5.32%)
mT5_He_All	65.09%	68.28%	(+ 3.19%)

Table 5.2: Execution Accuracy with and without PICARD for All mT5-XL Models

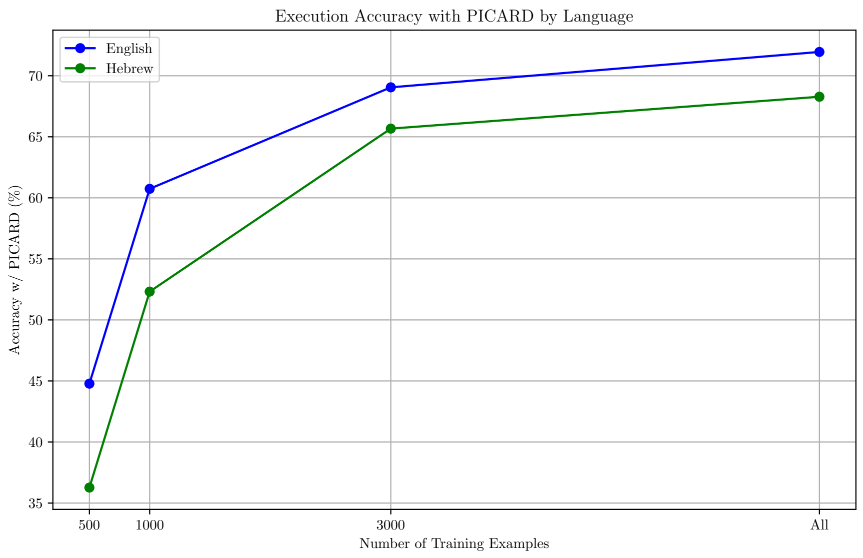


Figure 5.6: Execution Accuracy by Number of Training Examples

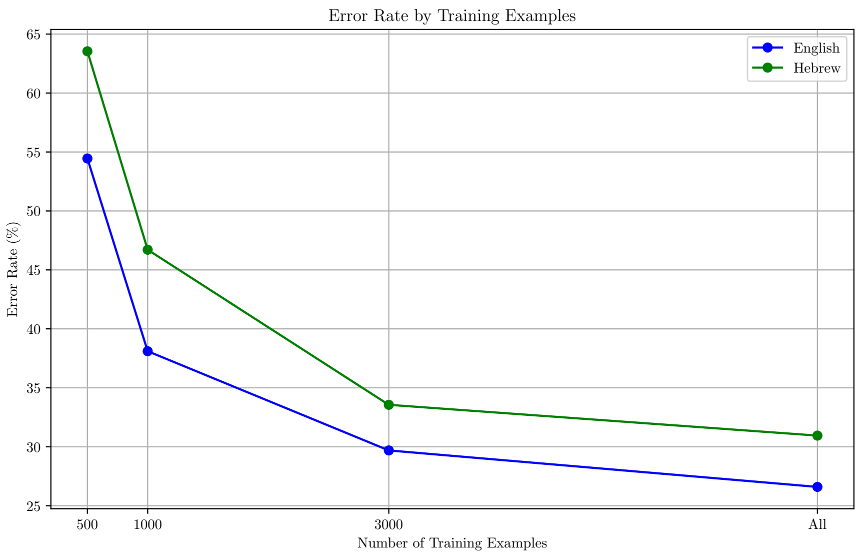


Figure 5.7: Error Rate

500 Training Samples

Difficulty	Errors	Samples	Error Rate
Easy	71	248	28.6%
Medium	214	446	48.0%
Hard	126	174	72.4%
Extra-hard	152	166	91.6%
Total	563	1034	54.4%

1000 Training Samples

Difficulty	Errors	Samples	Error Rate
Easy	43	248	17.3%
Medium	141	446	31.6%
Hard	90	174	51.7%
Extra-hard	120	166	72.3%
Total	394	1034	38.1%

3000 Training Samples

Difficulty	Errors	Samples	Error Rate
Easy	35	248	14.1%
Medium	104	446	23.3%
Hard	68	174	39.1%
Extra-hard	100	166	60.2%
Total	307	1034	29.7%

All Training Samples

Difficulty	Errors	Samples	Error Rate
Easy	31	248	12.5%
Medium	81	446	18.2%
Hard	71	174	40.8%
Extra-hard	92	166	55.4%
Total	275	1034	26.6%

Figure 5.8: Errors by Difficulty on English Models

5.4.2 Schema Link Robustness

In Table 5.7, we observe that the effect of a small perturbation in the input question significantly affects model performance with QPL as observed on

500 Training Samples

Difficulty	Errors	Samples	Error Rate
Easy	93	248	37.5%
Medium	271	446	60.8%
Hard	137	174	78.7%
Extra-hard	156	166	94.0%
Total	657	1034	63.5%

1000 Training Samples

Difficulty	Errors	Samples	Error Rate
Easy	61	248	24.6%
Medium	202	446	45.3%
Hard	94	174	54.0%
Extra-hard	126	166	75.9%
Total	483	1034	46.7%

3000 Training Samples

Difficulty	Errors	Samples	Error Rate
Easy	48	248	19.4%
Medium	126	446	28.3%
Hard	69	174	39.7%
Extra-hard	104	166	62.7%
Total	347	1034	33.6%

All Training Samples

Difficulty	Errors	Samples	Error Rate
Easy	41	248	16.5%
Medium	111	446	24.9%
Hard	73	174	42.0%
Extra-hard	95	166	57.2%
Total	320	1034	30.9%

Figure 5.9: Errors by Difficulty on Hebrew Models

SQL. We also see that combining two perturbations (round-trip translation and synonym substitution) causes the performance to decrease even more, which indicates that each type of noise presents a different linguistic challenge. The overall drop in performance is as large as 11.3% for the combined forms of perturbation. This significant drop demonstrates that the task of schema linking remains a significant challenge that requires specific attention beyond the problems of compositional generalization.

Dataset	Execution Accuracy
Spider-QPL	80.0%
Spider-QPL-RT	74.4%
Spider-QPL-Syn	73.2%
Spider-QPL-Syn-RT	68.7%

Table 5.7: Execution Accuracy on Robustness Datasets

5.5 Conclusion

The experiments and results presented in this chapter highlight several insights into the challenges and possibilities of cross-lingual semantic parsing

500 Training Samples

Schema	Errors	Samples	Error Rate
battle_death	**11**	**16**	**68.8%**
car_1	**68**	**92**	**73.9%**
concert_singer	**25**	**45**	**55.6%**
course_teach	12	30	40.0%
cre_Doc_Template_Mgt	**37**	**84**	**44.0%**
dog_kennels	**60**	**82**	**73.2%**
employee_hire_evaluation	**16**	**38**	**42.1%**
flight_2	**41**	**80**	**51.2%**
museum_visit	**11**	**18**	**61.1%**
network_1	22	56	39.3%
orchestra	13	40	32.5%
pets_1	**20**	**42**	**47.6%**
poker_player	9	40	22.5%
real_estate_properties	1	4	25.0%
singer	8	30	26.7%
student_transcripts_tracking	**52**	**78**	**66.7%**
tvshow	24	62	38.7%
voter_1	**7**	**15**	**46.7%**
world_1	**97**	**120**	**80.8%**
wta_1	**29**	**62**	**46.8%**
Total	563	1034	54.4%

1000 Training Samples

Schema	Errors	Samples	Error Rate
battle_death	**9**	**16**	**56.2%**
car_1	**57**	**92**	**62.0%**
concert_singer	10	45	22.2%
course_teach	0	30	0.0%
cre_Doc_Template_Mgt	23	84	27.4%
dog_kennels	**38**	**82**	**46.3%**
employee_hire_evaluation	6	38	15.8%
flight_2	**33**	**80**	**41.2%**
museum_visit	**8**	**18**	**44.4%**
network_1	17	56	30.4%
orchestra	6	40	15.0%
pets_1	12	42	28.6%
poker_player	0	40	0.0%
real_estate_properties	1	4	25.0%
singer	3	30	10.0%
student_transcripts_tracking	**42**	**78**	**53.8%**
tvshow	14	62	22.6%
voter_1	2	15	13.3%
world_1	**87**	**120**	**72.5%**
wta_1	**26**	**62**	**41.9%**
Total	394	1034	38.1%

Table 5.3: Errors by Schema on English Models (500 and 1000 Training Samples)

and schema linking robustness.

Our exploration into the cross-lingual text-to-QPL task, particularly with the focus on Hebrew, has shown that, while it is completely possible to fine-tune a multilingual pre-trained language model such as mT5, on the Hebrew text-to-QPL task and achieve reasonable performance (3.7% decrease from English using the same model), it is preferable to use a performant English-only pre-trained language model and automatic translation to achieve better

3000 Training Samples

Schema	Errors	Samples	Error Rate
battle_death	6	16	37.5%
car_1	**47**	**92**	**51.1%**
concert_singer	6	45	13.3%
course_teach	2	30	6.7%
cre_Doc_Template_Mgt	17	84	20.2%
dog_kennels	32	82	39.0%
employee_hire_evaluation	1	38	2.6%
flight_2	29	80	36.2%
museum_visit	5	18	27.8%
network_1	11	56	19.6%
orchestra	1	40	2.5%
pets_1	8	42	19.0%
poker_player	2	40	5.0%
real_estate_properties	1	4	25.0%
singer	1	30	3.3%
student_transcripts_tracking	**35**	**78**	**44.9%**
tvshow	13	62	21.0%
voter_1	2	15	13.3%
world_1	**64**	**120**	**53.3%**
wta_1	24	62	38.7%
Total	307	1034	29.7%

All Training Samples

Schema	Errors	Samples	Error Rate
battle_death 6	16	37.5%	
car_1	**43**	**92**	**46.7%**
concert_singer	7	45	15.6%
course_teach	0	30	0.0%
cre_Doc_Template_Mgt	12	84	14.3%
dog_kennels	29	82	35.4%
employee_hire_evaluation	3	38	7.9%
flight_2	27	80	33.8%
museum_visit	3	18	16.7%
network_1	12	56	21.4%
orchestra	2	40	5.0%
pets_1	8	42	19.0%
poker_player	3	40	7.5%
real_estate_properties	1	4	25.0%
singer	1	30	3.3%
student_transcripts_tracking	29	78	37.2%
tvshow	11	62	17.7%
voter_1	2	15	13.3%
world_1	**60**	**120**	**50.0%**
wta_1	16	62	25.8%
Total	275	1034	26.6%

Table 5.4: Errors by Schema on English Models (3000 and All Training Samples)

performance (73.4% vs. 68.3%) without the need to fine-tune on a Hebrew dataset. Furthermore, we saw that in the low-resource setting, using 500 examples of manually translated questions does not help the model learn better. While not to say that quantity beats quality, it would appear that current automatic translation solutions are "good enough" for this task.

The robustness experiments illuminate the challenges in schema linking under question perturbation. The noticeable drop in performance in the face

500 Training Samples

Schema	Errors	Samples	Error Rate
battle_death	**10**	**16**	**62.5%**
car_1	**79**	**92**	**85.9%**
concert_singer	**27**	**45**	**60.0%**
course_teach	**13**	**30**	**43.3%**
cre_Doc_Template_Mgt	**45**	**84**	**53.6%**
dog_kennels	**63**	**82**	**76.8%**
employee_hire_evaluation	**18**	**38**	**47.4%**
flight_2	**39**	**80**	**48.8%**
museum_visit	**15**	**18**	**83.3%**
network_1	**36**	**56**	**64.3%**
orchestra	**19**	**40**	**47.5%**
pets_1	**27**	**42**	**64.3%**
poker_player	**22**	**40**	**55.0%**
real_estate_properties	**2**	**4**	**50.0%**
singer	12	30	40.0%
student_transcripts_tracking	**65**	**78**	**83.3%**
tvshow	23	62	37.1%
voter_1	**11**	**15**	**73.3%**
world_1	**97**	**120**	**80.8%**
wta_1	**34**	**62**	**54.8%**
Total	657	1034	63.5%

1000 Training Samples

Schema	Errors	Samples	Error Rate
battle_death	**7**	**16**	**43.8%**
car_1	**71**	**92**	**77.2%**
concert_singer	**20**	**45**	**44.4%**
course_teach	9	30	30.0%
cre_Doc_Template_Mgt	26	84	31.0%
dog_kennels	**45**	**82**	**54.9%**
employee_hire_evaluation	6	38	15.8%
flight_2	32	80	40.0%
museum_visit	**8**	**18**	**44.4%**
network_1	21	56	37.5%
orchestra	10	40	25.0%
pets_1	**22**	**42**	**52.4%**
poker_player	12	40	30.0%
real_estate_properties	**3**	**4**	**75.0%**
singer	6	30	20.0%
student_transcripts_tracking	**49**	**78**	**62.8%**
tvshow	17	62	27.4%
voter_1	**10**	**15**	**66.7%**
world_1	**76**	**120**	**63.3%**
wta_1	**33**	**62**	**53.2%**
Total	483	1034	46.7%

Table 5.5: Errors by Schema on Hebrew Models (500 and 1000 Training Samples)

of synonyms and round-trip translation highlights the sensitivity of current models to variations in question formulation. This emphasizes the need for more sophisticated approaches that can better handle linguistic variability and understand the underlying semantics of questions, regardless of their surface form. This conclusion supports the need to study further data augmentation methods that focus on introducing more variability on the side of questions with various paraphrasing devices.

3000 Training Samples

Schema	Errors	Samples	Error Rate
battle_death	5	16	31.2%
car_1	**53**	**92**	**57.6%**
concert_singer 9	45	20.0%	
course_teach 3	30	10.0%	
cre_Doc_Template_Mgt	22	84	26.2%
dog_kennels	30	82	36.6%
employee_hire_evaluation	3	38	7.9%
flight_2	24	80	30.0%
museum_visit	6	18	33.3%
network_1	15	56	26.8%
orchestra	1	40	2.5%
pets_1	12	42	28.6%
poker_player	5	40	12.5%
real_estate_properties	1	4	25.0%
singer	2	30	6.7%
student_transcripts_tracking	**46**	**78**	**59.0%**
tvshow	12	62	19.4%
voter_1	5	15	33.3%
world_1	**70**	**120**	**58.3%**
wta_1	23	62	37.1%
Total	347	1034	33.6%

All Training Samples

Schema	Errors	Samples	Error Rate
battle_death	6	16	37.5%
car_1	**49**	**92**	**53.3%**
concert_singer	7	45	15.6%
course_teach	0	30	0.0%
cre_Doc_Template_Mgt	13	84	15.5%
dog_kennels	34	82	41.5%
employee_hire_evaluation	4	38	10.5%
flight_2	32	80	40.0%
museum_visit	4	18	22.2%
network_1	14	56	25.0%
orchestra	2	40	5.0%
pets_1	9	42	21.4%
poker_player	4	40	10.0%
real_estate_properties	1	4	25.0%
singer	2	30	6.7%
student_transcripts_tracking	**37**	**78**	**47.4%**
tvshow	10	62	16.1%
voter_1	4	15	26.7%
world_1	**67**	**120**	**55.8%**
wta_1	21	62	33.9%
Total	320	1034	30.9%

Table 5.6: Errors by Schema on Hebrew Models (3000 and All Training Samples)

CHAPTER 6

Conclusion

6.1 Main Contributions

In this work, we set out to answer three main research questions:

RQ1 How can we improve the compositional generalization of text-to-SQL models?

RQ2 How can we make the output of these models more interpretable to users?

RQ3 How robust are models to question formulation variations in a multilingual setting?

We introduced Query Plan Language (QPL), an intermediate language that bridges the gap between natural language questions and complex SQL queries by decomposing queries according to the operational semantics of the retrieval process. The work is summarized in Eyal, Bachar, Haroche, and

Elhadad [18], Eyal, Mahabi, Haroche, Bachar, and Elhadad [19], and Eyal and Elhadad [17].

In Chapter 3, we reported on experiments indicating that QPL is easier to learn than SQL for pre-trained language models using the same neural architecture. We also delved into the subject of question decomposition for QPL generation and established a benchmark for future work where semantically complex questions in natural language can be decomposed into a sequence of simpler questions.

Chapter 4 showed that neural architectures are not alone in preferring QPL over SQL, and programmers share the same sentiment. We described a user experiment which points to the difficulty programmers feel when aligning complex data retrieval queries with the intent of semantically challenging questions. The results indicate that a modular language like QPL holds the promise to be more interpretable than a compact language like SQL. This small-scale study points to promising future work to improve QPL's interpretability.

Finally, in Chapter 5, we shifted our focus from English as the main language for text-to-QPL to a different source language, namely Hebrew, and explored how well cross-lingual text-to-QPL performs in its hardest setting: English schema items and values with questions in a different language. We saw that advancements in machine translation allow us to train a formidable Hebrew text-to-QPL model by automatically translating Hebrew questions to English and using our best English text-to-QPL model. The experiments, however, also demonstrate that current models suffer from low robustness in aspects related to schema linking when the questions exhibit high variability with respect to the terms used in the database schema.

6.2 Future Work

6.2.1 Text-to-SQL Improvements

In addition to the work presented here, we believe the QPL approach has additional potential, which we wish to explore.

With the shortcomings of the Spider dataset, more challenging datasets have emerged, such as BIRD-SQL [44] and UNITE [39], which have more realistic and complex questions and queries across different domains. As shown in this work, QPL works well for complex questions, and we would like to verify that the same applies to these more challenging datasets. In addition to cross-domain benchmarks, we would also like to use QPL in specific complex domains, such as Electronic Health Records [41].

For a more extensive study of compositional generalization, a compositional split should be created and evaluated based on the characteristics of QPL, using techniques such as DBCA [36].

With the rise of LLMs, we would like to enable users to interact with a model that will generate QPL incrementally, according to the user's specifications. This would also require an auto-regressive text-to-QPL model, which, given a question and the first steps of a QPL, can then produce the next step.

To create more training data to train text-to-QPL models, we need an extensive data augmentation method to synthesize new question-and-QPL pairs by composing smaller QPL into larger ones and synthesizing a question from the new QPL. QPL enables simple syntactic transformations based on the modular structure of the language, which lends itself to effective scrambling without semantic impact. On the side of questions, paraphrasing decomposed questions also promises to yield effective variations.

Finally, we would like to explore disentangling the knowledge for planning the QPL (what operators and in what order) from the knowledge required to run the QPL based on a specific schema.

6.2.2 General Semantic Parsing Directions

Text-to-SQL is a specific instance of the more general task of Semantic Parsing. In this more general setting, the task consists of mapping natural language questions and utterances to expressions in a formal language with known formal semantics.

Other key examples of semantic parsing include text-to-code, where the target language is represented as Abstract Syntax Trees (AST) in the selected programming language, or different query languages, in particular over knowledge graphs with languages like SparQL [2, 3] or Cypher [21].

Our main contribution with the definition of QPL is a focus on the following principle: it is beneficial when researching semantic parsing to question the level of abstraction that is implicitly adopted in the definition of the target formal language. The definition of intermediary languages that mediate the translation from natural language and the target formal language has multiple benefits:

1. **Ease of learning**: intermediary languages may be easier to map to the source natural language utterances because they have a similar syntax or because their modular nature leads to better generalization capabilities.

2. **Interpretability**: modular intermediary languages can be broken down into smaller units, which leads to better analysis of which part of the target semantic form is erroneous in case of breakdown.

3. **Grounded question decomposition**: the strategy of learning to decompose a complex question into sub-questions has been researched in many forms. The definition of modular intermediary languages for semantic parsing allows us to map complex questions to sub-questions on the basis of the target language. This general data augmentation strategy allows us to define datasets of question decompositions where the decomposition is grounded in the semantics of the target oanguagel

In recent followup work [50], the idea of grounded question decomposition has been explored by defining multi-step prompting strategies to solve text-to-QPL: a top-down question decomposer is learned to decompose complex natural language questions into an abstract query plan; this abstract query plan is then transformed bottom-up into a fully fleshed QPL program. This multi-step complex prompting strategy leads to marked improvements when deadling with complex questions.

The work of Bachar [1] compares different concrete syntaxes for QPL programs in the context of text-to-QPL. The comparison of bottom-up and top-down encodings of QPL ASTs leads to different ways in which a text-to-QPL model is trained. An ensemble model combining text-to-QPL models trained each on a different concrete syntax (JSON, YAML, parenthesized QPL with embedded sub-commands).

We consider that training models on QPL is a strategy similar to the Chain-of-Thought (CoT) prompting strategy: the QPL structure corresponds to the steps of the computation when executing recursively the sub-expressions according to the operational semantics of SQL. Learning to generalize to complex programs on the basis of low-level executions remains a challenging open question: can LLMs (pre-trained on next-word prediction) learn to plan and learn to reason according to the semantic inference rules of the target language properties. Semantic parsing with modular intermediate languages promises to help study this fundamental questions.

APPENDIX A

The Inception of QPL

The Query Plan Language (QPL) has had three generations during our research. This appendix will cover these different generations, what led to their syntax, and how they evolved.

The first generation of QPL had a nested structure, similar to a LISP program (Figure A.1). The main advantage of this syntax is that every sub-tree in the nested structure is valid QPL, so it can be built compositionally by wrapping a sub-tree in another operator. However, there were a few major downsides to this syntax:

- It is hard to read deeply nested QPL, as the innermost nodes are indented too far to the right.
- The language models we use for our experiments were trained on natural language, not code, and this code-like syntax made it more difficult for the model to learn.
- Converting this structure to CTE is harder, as CTEs are a linear se-

quence of bindings, while this structure is a tree and requires some post-processing.

```
[
  [
    Scan Table [ visitor ]
      Predicate [ Level_of_membership = 1 ]
      Output [ ID ],
    Scan Table [ visit ]
      Output [ visitor_ID , Total_spent ]
  ] Into: Join Predicate [ ID = visitor_ID ]
    Output [ visit.Total_spent ]
] Into: Aggregate Output [ SUM(Total_spent) ]
```

Figure A.1: Nested QPL

Given the above shortcomings, we decided to convert the tree-like syntax to a syntax that will be easier for large language models to learn and for humans to read and understand. This led to the formulation of Flat QPL (Figure A.2). Flat QPL takes the nested structure and flattens it to a list of numbered lines. Every operator that is not a Scan, i.e., not a leaf operator, takes line numbers as inputs, and these are the only inputs that the line is allowed to use. In this structure, we don't have the problem of readability, as there's no indentation; it is easier to go line-by-line and understand the execution plan rather than reading from the inside-out; and perhaps most importantly, large language models had an easier time "making sense" of this new language, as it was just a series of steps.

The last step in QPL's evolution was the removal of extraneous information and ambiguity. This final version of QPL (Figure A.3) differs from the previous version in the following ways:

- No table qualification where it's not needed, e.g., in a Scan node scanning the `visit` table, we don't need to qualify every column with `visit.`.

```
#1 = Scan Table [ visitor ]
         Predicate [ Level_of_membership = 1 ]
         Output [ ID ]
#2 = Scan Table [ visit ]
         Output [ visitor_ID , Total_spent ]
#3 = Join [ #1, #2 ] Predicate [ ID = visitor_ID ]
                     Output [ visit.Total_spent ]
#4 = Aggregate [ #3 ] Output [ SUM(Total_spent) ]
```

Figure A.2: Flat QPL

- Line numbers are used as qualifiers in binary operators (Join, Union, Intersect, Except) to denote from which input a column comes.
- Every aggregation now has an alias and that alias will be referenced whenever that result is used.

These changes mainly improved the readability of QPL.

```
#1 = Scan Table [ visitor ]
         Predicate [ Level_of_membership = 1 ]
         Output [ ID ]
#2 = Scan Table [ visit ]
         Output [ visitor_ID , Total_spent ]
#3 = Join [ #1, #2 ] Predicate [ #1.ID = #2.visitor_ID ]
                     Output [ #2.Total_spent ]
#4 = Aggregate [ #3 ] Output [ SUM(Total_spent) AS Sum_Total_spent ]
```

Figure A.3: QPL's final form

APPENDIX B

Grammar of QPL

```
<qpl> ::= <line>+
<line> ::= #<integer> = <operator>
<operator> ::= <scan>
             | <aggregate>
             | <filter>
             | <sort>
             | <topsort>
             | <join>
             | <except>
             | <intersect>
             | <union>

-- Leaf operator
<scan> ::= Scan Table [ <table-name> ] <pred>? <distinct>? <output-non-qualif>

-- Unary operators
<aggregate> ::= Aggregate [ <input> ] <group-by>? <output-non-qualif>
```

```
<filter>    ::= Filter [ <input> ] <pred> <distinct>? <output-non-qualif>
<sort>      ::= Sort [ <input> ] <order-by> <with-ties>? <output-non-qualif>
<topsort>   ::= TopSort [ <input> ] Rows [ <number> ] <order-by>
                            <withTies>? <output-non-qualif>

-- Binary operators
<join>      ::= Join [ <input> , <input> ] <pred>? <distinct>? <output-qualif>
<except>    ::= Except [ <input> , <input> ] <pred> <output-qualif>
<intersect> ::= Intersect [ <input> , <input> ] <pred>? <output-qualif>
<union>     ::= Union [ <input> , <input> ] <output-qualif>

<group-by>   ::= GroupBy [ <column-name> (, <column-name>)* ]
<order-by>   ::= OrderBy [ <column-name> <dir> (, <column-name> <dir>)* ]
<with-ties>  ::= WithTies [ true | false ]
<dir>        ::= ASC | DESC
<pred>       ::= Predicate [ <comparison> (AND | OR <comparison>)* ]
<distinct>   ::= Distinct [ true | false ]
<output-non-qualif> ::= Output [ <column-name> (, <column-name>)* ]
<output-qualif> ::= Output [ <qualif-column-name> (, <qualif-column-name>)* ]
<qualif-column-name> ::= # <number> . <column-name>
<input> ::= <integer>
```

APPENDIX C

Constrained Decoding: PICARD for QPL

Most current semantic parsing models use fine-tuned Large Language Models (LLMs) in an encoder-decoder architecture: the context (schema encoding and question) is encoded by the model into a vector representation, which is then used as an input to the decoder, which outputs tokens in an auto-regressive manner to generate the target SQL query. When using pre-trained language models that have been trained on natural language, the decoder has difficulty generating syntactically correct SQL (or QPL) output. PICARD [64] introduced a general approach to this overall difficulty called *constrained decoding*: instead of eagerly selecting the most likely token at each stage of the decoding, PICARD performs a beam-search strategy over multiple candidate sequences. A token is sampled from the LLM at each stage and passed to the k most successful prefix sequences. The prefix sequence and the candidate token are inspected by an incremental parser for the target language (SQL for PICARD, QPL in our case). If the parser confirms the token can be appended to the candidate prefix and still obey the syntax of the language, it is kept as a new candidate; otherwise, the beam search samples another candidate from the LLM.

In our experiments, we reimplemented PICARD, which originally tested the constrained decoding strategy for SQL, by designing an incremental parser for QPL. Our re-implementation consists of the QPL incremental parser and a server that glues HuggingFace Transformers' beam search with the incremental parser.

To understand how the parser works, we will begin by understanding its type:

```scala
type SchemaReader[A] = ReaderT[Parser, SqlSchema, A]
type QplParser[A] = StateT[SchemaReader, QplState, A]
```

A `QplParser[A]` is a `Parser[A]`. A `Parser[A]` knows how to take a string and convert it to some data type of our choosing, and we denote that with the generic type `A`. In addition to being able to parse a string, a `QplParser[A]` has two additional capabilities: the capability to read an environment of type `SqlSchema` and the capability to maintain "global" state of type `QplState`. These two capabilities allow us to perform semantic checks during parsing and fail the parse if it doesn't make sense given the environment and/or the state.

When parsing a full QPL program, we are using a `QplParser[Qpl]`, with the `Qpl` type being defined as:

```scala
type Qpl = List[Line]

// `Operation` is one of the 10 QPL node types
case class Line(idx: Int, operation: Operation)
```

Before delving into the specifics of each sub-parser (one for each node type), we need to understand the state maintained by the parser to construct a lossy abstract syntax tree (AST) of QPL. The AST is lossy because we don't need to interpret or compile it at any point. We're only interested in whether the parse is successful or not. While we could parse QPL and keep the most trivial value () of type Unit, that would make it harder to debug. The structure of the state is as follows:

```
case class QplState(
  currentIdx: Int,
  seen: Set[Int],
  idxToTable: Map[Int, Table]
)

enum Table {
  case Named(name: String, columns: List[Column])
  case Indexed(idx: Int, columns: List[Column])
}

enum Column {
  case Dummy // A "1 AS One" column
  case Plain(name: String, typ: ColumnType, keys: List[KeyType])
  case Aliased(name: String, typ: ColumnType, keys: List[KeyType])
}
```

For each line index, we keep a Table. The Table contains a list of Column, and each Column contains its name, its data type, and whether it's a primary or foreign key (a column can be several keys, as is the case for cross tables, or none at all).

For example, when parsing the following QPL:

```
#1 = Scan Table [ singer_in_concert ] Output [ concert_ID ]
#2 = Aggregate [ #1 ] GroupBy [ concert_ID ]
                      Output [ countstar AS Count_Star , concert_ID ]
#3 = Scan Table [ concert ] Output [ Theme , concert_Name , concert_ID ]
#4 = Join [ #2 , #3 ] Predicate [ #2.concert_ID = #3.concert_ID ]
                      Output [ #3.Theme , #2.Count_Star , #3.concert_Name ]
```

When we reach node #3, our state will have the following information:

- We are at index 3
- We have previously seen indices 1 and 2
- The `singer_in_concert` table from line #1 has output the column `concert_ID`, which is both a primary key of the `singer_in_concert` table and a foreign key of the `concert` table (this is the information we have from the schema)
- The indexed table from line #2 has output both `concert_ID` and the special `COUNT(*)` column.

Below are the specific semantic checks that each sub-parser performs when parsing a QPL line:

- `Scan` nodes have a predicate and an output. In the predicate, we make sure that the types of the two sides of the comparison are the same and that the column name in the predicate is indeed part of the scanned table. In the output, we make sure that all columns are part of the scanned table as well. The following `Scan` line fails parsing because of a type mismatch between the `Year` column and the constant `'zero'`:

  ```
  Scan Table [ concert ] Predicate [ Year >= 'zero' ]
                         Output [ Stadium_ID , Year ]
  ```

- `Aggregate` nodes have a `GroupBy` clause and an output. The output will have at least one `Aliased` column, which signifies a column that looks like `SUM(column) AS Sum_column`. We make sure that the `GroupBy` columns belong to the input table and that all aggregates have the same naming convention using the `AS` keyword. The following `Aggregate` line fails to parse as it does not conform to the naming convention:

  ```
  Aggregate [ #1 ] Output [ AVG(Age) AS AgeAverage ]
  ```

- `Filter` nodes are similar to `Scan` nodes in that their predicates are similar, but `Filter` requires looking up the input columns' types to make sure the types in the two sides of the predicate match.
- `Top` nodes are simple, with only a `Rows` clause that has to be a strictly positive integer.
- `Sort` nodes have an `OrderBy` clause, which we must make sure uses columns previously output by the node's input.
- `TopSort` is essentially `Sort` and `Top` in one operator and performs the same checks as these two nodes.
- `Join` nodes are the most involved with respect to semantic checks. In addition to type checking in the `Predicate` clause, we also must check that if the predicate is an equality comparison, then one side must be a primary key while the other must be a foreign key, and those keys should reference the same table.
- `Intersect` and `Except` nodes are similar in that they have a predicate but are not as complex as `Join`, as these nodes don't require a primary key-foreign key relationship in their predicate.
- `Union` nodes are the most simple nodes as they require no semantic checks other than whether the output columns appear in the input.

There are also checks that are being performed for every sub-parser, such as no duplicate columns in the output list, no output of columns that were not output by preceding lines, and no inputs that were not seen before, i.e., we don't parse `#2` as input if line `#2` hasn't been parsed yet.

APPENDIX D

From SQL to Executable QPL

In order to convert the SQL queries in the Spider dataset to QPL and then to executable SQL, we go through four phases:

1. Translate the SQL queries from the SQLite dialect to the T-SQL dialect
2. Generate execution plans
3. Construct QPL from execution plans
4. Generate CTEs from QPL

We will go over each one of these phases in detail.

D.1 SQLite to T-SQL Conversion

An implementation detail of the conversion from SQL to QPL is that we need to generate an execution plan using Microsoft SQL Server 2019. SQL

Server does not support the SQLite dialect that Spider is written in but uses its own dialect called T-SQL, so a translation between dialects is needed.

In order to translate SQLite to T-SQL, we use the existing tokenization of queries in Spider to reconstruct the query. We make the following modifications:

- While SQLite uses the `LIMIT` keyword at the end of the query to limit the number of results, T-SQL uses the `TOP` keyword after the `SELECT` keyword to do the same thing. We search for uses of `LIMIT` and for each one, replace it with the respective `TOP` along with the integer argument to that keyword.
- In SQLite, it is valid to select an aggregated column (`MIN`, `MAX`, `SUM`, `COUNT`, `AVG`) and a non-aggregated column without specifying the column by which the aggregates are grouped by. In T-SQL, this produces an error. To mitigate this, we review the column list in the `SELECT` clause, take the non-aggregated columns, and copy them to a new (or existing) `GROUP BY` clause.
- SQLite supports a `JOIN` operation that has no predicate, i.e., while we would usually see `T1.C1 JOIN T2.C2 ON T1.C1 = T2.C2`, SQLite permits not specifying the `ON` part. This results in a cross-join (a cartesian product of the two tables), and in T-SQL, we specify that by adding the `CROSS` modifier to `JOIN`.

D.2 Execution Plan Generation

Once we have the T-SQL queries, we can run them through SQL Server and get their execution plans. An SQL Server execution plan is an XML file containing mostly statistics on how performant each part of the query is. For our purposes, we ignore this data and use the tree structure embedded in the execution plan to get information on which building blocks were used when constructing the query. Figure D.1 shows an example of such a tree structure.

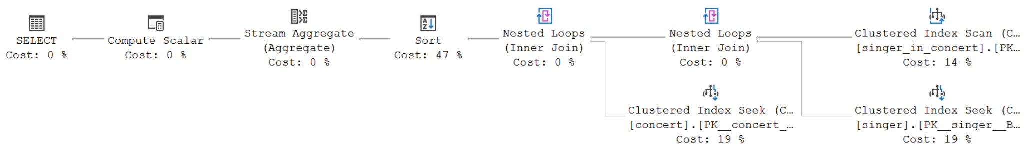


Figure D.1: Execution Plan

D.3 Constructing QPL from Execution Plans

We now need to parse the execution plans' XML and convert it to our own language, QPL. This process is essentially about *discarding information*, keeping only the bare minimum to create meaningful QPL.

During parsing of the XML, we identified 22 different node types. As QPL has only 10 operators, it is obvious that some XML nodes contain no valuable information or some contain duplicate information. For example, in an XML execution plan, there is a difference between an `<IndexScan>` node and an `<IndexSeek>` node. In QPL, however, these are both mapped to a `Scan` node, as the semantics of both `<IndexSeek>` and `<IndexScan>` are the same but with different implementations. The same goes for different implementations of joins: there are different algorithms to perform a join, such as nested loops or hash table lookup, but in QPL, we ignore those distinctions and call the node simply `Join`. Another node we ignore is `<ComputeScalar>`, which is used for simple computations that don't affect the number of rows being output, e.g., the SQL operator `AVG(C)` is actually implemented as two operators one after the other: aggregate the sum and count of `C`, and then compute the average by dividing the two. Another computation is type conversions, such as integers to strings and integer size conversions. These do not affect the semantics of the query, and so in QPL, we completely remove the `<ComputeScalar>` node and, when applicable, "push" the operator to its parent.

At the end of the conversion from execution plans to QPL, we get "raw" QPL. It is not yet in its final form, but it has all the information in order to get it there. Figure D.2 shows an example of raw QPL. Raw QPL essentially keeps all the qualified names, even when unnecessary (for example, if we scan the `singer` table, we know the predicate can only use columns of that table, so `singer.Country` can be shortened to `Country`), and does not yet use the line numbering scheme inside QPL's operators.

```
#1 = Scan Table [ singer ]
          Predicate [ singer.Country = 'France' ]
          Output [ singer.Age , singer.Country ]
#2 = Aggregate [ #1 ] Output [ AVG(singer.Age) ,
                               MIN(singer.Age) ,
                               MAX(singer.Age) ]
```

Figure D.2: Raw QPL

The raw QPL now needs to be post-processed to get it to its final form. In this stage, we either remove the table qualification completely if the operator is unary (in which case, it is obvious which table the column belongs to) or substitute the table with its matching line number if it's a binary operator. We also take care to add aliases for all aggregations, e.g., turn `SUM(C)` to `SUM(C) AS Sum_C`. This is important for two reasons: the first is that aggregations don't have a name for the returned column by default, which can be a problem when using Python and the `pandas` library to read a result set into a key-value data structure, as the column name acts as a key; second, because we're dealing with a data-flow graph, we need to be able to propagate values from earlier lines to later lines. We do this by giving an aggregation a name and propagating it to the later lines that use the exact same aggregation.

D.4 QPL to CTE Conversion

Once we have the final version of QPL, we need to execute it. Fortunately, most SQL dialects, including T-SQL, support the Common Table Expression

(CTE) syntax. CTEs allow us to create bindings of intermediary results to tables and use them to create larger SQL queries. Figure D.3 shows an example SQL query and its equivalent CTE.

To convert QPL to CTE, we go over the QPL line-by-line and convert each operator to its respective CTE, taking care to remember what names we give to each line so we can use it in later CTEs. Once we have one CTE per line, we tie everything together by taking the last CTE name and running `SELECT * FROM LastCTE`.

SQL:

```
SELECT AVG(age) AS Avg_age,
       MIN(age) AS Min_age,
       MAX(age) AS Max_age
FROM singer
WHERE country = 'France'
```

CTE:

```
WITH
CTE_1 AS (
  SELECT age
  FROM singer
  WHERE country = 'France'
),
CTE_2 AS (
  SELECT AVG(age) AS Avg_age,
         MIN(age) AS Min_age,
         MAX(age) AS Max_age
  FROM CTE_1
)
SELECT * FROM CTE_2
```

Figure D.3: SQL and equivalent CTE

APPENDIX E

User Experiment Details

We designed a user experiment to test the following hypothesis: programmers identify whether a complex query in QPL corresponds to the intent of a natural language question better than for SQL.

The programmers who participated in the experiment knew SQL (with more than 5 years of experience in an industrial job) but had never heard of QPL before. We, therefore, introduced QPL with a set of examples. This Appendix shows the documents we used.

The subjects of our user experiment received examples of pairs of questions and SQL/QPL, along with instructions on what they should do, given their test examples. The examples given are as follows:

```
Example 1:

Schema: car_1
cars_data : Id , MPG , Cylinders , Edispl , Horsepower , Weight ,
            Accelerate , Year
```

```
model_list : ModelId , Maker , Model
continents : ContId , Continent
countries : CountryId , CountryName , Continent
car_names : MakeId , Model , Make
car_makers : Id , Maker , FullName , Country

Question: What are the names and ids of all
countries with at least one car maker?


QPL:
#1 = Scan Table [ countries ] Output [ CountryId , CountryName ]
#2 = Scan Table [ car_makers ] Output [ Country ]
#3 = Join [ #1, #2 ] Predicate [ #1.CountryId = #2.Country ]
     Output [ #1.CountryId , #1.CountryName ]
#4 = Aggregate [ #3 ] GroupBy [ CountryId ]
     Output [ CountryId , CountryName , countstar as Count_Star ]
#5 = Filter [ #4 ] Predicate [ Count_Star >= 1 ]
     Output [ CountryId , CountryName ]

Answer: True

Example 2:

Schema: student_transcripts_tracking
Addresses : address_id , line_1 , line_2 , line_3 , city , zip_postcode ,
            state_province_county , country , other_address_details
Sections : section_id , course_id , section_name , section_description ,
           other_details
Transcripts : transcript_id , transcript_date , other_details
Transcript_Contents : student_course_id , transcript_id
Students : student_id , current_address_id , permanent_address_id ,
           first_name , middle_name , last_name , cell_mobile_number ,
           email_address , ssn ,  date_first_registered ,
           date_left , other_student_details
Student_Enrolment_Courses : student_course_id , course_id ,
                            student_enrolment_id
Courses : course_id , course_name , course_description , other_details
Departments : department_id , department_name , department_description ,
              other_details
Semesters : semester_id , semester_name , semester_description , other_details
Degree_Programs : degree_program_id , department_id , degree_summary_name ,
                  degree_summary_description , other_details
Student_Enrolment : student_enrolment_id , degree_program_id , semester_id ,
                    student_id , other_details

Question: What are the first, middle, and last
names, along with the ids, of all students who
enrolled in 2 degree programs in one semester?

QPL:
#1 = Scan Table [ Students ]
     Output [ student_id , first_name ,
              middle_name , last_name ]
#2 = Scan Table [ Student_Enrolment ] Output [ student_id ]
#3 = Join [ #1, #2 ] Predicate [ #1.student_id = #2.student_id ]
     Output [ #1.student_id , #1.first_name ,
              #1.middle_name , #1.last_name ]
#4 = Aggregate [ #3 ] GroupBy [ student_id ]
     Output [ student_id , first_name ,
              middle_name , last_name ,
              countstar as Count_Star ]
#5 = Filter [ #4 ] Predicate [ Count_Star = 2 ]
     Output [ student_id , first_name ,
              middle_name , last_name ]
```

```
Answer: False

Example 3:

Schema: flight_2
airlines : uid , Airline , Abbreviation , Country
airports : City , AirportCode , AirportName , Country , CountryAbbrev
flights : Airline , FlightNo , SourceAirport , DestAirport

Question: Which city has most number of arriving flights?


SQL:
SELECT TOP 1 T1.city
FROM AIRPORTS AS T1
       JOIN flights AS T2
         ON T1.airportcode = T2.destairport
GROUP BY T1.city
ORDER BY COUNT(*) DESC

Answer: True

Example 4:

Schema: cre_Doc_Template_Mgt
Ref_Template_Types : Template_Type_Code , Template_Type_Description
Templates : Template_ID , Version_Number , Template_Type_Code ,
            Date_Effective_From , Date_Effective_To ,
            Template_Details
Documents : Document_ID , Template_ID , Document_Name ,
            Document_Description , Other_Details
Paragraphs : Paragraph_ID , Document_ID , Paragraph_Text , Other_Details
```

```
Question: Which template type code is used by most
number of documents?


SQL:
SELECT TOP 1 T1.template_type_code
FROM Templates AS T1
       JOIN documents AS T2
         ON T1.template_id = T2.template_id
GROUP BY T1.template_type_code
ORDER BY COUNT(*) DESC

Answer: False
```

After being presented with the above examples, the subjects were asked to perform the following tasks:

1. For each pair of question and SQL/QPL, answer whether the code retrieves the correct information as asked in the question.
2. Record how long did it take you to reach a decision.
3. Rank the difficulty of the example, with 1 being “very easy” and 5 being “very hard”.

APPENDIX F

Question Decomposition Prompt

We report in Chapter 3 on experiments where we learn to decompose natural language questions into a sequence of simpler natural language questions. We use GPT-3.5 (September 2023 version) through the OpenAI API in an in-context learning (ICL) method to synthesize a training dataset by computing decomposed natural language questions for each step in the corresponding QPL program in the training dataset. We bootstrap the method by manually converting 10 specific QPL programs into a sequence of simple natural language questions. We then feed some of these examples into the complex ICL prompt shown below.

For each sample in the training and development sets of Spider-QPL, we insert into this prompt the encoding of the schema (simple encoding in the form of table names and column names), then the sample question and the corresponding QPL program. We then expect GPT-3.5 to produce a question decomposition in the form of one question for each step in the QPL program.

As specified in the prompt, we require the output to contain exactly the same number of questions as there are steps in the QPL prompt. When this is not the case, we retry the prompt up to three times. In all cases, GPT-3.5 eventually produced an output that obeys this constraint.

The dataset that is eventually synthesized contains a list of pairs (original Spider question, sequence of decomposed questions in natural language). We then train a Question Decomposer model on this synthetic dataset, which, given a question and a schema as input, returns a sequence of simple questions.

```
QPL is a formalism to describe data retrieval operations over an
SQL schema in a modular manner. A QPL plan is a sequence of instructions
for querying tabular data to answer a natural language question.
Forget everything you know about SQL. Only use the following explanations.

A schema is specified as a list of <table> specifications in the
format: <table>: <comma separated list of columns>

A plan contains a sequence of operations. All operations return
a stream of tuples. All operations take as input either a physical
table from the schema (for the Scan operation) or the output of other
operations.

This is the formal specification for each operation:

{Grammar from Appendix B}

Let's think step by step to convert QPL plan to a natural language
plan given scheme, question, and QPL that describes the question.

In the natural language plan: 1. You must have exactly the same
number of questions as there are steps in the QPL. 2. The questions
you generate must follow exactly the same order as the steps in the
QPL.

Example 1:

Schema:
Table Visitor (ID, Name, Age, Level_of_membership)
Table Museum (Museum_ID, Name, Open_Year, Num_of_staff)
Table Visit (Visitor_ID, Museum_ID, Total_Spent, Num_of_Ticket)
```

```
Question:
What is the total ticket expense of the visitors whose membership
level is 1?

QPL Plan:
#1 = Scan Table [ visitor ] Predicate [ visitor.Level_of_membership
= 1 ] Output [ ID ]
#2 = Scan Table [ visit ] Output [ visitor_ID , Total_spent ]
#3 = Join [ #1, #2 ] Predicate [ visitor.ID = visit.visitor_ID ]
Output [ visit.Total_spent ]
#4 = Aggregate [ #3 ] Output [ SUM(visit.Total_spent) ]

Natural Language Plan:
#1 = Scan the table Visitor to find who are the visitors with membership
level 1
#2 = Scan the table Visit to find what is the total spent by visitors
during their visits
#3 = Join #1 and #2 to find what is the total spent by each visitor
with membership level 1 during their visits
#4 = Group #3 by Visitor and aggregate the sum of total spent to
find what is the total spent by all visitors with membership level
1 during their visit

{ Five more examples follow in a similar way. }

Now your turn:

Schema: {schema}

Question: {question}

QPL Plan: {qpl}

Natural Language Plan:
```

The full prompt can be found in the GitHub repository for the QPL project at https://github.com/bgunlp/qpl.

Bibliography

[1] Amir Bachar. Text-to-sql parsing with an ensemble of intermediate target languages. Master's thesis, Ben-Gurion University of the Negev, 2024.

[2] Debayan Banerjee, Pranav Ajit Nair, Jivat Neet Kaur, Ricardo Usbeck, and Chris Biemann. Modern baselines for sparql semantic parsing. In *Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '22, page 2260–2265, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450387323. doi: 10.1145/3477495.3531841. URL `https://doi.org/10.1145/3477495.3531841`.

[3] Debayan Banerjee, Pranav Nair, Ricardo Usbeck, and Chris Biemann. The role of output vocabulary in T2T LMs for SPARQL semantic parsing. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Findings of the Association for Computational Linguistics: ACL 2023*, pages 12219–12228, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-acl.774. URL `https://aclanthology.org/2023.findings-acl.774`.

[4] 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. doi: 10.18653/v1/P19-1448. URL `https://aclanthology.org/P19-1448`.

[5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[6] Shulin Cao, Jiaxin Shi, Liangming Pan, Lunyiu Nie, Yutong Xiang, Lei Hou, Juanzi Li, Bin He, and Hanwang Zhang. KQA pro: A dataset with explicit compositional programs for complex question answering over knowledge base. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6101–6119, Dublin, Ireland, May 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.acl-long.422. URL `https://aclanthology.org/2022.acl-long.422`.

[7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[8] Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks, 2022.

[9] Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Yunxuan Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, Albert Webson, Shixiang Shane Gu, Zhuyun Dai, Mirac Suzgun, Xinyun Chen, Aakanksha Chowdhery, Alex Castro-Ros, Marie Pellat, Kevin Robinson, Dasha Valter, Sharan Narang, Gaurav Mishra, Adams Yu, Vincent Zhao, Yanping Huang, Andrew Dai,

Hongkun Yu, Slav Petrov, Ed H. Chi, Jeff Dean, Jacob Devlin, Adam Roberts, Denny Zhou, Quoc V. Le, and Jason Wei. Scaling instruction-finetuned language models, 2022.

[10] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, jun 1970. ISSN 0001-0782. doi: 10.1145/362384.362685. URL `https://doi.org/10.1145/362384.362685`.

[11] 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 *Human Language Technology: Proceedings of a Workshop held at Plainsboro, New Jersey, March 8-11, 1994*, 1994. URL `https://aclanthology.org/H94-1010`.

[12] Xiang Deng, Ahmed Hassan Awadallah, Christopher Meek, Oleksandr Polozov, Huan Sun, and Matthew Richardson. Structure-grounded pre-training for text-to-SQL. In Kristina Toutanova, Anna Rumshisky, Luke Zettlemoyer, Dilek Hakkani-Tur, Iz Beltagy, Steven Bethard, Ryan Cotterell, Tanmoy Chakraborty, and Yichao Zhou, editors, *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1337–1350, Online, June 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.naacl-main.105. URL `https://aclanthology.org/2021.naacl-main.105`.

[13] Zhenyun Deng, Yonghua Zhu, Yang Chen, M. Witbrock, and Patricia J. Riddle. Interpretable amr-based question decomposition for multi-hop question answering. In *International Joint Conference on Artificial Intelligence*, 2022.

[14] Zhenyun Deng, Yonghua Zhu, Yang Chen, Michael Witbrock, and Patricia Riddle. Interpretable amr-based question decomposition for multi-hop question answering. In Lud De Raedt, editor, *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence,*

IJCAI-22, pages 4093–4099. International Joint Conferences on Artificial Intelligence Organization, 7 2022. doi: 10.24963/ijcai.2022/568. URL `https://doi.org/10.24963/ijcai.2022/568`. Main Track.

[15] Li Dong and Mirella Lapata. Language to logical form with neural attention. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 33–43, Berlin, Germany, August 2016. Association for Computational Linguistics. doi: 10.18653/v1/P16-1004. URL `https://aclanthology.org/P16-1004`.

[16] Longxu Dou, Yan Gao, Mingyang Pan, Dingzirui Wang, Wanxiang Che, Dechen Zhan, and Jian-Guang Lou. Multispider: towards benchmarking multilingual text-to-sql semantic parsing. In *Proceedings of the Thirty-Seventh AAAI Conference on Artificial Intelligence and Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence and Thirteenth Symposium on Educational Advances in Artificial Intelligence*, AAAI'23/IAAI'23/EAAI'23. AAAI Press, 2023. ISBN 978-1-57735-880-0. doi: 10.1609/aaai.v37i11.26499. URL `https://doi.org/10.1609/aaai.v37i11.26499`.

[17] Ben Eyal and Michael Elhadad. Cross-lingual text-to-SQL and schema linking robustness. *Submitted*, 2024.

[18] Ben Eyal, Amir Bachar, Ophir Haroche, and Michael Elhadad. Semantic parsing for complex data retrieval: Targeting query plans vs. sql for no-code access to relational databases, 2023.

[19] Ben Eyal, Moran Mahabi, Ophir Haroche, Amir Bachar, and Michael Elhadad. Semantic decomposition of question and SQL for text-to-SQL parsing. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 13629–13645, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-emnlp.910. URL `https://aclanthology.org/2023.findings-emnlp.910`.

[20] 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. doi: 10.18653/v1/P18-1033. URL `https://aclanthology.org/P18-1033`.

[21] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 1433–1445, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450347037. doi: 10.1145/3183713.3190657. URL `https://doi.org/10.1145/3183713.3190657`.

[22] Grant Fritchey. *SQL Server Execution Plans: Third Edition*. Red Gate Books, 2018.

[23] Ruiliu Fu, Han Wang, Xuejun Zhang, Jun Zhou, and Yonghong Yan. Decomposing complex questions makes multi-hop QA easier and more interpretable. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih, editors, *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 169–180, Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.findings-emnlp.17. URL `https://aclanthology.org/2021.findings-emnlp.17`.

[24] 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 Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli, editors, *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing*

(Volume 1: Long Papers), pages 2505–2515, Online, August 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.acl-long.195. URL `https://aclanthology.org/2021.acl-long.195`.

[25] Yujian Gan, Xinyun Chen, and Matthew Purver. Exploring underexplored limitations of cross-domain text-to-SQL generalization. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih, editors, *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. doi: 10.18653/v1/2021.emnlp-main.702. URL `https://aclanthology.org/2021.emnlp-main.702`.

[26] Yujian Gan, Xinyun Chen, Jinxia Xie, Matthew Purver, John R. Woodward, John Drake, and Qiaofu Zhang. Natural SQL: Making SQL easier to infer from natural language specifications. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 2030–2042, Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.findings-emnlp.174. URL `https://aclanthology.org/2021.findings-emnlp.174`.

[27] Yujian Gan, Xinyun Chen, Qiuping Huang, and Matthew Purver. Measuring and improving compositional generalization in text-to-SQL via component alignment. In *Findings of the Association for Computational Linguistics: NAACL 2022*, pages 831–843, Seattle, United States, July 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.findings-naacl.62. URL `https://aclanthology.org/2022.findings-naacl.62`.

[28] Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. Text-to-sql empowered by large language models: A benchmark evaluation. *CoRR*, abs/2308.15363, 2023.

[29] Chunxi Guo, Zhiliang Tian, Jintao Tang, Pancheng Wang, Zhihua Wen, Kang Yang, and Ting Wang. A case-based reasoning framework for adaptive prompting in cross-domain text-to-sql, 2023.

[30] Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. Towards complex text-to-SQL in cross-domain database with intermediate representation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4524–4535, Florence, Italy, July 2019. Association for Computational Linguistics. doi: 10.18653/v1/P19-1444. URL `https://aclanthology.org/P19-1444`.

[31] Jiaqi Guo, Qian Liu, Jian-Guang Lou, Zhenwen Li, Xueqing Liu, Tao Xie, and Ting Liu. Benchmarking meaning representations in neural semantic parsing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1520–1540, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.118. URL `https://aclanthology.org/2020.emnlp-main.118`.

[32] Jonathan Herzig and Jonathan Berant. Span-based semantic parsing for compositional generalization. In Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli, editors, *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 908–921, Online, August 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.acl-long.74. URL `https://aclanthology.org/2021.acl-long.74`.

[33] Xiang Huang, Sitao Cheng, Yiheng Shu, Yuheng Bao, and Yuzhong Qu. Question decomposition tree for answering complex questions over knowledge bases, 2023.

[34] Wonseok Hwang, Jinyeung Yim, Seunghyun Park, and Minjoon Seo. A comprehensive exploration on wikisql with table-aware word contextualization. *CoRR*, abs/1902.01069, 2019. URL `http://arxiv.org/abs/1902.01069`.

[35] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy,

and Luke Zettlemoyer. Learning a neural semantic parser from user feedback. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 963–973, Vancouver, Canada, July 2017. Association for Computational Linguistics. doi: 10.18653/v1/P17-1089. URL `https://aclanthology.org/P17-1089`.

[36] Daniel Keysers, Nathanael Schärli, Nathan Scales, Hylke Buisman, Daniel Furrer, Sergii Kashubin, Nikola Momchev, Danila Sinopalnikov, Lukasz Stafiniak, Tibor Tihon, Dmitry Tsarkov, Xiao Wang, Marc van Zee, and Olivier Bousquet. Measuring compositional generalization: A comprehensive method on realistic data. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL `https://openreview.net/forum?id=SygcCnNKwr`.

[37] Najoung Kim and Tal Linzen. COGS: A compositional generalization challenge based on semantic interpretation. In Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 9087–9105, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.731. URL `https://aclanthology.org/2020.emnlp-main.731`.

[38] Brenden Lake and Marco Baroni. Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks. In *International conference on machine learning*, pages 2873–2882. PMLR, 2018.

[39] Wuwei Lan, Zhiguo Wang, Anuj Chauhan, Henghui Zhu, Alexander Hanbo Li, Jiang Guo, Shenmin Zhang, Chung-Wei Hang, Joseph Lilien, Yiqun Hu, Lin Pan, Mingwen Dong, J. Wang, Jiarong Jiang, Stephen M. Ash, Vittorio Castelli, Patrick Ng, and Bing Xiang. Unite: A unified benchmark for text-to-sql evaluation. *ArXiv*, abs/2305.16265, 2023.

[40] Dongjun Lee. Clause-wise and recursive decoding for complex and cross-domain text-to-SQL generation. In Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan, editors, *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 6045–6051, Hong Kong, China, November 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-1624. URL `https://aclanthology.org/D19-1624`.

[41] Gyubok Lee, Hyeonji Hwang, Seongsu Bae, Yeonsu Kwon, Woncheol Shin, Seongjun Yang, Minjoon Seo, Jong-Yeup Kim, and Edward Choi. Ehrsql: A practical text-to-sql benchmark for electronic health records. *Advances in Neural Information Processing Systems*, 35:15589–15601, 2022.

[42] Fei Li and H. V. Jagadish. Constructing an interactive natural language interface for relational databases. *Proc. VLDB Endow.*, 8(1):73–84, sep 2014. ISSN 2150-8097. doi: 10.14778/2735461.2735468. URL `https://doi.org/10.14778/2735461.2735468`.

[43] Jinyang Li, Binyuan Hui, Ge Qu, Binhua Li, Jiaxi Yang, Bowen Li, Bailin Wang, Bowen Qin, Rongyu Cao, Ruiying Geng, Nan Huo, Chenhao Ma, Kevin C. Chang, Fei Huang, Reynold Cheng, and Yongbin Li. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *ArXiv*, abs/2305.03111, 2023.

[44] Jinyang Li, Binyuan Hui, Ge Qu, Binhua Li, Jiaxi Yang, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin C. C. Chang, Fei Huang, Reynold Cheng, and Yongbin Li. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls, 2023.

[45] Linyang Li, Ruotian Ma, Qipeng Guo, Xiangyang Xue, and Xipeng Qiu. BERT-ATTACK: Adversarial attack against BERT using BERT. In Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu, editors,

Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP), pages 6193–6202, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.500. URL `https://aclanthology.org/2020.emnlp-main.500`.

[46] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. Gated graph sequence neural networks. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016. URL `http://arxiv.org/abs/1511.05493`.

[47] 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. doi: 10.18653/v1/2020.findings-emnlp.438. URL `https://aclanthology.org/2020.findings-emnlp.438`.

[48] Aiwei Liu, Xuming Hu, Lijie Wen, and Philip S. Yu. A comprehensive evaluation of chatgpt's zero-shot text-to-sql capability, 2023.

[49] Xiping Liu and Zhao Tan. Divide and prompt: Chain of thought prompting for text-to-sql, 2023.

[50] Moran Mahabi. Schema-grounded question decomposition and structured inference methods for text-to-sql. Master's thesis, Open University, 2024.

[51] Qingkai Min, Yuefeng Shi, and Yue Zhang. A pilot study for Chinese SQL semantic parsing. In Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan, editors, *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3652–3658, Hong Kong, China, November 2019. As-

sociation for Computational Linguistics. doi: 10.18653/v1/D19-1377. URL `https://aclanthology.org/D19-1377`.

[52] Sewon Min, Victor Zhong, Luke Zettlemoyer, and Hannaneh Hajishirzi. Multi-hop reading comprehension through question decomposition and rescoring. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 6097–6109, Florence, Italy, July 2019. Association for Computational Linguistics. doi: 10.18653/v1/P19-1613. URL `https://aclanthology.org/P19-1613`.

[53] Anssi Moisio, Mathias Creutz, and Mikko Kurimo. On using distribution-based compositionality assessment to evaluate compositional generalisation in machine translation, 2023.

[54] Linyong Nan, Yilun Zhao, Weijin Zou, Narutatsu Ri, Jaesung Tae, Ellen Zhang, Arman Cohan, and Dragomir Radev. Enhancing few-shot text-to-sql capabilities of large language models: A study on prompt design strategies, 2023.

[55] Yilin Niu, Fei Huang, Wei Liu, Jianwei Cui, Bin Wang, and Minlie Huang. Bridging the gap between synthetic and natural questions via sentence decomposition for semantic parsing. *Transactions of the Association for Computational Linguistics*, 11:367–383, 2023. doi: 10.1162/tacl_a_00552. URL `https://aclanthology.org/2023.tacl-1.22`.

[56] OpenAI. Gpt-4 technical report, 2023.

[57] Barbara Partee et al. Compositionality. *Varieties of formal semantics*, 3:281–311, 1984.

[58] Ethan Perez, Patrick Lewis, Wen-tau Yih, Kyunghyun Cho, and Douwe Kiela. Unsupervised question decomposition for question answering. In Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 8864–8880, Online, November 2020. Association for Computational Linguistics. doi:

10.18653/v1/2020.emnlp-main.713. URL `https://aclanthology.org/2020.emnlp-main.713`.

[59] 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. ISBN 1581135866. doi: 10.1145/604045.604070. URL `https://doi.org/10.1145/604045.604070`.

[60] Mohammadreza Pourreza and Davood Rafiei. Din-sql: Decomposed in-context learning of text-to-sql with self-correction. *arXiv preprint arXiv:2304.11015*, 2023.

[61] 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. URL `http://jmlr.org/papers/v21/20-074.html`.

[62] Nitarshan Rajkumar, Raymond Li, and Dzmitry Bahdanau. Evaluating the text-to-sql capabilities of large language models, 2022.

[63] Irina Saparina and Anton Osokin. SPARQLing database queries from intermediate question decompositions. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih, editors, *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8984–8998, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.708. URL `https://aclanthology.org/2021.emnlp-main.708`.

[64] Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. PICARD: Parsing incrementally for constrained auto-regressive decoding from language models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 9895–9901, Online and Punta

Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.779. URL `https://aclanthology.org/2021.emnlp-main.779`.

[65] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, SIGMOD '79, page 23–34, New York, NY, USA, 1979. Association for Computing Machinery. ISBN 089791001X. doi: 10.1145/582095.582099. URL `https://doi.org/10.1145/582095.582099`.

[66] Peter Shaw, Ming-Wei Chang, Panupong Pasupat, and Kristina Toutanova. Compositional generalization and natural language variation: Can a semantic parsing approach handle both? 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 922–938, Online, August 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.acl-long.75. URL `https://aclanthology.org/2021.acl-long.75`.

[67] Tom Sherborne and Mirella Lapata. Meta-learning a cross-lingual manifold for semantic parsing. *Transactions of the Association for Computational Linguistics*, 11:49–67, 2023. doi: 10.1162/tacl_a_00533. URL `https://aclanthology.org/2023.tacl-1.4`.

[68] Peng Shi, Linfeng Song, Lifeng Jin, Haitao Mi, He Bai, Jimmy Lin, and Dong Yu. Cross-lingual text-to-SQL semantic parsing with representation mixup. In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang, editors, *Findings of the Association for Computational Linguistics: EMNLP 2022*, pages 5296–5306, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.findings-emnlp.388. URL `https://aclanthology.org/2022.findings-emnlp.388`.

[69] Peng Shi, Rui Zhang, He Bai, and Jimmy Lin. XRICL: Cross-lingual retrieval-augmented in-context learning for cross-lingual text-to-SQL semantic parsing. In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang, editors, *Findings of the Association for Computational Linguistics: EMNLP 2022*, pages 5248–5259, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.findings-emnlp.384. URL `https://aclanthology.org/2022.findings-emnlp.384`.

[70] 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. doi: 10.3115/1117794.1117811. URL `https://aclanthology.org/W00-1317`.

[71] Anh Tuan Nguyen, Mai Hoang Dao, and Dat Quoc Nguyen. A pilot study of text-to-SQL semantic parsing for Vietnamese. In Trevor Cohn, Yulan He, and Yang Liu, editors, *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 4079–4085, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.364. URL `https://aclanthology.org/2020.findings-emnlp.364`.

[72] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc. ISBN 9781510860964.

[73] Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. RAT-SQL: Relation-aware schema encoding and linking for text-to-SQL parsers. In *Proceedings of the 58th Annual Meet-*

ing of the Association for Computational Linguistics, pages 7567–7578, Online, July 2020. Association for Computational Linguistics.

[74] Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2609–2634, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.147. URL `https://aclanthology.org/2023.acl-long.147`.

[75] David H.D. Warren and Fernando C.N. Pereira. An efficient easily adaptable system for interpreting natural language queries. *American Journal of Computational Linguistics*, 8(3-4):110–122, 1982. URL `https://aclanthology.org/J82-3002`.

[76] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 24824–24837. Curran Associates, Inc., 2022. URL `https://proceedings.neurips.cc/paper_files/paper/2022/file/9d5609613524ecf4f15af0f7b31abca4-Paper-Conference.pdf`.

[77] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, On-

line, October 2020. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/2020.emnlp-demos.6.

[78] Tomer Wolfson, Mor Geva, Ankit Gupta, Matt Gardner, Yoav Goldberg, Daniel Deutch, and Jonathan Berant. Break It Down: A Question Understanding Benchmark. *Transactions of the Association for Computational Linguistics*, 8:183–198, 04 2020. ISSN 2307-387X. doi: 10.1162/tacl_a_00309. URL https://doi.org/10.1162/tacl_a_00309.

[79] Tomer Wolfson, Daniel Deutch, and Jonathan Berant. Weakly supervised text-to-SQL parsing through question decomposition. In *Findings of the Association for Computational Linguistics: NAACL 2022*, pages 2528–2542, Seattle, United States, July 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.findings-naacl.193. URL https://aclanthology.org/2022.findings-naacl.193.

[80] William A Woods. Progress in natural language understanding: an application to lunar geology. In *Proceedings of the June 4-8, 1973, national computer conference and exposition*, pages 441–450, 1973.

[81] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation, 2016.

[82] Linting Xue, Noah Constant, Adam Roberts, Mihir Kale, Rami Al-Rfou, Aditya Siddhant, Aditya Barua, and Colin Raffel. mT5: A massively multilingual pre-trained text-to-text transformer. In Kristina Toutanova, Anna Rumshisky, Luke Zettlemoyer, Dilek Hakkani-Tur, Iz Beltagy, Steven Bethard, Ryan Cotterell, Tanmoy Chakraborty,

and Yichao Zhou, editors, *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 483–498, Online, June 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.naacl-main.41. URL `https://aclanthology.org/2021.naacl-main.41`.

[83] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. Sqlizer: Query synthesis from natural language. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017. doi: 10.1145/3133887. URL `https://doi-org.ezproxy.bgu.ac.il/10.1145/3133887`.

[84] Jingfeng Yang, Haoming Jiang, Qingyu Yin, Danqing Zhang, Bing Yin, and Diyi Yang. SEQZERO: Few-shot compositional semantic parsing with sequential prompts and zero-shot models. In Marine Carpuat, Marie-Catherine de Marneffe, and Ivan Vladimir Meza Ruiz, editors, *Findings of the Association for Computational Linguistics: NAACL 2022*, pages 49–60, Seattle, United States, July 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.findings-naacl.5. URL `https://aclanthology.org/2022.findings-naacl.5`.

[85] Yunhu Ye, Binyuan Hui, Min Yang, Binhua Li, Fei Huang, and Yongbin Li. Large language models are versatile decomposers: Decompose evidence and questions for table-based reasoning. *ArXiv*, abs/2301.13808, 2023.

[86] Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. In Regina Barzilay and Min-Yen Kan, editors, *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 440–450, Vancouver, Canada, July 2017. Association for Computational Linguistics. doi: 10.18653/v1/P17-1041. URL `https://aclanthology.org/P17-1041`.

[87] Tao Yu, Michihiro Yasunaga, Kai Yang, Rui Zhang, Dongxu Wang, Zifan Li, and Dragomir Radev. SyntaxSQLNet: Syntax tree networks for

complex and cross-domain text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1653–1663, Brussels, Belgium, October-November 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-1193. URL `https://aclanthology.org/D18-1193`.

[88] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921, Brussels, Belgium, October-November 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-1425. URL `https://aclanthology.org/D18-1425`.

[89] 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. ISBN 026251091X.

[90] Luke S. Zettlemoyer and Michael Collins. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *Proceedings of the Twenty-First Conference on Uncertainty in Artificial Intelligence*, UAI'05, page 658–666, Arlington, Virginia, USA, 2005. AUAI Press. ISBN 0974903914.

[91] Haoyu Zhang, Jingjing Cai, Jianjun Xu, and Ji Wang. Complex question decomposition for semantic parsing. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4477–4486, Florence, Italy, July 2019. Association for Computational Linguistics. doi: 10.18653/v1/P19-1440. URL `https://aclanthology.org/P19-1440`.

[92] Jiajie Zhang, Shulin Cao, Tingjia Zhang, Xin Lv, Jiaxin Shi, Qingwen Tian, Juanzi Li, and Lei Hou. Reasoning over hierarchical ques-

tion decomposition tree for explainable question answering. *ArXiv*, abs/2305.15056, 2023.

[93] Wenting Zhao, Konstantine Arkoudas, Weiqi Sun, and Claire Cardie. Compositional task-oriented parsing as abstractive question answering. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 4418–4427, Seattle, United States, July 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.naacl-main.328. URL `https://aclanthology.org/2022.naacl-main.328`.

[94] Victor Zhong, Caiming Xiong, and Richard Socher. Seq2sql: Generating structured queries from natural language using reinforcement learning. *CoRR*, abs/1709.00103, 2017.

[95] Ben Zhou, Kyle Richardson, Xiaodong Yu, and Dan Roth. Learning to decompose: Hypothetical question decomposition based on comparable texts. *ArXiv*, abs/2210.16865, 2022.

תקציר

משימת ההמרה מטקסט ל-SQL מהווה תחום משמעותי בעיבוד שפה טבעית (NLP) וזכתה לתשומת לב רבה בשל השלכותיה על אינטראקציית אדם-מחשב וניהול מסדי נתונים. משימה זו, של המרה משאלות בשפה טבעית ל-SQL (Structured Query Language), מקלה על התקשורת בין משתמשים שאינם מומחים לבין מסדי נתונים רלציוניים. חשיבותה של המשימה נובע משכיחותם של מסדי נתונים רלציוניים במערכות רבות בעולם, ומודלי טקסט ל-SQL מנגישים את העבודה מול מסדי נתונים אלו.

מעבר לחשיבותה המעשית, משימת ההמרה מטקסט ל-SQL היא משימה תיאורטית שימושית שכן היא מאפשרת לבחון לעומק ביצועים של מודלי NLP בתחום הבנת השפה. משתמש של מודל כזה יכול לנתח את הפלט המתקבל ולוודא תכונות רצויות של השאילתה, כגון פריטים מסוימים בסכימה (שמות טבלאות או עמודות), או פעולות Join מסוימות, זאת בנוסף לבדיקה האם תוצאת ההרצה של השאילתה אכן נכונה.

בעבודה זו, אנו עונים על שאלות המחקר הבאות: (1) איך להפוך את משימת ההמרה מטקסט ל-SQL יותר קומפוזיציונלית; (2) כיצד להפוך את הפלט של מודל המבצע את המשימה להיות יותר מובן למשתמשיו; (3) כיצד ניתן לגרום למודל המבצע את המשימה להיות עמיד יותר בפני שינויים קטנים בשאלות הקלט.

תרומתנו הראשונה מציגה את השפה QPL (Query Plan Language), מתודולוגיה חדשנית בהמרה מטקסט ל-SQL שבה אנו משתמשים בתוכניות הרצה של שאילתות SQL כשפת ביניים על-מנת לבדוק את ההשערה ששפת יעד מודלורית מובילה לתהליך

למידה יותר קומפוזיציונלי. המימוש של QPL כשלב ביניים מציע שיטה מתקדמת יותר לפרשנות שאלות משתמשים ויצירת פקודות SQL מתאימות, ובכך משפר את הדיוק והיעילות של עיבוד שאילתות במסדי נתונים.

אנו גם מגדירים את מקבץ הנתונים Spider-QPL ומבצעים ניסויים על שתי משימות שונות, המרת טקסט ל-SQL והמרת טקסט ל-QPL, תחת אותה ארכיטקטורה נוירונית, ורואים שמודלי שפה מאומנים מראש (pre-trained language models) יכולים להיות מכווננים (fine-tuned) להכללה קומפוזיציונלית טובה יותר בהמרה ל-QPL מאשר ל-SQL.

בתרומתנו השנייה, אנו מראים שמשתמשים עם ידע תכנותי מוגבל מוצאים את שפת QPL מובנת יותר מאשר SQL, במיוחד עבור שאילתות מורכבות. תכונה זו של QPL חשובה ביותר שכן היא מאפשרת השתתפות רחבה יותר של משתמשים עם מסדי נתונים.

שכיחותה של השפה האנגלית כשפה השלטת בכל הנוגע לעיצוב ולמבנה של מסדי נתונים, דהיינו, שמות הטבלאות והעמודות הינם באנגלית, מהווה אתגר משמעותי למשתמשים שאינם מומחים ושאינם שולטים בשפה האנגלית. תרומתנו השלישית חוקרת את תחום המרת טקסט ל-SQL במסגרת רב-לשונית. אנו חוקרים את המשימה של מענה על שאלות בעברית אשר פונות למסדי נתונים המוגדרים באנגלית בלבד. ניסויינו מצביעים על כך שההתקדמות האחרונה במודלי שפה גדולים מאפשר אימון של "ממירים" רב-לשוניים ע"י שימוש בשירותי תרגום אוטומטיים על שאלת המקור. השונות באוצר המילים המשמש לפנייה לעמודות וטבלאות במסדי הנתונים לאחר שהשאלה תורגמה מצביעה על כך שנחוצה עבודה נוספת על בעיית הקישור לסכימה (schema linking). בסדרת ניסויים, אנו מאשררים כי גם המודלים החדישים ביותר מראים רמת חוסן נמוכה במשימת ההמרה מטקסט ל-SQL ומראים כי התקפות תרגום "הלוך ושוב" (תרגום מאנגלית לשפת יעד וממנה חזרה לאנגלית) הן עדיין מאתגרות.

לסיכום, מחקר זה מקדם את הבנתנו של ניתוח סמנטי ע"י הדגשת החשיבות של הצבת שפת יעד פורמלית ומודולרית בעלת סמנטיקה אופרציונלית פשוטה ע"מ לתמוך בהכללה קומפוזיציונלית. תוצאות הניסויים שלנו מראות שגם כאשר מודלים מאומנים מגיעים לרמת דיוק גבוהה ביותר (מעל 90%) לפי אמות מידה סטנדרטיות של המרת טקסט ל-SQL, עלינו לשמור על רמת ביטחון נמוכה שתוצאת המודל אכן עונה על צרכי המשתמשים, שכן למשתמשים יש הבנה מוגבלת של שאילתות מורכבות. לבסוף, שאילתות רב-לשוניות והתקפות תרגום "הלוך ושוב" מדגימות את חשיבות המאמץ לשיפור היכולת של מודלים לזהות שונות לשונית, בדומה לדרכים הרבות והיצירתיות של אנשים להתייחס לאותו תוכן.

העבודה נעשתה בהדרכת פרופ' מיכאל אלחדד

במחלקה למדעי המחשב

בפקולטה למדעי הטבע

המרת טקסט ל-SQL: שימוש בתוכניות פעולה כשפת ביניים

מחקר לשם מילוי חלקי של הדרישות לקבלת תואר "דוקטור לפילוסופיה"

מאת

בן אייל

הוגש לסינאט אוניברסיטת בן גוריון בנגב

כ"א בשבט ה'תשפ"ד 31 בינואר 2024

באר-שבע