Natural Language Data Interfaces: A Data Access Odyssey

Georgia Koutrika
Athena Research Center, Athens, Greece

Abstract

Back in 1970's, E. F. Codd worked on a prototype of a natural language question and answer application that would sit on top of a relational database system. Soon, natural language interfaces for databases (NLIDBs) became the holy grail for the database community. Different approaches have been proposed from the database, machine learning and NLP communities. Interest in the topic has had its peaks and valleys. After a long and adventurous journey of almost 50 years, there is a rekindled interest in NLIDBs in recent years, fueled by the need for democratizing data access and by the recent advances in deep learning and natural language processing in particular. There is a surge of works on natural language interfaces for databases using neural translation, and suddenly it becomes hard to keep up with advancements in the field. Are we close to finding the holy grail of data access? What are the lurking challenges that we need to surpass and what research opportunities arise? Finally, what is the role of the database community?

2012 ACM Subject Classification Computing methodologies → Machine translation; Information systems → Data management systems

Keywords and phrases natural language data interfaces, NLIDBs, NL-to-SQL, text-to-SQL, conversational databases

Digital Object Identifier 10.4230/LIPIcs.ICDT.2024.1

Category Invited Talk

1 Introduction

E. F. Codd, the father of relational databases, said that "*If we are to satisfy the needs of casual users of databases, we must break the barriers that presently prevent these users from freely employing their native language*" [11]. Throughout the 1970s, he worked on a prototype of a natural language question and answer application that would sit on top of a relational database system, called Rendezvous. Rendezvous allowed a user with no knowledge of database systems – and even limited knowledge of a given database's content – to engage in a dialog with the system. Almost 50 years after, natural language interfaces for databases (NLIDBs) are into the spotlight.

NLIDBs or *natural language data interfaces* are appealing for a number of reasons [3]. They are more suitable for occasional users, alleviating the need for the user to spend time learning the system's query language to access data. Some questions are easier expressed in natural language (e.g., questions involving negation, or quantification). Moreover, natural language (NL) questions can be brief and support anaphoric and elliptical expressions, where the meaning of each question is complemented by the discourse context.

The first NLIDBs appeared back in the sixties. For instance, LADDER was developed as a management aid to Navy decision makers [31]. These early systems interfaced application-specific non-SQL database systems, and they could not be used with different data. Several approaches have followed over the years focusing on translating NL questions to SQL over relational data. Industrial systems also made their appearance. For example, in the late 90's, Microsoft SQL Server shipped with the English Query feature. Some systems enabled keyword searches. They relied on data indexes to find relations that contained the query

keywords and on the database schema to join them and return the answer to a query (e.g., [32, 51]). Parsing-based approaches parsed the input question to understand its grammatical structure and then map it to the structure of the desired SQL query (e.g., [47, 77]).

Recently, the use of deep learning techniques, and in particular LLMs, has given a great boost in the development of NLIDBs [22, 27, 38, 48, 56, 86]. The creation of two large datasets, WikiSQL [86] and Spider [81], for training NLIDBs has brought several developments in this field, with new systems popping up like mushrooms. These have popularized the term *Text-to-SQL* (or NL-to-SQL) to refer to NLIDBs that focus on translating NL questions to SQL. For the first time, it seems possible that technological barriers can be broken and human-like interaction with data can become a reality.

2 The Inherent Challenges of Natural Language

While using natural language as a query language appears very appealing, it also brings challenges in query understanding and answering that standard database query languages, such as SQL, are free of, by design. These challenges, however, plague NLIDBs.

Ambiguity. Natural language is ambiguous for human-computer interaction. Ambiguity allows more than one interpretation. Unfortunately, there are several types of ambiguity that a NLIDB needs to resolve [25]. For example, *"Paris"* may refer to the city or a person (lexical ambiguity). The question *"Find all German movie directors"* can be parsed into *"directors that have directed German movies"* or *"directors from Germany that have directed a movie"* (syntactic ambiguity). The question "*Are Brad and Angelina married*?" is an example of semantic ambiguity, as it is unsure if it means they are married to each other or separately. Context-dependent ambiguity refers to a term having different meanings in different contexts. For example, *"top"* in *"top scorer"* means the highest (total) number of goals, while in *"top movies"*, it signifies the greatest rating. As a result of ambiguity, there may be multiple potential interpretations of a natural language query. Generating and processing them takes a toll on the system efficiency. It also affects the system's effectiveness, i.e., its ability to find which interpretation captures the user intent and correctly answer the user query.

Paraphrasing. Completely different words or sentences can have the same meaning. For instance, *"How many people live in Amsterdam?"* and *"What is the population of Amsterdam?"*. Dealing with different NL utterances (paraphrasing) is a challenge, as each one may need different handling. For instance, the second NL query may be easier for a system because it is likely that a *population* attribute exists in the database schema.

Inference. A natural language sentence may not contain all information needed for a system to fully understand it. In *elliptical queries*, one or more words are omitted but can still be understood in the context of the sentence. An example is *"Who was the president before Obama"*. The fact that the query refers to US presidents needs to be inferred. In *follow-up questions*, which are common in conversations between humans, missing information can be understood in the context of the dialog. We ask a question, receive an answer, and then ask a follow-up question assuming that the context of the first question is known. For example, *"Q: Which is the capital of Germany?"*, *"A: Berlin"*, *"Q: What about France?"*. The system has to infer that the user is asking for the capital of France.

User mistakes. Spelling, syntactical or grammatical errors can be understood by a human but are hard to be recognized and ignored by a system. For example, if the user asks for "*movies by Brad Bitt*", can the system recognize that the user refers to Brad Pitt, and make the necessary correction, and not to a Brad Bitt that the system does not happen to know based on the underlying data?

3 A trip down Memory Lane

Prototype NLIDBs appeared in the late sixties and early seventies [12, 63]. For example, Lunar [74] was a natural language interface to a database with chemical analyses of moon rocks. LADDER [31] used semantic grammars, a technique that interleaves syntactic and semantic processing. Early eighties witnessed a lot of research on NLIDBS [3]. For instance, Chat-80 was implemented entirely in Prolog [72]. It transformed English questions into Prolog expressions, which were evaluated against a Prolog database. A number of commercial systems, such as IBM's LanguageAccess [53] and Intellect [28] from Trinzic, appeared.

Some of the early systems relied on *pattern-matching* techniques to answer the user's questions (e.g., [36]). A primitive pattern-matching system could use rules like:

pattern: ... "*capital*" ... <*country*>
action : *Report Capital of row where Country* = <*country*>

This rule says that if a user's request contains the word "*capital*" followed by a country name (i.e., a name appearing in the Country column), then the system should locate the row which contains the country name, and print the corresponding capital.

Syntax-based systems used a grammar that described the possible syntactic structures of the user's questions (e.g., Lunar [74]). A user question would be first parsed (i.e. analysed syntactically) resulting in a syntax tree like the one shown in Figure 1. Then, the resulting parse tree would be directly mapped to a query in some database query language. To perform this mapping, the system would use specific mapping rules that would specify how each part of the tree would map to a part of the database query. These syntax-based NLIDBs usually interfaced to application-specific database systems that provided database query languages carefully designed to facilitate the mapping from the parse tree to the database query. At this point, it was usually difficult to devise mapping rules to transform directly the parse tree into some expression in a real-life database query language (e.g. SQL) [3].

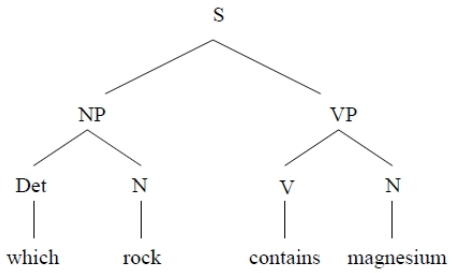

Figure 1 An example syntax tree from Lunar [3].

In *semantic-grammar systems* (e.g., LADDER [31], EUFID [63]), the question-answering is still done by parsing the input and mapping the parse tree to a database query. The difference is that the grammar categories (i.e., the non-leaf nodes that will appear in the parse tree) would correspond to semantic concepts (e.g., Substance, Radiation, or Specimen) instead of syntactic constituents (e.g., noun-phrase, noun, sentence). An example tree is

shown in Figure 2. Semantic grammars contain hard-wired knowledge about a specific knowledge domain, and semantic grammar categories are usually chosen to enforce semantic constraints. Since the grammar is domain-specific, such systems are very difficult to port to other knowledge domains.

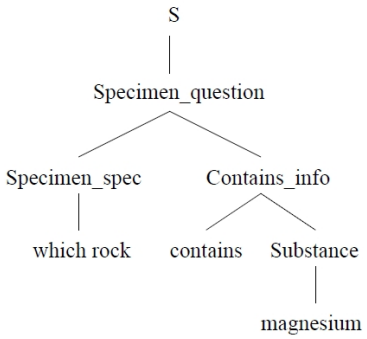

Figure 2 An example semantic tree [3].

Most recent systems in that period would first transform the natural language question into an *intermediate logical query*, that expresses the meaning of the user question in terms of high-level world concepts, which are independent of the database structure. The logical query is then translated to an expression in the database query language (e.g., CLE [2], TEAM [26]). For example, the Essex system [15] used a principled multi-stage transformation process: the system first generated a logic query, expressed in a version of untyped λ-calculus, which was then transformed into a first-order predicate logic expression, which was subsequently translated into universal-domain relational calculus, domain relational calculus, tuple relational calculus, and finally SQL.

In many systems, the syntax rules linking non-terminal symbols (non-leaf nodes in the parse tree) and the corresponding semantic rules are domain-independent; i.e., they could be used in any application domain. The information, however, describing the possible words (leaf nodes) and the logic expressions corresponding to the possible words is domain-dependent, and has to be declared in the lexicon. The logic query does not refer to database objects (e.g. tables, columns), and it does not specify how to search the database to retrieve the necessary information. In order to retrieve the information requested by the user, the logic query has to be transformed into a query expressed in some database query language supported by the underlying DBMS, using the mapping to database information.

Figure 3 captures the evolution of systems during this period. In early NLIDBs, the syntax/semantics rules were based on rather ad hoc ideas, and expressed in idiosyncratic formalisms. In the nineties and later, systems follow some representation-based approach and adopt advances in the general natural language processing field, for better syntactic and semantic parsing and interpretation.

4 The Database-Way Era

After 2000, the next 15 years witness the emergence of systems that focus on translating keyword or NL queries to SQL. They adopt the general architecture shown in Figure 4 [25].

The *parser* is responsible for gathering linguistic information on the user query. Its output varies in complexity ranging from a simple set containing meaningful keywords (e.g., *movie, actor, "Brad Pitt"*) (e.g., Discover [33]), to a fully structured parse tree (e.g., NaLIR[47]).

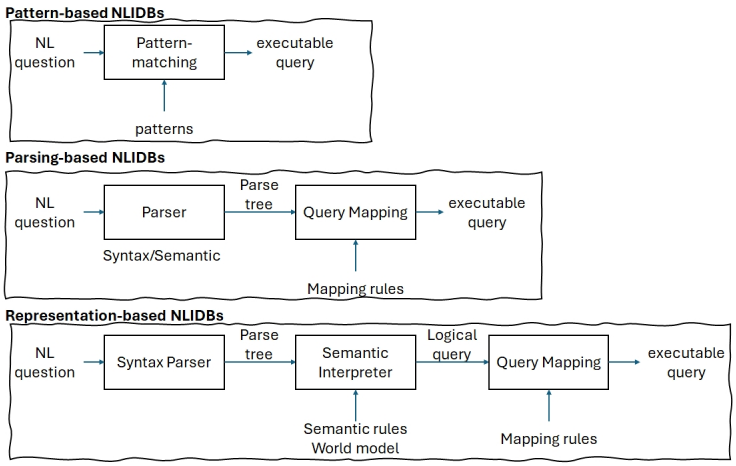

Figure 3 Early NLIDBs system architectures.

The *entity mapper* is responsible for mapping the terms extracted by the Parser to database elements. Some systems use inverted indexes to map the query terms to values in the database (e.g., Discover [33]). Others also search the database metadata, such as relation and attribute names (e.g., DiscoverIR[32]). NaLIR [47] also identifies the nodes in the parse tree that can be mapped to SQL elements (select, operator, function, quantifier and logic nodes) using a knowledge base of phrases – for example, the function node *COUNT* corresponds to the phrase *"number of"*.

The *interpretation generator* infers the semantics of the query as a whole. Dependencies between terms and phrases are analyzed in order to extract joins, aggregate functions, comparisons, etc., depending on the operations that every NLIDB system supports. The output is an intermediate well-defined structured format that captures the meaning of the query and the order of the operations to be done. This intermediate representation is an *interpretation* of the query from the system's perspective. Intermediate representations are useful because they are easier to modify, manipulate, and rank, as opposed to SQL queries. Ambiguity at the term level can be transferred from the output of the entity mapper, due to multiple mappings for a term. Ambiguity also exists in the linguistic dependencies between terms, in the extraction of joins, in the order of the operations, and so forth. As a result, the output of this step may be *multiple candidate interpretations* ranked according to how well the system thinks an interpretation captures the user intent.

Indexes and knowledge bases are used by the entity mapper and interpretation generator. The database inverted indexes are used to map input terms extracted by the parser to database values. Knowledge bases are additional sources that a system can exploit to understand the query, such as dictionaries containing word definitions and context-specific definitions, synonym lists, grammar rules and syntactic patterns.

The *SQL translator & executor* translates the intermediate interpretation to SQL. This task heavily depends on the quality of the highest ranked interpretations, since only a selected few are executed. Each system has its own set of rules to convert an intermediate representation into a syntactically correct SQL query. Some systems may rank the results returned by every query, which means that the tuples returned by one query do not always adopt the score of that query that produced them, but may have their own ranking.

These approaches bring the following notable novelties (a detailed experimental evaluation of representative systems sheds light into their capabilities [25]):

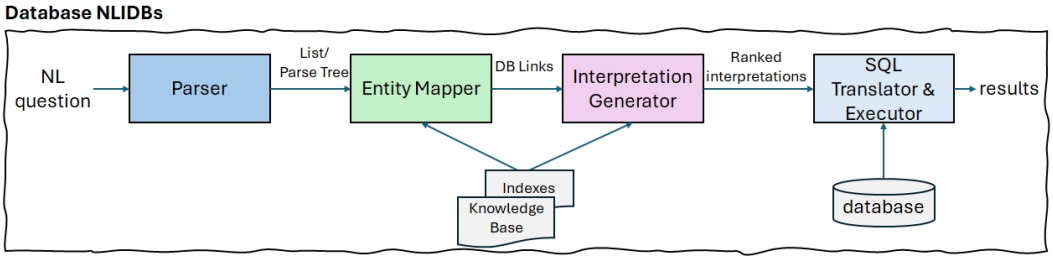

Figure 4 Database NLIDBs system architecture.

(1) A database mapping problem. Query translation is viewed as a database mapping problem: mapping query terms to database elements (tables, columns and values) and finding the desired interconnections of these data elements that capture the user intent. For this reason, the entity mapper, which maps the query terms to database elements, comes right after the parser as a crucial step towards understanding the query from the perspective of the database. Together, they ensure that the final SQL will be *semantically correct*. Note that, in earlier, representation-based, systems, this mapping would happen at a later stage through the semantic interpreter (using rules and some model of the world). Discover [33] introduced the concept of *generating query interpretations as subgraphs of the database schema graph*, called *joining networks* or (trees), adopted by several systems (e.g., DiscoverIR [32], Spark [51], SODA [4]). The interpretation generator constructs multiple joining networks that connect the relations that contain the query terms through edges that are the primary key-foreign key relationships between them. Précis [41, 62] extended the concept of joining networks and introduced the *logical database subset* that contains not only items directly related to the given query keywords but also items implicitly related to them, with the purpose of providing to the user greater insight into the data. ATHENA [57] translated the input natural language query (NLQ) into an intermediate query language over a domain ontology, which was subsequently translated into SQL.

(2) Query disambiguation. Dealing with the ambiguity of natural language is important for these systems. Query disambiguation is performed at two levels: interpreting a single term (entity mapper), and interpreting the whole query (interpretation generator). A query term may have *multiple* potential mappings to database elements. The entity mapper may assign a score to each possible mapping – for example, an information retrieval-style score (e.g., [32]) – in order to distinguish between likely and not likely mappings. On the other hand, terms may not map directly to any database element due to synonyms and non-exact matches. The system may attempt to find the closest mappings using a knowledge base with synonyms (e.g., SODA [4]), or a string similarity algorithm (e.g., NaLIR[47]). To minimize the chances of ambiguity and properly interpret the query intent, some systems impose stricter syntactic constraints on their input (e.g., [4], [83]). The position of the keywords, functions and operators matters and minor changes can alter the semantics or even render the query incomprehensible. For example, a query could be "*count movies actor "Brad Pitt"*". Using this approach, ExpressQ [83] was the first system to accept keyword queries with aggregate functions and groupBy. The interpretation generator may rank query interpretations using simple ranking schemes, like the size of the joining network (Discover [33]), the sum of term scores normalized by the tree size ([32]) or a function of the edge weights [62]. More elaborated formulas have been used. For example, Spark [51] models a joining tree as a *virtual document*, and computes its score as a function of an information retrieval score, a

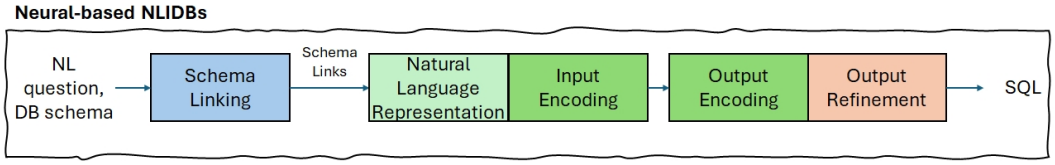

Figure 5 Neural NLIDBs system architecture.

completeness factor that quantifies how many different query keywords are included in the tree, and the tree size. NaLIR [47] ranks query interpretations based on: (i) the number of parse tree nodes that violate the grammar, (ii) the mappings between the parse tree nodes and the database elements that can help infer the desired structure of a parse tree, and (iii) the similarity to the original linguistic parse tree.

(3) Execution-coupled translation. A characteristic of these systems is that they are responsible not just for translating the query but also for generating the final response to the query. The SQL translator & executor ensures the *syntactic correctness of the generated SQL queries*. The response is ranked according to how well it matches the user query. A lot of effort has been put into the execution algorithm, aiming to return the top-k results faster. To achieve this, heuristics are applied to stop the execution without materializing all the possible joining trees (e.g., DiscoverIR [32]), or accessing fewer parts of the data (Spark [51]). Précis [41] progressively populates the relations in a logical subset to reduce the number of database rows fetched to the ones needed in the answer. ⌟

Early systems of this period are keyword-based, adopting the fundamental characteristic of search engines [1]. NaLIR [47] supported NL queries and used the Stanford Parser [14] to generate a dependency parse tree, where nodes represent the terms and edges represent the linguistic relationships between them. Its interpretation generator corrected any possible flaws in the structure of the parse tree to make it grammatically valid. ATHENA++ [59] combines linguistic patterns from NL queries with deep domain reasoning using ontologies to enable nested query detection and generation for specific domains.

5 The Deep Learning Era

If in the end of the nineties, "*the development of NLIDBs is no longer as fashionable a topic within academic research*" [3], after 2017, there has been an explosion of works on NLIDBs [38, 37]. These approaches tackle the text-to-SQL problem as a *language translation problem*, and train a neural network on a large amount of {NL query/SQL} pairs [38].

Figure 5 shows the architecture of a neural NLIDB [38]. In its core lies the neural network that typically follows an encoder-decoder architecture. The encoder takes one or more inputs of variable shapes and transforms them into one or more internal representations with fixed shapes that are consumed by the decoder. Additionally, the encoder usually infuses the representation of each input with information from the rest of the inputs, so as to create a more informed representation that better captures the instance of the problem at hand, through a mechanism called neural attention. The decoder uses the representations calculated by the encoder and makes predictions on the most probable SQL query (or parts of it).

Given that the inputs of an NLIDB are mainly textual, the *natural language representation* creates an efficient numerical representation of the input that can be accepted by the encoder. Early systems used pre-trained word embeddings for NL representation, such as GloVe [54] (e.g., SQLNet [75], IncSQL [61]). Recent systems rely on Pre-trained Language Models (PLMs) such as BERT [16] (e.g., HydraNet [52], ValueNet [7]), that provide better representations.

Apart from the NL question, the inputs to an NLIDB may include table and column names, values, primary-to-foreign key relationships and relationships between columns and tables. *Input encoding* structures the inputs so that the encoder can process them. Options range from encoding each column with the NL query separately as in HydraNet [52] to schema graph encoding (e.g., RAT-SQL [68]). *Output decoding* consists of designing the structure of the predictions that the network will make, as well as choosing the appropriate network for making such predictions (e.g., a SQL query can be viewed as a simple string, or as a structured program which follows a certain grammar).

Neural NLIDBs have the following notable features (see [38] for a detailed survey):

(1) A neural translation problem. Viewing the problem as a language translation one, neural text-to-SQL systems take as input a NL query and return a SQL query. Depending on how their decoder generates the output, they can be divided into three categories [10]: (*a*) sequence-based, (*b*) sketch-based slot-filling, and (*c*) grammar-based approaches.

Sequence-based approaches generate the predicted SQL, or a large part of it, as a sequence of words (comprising SQL tokens and schema elements) [8, 50, 86]. Seq2SQL [86] was one of the first neural networks created specifically for the text-to-SQL task and was based on a previous work focusing on generating logical forms using neural networks [17]. The system predicts an aggregation function and the column for the SELECT clause as classification tasks and generates the WHERE condition clause using a seq-to-seq network. The latter network is burdened with generating parts of the query and can lead to syntactic errors, which is its major drawback. The network architecture combines LSTM and linear layers, and its inputs are represented as GloVe embeddings. More recent sequence-based approaches are more effective thanks to the use of large pre-trained seq-to-seq Transformer [67] models (e.g., T5 [55], BART [46]) and the use of smarter decoding techniques that constrain the predictions of the decoder and prevent it from producing invalid queries (e.g., PICARD [58]).

Sketch-based slot-filling approaches (e.g., XSQL [30], SQLOVA [35], HydraNet [52], SQLNet [75]) consider a query sketch with a number of empty slots that must be filled in, and use neural networks to predict the most probable elements for each slot, such as the table columns that appear in the SELECT clause. In this way, the SQL generation task is transformed into a set of classification tasks. SQLNet [75], one of the first sketch-based approaches, was based on the observation that the way Seq2SQL [86] chose to generate the WHERE clause was prone to errors that could be avoided. For this reason, a set of query sketches were developed and separate neural networks were created to fill each type of slot. While dividing the text-to-SQL problem into small sub-tasks makes it easier to generate syntactically correct queries, sketch-based approaches have two drawbacks. Firstly, the resulting neural network architecture may end up being quite complex since dedicated networks may be used for each slot or part of the query. Furthermore, it is hard to extend to complex SQL queries, because generating sketches for any type of SQL query is not trivial.

Grammar-based approaches (e.g., RyanSQL [10], IRNet [27], IncSQL [61], Rat-SQL [68]) produce a sequence of grammar rules instead of simple tokens in their output. These grammar rules are instructions that, when applied, can create a structured query. The most often used grammar-based decoders by text-to-SQL systems have been previously proposed for code generation as an Abstract Syntax Tree (AST) [78, 79]. These models take into account the grammar of the target code language (in our case, the SQL grammar) and consider the target program to be an AST, whose nodes are expanded at every tree level using the grammar rules, until all branches reach a terminal rule. When it reaches a terminal rule, the model might generate a token, for example, a table name, an operator or a condition value, in the

case of text-to-SQL. The decoder uses a LSTM-based architecture that predicts a sequence of actions, where each action is the next rule to apply to the program AST. Because the available predictions are based both on the given grammar and the current state of the AST, the possibility of generating a grammatically incorrect query is greatly reduced.

(2) Learning schema linking. Schema linking aims at the discovery of possible mentions of database elements in the NL question. For this purpose, query candidates are extracted from the question and are matched to database candidates from the underlying database. Query candidates may be single words, n-grams (IRNet [27]), or named entities (ValueNet [7], TypeSQL [80]). Database candidates are tables, columns, and values stored in the database. To accelerate the search of discovered query candidates in the database values, indexes have been widely used in earlier, non-neural, text-to-SQL systems [32, 47]. ValueNet [7] also adopts this approach. The second part of schema linking is the process of mapping the query candidates to database candidates. Each mapping is called a *schema link*. These discovered schema links are fed into the neural network that is responsible for the translation.

The mapping can be performed using exact or partial matching (IRNet [27]) and approximate string matching (ValueNet [7]). Text-to-SQL systems [5, 6] have also used learned word embeddings from the area of semantic parsing [42]. The system learns word embeddings using the words of the text-to-SQL training corpus and combines them with additional features that are calculated using NER, edit distance and indicators for exact token and lemma match. These embeddings are then used to calculate the similarity of query candidates to DB candidates. While this approach is expensive, it allows for more flexible and intelligent matching. Given the complexity of schema linking, it is also possible to train a model to perform schema linking with better results. For example, a Conditional Random Field (CRF) model [43] can be trained on a small group of hand-labelled samples to recognize column links, table links and value links for numerical and textual values [8].

SDSQL [34] follows a very different approach. It is simultaneously trained on two tasks: (*a*) the text-to-SQL task, similarly to all systems, and (*b*) the *Schema Dependency Learning* task for discovering schema links using a deep biaffine network [18, 19]. In this case, the schema links discovered by the system are not directly used for predicting the SQL query; still, training for both tasks simultaneously has a positive effect on the system performance.

Neural Attention. While attention layers do not directly determine a match, they can highlight connections between query and DB candidates, which can improve the system's internal representation and boost its performance. SQLNet [75] was the first system to introduce such a mechanism, named *Column Attention*, that processes the NLQ and column names and finds relevant columns for each word of the NLQ. PLMs based on the Transformer neural architecture [67], which encapsulates an attention mechanism, have become very popular for input encoding, greatly benefiting the accuracy of text-to-SQL systems (e.g., [58]). RAT-SQL [68] proposed a modified Transformer layer, called Relation-Aware Transformer (RAT), that biases the attention mechanism of the Transformer towards already-known relations from the DB schema and discovered schema links.

(3) Output refinement. Output refinement can be applied on a trained model to avoid producing incorrect SQL queries. *Execution-guided decoding* [70] can execute partially complete SQL queries at prediction time and decide to avoid a certain prediction if the execution fails or if it returns an empty output. Execution-guided decoding is system-agnostic and can increase the system accuracy. *Constrained decoding* is a method for incrementally parsing and constraining auto-regressive decoders, to prevent them from

producing grammatical or syntactical errors (PICARD [58]). For each token prediction, PICARD examines the generated sequence so far along with the k most probable next tokens and discards all tokens that would produce a grammatically incorrect SQL query, use an attribute that is not present in the DB at hand, or use a table column without having its table in the query scope. Other systems with sequence-based decoders have proposed similar decoding techniques to avoid errors (e.g., SeaD [76] and BRIDGE [50]).

(4) Datasets & evaluation. A *text-to-SQL dataset* (or *benchmark*) is a set of NL/SQL query pairs defined over one or more databases, used to train and evaluate neural text-to-SQL systems. Text-to-SQL datasets become a critical asset due to their integral role in enabling the development of such systems and serving as a common reference for evaluation. Previous, non-neural systems did not use common datasets, instead they employed a variety of small datasets that combined different databases and query sets of varying size and complexity. The lack of a common dataset to be used by different system evaluations impeded a fair system comparison and a clear view of the text-to-SQL landscape in previous years. This situation drastically changes with the emergence of WikiSQL [86] and Spider [81], in 2017 and 2018 respectively. These are the first large-scale, multi-domain benchmarks that made it possible to train and evaluate neural text-to-SQL systems and provided a common tool to compare different systems easily. While other benchmarks have followed (e.g., [9, 24, 44, 49, 84]), these two remain the most popular ones. ⌟

The first neural NLIDBs (e.g., [75, 86]) could generate simple queries over single tables of the WikiSQL dataset. Recent systems [7, 27, 68] can generate complex SQL queries over relational databases and achieve high performance scores on Spider. Schema linking is not always an explicit component of a neural NLIDB. Systems such as Seq2SQL [86], SQLNet [75], and HydraNet [52] do not perform schema linking, relying mainly on the neural network to correctly translate the NL question over the underlying data. However, it is not clear whether pre-trained neural architectures defy the need for schema linking. On the other hand, there is little evidence on how fast and scalable schema linking approaches are, especially for very large databases, with the exception of DBTagger [66] that provides experimental insights into the time and memory requirements of its schema linking approach.

Most systems do not ensure that the generated SQL is syntactically correct (e.g., Seq2SQL [86], SQLNet [75], HydraNet [52], IncSQL [61], RyanSQL [10]). Output refinement can be applied on a trained model to avoid producing incorrect SQL queries. However, it adds an additional burden to the system and increases the time needed to generate a SQL query. Its effectiveness versus the incurred overhead are yet to be studied.

The remarkable rise of neural NLIBDs has not been followed by their widespread adoption in commercial products yet. There are many obstacles on the way to deliver the promise of truly enabling accessing data using natural language. A significant one is their view of the text-to-SQL problem as a language translation problem and their focus on translation accuracy over a specific dataset [29]. This is one side of the coin though, as we explain next.

6 Not (just) a Language Translation Problem: The Role of SQL and the Database Schema

Even if a system eliminated all linguistic problems and could perfectly understand a NL query, additional challenges stem from the *SQL expressivity* and the *schema of the data*.

The language SQL was developed by IBM, as a human-friendly way to query relational data. However, SQL is a structured language with a strict grammar, which leads to limited expressivity when compared to natural language or other programming languages. For

example, it lacks constructs for iteration and recursion, amongst other things. There are queries that are easy to express in natural language, but their respective SQL is more complex and less intuitive. For example, the query *"Return the movie with the best rating"* may be mapped to a nested SQL query. While the original NL query is simple, building the complex SQL query may be challenging for the system. Full natural language is more expressive than single SQL queries. There are computationally reasonable queries that might be expressed in NL, but which cannot be answered by a single SQL query. Finally, while a sentence in natural language may contain mistakes, and still be understood by a human, a SQL query needs to be syntactically and semantically correct to be executable over the underlying database.

The database schema also poses challenges. As one instance, a query like *"Who is the director of "Beautiful Mind"* may hide implicit join operations due to database normalization. Moreover, similar NL queries may need different processing on two different database schemas. For example, in a university database, every person is either a Student or a Faculty member, so these comprise two relations. On the other hand, movies have several genres that cannot be stored as different tables. They are stored in a Genre relation and are connected with movies through a many-to-many relationship. As a result, similar queries, such as *"comedies released in 2018"* and *"students enrolled in 2018"* are handled differently. In the former case, the system maps *"comedies"* to a value in the *Genre* table and joins it with the *Movie* table whereas it just maps *"students"* to the *Student* relation in the latter case.

All these challenges make the text-to-SQL problem challenging. Not only it is difficult to understand a NL query but it is also difficult to build the correct SQL query. Perhaps even more critically, similar questions may lead to a different outcome over different databases: one question may be successfully translated over one database and the other may not, due to issues such as ambiguity, paraphrasing, and different database schemas. Neural approaches addressing the problem as purely a language translation one ignore the particularities of the problem that stem from SQL and the database schema. Not surprisingly though, database approaches that view the text-to-SQL problem as mainly a database mapping one, also fall short in appreciating its multi-dimensional nature. Both perspectives, neural and database, have their merits to consider along their shortcomings in the quest of better NLIDBs.

7 Where we Stand: Limitations and Lessons Learnt

The systems that approach the text-to-SQL as a database mapping problem provide a more grounded solution that leverages the underlying data and relationships to intrepret the NL question and build an executable SQL query. They also include explicit methods for dealing with query ambiguity both at the level of query terms as well as at the level of the query. However, existing approaches struggle with more complex and diverse NL queries and cannot easily cope with NL challenges, such as synonyms, paraphrasing and typos [25].

On the other hand, neural NLIDBs, often completely ignoring the underlying data, promise to be more generalizable both in terms of the different types of NL queries the methods can understand as well as the different databases they can work on thanks to their extensive training. However, in practice, existing approaches focus on limited-scope problems and their accuracy severely degrades with more complex and diverse NL and SQL queries as well as complex databases [39]. They depend on training data and cannot cope with unseen databases and queries. For example, Spider [81], which is very popular for training and evaluating text-to-SQL systems, contains queries over 200 relational databases from 138 different domains. These are toy databases with simple schemas and small sizes not resembling real-world databases. Moreover, most neural approaches support size-limiting

input representations that cannot possibly leverage the wealth of a real-world database comprising hundreds of tables and attributes. These limitations become highly relevant when applying a text-to-SQL system to an actual database [29] used in a business, research or any other real-world use case. Such databases can pose difficulties not encountered in the datasets used to train and evaluate such systems, for example, a large number of tables and attributes and table and column names that use domain-specific terminology. Last but not least, models used so far are typically quite large, questioning their practical use [1].

8 The Challenges of NLIDBs

The challenges coming from the NL side and the challenges from the SQL and database side combined create a set of novel and unique to NLIDBs challenges that haunt current efforts.

Query answerability: How to tell if the NL question can be answered or not and why. For SQL queries, the answerability question can be answered deterministically based on two checks: syntactic and semantic correctness. When a database receives a SQL query, the parser checks whether the query adheres to the SQL syntax (*syntactic parsing*) and whether it contains tables and columns that exist in the underlying database (*semantic parsing*). Both checks are easy, fast, and conclusive. If the query contains any (syntactic or semantic) error, the system stops query processing and informs the user of the exact problem that makes the query unanswerable. The user can correct the query accordingly. In a NLIDB, the query answerability question is more convoluted: *does the NL question map to SQL that captures the query intent and is executable over the underlying database*? It is hard to define a set of comprehensive checks needed but it is also hard to answer them always successfully.

Two important checks are still (a) semantic correctness, meaning that the concepts mentioned in the query can be mapped to database columns and tables, and (b) syntactic correctness of the generated SQL. In database approaches to NLIDBs, the entity mapper performs the semantic parsing while the interpretation generator is responsible for the syntactic parsing since it makes sure that syntactically correct SQL queries are generated. Due to issues such as the language ambiguity, and the linguistic and conceptual gap between how the user formulates the query and how the system stores the data, *NLIDBs may fail to recognize that a question is answerable*. For instance, for the query *"How many people live in Amsterdam?"*, a NLIDB may not link the *population* attribute in the database schema to "*How many people live*". A *NLIDB may also fail to recognize that a NL question cannot be answered*, and still try to generate a SQL query. These problems are even more pronounced in neural NLIDBs. Especially those that do not employ schema linking and output refinement techniques are more prone to generate non-executable queries.

For NL questions, there have been some attempts to define query answerability devising categories of unanswerable questions [69, 85]. For example, the *calculation unanswerable category* requires mapping the concept mentioned in the user question to composite operations over existing table columns that are not known SQL operations or even user-defined functions [69]. The *out-of-scope category* means that the question is out of SQL's operation scope, such as when the user requests for charts [69]. *Improper to DB* refers to questions such as small talk or asking-opinion questions that are not proper to any databases [85].

Neural end-to-end parsing models ignore modeling questions in a fine-grained manner, which results in an inability to precisely detect and locate the specific reasons for unanswerable questions [69]. Furthermore, most existing text-to-SQL datasets used for training lack

[1] The training cost as well as the energy consumption [60] of such big models are important concerns.

ambiguous and unanswerable questions (e.g. Spider [81]) or if they contain, they are not marked in a way that the system can recognize them and learn accordingly (e.g., WikiSQL [86]). A NLIDB parsing system should detect answerable and unanswerable questions, and it should locate the specific reasons and generate corresponding explanations to guide the user in rectification. To that end, some recent efforts have emerged [82, 69]. For example, a weakly supervised DTE (Detecting-Then-Explaining) model for error detection, localization, and explanation [69] is trained on a dataset called TRIAGESQL [85].

Query coverage: What is the set of NL questions allowed. A NLIDB's query capabilities are not obvious to the user [25, 64]. Users find it difficult to understand what kinds of questions the system can or cannot cope with. It is often not clear to the user whether a rejected question is outside the system's query coverage (what types of NL statements it can recognize as queries that can be mapped to SQL), or whether it is outside the system's conceptual coverage (how the data is actually stored). Users are often forced to rephrase their questions, until a form the system can understand is reached. In other cases, users never try questions the NLIDB could in principle handle, because they think the questions are outside the subset of natural language supported by the NLIDB. To frame the query coverage of the system, some NLIDBs explicitly restrict the set of natural language expressions the user is allowed to input, so that the limits of this subset are obvious to the user (e.g., the early PRE [21], SODA [4] and ExpressQ [83]). Clearly, there is a trade-off for an NLIDB between: (i) the usability and expressivity of natural language, and (ii) the reduction of ambiguity by imposing a more structured input syntax, which may lead to higher effectiveness.

In order to understand and expand the query coverage of NLIDBs, benchmarks play a critical role. Existing ones fail to address the question of what type of NL and SQL queries a system can understand and build, respectively. One important reason for that is the lack of a query categorization. Spider [81] has four very coarse-grained classes of queries. The first organized effort to understanding the query coverage of a NLIDB is a query benchmark [25] consisting of keyword and natural language queries over three datasets, divided into 17 categories that aim to test the systems on specific linguistic and SQL aspects of the problem of translating free-form queries to SQL. The authors also provide a full experimental methodology that studies both the effectiveness (correctness of answers), and the efficiency (execution time) of the systems.

Other benchmarks focus on testing specific query capabilities. For instance, Spider-DK [24] extends Spider to explore system capabilities at cross-domain generalization (i.e., robustness to domain-specific vocabulary across different domains), while Spider-Syn [23] focuses on robustness to synonyms and different vocabulary. TRIAGESQL [85] focuses on the answerability problem and defines four types of unanswerable questions along with answerable questions. Clearly, more effort is needed in coming up with benchmarks that can provide clear insights into query coverage.

There are several types of queries that current benchmarks do not cover and neural systems are not trained to answer. For instance, *meta-knowledge questions* are questions referring to knowledge about knowledge (e.g., ASK [65]), such as "*What information is in the database*?", "*What is known about ships*?" [62] or "*What are the possible employee job titles*?" In *modal questions*, the user asks whether something can or must be the case. For example: "*Can a female employee work in sales*?" Furthermore, NLIDBs cannot answer temporal questions [3] because they cannot cope with the semantics of natural language temporal expressions (e.g. tenses/aspects, temporal subordinators), and they were designed to interface to "snapshot" databases, that do not facilitate the manipulation of time-dependent information.

Domain portability: How to cope with new domain knowledge. Early NLIDBs were each designed for a particular database application. Lunar [74], for example, was designed to support English questions, referring to a database of a particular structure, holding data about moon rocks. Such application-tailored NLIDBs were very difficult to port to different applications or databases. Different to the built-in formal query language interpreters that commercial database systems come with, more recent NLIDBs usually require tedious configuration phases before they can be used for a different database or domain. A knowledge engineer needs to capture the domain knowledge, such as rules, knowledge bases or ontologies (e.g., CLE [2], SODA [4], ATHENA [57]). The db administrator needs to create inverted indexes (e.g., TEAM [26], DISCOVER [33]). Neural systems cannot make good predictions for unseen NL questions and domains unless trained.

The top neural NLIDBs achieve high accuracy numbers on Spider. However, the majority of databases in the Spider benchmark are of low complexity, and contain general knowledge. Furthermore, most queries are simple without complex structures or operations. Neural machine translation systems pre-trained on common knowledge datasets, like Spider, typically fail in complex domains due to the large disparity in subject matter [84]. Real-world databases often store large amounts of data with hundreds of attributes. These attributes may have non-descriptive names or store numerical measurements (an example is the astrophysics database called Sloan Digital Sky Survey (SDSS)[2]). Learning the mapping of a token from a natural language question to the relevant database attribute may necessitate additional training as well as ontologies that describe the meaning of attributes and tables. On top of that, domain-specific queries may be more elaborate containing functions and mathematical operators between attributes.

Given the difficulties that arise when creating an NL interface for a real-world database, one approach is to build specialized, domain-specific benchmarks for training and evaluating such systems. For instance, Spider-DK [24] extends Spider to explore system capabilities at cross-domain generalization (i.e., robustness to domain-specific vocabulary across different domains). BIRD [49] is a dataset with questions over large-scale databases that can better represent real use-case scenarios. ScienceBenchmark [84] focuses on three real-world, highly domain-specific databases. EHRSQL [45] contains questions over two databases related to health records. Manually crafting a benchmark for a new domain requires domain-specific expertise and is challenging due to the volume of data needed. Hence, an alternative approach is *data augmentation*, i.e. automatic benchmark generation [84].

Verification: How to tell if the system response matches the NL question. A NLIDB may generate queries in its output that contain errors and do not match the NL question's intent. For example, the output SQL may contain wrong columns or tables, values that are not found in the data or they are found in a different form, unnecessary or wrong joins, and so forth. Even when an NLIDB cannot understand a question or the query is not answerable, the system may still try to generate a SQL query (as most neural systems do). Another problem is that NL questions often have several readings, and the system may select a reading of a question that is different from the reading the user had in mind. In these cases, it may be hard for the user to understand that the system has actually answered a different question. To avoid such misunderstandings, TQA [13] was a very early system that contained a module that converted the SQL query back to natural language. Query explanations in natural language provide a means for users to cross-check their question to the explanations of the predicted SQL queries and validate the results [20, 40, 71].

[2] `https://www.sdss.org/`

Efficiency. For a NLIDB, focusing on improving its query answering capabilities is only one side of the coin. Its response time also matters for their adoption. It consists of two parts: the *NL translation time* and the *SQL execution time*.

The NL translation time is an overhead to the overall query execution time that the user will experience, and hence needs to be optimized. Unfortunately, current systems overlook the significance of optimized execution, and employ methods that are time-consuming and do not scale well to large databases. Neural NLIDBs typically rely on very complex models. While the use of large PLMs usually translates to higher accuracy, it also translates to higher inference times. Output refinement techniques are also adding extra overhead. For example, one of the best-performing models on the Spider dataset, T5-3B+PICARD, uses a large PLM along with a computationally-intensive output refinement technique [58]. Schema linking also contributes to the overall translation time. These techniques have been shown to be beneficial for systems working on the Spider dataset, but they have yet to be tested on large-scale databases, where their overhead may be significant. Even though using indices and other DB lookup techniques might speed up schema linking, it is still questionable if looking up multiple words or n-grams for every NLQ, is efficient in a real application.

The SQL execution time is also important. A NL question may be written in SQL in many equivalent but not equally efficient ways. It is the NLIDB's responsibility to choose the most efficient one. Current neural systems only focus on the translation. Early text-to-SQL systems originating from the database community [32, 33, 47, 51, 83] not only tried to generate correct SQL queries but also optimal in terms of execution speed. Hence, many of them contained logic for generating code that would return the desired results fast.

A NLIDB should have a fast response time, even when the question cannot be interpreted. Ultimately, allowing the user to express questions in natural language should free them from the technical details of how this query should be expressed in the underlying system language and how it should be executed efficiently.

Reasoning. Most NLIDBs are direct interfaces to the underlying database, in the sense that they simply translate user questions to suitable database queries. In some cases, it may not be possible to answer a natural language question, although all the necessary raw data are present in the database. Questions involving common sense or domain expertise are typical examples. In these cases, to produce the answer, the NLIDB must be able to carry out reasoning based on the data stored in the database [3].

9 Looking Forward: From SQL to NQL

In practice, no modern DBMS comes with an integrated NL query language, nor does exist a NL client that connects to a database seamlessly like an SQL client and allows a user to pose queries in NL. Nevertheless, with the galloping progress of deep learning methods, the emergence of LLMs and vector databases, and several other developments, many researchers go as far as to envision that NQL (Natural Query Language) will replace SQL. In any scenario, below, we discuss some requirements for a NQL (Natural Query Language) that open up several fascinating research directions. Interestingly, such requirements have been discussed in early systems before the advent of neural NLIDBs (e.g., [63]).

R1. Query expressivity: Using a query language such as SQL, the user knows exactly what queries are possible. In a similar vein, the set of NL queries that a NLIDB supports should be clearly defined so that a user is aware of the available query capabilities.

R2. Data independence: A NLIDB should support the same query expressivity for different databases. In other words, the same type of NL query should be possible over any database. For example, if the user could ask "what is the average X of Y" in one database, then this type of query should be possible in any other database.

R3. Performance: The system should transparently find the most efficient way to answer a NL query, minimizing both the translation overhead and the query execution cost.

R4. Scalability: A NLIDB should be feasible and scalable over any database.

Requirement R1 is important because up to now almost none of the known text-to-SQL systems provides a clearly defined query language or specification of its query capabilities. For the user, it is a trial-and-error process to see what queries can be understood and answered by the system. Is it possible to come up with a query language specification that systems can refer to in order to describe their query expressivity?

Towards R1, a query categorization in the spirit of [25] may be a good starting point. This could enable the creation of appropriate benchmarks for the comparison of the query capabilities of different systems. Even devising an appropriate query categorization and an appropriate benchmark raises several challenges: what categories to choose, what queries should be in each category, which datasets to use. Furthermore, one should take into account SQL equivalence (different SQL queries that return the same results), and NL ambiguity (a NL query may have more than one correct translation over the data).

Requirement R2 complements R1 in saying that the same query expressivity should be supported over any database. This comes naturally with query languages such as SQL. For instance, SPJ queries can be supported over any data. For a NLIDB, that does not hold. Going from one database to another, the same type of queries may not be supported. As we have already pointed out, this is a major concern for neural systems.

One could build specialized, domain-specific benchmarks for training and evaluating text-to-SQL systems for a new database. Manually crafting such benchmarks is time-consuming. Data augmentation, i.e., automatic benchmark generation, is an open research direction [73]. However, benchmarks provide a means to demonstrate query expressivity. How does one ensure data independence is a different beast and finding better training datasets is not the solution to the problem. Rethinking the system design is needed instead.

Towards this direction, approaches that have been proposed by the database community have been shown to be more effective from the data independence perspective, since they rely on the information that the database provides. This potentially points to the need of re-thinking our approach to the text-to-SQL problem. Some parts of the solution may require DB methods to ensure data independence and some other parts may use neural models to generalize system knowledge, for example on the diversity and complexity of NL queries. How would a system that combines such capabilities look like?

Requirement R3 is about making NQL queries efficient. While the state-of-the-art systems are still dealing with "getting the answer right", they are mostly overlooking the "getting the answer fast". Improving translation speed by building efficient methods is necessary. But this may not be enough. Text-to-SQL systems originating from the DB community not only tried to generate correct SQL queries but also optimal in terms of execution speed. Improving the overall NQL answering time, i.e., both the translation and the execution, opens up several research opportunities, from building more efficient models to mapping translation and execution to operators and building NQL query plans that can be optimized. In fact, implementing natural language query capabilities closer to the DBMS would open up several opportunities to leverage both worlds, the database and NLP, from NLQ otpimization to learning and improving the system's query capabilities.

R4 highlights the need for realistic solutions. Deep learning text-to-SQL systems typically rely on very complex models, which have been trained and evaluated on toy databases (contained in existing benchmarks). In several cases, it may not be possible to have the required resources to train such enormous models. Furthermore, since these models require that the database schema is given as input, they do not scale well to very large databases, with hundreds of attributes and tables (such as astrophysics and biological data). Instead of focusing on increasing the model complexity aiming at translation accuracy, we need to design solutions that also take into account system efficiency, complexity, and scale.

10 Conclusion

Querying data in natural language has been the holy grail of the database community for several decades. Several efforts ended up in frustrated users with unmet expectations and disappointed researchers and developers. Commercial products were given up. However, the landscape has changed. On the one hand, technologies evolve and become increasingly more powerful. On the other hand, people are becoming accustomed to interacting with devices and software using natural language. In a few years, the way we interact with data will probably be very different from what we know nowadays.

References

1 Katrin Affolter, Kurt Stockinger, and Abraham Bernstein. A comparative survey of recent natural language interfaces for databases. *CoRR*, abs/1906.08990, 2019. doi:10.48550/arXiv.1906.08990.

2 Hiyan Alshawi, David M. Carter, Jan van Eijck, Robert C. Moore, Douglas B. Moran, and Stephen G. Pulman. Overview of the core language engine. In *Proceedings of the International Conference on Fifth Generation Computer Systems, FGCS 1988, Tokyo, Japan, November 28-December 2, 1988*, pages 1108–1115. OHMSHA Ltd. Tokyo and Springer-Verlag, 1988.

3 Ion Androutsopoulos, Graeme D. Ritchie, and Peter Thanisch. Natural language interfaces to databases – An introduction. *Natural Language Engineering*, 1(1):29–81, 1995. doi:10.1017/S135132490000005X.

4 Lukas Blunschi, Claudio Jossen, Donald Kossmann, Magdalini Mori, and Kurt Stockinger. SODA: Generating sql for business users. *PVLDB*, 5(10):932–943, 2012. doi:10.14778/2336664.2336667.

5 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, jul 2019. Association for Computational Linguistics. doi:10.18653/v1/P19-1448.

6 Ben Bogin, Matt Gardner, and Jonathan Berant. Global reasoning over database structures for text-to-SQL parsing. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3657–3662, Hong Kong, China, nov 2019. Association for Computational Linguistics. doi:10.18653/v1/D19-1378.

7 Ursin Brunner and Kurt Stockinger. Valuenet: A neural text-to-sql architecture incorporating values. *arXiv*, abs/2006.00888, 2020. doi:10.48550/arXiv.2006.00888.

8 Ruichu Cai, Boyan Xu, Zhenjie Zhang, Xiaoyan Yang, Zijian Li, and Zhihao Liang. An encoder-decoder framework translating natural language to database queries. In Jérôme Lang, editor, *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI*, pages 3977–3983. ijcai.org, 2018. doi:10.24963/ijcai.2018/553.

9 Shuaichen Chang, Jun Wang, Mingwen Dong, Lin Pan, Henghui Zhu, Alexander Hanbo Li, Wuwei Lan, Sheng Zhang, Jiarong Jiang, Joseph Lilien, et al. Dr. spider: A diagnostic evaluation benchmark towards text-to-sql robustness. *arXiv preprint*, 2023. doi:10.48550/arXiv.2301.08881.

10 DongHyun Choi, Myeong Cheol Shin, EungGyun Kim, and Dong Ryeol Shin. Ryansql: Recursively applying sketch-based slot fillings for complex text-to-sql in cross-domain databases, 2020. doi:10.48550/arXiv.2004.03125.

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

12 Ann A. Copestake and Karen Sparck Jones. Natural language interfaces to databases. *Knowl. Eng. Rev.*, 5(4):225–249, 1990. doi:10.1017/S0269888900005476.

13 Fred Damerau. Problems and some solutions in customization of natural language database front ends. *ACM Trans. Inf. Syst.*, 3(2):165–184, 1985. doi:10.1145/3914.3915.

14 Marie-Catherine de Marneffe, Bill MacCartney, and Christopher D. Manning. Generating typed dependency parses from phrase structure parses. *LREC*, pages 449–454, 2006. URL: http://www.lrec-conf.org/proceedings/lrec2006/summaries/440.html.

15 A.N. De Roeck. *A Natural Language System Based on Formal Semantics*. Universiti Sains Malaysia, 1991. URL: https://books.google.gr/books?id=hrtdnQAACAAJ.

16 Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019. doi:10.48550/arXiv.1810.04805.

17 Li Dong and Mirella Lapata. Language to logical form with neural attention, 2016. doi:10.48550/arXiv.1601.01280.

18 Timothy Dozat and Christopher D. Manning. Deep biaffine attention for neural dependency parsing, 2017. doi:10.48550/arXiv.1611.01734.

19 Timothy Dozat and Christopher D. Manning. Simpler but more accurate semantic dependency parsing, 2018. doi:10.48550/arXiv.1807.01396.

20 Stavroula Eleftherakis, Orest Gkini, and Georgia Koutrika. Let the database talk back: Natural language explanations for SQL. In *Proceedings of the 2nd Workshop on Search, Exploration, and Analysis in Heterogeneous Datastores (SEA-Data 2021) co-located with 47th International Conference on Very Large Data Bases (VLDB 2021), Copenhagen, Denmark, August 20, 2021*, volume 2929 of *CEUR Workshop Proceedings*, pages 14–19. CEUR-WS.org, 2021. URL: https://ceur-ws.org/Vol-2929/paper3.pdf.

21 Samuel S. Epstein. Transportable natural language processing through simplicity - the PRE system. *ACM Trans. Inf. Syst.*, 3(2):107–120, 1985. doi:10.1145/3914.3985.

22 Han Fu, Chang Liu, Bin Wu, Feifei Li, Jian Tan, and Jianling Sun. Catsql: Towards real world natural language to SQL applications. *Proc. VLDB Endow.*, 16(6):1534–1547, 2023. doi:10.14778/3583140.3583165.

23 Yujian Gan, Xinyun Chen, Qiuping Huang, Matthew Purver, John R. Woodward, Jinxia Xie, and Pengsheng Huang. Towards robustness of text-to-SQL models against synonym substitution. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 2505–2515, Online, aug 2021. Association for Computational Linguistics. doi:10.18653/v1/2021.acl-long.195.

24 Yujian Gan, Xinyun Chen, and Matthew Purver. Exploring underexplored limitations of cross-domain text-to-sql generalization. *arXiv preprint*, 2021. doi:10.48550/arXiv.2109.05157.

25 Orest Gkini, Theofilos Belmpas, Georgia Koutrika, and Yannis E. Ioannidis. An in-depth benchmarking of text-to-sql systems. In Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 632–644. ACM, 2021. doi:10.1145/3448016.3452836.

26 Barbara J. Grosz. TEAM: A transportable natural-language interface system. In *1st Applied Natural Language Processing Conference, ANLP 1983, Miramar-Sheraton Hotel, Santa Monica, California, USA, February 1-3, 1983*, pages 39–45. ACL, 1983. doi:10.3115/974194.974201.

27 Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. Towards complex text-to-sql in cross-domain database with intermediate representation, 2019. doi:10.48550/arXiv.1905.08205.

28 Larry R. Harris. Experience with INTELLECT: artificial intelligence technology transfer. *AI Mag.*, 5(2):43–50, 1984. URL: https://ojs.aaai.org/index.php/aimagazine/article/view/437.

29 Moshe Hazoom, Vibhor Malik, and Ben Bogin. Text-to-SQL in the wild: A naturally-occurring dataset based on stack exchange data. In *1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*, pages 77–87, aug 2021.

30 Pengcheng He, Yi Mao, Kaushik Chakrabarti, and Weizhu Chen. X-sql: reinforce schema representation with context, 2019. doi:10.48550/arXiv.1908.08113.

31 Gary G. Hendrix, Earl D. Sacerdoti, Daniel Sagalowicz, and Jonathan Slocum. Developing a natural language interface to complex data. *ACM Trans. Database Syst.*, 3(2):105–147, jun 1978. doi:10.1145/320251.320253.

32 Vagelis Hristidis, Luis Gravano, and Yannis Papakonstantinou. Efficient IR-style keyword search over relational databases. In *VLDB*, pages 850–861, 2003. doi:10.1016/B978-012722442-8/50080-X.

33 Vagelis Hristidis and Yannis Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, pages 670–681, 2002. doi:10.1016/B978-155860869-6/50065-2.

34 Binyuan Hui, Xiang Shi, Ruiying Geng, Binhua Li, Yongbin Li, Jian Sun, and Xiaodan Zhu. Improving text-to-sql with schema dependency learning, 2021. doi:10.48550/arXiv.2103.04399.

35 Wonseok Hwang, Jinyeong Yim, Seunghyun Park, and Minjoon Seo. A comprehensive exploration on wikisql with table-aware word contextualization, 2019. doi:10.48550/arXiv.1902.01069.

36 Tim Johnson. Natural language computing: The commercial applications. *Knowl. Eng. Rev.*, 1(3):11–23, 1984. doi:10.1017/S0269888900000588.

37 George Katsogiannis-Meimarakis and Georgia Koutrika. A deep dive into deep learning approaches for text-to-sql systems. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD/PODS '21, pages 2846–2851, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3448016.3457543.

38 George Katsogiannis-Meimarakis and Georgia Koutrika. A survey on deep learning approaches for text-to-sql. *The VLDB Journal*, 2023. doi:10.1007/s00778-022-00776-8.

39 Hyeonji Kim, Byeong-Hoon So, Wook-Shin Han, and Hongrae Lee. Natural language to sql: Where are we today? *Proc. VLDB Endow.*, 13(10):1737–1750, 2020. doi:10.14778/3401960.3401970.

40 Andreas Kokkalis, Panagiotis Vagenas, Alexandros Zervakis, Alkis Simitsis, Georgia Koutrika, and Yannis E. Ioannidis. Logos: a system for translating queries into narratives. In K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, and Ariel Fuxman, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 673–676. ACM, 2012. doi:10.1145/2213836.2213929.

41 Georgia Koutrika, Alkis Simitsis, and Yannis E. Ioannidis. Précis: The essence of a query answer. In Ling Liu, Andreas Reuter, Kyu-Young Whang, and Jianjun Zhang, editors, *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, pages 69–78. IEEE Computer Society, 2006. doi:10.1109/ICDE.2006.114.

42 Jayant Krishnamurthy, Pradeep Dasigi, and Matt Gardner. Neural semantic parsing with type constraints for semi-structured tables. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 1516–1526, Copenhagen, Denmark, sep 2017. Association for Computational Linguistics. doi:10.18653/v1/D17-1160.

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

44 Chia-Hsuan Lee, Oleksandr Polozov, and Matthew Richardson. Kaggledbqa: Realistic evaluation of text-to-sql parsers. *arXiv preprint*, 2021. doi:10.48550/arXiv.2106.11455.

45 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. URL: http://papers.nips.cc/paper_files/paper/2022/hash/643e347250cf9289e5a2a6c1ed5ee42e-Abstract-Datasets_and_Benchmarks.html.

46 Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension, 2019. doi:10.48550/arXiv.1910.13461.

47 Fei Li and H. V. Jagadish. Constructing an interactive natural language interface for relational databases. *PVLDB*, 8(1):73–84, sep 2014. doi:10.14778/2735461.2735468.

48 Haoyang Li, Jing Zhang, Cuiping Li, and Hong Chen. RESDSQL: decoupling schema linking and skeleton parsing for text-to-sql. In Brian Williams, Yiling Chen, and Jennifer Neville, editors, *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023, Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence, IAAI 2023, Thirteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2023, Washington, DC, USA, February 7-14, 2023*, pages 13067–13075. AAAI Press, 2023. doi:10.1609/aaai.v37i11.26535.

49 Jinyang Li, Binyuan Hui, Ge Qu, Binhua Li, Jiaxi Yang, Bowen Li, Bailin Wang, Bowen Qin, Rongyu Cao, Ruiying Geng, et al. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *arXiv preprint*, 2023. doi:10.48550/arXiv.2305.03111.

50 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, nov 2020. Association for Computational Linguistics. doi:10.18653/v1/2020.findings-emnlp.438.

51 Yi Luo, Xuemin Lin, Wei Wang, and Xiaofang Zhou. Spark: Top-k keyword query in relational databases. In *ACM SIGMOD*, pages 115–126, 2007. doi:10.1145/1247480.1247495.

52 Qin Lyu, Kaushik Chakrabarti, Shobhit Hathi, Souvik Kundu, Jianwen Zhang, and Zheng Chen. Hybrid ranking network for text-to-sql. Technical Report MSR-TR-2020-7, Microsoft Dynamics 365 AI, mar 2020. URL: https://www.microsoft.com/en-us/research/publication/hybrid-ranking-network-for-text-to-sql/.

53 Nikolaus Ott. Aspects of the automatic generation of SQL statements in the natural language query interface. *Inf. Syst.*, 17(2):147–159, 1992. doi:10.1016/0306-4379(92)90009-C.

54 Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014. URL: http://www.aclweb.org/anthology/D14-1162, doi:10.3115/v1/d14-1162.

55 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, 2020. doi:10.48550/arXiv.1910.10683.

56 Ohad Rubin and Jonathan Berant. SmBoP: Semi-autoregressive bottom-up semantic parsing. In *Proceedings of the 5th Workshop on Structured Prediction for NLP (SPNLP 2021)*, pages 12–21, Online, aug 2021. Association for Computational Linguistics. doi:10.18653/v1/2021.spnlp-1.2.

57 Diptikalyan Saha, Avrilia Floratou, Karthik Sankaranarayanan, Umar Farooq Minhas, Ashish R. Mittal, Fatma Özcan, IBM Research. Bangalore, and IBM Research. Almaden. *ATHENA: An Ontology-Driven System for Natural Language Querying over Relational Data Stores.* VLDB, 2016. URL: `http://www.vldb.org/pvldb/vol9/p1209-saha.pdf`, `doi:10.14778/2994509.2994536`.

58 Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. Picard: Parsing incrementally for constrained auto-regressive decoding from language models, 2021. `doi:10.48550/arXiv.2109.05093`.

59 Jaydeep Sen, Chuan Lei, Abdul Quamar, Fatma Özcan, Vasilis Efthymiou, Ayushi Dalmia, Greg Stager, Ashish R. Mittal, Diptikalyan Saha, and Karthik Sankaranarayanan. ATHENA++: natural language querying for complex nested SQL queries. *Proc. VLDB Endow.*, 13(11):2747–2759, 2020. URL: `http://www.vldb.org/pvldb/vol13/p2747-sen.pdf`.

60 Or Sharir, Barak Peleg, and Yoav Shoham. The cost of training NLP models: A concise overview. *CoRR*, abs/2004.08900, 2020. `doi:10.48550/arXiv.2004.08900`.

61 Tianze Shi, Kedar Tatwawadi, Kaushik Chakrabarti, Yi Mao, Oleksandr Polozov, and Weizhu Chen. Incsql: Training incremental text-to-sql parsers with non-deterministic oracles, 2018. `doi:10.48550/arXiv.1809.05054`.

62 Alkis Simitsis, Georgia Koutrika, and Yannis Ioannidis. Précis: from unstructured keywords as queries to structured databases as answers. *The VLDB Journal*, 17(1):117–149, 2008. `doi:10.1007/s00778-007-0075-9`.

63 Marjorie Templeton and John Burger. Problems in natural-language interface to DBMS with examples from EUFID. In *First Conference on Applied Natural Language Processing*, pages 3–16, Santa Monica, California, USA, feb 1983. Association for Computational Linguistics. `doi:10.3115/974194.974197`.

64 Harry R. Tennant, Kenneth M. Ross, Richard M. Saenz, Craig W. Thompson, and James R. Miller. Menu-based natural language understanding. In Mitchell P. Marcus, editor, *21st Annual Meeting of the Association for Computational Linguistics, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, June 15-17, 1983*, pages 151–158. ACL, 1983. `doi:10.3115/981311.981341`.

65 Bozena Henisz Thompson and Frederick B. Thompson. Introducing ask, A simple knowledgeable system. In *1st Applied Natural Language Processing Conference, ANLP 1983, Miramar-Sheraton Hotel, Santa Monica, California, USA, February 1-3, 1983*, pages 17–24. ACL, 1983. `doi:10.3115/974194.974198`.

66 Arif Usta, Akifhan Karakayali, and Özgür Ulusoy. Dbtagger: Multi-task learning for keyword mapping in NLIDBs using bi-directional recurrent neural networks. *Proc. VLDB Endow.*, 14(5):813–821, jan 2021. `doi:10.14778/3446095.3446103`.

67 Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017. `doi:10.48550/arXiv.1706.03762`.

68 Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. Rat-sql: Relation-aware schema encoding and linking for text-to-sql parsers, 2020. `doi:10.48550/arXiv.1911.04942`.

69 Bing Wang, Yan Gao, Zhoujun Li, and Jian-Guang Lou. Know what I don't know: Handling ambiguous and unknown questions for text-to-sql. In Anna Rogers, Jordan L. Boyd-Graber, and Naoaki Okazaki, editors, *Findings of the Association for Computational Linguistics: ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 5701–5714. Association for Computational Linguistics, 2023. `doi:10.18653/V1/2023.FINDINGS-ACL.352`.

70 Chenglong Wang, Kedar Tatwawadi, Marc Brockschmidt, Po-Sen Huang, Yi Mao, Oleksandr Polozov, and Rishabh Singh. Robust text-to-sql generation with execution-guided decoding, 2018. `arXiv:1807.03100`.

71 Weiguo Wang, Sourav S. Bhowmick, Hui Li, Shafiq R. Joty, Siyuan Liu, and Peng Chen. Towards enhancing database education: Natural language generation meets query execution plans. In Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava, editors, *SIGMOD '21: International Conference on Management of Data*, pages 1933–1945. ACM, 2021. doi:10.1145/3448016.3452822.

72 David H. D. Warren and Fernando C. N. Pereira. An efficient easily adaptable system for interpreting natural language queries. *Am. J. Comput. Linguistics*, 8(3-4):110–122, 1982.

73 Nathaniel Weir, Prasetya Utama, Alex Galakatos, Andrew Crotty, Amir Ilkhechi, Shekar Ramaswamy, Rohin Bhushan, Nadja Geisler, Benjamin Hättasch, Steffen Eger, Ugur Çetintemel, and Carsten Binnig. Dbpal: A fully pluggable NL2SQL training pipeline. In David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 2347–2361. ACM, 2020. doi:10.1145/3318464.3380589.

74 William A. Woods. Procedural semantics for a question-answering machine. In *Proceedings of the AFIPS '68 Fall Joint Computer Conference, December 9-11, 1968, San Francisco, California, USA - Part I*, volume 33, pages 457–471, 1968. doi:10.1145/1476589.1476653.

75 Xiaojun Xu, Chang Liu, and Dawn Song. Sqlnet: Generating structured queries from natural language without reinforcement learning, 2017. doi:10.48550/arXiv.1711.04436.

76 Kuan Xuan, Yongbo Wang, Yongliang Wang, Zujie Wen, and Yang Dong. Sead: End-to-end text-to-sql generation with schema-aware denoising, 2021. doi:10.48550/arXiv.2105.07911.

77 Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. Sqlizer: Query synthesis from natural language. *PACMPL*, pages 63:1–63:26, 2017. doi:10.1145/3133887.

78 Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation, 2017. doi:10.48550/arXiv.1704.01696.

79 Pengcheng Yin and Graham Neubig. TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 7–12, Brussels, Belgium, nov 2018. Association for Computational Linguistics. doi:10.18653/v1/D18-2002.

80 Tao Yu, Zifan Li, Zilin Zhang, Rui Zhang, and Dragomir Radev. Typesql: Knowledge-based type-aware neural text-to-sql generation, 2018. doi:10.48550/arXiv.1804.09769.

81 Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task, 2019. doi:10.48550/arXiv.1809.08887.

82 Wei Yu, Haiyan Yang, Mengzhu Wang, and Xiaodong Wang. Bravely say I don't know: Relational question-schema graph for text-to-sql answerability classification. *ACM Trans. Asian Low Resour. Lang. Inf. Process.*, 22(4):111:1–111:18, 2023. doi:10.1145/3579030.

83 Zhong Zeng, Mong Li Lee, and Tok Wang Ling. Answering keyword queries involving aggregates and groupby on relational databases. *EDBT*, pages 161–172, 2016. doi:10.5441/002/edbt.2016.17.

84 Yi Zhang, Jan Deriu, George Katsogiannis-Meimarakis, Catherine Kosten, Georgia Koutrika, and Kurt Stockinger. Sciencebenchmark: A complex real-world benchmark for evaluating natural language to sql systems, 2023. doi:10.48550/arXiv.2306.04743.

85 Yusen Zhang, Xiangyu Dong, Shuaichen Chang, Tao Yu, Peng Shi, and Rui Zhang. Did you ask a good question? A cross-domain question intention classification benchmark for text-to-sql. *CoRR*, abs/2010.12634, 2020. doi:10.48550/arXiv.2010.12634.

86 Victor Zhong, Caiming Xiong, and Richard Socher. Seq2sql: Generating structured queries from natural language using reinforcement learning, 2017. doi:10.48550/arXiv.1709.00103.