Natural SQL: Making SQL Easier to Infer from Natural Language Specifications

Yujian Gan[1]   **Xinyun Chen**[2]   **Jinxia Xie**[4]   **Matthew Purver**[1,3]
John R. Woodward[1]   **John Drake**[5]   **Qiaofu Zhang**[4]

[1]Queen Mary University of London   [2]UC Berkeley   [3]Jožef Stefan Institute
[4]Guangxi University of Finance and Economics   [5]University of Leicester
{y.gan,m.purver,j.woodward}@qmul.ac.uk
xinyun.chen@berkeley.edu   john.drake@leicester.ac.uk
{jinxia_xie,qiaofuzhang}@hotmail.com

Abstract

Addressing the mismatch between natural language descriptions and the corresponding SQL queries is a key challenge for text-to-SQL translation. To bridge this gap, we propose an SQL intermediate representation (IR) called Natural SQL (NatSQL). Specifically, NatSQL preserves the core functionalities of SQL, while it simplifies the queries as follows: (1) dispensing with operators and keywords such as *GROUP BY, HAVING, FROM, JOIN ON*, which are usually hard to find counterparts for in the text descriptions; (2) removing the need for nested subqueries and set operators; and (3) making schema linking easier by reducing the required number of schema items. On Spider, a challenging text-to-SQL benchmark that contains complex and nested SQL queries, we demonstrate that NatSQL outperforms other IRs, and significantly improves the performance of several previous SOTA models. Furthermore, for existing models that do not support executable SQL generation, NatSQL easily enables them to generate executable SQL queries, and achieves the new state-of-the-art execution accuracy [1].

1   Introduction

Automatic generation of SQL queries from natural language (NL) has been studied in the literature for a number of years (Warren and Pereira, 1982; Androutsopoulos et al., 1995; Ana-Maria Popescu et al., 2003; Li et al., 2006; Dong and Lapata, 2018; Li and Jagadish, 2014; Iacob et al., 2020). More recently, WikiSQL (Zhong et al., 2017), the first large-scale cross-domain text-to-SQL dataset, has attracted much attention from the research community (Xu et al., 2017; Wang et al., 2018; He et al., 2019). Although the current state-of-the-art approach has achieved over 90% execution accuracy on WikiSQL (He et al., 2019), since the SQL

queries in this benchmark only cover a single *SELECT* column and aggregation, as well as *WHERE* conditions, it does not represent the true complexity of SQL generation. To facilitate more realistic evaluation, Yu et al. (2018b) introduced Spider, the first large-scale cross-domain text-to-SQL benchmark with complex and nested SQL queries, on which previous models designed for WikiSQL suffer a significant performance drop.

To synthesize SQL queries with more complex structures, intermediate representation (IR) is widely employed by the previous SOTA models on the Spider dataset (Wang et al., 2020; Guo et al., 2019; Yu et al., 2018a; Shi et al., 2020). However, previous IRs are either too complicated or have limited coverage of SQL structures. Besides, although the existing IRs eliminate part of the mismatch between intent expressed in NL and the implementation details in SQL, there is still some mismatch that can be further eliminated by improving the IR.

In this work, we present *Natural SQL (NatSQL)*, a new intermediate representation that offers simplified queries over other IRs, while preserving a high coverage of SQL structures. More importantly, NatSQL further eliminates the mismatch between NL and SQL, and can easily support executable SQL generation. Figure 1 presents a sample comparison between NatSQL and other IRs. We observe that there is a mismatch between the NL word `and` and the *INTERSECT* SQL keyword, since in another similar question shown in Figure 5, the `and` no longer corresponds to the *INTERSECT* keyword. To translate the NL question into a corresponding query, previous IRs need the models to distinguish whether the word `and` corresponds to *INTERSECT*, this is not required for NatSQL. Among all IRs, NatSQL provides the simplest and shortest translation, while the NatSQL structure also aligns best with the NL question.

NatSQL preserves the core functionalities of SQL, while simplifying the queries as follows:

---

[1]Our code and dataset are available at https://github.com/ygan/NatSQL.

Figure 1: A sample question in Spider dataset with corresponding SQL and IRs.

(1) dispensing with operators and keywords such as *GROUP BY, HAVING, FROM, JOIN ON*, which are usually hard to find counterparts for in the text descriptions; (2) removing the need for nested sub-queries and set operators, using only one *SELECT* clause in NatSQL; and (3) making schema linking easier by reducing the required number of schema items that are normally not mentioned in the NL question. The design of NatSQL easily enables executable SQL generation, which is not naturally supported by other IRs.

We compare NatSQL with SQL and other IRs by incorporating them into existing open-source neural network models that achieve competitive performance on Spider. Our experiments show that NatSQL boosts the performance of these existing models, and outperforms both SQL and other IRs. In particular, equipping RAT-SQL+GAP with NatSQL achieves a new state-of-the-art execution accuracy on the Spider benchmark. These results suggest that to improve the ability of text-to-SQL models to understand and reason about the NL descriptions, designing IRs to better reveal the correspondence between natural language and query languages is a promising direction.

2 Review: Text-to-SQL Paradigm

Most existing text-to-SQL models generate the SQL keywords (blue character in Figure 1) and SQL schema items (black character in Figure 1)

separately. Based on this paradigm, we investigate how we can design an IR to improve both SQL keyword generation and schema item generation.

2.1 Generating SQL Keywords

Neural text-to-SQL models usually generate the SQL keywords according to the similarity linking scores between the hidden state from the question and the production rule embeddings. For example, in Figure 1, we conjecture a good text-to-SQL model should be able to give a higher linking score between the word '*less*' and the SQL '$<$' keyword.

However, SQL is designed for effectively querying relational databases, not for representing the meaning of NL questions. Hence, there inevitably exists a mismatch between intents expressed in natural language and the implementation details in SQL (Guo et al., 2019). For example, in Figure 1, the *GROUP BY* and *JOIN ON* clauses are not mentioned in the question. One solution is to use an IR to remove the SQL clauses that are hard to predict. Experiments show that the SemQL IR can improve the accuracy of previous models (Guo et al., 2019).

2.2 Generating Schema Items

Text-to-SQL models usually generate the schema items according to the similarity linking scores between tokens in the question and database schemas. Intuitively, a model is supposed to predict higher scores to schema items that are mentioned in the

NatSQL	=	*SELECT* , Column , { ',' Column } ,
		[*WHERE* W_Cond] ,
		[*ORDER BY* Order_By] ;
Column	=	Agg_Col | Table_Col ;
Agg_Col	=	Agg_Fun , '(' Table_Col ')' ;
Agg_Fun	=	'*avg*' | '*count*' | '*max*' | '*min*' | '*sum*' ;
Table_Col	=	TABLE_NAME , '.' , COLUMN_NAME
		| <u>TABLE_NAME , '.' , *</u> ;
W_Cond	=	[Conjunct], Condition , { Conjunct Condition } ;
Condition	=	Cond_L , W_Oper , Cond_R ,
		['and' , NUMBER] ;
Conjunct	=	'*and*' | '*or*' | '<u>*except*</u>' | '<u>*intersect*</u>'
		| '<u>*union*</u>' | '<u>*sub*</u>' ;
W_Oper	=	'*between*' | '=' | '>' | '<' | '>='
		| '<=' | '! =' | '*in*' | '*like*' | '*is*'
		| '*exists*' | '*not in*' | '*not like*'
		| '*not between*' | '*is not*' | '<u>*join*</u>' ;
Cond_R	=	NUMBER | STRING | Column ;
Cond_L	=	Column | "<u>@</u>" ;
Order_By	=	Column , [*DESC* | *ASC*] ,
		[*LIMIT* , NUMBER]

Table 1: The main grammar of NatSQL. Here we highlight the differences of production rules from SQL.

question. To achieve this goal, some existing neural networks implement a schema linking mechanism, by recognizing the tables and columns mentioned in a question (Guo et al., 2019; Bogin et al., 2019a; Wang et al., 2020).

Schema linking is essential for text-to-SQL tasks. As shown in the ablation study of IRNet (Guo et al., 2019) and RAT-SQL (Wang et al., 2020), removing the schema linking results in a dramatic decrease in performance. The importance of schema linking raises a question about generating schema items not mentioned in the question. Some models use graph neural networks to find these unmentioned schema items, and some models delete unmentioned schema items based on the IR; e.g., in Figure 1, the IRs remove the *JOIN ON* and *GROUP BY* clauses with the unmentioned schema items.

3 NatSQL

3.1 Overview

Table 1 presents the grammar specification of NatSQL. NatSQL only retains the *SELECT*, *WHERE* and *ORDER BY* clauses from SQL, dispensing with other clauses such as *GROUP BY, HAVING, FROM, JOIN ON*, set operators and subqueries. Tokens in capital italics are keywords of SQL and NatSQL, and other capital tokens represent special meanings, where 'TABLE_NAME' and 'COLUMN_NAME' are defined for databases, and 'NUMBER' and 'STRING' represent the data types.

Except for the deleted clauses, the differences between NatSQL and SQL are underlined in Table 1. NatSQL implements the function of the deleted clauses by adding new keywords and allowing *conjunct* to appear before the WHERE condition. In terms of language format, NatSQL does not add new clauses, and can retain deleted clauses as needed, as in the variant NatSQL$_G$ discussed in Section 3.3.

The main design principle of NatSQL is to simplify the structure of SQL and bring its grammar closer to natural language. Considering the example in Figure 1, the set operator '*INTERSECT*', used to combine *SELECT* statements, is never mentioned in the question. *INTERSECT* is introduced in SQL to allow the combination of the results of multiple functions. Such implementation details, however, are rarely considered by end users and therefore rarely mentioned in questions (Guo et al., 2019).

3.2 Overall Comparison

Starting from SyntaxSQLNet (Yu et al., 2018a), several types of IR have been developed for text-to-SQL models on the Spider dataset. The main limitation of SyntaxSQLNet is that it removes the *FROM* and *JOIN ON* clauses, which may result in the failure to find the correct table when converted to SQL. For example, in Figure 1, SyntaxSQLNet IR misses the *inventory* table, thus it cannot generate the correct *JOIN ON* clause that appears in the original SQL. The IR for RAT-SQL (Wang et al., 2020) is mostly close to SQL, and it avoids missing tables since it only removes the *JOIN ON* clause from SQL. Zhong et al. (2020) and Lee (2019) also utilize an IR that is similar to the IR in RAT-SQL and SyntaxSQLNet.

Guo et al. (2019) introduced SemQL, an intermediate language, to facilitate SQL prediction. As with NatSQL, SemQL removes the keywords *FROM, JOIN ON, GROUP BY, HAVING* from SQL. Although SemQL and NatSQL remove both *FROM* and *JOIN ON* clauses, SemQL and NatSQL avoid missing a table by moving the table into the '*' column. NatSQL improves on SemQL in the following ways:

(1) Compatible with a wider range of SQL queries than SemQL.

(2) Simplify the structure of queries with set operators, i.e., *INTERSECT*, *UNION*, and *EXCEPT*, denoted as *IUE* hereafter.

(3) Eliminate nested subqueries.

Ques 1:	Find ... who have a pet.
NatSQL:	... **WHERE** @ *join* has_pet.*
Ques 2:	Find ... who have two pet.
NatSQL:	... **WHERE** *count*(has_pet.*) = 2

Table 2: A modified example based on Figure 2

Figure 2: An example about the scalability and readability of NatSQL.

(4) Reduce the number of schema items to predict. (5) NatSQL uses the same keywords and syntax as SQL, which makes it easier to read and expand than SemQL.

There are four examples in Figure 1, 2, 3 and 4 demonstrating the differences between SQL, SemQL, and NatSQL statements representing the same natural language question.

3.3 Scalability of NatSQL

We take an SQL query with multiple tables as an example. In Figure 2, since the SemQL misses the *has_pet* table, SemQL cannot be converted to the target SQL, indicating that SemQL is not compatible with this type of SQL query. The SyntaxSQL-Net IR is also not compatible, but the RAT-SQL IR can convert this query appropriately.

While both SemQL and NatSQL completely remove all *FROM* and *JOIN ON* clauses, NatSQL introduces a new *WHERE* condition operator *join* for these unremovable *JOIN ON* clauses, as shown in Figure 2. With this extra *WHERE* condition, NatSQL can be converted to the target SQL. Alternatively, you could use the NatSQL augmented with *FROM* clause version. We recommend the original version since its experimental result is better and the sub-question 'who have a pet' looks like a *WHERE* condition. We modify this example in Table 2 to illustrate why it looks like a *WHERE* condition. Usually, NatSQL does not need the *join* operator for generating *JOIN ON* clause, such as the '*Ques 2*' in Table 2, except in cases when it cannot infer the correct *JOIN ON* clause from other clauses.

NatSQL$_G$. Since each database has different compatibility with SQL, we allow NatSQL to retain the deleted clauses as needed. NatSQL$_G$ is NatSQL augmented with *GROUP BY*, which improves the compatibility in the SQLite database where the Spider benchmark is built on.

3.4 NatSQL for SQL Keyword Generation

By simplifying the set operators and nested sub-queries, NatSQL improves text-to-SQL models.

3.4.1 Simplifying Queries with Set Operators

It is typically hard to generate queries with IUE (*INTERSECT, UNION*, and *EXCEPT*) set operators for text-to-SQL models, where the corresponding F1 score is usually the lowest among all breakdown metrics on the Spider benchmark (Guo et al., 2019; Bogin et al., 2019a; Wang et al., 2020). The main reason is that the related questions are generally longer and more complicated, while the mismatch between NL and SQL queries further increases the prediction difficulty, as discussed in Section 2.1.

Figure 3 compares the SQL queries corresponding to two similar problems. The second question in Figure 3 contains an extra condition: 'more than 1 room'. This extra condition changes the structure of the entire SQL query. Although IRs have been widely used for complex SQL, enthusiasts of end-to-end models expect the text-to-SQL model to automatically distinguish whether the word token 'or' in Figure 3 corresponds to *UNION* or *OR* keyword. However, most models cannot do that and would generate a *OR* clause for both questions. This example is similar to the comparison between Figure 1 and Figure 5 discussed on Section 1.

NatSQL bridges this gap by unifying them into a simple *OR* operator that will be converted to a *UNION* clause when it cannot concatenate its following conditions. The reasons for the failure to concatenate conditions include: (1) the precedence of the following conditions is higher (e.g., the precedence of *AND* is higher than *OR*); (2) the two conditions cannot be connected, or they are disjoint such as the example in Figure 1. The '*count(film_actor.*)>5*' condition cannot be con-

Question:
Find names of properties that are houses **or** apartments?

SQL: (Almost the same as Other IRs)
`SELECT name FROM Properties WHERE code = "House" OR code = "Apartment"`

NatSQL:
`SELECT name FROM Properties WHERE code = "House" OR code = "Apartment"`

Question:
Find names of properties that are houses **or** apartments with more than 1 room?

SQL: (Almost the same as Other IRs)
`SELECT name FROM prop WHERE code = "House" UNION SELECT name FROM prop WHERE code = "Apartment" AND room > 1`

NatSQL:
`SELECT prop.name WHERE prop.code = "House" OR prop.code = "Apartment" AND prop.room > 1`

Figure 3: An example about the mismatch between NL and IUE set operators.

Question:
Find the number of visitors who did not visit any museum opened after 2010.

SQL:
`SELECT count(*) FROM visitor WHERE id NOT IN ( SELECT t2.visitor_id FROM museum AS t1 JOIN visit AS t2 ON t1.Museum_ID = t2.Museum_ID WHERE t1.open_year > 2010 )`

SemQL:
`SELECT count(visitor.*) WHERE visitor.id NOT IN ( SELECT visit.visitor_id WHERE museum.open_year > 2010 )`

It is hard to construct schema linking for column 'id', because the question doesn't mention it:

NatSQL: @ is a placeholder
`SELECT count(visitor.*) WHERE @ NOT IN visit.* and museum.open_year > 2010`

Figure 4: A sample question in Spider dataset with corresponding SQL, SemQL and NatSQL queries.

nected with the '*count(inventory.*)<3*' condition because they belong to different tables. Based on the same rules, NatSQL simplifies the SQL with other set operators, the details of which can be found in Appendix A.

3.4.2 Eliminating Nested Subqueries

Since the subqueries in both NatSQL and SemQL only appear in *WHERE* conditions, only one column in the *SELECT* clause of a subquery is required. NatSQL keeps this *SELECT* column in '*Cond_R*' (right column of *WHERE* conditions) instead of a whole *SELECT* clause. Since this meets the *WHERE* condition format, NatSQL can remove the brackets and subqueries from SQL, as shown in Figure 4.

3.5 How NatSQL Help Schema Item Generation

NatSQL helps schema item generation by reducing the number of schema items that need to be predicted. For example, in Figure 4, without an in-depth analysis of the database schema, by looking at the natural language description itself, it is difficult to infer the grey shaded columns in SQL and SemQL (in this example, they are column '*id*' in table '*visitor*' and column '*visitor_id*' in table '*visit*'). We cannot build a schema linking for these

columns, even though the schema linking is important to boost performance as discussed in Section 2.2.

NatSQL solves this problem by replacing some of the columns with a table only or @, where @ is a place holder of NatSQL. We can find that all columns of NatSQL in Figure 4 are mentioned in the question. Specifically, NatSQL uses @ to replace the '*visitor.id*' and uses '*visit.*' to replace '*visit.visitor_id*'.

@ is a placeholder in NatSQL that only appears in '*Cond_L*', which denotes that we need to infer a column to replace it. The '*' keyword does not appear in the *WHERE* condition without an aggregation function, so NatSQL uses it to represent a table. With this table, we can infer the correct column in the target SQL to replace the @ and '*table.*' according to Algorithm 1.

3.6 Executable SQL Generation

Many previous text-to-SQL models (Guo et al., 2019; Wang et al., 2020; Bogin et al., 2019a) only focus on the Spider exact match accuracy, i.e., they only generate the SQL queries without condition values. These queries are not executable until filling in the condition values. However, it is not easy to fill in the values correctly. On the one hand, there are too many possible condition value slots

---

Algorithm 1 Infer columns to replace the @ and table.* in NatSQL

Input: t_list ▷ All tables before @, which include the table 'visitor' in Figure 4

         $table\_r$ ▷ The table next to the @, which is the table 'visit' in Figure 4

Output: Two columns to replace the @ and table.*

1: **for** Every $table$ in t_list **do**
2:      **if** There is foreign key relationship between $table$ and $table\_r$ **then**
3:          **return** These two foreign key columns
4: **for** Every $table$ in t_list **do**
5:      **if** There are columns with the same name in both $table$ and $table\_r$ **then**
6:          **return** The same name columns
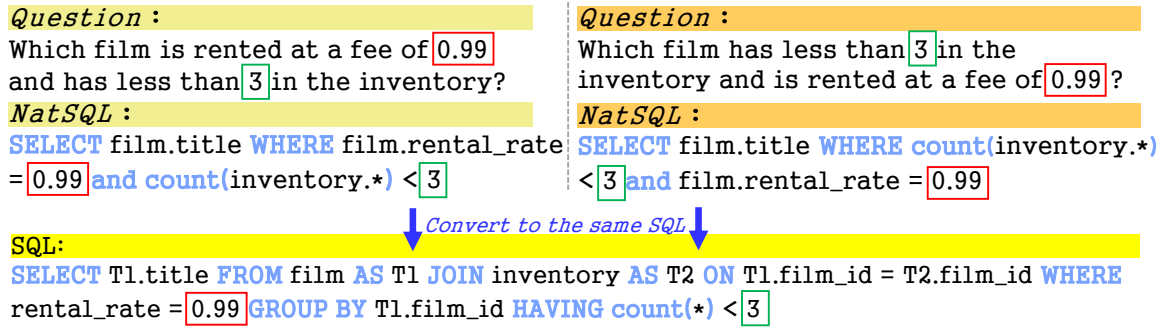7: **return** Their primary keys

---

Figure 5: Fill the values in order of appearance (see more discussion in Appendix B).

that need to be searched. The slots can appear in: *WHERE* clause, *WHERE* clause in a subquery, *WHERE* clause after set operators, *HAVING* clause, etc. On the other hand, when there are multiple value slots, it is easier to confuse where to fill. For example, in Figure 5, the two different questions correspond to the same SQL query, making it hard to copy the right values from the question to SQL.

Because the condition value slots of NatSQL only appear in the *WHERE* clause, generating condition values becomes much easier, as shown in Figure 5. Unlike the models (Lin et al., 2020; Rubin and Berant, 2021) trained to copy the values from questions to SQL queries, NatSQL simply copies the possible values (numbers or database cell values) from questions to SQL in the order of appearance without training. This feature enables the models designed only for the Spider exact match metrics to generate executable SQL.

4 Experiments

4.1 Experimental Setup

We evaluate NatSQL on the Spider benchmark (Yu et al., 2018b). There are 7000, 1034 and 2147 samples for training, development and testing respectively, where 206 databases are split into 146

for training, 20 for development and 40 for testing.

We first evaluate the gold NatSQL and other IRs using the exact match and execution match metrics in (Yu et al., 2018b). Exact match measures whether the predicted query without condition values as a whole is equivalent to the gold query. Execution match measures whether the execution result of the predicted query from the database is the same as the gold query. We then evaluate NatSQL and other IRs using existing open-source models that provide competitive performance on Spider: (1) GNN (Bogin et al., 2019a); (2) IRNet (Guo et al., 2019); (3) RAT-SQL (Wang et al., 2020); (4) RAT-SQL+GAP (Shi et al., 2020). Although some of these models are not designed for the generation of executable SQL queries, with the approach discussed in Section 3.6, we utilize NatSQL to generate executable SQL and evaluate the execution match performance.

4.2 Comparison Between IRs

4.2.1 Gold IRs

In Table 3, we present the exact match and execution match accuracies of the gold IRs on the Spider development set, where the metrics are defined by Yu et al. (2018b) for the Spider benchmark.

We observe that NatSQL can be converted to

Language	Exact Match	Execution Match
SQL	100%	100%
SemQL	86.2%	Unsupported
IR(RAT-SQL)	97.7%	97.1%
NatSQL	93.3%	95.3%
NatSQL$_G$	96.2%	96.5%

Table 3: The comparison between gold IRs on Spider development set.

Ques:	Find students whose age is 10 or 16.
SQL 1:	... **WHERE** age = 10 **or** age = 16
NatSQL 1:	... **WHERE** age = 10 **or** age = 16
SQL 2:	... **WHERE** age = 10 **UNION** ... **WHERE** age = 16
NatSQL 2:	... **WHERE** age = 10 **union** age = 16

Table 4: Equivalent SQL queries with its NatSQL

Approach	Exact	Execution
GNN + SQL	47.5%	
GNN + SemQL	51.6%	
GNN + NatSQL	**53.8%**	**58.0%**
IRNet + SemQL	51.8%	
IRNet + NatSQL	**52.9%**	**52.6%**
RAT-SQL + IR(RAT-SQL)	62.7%	
RAT-SQL + SemQL	58.4%	
RAT-SQL + NatSQL	64.4%	66.7%
RAT-SQL + NatSQL$_G$	**65.2%**	**67.3%**
extend BERT:		
RAT-SQL + IR(RAT-SQL)	69.5%	
RAT-SQL + NatSQL	71.7%	72.8%
RAT-SQL + NatSQL$_G$	**72.1%**	**73.0%**
extend GAP:		
RAT-SQL + IR(RAT-SQL)	71.8%	
RAT-SQL + NatSQL	**73.7%**	74.6%
RAT-SQL + NatSQL$_G$	**73.7%**	**75.0%**

Table 5: Exact and execution match accuracy on Spider development set.

more gold SQL than SemQL, because NatSQL can handle the unremovable *JOIN ON* clauses, as discussed in Section 3.3. Such SQL queries comprise around 5% of the entire Spider dataset. Other performance improvement comes from the fact that NatSQL is more compatible with subqueries and that its capability to generate SQL is better. More importantly, SemQL is designed only for the exact match metrics of Spider, and cannot directly be used to generate executable SQL.

The IR of RAT-SQL is the most similar to SQL and thus has the highest coverage among all IRs. However, NatSQL$_G$ further simplifies the queries with only 0.6% execution accuracy degradation, whilst enabling better model prediction performance. NatSQL$_G$ outperforms NatSQL when comparing the gold queries, but the gap is small when they are utilized by models. We defer more breakdown analysis to Appendix C.

The result in the training set is close to that in the development set. It should be noted that the exact match accuracy will slightly vary in different NatSQL versions. The accuracy depends on the attitude towards equivalent SQL queries. Table 4 presents two equivalent SQL queries with their corresponding NatSQL queries. Considering that *UNION* is not mentioned in the question, we prefer to sacrifice the exact match accuracy for a more succinct NatSQL representation, i.e., we will use the first NatSQL query in Table 4 to represent the second SQL, even though it can not be converted into the second SQL query. Although our preference slightly affects the exact match accuracy

in the Spider benchmark, it brings greater potential and convenience when outside Spider.

4.2.2 IRs for Prediction

Table 5 presents the exact match accuracy of four models with SemQL, its default IR (or SQL), and NatSQL separately. We observe that NatSQL consistently outperforms SemQL with all of these model architectures, including IRNet. Note that the original Spider dataset additionally includes 1,659 training samples from 6 earlier text-to-SQL benchmarks (Academic, GeoQuery, IMDB, Restaurants, Scholar and Yelp), which were used to train models with SemQL in the IRNet. To provide a fair comparison with other baselines, we didn't include these additional samples for all models in our evaluation, thus our presented result for IRNet+SemQL (51.8%) is lower than the number reported in the IRNet paper (53.2%).

Note that SemQL causes performance decline for RAT-SQL. We hypothesize that this is because the exact match accuracy of the gold SemQL is only 86.2%. With the improvement of model architectures, such a gap will affect the prediction accuracy more negatively. Although the accuracy of gold RAT-SQL IR is higher than that of NatSQL, NatSQL still outperforms the original RAT-SQL model, and NatSQL$_G$ slightly improves the performance over NatSQL.

Meanwhile, NatSQL helps these models generate executable SQL queries. Execution match accuracy improves with the improvement of the

Approach	Easy	Medium	Hard	Extra
GNN + SemQL	68.5%	**58.9%**	36.8%	24.1%
GNN + NatSQL	**72.0%**	58.0%	**42.0%**	**28.2%**
IRNet + SemQL	69.8%	53.0%	46.0%	30.1%
IRNet + NatSQL	**70.6%**	**54.1%**	46.0%	**32.5%**
RAT-SQL + IR(RAT-SQL)	80.4%	63.9%	55.7%	40.6%
RAT-SQL + NatSQL$_G$	**82.4%**	**65.0%**	**59.2%**	**46.5%**
extend BERT:				
RAT-SQL + IR(RAT-SQL)	86.4%	73.6%	62.1%	42.9%
RAT-SQL + NatSQL$_G$	**88.4%**	**76.6%**	**62.6%**	**46.4%**
extend GAP:				
RAT-SQL + IR(RAT-SQL)	88.3%	74.0%	64.4%	44.0%
RAT-SQL + NatSQL$_G$	**91.6%**	**75.2%**	**65.5%**	**51.8%**

Table 6: Exact match accuracy by difficulty on Spider development set.

Approach	Exact	Execution
IRNet + BERT (Guo et al., 2019)	54.7%	–
RATSQL + BERT (Wang et al., 2020)	65.6%	–
BRIDGE v2 + BERT(ensemble) (Lin et al., 2020)	67.5%	68.3%
COMBINE (Anonymous)	67.7%	68.2%
SmBoP + GraPPa (Rubin and Berant, 2021)	69.5%	71.1%
RATSQL + GAP (Shi et al., 2020)	69.7%	–
DT-Fixup SQL-SP + RoBERTa (Anonymous)	**70.9%**	–
RAT-SQL + GAP + NatSQL$_G$ (Ours)	68.7%	**73.3%**

Table 7: Results on Spider test set, compared to other models at the top of the leaderboard.

exact match, and most execution match accuracy is better than that of exact match. The execution match accuracy of IRNet is slightly lower than the exact match, because the IRNet does not predict the *DISTINCT* keyword while the exact match metric does not check this aspect.

Breakdown results. Based on the complexity of the SQL, the examples in Spider are classified into four types: `easy`, `medium`, `hard`, and `extra hard`. We provide a breakdown comparison on the Spider development set, as shown in Table 6. The improvement brought by Nat-SQL mainly comes from the `extra hard` SQL, which demonstrate an average 4.74% absolute improvement across these models. This improvement is in line with the design of NatSQL, i.e., most `extra hard` SQL queries contain set operators or subqueries, while NatSQL has simplified these components. Since `easy` and `medium` SQL queries categorized in the Spider dataset are more similar to NatSQL queries, it is expected that the improvement on simple SQL is less significant. However, we still observe that NatSQL consistently increases the accuracy on most samples of different difficulty levels.

4.3 Overall Performance Analysis

First, we present the exact and execution match accuracy of our approach applied to RAT-SQL augmented with GAP in Table 7, where we compare with various baselines at the top of the Spider leaderboard. By incorporating NatSQL into the RAT-SQL model with GAP, we demonstrate that our approach achieves a new state-of-the-art on Spider execution benchmark, surpassing its best counterparts by 2.2% absolute improvement.

Considering that the gap between dev and test in exact match is larger than that in execution match,

we speculate that there are two reasons why our exact match accuracy has dropped by 1% compared to RAT-SQL+GAP. From the complexity breakdown accuracy between dev and test, we observe that the main performance degradation comes from the `extra hard` SQL queries. Since there are many subqueries in `extra hard` SQL queries, some limitations of the Spider exact match evaluation process (discussed in Appendix C) may have a negative effect on our prediction results. On the other hand, some degradation may come from equivalent SQL queries. As we discuss in Section 4.2.1 and Table 4, it is not mandatory to keep the NatSQL queries consistent with the original SQL queries. As a result, the model trained by NatSQL may output equivalent SQL queries that do not match exactly but that get the same query result. Therefore, our evaluation shows that NatSQL is more suitable for generating executable SQL queries.

5 Related Work

Natural Language Interface to Database The study of Natural Language Interface to Database (NLIDB) has a long history that can be traced back to the 1970s (Warren and Pereira, 1982; Androut-sopoulos et al., 1995; Popescu et al., 2004; Li et al., 2006; Iacob et al., 2020). Most of the early work focuses on single-domain datasets, including ATIS, GeoQuery (Iyer et al., 2017), Restaurants (Ana-Maria Popescu et al., 2003; Tang and Mooney, 2000; Giordani and Moschitti, 2012), Scholar (Iyer et al., 2017), Academic (Li and Jagadish, 2014), Yelp and IMDB (Yaghmazadeh et al., 2017) and so on. Finegan-Dollak et al. (2018) shows some models dealing with specific databases that only learn to match semantic parsing results. It is a challenge to generate SQL queries in a cross-domain setting, such as the case of the WikiSQL (Zhong et al., 2017) and Spider (Yu et al., 2018b) benchmarks. SyntaxSQLNet (Yu et al., 2018a) was the first study to use the Spider benchmark. Following

this work, many models are presented to address this problem (Bogin et al., 2019a; Guo et al., 2019; Zhang et al., 2019; Bogin et al., 2019b; Wang et al., 2020; Rubin and Berant, 2021; Lin et al., 2020).

Intermediate Representations for NLIDB Early work on IR of SQL tried to use an IR to translate a natural language question and then convert it to SQL queries (Woods, 1978; Li and Jagadish, 2014). Li et al. (2014) proposed an IR for SQL called Schema-free SQL, for users who do not need to know all of the schema information. The IR in SyntaxSQLNet (Yu et al., 2018a) represents an SQL statement without *FROM* and *JOIN ON* clauses. SemQL (Guo et al., 2019) removes the *FROM, JOIN ON* and *GROUP BY* clauses, and combines the *WHERE* and *HAVING* conditions. The IR in EditSQL (Zhang et al., 2019) also combines the *WHERE* and *HAVING* conditions but keeps the *GROUP BY* clause. IR is also used to improve compositional generalization in semantic parsing (Herzig et al., 2021). Compared to existing IRs for SQL, our NatSQL further simplifies the SQL language, moving closer towards bridging the gap between natural language descriptions and SQL statements.

6   Conclusion

In this paper, we propose NatSQL, a new SQL intermediate representation that reduces the difficulty of schema linking and simplifies the SQL structure. By incorporating NatSQL into existing neural models for text-to-SQL generation, we show that NatSQL is easier to infer from natural language specification than the full-fledged SQL and other intermediate representation languages. Furthermore, NatSQL enables existing models to easily generate executable SQL queries without modifying their architecture. Experimental results on the challenging Spider benchmark demonstrate that NatSQL consistently improves the prediction performance of several neural network architectures and achieves the state-of-the-art, showing the effectiveness of our approach.

Acknowledgements

References

Ana-Maria Popescu, Oren Etzioni, and Henry Kautz. 2003. Towards a Theory of Natural Language Interfaces to Databases. In *Proceedings of the 8th International Conference on Intelligent User Interfaces*, pages 149–157.

I Androutsopoulos, G D Ritchie, and P Thanisch. 1995. Natural language interfaces to databases – an introduction. *Natural Language Engineering*, 1(1):29–81.

Ben Bogin, Jonathan Berant, and Matt Gardner. 2019a. 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. Association for Computational Linguistics.

Ben Bogin, Matt Gardner, and Jonathan Berant. 2019b. Global Reasoning over Database Structures for Text-to-SQL Parsing. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3659–3664, Hong Kong, China. Association for Computational Linguistics.

Li Dong and Mirella Lapata. 2018. Coarse-to-Fine Decoding for Neural Semantic Parsing. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 731–742, Stroudsburg, PA, USA. Association for Computational Linguistics.

Catherine Finegan-Dollak, Jonathan K. Kummerfeld, Li Zhang, Karthik Ramanathan, Sesh Sadasivam, Rui Zhang, and Dragomir Radev. 2018. Improving text-to-SQL evaluation methodology. pages 351–360.

Alessandra Giordani and Alessandro Moschitti. 2012. Automatic Generation and Reranking of SQL-derived Answers to NL Questions. In *Proceedings of the Second International Conference on Trustworthy Eternal Systems via Evolving Software, Data and Knowledge*, pages 59–76.

Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. 2019. Towards Complex Text-to-SQL in Cross-Domain

Database with Intermediate Representation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4524–4535, Florence, Italy. Association for Computational Linguistics.

Pengcheng He, Yi Mao, Kaushik Chakrabarti, and Weizhu Chen. 2019. X-SQL: REINFORCE CONTEXT INTO SCHEMA REPRESENTATION. *https://www.microsoft.com/en-us/research/uploads/prod/2019/03/X_SQL-5c7db555d760f.pdf*.

Jonathan Herzig, Peter Shaw, Ming-Wei Chang, Kelvin Guu, Panupong Pasupat, and Yuan Zhang. 2021. Unlocking compositional generalization in pre-trained models using intermediate representations. *CoRR*, abs/2104.07478.

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

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy, and Luke Zettlemoyer. 2017. Learning a Neural Semantic Parser from User Feedback. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 963–973.

Dongjun Lee. 2019. Clause-Wise and Recursive Decoding for Complex and Cross-Domain Text-to-SQL Generation. 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 6045–6051, Hong Kong, China. Association for Computational Linguistics.

Fei Li and H. V. Jagadish. 2014. Constructing an interactive natural language interface for relational databases. *Proceedings of the VLDB Endowment*, 8(1):73–84.

Fei Li, Tianyin Pan, and Hosagrahar V. Jagadish. 2014. Schema-free SQL. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data - SIGMOD '14*, pages 1051–1062, New York, New York, USA. ACM Press.

Yunyao Li, Huahai Yang, and H V Jagadish. 2006. Constructing a Generic Natural Language Interface for an XML Database. In *Advances in Database Technology - EDBT 2006*, pages 737–754, Berlin, Heidelberg. Springer Berlin Heidelberg.

Xi Victoria Lin, Richard Socher, and Caiming Xiong. 2020. Bridging Textual and Tabular Data for Cross-Domain Text-to-SQL Semantic Parsing. In *Find-ings of the Association for Computational Linguistics: EMNLP 2020*, pages 4870–4888, Online. Association for Computational Linguistics.

Ana-Maria Popescu, Alex Armanasu, Oren Etzioni, David Ko, and Alexander Yates. 2004. Modern natural language interfaces to databases: composing statistical parsing with semantic tractability. In *Proceedings of the 20th international conference on Computational Linguistics - COLING '04*, pages 141–es, Morristown, NJ, USA. Association for Computational Linguistics.

Ohad Rubin and Jonathan Berant. 2021. SmBoP: Semi-autoregressive bottom-up semantic parsing. pages 311–324.

Peng Shi, Patrick Ng, Zhiguo Wang, Henghui Zhu, Alexander Hanbo Li, Jun Wang, Cícero Nogueira dos Santos, and Bing Xiang. 2020. Learning contextual representations for semantic parsing with generation-augmented pre-training. *CoRR*, abs/2012.10309.

Lappoon R Tang and Raymond J Mooney. 2000. Automated Construction of Database Interfaces: Intergrating Statistical and Relational Learning for Semantic Parsing. In *2000 Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora*, pages 133–141.

Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2020. RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7567–7578, Online. Association for Computational Linguistics.

Chenglong Wang, Kedar Tatwawadi, Marc Brockschmidt, Po-Sen Huang, Yi Mao, Oleksandr Polozov, and Rishabh Singh. 2018. Robust Text-to-SQL Generation with Execution-Guided Decoding.

David H D Warren and Fernando C N Pereira. 1982. An Efficient Easily Adaptable System for Interpreting Natural Language Queries. *Comput. Linguist.*, 8(3-4):110–122.

W.A. Woods. 1978. Semantics and Quantification in Natural Language Question Answering. *Advances in Computers*, 17:1–87.

Xiaojun Xu, Chang Liu, and Dawn Song. 2017. SQL-Net: Generating Structured Queries From Natural Language Without Reinforcement Learning. *Mathematics of Computation*, 22(103):651.

Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. SQLizer: Query Synthesis from Natural Language. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM*, pages 63:1—63:26.

Tao Yu, Michihiro Yasunaga, Kai Yang, Rui Zhang, Dongxu Wang, Zifan Li, and Dragomir Radev. 2018a. SyntaxSQLNet: Syntax tree networks for complex and cross-domain text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1653–1663, Brussels, Belgium. Association for Computational Linguistics.

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. 2018b. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921, Brussels, Belgium. Association for Computational Linguistics.

Rui Zhang, Tao Yu, Heyang Er, Sungrok Shim, Eric Xue, Xi Victoria Lin, Tianze Shi, Caiming Xiong, Richard Socher, and Dragomir Radev. 2019. Editing-based SQL query generation for cross-domain context-dependent questions. pages 5338–5349.

Victor Zhong, Mike Lewis, Sida I Wang, and Luke Zettlemoyer. 2020. Grounded adaptation for zero-shot executable semantic parsing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6869–6882.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning. *CoRR*, abs/1709.0.

A Further Discussion on Set Operators

Based on the rules discussed on Section 3.4.1, NatSQL can simplify the SQL with *INTERSECT* (example is shown in Figure 1) and *EXCEPT*. As to the case that the set operator itself represents part of a condition, NatSQL allows them to follow the *WHERE* keyword. As illustrated in Table 8, this type of SQL is mainly related to the *EXCEPT* operator.

The NatSQL prediction work in Table 8 is easier than others. NatSQL here only needs to predict the '*cartoon*' table, instead of predicting the '*cartoon.channel*' column. Predicting a table is easier than predicting a column because the premise of finding the correct column is to find the correct table. Besides, many models incorrectly output '*cartoon.id*' instead of '*cartoon.channel*' because the annotation of '*cartoon.id*' is the same as '*tv_channel.id*' column.

Ques	Find the id of tv channels that do not play any cartoon
SQL	**SELECT** id **FROM** tv_channel **EXCEPT** **SELECT** channel **FROM** cartoon
SemQL	**SELECT** tv_channel.id **EXCEPT** **SELECT** cartoon.channel
NatSQL	**SELECT** tv_channel.id **WHERE except** cartoon.*

Table 8: An example of none *WHERE* conditions before the IUE.

In addition to the conditions mentioned in Section 3.4.1 that cannot be concatenated, Table 9 present one more example. These two conditions can not concatenate because one *WHERE* condition can not concatenate a *HAVING* condition by a *OR* operator.

Ques	Which film is rented at a fee of 0.99 **or** has less than 3 in the inventory?
SemQL	**SELECT** film.title **WHERE** film.rental_rate = 0.99 **UNION** **SELECT** film.title **WHERE** count(inventory.*)< 3
NatSQL	**SELECT** film.title **WHERE** film.rental_rate = 0.99 **OR** count(inventory.*)< 3

Table 9: An example modified from that in Figure 5.

B Further Discussion on Executable SQL Generation

In Section 3.6, we discuss that different questions in Figure 5 will be converted to different NatSQL, where training data is the key. Firstly, in the dataset, for SQL with multiple *WHERE* conditions, the order of the conditions is mostly consistent with the question. Secondly, the NatSQL further expands this type of training data. For example, the NatSQL queries in Figure 1,3,4 contain more *WHERE* conditions than SQL and other IRs, and these conditions appear in the order they are mentioned. These training data make it possible for models to generate different NatSQL according to the different questions in Figure 5.

C Gold NatSQL Error Analysis

Table 10 presents the F1 score of NatSQL for different SQL components. We observe that the main errors come from *GROUP BY* and IUE matching. Although NatSQL cannot be converted to all gold *GROUP BY* clauses, most of these errors don't affect the execution results. The IUE errors occur because NatSQL only supports one IUE operator per query.

Some other errors are due to the limitation of the exact match evaluation method when

Component	F1	Component	F1
select	0.997	where	0.969
group	0.879	order	0.996
and/or	0.998	IUE	0.900
keywords	0.989		

Table 10: Partial matching F1 score of NatSQL on the Spider development set.

evaluating the *JOIN ON* clause of subqueries and sub-subqueries. Specifically, when the *FROM* and *JOIN* in a generated subquery is not identical to the gold SQL, the Spider evaluation scheme considers it to be wrong. For example, the following two SQL statements have the same semantic meaning, but they are recognized as different by the Spider exact match evaluation method, thus results in an exact match error.

... col in (**SELECT** col **FROM** T1 **JOIN** T2 ...)
... col in (**SELECT** col **FROM** T2 **JOIN** T1 ...)

D   SQL, SemQL and NatSQL Examples

We present more examples in Table 11.

Ques:	What are the name of the countries where there is not a single car maker?
SQL:	**SELECT** CountryName **FROM** countries **EXCEPT SELECT** T1.CountryName **FROM** countries AS T1 **JOIN** car_makers AS T2 **ON** T1.countryId = T2.Country;
SemQL:	Not Support
NatSQL:	**SELECT** countries.countryname **WHERE except** @ is car_makers.*

Ques:	Find the last name of the students who currently live in the state of North Carolina but have not registered in any degree program.
SQL:	**SELECT** T1.staff_name **FROM** staff AS T1 **JOIN** Staff_DA AS T2 **ON** T1.staff_id = T2.staff_id **WHERE** T2.job_title_code = "Sales Person" **EXCEPT** **SELECT** T1.staff_name **FROM** staff AS T1 **JOIN** Staff_DA AS T2 **ON** T1.staff_id = T2.staff_id **WHERE** T2.job_title_code = "Clerical Staff"
SemQL:	**SELECT** staff.staff_name **WHERE** Staff_DA.job_title_code = "Sales Person" **EXCEPT** **SELECT** staff.staff_name **WHERE** Staff_DA.job_title_code = "Clerical Staff"
NatSQL:	**SELECT** staff.staff_name **WHERE** Staff_DA.job_title_code = "Sales Person" **AND** Staff_DA.job_title_code != "Clerical Staff"

Ques:	Find id of the tv channels that from the countries where have more than two tv channels.
SQL:	**SELECT** id **FROM** tv_channel **GROUP BY** country **HAVING** count(*) > 2
SemQL:	**SELECT** tv_channel.id **WHERE** count (tv_channel.*) > 2
NatSQL:	**SELECT** tv_channel.id **WHERE** count (tv_channel.*) > 2

Ques:	List all song names by singers above the average age.
SQL:	**SELECT** song_name **FROM** singer **WHERE** age > (**SELECT** avg(age) **FROM** singer)
SemQL:	**SELECT** singer.song_name **WHERE** singer.age > (**SELECT** avg(singer.age))
NatSQL:	**SELECT** singer.song_name singer **WHERE** @ > avg (age)

Ques:	Which district has both stores with less than 3000 products and stores with more than 10000 products?
SQL:	**SELECT** district **FROM** shop **WHERE** Number_products < 3000 **INTERSECT SELECT** district **FROM** shop **WHERE** Number_products > 10000
SemQL:	**SELECT** shop.district **WHERE** shop.Number_products < 3000 **INTERSECT SELECT** shop.district **WHERE** shop.Number_products > 10000
NatSQL:	**SELECT** shop.district **WHERE** shop.number_products < 3000 **and** shop.number_products > 10000

Table 11: SQL, SemQL and NatSQL examples from the Spider.