2023 IEEE 39th International Conference on Data Engineering (ICDE) | 979-8-3503-2227-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICDE55515.2023.00016

GAR: A Generate-and-Rank Approach for Natural Language to SQL Translation

Yuankai Fan, Zhenying He, Tonghui Ren, Dianjun Guo, Lin Chen, Ruisi Zhu, Guanduo Chen
Yinan Jing, Kai Zhang, X.Sean Wang
School of Computer Science, *Fudan University*, Shanghai, China
{ykfan19, zhenying, thren20, djguo20, lin_chen20, rszhu20, gdchen18, jingyn, zhangk, xywangCS}@fudan.edu.cn

***Abstract*—A Natural Language (NL) Interface to Databases (NLIDB) aims to help end-users access databases. State-of-the-art approaches primarily construct language translation models to convert NL queries to SQL queries. While these models exhibit good performance on NLIDB benchmarks, the translation accuracy seems to have stalled at between 70%-75%, and most erroneous translations happen with complex queries that require an understanding of the structure and semantics specific to a database. This paper proposes a Generate-And-Rank approach called GAR. GAR assumes that a set of sample SQL queries is given to represent the possible user-intended queries to the database. In order to provide a broad coverage, akin to avoiding over-fitting, GAR extracts the basic components from the sample set to form the basic building blocks to generate a set of generalized SQL queries. By leveraging a simple rule-based SQL to NL technique, a less natural NL expression called a dialect expression for each sample and generalized SQL query is obtained. Finally, a learning-to-rank method is used for a given NL query to retrieve the best dialect expression and hence the resulting SQL query. Extensive experiments are performed to study GAR in comparison with other approaches. The results show that GAR achieves better performance on the NLIDB benchmarks, including in particular a 78.5% translation accuracy on the popular SPIDER benchmark, outperforming the best reported accuracy in the literature. An extension to GAR, called GAR-J, is further introduced to aid the translation by annotating join semantics in the sample queries. The experimental results show that GAR-J can further improve translation accuracy on queries with joins. Code for GAR can be found at https://github.com/Kaimary/GAR.**

***Index Terms*—NLIDB, NL2SQL, SQL, learning-to-rank**

I. INTRODUCTION

Designing user-friendly query interfaces for databases is becoming an increasingly important goal [1], [2]. Recently, due to the maturity of language translation techniques, some interfaces have taken the form of translating natural language (NL) queries to SQL queries using machine learning methods [3]–[11]. The main idea is to consider the NL interface to databases (NLIDB) problem as a language translation task and train a generalized sequence-to-sequence (Seq2Seq) model.

Despite the significant gains in terms of translation accuracy (defined as syntactic equivalence), the overall improvement, however, seems to have stalled. Indeed, the translation accuracy results reported in the literature on the popular SPIDER [12] benchmark are mostly below 75%, and the model [13] on top of the SPIDER leaderboard[1] achieves only 72.1% translation accuracy on the test set at the time of writing.

Translation difficulties mostly arise in complex queries. For example, take the two state-of-the-art models, GAP [9] and SMBOP [8]. The translation accuracy breakdown on the SPIDER benchmark, in terms of SQL "difficulty levels"[2], is as follows:

TABLE 1: Translation accuracy on SPIDER by SQL difficulty levels

Model	Easy	Medium	Hard	Extra Hard	Overall
GAP	0.915	0.742	0.644	0.494	0.727
SMBOP	0.890	0.791	0.644	0.470	0.737

As can be observed, the performance of these translation methods drops when queries become harder. The reason we believe is that the harder queries may require significantly more training data on the target database than usually (and practically) given in the benchmarks, and the training data on other databases are of little help. Indeed, *every database has its own peculiar structure and semantics, as if forming a "universe" of its own with its own way of expressing ideas*.

Consider the example in Fig. 1 that shows a "Hard" NL-SQL queries pair and the translation results of the above two models. Both models correctly use the join path `employee-evaluation`, but fail to infer the "highest one time bonus" with the right SQL clauses. GAP mistakenly infers the employee with "the most bonuses" semantics by counting the number of the records for each employee, while SMBOP incorrectly gives the "total bonus" of each employee. This example shows that the translation may fail if a given database's peculiar semantics and structure are not sufficiently considered.

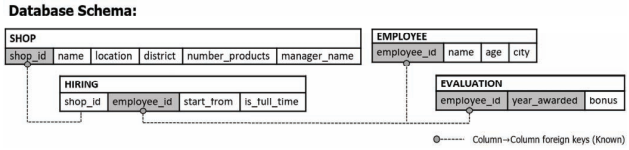
To add more semantics into the translation, an ontology-based approach [14] is reported to achieve some improvements, but the approach is rule-based that requires the user-given NL queries be annotated to map to the ontology elements before translation. Another recent work [15] stipulates that providing a small number of in-domain training examples may be a more promising approach.

In this paper, we propose an approach called GAR. *The basic assumption is that a set of sample queries on a database is given, and we can use SQL queries that are "component-similar" to the given samples to answer NL queries.* Here, by "component-similar" we mean the queries that are built

[1]https://yale-lily.github.io/spider. Note that the leaderboard uses unknown testing queries and databases to evaluate submitted NLIDB algorithms.

[2]SPIDER defines the SQL difficulty based on the number of SQL clauses, so that queries that contain more SQL keywords are considered to be harder.

2375-026X/23/$31.00 ©2023 IEEE
DOI 10.1109/ICDE55515.2023.00016



(a) A database schema in the SPIDER benchmark

NL Query:	*Find the name of the employee who got the highest one time bonus.*
SQL Query: (Gold)	`SELECT T1.name FROM employee AS T1 JOIN evaluation AS T2 ON T1.employee_id=T2.employee_id ORDER BY T2.bonus DESC LIMIT 1`
	GAP model [9]
Generated SQL: (Incorrect)	`SELECT T1.name FROM employee AS T1 JOIN evaluation AS T2 ON T1.employee_id=T2.employee_id GROUP BY T2.employee_id ORDER BY COUNT(*) DESC LIMIT 1`
	SMBOP model [8]
Generated SQL: (Incorrect)	`SELECT T1.name FROM employee AS T1 JOIN evaluation AS T2 ON T1.employee_id=T2.employee_id GROUP BY T2.employee_id ORDER BY SUM(T2.bonus) DESC LIMIT 1`

(b) A pair of NL-SQL queries and the corresponding translation results of two NLIDB models

Fig. 1: An example from the SPIDER benchmark

from some variations of the SQL components appeared in the sample set. The more query varieties we have in the sample set, the more capable our translation will be. (More details can be found in Section III-A.)

With the above assumption, the specific database semantics and structure that the users are interested in are made more explicit in terms of the components in the sample SQL queries. In this manner, GAR can handle much more complex queries that the users may ask, as the space for SQL clause combinations is significantly reduced. In the example of Fig. 1, the given "gold" SQL query as a sample query indicates that the semantics of combining the ordering operation with column `bonus` is what's interesting to the users. From the sample, GAR generates component-similar queries to gain more data for learning. (Detailed explanation of why GAR works correctly with the example in Fig. 1 will be given in late sections when the steps of GAR are presented.) In a way, GAR works in the same spirit of GAN [16] that strives to compensate for the lack of training data.

Briefly, GAR starts with a set of sample SQL queries on a given database and works as follows. We first generalize the sample set in an attempt to capture all the component-similar SQL queries. We then translate the sample and generalized SQL queries to NL expressions (SQL2NL). These NL expressions are mostly correct but less natural, hence we call them "dialect" expressions. Note that these dialect expressions are formed from the SQL queries on the given database schema, so the schema information is considered in GAR. The above process of generating the dialect expressions is called the data preparation process. For a given NL query on the given database, GAR looks into the set of dialect expressions generated in the data preparation process and employs a learning-to-rank (LTR) model to find the closest dialect expression and hence the SQL translation result.

To evaluate the effectiveness of GAR, we conduct our experiments on three public benchmarks, namely GEO [17], SPIDER and MT-TEQL [18]. GAR attains 65.2% overall accuracy on GEO[3], 78.5% overall accuracy on SPIDER[4], outperforming the best reported accuracy of 76.4% on the validation set[5] and 78.4% accuracy on MT-TEQL.

Although improving the state-of-the-art, GAR may still mistranslate queries, especially those involving join operations, since the join operations often raise the level of abstraction in semantics that is difficult for GAR to infer from the table/column names alone. Therefore, we extend GAR to GAR-J by adding annotations to the join operations in the SQL to NL step to help the translation. With GAR-J, the translation accuracies improve to 67.8% and 78.9% on the GEO and SPIDER benchmarks, respectively. In addition, we develop a new benchmark (QBEN) that emulates a collection of manually-curated NL-SQL query pairs in which the join operations have semantic meanings that are more than simple compositions of table/column names. GAR-J achieves 70.0% accuracy on QBEN, while other NLIDB algorithms, including GAR, can only achieve about 20-40% accuracy.

To summarize, our contributions is four-fold:

- We propose a novel GAR approach based on a set of sample SQL queries to tackle the NL to SQL (NL2SQL) translation problem. With such samples and the SQL2NL technique, the peculiar semantics of databases may be appropriately captured in the generated dialect expressions.
- We formulate the NL2SQL task as a semantic matching problem, and utilize the LTR techniques in information retrieval to achieve the NL query to dialect expression, hence to SQL query translation.
- We extend GAR to GAR-J to handle queries involving joins with the help of annotation. We propose a new benchmark QBEN that explicitly tests the robustness of NLIDB systems for queries with joins.
- We perform a series of experiments to evaluate GAR and GAR-J. The experiments show the better performance of GAR and GAR-J over the existing ones. In addition, with the experiments, we explore and gain more understanding of the various aspects of GAR.

The remainder of this paper is organized as follows. First, we give an overview of GAR in Section II. We then detail the methodologies introduced in GAR in Section III. Next, we discuss GAR-J in Section IV. We report the results of our experiments in Section V. Finally, we discuss the related works in Section VI and conclude in Section VII.

[3]Earlier rule-based methods may achieve better results on GEO but are explicitly tuned, so we do not compare them in this paper.

[4]We assessed the validation set, as we cannot submit GAR to the scoreboard for evaluation on the test set due to lacking sample SQL queries.

[5]The validation set is the test data. It will not be used in training phase.

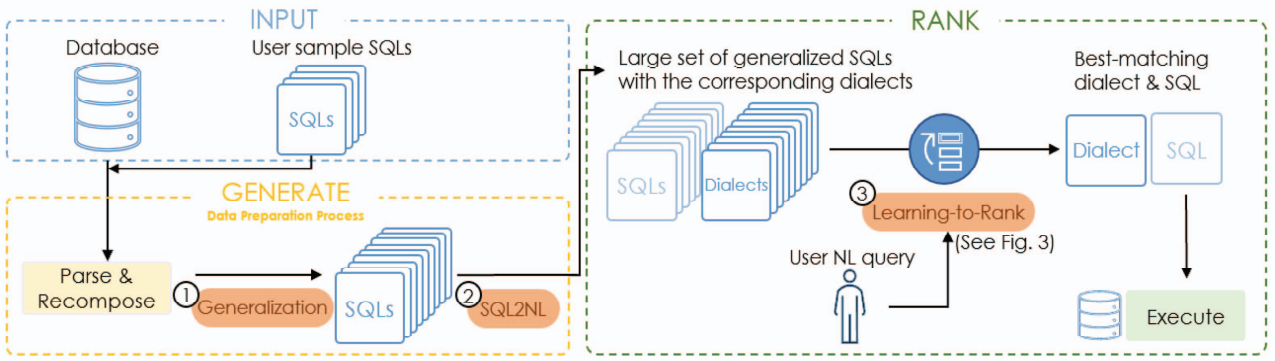


Fig. 2: Overview of GAR

II. GAR IN BRIEF

In this section, we first provide a short explanation about the usage of sample queries, as GAR relies on them to capture user interests. Then, we present an overview of GAR.

A. Sample Queries

As mentioned earlier, in order to provide correct NL2SQL translation when training data is lacking, especially when dealing with difficult queries with multiple SQL clauses, we start with a set of sample queries to generate more data. That is, GAR requires sample SQL queries to represent how users query the database (and reflect how the particular table organization is used in queries), and learns from them as a starting point. The goal is to correctly translate queries that are component-similar (defined after) to the given samples.

The requirement of sample queries on a database may be hinted by some recent studies [19]–[21] that realized several generalization challenges existing under a zero-shot cross-domain setting (i.e., apply the trained model to an unseen database) and proposed to perform few-shot learning for the NL2SQL problem [15]. For example, the study in [21] shows that the generalization performance of existing models has a notable drop when domain knowledge is required for unseen domains. Therefore, we believe that learning from sample queries on a target database is a correct approach, since each database is almost like a new "domain".

B. Overview

A high-level view of GAR can be seen in Fig. 2. Given a set of sample SQL queries, GAR first uses the following data preparation process to generate dialect expressions during an offline initialization phase: generalization and SQL2NL. After the data preparation process, an LTR model is used to rank the generated dialect expressions for the final translation. In a way, this generate-and-rank method used in GAR is similar to the one applied in Alphacode [22] to generate competition-level code. We describe each step below.

Generalization. The step in Fig. 2-① uses a set of generalization rules to generalize the sample queries to provide a good coverage for component-similar queries (defined after) while limiting the resulting set to a manageable size. For example, assume that the "gold" SQL query in Fig. 1b is given as a sample query of the database. GAR should allow users to ask an NL query like *"Find the age of the employee who got the highest one time bonus."*, which is expressed by a SQL query that is component-similar to the "gold" SQL query.

To generalize the sample SQL queries, GAR first converts each sample query to the corresponding parse tree, and then implements a generalization process by recomposing, in a controlled manner, the sub-trees in the set of parse trees to synthesize a large set of component-similar SQL queries.

SQL2NL. The step in Fig. 2-② uses a template-based SQL2NL method [23]. Each clause of a SQL query is mapped to an NL phrase mechanically with the help of a parsed query tree. Then the phrases are combined into a sentence, which we call a dialect expression. For example, the following is the dialect expression for the gold SQL query in Fig. 1b: *"Find the name of employee regarding to evaluation with employee. Return the top one result in descending order of one bonus of the employee evaluation."*. Note that here we are able to generate the phrase "one bonus" (instead of "all bonus" or "total bonus") since `employee_id` is not the unique key of the `evaluation` table.

LTR. The LTR model in Fig. 2-③ follows the LTR approach [24] in information retrieval that trains a neural network in a supervised manner. In GAR, the LTR model learns to rank the semantic similarities between NL queries and dialect expressions (explained below). The model is used to rank the dialect expressions and then find the best-matching one for a given NL query, leading to the translation result. For example, for the given NL query *"Find the name of the employee who got the highest one time bonus."*, the model will output the dialect expression example in the previous paragraph as the highest ranked dialect expression result, and hence the "gold" SQL query in Fig. 1b can be found as the translation result.

Fig. 3 shows the overall training phase of the LTR model implemented in GAR. The training data of LTR model is composed of a set of triples $\{(q_i, d_i, s_i) | q_i \in Q, d_i \in D, 0 \leq s_i \leq 1\}_{i=1}^{N}$, where q_i represents an NL query, d_i denotes a dialect expression, and s_i represents the semantic similarity score between d_i and q_i, such that the more similar d_i and q_i are, the more the score s_i tends to 1. In this paper, given an

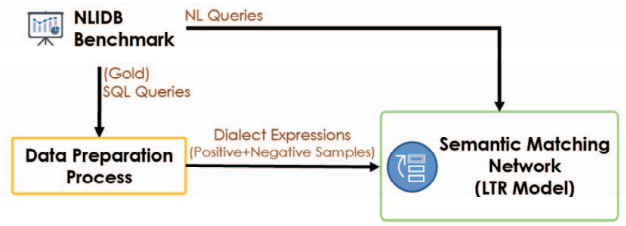


Fig. 3: Training phase of the LTR model

NLIDB benchmark, we use all the NL queries in the benchmark as Q, and employ the corresponding "gold" SQL queries as the sample queries to generate the set of SQL queries and their dialect expressions via the data preparation process in the steps of Fig. 2-① and ②. The semantic similarity scores are calculated as follows: If d_i is exactly generated from the "gold" SQL query of q_i, then s_i is set to 1. Such a triple is called a positive sample. Otherwise, s_i is calculated by comparing each clause between (1) the corresponding SQL query that generates d_i in the data preparation process and (2) the given "gold" SQL query for q_i in the benchmark. Such a triple is called a negative sample. The exact calculation of s_i is given in the next section.

III. GAR

In this section, we describe in detail the GAR data preparation process, the LTR model and its training method.

A. Compositional-based SQL Generalizer

The main observation of the generalization step is that SQL is compositional in a context-free manner, which indicates that each SQL query is formed by components that can be recomposed to construct new SQL queries. We employ this idea to introduce component-level generalization to generate component-similar queries.

We first give the concept of the component, which is similar to that of a query fragment introduced in [2].

Definition 1. *A* SQL component *takes the form of one of the seven types given in Table 2.*

TABLE 2: Query component types and examples

Type	Component Example	NL Semantics
select	`SELECT employee.name`	Find the employee name
from	`FROM employee`	Employee
where	`WHERE employee.name="John"`	The employee named John
group	`GROUP BY employee.id`	For each employee ID
order	`ORDER BY evaluation.bonus DESC LIMIT 1`	The highest one time bonus
join	`FROM employee JOIN evaluation ON id=employee_id`	The employee with evaluation
compound	`INTERSECT id FROM employee WHERE name="John"`	(Find the ID of) the employee named John

As can be seen in the Table 2, some component types are aligned with the corresponding SQL clauses, and others are some combinations of them.

Definition 2. *Given a set of sample queries, the* component-level generalization *is a process that generates SQL queries by recomposing the components that are in the given sample queries. The queries generated by this process are said to be* component similar *to the sample queries.*

Note that since we assume all the query components come from the given sample queries, GAR in the current setting may fail on some "simple" cases where the SQL query includes one or more simple but unseen query components. For example, if the sample queries only have `GROUP BY employee.id` but not the `GROUP BY employee.name` component, GAR is not able to generate the SQL queries that include the latter component. It will be an interesting future work direction to see how such a limitation may be resolved, e.g., by examining the database schema to obtain more basic components.

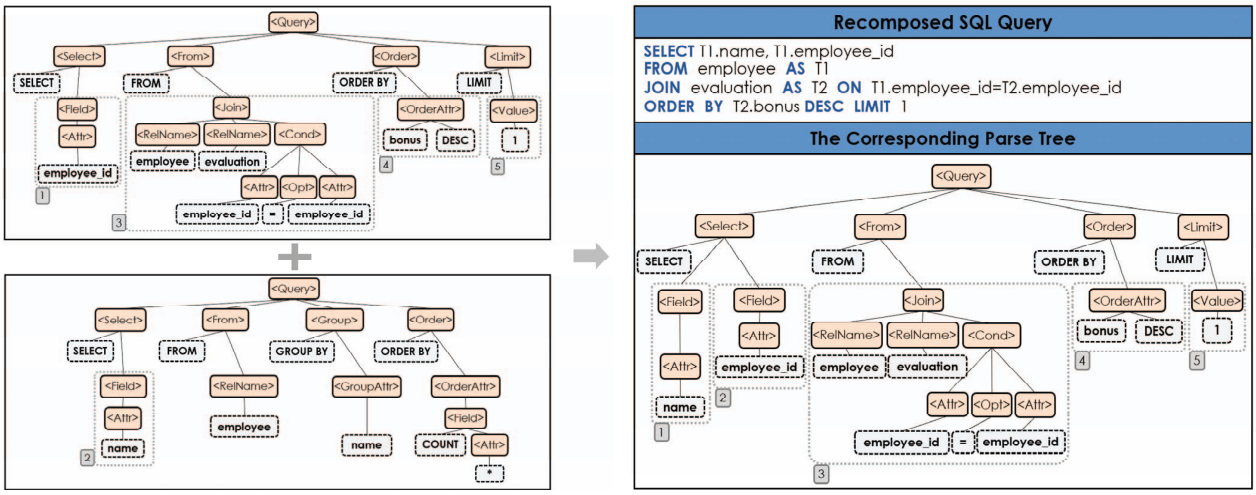
Based on the above definition, we use the parse tree to represent the compositional characteristics of a SQL query. A parse tree is an abstract syntax tree that represents the syntactic structures of a SQL query according to the grammar [25]. For example, Fig. 4a shows the two parse trees, where each parse tree is constructed by a set of sub-trees (i.e., components); each sub-tree is based on the production of the terminals (dotted-line nodes) and non-terminals (solid-line nodes). The set of sub-trees can be recomposed into a new parse tree, as shown in Fig. 4b, and hence to a new SQL query. The new query may further be fed to a SQL compiler to ensure its correctness.

Since the generalization process should not depend on the specific literal values (e.g., a string value specified in a predicate in a `WHERE` clause), we mask out literal values with placeholders when converting the SQL queries to the corresponding parse trees. In this way, we preserve the semantic structure of the SQL queries irrespective of the exact values used in the queries. This step makes the generalization process succinctly handle queries with different literal values. A SQL query that GAR handles represents the query with all possible instantiation of its specific values.

1) Component-level Generalization: Given a set of sample SQL queries, the generalization process is **recursive**: We first randomly select two parse trees from the given query parse trees. Secondly, we randomly choose a non-terminal node type and select two sub-trees rooted with this node type from the two chosen parse trees, respectively, and then recompose the two parse trees by shuffling the two sub-trees. Thirdly, we perform the syntactic and semantic checks of the newly recomposed parse trees to ensure their correctness. Finally, we put those valid recomposed parse trees back into the original set and repeat the above steps until no more new parse tree is generated. This process may generate an extensive set and may not even stop in its general form. Therefore, we add some composition rules (see below) to force the stopping condition to satisfy and to make the resulting set manageable. We present the compositional generalization process in Algorithm 1.

Recomposition Rules We observe that many generalized SQL queries are syntactically correct but semantically not component-similar to the sample set. Therefore, to make the above generalization proceyouss tractable, we introduce the following recomposition rules to prune out unnecessary compositions during the generalization process.

- *Rule 1 (Join Rule).* As we discuss in Section IV, the join operations may raise abstraction levels in semantics that the SQL queries express. Since we only answer those NL



(a) The parse trees of two SQL queries

(b) A recomposed example

Fig. 4: Two parse trees and an example of a recomposed SQL query result

Algorithm 1: Compositional Generalization Algorithm

Inputs : Given a set of parse trees T
Output: A new set of parse trees T
Procedure GENERALIZE-QUERIES (T) **:**
  if no new parse tree generated in the previous iteration **then**
    return T
  $t1, t2 \leftarrow$ random select two parse trees from the set T
  $nt \leftarrow$ random select an nonterminal from t1 and t2
  $st_1, st_2 \leftarrow$ FIND-SUBTREES ($nt, t1, t2$)
  // Shuffle the substrees st_1 and st_2 in $t1$ and $t2$ to form two new trees
  $t1_{new}, t2_{new} \leftarrow$ RECOMPOSE-TREES ($st_1, st_2, t1, t2$)
  // If $t1_{new}$ ($t2_{new}$) does not satisfy the recomposition rules (see the below section) or is not syntactically valid, do not add into T
  if VALIDATE-TREE ($t1_{new}$) **then**
    $T = T \cup t1_{new}$
  if VALIDATE-TREE ($t2_{new}$) **then**
    $T = T \cup t2_{new}$
  // Recursive call
  GENERALIZE-QUERIES (T)

queries if they can be expressed by the SQL queries that have the user interested semantics in the given sample queries, we require the join operations existing in the generalized SQL queries to be consistent with those in sample queries. In other words, the recomposed parse trees of those SQL queries derived from different join paths with those used in sample queries are excluded from the generalized set. In the example of Fig. 4a, the "join" type sub-tree of the top parse tree represents the join operation of the corresponding SQL query. Suppose the sub-tree is recomposed with a new "join" type sub-tree whose terminal node is the `shop` table, and this new join operation does not appear in the sample queries. In that case, the recomposed parse tree is excluded.

Note that the join operation can appear explicitly with the "join" keyword but may also be expressed with the `WHERE` clause or correlated subquery. The generalization process makes the best effort to apply this rule.

- *Rule 2 (Syntactic Restriction).* Since SQL allows unlimited nesting and other ways to form a very large number of SQL queries, we define a set of constraints to limit the syntactic complexity of each SQL clause of the generalized SQL queries. For example, we define a constraint for the `WHERE` SQL clause that specifies the maximum number of predicates that generalized SQL queries can have. All the constraints are collected from the given sample queries, which indicates that the complexity of generalized SQL queries should be similar to the one in the sample queries.
- *Rule 3 (Frequency Preservation).* Intuitively, the generalized SQL queries should reflect the user preference if the user is more interested in particular semantics in the given database. Hence, the generalization process generates more parse trees with sub-trees that occur more frequently in the parse tree set for the sample SQL queries.
- *Rule 4 (Sub-query Preservation).* We observe that in most cases, a subquery only appears as a whole in various queries. Therefore, we treat a subquery as a whole when performing the recomposition. That is, we do not change any child nodes of a subquery when making the generalization.

In this paper, we use the above four rules. It is an intended future research problem to study if there are other rules to use in generating component-similar queries, and to study how this component-similar should be formally defined and enforced.

B. Template-assisted Dialect Builder

An essential question for GAR is how to translate a SQL query to a dialect expression that is as correct, natural, and

meaningful as possible. The dialect builder of GAR is based on the SQL2NL techniques that appeared in the literature [23], [26]. Since the method in [26] currently is limited to simple SQL queries, we follow the method introduced in [23] to construct our dialect builder.

In this section, we provide a short description of the approach in [23] (we call the method GRAPH-NL in the rest of the paper) and then explain the improvements needed for the purpose of GAR. We will discuss further enhancements introduced in GAR-J for handling join queries in Section IV.

GRAPH-NL first treats a SQL query as a string and splits the string into chunks representing each clause (much like a sub-tree in a parse tree) in the query and then builds up a graph for the SQL query clause by clause. A query graph $G_q(V_q, E_q)$ is a directed graph, where each node in V_q denotes a query element in the SQL query, such as the column `name` in the `SELECT` clause; Each edge in E_q is associated with a type, which denotes a specific relationship between two query elements. For example, as for the SQL query in Fig. 4b, the `employee` table node has a "select" type edge that connects with the `name` column node and a "join" type edge that connects with the `evaluation` table node. Next, GRAPH-NL assigns each graph element, either a node or an edge, a label[6] to represent its semantics. Finally, GRAPH-NL traverses the query graph according to the different traversal algorithms defined in the paper and then concatenates the element-based labels found on the way to generate the NL expression by using some descriptive expressions (e.g., *'Find'*, *'for'*, etc.).

GAR follows the methods introduced in GRAPH-NL but made some changes: (1) Instead of using query graphs, GAR directly uses parse trees to represent the SQL queries. Similarly, GAR first assigns the "labels" for the nodes in a parse tree and then traverses the parse tree in a pre-order fashion to generate NL expression; (2) To support subqueries, GAR treats each subquery as a whole (in the same way as the recomposition process we described earlier in Section III-A), and uses a specific node type to represent; (3) By leveraging the database schema information, GAR adds more semantics specific to the database along with the generation of dialect expressions. For example, for the "one bonus" (or the "total bonus") semantics of column `bonus` in Fig. 1b, GAR first checks the database key information of the underlying table `evaluation`. Since the `evaluation` table has compound keys (i.e., `employee_id` and `year_awarded`), GAR can recognize the "one bonus" semantics of the column `bonus`.

Fig. 5 shows the dialect generation process for the top parse tree in Fig. 4a, where each of the NL phrases highlighted below is generated from the corresponding sub-tree of the parse tree, and the dialect expression is then constructed by concatenating those NL phrases.

[6]In our experiments, since the SPIDER benchmark provides the annotations of column names and table names of its databases, we use those annotations as node labels and provide the column and table annotations for GEO and QBEN benchmarks in the same way. As for edge labels, we follow GRAPH-NL to use default labels.

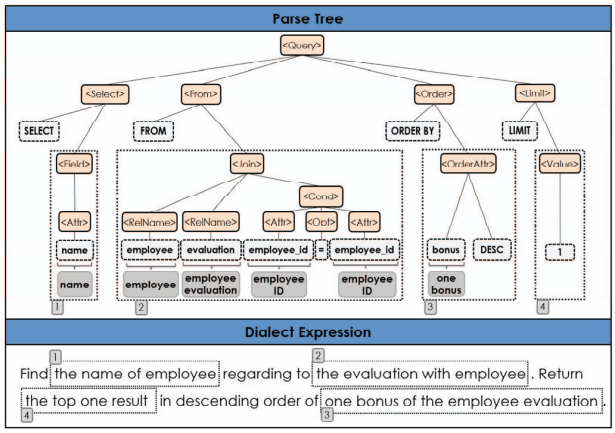
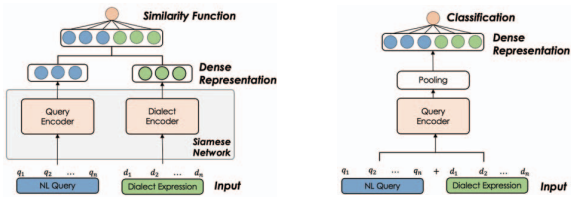


Fig. 5: An example of dialect expression generation

C. Neural Semantic Matching Network

Following the earlier works [27], [28] in the information retrieval field, GAR employs the LTR technique to implement a two-stage ranking pipeline with two separate machine learning models to construct the semantic matching network. In the first stage, a coarse-grained "retrieval model" is used to narrow the relatively large set to a relatively small collection of potential best-matching dialect expressions. Then a second-stage fine-grained "re-ranking model" is used on the small set from the first stage to get the final top-ranked dialect expressions.

Fig. 6 presents the overall network architectures of the two models. More specifically, the network for the retrieval model on the left side is based on the Siamese BERT-network introduced in [29]. This retrieval model modifies the pre-trained BERT network [30] using Siamese and triplet network structures [31] to derive sentence embeddings. On the other hand, the network of the re-ranking model on the right side uses the sentence pair classification architecture derived from BERT architecture to predict the possibility of relevance (binary classification) between the input NL-dialect query pairs.



(a) Retrieval model network (b) Re-ranking model network

Fig. 6: The architectures of the two ranking models

1) Retrieval Model: To select a subset of the dialect expressions from an extensive set, an efficient retrieval model is helpful in quickly inferring the semantic similarity of each dialect expression with the given NL query. The semantic similarities inferred by the retrieval model may not be accurate, but is suitable for the first-stage subset retrieval.

Formally, given an NL query q and a set of dialect expressions $D = \{d_1, d_2, \cdots, d_n\}$, the inference objective of the retrieval model is to produce the relevant scores $S = \{s_1, s_2, \cdots, s_n\}$ for the corresponding dialect expression $d_i \in D$ with respect to q. We can then use a threshold k to select the first k highly ranked results as the subset of D.

Training Data The training data of the retrieval model is a set of triples $\{(q_i, d_i, s_i)\}_{i=1}^{N}$, where q_i is an NL query, d_i is a query dialect and s_i is the semantic similarity score between d_i and q_i. We calculate the score s_i as follows. First, s_i is set to 1 initially, and then we compare each clause of the SQL query that is used to obtain the dialect d_i with the "gold" query that is given for q_i. If a clause is not the same, we give a punishment on the s_i value. Finally, the calculation process ends until we have compared all the clauses or the s_i value drops to 0.

2) Re-ranking Model: Note that the retrieval model provides the re-ranking model with a much less but more relevant subset of dialect expressions. Hence, the objective of the second-stage re-ranking model is to accurately rank the subset based on the semantics similarities with the given NL query and then to find the top-ranked dialect expression. Since more relevant dialect expressions with the given NL query often share "confounding characteristics", the re-ranking model should be able to recognize them from the set.

Similar with the retrieval model, given an NL query q and a set of dialect expressions $D = \{d_1, d_2, \cdots, d_n\}$, the inference objective of the re-ranking model is to get the ranked dialect expression set $D^{'} = \{d_1^{'}, d_2^{'}, \cdots, d_n^{'}\}$ with their corresponding relevant scores $S^{'} = \{s_1, s_2, \cdots, s_n\}$.

Training Data The training data of the re-ranking model is defined as a set of triples $\{(q_i, d_i, l_i)\}_{i=1}^{N}$, where q_i is an NL query, d_i is a dialect expression and l_i is the corresponding binary scores that indicate if d_i is generated by the "gold" SQL query of the NL query q_i or not.

As we use the listwise approach [24] to train the re-ranking model, we further group the training triples by each NL query q_i. Therefore, we finally obtain a set of triples $\{(q_j, D_j, L_j)\}_{j=1}^{M}$, where q_j is an NL query, $D_j = \{d_{j1}, d_{j2}, \cdots, d_{jn}\}$ is the list of dialect expressions with respect to q_i, and $L_j = \{l_{j1}, l_{j2}, \cdots, l_{jn}\}$ are the corresponding boolean values of D_j.

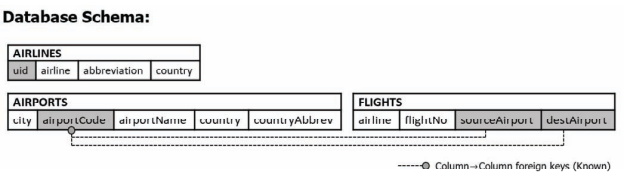
In addition, to make the training more efficient, we use a subset of $D^{'}$ of D for the training of the re-ranking model. That is, after the retrieval model is trained, we set a threshold k, and use the retrieval model to inference a subset of $D^{'}$, where the s_i value of the d_i is among top k.

IV. GAR-J

This section discusses the join operations in queries. We first introduce the join annotation to capture the specific semantics and then describe how GAR-J incorporates join annotations to improve the GRAPH-NL method to handle queries with joins.

A. Join Annotation

We observe that the semantics expressed in a SQL query may become much more abstract if the query involves join operations. The intuition is that the output of a join operation is a new "table", and its semantics may not be directly inferred from the names of the tables involved in the join or the names of their columns. The join condition also plays an important role in the semantics of join. Consider the example in Fig. 7a. The meaning of the `airports-flights` join path through the foreign key `destAirport` refers to "arriving flights", which is hard to infer only from the textual information of the table names. Hence, GAR and SMBOP fail similarly by using the incorrect foreign key `sourceAirport` to join the two tables, while GAP fails to generate the join condition at all.



(a) A database schema in the SPIDER benchmark

NL Query:	*Which city has most number of arriving flights?*
SQL Query: (Gold)	`SELECT T1.city FROM airports AS T1 JOIN flights AS T2 ON T1.airportCode = T2.destAirport GROUP BY T1.city ORDER BY COUNT(*) DESC LIMIT 1`
	GAR model
Generated SQL: (Incorrect)	`SELECT T1.city FROM airports AS T1 JOIN flights AS T2 ON T1.airportCode = T2.sourceAirport GROUP BY T1.city ORDER BY COUNT(*) DESC LIMIT 1`
	GAP model
Generated SQL: (Incorrect)	`SELECT T1.city FROM airports AS T1 JOIN flights AS T2 GROUP BY T1.city ORDER BY COUNT(*) DESC LIMIT 1`
	SMBOP model
Generated SQL: (Incorrect)	`SELECT T2.city FROM flights AS T1 JOIN airports AS T2 ON T1.sourceAirport = T2.airportCode GROUP BY T2.city ORDER BY COUNT(*) DESC LIMIT 1`

(b) A pair of NL-SQL queries and the corresponding translation results of GAR and the other two NLIDB models

Fig. 7: An example from the SPIDER benchmark

Based on these observations, we introduce join annotations to capture the peculiar join semantics. We formulate the join annotation with the following four aspects of information:

- *Joining Tables.* The annotation specifies which tables are involved in the given join operation.
- *Join Condition.* The annotation specifies the condition used in the given join operation. A condition conveys a relationship used in joining the tables.
- *Join Description.* The annotation provides a description that represents the semantics of the "new" table, much like a new table name.
- *Table Keys.* The annotation provides the key information of the "new" table.

As an example, the join operation used in the sample SQL query in Fig. 7b is annotated as follows,

Join Annotation	
Joining Tables:	`airports` and `flights`
Join Condition:	`airports.airportCode=flights.destAirport`
Join Description:	*the flights arrive in the airports*
Table Keys:	*flight*

How to add join annotations automatically or semi-automatically is an interesting research problem. In this paper, we use a manual way to add join annotations by looking into the given sample SQL queries for each given database, and demonstrate its usefulness for queries with joins in our experiments as shown in Section V-E.

B. Dialect Builder in GAR-J

Note that the NL expression generated by GRAPH-NL is simply concatenating the labels of the graph elements, which may only express the operations but fail to capture the implicit semantics. For example, the following NL expression is the translation result in GAR for the "gold" SQL query presented in Fig. 7b,

Find the city of airports regarding to airports with flights. Return the top one result for each city of airports in descending order of the number of airports with flights.

Although the above NL sentence mostly talks about what the query operations do, it does not capture the idea that the description is for arriving flights in airports and the asterisk ("*") used in the `ORDER BY` clause means "flights". This will prevent GAR from finding the correct dialect expression (and hence the SQL query).

To tackle the above problem, GAR-J further incorporates the join annotations of a given databases into the translation process, and hence improves the dialect translation results. More specifically, GAR-J first examines the parse tree and adds the join annotations as the "labels" to the tree as follows:

- If a set of sub-trees of the parse tree can be mapped to a join operation of the SQL query, we first normalize the parse tree by converting other forms (e.g., with the `WHERE` clause) of the join operation to be the form of the "join" type. After the normalization, the set of sub-trees should be rooted with the same non-terminal node, and hence we add the join annotation to the non-terminal node to represent the "join" semantics.
- If a terminal node is an asterisk ("*"), we look into the parse tree to find the related node associated with either a table annotation (name) or join annotation, and then use the "Table Keys" information of the annotation to annotate this asterisk node.

We also present the dialect generation process for the "gold" SQL query in Fig. 7b below. As can be seen, in Fig. 8, each NL phrase is formed by the associated labels of either the terminal nodes or the "join" non-terminal node found in each sub-tree. In particular, since GAR (and hence GAR-J) knows the key information (i.e., flight) of the join annotation, the asterisk node is annotated as "flight", and hence `COUNT(*)` is interpreted to "the number of flights" accordingly.

V. EXPERIMENTAL EVALUATION

This section evaluates GAR using the three existing NLIDB benchmarks and then further discusses the importance of the join semantics using the new QBEN benchmark as well as the results of a user study to show the annotation cost. We use the normalization script provided by the SPIDER benchmark to do the query normalization[7] and then evaluate the translation accuracy results on the validation set of SPIDER and the test sets of GEO, MT-TEQL and QBEN, respectively.

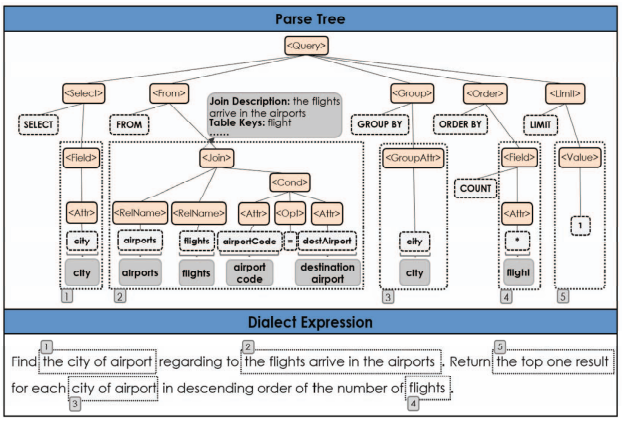


Fig. 8: An example of dialect expression generation with the help of join annotation.

A. Experimental Setup

1) Benchmarks: We use three NLIDB benchmarks to evaluate the performance of GAR: GEO, SPIDER and MT-TEQL. Table 3 provides the detailed statistics about the three benchmarks (and the QBEN benchmark).

GEO is a dataset that consists of NL queries over some geographical data in the form of a relational database with a single table about the United States (GEObase). The corresponding "gold" SQL queries are provided in [32]. The train, validation, and test sets are all on the same database table.

SPIDER is a large-scale benchmark for complex and cross-domain NL2SQL tasks. The benchmark splits SQL queries into four types: *Easy, Medium, Hard*, and *Extra Hard*, based on their hardness level. Unlike other existing NLIDB benchmarks, SPIDER uses different databases in train and validation data sets. That is, a database schema is used exclusively for either training or validation, but not both.

Since the test set of SPIDER is hidden behind an evaluation server, the experiments we perform are on the validation set.

MT-TEQL proposes a metamorphic testing-based framework to conduct semantics-preserving transformations toward utterances and schemas. MT-TEQL starts from the SPIDER validation set and automatically generates a test set of a total of 62,430 transformed testing samples. In our experiment, we randomly sampled 10,000 testing queries as the test set.

2) Training Settings: The following illustrates the implementation details about the retrieval model and re-ranking model used in our experiments[8]:

[7]The script decomposes each SQL into several clauses, respectively.

[8]We also tried some alternatives to the retrieval and re-ranking models, but none worked well.

TABLE 3: The statistics of NLIDB benchmarks

Benchmark	Data Set	Databases	Average Number of Tables per Database	Total Queries	Nested	With ORDERBY	With GROUPBY	Having Compound Queries
GEO	Train/Validation/Test	the same one	the same one	585/47/280	188/19/98	17/0/18	11/0/14	0/0/0
SPIDER	Train/Validation	146/20	4.1/4.17	8659/1034	1249/155	1803/237	1953/277	526/78
MT-TEQL	Validation/Test	20/63,464	4.17/4.34	1034/62,430	155/9949	237/14813	277/16404	78/4588
QBEN	Train/Sample/Test	146/7/7	4.1/4.86/4.86	8659/293/200	1249/28/17	1803/38/24	1953/39/25	526/1/1

Retrieval Model The embedding layer of the retrieval model is initialized with stsb-mpnet-base-v2[9] pre-trained model. We use the Adam [33] optimizer with a learning rate of 2e-5 and warmup over the first 10% of total steps to fine-tune the model.

Re-ranking Model is initialized with the ROBERTA [34] pre-trained model. We use the Adam optimizer with a learning rate of 5e-6 and adopt a learning rate schedule that reduces the learning rate by a factor of 0.5 once learning stagnates, i.e., the training metric has stopped improving, in training.

To accelerate the training phase (and the inference phase), we only leverage the trained retrieval model to encode both the NL queries and the large set of dialect expressions to get the corresponding sentence embeddings. We then use the Faiss library [35] for efficient similarity search to get the closest subset of dialect expressions for each given NL query.

To better support the listwise learning paradigm, we further group the training triples by NL query. We set the threshold k to 100 to obtain a list of 100 dialects for each NL query and set the batch size to 2 due to the GPU memory limit. We use the listwise algorithm NeuralNDCG [36] to train the model.

3) Inference Settings: We use the same threshold (i.e., k=100) as in the training phase to get a subset from a large dialect expression collection for the retrieval model and then pass it along to the re-ranking model for the final inference.

Sample Queries Since GEO and SPIDER benchmarks only provide test queries for their databases, we adopt the following evaluation setting to evaluate GAR. We first use the SQL queries of the SPIDER validation set and the GEO test set to generate generalized query sets. Then we rule out all the ground truth queries from the generalized query sets and use the sets as the sample queries. For the MT-TEQL and QBEN benchmarks, we use the SPIDER validation set and the sample query set as sample queries and then evaluate on the test set.

For each database of the benchmarks, we randomly chose 20,000 generalized SQL queries[10] from the large sets resulting from the data preparation process and then made the inference. We run the data preparation process five times for each database in the benchmarks and report the average results.

Value Post-processing As GAR masks out the specific values during the generalization process and does not use the cell values of the databases for the translation process, after getting the top-ranked results, we examine the dialect expressions in the result set: If a value (e.g., "Spain") appeared in the given NL query, it strongly indicates that a particular column (e.g., "country name") should be mentioned in the dialect expression. Otherwise, the result set will drop those dialect expressions that do not include the column name.

We also use this post-processing step to specify values for the translation results of GAR for the purpose of evaluating on *execution accuracy* metric described below.

4) Evaluation Metrics: Following are the metrics we adopt to assess model performance:

Translation Accuracy If the top-ranked SQL query exactly matches the "gold" SQL, then the translation is said to be accurate. It is a performance lower bound since a semantically correct SQL may differ from the "gold" one syntactically.

This metric is the same as the *Exact Match Accuracy* metric suggested by SPIDER. After query normalization, the metric is calculated by conducting set comparison in each SQL clause.

Execution Accuracy evaluates if the execution result matches the ground truth by executing the generated SQL query against the underlying relational database. This metric is the same as the *Execution Match Accuracy* metric introduced in SPIDER.

Translation Precision at K (denoted Precision@K), where K is a positive integer, is the number of NL queries that an NLIDB system has the "gold" SQL queries in the top-K translation results divided by the total number of NL queries. In our experiments, we choose K to 1, 3, and 10, respectively.

Translation MRR (Mean Reciprocal Rank) is a statistical measure [37] that can be used to evaluate a ranked list of SQL queries for each NL query. The metric is defined as follows,

$$MRR = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{rank_i} \tag{1}$$

where N denotes the number of given NL queries, $rank_i$ refers to the rank position of the "gold" SQL query for the i^{th} NL query. Thus, the closer the value of MRR is to 1, the more effective the translation ranking scheme is.

B. SPIDER&GEO *Results*

We compare GAR with four state-of-the-art machine learning-based models[11], GAP, SMBOP, RAT-SQL [6] and BRIDGE [7]. Fig. 9 shows the overall accuracy of GAR comparing to the four models on the two existing benchmarks.

The results of the five methods on the GEO benchmark are almost on par, all at around 70.0%, which is not as good as those earlier rule-based NLIDB systems. The main reason is that machine learning-based models may not get sufficiently trained since the GEO benchmark only has one database, and its number of training queries is small.

One notable result is that in the current setting, GAR achieves 78.5% translation accuracy on the SPIDER validation

[9]https://huggingface.co/sentence-transformers/stsb-mpnet-base-v2

[10]We attempted to increase the generalization size to 30,000, but the result was not significantly different, so we fixed the size in the current setting.

[11]Another state-of-the-art model NATSQL [10] was developed based on RAT-SQL, but we fail to reproduce as the model code is unavailable.

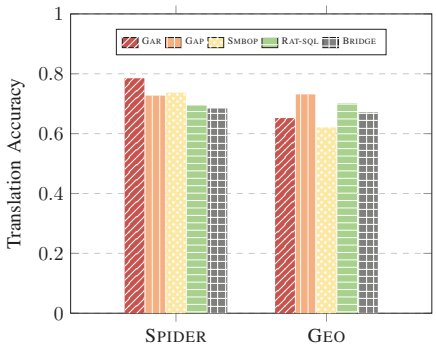


Fig. 9: Translation accuracy on the corresponding validation/test set of the two NLIDB benchmarks.

set, outperforming the best reported accuracy 76.4% on the SPIDER leaderboard by 2.1%.

TABLE 4: Breakdown results on the SPIDER validation set

Model	Easy	Medium	Hard	Extra Hard	Overall	*Exec.*
GAR	0.907	0.816	**0.787**	**0.518**	**0.785**	0.726
SMBOP	0.890	0.791	0.644	0.470	0.737	0.752
BRIDGE	0.911	0.733	0.540	0.392	0.687	0.680
GAP	0.915	0.742	0.644	0.494	0.727	0.349
RAT-SQL	0.851	0.735	0.580	0.476	0.694	0.341

Next, we conduct additional experiments to better understand GAR. Table 4 provides a breakdown of the translation accuracy and the execution accuracy on the SPIDER validation set. Unsurprisingly, the performance of all the models drops with increasing difficulty. However, the performance of GAR is much more stable over the four categories. In particular, GAR attains 78.7% accuracy for the "Hard" queries (176 out of 1034 queries) and achieves 51.8% accuracy in the "Extra Hard" category (166 out of 1034 queries), which surpasses the best counterpart (i.e., SMBOP) by 4.8% absolute improvement. In addition, by using value post-processing step, GAR achieves 72.6% execution accuracy on the SPIDER validation set, surpassing the other models, except the SMBOP model.

We also present the results on the SPIDER validation set in terms of different SQL clause types in Table 5. Overall, the performance of GAR is better over different SQL clauses compared with the other four models. Notably, GAR is better at handling complex SQL queries with nested sub-queries, which achieves 69.8% accuracy.

TABLE 5: Translation accuracy on SPIDER by SQL clause types

Model	Nested	Negation	ORDERBY	GROUPBY	Others
GAR	**0.698**	0.811	0.745	0.679	0.853
GAP	0.472	0.600	0.710	0.679	0.825
SMBOP	0.509	0.611	0.732	0.705	0.819
RAT-SQL	0.453	0.558	0.688	0.649	0.784
BRIDGE	0.528	0.589	0.636	0.568	0.793

Next, we study the effectiveness of the final ranking of GAR. Note that in order to calculate the MRR values, we treat the reciprocal rank as 0 if the "gold" dialect expression is not returned in the final top-10 ranked results for a given NL query. Table 6 shows the precision and MRR values over the two NLIDB benchmarks, which indicates that in most cases, GAR can correctly select the closest dialect expression (and hence the SQL query) in the first few returned results.

TABLE 6: Precision and MRR values of GAR

Dataset	MRR	Precision@1	Precision@3	Precision@10
SPIDER	0.823	0.785	0.859	0.875
GEO	0.680	0.652	0.679	0.680

To evaluate the efficiency of GAR, we compare the processing time with that of the other four models. Note that since the data preparation process of GAR can be done entirely offline, we make the the comparison in an online setting. That is, we assume that all the trained neural network models in all the methods have already been loaded into the memory, and in particular, the generalized queries for the underlying database in GAR have been generated offline[12]. Table 10 shows that all models can react to a user query in real-time (100s milliseconds), though GAR needs to take about 2× more processing time than the other models (except SMBOP) since ranking a large set of dialect expressions may be relatively time-consuming. Note that SMBOP fails on almost all "Extra Hard" queries and returns invalid queries as the translation results, so a notable drop in its response time can be observed.

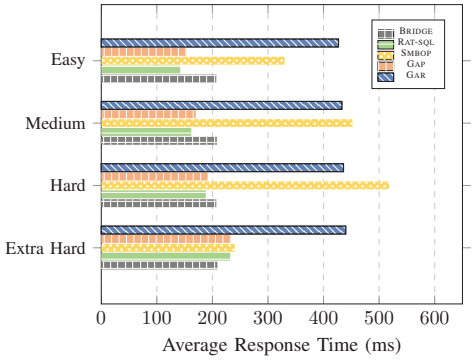


Fig. 10: Comparison of the average response time on the SPIDER validation set, in terms of SQL difficulty levels.

C. MT-TEQL Results

Table 7 presents the results experimented on MT-TEQL. GAR achieves 78.4% translation accuracy on the unknown test set by utilizing the SPIDER validation set as the sample queries, outperforming the other two baseline models.

TABLE 7: Translation results on a randomly selected test subset (10,000 queries) of the MT-TEQL benchmark

Model	Overall	*Exec.*
GAR + SPIDER validation set	0.784	0.693
SMBOP	0.726	0.705
BRIDGE	0.648	0.626
GAP	N/A	N/A
RAT-SQL	N/A	N/A

Since the MT-TEQL benchmark does not publish the test databases, we cannot evaluate the RAT-SQL and GAP models as they rely on the database content for the schema linking.

[12]For the 20,000 generalization size, GAR approximately takes 65s to generate the SQL queries for each database in SPIDER benchmark on average. In our work, we take the naive approach to make the generalization and leave the optimization for future exploration.

D. Ablation Study

We conduct an ablation study for both the dialect builder and second-stage re-ranking model[13] to verify the effectiveness of these designs. As shown in Table 8, the performance of the first-stage retrieval model drops sharply without using the dialects, while the re-ranking model retains a good performance result in the setting. Moreover, we can see that the performance of GAR has a remarkable drop without using the re-ranking model, proving its importance in our approach.

TABLE 8: The ablation study of GAR on the SPIDER validation set. The "w/o Dialect Builder" denotes learning the two ranking models using SQL queries directly.

Model	Retrieval Model Miss Count	Re-ranking Model Miss Count	Overall
Base Model (GAR)	33	132	0.785
w/o Dialect Builder	578	60	0.330
w/o Re-ranking Model	527	N/A	0.435

E. Experiments with GAR-J

To explore the importance of the join semantics, we add the evaluation of the combination of GAR with join annotations (GAR-J). Specifically, we manually add the join annotations for QBEN benchmark (and also GEO and SPIDER), generate the dialect expressions for all the generalized SQL queries with the help of the annotations, and then re-train the two ranking models using the same training settings as above.

QBEN uses the same train set as SPIDER but develops a sample set and a test set that consists of 7 databases that are different from those in the train set. The sample and test sets include 293/200 manually-curated NL-SQL query pairs. Each database has a number of sample and test queries, where the test queries for each are component-similar to those in the sample set. The test databases of QBEN are made so that some insights into the schemas are needed to construct the correct SQL queries. Below is an example from the test set of QBEN:

NL: *Give the "red bull" team mechanics' first names.*

SQL:
```
SELECT T2.FName FROM mechanic AS T1
JOIN team_member AS T2 ON T1.MechanicCode=T2.uid
JOIN teams AS T3 ON T2.TeamCode=T3.uid
WHERE T3.Name="red bull"
```

In the above, the meaning of the table formed by joining `mechanic`, `teams`, and `team_member` tables through the foreign keys is unclear from the table names. Hence, all five models, including GAR, fail the translation.

Results As shown in Fig. 11, GAR-J achieves 70.0% accuracy, while GAR achieves 47.0% accuracy, and the other machine learning-based models can only achieve about 20-30% accuracy on the QBEN benchmark. It is important to note that performing annotations bring about 23.0% absolute improvements for GAR-J. The result demonstrates that join annotation can efficiently capture the implicit semantics that is hard to only infer the textual information of the table and column names, especially for the queries with multiple joins.

[13]Since GAR leverages the retrieval model to filter extensive "irrelevant" queries, relying only on the re-ranking model requires a prohibitive computing cost. Therefore, we exclude the retrieval model from the ablation study.

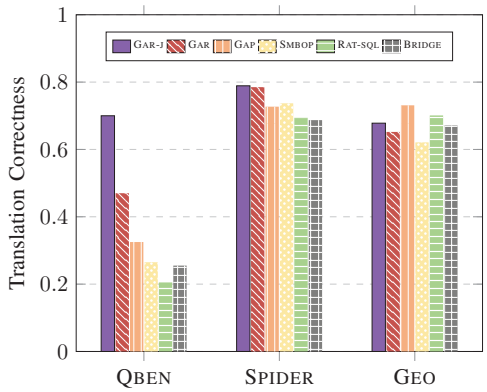


Fig. 11: Translation accuracy on the corresponding validation/test set of the three NLIDB benchmark.

Another interesting finding is that the contribution of join annotations may vary greatly across different NLIDB benchmarks. For SPIDER and GEO benchmarks, GAR-J achieves smaller improvements than the counterpart. We analyze the results and reveal that most of the tables of the databases in the two benchmarks are exact meaningful names, closely matching the terms mentioned in the NL queries. Since GAR is built on top of existing pre-trained language models, GAR can infer most join semantics from the table names and select the SQL query of the corresponding correct dialect expression.

User Study In the rest of this section, we report a user study to assess the annotation cost needed to apply GAR-J. We asked 10 computer science students who have knowledge of SQL and tasked them with annotating the join semantics of each database in the NLIDB benchmarks. Firstly, participants were briefed about the study and then provided sample annotations on a toy database to explain the process. The databases were equally distributed among all the participants and they used the schema information and the given queries in the corresponding benchmark to annotate. We recorded the completion time of each participant for each database.

As shown in Fig. 12, the participants on average took about 3 minutes to complete for small databases with 1∼2 tables; For databases with 3∼5 tables, the time cost was about 7 minutes; For rather complicated databases with 6∼10 tables, it took about 13 minutes on average.

The study shows that the annotation cost is relatively low and can even be completed within a few minutes for small databases. Yet the time requires for the task increases when the database involves more tables and more sample queries.

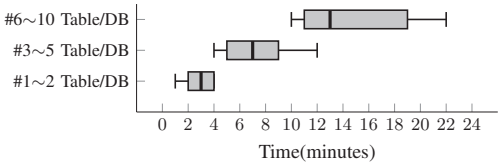


Fig. 12: **User Study Results**: box plot of completion time for different scales of schema size of databases.

F. Analysis of GAR

To gain more understanding of GAR (and GAR-J), we examine the failed cases in terms of translation accuracy. Table 9 lists the number of incorrectly answered NL queries in the three NLIDB benchmarks, with respect to the three steps (data preparation process, retrieval model, and re-ranking model). We identify the following three major causes for the failures.

TABLE 9: Error analysis on each step of GAR/GAR-J

Dataset	Data Preparation Miss Count		Retrieval Model Miss Count		Re-ranking Model Miss Count	
	GAR	GAR-J	GAR	GAR-J	GAR	GAR-J
SPIDER	86	86	33	33	132	120
GEO	8	8	63	63	31	13
QBEN	7	7	34	21	65	32

- **Generalization Coverage Problem.** 22.8% of the failed cases (i.e., the data preparation miss count) are because GAR (and GAR-J) fail to generate the "gold" SQL queries. The reason is that some databases in the benchmarks are provided a relatively varied and large number of SQL queries, leading to a poor coverage under the pre-defined generalization size (20,000). To some extent, such failures can be eliminated by increasing the size of the generalization set, but in view of the extremely large sample SQL query set, finding a more effective and efficient way to balance between achieving broad generalization coverage and limiting the number of resulting queries may be indispensable.
- **Dialect Interpretation Problem.** 25.4% of the failed cases are caused by the retrieval model mostly due to semantic-equivalent sentences with a clear length difference. For example, for the NL query *How long is the longest river in California?* (GEO), following is the dialect generated,

  Find the length of river. Return results only for river that traverse is California and river that length is the maximum length of river that river that traverse is California.

  The above dialect expression mostly captures the idea of finding the longest river in California, but the interpretation is lengthy. Learning a neural network model to help summarize lengthy sentences may avoid such a problem.
- **Re-ranking Problem.** 53.7% of the failed cases are caused by the re-ranking model failing to distinguish similar sentences with minor differences over certain words among the top-ranked results. For instance, an example in SPIDER,

  NL Query: *What is the name and capacity of the stadium with the most concerts after 2013?*
  Dialect for the Gold SQL: *Find the capacity of stadium, the name of stadium regarding to stadiums that had concerts. Return the top one result only for concert that hosting year* ***is or after*** *2014 for each of the stadium in descending order of the number of concerts.*
  Top-ranked Dialect: *Find the capacity of stadium, the name of stadium regarding to stadiums that had concerts. Return top one results only for concert that hosting year* ***is*** *2014 for each of the stadium in descending order of the number of concerts.*

  Such failures may be avoided if we can find a way to prevent the re-ranking model from collapsing from the confounding characteristics among similar sentences.

VI. RELATED WORK

Natural Language Interface to Database. NLIDBs have been studied for several decades in database management and NLP communities. Early works [2], [14], [17], [38]–[42] are rule-based approaches, which use handcrafted grammar and rules to map NL queries to SQL queries specific to a certain database. With the recent success of neural machine translation, many machine learning-based approaches [3], [5]–[13], [43], [44] have been proposed to build NLIDB systems, which treat the NL2SQL problem as a translation task and employ the encoder-decoder architecture to tackle the problem.
Natural language translations. Various ideas have been proposed to address the SQL2NL problem [23], [26], [45]–[48]. [45], [46] discuss the usefulness of translating SQL queries into NL perspective. Earlier attempts [49], [50] explicitly study the problem of translating small databases under certain constraints. [48] employs an iterative training procedure by recursively augmenting the training set to generate the text.
Learning-to-rank The framework of LTR has been successfully applied in multiple areas, such as question answering [51], recommendation [52], and document retrieval [53]. With the recent advances in pre-training for text, many recent works in this field have been proposed [27], [28], [54] by utilizing the pre-trained language models [30], [34].

VII. CONCLUSION & FUTURE WORK

This paper proposed a practical approach named GAR for the NL2SQL problem. GAR learns from sample queries to generate a large set of SQL queries with corresponding dialect expressions and utilizes the LTR technique to find the final result. Experimental results showed that GAR beats the four baseline methods on existing NLIDB benchmarks. In addition, we extended GAR to GAR-J, which incorporates join annotations into the SQL2NL process, to further improve translation accuracy, particularly in the QBEN benchmark.

There are a number of directions for future work. For example, in the current setting, GAR assumes that the given sample queries are representative enough to generate all possible user-intended queries. To make GAR capable of handling "out-of-domain" queries, we may adopt existing translation models as the backbone and employ the model outputs to augment sample queries for better coverage. Also, exploring an automatic or semi-automatic way to add the join annotations would be an interesting research topic. Lastly, another future work is to consider augmenting the query components by examining the underlying database schema to get some more basic components for generalization.

VIII. ACKNOWLEDGMENTS

The authors would like to thank all the anonymous reviewers for their insightful comments and suggestions. This work was supported by NSFC (Grant No. 61732004, 62272106) and the Zhejiang Lab (Grant No. 2021PE0AC01).

REFERENCES

[1] I. Gur, S. Yavuz, Y. Su, and X. Yan, "Dialsql: Dialogue based structured query generation," in *ACL*, 2018.

[2] C. Baik, H. V. Jagadish, and Y. Li, "Bridging the semantic gap with SQL query logs in natural language interfaces to databases," in *ICDE*, 2019.

[3] B. Bogin, J. Berant, and M. Gardner, "Representing schema structure with graph neural networks for text-to-sql parsing," in *ACL*, 2019.

[4] B. Bogin, M. Gardner, and J. Berant, "Global reasoning over database structures for text-to-sql parsing," in *EMNLP*, 2019.

[5] 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*, 2019.

[6] B. Wang, R. Shin, X. Liu, O. Polozov, and M. Richardson, "RAT-SQL: relation-aware schema encoding and linking for text-to-sql parsers," in *ACL*, 2020.

[7] X. V. Lin, R. Socher, and C. Xiong, "Bridging textual and tabular data for cross-domain text-to-sql semantic parsing," in *EMNLP*, 2020.

[8] O. Rubin and J. Berant, "Smbop: Semi-autoregressive bottom-up semantic parsing," in *NAACL*, 2021.

[9] P. Shi, P. Ng, Z. Wang, H. Zhu, A. H. Li, J. Wang, C. N. dos Santos, and B. Xiang, "Learning contextual representations for semantic parsing with generation-augmented pre-training," in *AAAI*, 2021.

[10] Y. Gan, X. Chen, J. Xie, M. Purver, J. R. Woodward, J. H. Drake, and Q. Zhang, "Natural SQL: making SQL easier to infer from natural language specifications," in *EMNLP*, 2021.

[11] T. Scholak, N. Schucher, and D. Bahdanau, "PICARD: parsing incrementally for constrained auto-regressive decoding from language models," in *EMNLP*, 2021.

[12] 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*, 2018.

[13] B. Hui, R. Geng, L. Wang, B. Qin, Y. Li, B. Li, J. Sun, and Y. Li, "S^2sql: Injecting syntax to question-schema interaction graph encoder for text-to-sql parsers," in *ACL*, 2022.

[14] J. Sen, C. Lei, A. Quamar, F. Özcan, V. Efthymiou, A. Dalmia, G. Stager, A. R. Mittal, D. Saha, and K. Sankaranarayanan, "ATHENA++: natural language querying for complex nested SQL queries," *PVLDB*, 13(11):2747–2759, 2020.

[15] A. Suhr, M. Chang, P. Shaw, and K. Lee, "Exploring unexplored generalization challenges for cross-database semantic parsing," in *ACL*, 2020.

[16] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. C. Courville, and Y. Bengio, "Generative adversarial nets," in *NIPS*, 2014.

[17] J. M. Zelle and R. J. Mooney, "Learning to parse database queries using inductive logic programming," in *AAAI*, 1996.

[18] P. Ma and S. Wang, "Mt-teql: Evaluating and augmenting neural NLIDB on real-world linguistic and schema variations," *PVLDB*, 15(3):569–582, 2021.

[19] Y. Gan, X. Chen, Q. Huang, M. Purver, J. R. Woodward, J. Xie, and P. Huang, "Towards robustness of text-to-sql models against synonym substitution," in *ACL*, 2021.

[20] X. Deng, A. H. Awadallah, C. Meek, O. Polozov, H. Sun, and M. Richardson, "Structure-grounded pretraining for text-to-sql," in *NAACL*, 2021.

[21] Y. Gan, X. Chen, and M. Purver, "Exploring underexplored limitations of cross-domain text-to-sql generalization," in *EMNLP*, 2021.

[22] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago, T. Hubert, P. Choy, C. de Masson d'Autume, I. Babuschkin, X. Chen, P.-S. Huang, J. Welbl, S. Gowal, A. Cherepanov, J. Molloy, D. Mankowitz, E. Sutherland Robson, P. Kohli, N. de Freitas, K. Kavukcuoglu, and O. Vinyals, "Competition-level code generation with alphacode," *arXiv preprint*, 2022.

[23] G. Koutrika, A. Simitsis, and Y. E. Ioannidis, "Explaining structured queries in natural language," in *ICDE*, 2010.

[24] Z. Cao, T. Qin, T. Liu, M. Tsai, and H. Li, "Learning to rank: from pairwise approach to listwise approach," in *ICML*, 2007.

[25] K. Lin, B. Bogin, M. Neumann, J. Berant, and M. Gardner, "Grammar-based neural text-to-sql generation," *CoRR*, 2019.

[26] K. Xu, L. Wu, Z. Wang, Y. Feng, and V. Sheinin, "Sql-to-text generation with graph-to-sequence model," in *EMNLP*, 2018.

[27] Z. Dai and J. Callan, "Deeper text understanding for IR with contextual neural language modeling," in *SIGIR*, 2019.

[28] S. Han, X. Wang, M. Bendersky, and M. Najork, "Learning-to-rank with BERT in tf-ranking," *CoRR*, 2020.

[29] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks," in *EMNLP*, 2019.

[30] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in *NAACL*, 2019.

[31] F. Schroff, D. Kalenichenko, and J. Philbin, "Facenet: A unified embedding for face recognition and clustering," in *CVPR*, 2015.

[32] S. Iyer, I. Konstas, A. Cheung, J. Krishnamurthy, and L. Zettlemoyer, "Learning a neural semantic parser from user feedback," in *ACL*, 2017.

[33] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *ICLR*, 2015.

[34] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized BERT pretraining approach," *CoRR*, 2019.

[35] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with gpus," *arXiv preprint*, 2017.

[36] P. Pobrotyn and R. Bialobrzeski, "Neuralndcg: Direct optimisation of a ranking metric via differentiable relaxation of sorting," *CoRR*, 2021.

[37] D. A. Hull, "Xerox TREC-8 question answering track report," in *TREC*, 1999.

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

[39] A. Simitsis, G. Koutrika, and Y. E. Ioannidis, "Précis: from unstructured keywords as queries to structured databases as answers," *PVLDB*, 17(1):117–149, 2008.

[40] L. S. Zettlemoyer and M. Collins, "Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars," in *Uncertainty in Artificial Intelligence, UAI*, 2005.

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

[42] 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.

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

[44] T. Yu, C. Wu, X. V. Lin, B. Wang, Y. C. Tan, X. Yang, D. R. Radev, R. Socher, and C. Xiong, "Grappa: Grammar-augmented pre-training for table semantic parsing," in *ICLR*, 2021.

[45] Y. E. Ioannidis, "From databases to natural language: The unusual direction," in *NLDB*, 2008.

[46] A. Simitsis and Y. E. Ioannidis, "Dbmss should talk back too," in *Conference on Innovative Data Systems Research, CIDR*, 2009.

[47] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *ACL*, 2016.

[48] C. Shu, Y. Zhang, X. Dong, P. Shi, T. Yu, and R. Zhang, "Logic-consistency text generation from semantic parses," in *ACL*, 2021.

[49] A. Simitsis and G. Koutrika, "Comprehensible answers to précis queries," in *CAiSE*, 2006.

[50] A. Simitsis, G. Koutrika, Y. Alexandrakis, and Y. E. Ioannidis, "Synthesizing structured text from logical database subsets," in *EDBT*, 2008.

[51] L. Yang, Q. Ai, D. Spina, R. Chen, L. Pang, W. B. Croft, J. Guo, and F. Scholer, "Beyond factoid QA: effective methods for non-factoid answer sentence retrieval," in *ECIR*, 2016.

[52] Y. Duan, L. Jiang, T. Qin, M. Zhou, and H. Shum, "An empirical study on learning to rank of tweets," in *COLING*, 2010.

[53] T. Liu, *Learning to Rank for Information Retrieval*. Springer, 2011.

[54] X. Sun, H. Tang, F. Zhang, Y. Cui, B. Jin, and Z. Wang, "TABLE: A task-adaptive bert-based listwise ranking model for document retrieval," in *CIKM*, 2020.