Natural language to SQL: Where are we today?

Hyeonji Kim[†] Byeong-Hoon So[†] Wook-Shin Han[†*] Hongrae Lee[‡]
POSTECH, Korea[†], Google, USA[‡]
{hjkim, bhso, wshan}@dblab.postech.ac.kr[†], hrlee@google.com[‡]

ABSTRACT

Translating natural language to SQL (NL2SQL) has received extensive attention lately, especially with the recent success of deep learning technologies. However, despite the large number of studies, we do not have a thorough understanding of how good existing techniques really are and how much is applicable to real-world situations. A key difficulty is that different studies are based on different datasets, which often have their own limitations and assumptions that are implicitly hidden in the context or datasets. Moreover, a couple of evaluation metrics are commonly employed but they are rather simplistic and do not properly depict the accuracy of results, as will be shown in our experiments. To provide a holistic view of NL2SQL technologies and access current advancements, we perform extensive experiments under our unified framework using *eleven* of recent techniques over 10+ benchmarks including a new benchmark (WTQ) and TPC-H. We provide a comprehensive survey of recent NL2SQL methods, introducing a taxonomy of them. We reveal major assumptions of the methods and classify translation errors through extensive experiments. We also provide a practical tool for validation by using existing, mature database technologies such as query rewrite and database testing. We then suggest future research directions so that the translation can be used in practice.

PVLDB Reference Format:
Hyeonji Kim, Byeong-Hoon So, Wook-Shin Han, Hongrae Lee. Natural language to SQL: Where are we today?. *PVLDB*, 13(10): 1737-1750, 2020.
DOI: https://doi.org/10.14778/3401960.3401970

1. INTRODUCTION

Given a relational database D and a natural language question q_{nl} to D, translating natural language to SQL (NL2SQL) is to find an SQL statement q_{sql} to answer q_{nl}. This problem is important for enhancing the accessibility of relational database management systems (RDBMSs) for end users. NL2SQL techniques can be applied to commercial RDBMSs, allowing us to build natural language interfaces to relational databases.

In recent years, NL2SQL has been actively studied in both the database community and the natural language community. [25, 35] from the database community have proposed rule-based techniques that use mappings between natural language words and SQL keywords/data elements. [3] have improved the quality of the mapping and inferring join condition by leveraging (co-)occurrence information of SQL fragments from the query log. In the natural language community, many studies [22, 44, 53, 46, 21, 24, 47, 6, 20] have been conducted in recent years for the NL2SQL problem by using the state-of-the-art deep learning models and algorithms.

Although the aforementioned studies evaluate the performance of their methods, comparison with the existing methods has not been performed properly. All of them compare to a limited set of previous methods only. For example, the studies in the natural language community [22, 44, 53, 46, 21, 24] exclude comparisons with [25, 35] from the database community. In addition, previous studies evaluate their performance using only a subset of existing benchmarks. For example, [44, 53, 46, 21, 24] use only the WikiSQL benchmark [52], while [22] uses only the GeoQuery benchmark [49, 50, 32, 17, 22], the ATIS benchmark [33, 12, 22, 51], and the Scholar benchmark [22]. [47, 6, 20] use only the Spider benchmark [48].

In this paper, we present comprehensive experimental results using all representative methods and various benchmarks. We provide a survey of existing methods and perform comparative experiments of various methods. Note that every NL2SQL benchmark has a limitation in that it covers a limited scope of the NL2SQL problem.

A more serious problem in the previous studies is that all accuracy measures used are misleading. There are four measures: 1) string matching, 2) parse tree matching, 3) result matching, and 4) manual matching. In string matching, we compare a generated SQL query to the corresponding ground truth (called the gold SQL query). This can lead to inaccurate decisions for various reasons, such as the order of conditions in the where clause, the order of the projected columns, and aliases. For example, if one query contains "P_1 AND P_2" in its WHERE clause, and another contains "P_2 AND P_1", where P_1 is "city.state_name = 'california'" and P_2 is "city.population $>$ 1M", then the string match judges that both queries are different. Parse tree matching compares parse trees from two SQL queries. This approach has less error than the first one, but it is still misleading. For example, a nested query and its flattened query are equivalent, but their parse trees would be different from each other. Result matching compares the execution results of two SQL queries in a given database. This is based on the idea that the same SQL queries would produce the same results. However, this would overestimate if two different SQL queries produce the same results by chance. In manual matching, users validate the translation results by checking the execution results or the SQL queries. This requires considerable manual effort and cannot guarantee reliability.

[*] Corresponding author

Proceedings of the VLDB Endowment, Vol. 13, No. 10
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3401960.3401970

In this paper, we perform extensive experiments using 14 benchmarks including a new benchmark (WTQ) and TPC-H for measuring the accuracy of NL2SQL methods. We conduct an experiment to identify the problem of the measures introduced above. Specifically, we measure the performance of the existing methods using three existing measures, omitting the fourth one which involves the manual inspection. We use a precise metric considering semantic equivalence and propose a practical tool to measure it automatically. In order to accurately measure the translation accuracy of NL2SQL, we need to judge the semantic equivalence between two SQL queries. However, the existing technique for determining semantic equivalence [11] is not comprehensive to use for our experiments, since it only supports restrictive forms of SQL queries. We propose a practical tool to judge semantic equivalence by using database technologies such as query rewrite and database testing and measure accuracy based on it. Note that this has been an important research topic called semantic equivalence of SQL queries in our field. Nevertheless, we still have no practical tool for supporting semantic equivalence for complex SQL queries.

The main contributions of this paper are as follows: 1) We provide a survey and a taxonomy of existing NL2SQL methods, and classify the latest methods. 2) We fairly and empirically compare eleven state-of-the-art methods using 14 benchmarks. 3) We show that all the previous studies use misleading measures in their performance evaluation. To solve this problem, we propose a practical tool for validating translation results by taking into account the semantic equivalence of SQL queries. 4) We analyze the experimental results in depth to understand why one method is superior to another for a particular benchmark. 5) We report several surprising, important findings obtained.

The remainder of the paper is organized as follows. Section 2 introduces fundamental natural language processing and deep learning concepts. We introduce existing NL2SQL benchmarks used in recent studies in Section 3. In Section 4, we show a brief history and review the state-of-the-art NL2SQL methods. We explain our validation methodology and propose a practical tool for measuring accuracy in Section 5. Section 6 presents experimental results and in-depth analysis for them. We discuss insights and questions for future research in Section 7. We conclude in Section 8.

2. BACKGROUND

Dependency parsing: A syntactic dependency [15] is a binary asymmetric relation between two words in a sentence. Each dependency bears a grammatical function (e.g., subject, object, determiner, modifier) of each word wrt. another word. It can be represented as a typed and directed arrow from one word to another. All arrows in a sentence usually form a rooted tree, called a syntactic dependency tree, where the words of the given sentence are nodes and h is the parent of d for each arrow $h \rightarrow d$. Dependency parsing is a task for finding syntactic dependencies of a given sentence [23].

Recurrent neural networks (RNNs): A basic RNN takes an input sequence of vectors $[x_1 x_2 \dots x_\tau]$ of length τ as well as an initial hidden state h_0, and generates a sequence of hidden states $[h_1 h_2 \dots h_\tau]$ and a sequence of output vectors $[y_1 y_2 \dots y_\tau]$. Specifically, h_t at time step t is calculated by $h_t = f_\theta(x_t, h_{t-1})$, where f_θ is a function with a parameter θ, which is commonly referred to as an RNN cell.

If an RNN cell is implemented as just a fully connected layer with an activation function, it would not effectively accumulate information from previous time steps in its hidden state of the RNN. Such a basic RNN would not effectively handle long sequences and face the notorious *vanishing and exploding gradient* problem [5]. In order to avoid the problem, long short-term memory (LSTM), gated recurrent units (GRUs), or residual networks (ResNets) have been proposed. For example, an LSTM cell maintains an additional cell state c_t which remembers information over time and three gates to regulate the flow of information into and out of the cell. That is, h_t and c_t are computed using the gates from c_{t-1}, h_{t-1}, and x_t.

Sequence-to-sequence models: A sequence-to-sequence model (Seq2Seq) translates a source sentence into a target sentence. It consists of an encoder and a decoder, each of which is implemented by an RNN – usually an LSTM. The encoder takes a source sentence and generates a fixed-size context vector, while the decoder takes the context vector C and generates a target sentence.

Attention mechanism: One fundamental problem in the basic Seq2Seq is that the final RNN hidden state h_τ in the encoder is used as the single context vector for the decoder. Encoding a long sentence into a single context vector would lead to information loss and inadequate translation, which is called the *hidden state bottleneck* problem [2]. To avoid this problem, the attention mechanism has been actively used.

Seq2Seq with attention: At each step of the output word generation in the decoder, we need to examine all the information that the source sentence holds. That is, we want the decoder to attend to different parts of the source sentence at each word generation. The attention mechanism makes Seq2Seq *learn what to attend to*, based on the source sentence and the generated target sequence thus far.

The attention encoder passes all the hidden states to the decoder instead of only the last hidden state [1, 27]. In order to focus on the parts of the input that are relevant to this decoding time step, the attention decoder 1) considers all encoder hidden states, 2) computes a softmax score for each hidden state, and 3) computes a weighted sum of hidden states using their scores. Then, it gives more attention to hidden states with high scores.

Pointer network: Although the attention mechanism solves the hidden state problem, there is another fundamental problem in language translation, called the *unknown word* problem [19]. In the Seq2Seq model, the decoder generates a unique word in a predefined vocabulary using a softmax classifier, where each word corresponds to an output dimension. However, the softmax classifier cannot predict words out of the vocabulary.

In order to solve the unknown word problem, pointer networks have been widely used. A pointer network generates a sequence of pointers to the words of the source sentence [37]. Thus, it is considered as a variation of the Seq2Seq model with attention.

3. BENCHMARKS

Existing NL2SQL methods address limited-scope problems under several explicit or implicit assumptions about q_{nl}, q_{sql}, or D. These assumptions are directly related to the characteristics of its target benchmark since each method is optimized for a particular benchmark without considering the general problem.

WikiSQL: WikiSQL [52] is the most widely used and largest bench mark, containing 26,531 tables and 80,654 (q_{nl}, q_{sql}) pairs over a given single table. Tables are extracted from HTML tables from Wikipedia. Then, each q_{sql} is automatically generated for a given table under the constraint that the query produces a non-empty result set. Each q_{nl} is generated using a simple template and paraphrased through Amazon Mechanical Turk.

All SQL queries in WikiSQL are based on a simple syntax pattern, that is, SELECT <aggregation function> <column name> FROM T [WHERE <column name> <operator> <constant value> (and <column name> <operator> <constant value>)*], where T is a given single table. This allows only a single projected column and selection with conjunctions. Note that this grammar expresses none of grouping, ordering, join, or nested queries.

ATIS/GeoQuery: ATIS [33, 12, 22, 51] and GeoQuery [49, 50, 32, 17, 22] are widely used for *semantic parsing* which is the task of translating q_{nl} into a formal meaning representation. ATIS is about flight booking and contains a database of 25 tables and 5,410 (q_{nl}, q_{sql}) pairs. GeoQuery consists of seven tables in the US geography database and 880 (q_{nl}, q_{sql}) pairs [32]. Unlike WikiSQL, all queries in ATIS and GeoQuery are on a single domain. Thus, if we use them for training a deep-learning model, the model will work on a specific domain only. Both benchmarks have various queries including join and nested queries. ATIS does not have any grouping or ordering query, whereas GeoQuery does.

MAS: MAS [25] has been proposed to evaluate NaLIR [25]. Microsoft Academic Search provides a database of academic social networks and a set of queries. The authors of NaLIR select 196 queries of them. MAS is on a single domain like ATIS and GeoQuery. It contains a database of 17 tables and 196 (q_{nl}, q_{sql}) pairs. MAS has various SQL queries containing join, grouping, and nested queries, but not ordering queries. Each q_{nl} in MAS has the following constraints. First, q_{nl} starts with "return me". While a user may pose a query using an interrogative sentence or a list of keywords in real world situations, MAS does not include such cases. Second, each q_{nl} is grammatically correct.

Spider: Recently, [48] proposed a new NL2SQL benchmark named Spider. [48] claims that existing benchmarks have limited quality, that is, they have a few queries, simple queries only, or on a single database. Spider is a large-scale cross-domain benchmark with 200 databases of 138 different domains and 10,181 (q_{nl}, q_{sql}) pairs.

4. NL2SQL METHODS

4.1 A brief history

The construction of natural language interfaces for databases has been studied in both the database and natural language communities for decades. Figure 1 shows its brief history.

In the 1980s, methods using intermediate logical representation were proposed [39, 18]. They translate q_{nl} into logical queries independent of the underlying database schema, and then convert these logical queries to database queries. However, they still rely on hand-crafted mapping rules for translation.

From the early 2000s, more advanced rule-based methods [31, 25, 35, 43, 3] were proposed. [31] used an off-the-shelf natural language parser in order to integrate the advances in natural language processing without training a parser for a specific database. However, the coverage was limited to *semantically tractable* questions [31]. This limitation is mainly caused by the assumption that there is a one-to-one correspondence between words in the question and a subset of database elements. In order to broaden coverage, [25, 43, 35] proposed a ranking-based approach. During the mapping between words in the question and the database elements, they found multiple candidate mappings and calculated and ranked the mapping score for each candidate. NaLIR [25] further improved performance with user interaction. ATHENA [35] intermediately used a domain specific ontology to exploit its richer semantic information. SQLizer [43] parses q_{nl} into a logical form using Sempre [30], then iteratively refines the form. Templar [3] is an optimization technique for mapping and join path generation using a query log. Although these methods achieved significant performance improvement, they still rely on manually-defined rules.

Recently, deep-learning-based (DL-based) methods [22, 52, 42, 24, 4, 44, 21, 53, 46, 47, 6, 20] have been actively proposed in the NLP community by exploiting the state-of-the-art deep learning technologies. One of the main challenges in developing a DL-based NL2SQL method is the lack of training data. NSP [22] used an interactive learning algorithm and a data augmentation technique using templates and paraphrasing. DBPal [4] used a data augmentation technique similar to NSP, which uses more varied templates and more diverse paraphrasing techniques than NSP. On the other hand, a new benchmark named WikiSQL was published in [52]. Accordingly, many studies [42, 24, 44, 21, 53, 46] have been conducted to improve accuracy on WikiSQL. Seq2SQL [52], SQLNet [42], Coarse2Fine [24], and STAMP [53] proposed a new deep-learning model specific to WikiSQL. PT-MAML [21] adapted the latest learning algorithm called *meta learning* [14] for WikiSQL. TypeSQL [46] tagged each word in the question with a data type or a column name and used the tags as input to its deep learning model. More recently, as [48] proposed a new NL2SQL benchmark named Spider, SyntaxSQLNet [47], GNN [6], and IRNet [20] targeting Spider have been proposed.

4.2 Taxonomy

We provide a general taxonomy for NL2SQL, and then classify the recent methods according to the taxonomy. The taxonomy in Figure 2 considers three dimensions and four sub-dimensions in the technique dimension.

In our survey, we include existing methods published at major conferences of database and natural language processing areas from 2014 to September, 2019. We review a total of 16 methods [25, 35, 43, 3, 42, 24, 44, 21, 53, 46, 47, 6, 20].

4.2.1 Input

Regarding the input dimension, we have four sub-dimensions: pre-processing of q_{nl}; a table/database as input; schema only and schema + value; and additional inputs (Figure 3). Existing methods take two inputs, q_{nl} and a database D, consisting of a set of tables along with the database schema S_D. Here, each table has a set of records, and each record has a set of column values, V_D. Note that Templar is presented in the 'technique' dimension only, since it is an optimization technique for rule-based methods.

All existing methods take an English text as q_{nl}. They use different pre-processing techniques as follows: All DL-based methods use pre-trained embedding vectors of tokens (in q_{nl}, S_D, or V_D) as Word2Vec [28]. For rule-based methods, NaLIR parses q_{nl} into the corresponding dependency parse tree. SQLizer transforms q_{nl} into a logical form using Sempre [30]. ATHENA uses a sequence of words. Some techniques also require additional inputs: an open-domain knowledge base such as Freebase [7] for detecting named entities; domain-specific dictionary/ontology; a pre-built mapping table between phrases in q_{nl} and SQL keywords; WordNet [29]; and a word embedding matrix for calculating word similarity.

Although all methods use general pre-processing techniques, and there is no explicit constraint on q_{nl}, each method has implicit assumptions on q_{nl}. For example, NaLIR uses a manually-built mapping table between NL phrases and SQL keywords. To generate proper SQL keywords, q_{nl} must include phrases in the table.

For D, we examine their domain adaptability, constraints, and utilization. Unlike other methods taking a (q_{nl}, D) pair as an input for inference, NSP and DBPal use D only for vocabulary building during training. During inference, they assume that q_{nl} is over D seen in the training. Some assume that D consists of a single table only. ATHENA requires a specific dictionary and an ontology for each D. For database utilization, some use S_D only, while others use both S_D and V_D.

4.2.2 Technique: translation

We explain the translation dimension first for a better understanding of the existing methods (Figure 4). In the translation

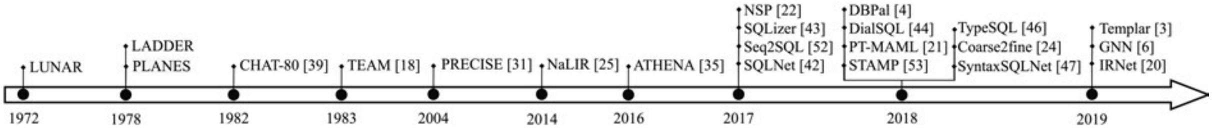

Figure 1: A brief history of natural language interface to relational databases.

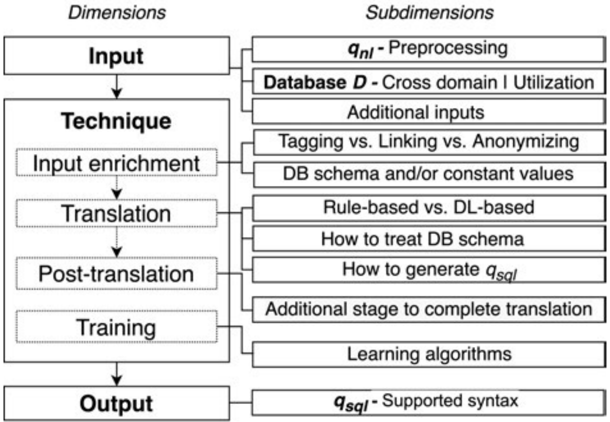
Dimensions
Subdimensions
Input
Technique
Input enrichment
Translation
Post-translation
Training
Output
q_{nl} - Preprocessing
Database D - Cross domain | Utilization
Additional inputs
Tagging vs. Linking vs. Anonymizing
DB schema and/or constant values
Rule-based vs. DL-based
How to treat DB schema
How to generate q_{sql}
Additional stage to complete translation
Learning algorithms
q_{sql} - Supported syntax

Figure 2: An overview of the classification criteria.

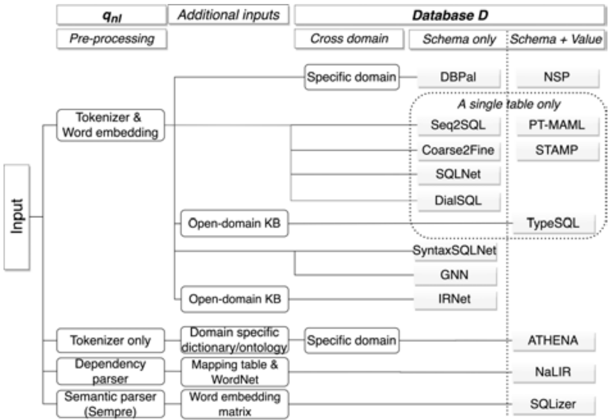

Figure 3: Taxonomy over the 'input' dimension.

phase, each method translates q_{nl} to q_{sql} directly or to a certain intermediate representation.

Rule-based methods: [25, 35, 43] generate q_{sql} by applying a fixed set of rules to q_{nl}. NaLIR and ATHENA translate q_{nl} into a tree-structured intermediate representation. NaLIR transforms a dependency parse tree into a valid parse tree. The transformation is performed by using a simple algorithm which arbitrarily moves a sub-tree of the initial tree and by applying a set of node-insertion rules. ATHENA builds an interpretation tree of q_{nl}, where each node corresponds to a concept/property, and an edge represents a relation in a given ontology. SQLizer transforms the input logical form into q_{sql} by iteratively modifying the logical form.

During the translation, each method ranks candidate representations. NaLIR uses the edit distance between the initial tree and the tree modified by the heuristic sub-tree movement; ATHENA proposes a modified Steiner Tree algorithm; and SQLizer defines a specific score function based on its own rules.

DL-based methods: [22, 52, 42, 24, 4, 44, 21, 53, 46, 47, 6, 20] generate q_{sql} by using an encoder-decoder model. All of the methods encode q_{nl} by using an RNN. We classify these methods according to how they treat S_D and how they generate q_{sql}.

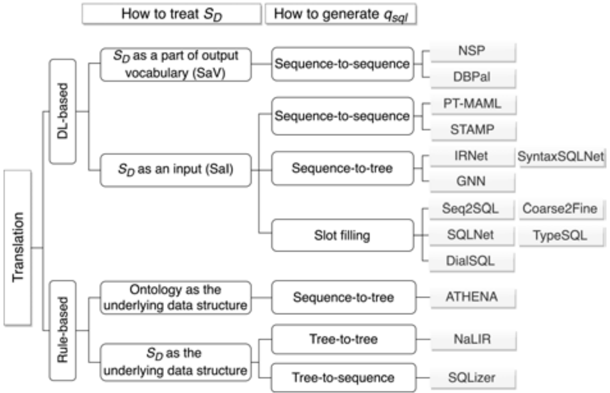

Figure 4: Taxonomy over the 'translation' dimension.

Each method treats S_D as either a part of the output vocabulary (SaV) or the input (SaI). The SaV methods put all table/column names into the output vocabulary and decode schema entities by selecting words from the vocabulary. In contrast, the SaI methods take the database schema as input and decode schema entities by copying from the input based on the pointer mechanism.

Regarding the generation of q_{sql}, there exist three types of DL-based methods: sequence-to-sequence, sequence-to-tree, and slot filling. The sequence-to-sequence methods generate an SQL query as a sequence of words. The sequence-to-tree methods generate a syntax tree of an SQL query. The slot-filling methods treat an SQL query as a set of slots, and decode the whole query by using a relevant decoder for each slot.

NSP and DBPal, which are SaV methods, focus on with a limited size of training data rather than the design of a specialized model. Both of them use a Seq2Seq with attention. Unlike traditional machine translation methods that use large vocabulary, they take advantage of small vocabulary with only SQL keywords, table/column names, and constant values in training examples. This significantly reduces the difficulty of translation problems. Before training, we need to put SQL keywords, table/column names, and constant values into the output vocabulary of the decoder. Then, the decoder can generate an SQL query by selecting a sequence of words from its output vocabulary. Since the SaV methods build vocabulary from the training examples, they can process queries well if they contain words in the vocabulary. Thus, they need to re-train their model for a new database.

The SaI methods are subdivided into three classes according to the structure of a schema encoder: a sequence, a set of tables/-columns, and a graph. PT-MAML, TypeSQL, and Coarse2Fine take S_D as a sequence of column names by using a single RNN. Seq2SQL, SQLNet, STAMP, SyntaxSQLNet, and IRNet treat S_D as a set of tables/columns by using an RNN for each table and for each column. GNN treats S_D as a graph by using a graph neural network [26] to utilize the structural information of S_D.

The SaI methods can also be classified into three categories according to how they generate q_{sql}. PT-MAML and STAMP treat q_{sql} as a sequence of words. The decoders of PT-MAML and STAMP have three types of output layers: SQL keywords, column names, and constant values. PT-MAML decodes a sequence guided by a fixed syntax pattern as in [38], while STAMP dynamically selects the type of layers for each decoding step. SyntaxSQLNet, GNN, and IRNet generate a syntax tree. Specifically, SyntaxSQLNet has nine types of modules for the individual parts of SQL, such as aggregation, the WHERE condition, group by, order by, and intersect/union. It dynamically selects one of the types to be generated for each decoding step, guided by a specific subset of the SQL syntax. GNN and IRNet use a grammar-based sequence-to-tree decoder [41, 9, 45, 34] which generates a derivation tree rather than a sequence of words. The grammar-based decoder can immediately check for grammatical errors at every decoding step, allowing the generation of various SQL queries, including join and nested queries, without syntax errors. The other methods belonging to slot-filling use the same fixed template as the syntax pattern of WikiSQL, so they have three types of decoders, each for projection, aggregation, and selection.

During the translation, IRNet generates an intermediate representation named SemQL. The authors argue that, due to significant difference between the SQL (context-free) grammar and the natural language grammar, it is difficult to translate q_{nl} into q_{sql} directly [20]. SemQL is more abstract than SQL and thus easily captures the intent expressed in q_{nl}. For example, the SemQL query does not have the GROUP BY, HAVING, or FROM clauses. Instead, only conditions in the WHERE and HAVING clauses need to be expressed. IRNet has a method for translating a SemQL query into the equivalent SQL query. When columns expressed in SemQL are from multiple tables, the method heuristically adds primary key-foreign key (pk-fk) join conditions during the translation to SQL. Clearly, many rich functionalities in SQL are lost in SemQL, which is an inherent disadvantage of SemQL.

Unlike rule-based methods, the advantage of DL-based methods is that they can learn the background knowledge of training examples, so that it is possible to generate the desired SQL even if q_{nl} is not concrete. For example, in GeoQuery, DL-based methods may understand that a term such as “major cities” means “city.population $>$ 150,000” from the training data.

4.2.3 Technique: input enrichment

Before the translation phase, some NL2SQL methods try to obtain or remove information about inputs. Six of the 16 methods are omitted since there is no input enrichment in these methods.

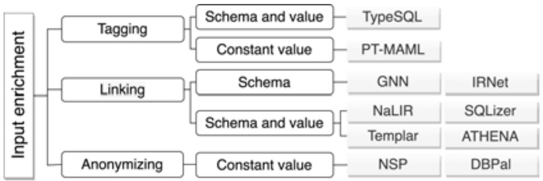

Figure 5: Taxonomy over the ‘input enrichment’ dimension.

Tagging: TypeSQL performs string matching between phrases in q_{nl} and the entities in a database or a knowledge base. Then, for each entity mention, TypeSQL tags it with the name of the corresponding entity (i.e., its type). The tag of each word is embedded into a vector and is concatenated to an embedding vector of the word. A sequence of the concatenated vectors of words in q_{nl} is used as input to the natural language encoder.

PT-MAML performs tagging only for constant values using similarity matching. After finding constant values in q_{nl}, PT-MAML normalizes it to the corresponding entity in V_D. The type of each entity is added to q_{nl} just before the position of the entity.

Linking: GNN calculates a linking score between each word in q_{nl} and the database schema entity (i.e., table or column). This technique, named *schema linking*, was developed to put information about references into the entities of S_D within q_{nl} for the model. The schema linking module has two sub-modules; one is to calculate similarity between word embedding vectors of the question word and schema entity, and the other is to calculate similarity using a neural network with inputs of features obtained by directly comparing words. The linking scores are input to both the q_{nl} encoder and the S_D encoder. Unlike the string-matching-based tagging of TypeSQL, GNN’s schema linking module is trainable and is learned together with the encoder-decoder.

IRNet also uses a schema-linking technique, like GNN, but the linking of IRNet is based on string matching like TypeSQL. Unlike TypeSQL, however, IRNet does not use V_D and allows partial matching. IRNet tags each entity mention with the type of the corresponding entity. IRNet also tags each referenced entity in a database with a unique ID indicating that it is referenced in the question. IRNet uses heuristic rules to choose one of multiple entities. The tags of each word and each database entity are used as input to the q_{nl} encoder and to the S_D encoder, respectively.

All rule-based methods construct mapping between phrases in q_{nl} and entities of S_D or V_D based on similarity matching. NaLIR, SQLizer, and Templar try to find the *best* mapping, while ATHENA considers all mapping candidates in the translation phase. For ranking, NaLIR and SQLizer use a similarity-based score, while Templar uses both similarity information and a given SQL query log.

Anonymizing: NSP and DBPal aim to reduce their vocabulary by using *constant value anonymization*, which converts each constant value into a specific symbol in q_{nl} and q_{sql}. DBPal does not specify how it detects constant values in q_{nl}. NSP uses a simple heuristic algorithm to anonymize a constant value, that is, it finds a value in the given ground truth SQL query by searching for a quotation mark and then matches the value with some words in q_{nl}. In fact, this is not applicable in practice since the ground truth SQL query for q_{nl} is unknown. In an ideal situation, NSP would not store any word in its vocabulary except SQL keywords and database schemas. This means that NSP assumes that constant values can always be anonymized, which is impossible at the current level of technology. NSP will fail to translate if a query has any non-anonymous constant values.

4.2.4 Technique: post-translation

Some NL2SQL methods complete the translation by either filling in the missing information or by refining q_{sql} in the post-translation phase. Figure 6 shows four kinds of post-processing. NSP and DBPal recover anonymized constant values in q_{sql}. ATHENA, NaLIR, and IRNet, which use an intermediate representation, translate the representation to q_{sql}. IRNet and SyntaxSQLNet complete q_{sql} by adding join predicates using a heuristic algorithm. Templar infers join predicates by using the SQL query log. NaLIR and NSP can utilize user feedback about the correctness of the translation. DialSQL can encode a dialogue, so that it refines q_{sql} repeatedly throughout the dialogue with a user.

4.2.5 Technique: training

Every DL-based method trains its model by supervised learning. The following six DL-based methods propose modified learning algorithms. NSP and DBPal augment training data with predefined templates and paraphrasing techniques, such as [16]. In order to fine-tune the selection module, Seq2SQL and STAMP additionally use reinforcement learning, which compares the execution results of the generated SQL statement with that of the corresponding

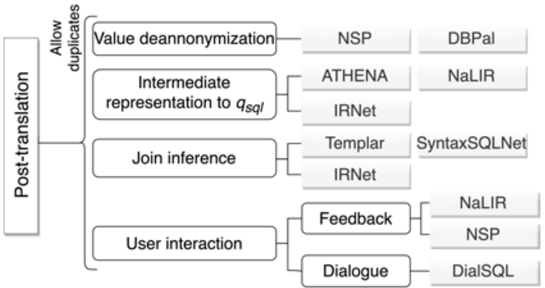

Figure 6: Taxonomy over the 'post-translation' dimension.

ground truth. PT-MAML exploits meta-learning techniques [14] by dividing queries into six tasks according to the aggregation function: sum, avg, min, max, count, and select. DialSQL constructs a simulator to generate dialogues used in training.

Templar uses an SQL query log (i.e., a set of SQL queries) to reinforce the key mapping and join inference. Specifically, given D and an SQL log on D, it decomposes each query in the log into individual query fragments and then constructs a *query fragment graph* for D; the graph statistically captures the occurrence and co-occurrence frequencies of query fragments in the log. Then, it defines specific score functions for both keyword mapping and join inference based on the query fragment graph.

4.2.6 Output

Figure 7 shows a taxonomy over the 'output' dimension. The supported SQL syntax of every method is limited but it is often unclear, especially for rule-based methods. Hence, we group existing methods into three categories, and we indicate which design choice leads to the restriction. There are four reasons: 1) pre-defined syntax patterns and/or types of slots, 2) heuristic translation rules, 3) intermediate representations with limited coverage of syntax, and 4) limited training examples.

All methods generate ranked SQL queries and return the best one. The rule-based methods use specific ranking algorithms as explained in Section 4.2.2, while the DL-based methods use implicit ranking based on softmax scores.

5. VALIDATION METHODOLOGY

5.1 Semantic equivalence

One important issue in NL2SQL is how to measure translation accuracy. Translation accuracy in NL2SQL is measured as the number of correctly translated SQL queries over the total number of test queries. We can judge the correctness by comparing each translated SQL query using an NL2SQL method with the gold SQL query given in the test dataset. In this section, we propose a tool for validating translation results, which is based on the semantic equivalence to overcome the limitations of existing measures.

First, we need a formal definition of semantic equivalence of two SQL queries. Given two SQL queries q_1 and q_2, they are *semantically equivalent* iff they always return the same results on *any* input database instance. Otherwise, they are semantically inequivalent. To correctly compare two SQL queries, we must use the semantic equivalence of the two queries.

Existing NL2SQL methods do not consider the semantic equivalence, or require a lot of manual efforts when measuring accuracy. Figure 8 shows the validation methodology of each method.

5.2 Validation tool

Despite a lot of research on semantic equivalence, the state-of-the-art tools such as Cosette [11, 10] support a limited form of SQL queries. Thus, we have to rely on expensive manual efforts for judging semantic equivalence.

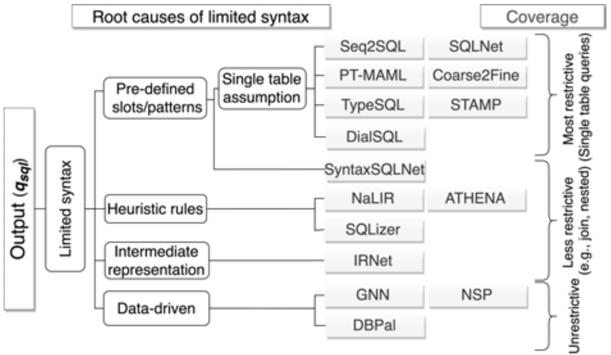

Figure 7: Taxonomy over the 'output' dimension.

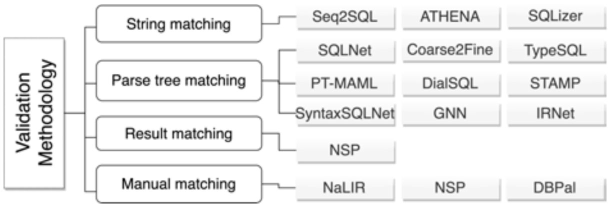

Figure 8: Taxonomy: validation methodology.

In order to reduce the manual effort significantly, we exploit result matching and parse tree matching. Both can be used as early termination conditions in our tool. That is, if two queries have different execution results, we are sure that the queries are semantically inequivalent. If two queries have the same syntactic structure, the queries are semantically equivalent.

We improve the quality of result matching by using database testing techniques [8]. In database testing, we generate datasets (i.e., database instances), run queries over these datasets, and find bugs in database engines. Thus, we generate different datasets so that every query has non-empty results for at least one of these datasets. As we increase the number of database instances, different queries would produce different execution results for some database instances. Thus, we can effectively judge that two queries are semantically inequivalent by executing on the generated datasets.

We improve the quality of syntactic matching by exploiting query rewriting techniques. We use a query rewriter which transforms an SQL query into an equivalent, normalized query. This is done by using various rewrite rules. If two SQL queries are semantically equivalent to each other, the rewriter is likely to transform them into the same normalized query. By comparing two rewritten queries, we can determine whether they are semantically equivalent.

In this paper, we propose a *multi-level framework* for determining the semantic equivalence of two SQL queries (Figure 9). Note that the order of the individual steps does not affect the overall effectiveness (i.e. the number of resolved cases of whether two queries are semantically equivalent or inequivalent). First, we compare the execution results of two SQL queries. If their execution results are not equivalent, we determine that they are semantically inequivalent. However, when the size of a given database is small, it is highly likely that two completely different SQL queries return the same empty results. We resolve this problem by comparing execution results on the generated datasets using the database testing technique as well as on the given database. Next, we use an existing prover, such as Cosette, that exploits automated constraint solving and interactive theorem proving and returns a counter example or a proof of equivalence for a limited set of queries. For queries that are not supported by the prover, we use the query rewriter in a commercial DBMS and compare the parse trees of the two rewritten SQL queries. Specifically, two queries are equivalent if their parse trees are structurally identical; i.e., every node in one parse tree has the corresponding matching node in the other. After a series of comparisons, unresolved cases can be manually verified. Alternatively, the tool determines whether any unresolved cases are semantically inequivalent, which would lead to slightly incorrect decisions. In our extensive experiments, our tool achieved 99.61% accuracy.

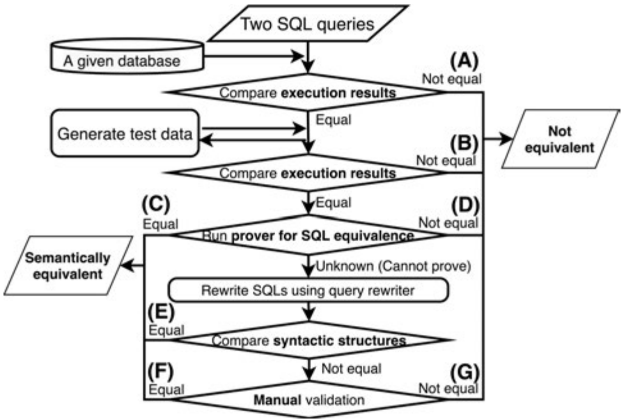

Figure 9: A flowchart of our multi-level framework.

We implement the multi-level validation tool by using 1) Evo-SQL [8] as a database instance generator, 2) Cosette as a prover, and 3) the query rewriter in IBM DB2. The manual validation is done by graduate-level computer science students.

6. EXPERIMENTS

In this section, we first show the effectiveness of the proposed validation tool in Section 5, and then evaluate the performance of the eleven methods reviewed in Section 4. [35, 43, 4, 53, 44] are excluded from our evaluation since the authors did not disclose their source codes or binary executables. This performance evaluation consists mainly of two parts: 1) experiments using *simple queries* (following the syntax pattern in WikiSQL) (Section 6.2), and 2) experiments using *complex queries* (Section 6.3).

The main goals of this experimental study are as follows: 1) We show that all existing accuracy measures are misleading. 2) We evaluate the effectiveness of our validation tool. 3) We evaluate the performance of the eleven NL2SQL methods by using 13 benchmarks including a new benchmark (WTQ). We additionally use TPC-H [1]. 4) We analyze translation errors in depth and identify the advantages and disadvantages of each method.

Benchmarks. We use a total of 13 NL2SQL benchmarks including a newly released benchmark, WTQ. We additionally use TPC-H. The WTQ benchmark consists of 9,287 randomly sampled questions from the existing WikiTableQuestions [30]. WikiTableQuestions has a salient feature, compared to the existing benchmarks: *complex* questions in *different* domains. WikiTableQuestions consists of questions for web tables on various domains, and it has complex queries that include various operations such as ordering, grouping, and nested queries. However, the WikiTableQuestions benchmark contains q_{nl}'s and their execution results without gold q_{sql}'s. Thus, we collect q_{sql}'s through crowd-sourcing.

Table 1 shows the statistics of the 13 benchmarks. If the data split (i.e., training, validation, and test examples) is published, we use the published one, otherwise we perform the random split with the ratio of 11:1:5.6, for training, validation, and test examples, respectively. For the Advising benchmark, we use two versions of splits published in [13], namely question-based split and query-based split. The question-based split is a traditional method used in the other benchmarks. This method regards each pair (q_{nl}, q_{sql}) as a single item so that each pair belongs to either a training set, a validation set, or a test set. Meanwhile, the query-based split ensures that each q_{sql} belongs to either a training set, a validation set, or a test set. We use validation examples for evaluating on Spider since test examples of Spider are not published. For MAS, IMDB, and YELP, we manually wrote a q_{sql} for each q_{nl}, since these benchmarks do not contain gold SQL queries. For ATIS, we manually removed 93 incorrect examples.

Table 1: The statistics for the NL2SQL benchmarks.

Benchmark	Total queries	Training queries	Validation queries	Test queries	Tables	Rows	Size (MB)
WikiSQL [52]	80654	56355	8421	15878	26531	459k	420
ATIS [33, 12, 22, 51]	5317	4379	491	447	25	162k	39.2
Advising [13] (querysplit)	4387	2040	515	1832	15	332k	43.8
Advising [13] (questionsplit)	4387	3585	229	573	15	332k	43.8
GeoQuery [49, 50, 32, 17, 22]	880	550	50	280	7	937	0.14
Scholar [22]	816	498	100	218	10	144M	8776
Patients [4]	342	214	19	109	1	100	0.016
Restaurant [36, 32]	251	157	14	80	3	18.7k	3.05
MAS [25]	196	123	11	62	17	54.3M	4270
IMDB [43]	131	82	7	42	16	39.7M	1812
YELP [43]	128	80	7	41	7	4.48M	2232
Spider [48]	9693	8659	1034	-	873	1.57M	184
WTQ [30] **[Ours]**	9287	5804	528	2955	2102	58.0k	35.6

Since a simple query is defined for a given table, it can only be found in WikiSQL, Patients, and WTQ where a table is specified for each query. Table 2 shows the number of simple queries, indicated by the suffix "-s" to distinguish them from the original benchmarks. Note that WikiSQL-s is same as WikiSQL.

Table 2: The statistics for simple queries.

Benchmark	Training queries	Validation queries	Test queries	Total queries
WikiSQL-s	56355	8421	15878	80654
Patients-s	61	8	33	102
WTQ-s	1090	63	315	1468

Experimental setup. We use the source codes provided by the authors of each method. In order to show the effect on accuracy caused by the use of database entities, we evaluate the accuracy of TypeSQL with database values (TypeSQL-C) and without them (TypeSQL-NC). For Templar, we evaluate the augmented NaLIR as in [3] by extending its SQL parser to support the benchmarks used in our experiments. We use q_{sql}'s in training and validation sets for the SQL log. For Spider, we use only the training set as the log. We do not evaluate the augmented Pipeline since it requires manual parsing of q_{nl} into a set of keywords for all examples. For NaLIR and Templar, which assume that the SQL execution result is a set, we measure their accuracy by ignoring the DISTINCT keyword in the SELECT clause. We fixed a bug in the source code of PT-MAML that caused an error if a test query contains words not in the training data. For DL-based methods, if the authors published a hyper-parameter setting for a particular dataset, we used the same setting. Otherwise, we performed extensive grid searches using the hyper-parameters in Table 3, and repeated all experiments five times. For NSP on ATIS, the accuracy variance was relatively high, so we repeated the experiment ten times. The hyper-parameter tuning took about 3,600 GPU hours.

[1] http://www.tpc.org/tpch/

Coarse2Fine raises an error if all SQL queries in a mini-batch do not contain a WHERE clause due to the numerical instability in its algorithm. This error occurred in about 40% of experiments during tuning hyper-parameters on WTQ-s. We applied the following stopping criteria: Each model was trained for 300 epochs, and we set the checkpoint at the epoch that had the lowest validation loss. Our codes and benchmarks are publicly available at https://github.com/postech-db-lab-starlab/NL2SQL.

Table 3: Hyper-parameters.

The dimension of a word embedding vector	{100, 300}	Learning rate (LR)	{1e-3, 1e-4}
The number of layers	{1, 2}	Batch size	{64, 200}
Dropout rate	{0.3, 0.5}	LR decay	{1, 0.98, 0.8}
The dimension of context vector		{50, 300, 600, 800}	

Reproduction of DL-based methods. Table 4 contains accuracy results of the original papers and our experiments with the original source codes. Note that existing codes for measuring accuracy have bugs, but we use them solely for the reproduction purposes. For example, when a translated query contains syntax errors, acc$_{ex}$ in NSP judges that the query returns an empty result. Thus, when the corresponding gold SQL returns an empty result, acc$_{ex}$ in NSP determines that they are equivalent, which is wrong. acc$_{syn}$ of GNN/IRNet has bugs in comparison with join predicates and group by columns. Except for the results of GNN on Spider, we reproduced all experiments with an accuracy difference of at most 2.14%. The small discrepancies occurred whenever the authors did not specify the random seed they used. For GNN, the authors published a hyper-parameter set which is better than that which was used in their original paper. Hence, our result shows a higher accuracy.

Table 4: Reproduction of the accuracy in original papers (%).

Method	Benchmark	Original	Ours	Method	Benchmark	Original	Ours
NSP	GeoQuery	82.5	80.36	**TypeSQL-C**	WikiSQL	75.4	74.97
NSP	ATIS	79.24	78.13	**PT-MAML**	WikiSQL	62.8	62.72
NSP	Scholar	67	67.43	**Coarse2Fine**	WikiSQL	71.7	70.78
TypeSQL-NC	WikiSQL	66.7	67.06	**Syntax-SQLNet**	Spider	18.9	17.4
SQLNet	WikiSQL	61.3	61.27	**GNN**	Spider	40.7	**47.2**
Seq2SQL	WikiSQL	51.6	51.33	**IRNet**	Spider	53.2	53.0

6.1 Validating translation results

We show that all accuracy measures used in the previous studies are misleading. We define acc$_{sem}$ as N_{sem}/N, where N_{sem} is the number of generated queries that are semantically equivalent to the gold SQL queries, and N is the total number of queries. acc$_{ex}$ is an accuracy measure comparing the execution results of two SQL queries on a given database. acc$_{str}$ is calculated by comparing two SQL queries through string matching. acc$_{syn}$ is based on syntactic equivalence of two SQL queries. For the first step in Figure 9 and acc$_{ex}$, we set an execution timeout to 30 seconds for each q_{sql}.

Table 5 shows the comparison results of accuracy measures on various benchmarks. In this experiment, we measure accuracy for the translation results of NSP. This experiment aims to compare various accuracy measures, and it does not matter which NL2SQL method is used. Using simple queries, however, the difference among measures may not be revealed. Thus, we select NSP, which can translate complex queries, without loss of generality. The results show that the measurement error of existing measures is significant. acc$_{syn}$, acc$_{ex}$, and acc$_{str}$ differ from acc$_{sem}$ by up to 59.96% on ATIS, 27.58% on Advising (questionsplit), and 70.25% on ATIS, respectively. These would be even larger as SQL queries become more complex.

Table 5: Comparison of accuracy measures (%).

Benchmark	acc$_{sem}$	acc$_{syn}$	acc$_{ex}$	acc$_{str}$
WikiSQL	9.47	9.47	25.67	0.0
ATIS	70.25	10.29	77.18	0.0
Advising (querysplit)	0.27	0.27	29.8	0.27
Advising (questionsplit)	44.85	44.15	72.43	44.15
GeoQuery	70.0	69.29	78.21	68.93
Scholar	48.17	37.61	46.79	33.49
Patients	69.72	68.81	69.72	68.81
Restaurant	63.75	63.75	76.25	56.25
MAS	51.61	46.77	53.23	46.77
IMDB	16.67	16.67	19.05	16.67
YELP	0.0	0.0	19.51	0.0
Spider	0.0	0.0	0.87	0.0
WTQ	2.06	1.83	5.41	0.03

We now show the effectiveness of our multi-level validation tool. First, we demonstrate that each step of our tool is effective by calculating the number of resolved cases at each step. We perform this evaluation on the translation results of NSP. To ensure generality, we excluded WikiSQL and Spider from this experiment; WikiSQL has simple queries only which are easy to determine semantic equivalence. NSP shows acc_{ex} near to 0% on Spider so that it is easy to determine semantic inequivalence on Spider. We evaluate on the other eleven benchmarks.

Figure 10 shows the ratio of resolved pairs of the translated and gold queries at each step for each benchmark. Each legend corresponds to one of the six flows shown in Figure 9. As shown in Figure 10, the ratio of resolved cases at each step varies largely depending on the dataset. The average ratio of resolved cases on eleven benchmarks at "(A)," "(B)," "(C)," "(D)," "(E)," "(F)," and "(G)" is 69.59%, 10.27%, 6.39%, 0.77%, 9.64%, 0.39%, and 2.95%, respectively. The sum of the ratios of "(F)" and "(G)" is the percentage of queries unresolved by our tool's automatic process; 3.34% of the total pairs were unresolved by the automated process. While we must rely on the manual inspection to achieve 100% accuracy, our tool achieves 99.61% accuracy on average in the eleven benchmarks by determining all of unresolved cases to be inequivalent. Although the effect of the proposed tool may vary depending on the dataset, it has a very small error (0.39%) on average.

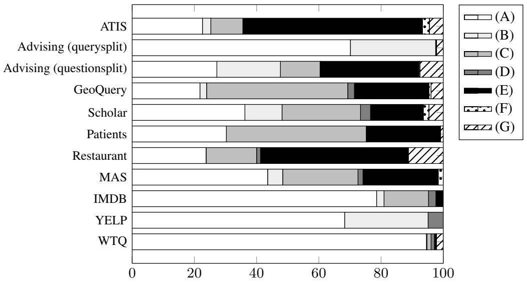

Figure 10: Resolved cases (%) for each step of our validation tool.

We further measure the number of unresolved cases when activating only one step at a time, i.e., each step corresponding to "(A)," "(B)," "(C)," and "(D)," or "(E)." The average ratio of unresolved cases on the eleven benchmarks is 30.41%, 48.76%, 83.87%, and 83.69%, respectively. These values are much larger than the value of 3.34% which uses all steps. The results show that our tool consists of complementary steps, and integrating these steps is an effective approach for removing unresolved cases.

6.2 Experiments using simple queries

Since SyntaxSQLNet, GNN, and IRNet generate SQL queries ignoring constant values, we compare their translation results with gold SQL queries without constant values. Given a generated SQL query and a gold SQL query, we replace all constant values in both with indicators. Then, we compare the two queries using our validation tool. We denote the accuracy as acc$_{-val}$.

Table 6 shows acc$_{sem}$ and acc$_{-val}$ of all methods on the three benchmarks which have simple queries. In order to analyze errors in depth, we measure the accuracy for each part of the SQL query (Table 7). We report accuracy values for projected columns (acc$_{sel}$), aggregation functions in the SELECT clause (acc$_{agg}$), and columns, operators, and constant values in the WHERE clause (acc$_{wh,col}$, acc$_{wh,op}$, and acc$_{wh,val}$, respectively).

Rule-based methods: The rule-based methods show low accuracy on all benchmarks. This is mainly due to the mapping table used in the translation process, which significantly limits q_{nl}'s that can be handled. For example, NaLIR mis-translates the question (A) in Table 8, since the mapping table does not have a mapping from "longest" phrase to *"MAX"* function. The accuracy of NaLIR might be slightly improved by extending the mapping table for each benchmark, but it must be done manually and requires considerable effort. Although Templar enhances the mapping by using an SQL log, such a linguistic mapping cannot be captured without using q_{nl}'s together. Since this fundamental problem is left unsolved, it does not improve accuracy in this experiment. This is because all three benchmarks have little or no training query per table, so that using a query log is not helpful. The join path generation technique of Templar also has no impact for single-table queries.

Table 6: Accuracy on simple queries (%).

	WikiSQL		Patients-s		WTQ-s	
	acc$_{sem}$	**acc$_{-val}$**	**acc$_{sem}$**	**acc$_{-val}$**	**acc$_{sem}$**	**acc$_{-val}$**
NaLIR	0.49	0.50	0.00	0.00	1.59	1.59
Templar	0.49	0.50	0.00	0.00	1.59	1.59
NSP	9.49	10.84	81.82	81.82	0.32	0.32
SyntaxSQLNet	-	49.91	-	21.21	-	21.59
Seq2SQL	51.33	54.02	33.33	33.33	1.27	2.54
PT-MAML	60.65	63.54	39.39	39.93	18.41	20.00
SQLNet	61.27	67.92	33.33	39.39	1.27	2.22
TypeSQL-NC	67.14	72.50	45.45	63.64	5.71	9.84
Coarse2Fine	70.78	72.56	42.42	45.45	2.54	6.98
TypeSQL-C	74.97	76.52	48.48	51.52	10.16	14.60
IRNet	-	73.60	-	57.58	-	44.13
GNN	-	79.01	-	60.61	-	4.44

Adaptability to unseen databases: If a method performs well for unseen databases, we say that the method has high adaptability. NSP performs poorly for WikiSQL compared to the other DL-based methods since NSP has limited adaptability to new databases. In NSP, entities in S_D are stored as a part of the output vocabulary. Therefore, NSP cannot support queries on tables not seen in training data. WikiSQL has 26,531 tables, and the vocabulary size is 55,294. The large size of vocabulary also results in significant performance degradation of NSP. On the other hand, the other methods need not maintain large vocabularies, since they use a schema as an input and the pointer mechanism. Note that in Patients, the output vocabulary size is 45. Another issue is that NSP cannot take a particular database as input. This is due to the inability of NSP to utilize the additional information about the table. We tested a modified NSP which selects the given table T for each q_{nl} in the FROM clause, and selects columns from T. However, acc$_{sem}$ of NSP stays at 11.5%, which was still far lower than the other DL-based methods. Note that these two problems also occur on WTQ-s.

Robustness with small datasets: The data augmentation technique of NSP is helpful for Patients-s since the number of training queries in Patients-s is relatively small (61 queries). In order to show the effectiveness of the data augmentation, we conduct additional experiments by applying the technique to all DL-based methods. The numbers of training examples after augmentation for Patients and Patients-s are 1,077 and 500, respectively. The results show that most methods benefit from the data augmentation technique (0%-27.27%).

Table 7: Partial accuracy on simple queries (%).

Benchmark	Method	acc$_{sel}$	acc$_{agg}$	acc$_{wh,col}$	acc$_{wh,op}$	acc$_{wh,val}$
WikiSQL	**NaLIR**	0.80	0.96	0.74	0.83	0.77
	Templar	0.80	0.96	0.74	0.83	0.77
	NSP	41.92	75.99	32.20	67.69	37.66
	SyntaxSQLNet	76.38	89.54	68.21	92.51	-
	Seq2SQL	88.52	89.75	65.43	92.54	84.05
	PT-MAML	85.70	88.63	82.23	90.12	85.39
	SQLNet	90.75	90.16	81.93	92.57	81.57
	TypeSQL-NC	92.51	89.94	86.44	93.97	86.01
	Coarse2Fine	91.46	90.41	86.01	96.13	92.87
	TypeSQL-C	92.17	90.12	92.13	96.12	94.97
	IRNet	93.37	89.71	85.43	93.69	-
	GNN	94.43	90.20	92.35	94.01	-
Patients-s	**NaLIR**	6.06	6.06	15.15	15.15	15.15
	Templar	3.03	6.06	6.06	6.06	6.06
	NSP	93.94	90.91	93.94	84.85	93.94
	SyntaxSQLNet	72.73	51.52	66.67	72.73	-
	Seq2SQL	87.88	63.64	60.61	63.64	60.61
	PT-MAML	93.94	57.58	81.82	75.76	93.94
	SQLNet	81.82	63.64	90.91	84.85	84.85
	TypeSQL-NC	96.97	69.70	90.91	90.91	72.73
	Coarse2Fine	87.88	51.52	87.88	90.91	96.97
	TypeSQL-C	90.91	72.73	81.82	84.85	96.97
	IRNet	72.73	66.67	75.76	78.79	-
	GNN	87.88	78.79	93.94	81.82	-
WTQ-s	**NaLIR**	2.22	3.49	1.59	1.90	1.90
	Templar	2.22	3.49	1.59	1.90	1.90
	NSP	16.51	33.65	11.43	29.84	9.21
	SyntaxSQLNet	43.17	60.00	34.29	50.48	-
	Seq2SQL	26.35	85.08	28.89	65.71	27.30
	PT-MAML	36.51	49.52	38.10	43.17	40.63
	SQLNet	26.03	88.57	27.30	66.35	28.25
	TypeSQL-NC	40.00	87.30	37.78	67.62	32.70
	Coarse2Fine	27.30	86.67	27.94	76.19	18.41
	TypeSQL-C	39.68	90.16	35.56	68.25	36.51
	IRNet	62.86	72.38	51.43	65.40	-
	GNN	21.27	50.16	13.33	31.75	-

Even after applying the data augmentation technique to all methods, NSP still has the best performance (81.82%) for Patients-s. The first reason is that 3.3% of test queries in Patients-s cannot be generated in most DL-based methods except for NSP, GNN, and IRNet. Since they generate constant values by using the pointer to words mechanism in q_{nl}, they generate an incorrect q_{sql} if a constant value does not appear in q_{nl}. For example, the question (B) in Table 8 doesn't have the constant value "flu" which is a part of the gold q_{sql}. On the other hand, NSP can generate constant values correctly if the values exist in the output vocabulary. Furthermore, it was helpful for NSP to use all training queries for training. The accuracy of NSP trained with simple queries stays at 69.7%, which is the same level as the accuracy of TypeSQL-NC with the data augmentation. Even though SyntaxSQLNet, GNN, and IRNet do not suffer from the problems described above, their accuracy is lower than that of NSP. This shows that the simple model of NSP can be more powerful on benchmarks with small numbers of training examples. For example, given the question (A) in Table 8, NSP predicts the correct column in the SELECT clause. NSP can infer the mapping between 'hospitalization period' in q_{nl} and the column 'length_of_stay' from four training examples. However, SyntaxSQLNet, GNN, and IRNet select the column 'age,' which is the most frequent one in the training data. Since the other methods are clearly inferior to these methods, we omit their detailed analysis due to space limitation.

NL complexity: The complexity of natural language is affected by various factors such as linguistic diversity in questions, a variety of domains, target operations of SQL, and number of sentences. WTQ has high complexity since it has diverse forms of questions including multiple independent clauses, 2,102 tables from diverse domains, and complex queries containing group by, order by, and nested queries. In this section, we first examine the first two, linguistic diversity in questions and the variety of domains, with simple queries. The other two points will be discussed in Section 6.3.

All methods have significantly poorer performance on WTQ-s than WikiSQL. Since WTQ-s has a wide variety of q_{nl}'s, various (unseen) questions are in the test data. For example, the question (C) in Table 8 has multiple independent clauses and negation, where all methods fail to translate properly. Furthermore, complicated column names/constant values in a number of tables result in more complex questions. For example, in the question (D) in Table 8, the column name *'number of autos da fe'* is not English. Most DL-based methods do not understand any column name which cannot be found in the word embedding matrix.

Aligning table/column references in q_{nl} to S_D: Exploiting the associations between q_{nl} and S_D can greatly affects the accuracy. On WikiSQL, GNN has the highest value of acc$_{-val}$ (79.01%), and TypeSQL-C and IRNet hold the second and third ranks, respectively. According to Table 7, GNN accurately predicted the column names compared to the other methods. This is due to the schema linking technique used by GNN. TypeSQL-C and IRNet also show better accuracy of column prediction than the others except GNN. Both use the tag information obtained by leveraging a database schema. In conclusion, how well we exploit the association greatly affects the accuracy in WikiSQL.

IRNet has significantly higher accuracy than all the other methods on WTQ-s. acc$_{wh,op}$ or acc$_{agg}$ of IRNet is similar to or less than those of other methods. However, IRNet shows significantly better accuracy in generating column names (acc$_{sel}$ and acc$_{wh,col}$). IRNet chooses columns more accurately than other methods, especially when the column names are long, and there are several similar columns in the table. For example, IRNet is the only method that correctly translates the question (D) in Table 8. IRNet performs tagging in advance by comparing table/column names and phrases in q_{nl} during pre-processing, which can be a great help for WTQ-s. TypeSQL and GNN have similar modules, but they are not suitable for WTQ-s which has long and varied column names containing unusual words; TypeSQL does not allow partial string matching and does not perform schema linking but tagging, and the schema linking of GNN is based on pre-trained word embedding.

Effectiveness of learning algorithms: We perform an additional experiment with Seq2SQL and PT-MAML, which uses reinforcement learning and meta learning respectively, after changing their learning algorithm to supervised learning with teacher forcing [40] which is used by the other DL-based NL2SQL methods. Without their learning algorithms, Seq2SQL and PT-MAML showed averages of 2.9% and 4.9% degradation in accuracy, respectively. Both methods gain their accuracy on average from their learning algorithms, but they are not superior to other DL-based methods.

6.3 Experiments using complex queries

We evaluate the existing methods on all test queries in twelve benchmarks, excluding WikiSQL which has simple queries only. Table 9 shows the accuracy of six methods, NaLIR, Templar, NSP, SyntaxSQLNet, GNN, and IRNet. We exclude the other methods from this experiment since they can support simple queries only. In summary, all six methods have serious errors when translating complex queries in the twelve benchmarks as seen in Table 9.

Rule-based methods: Rule-based methods show low accuracy in general, mostly due to the same issue discussed in Section 6.2. While Templar has the same or better accuracy than NaLIR on most benchmarks, it shows lower accuracy than of NaLIR on MAS. We observe that, for MAS, Templar often fails to find the correct mapping. Given two query fragments, qf_1 and qf_2, if qf_2 has larger co-occurrences in the query log than qf_1, the mapping of Templar can be distracted to choose qf_2, even if qf_1 is more semantically aligned with a given question. For question (E) in Table 8, 'domain.name' is qf_1 and 'publication.title' is qf_2 in our example. Templar incorrectly maps 'area' in q_{nl} to 'publication.title,' while NaLIR correctly maps it to 'domain.name.'

Adaptability: On Spider, SaI methods are more adaptable than the others. Spider is a full cross-domain benchmark, that is, the underlying databases for validation queries are not in the training data. Thus, NSP cannot support queries in Spider at all, whereas SaI methods, SyntaxSQLNet, GNN, and IRNet can, as explained in Section 4.2.2. However, SyntaxSQLNet showed lower performance than GNN and IRNet. The accuracy of table prediction of SyntaxSQLNet is 41.3%, which is much lower than 67.0% and 73.7% of GNN and IRNet. GNN and IRNet utilize the information of association between q_{nl} and S_D and design a schema encoder to handle various database schemas, whereas SyntaxSQLNet does not. That is, even among SaI methods, adaptability greatly varies depending on the way to treat S_D. The SQL log-based technique of Templar has no impact on Spider, since it cannot utilize the query fragment graph of the training SQL queries.

Robustness with small datasets: We observed that SyntaxSQLNet, IRNet and GNN are not trained properly in some benchmarks having small numbers of examples. In particular, in Scholar, IRNet was not properly trained, so we could not obtain any meaningful trained model. In IMDB, SyntaxSQLNet makes the same query, "SELECT T2.name FROM genre as T1 JOIN actor as T2 WHERE T1.genre = ⟨value⟩ OR T2.name = ⟨value⟩" in 50% of test cases. We also observed that GNN was not properly trained in Scholar, MAS, IMDB, or YELP; regardless of the question, GNN produced an odd output such as *"SELECT DISTINCT company.name FROM company WHERE company.id = company.id AND company.id AND company.id = ⟨value⟩"*. This trend is evident when a benchmark has fewer training queries relative to the high variety of questions.

SQL construct coverage: SyntaxSQLNet, GNN, and IRNet support limited forms of q_{sql} for the following reasons. SyntaxSQLNet uses limited types of slots as explained in Section 4.2.2. The grammar-based decoders of GNN and IRNet can generate more general queries, but their grammars support a subset of the SQL syntax. SemQL of IRNet has a much smaller coverage than SQL. 43.9% of test queries in the twelve benchmarks cannot be supported by IRNet. Most seriously, 80.1% of test queries in ATIS, 47.1% of test queries in Advising (querysplit), and 54.5% of test queries are not supported by IRNet. The public implementation of GNN only supports the minimal syntax of SQL to support Spider; it does not support queries having OR conjunctions, parentheses, or join conditions in the WHERE clause. It also does not support LIMIT statements without ORDER BY or GROUP BY. Neither does it support the IS NULL operator. We have extended the grammar of GNN to support all of them. However, 37.4% of test queries in twelve benchmarks are still unsupported due to the limitations of the internal data structure used by GNN. For example, it cannot handle self-join, correlated nested queries, or arithmetic operations having more than two operands. IRNet has similar limitations, too.

As queries become more complex and diverse, there has been an increase in the case where NSP generates invalid queries that cannot be executed. Table 10 shows types of translation error by NSP. Due to space limitation, we report the results on four benchmarks with the most diverse error cases. 59.5% of these invalid queries are due to translating incorrect table names or column names. An SQL query that accesses a table that is not in the FROM clause cannot be executed. For example, the q_{sql} for question (G) in Table 8

Table 8: Translation examples. Mis-translated parts are colored in red, where absent parts are shown with strikethroughs.

q_{nl}	Dataset		q_{sql}
(A) Display the longest hospitalization period.	Patients	(Gold, NSP)	SELECT MAX(length_of_stay) FROM patients
		(NaLIR)	SELECT ~~MAX~~ length_of_stay FROM patients
		(GNN, IRNet)	SELECT MAX(age) FROM patients (SyntaxSQLNet) SELECT MIN(age) FROM patients
(B) What is the cumulation of durations of stay of inpatients where diagnosis is influenza?	Patients	(Gold)	SELECT SUM(length_of_stay) FROM patients WHERE diagnosis = 'flu'
		(Coarse2Fine)	SELECT SUM(length_of_stay) FROM patients WHERE diagnosis = 'influenza'
(C) Who ran in the year 1920,but did not win?	WTQ	(Gold)	SELECT loser FROM *T* WHERE year = 1920
		(TypeSQL-C)	SELECT year FROM *T* WHERE winner = '1920'
		(PT-MAML)	SELECT 'number of votes winner' FROM *T* WHERE year = 1920
		(SyntaxSQLNet)	SELECT year FROM *T* WHERE year = <val> OR winner = <val>

		(IRNet)	SELECT year FROM *T* WHERE year = <val>INTERSECT
			SELECT year FROM *T* WHERE year $\neq$ <val>

(D) Which Spanish tribunal was the only one to not have any autos da fe during this time period?	WTQ	(Gold)	SELECT tribunal FROM *T* WHERE 'number of autos da fe' = 0
		(IRNet)	SELECT tribunal FROM *T* WHERE 'number of autos da fe' = <val>
		(GNN)	SELECT tribunal FROM *T* ~~WHERE 'number of autos da fe' = 0~~
		(SyntaxSQLNet)	SELECT tribunal FROM *T* WHERE tribunal = <val>
		(TypeSQL-C)	SELECT 'executions in effigie' FROM *T* WHERE penanced = ''
(E) Return me the area of PVLDB	MAS	(Gold, NaLIR)	SELECT DISTINCT d.name FROM journal j, domain_journal dj, domain d WHERE j.jid = dj.jid AND dj.did = d.did AND j.name = 'PVLDB'
		(Templar)	SELECT DISTINCT p.title FROM journal j, publication p WHERE j.jid = p.jid AND j.name = 'PVLDB'

(F) How is the workload in EECS 751?	Advising	(Gold)	SELECT DISTINCT pc.workload FROM course c, program_course pc
			WHERE pc.course_id = c.course_id AND c.department = 'EECS' AND c.number = 751

	(querysplit)	(GNN)	SELECT DISTINCT name FROM program WHERE name = <val> AND name = <val>

	(question-split)	(GNN)	SELECT DISTINCT pc.workload FROM course c, program_course pc
			WHERE pc.course_id = c.course_id AND c.department = <val> AND name = <val>

		(NSP)	SELECT DISTINCT pc.workload FROM course c, program_course pc
			WHERE pc.course_id = c.course_id AND c.department = 'EECS' AND c.number = 559

(G) Number of papersin SIGIR conference	Scholar	(Gold)	SELECT DISTINCT count(p.paperId) FROM paper p, venue v WHERE p.venueId = v.venueId AND v.venueName = 'SIGIR'
		(NSP)	SELECT DISTINCT writes.authorId, count(p.paperId) FROM paper p, venue v WHERE p.venueId = v.venueId AND v.venueName = 'SIGIR'

(H) Who has been an opponent more often, Guam or Bangladesh?	WTQ	(Gold)	SELECT opponents FROM *T* WHERE opponents = 'Guam' OR opponents = 'Bangladesh'
			GROUP BY opponents ORDER BY COUNT(opponents) DESC LIMIT 1

		(IRNet)	SELECT venue FROM *T* WHERE opponents = <val> OR opponents = <val> ~~GROUP BY opponents~~ ORDER BY competition DESC LIMIT 1
		(SyntaxSQLNet)	SELECT venue FROM *T* WHERE opponents LIKE <val> AND opponents LIKE <val> ~~GROUP BY opponents~~ ORDER BY competition DESC LIMIT 1

Table 9: Accuracy on all test queries in various benchmarks (%).

Benchmark	NaLIR	Templar	NSP		SyntaxSQLNet	GNN	IRNet
	acc_{sem}	acc_{sem}	acc_{sem}	acc_{-val}	acc_{-val}	acc_{-val}	acc_{-val}
ATIS	0.0	0.0	70.25	76.06	0.67	5.13	8.71
Advising (querysplit)	0.0	0.0	0.27	5.4	0.38	1.75	0.16
Advising (questionsplit)	0.0	0.0	44.85	79.76	0.0	23.73	13.79
GeoQuery	0.36	0.0	70.0	72.5	49.64	38.21	60.36
Scholar	0.0	0.0	48.17	55.96	1.38	1.38	0.0
Patients	0.0	0.0	69.72	69.72	10.09	53.21	46.79
Restaurant	0.0	0.0	63.75	63.75	0.0	0.0	66.25
MAS	32.26	12.90	51.61	54.84	0.0	0.0	14.52
IMDB	0.0	11.90	16.67	21.43	0.0	2.38	14.29
YELP	4.88	9.76	0.0	9.76	0.0	0.0	7.32
Spider	0.39	0.39	0.0	0.0	14.70	45.16	50.87
WTQ	0.47	0.47	2.06	3.32	10.93	2.06	15.47

accesses the column *writes.authorId*, while the table *writes* is not in the FROM clause. Therefore, this query results in a syntax error. Many queries are in this case, and most of these errors occur in complex queries involving join. NSP memorizes join conditions that appear in the training and generates one of the memorized join conditions. This simple approach does not consider the relationship among FROM, SELECT, and join, therefore it often generates inappropriate join conditions.

Table 10: Error cases (NSP).

Benchmark	Single table queries			Multiple table queries				
	Correct	Incorrect		Correct	Incorrect			
		syntax error	others		syntax error	wrong from/join	wrong nesting	others
ATIS	46	0	15	268	43	3	53	19
Advising (questionsplit)	28	4	23	229	23	35	27	204
GeoQuery	134	1	21	62	15	6	22	19
Scholar	0	6	1	105	39	13	4	50

Generalizability to unseen examples: If a method performs well for unseen SQL query patterns, we say that the method has high generalizability. On Advising (querysplit), the accuracy of all DL-based methods is close to zero. The query-based split is difficult to handle, since all the SQL queries in the test data are ones that have never been seen in the training [13]. For the question (F) in Table 8, for example, GNN generates q_{sql} completely differently from the correct translation on Advising (querysplit), while it correctly translates on Advising (questionsplit).

NL complexity: Diversity of q_{sql} typically increases the complexity of q_{nl}. The number of possible expressions in q_{nl} increases exponentially with the number of SQL operations in the corresponding q_{sql}. For example, to properly translate question (H), one has to understand both 'more often' and 'Guam or Bangladesh.' No methods, including IRNet, that performed best on WTQ-s, understand such varied patterns, and they suffer from low accuracy. Therefore, their accuracy has been decreased on WTQ compared to WTQ-s.

Constant value anonymization: On Advising (questionsplit), the difference between acc_{sem} and acc_{-val} of NSP is 34.91%, which is significant. This is the percentage of test queries that NSP correctly translates except constant values. For example, NSP mis-translates the question (F) in Table 8 by one constant value. This is because the correct value '751' is not anonymized. This observation suggests the following two points: 1) the constant value anonymization technique of NSP is limited, and 2) there is a research opportunity to precisely translate constant values.

Handling multiple sentence questions: In order to further analyze how well these methods translate more complex and realistic queries, we conducted an additional experiment using the famous benchmark, TPC-H. TPC-H has a total of 22 queries to portray the activity of a product supplying enterprise. Since TPC-H doesn't have enough queries to use for training, we train DL-based methods using Spider, and then test them on the TPC-H queries. In order for DL-based methods to better understand the database schema of the TPC-H dataset, we changed the format of table and column names slightly by removing meaningless symbols and separating each word. For example, we modify the column name "L_EXTENDEDPRICE" to "extended price". We use the given explanation of each query named Business Question in the official documentation as q_{nl}. We rephrase them in a narrative format. Note that all q_{nl} in TPC-H are multiple-sentence questions.

Surprisingly, all methods show *0% accuracy* on TPC-H. NaLIR relies on the manually-built mapping table and processes a single sentence only, so it cannot handle long, complex q_{nl}s in TPC-H. Seq2SQL, SQLNet, PT-MAML, TypeSQL, and Coarse2Fine also do not support any question since there is no simple query in TPC-H. NSP doesn't work either since the database (i.e., the test dataset (TPC-H)) is different from the training dataset (Spider). Templar also does not benefit from using the given query log. SyntaxSQLNet, GNN, and IRNet claim to support various queries in cross-domain environments. However, they also fail to translate all queries in TPC-H correctly. Table 11 shows the error cases of the three methods. IRNet cannot support all queries in TPC-H due to the limited coverage of SemQL. SyntaxSQLNet and GNN can support two (Q2 and Q18) and three (Q2, Q6, and Q18) of 22 queries, respectively, but it mistranslates all the queries. For example, Q6 is for *"SELECT SUM(lineitem.extended _price * lineitem.discount) FROM lineitem WHERE lineitem.ship_date* $\geq$ *'1994-01-01' AND lineitem.ship_date* $<$ *'1994-02-01' AND lineitem. discount BETWEEN 0.05 AND 0.07 AND lineitem.quantity* $<$ *24"*, GNN answers *"SELECT SUM(lineitem.quantity) FROM lineitem WHERE lineitem.discount BETWEEN* $\langle$*value*$\rangle$ *AND* $\langle$*value*$\rangle$*"*, which is completely wrong. This experiment clearly illustrates the limitations of the latest deep-learning-based methods.

Table 11: Error cases on TPC-H (SyntaxSQLNet/GNN/IRNet).

Error cases		**# queries**
Not supported queries	Arithmetic operation	(GNN) 11 (IRNet, SyntaxSQLNet) 12
	CASE-WHEN-THEN statement	3
	More than 2 values in IN statement	3
	CREATE VIEW or Nesting in FROM clauses	2
	Function 'extract'	2
	SELECT count(*) as cnt ... ORDER BY cnt	2
	Correlated nested query	1
	EXISTS statement	1
	Self join	1
	More than four columns in SELECT clause	(IRNet) 2
Wrong translation		(GNN) 3 (SyntaxSQLNet) 2
Total		**22/22**

7. INSIGHTS AND QUESTIONS FOR FUTURE RESEARCH

Table 12 shows a summarized comparison of the eleven NL2SQL methods. There is no consistent winner over all benchmarks since all methods focus partially on limited-scope problems. Furthermore, the overall accuracy of each method is severely degraded when q_{nl} and/or q_{sql} become more complex and diverse (0% on TPC-H). As shown in Table 12, there are lots of challenging issues remaining in NL2SQL: handling unseen databases/q_{sql}'s, being robust on small training datasets, extending SQL construct coverage, and supporting various q_{nl} including multiple sentence questions.

Table 12: A comparison results summarized by whether each method is poor (▼), fair (-), or good (▲) for each dimension.

Ability	**NaLIR**	**Templar**	**NSP**	**Seq2SQL**	**SQLNet**	**TypeSQL**	**PT-MAML**	**Coarse2Fine**	**SyntaxSQLNet**	**GNN**	**IRNet**
Adaptability to new databases	-	▼	▼	-	-	-	-	-	-	▲	▲
Robustness with small datasets	■	-	▲	-	-	-	-	-	-	-	-
SQL construct coverage	-	-	-	▼	▼	▼	▼	▼	-	-	-
Generalizability to unseen q_{sql}	■	▼	▼	▼	▼	▼	▼	▼	▼	▼	▼
Supporting linguistic diversity	▼	▼	▼	▼	▼	▼	▼	▼	▼	▼	▼
Handling long, multiple sentences	▼	▼	▼	▼	▼	▼	▼	▼	▼	▼	▼

For cross-domain adaptability, schema entities must be treated as input rather than output vocabulary. On the other hand, in single-domain benchmarks, NSP outperforms the SaI methods in general. It would be an interesting research topic to develop a schema encoding technique that improves adaptability, while still enabling high accuracy in a single domain.

Aligning table/column references in q_{nl} to S_D can help improve accuracy. However, the existing techniques are still at a basic level and need to be significantly enhanced. We have shown that the alignment techniques of TypeSQL and GNN are effective only for some benchmarks. String matching rather than the matching based on word embedding vectors is often helpful especially for handling long and varied database entity names. Real-world databases may have a lot of rare words which cannot be found in the pretrained word embedding. In this case, IRNet, which is based on the string match, shows a distinct benefit, as seen in our experimental results on WTQ-s. However, its accuracy values for column prediction (i.e., acc$_{sel}$ and acc$_{wh,col}$) on WTQ-s are about 5-60%, which are still low. Those become even worse on complex queries (i.e., WTQ). If there is no phrase in q_{nl} that partially matches the corresponding schema entity name, as in question (C), IRNet fails to find the correct alignment. Developing alignment techniques is a remaining problem. Overcoming these limitations could be a promising future research directions.

Generating constant values in SQL queries is another challenging issue. Specifically, in the experiments on complex queries, all tested methods fail to correctly generate constant values in most cases or even have no functionality for generating constant values at all. Constant value anonymization of NSP does not work well when there are many constant values in V_D. This could also be ab important direction for future research.

8. CONCLUSION

In this paper, we present a comprehensive survey of existing NL2SQL methods and perform thorough performance evaluations. We introduce a taxonomy of NL2SQL methods and classify the latest methods. We fairly and empirically compare the state-of-the-art methods using many benchmarks. Specifically, we noticed the critical problem present in the previous experimental studies that use misleading measures. Thus, we accurately measured the quality of the NL2SQL methods by considering the semantic equivalence of SQL queries. We analyzed the experimental results in depth using various benchmarks including our one. From those results and analysis, we reported several important findings. We believe that this is the first work that thoroughly evaluates and analyzes the state-of-the-art NL2SQL methods on various benchmarks.

9. ACKNOWLEDGMENTS

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. NRF-2017R1A2B3007116) and Institute of Information communications Technology Planning Evaluation(IITP) grant funded by the Korea government(MSIT) (No. 2018-0-01398, Development of a Conversational, Self-tuning DBMS).

10. REFERENCES

[1] J. Abreu, L. Fred, D. Macêdo, and C. Zanchettin. Hierarchical attentional hybrid neural networks for document classification. In *ICANN*, pages 396–402, 2019.

[2] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. In *ICLR*, 2015.

[3] C. Baik, H. V. Jagadish, and Y. Li. Bridging the semantic gap with SQL query logs in natural language interfaces to databases. In *ICDE*, pages 374–385, 2019.

[4] F. Basik, B. Hättasch, A. Ilkhechi, A. Usta, S. Ramaswamy, P. Utama, N. Weir, C. Binnig, and U. Çetintemel. Dbpal: A learned nl-interface for databases. In *SIGMOD*, pages 1765–1768, 2018.

[5] Y. Bengio, P. Y. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Trans. Neural Networks*, 5(2):157–166, 1994.

[6] B. Bogin, J. Berant, and M. Gardner. Representing schema structure with graph neural networks for text-to-sql parsing. In *ACL*, pages 4560–4565, 2019.

[7] K. D. Bollacker, R. P. Cook, and P. Tufts. Freebase: A shared database of structured general human knowledge. In *AAAI*, pages 1962–1963, 2007.

[8] J. Castelein, M. F. Aniche, M. Soltani, A. Panichella, and A. van Deursen. Search-based test data generation for SQL queries. In *ICSE*, pages 1120–1230, 2018.

[9] J. Cheng, S. Reddy, V. A. Saraswat, and M. Lapata. Learning structured natural language representations for semantic parsing. In *ACL*, pages 44–55, 2017.

[10] S. Chu, B. Murphy, J. Roesch, A. Cheung, and D. Suciu. Axiomatic foundations and algorithms for deciding semantic equivalences of SQL queries. *PVLDB*, 11(11):1482–1495, 2018.

[11] S. Chu, C. Wang, K. Weitz, and A. Cheung. Cosette: An automated prover for SQL. In *CIDR*, 2017.

[12] D. A. Dahl, M. Bates, M. Brown, W. M. Fisher, K. Hunicke-Smith, D. S. Pallett, C. Pao, A. I. Rudnicky, and E. Shriberg. Expanding the scope of the ATIS task: The ATIS-3 corpus. In *ARPA Human Language Technology Workshop*, 1994.

[13] C. Finegan-Dollak, J. K. Kummerfeld, L. Zhang, K. Ramanathan, S. Sadasivam, R. Zhang, and D. R. Radev. Improving text-to-sql evaluation methodology. In *ACL*, pages 351–360, 2018.

[14] C. Finn, P. Abbeel, and S. Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *ICML*, pages 1126–1135, 2017.

[15] R. Frank. *Phrase structure composition and syntactic dependencies*, volume 38. Mit Press, 2004.

[16] J. Ganitkevitch, B. V. Durme, and C. Callison-Burch. PPDB: the paraphrase database. In *HLT-NAACL*, pages 758–764, 2013.

[17] A. Giordani and A. Moschitti. Translating questions to SQL queries with generative parsers discriminatively reranked. In *COLING*, pages 401–410, 2012.

[18] B. J. Grosz. TEAM: A transportable natural-language interface system. In *ANLP*, pages 39–45, 1983.

[19] Ç. Gülçehre, S. Ahn, R. Nallapati, B. Zhou, and Y. Bengio. Pointing the unknown words. In *ACL*, pages 140–149, 2016.

[20] J. Guo, Z. Zhan, Y. Gao, Y. Xiao, J. Lou, T. Liu, and D. Zhang. Towards complex text-to-sql in cross-domain database with intermediate representation. In *ACL*, pages 4524–4535, 2019.

[21] P. Huang, C. Wang, R. Singh, W. Yih, and X. He. Natural language to structured query generation via meta-learning. In *NAACL-HLT*, pages 732–738, 2018.

[22] S. Iyer, I. Konstas, A. Cheung, J. Krishnamurthy, and L. Zettlemoyer. Learning a neural semantic parser from user feedback. In *ACL*, pages 963–973, 2017.

[23] D. Jurafsky and J. H. Martin. *Speech and language processing: an introduction to natural language processing, computational linguistics, and speech recognition, 2nd Edition*. Prentice Hall series in artificial intelligence. Prentice Hall, Pearson Education International, 2009.

[24] M. Lapata and L. Dong. Coarse-to-fine decoding for neural semantic parsing. In *ACL*, pages 731–742, 2018.

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

[26] Y. Li, D. Tarlow, M. Brockschmidt, and R. S. Zemel. Gated graph sequence neural networks. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[27] Z. Lin, M. Feng, C. N. dos Santos, M. Yu, B. Xiang, B. Zhou, and Y. Bengio. A structured self-attentive sentence embedding. *CoRR*, abs/1703.03130, 2017.

[28] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *NIPS*, pages 3111–3119, 2013.

[29] G. A. Miller. Wordnet: A lexical database for english. *Commun. ACM*, 38(11):39–41, 1995.

[30] P. Pasupat and P. Liang. Compositional semantic parsing on semi-structured tables. In *ACL*, pages 1470–1480, 2015.

[31] A. Popescu, A. Armanasu, O. Etzioni, D. Ko, and A. Yates. Modern natural language interfaces to databases: Composing statistical parsing with semantic tractability. In *COLING*, 2004.

[32] A. Popescu, O. Etzioni, and H. A. Kautz. Towards a theory of natural language interfaces to databases. In *IUI*, pages 149–157, 2003.

[33] P. J. Price. Evaluation of spoken language systems: the ATIS domain. In *DARPA Speech and Natural Language Workshop*, pages 91–95, 1990.

[34] M. Rabinovich, M. Stern, and D. Klein. Abstract syntax networks for code generation and semantic parsing. In *ACL*, pages 1139–1149, 2017.

[35] D. Saha, A. Floratou, K. Sankaranarayanan, U. F. Minhas, A. R. Mittal, and F. Özcan. ATHENA: an ontology-driven system for natural language querying over relational data stores. *PVLDB*, 9(12):1209–1220, 2016.

[36] L. R. Tang and R. J. Mooney. Automated construction of database interfaces: Integrating statistical and relational learning for semantic parsing. In *EMNLP*, pages 133–141. Association for Computational Linguistics, 2000.

[37] O. Vinyals, M. Fortunato, and N. Jaitly. Pointer networks. In *NIPS*, pages 2692–2700, 2015.

[38] C. Wang, M. Brockschmidt, and R. Singh. Pointing out sql queries from text. *Technical Report MSR-TR-2017-45*, 2018.

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

[40] R. J. Williams and D. Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1(2):270–280, 1989.

[41] C. Xiao, M. Dymetman, and C. Gardent. Sequence-based structured prediction for semantic parsing. In *ACL*, 2016.

[42] X. Xu, C. Liu, and D. Song. Sqlnet: Generating structured queries from natural language without reinforcement learning. *CoRR*, abs/1711.04436, 2017.

[43] N. Yaghmazadeh, Y. Wang, I. Dillig, and T. Dillig. Sqlizer: query synthesis from natural language. *PACMPL*, 1(OOPSLA):63:1–63:26, 2017.

[44] S. Yavuz, I. Gur, Y. Su, and X. Yan. Dialsql: Dialogue based structured query generation. In *ACL*, pages 1339–1349, 2018.

[45] P. Yin and G. Neubig. A syntactic neural model for general-purpose code generation. In *ACL*, pages 440–450, 2017.

[46] T. Yu, Z. Li, Z. Zhang, R. Zhang, and D. R. Radev. Typesql: Knowledge-based type-aware neural text-to-sql generation. In *NAACL-HLT*, pages 588–594, 2018.

[47] T. Yu, M. Yasunaga, K. Yang, R. Zhang, D. Wang, Z. Li, and D. R. Radev. Syntaxsqlnet: Syntax tree networks for complex and cross-domain text-to-sql task. In *EMNLP*, pages 1653–1663, 2018.

[48] T. Yu, R. Zhang, K. Yang, M. Yasunaga, D. Wang, Z. Li, J. Ma, I. Li, Q. Yao, S. Roman, Z. Zhang, and D. R. Radev. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In *EMNLP*, pages 3911–3921, 2018.

[49] J. M. Zelle and R. J. Mooney. Learning to parse database queries using inductive logic programming. In *AAAI*, pages 1050–1055, 1996.

[50] L. S. Zettlemoyer and M. Collins. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *UAI*, pages 658–666, 2005.

[51] L. S. Zettlemoyer and M. Collins. Online learning of relaxed CCG grammars for parsing to logical form. In *EMNLP-CoNLL*, pages 678–687, 2007.

[52] V. Zhong, C. Xiong, and R. Socher. Seq2sql: Generating structured queries from natural language using reinforcement learning. *CoRR*, abs/1709.00103, 2017.

[53] M. Zhou, G. Cao, T. Liu, N. Duan, D. Tang, B. Qin, X. Feng, J. Ji, and Y. Sun. Semantic parsing with syntax- and table-aware SQL generation. In *ACL*, pages 361–372, 2018.