Gar++: Natural Language to SQL Translation with Efficient Generate-and-Rank

Yuankai Fan[1(✉)], Can Huang[1], Tonghui Ren[1], Zhenying He[1], X.Sean Wang[1], Xianglian Wu[2], Yue Wang[2], Jiaming Li[2], and Yifan Yang[2]

[1] Fudan University, Shanghai, China
{fanyuankai,zhenying,xywangcs}@fudan.edu.cn,
{huangcan22,thren22}@m.fudan.edu.cn

[2] Transwarp Technology (Shanghai) Co., Ltd., Shanghai, China
{xianglian,yue.wang01,jiaming.li,yifan.yang}@transwarp.io

Abstract. Web applications heavily depend on databases, yet the conventional database interface often presents challenges for efficient data utilization. It is imperative to address the considerable demand emanating from a vast array of end users seeking seamless input of their requirements and effortless retrieval of query results. Natural Language (NL) Interfaces to Databases serve to make databases accessible to end users. Mainstream approaches typically prioritize building language translation models for converting NL queries to SQL queries, while a novel generate-and-rank approach is proposed to achieve this through a procedure involving generation and ranking. Despite yielding superior translation results on the public benchmark, this generate-and-rank approach encounters efficiency issues that may impede its practical application. In this paper, we introduce Gar++, which extends the existing generate-and-rank approach for a more efficient generation and robust ranking procedure. Specifically, Gar++ utilizes the bloom filter to accelerate the data generation process by reducing unnecessary function calls. Additionally, Gar++ provides a brand-new implementation of the ranking module, specifically the re-ranking model, empowered with enhanced language understanding ability. We evaluate the effectiveness of Gar++ on three public benchmarks, namely GEO, Spider, and Mt-teql. Gar++ achieved an overall accuracy of 66.6% on Geo, 80.6% on Spider, and 78.4% on Mt-teql, respectively.

1 Introduction

Web applications are integral to our daily routines, providing a range of functions such as social networking and e-commerce. Yet, their seamless functioning relies heavily on databases for efficient storage, management, and access to data. The introduction of Natural Language Interfaces to Databases (NLIDBs) represents

Y. Fan and C. Huang—Equal contributions.

© The Author(s), under exclusive license to Springer Nature Singapore Pte Ltd. 2024
W. Zhang et al. (Eds.): APWeb-WAIM 2024, LNCS 14963, pp. 411–427, 2024.
https://doi.org/10.1007/978-981-97-7238-4_26

an innovative strategy aimed at improving database accessibility for a diverse array of non-technical users. Due to the continuous maturation of deep learning-based translation techniques and the increasing language capabilities of large language models (LLMs), some interfaces have emerged that convert natural language (NL) queries into SQL queries (NL2SQL), providing an intuitive way to explore the complex data stored in the database.

Existing NL2SQL interfaces primarily involve utilizing generalized sequence-to-sequence models to fulfill the translation from NL to SQL queries. Recently, a novel generate-and-rank framework, Gar [1], has been proposed and achieved excellent translation results. In detail, Gar [1] assumes some sample queries are provided and first generates a set of candidate SQL queries based on the given samples. Then, a rule-based SQL2NL process is involved that translates SQL into NL dialects. Finally, a two-stage learning-to-rank (LTR) procedure is chosen to select the best SQL matching according to the semantic similarity between NL queries and dialect counterparts.

Although Gar boasts remarkable translation accuracy, two key issues hinder its practical application: First, the offline generation process for candidate SQL queries can be time-consuming, taking several hours for some databases; Second, the ranking process, particularly the second-stage re-ranking process, falls short in effectively understanding the semantics of NL and SQL counterparts, leading to numerous mismatched translation.

In this paper, we propose Gar++ to address the aforementioned issues with Gar. In detail, Gar++ utilizes bloom filter to accelerate the generation process of candidate SQL queries, significantly reducing the processing time by avoiding unnecessary function calls. Furthermore, Gar++ also provides a brand-new implementation of the re-ranking module based on the powerful pre-trained language model, T5 [41], empowering this module with enhanced language comprehension capabilities.

To evaluate the effectiveness of Gar++, we conduct our experiments on three public benchmarks, namely Geo [13], Spider and Mt-teql [11]. Gar++ attains an overall accuracy of 66.6% on Geo, 80.6% on the validation set of Spider, and 78.4% on Mt-teql, demonstrating its effectiveness contributed to the generate-and-rank translation paradigm.

Contributions..We highlight our main contributions as follows:

- We introduce Gar++, which extends Gar [1] by enhancing its practicality by addressing two primary efficiency issues - inefficient candidate query generation and ineffective ranking procedure.
- We utilize the bloom filter to accelerate the generation of candidate SQL queries and avoid unnecessary function calls.
- We provide a brand-new implementation of the re-ranking module, empowered with enhanced language understanding ability.
- We evaluate the performance of Gar++ on three public benchmarks and improve the results of Gar.

The remainder of this paper is organized as follows. We first introduce the basic concepts, including the sample queries and the generate-and-rank approach for the NL2SQL task in Sect. 2. We then introduce the details of GAR++ in Sect. 3. We report the experimental results in Sect. 4. Finally, we discuss the related works in Sect. 5 and conclude in Sect. 6.

2 Background

This section first describes sample queries as we rely on them to capture user interests. Then, we re-cap the generate-and-rank approach of GAR [1].

2.1 Sample Queries

As previously stated, to ensure accurate NL2SQL translation when training data is insufficient, particularly when addressing complex queries with multiple SQL clauses, GAR++ initiates by utilizing a set of sample queries to augment the data. This entails the need for sample SQL queries to depict user interactions with the database and reflect the utilization of specific table structures within queries. GAR++ then learns from these samples as a foundation, aiming to accurately translate queries that share similar components with the provided examples.

Recent studies [14–16] have shed light on the necessity of sample queries for databases, as they identified several generalization challenges in a zero-shot cross-domain scenario (i.e., applying a trained model to an unseen database) and advocated for few-shot learning in NL2SQL [12]. For instance, research cited in [16] demonstrates a significant decline in the generalization performance of existing models when domain-specific knowledge is needed for unseen domains.

2.2 Generate-and-Rank Approach for NL2SQL

In this paper, we extend the generate-and-rank approach, GAR, introduced in [1]. Here, we provide a short description of it. GAR has two stages. In the first stage, GAR begins by generalizing the sample set to encompass all SQL queries with similar components. Next, we translate both the sample queries and the generalized ones into natural language expressions (SQL2NL). While these expressions are generally accurate, they may lack naturalness, so we refer to them as "dialect" expressions. In the second stage, when presented with an NL query on a specific database, GAR examines the set of dialect expressions generated during data preparation. It then utilizes an LTR model to identify the most similar dialect expression and, consequently, the SQL translation result.

3 GAR++

A high-level view of GAR++ (based on GAR) can be seen in Fig. 1. In this section, we first describe the overall data generation process and the LTR procedure of

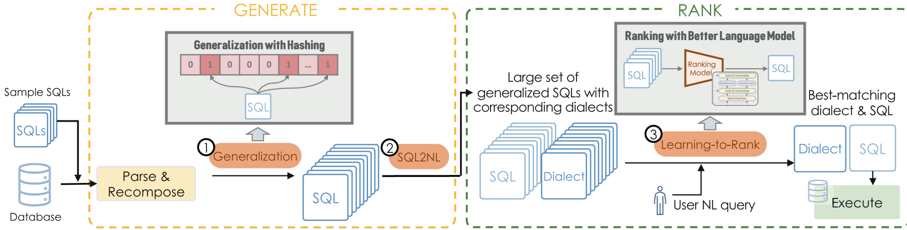


Fig. 1. Overview of GAR++. (1) Generalization: GAR++ generates all possible SQL candidates based on a given small set of sample queries using a hashing technique; (2) SQL2NL: GAR++ synthesizes paired NL description for each generalized SQL query; (3) Ranking: GAR++ utilizes language models to build a two-stage ranking pipeline to determine the best-matching SQL query.

GAR++ in brief, and then discuss in detail the improvements introduced by GAR++ on these processes, respectively.

In the data generation stage, given the sample SQL, GAR++ first recomposes the query parse tree and generates the candidate SQL queries, then a standard bloom filter is utilized to accelerate the verification procedure which involves a hashing technique to avoiding repeated verification for the same parse tree. After the SQL generation process, the same rule-based SQL2NL procedure with GAR is chosen to translate candidate SQL to NL-like dialect.

In the LTR stage, GAR++ utilizes the same bert-based retrieval model to get a relatively small collection of the potential best-matching SQL dialect from a large candidate set. Then a robust T5-based re-ranking network is proposed in GAR++. The network incorporates a T5-like query encoder followed by a pooling layer to compute the ranking score, showcasing an enhanced language comprehension ability that stems from the pre-trained nature of T5. Then the best-matching SQL dialect and SQL are chosen from the candidate set.

3.1 Generalization with Hashing

Given a set of sample SQL queries, the original query generalization process is recursive (as presented in Algorithm 1). It first randomly selects 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.

We observe that the naive *trial and error* method used in Algorithm 1, which involves recomposing to get candidate SQL queries and verifying their validation to determine acceptance or rejection, yields poor time performance for the overall generalization process. In this section, we first analyze the performance issue

```
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 ← random select two parse trees from the set T
    nt ← random select an nonterminal from t1 and t2
    st_1, st_2 ← FIND-SUBTREES(nt, t1, t2)
    // Shuffle the subtrees st_1, st_2 in t1, t2 to form two new trees
    t1_new, t2_new ← 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 ∪ t1_new
    if VALIDATE-TREE(t2_new) then ◁
        T = T ∪ t2_new
    // Recursive call
    GENERALIZE-QUERIES(T)
```

and then present an optimized version of the component-level generalization algorithm to improve its performance.

In the base version of the generalization algorithm, an additional validation step (VALIDATE-TREE in Algorithm 1) is consistently required for a newly-recomposed SQL query t_{new} before it can be added to the result set T. Due to the nature of the trial and error algorithm, this validation step may be repeatedly invoked for the same recomposed SQL query. Furthermore, this repetition may occur more frequently as more generalized SQL queries are generated. For example, Fig. 2 shows the number of verification calls made during the generalization process on a specific database, with a target generalization size of 10,000.

Taking into account the aforementioned performance issue, we propose employing the standard *bloom filter* [40] to enhance the generalization process in the following manner: For a given fixed generalized size k, we allocate a bit vector consisting of m_k bits and use n independent hash functions $\{h1, h2, ...h_n\}$ with the range of each h_i being integer values between 0 and $m_k - 1$. Initially, all m_k bits in the vector are set to 0. To process each recomposed SQL query t_{new}, we then apply n hash functions and utilize the resulting hashed values as indices in the bit vector. If all bits indexed by $h_i(t_{new})$ in the vector are 1 for every $i \in \{1, 2, ...n\}$, we skip the validation step and directly abandon the SQL query; Otherwise, if any of the bits are 0, we then change those bits from 0 to 1, and proceed the validation step. In this way, duplicate recomposed results during the generalization process can be effectively avoided.

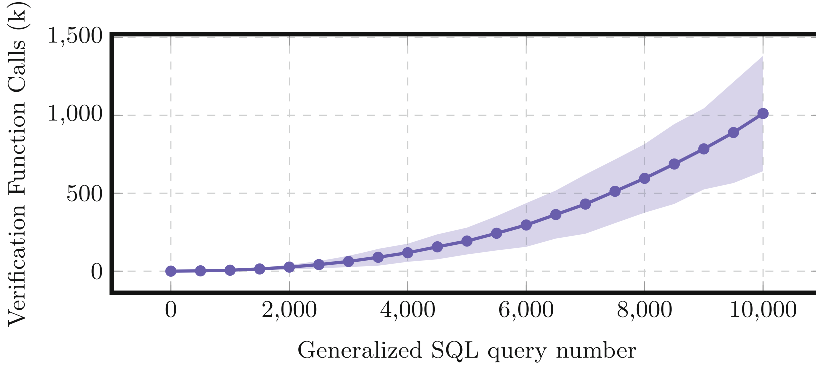


Fig. 2. Simulated generalization on a specific database (named) in the SPIDER benchmark.

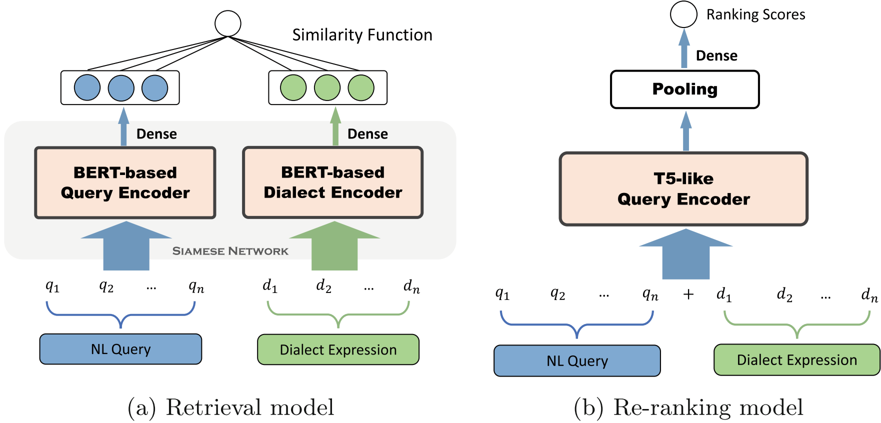
3.2 Ranking with Better Language Model

Following GAR, GAR++ implements a two-stage ranking pipeline with two separate models. In the first stage, a coarse-grained "retrieval model" reduces a large set of potential best-matching dialect expressions to a smaller, more manageable collection. In the second stage, a fine-grained "re-ranking model" is applied to this smaller set to determine the final top-ranked dialect expressions.

Figure 3 presents the overall architectures of the two ranking models. More specifically, the network for the retrieval model is based on the *Siamese* BERT-*network* introduced in [23]. This retrieval model modifies the pre-trained BERT network [21] using Siamese and triplet network structures [22] to derive sentence embeddings. On the other hand, the network of the re-ranking model uses the *encoder-only structure* [45] derived from T5 architecture to output real numbers (i.e., similarity scores) between the input NL-dialect query pairs.

Re-ranking Model Design. Despite the extensive exploration of BERT-based pre-trained language models [20,46,47], which has led to substantial progress in text ranking, there are still instances of unsatisfactory performance, especially when it comes to fine-grained ranking. Table 1 presents an example of the ranking results produced by the BERT-based ranking model introduced in [20]. Mismatched sentences are usually partially irrelevant with phrases of inconsistent semantics (the mismatched "less than", "the abbreviation", etc.).

We posit that the aforementioned issue predominantly stems from two key factors: (1) **Discrepancy in Optimization Objective.** Most existing approaches are fine-tuned not explicitly for ranking but models geared towards text generation [49] or classification [48]. This divergence in focus might inadvertently hamper their optimization for ranking performance. (2) **Constrained Model Capacity.** In light of the growing emergence of increasingly more powerful language models such as T5 [41] and GPT-3 [51], which have showcased the



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

Fig. 3. The architectures of the two ranking models. (a) The retrieval model is built upon the dual-tower architecture, with one encoder used for NL query embedding and the other for dialect embedding; (b) The re-ranking model uses a unified encoder based on a pre-trained language model to encode both inputs.

Table 1. An example of an NL query, a group of mismatched dialects, and the corresponding matched dialect. Text segments underscored indicate a mismatch.

NL Query	*Which airlines have at least 10 flights?*	**Similarity Score**
Mismatched Dialects	Find the airline regarding to airlines with flights. Return results only for the number of flights **less than** value for each airline of airlines.	0.82
	Find **the number of flights**, the airline regarding to airlines with flights. Return results only for the number of flights **less than** value for each airline of airlines.	0.76
	Find the airline, **the abbreviation** regarding to airlines with flights. Return results only for the number of flights **less than** value for each airline of airlines.	0.71
Matched Dialect	Find the airline regarding to airlines with flights. Return results only for the number of flights greater than value for each airline of airlines.	0.67

potential for larger and more capable models, the model capacity of BERT-based counterparts could potentially become a limiting factor for text ranking.

Hence, we take inspiration from prior works [45,48] to craft the re-ranking model rooted in the T5 language model, subjecting it to fine-tuning for targeted ranking purpose. Specifically, we use the encoder of T5 as the query encoder to encode NL queries and dialect expressions jointly. We add a pooling layer (e.g., mean pooling) to aggregate them into a unified embedding vector. This vector is then fed into a dense layer, directly projecting it into the ranking score. This model structure allows us to obtain ranking scores for each NL-dialect pair effortlessly, enabling us to utilize a training approach akin to LTR models. We can then perform fine-tuning using suitable ranking losses to refine the model.

Similar to 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 $\{(q_i, d_i, s_i)\}_{i=1}^{N}$, where q_i is an NL query, d_i is a dialect expression and s_i is the corresponding similarity score between d_i and q_i.

As we use the listwise approach [39] 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 $\{(q_j, D_j, S_j)\}_{j=1}^{M}$, 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 $S_j = \{s_{j1}, s_{j2}, \cdots, s_{jn}\}$ are the corresponding similarity scores of D_j.

4 Experimental Evaluation

This section evaluates Gar++ using the three existing NLIDB benchmarks. We use the normalization script provided by the Spider benchmark to do the query normalization and then evaluate the translation accuracy results on the validation set of Spider and the test sets of Geo and Mt-teql, respectively.

4.1 Experimental Setup

Benchmarks. We use three benchmarks to evaluate Gar++: Geo, Spider and Mt-teql. Table 2 shows the statistics about the three benchmarks.

Table 2. The statistics of NLIDB benchmarks

Benchmark	Data Set	Databases	Average Tables per Database	Total Queries	Nested	Having Compound Queries
Geo	Train/Dev/Test	the same one	the same one	585/47/280	188/19/98	0/0/0
Spider	Train/Dev	146/20	4.1/4.17	8659/1034	1249/155	526/78
Mt-teql	Dev/Test	20/63,464	4.17/4.34	1034/62,430	155/9949	78/4588

Geo is a dataset that consists of NL queries concerning geographical data, specifically a relational database featuring a single table focused on the United States. The "gold" SQL queries are provided in [24]. All three sets - train, validation, and test - are based 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. As the test set is concealed within an evaluation server, our experiments focus on the validation set instead.

Mt-teql is a framework that uses metamorphic testing to perform semantics-preserving transformations on 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.

Training Settings. The following illustrates the implementation details of the ranking models used in our experiments:

Retrieval Model. The embedding layer is initialized with *stsb-mpnet-base-v2* pre-trained model. We use the Adam [25] optimizer with a learning rate of 2e-5 and warm-up over the first 10% of total steps to fine-tune the model.

Re-ranking Model is initialized T5 with the Large scale. We employ the Adam optimizer with a learning rate of 5e-5 and leverage DeepSpeed[1] while enabling ZeRO-3 [52] optimization for distributed training across two GPUs.

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 [27] 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. We use the listwise algorithm NeuralNDCG [28] to train the model.

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 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 excluded all the ground truth queries from the generalized query sets and used the sets as sample queries. For Mt-teql and Qben, 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 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 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

[1] https://www.deepspeed.ai.

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.

Evaluation Metrics. Here are the metrics we use to assess model performance:

Translation Accuracy. If the top-ranked SQL query exactly matches the "gold" SQL, then the translation is deemed accurate. It is a performance lower bound since semantically correct SQL can vary syntactically. This metric aligns with the *Exact Match Accuracy* metric suggested by SPIDER.

Execution Accuracy evaluates if the result matches the ground truth by executing the generated SQL query against the underlying database. This metric aligns with the *Execution Match Accuracy* metric introduced in SPIDER.

Translation Precision at K (denoted Precision@K), where K is a positive integer, represents the number of NL queries where an NLIDB system has the "gold" SQL queries among the top-K translation results, divided by the total number of NL queries. We specifically select K to 1, 3, and 10.

Translation MRR (Mean Reciprocal Rank) is a statistical measure [29] used to evaluate a ranked list of SQL queries. The metric is defined as follows,

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

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

4.2 SPIDER&GEO Results

We compare GAR++ with five state-of-the-art machine learning-based models (including GAR), GAP, SMBOP, RAT-SQL [5] and BRIDGE [6]. Figure 4 shows the overall accuracy of GAR++ compared to the five models on the two existing benchmarks.

The results of the six 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.

An outstanding achievement of GAR++ is its translation accuracy of 80.6% on the SPIDER validation set, marking a notable improvement of 2.1% points from 78.5% in GAR.

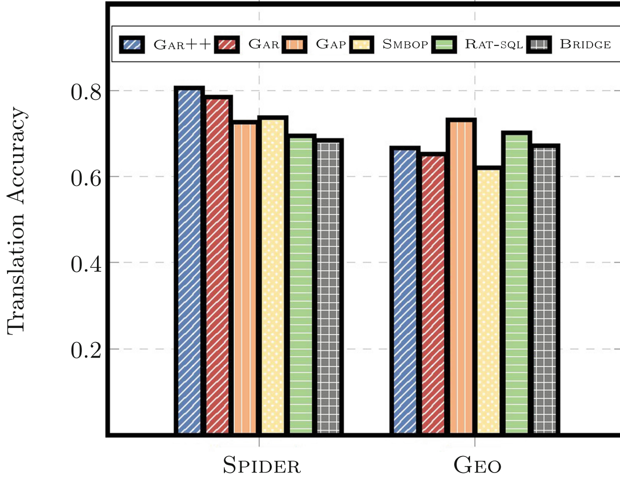


Fig. 4. Translation accuracy on the validation/test sets of the benchmarks.

Next, we conduct additional experiments to better understand Gar++. Table 3 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 53.0% accuracy in the "Extra Hard" category (166 out of 1034 queries), which surpasses Gar by 1.2% absolute improvement, while observes a slight performance drop of 1.1% in the "Hard" category.

Table 3. Breakdown results on the Spider validation set

Model	Easy	Medium	Hard	Extra	Overall	*Exec.*
Gar++	**0.940**	**0.845**	0.776	**0.530**	**0.806**	**0.731**
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

We also present the results on the Spider validation set in terms of different SQL clause types in Table 4. Overall, the performance of Gar++ is better over different SQL clauses compared with the other four models. Notably, Gar++ is better at handling SQL queries with grouping and ordering, which achieves 76.0% and 79.2% accuracy, respectively.

Table 4. Translation accuracy on SPIDER by SQL clause types

Model	Nested	Negation	ORDERBY	GROUPBY	Others
GAR++	0.660	**0.811**	**0.792**	**0.760**	**0.853**
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. Table 5 shows that in most cases, GAR++ can correctly select the closest dialect expression (and hence the SQL query) in the first few returned results.

Table 5. Precision and MRR values of GAR++

Dataset	MRR	Precision@1	Precision@3	Precision@10
SPIDER	0.824	0.806	0.843	0.854
GEO	0.680	0.666	0.682	0.687

Performance Evaluation. To assess the efficiency of GAR++, we conduct two additional experiments for performance evaluation. We first evaluate *the generalization performance* before and after implementing the optimization discussed in Sect. 3, and then 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 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.

Table 6 presents the generalization performance results of GAR++. As can be seen, the average validation calls can be effectively reduced after the optimization introduced in Sect. 3, resulting in the performance upgrade over the overall generalization time cost.

4.3 MT-TEQL Results

Table 7 presents the results experimented on MT-TEQL[2]. With enhanced with T5, GAR++ achieves 80.4% translation accuracy on the unknown test set by

[2] Since MT-TEQL 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.

Table 6. The generalization performance on the SPIDER validation set comprises an average database size of four tables and includes 23 sample queries.

	Average Validation Calls (k)	Average Generalization Time (s)
Base Version (GAR)	656	537
Optimized Version (GAR++)	19	394

Table 7. Translation Results on a randomly selected test query sub-set (including 10,000 queries) of the MT-TEQL benchmark

Model	Overall	*Exec.*
GAR++ (w/ SPIDER validation set)	0.804	0.701
GAR (w/ SPIDER validation set)	0.784	0.693
SMBOP	0.726	0.705
BRIDGE	0.648	0.626

utilizing the SPIDER validation set as the sample queries, which attains a 2.1% improvement over GAR and outperforms the other two baseline models.

4.4 Ablation Study

We conduct an ablation study for both the dialect builder and second-stage re-ranking model[3] 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. Furthermore, it is evident that the performance of GAR++ experiences a notable drop when the re-ranking model is not employed, highlighting its crucial role in our approach.

Table 8. The ablation study on 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	88	0.806
w/o Dialect Builder	578	60	0.330
w/o Re-ranking Model	527	N/A	0.435

[3] 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.

5 Related Work

Natural Language Interfaces to Databases (NLIDBs). NLIDBs have been studied for several decades in database management and NLP communities. With the recent success of neural machine translation, many machine learning-based approaches [3–10,30,50] 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. More recently, with the outstanding performance of large language models (LLMs) in many NLP tasks, researchers have begun applying LLMs to the NL2SQL task using various techniques, such as in-context learning [54,55], pre-training [42,57], bootstrapping framework [53], and multi-agents [56].

Natural Language Translation. Various ideas have been proposed to address the SQL2NL problem [17,18,31–33,37]. [31,32] discuss the usefulness of translating SQL queries into corresponding NL perspectives. Earlier attempts [43,44] explicitly study the problem of translating small databases under certain constraints. [33] employs an iterative training procedure by recursively augmenting the training set.

Learning-to-Rank. The framework of LTR has been successfully applied in multiple areas, such as question answering [34], recommendation [35], and document retrieval [36]. With the recent advances in pre-training for text, many recent works in this field have been proposed [19,20,38] by utilizing the pre-trained language models [21,53].

6 Conclusion

This paper introduced GAR++, an extension of the previous generate-and-rank approach GAR, to address the NL2SQL problem. GAR++ advances by learning from sample queries to efficiently generate a large set of SQLs paired with corresponding dialect expressions with hashing structures and by employing more potent ranking models to identify optimal matches. Experimental results on public benchmarks demonstrated that GAR++ can further improve the translation performance of GAR and offers a more efficient generate-and-rank solution for the NL2SQL problem.

Acknowledgements. The authors would like to thank all the anonymous reviewers for their insightful comments and suggestions. This work was supported by NSFC (62272106).

References

1. Fan, Y., et al.: GAR: a generate-and-rank approach for natural language to SQL translation. In: Proceedings of the 39th International Conference on Data Engineering, pp. 110–122 (2023)

2. Fan, Y., Ren, T., He, Z., Wang, X., Zhang, Y., Li, X.: GenSQL: a generative natural language interface to database systems. In: Proceedings of the 39th International Conference on Data Engineering, pp. 3603–3606 (2023)
3. Bogin, B., Berant, J., Gardner, M.: Representing schema structure with graph neural networks for text-to-SQL parsing. In: Association for Computational Linguistics, pp. 4560–4565 (2019)
4. Guo, J., et al.: Towards complex text-to-SQL in cross-domain database with intermediate representation. In: Association for Computational Linguistics, pp. 4524–4535 (2019)
5. Wang, B., Shin, R., Liu, X., Polozov, O., Richardson, M.: RAT-SQL: relation-aware schema encoding and linking for text-to-SQL parsers. In: Association for Computational Linguistics, pp. 7567–7578 (2020)
6. Lin, X.V., Socher, R., Xiong, C.: Bridging textual and tabular data for cross-domain text-to-SQL semantic parsing. In: Proceedings of the Conference on Empirical Methods Natural Language Process, pp. 4870–4888 (2020)
7. Rubin, O., Berant, J.: SmBoP: semi-autoregressive bottom-up semantic parsing. In: Proceedings of the Human Language Technology Annual Conference North American Chapter Association Computational Linguistics, pp. 311–324 (2021)
8. Shi, P., et al.: Learning contextual representations for semantic parsing with generation-augmented pre-training. In: Proceedings of the AAAI Conference Artificial Intelligence, pp. 13806–13814 (2021)
9. Scholak, T., Schucher, N., Bahdanau, D.: PICARD: parsing incrementally for constrained auto-regressive decoding from language models. In: Proceedings of the Conference on Empirical Methods Natural Language Process, pp. 9895–9901 (2021)
10. Yu, T., et al.: Spider: a large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In: Proceedings of the Conference on Empirical Methods Natural Language Process, pp. 3911–3921 (2018)
11. Ma, P., Wang, S.: MT-Teql: evaluating and augmenting neural NLIDB on real-world linguistic and schema variations. Proc. VLDB Endowment **15**(3), 569–582 (2021)
12. Suhr, A., Chang, M., Shaw, P., Lee, K.: Exploring unexplored generalization challenges for cross-database semantic parsing. In: Association for Computational Linguistics, pp. 8372–8388 (2020)
13. Zelle, J.M., Mooney, R.J.: Learning to parse database queries using inductive logic programming. In: Proceedings of the AAAI Conference Artificial Intelligence, pp. 1050–1055 (1996)
14. Gan, Y., et al.: Towards robustness of text-to-SQL models against synonym substitution. In: Association for Computational Linguistics, pp. 2505–2515 (2021)
15. Deng, X., Awadallah, A.H., Meek, C., Polozov, O., Sun, H., Richardson, M.: Structure-grounded pretraining for text-to-SQL. In: Proceedings of the Human Language Technology Annual Conference North America Chapter Association Computation Linguistics, pp. 1337–1350 (2021)
16. Gan,Y., Chen, X. Purver, M.: Exploring underexplored limitations of cross-domain text-to-SQL generalization. In: Proceedings of the Conference on Empirical Methods Natural Language Process, pp. 8926–8931 (2021)
17. Koutrika, G., Simitsis, A., Ioannidis, Y.E.: Explaining structured queries in natural language. In: Proceedings of the 12st International Conference on Data Engineering, pp. 333–344 (2010)
18. Xu, K., Wu, L., Wang, Z., Feng, Y., Sheinin, V.: SQL-to-text generation with graph-to-sequence model. In: Proceedings of the Conference on Empirical Methods Natural Language Process, pp. 931–936 (2018)

19. Dai, Z., Callan, J.: Deeper text understanding for IR with contextual neural language modeling. In: Proceedings of the ACM SIGIR International Conference on Research and Development in Information Retrieval, pp. 985–988 (2019)
20. Han, S., Wang, X., Bendersky, M., Najork, M.: Learning-to-rank with BERT in TF-ranking. arXiv preprint arXiv:2004.08476 (2020)
21. Devlin, J., Chang, M., Lee, K., Toutanova, K.: BERT: pre-training of deep bidirectional transformers for language understanding. In: Proceedings of the Human Language Technology Annual Conference North America Chapter Association Computational Linguistics, pp. 4171–4186 (2019)
22. Schroff, F., Kalenichenko, D., Philbin, J.: FaceNet: a unified embedding for face recognition and clustering. In: Proceeding of the IEEE Conference on Computer Vision Pattern Recognition, pp. 815-823 (2015)-
23. Reimers, N., Gurevych, I.: Sentence-BERT: sentence embeddings using siamese BERT-networks. In: Proceedings of the Conference on Empirical Methods Natural Language Process, pp. 3980–3990 (2019)
24. Iyer, S., Konstas, I., Cheung, A., Krishnamurthy, J., Zettlemoyer, L.: Learning a neural semantic parser from user feedback. In: Association for Computational Linguistics, pp. 963–973 (2017)
25. Kingma, D.P., Ba, J.: Adam: a method for stochastic optimization. In: Proceedings of the International Conference on Learning Representations (2015)
26. Liu, Y., et al.: RoBERTa: a robustly optimized BERT pretraining approach. arXiv preprint arXiv:1907.11692 (2019)
27. Johnson, J., Douze, M., Jégou, H.: Billion-scale similarity search with GPUs. IEEE Trans. Big Data **7**(3), 535–547 (2021)
28. Pobrotyn, P., Bialobrzeski, R.: NeuralNDCG: direct optimisation of a ranking metric via differentiable relaxation of sorting. arXiv preprint arXiv:2102.07831 (2021)
29. Hull, D.A.: Xerox TREC-8 question answering track report. In: TREC (1999)
30. SQLNet. Generating structured queries from natural language without reinforcement learning. arXiv (2017)
31. From databases to natural language: the unusual direction. In: Proceedings of the International Conference on Application Natural Language Information System, pp. 12–16 (2008)
32. DBMSs should talk back too. In: Conference on Innovative Data System Research (2009)
33. Shu, C., Zhang, Y., Dong, X., Shi, P., Yu, T., Zhang, R.: Logic-consistency text generation from semantic parses. In: Association for Computational Linguistics, pp. 4414–4426 (2021)
34. Yang, L., et al.: Beyond factoid QA: effective methods for non-factoid answer sentence retrieval. In: Proceedings of the 38th European Conference on Information Retrieval, pp. 115–128 (2016)
35. Duan, Y., Jiang, L., Qin, T., Zhou, M., Shum, H.: An empirical study on learning to rank of tweets. In: Proceeding of the International Conference on Computational Linguistics, pp. 295–303 (2010)
36. Learning to Rank for Information Retrieval, pp. 1–285. Springer (2011)
37. Iyer, S., Konstas, I., Cheung, A., Zettlemoyer, L.: Summarizing source code using a neural attention model. Association for Computational Linguistics (2016)
38. Sun, X., Tang, H., Zhang, F., Cui, Y., Jin, B., Wang, Z.: TABLE: a task-adaptive BERT-based ListwisE ranking model for document retrieval. In: Proceedings of the ACM 29th International Conference on Information Knowledge Management, pp. 2233–2236 (2020)

39. Cao, Z. , Qin, T., Liu, T., Tsai, M., Li, H.: Learning to rank: from pairwise approach to listwise approach. In: Proceedings International Conference on Machine Learning, pp. 129–136 (2007)
40. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM **13**(4), 422–426 (1970)
41. Raffel, C., et al.: Exploring the limits of transfer learning with a unified text-to-text transformer. J. Mach. Learn. Res. **21**, 140:1–140:67 (2020)
42. Sun, R., et al.: SQL-PaLM: improved large language model adaptation for text-to-SQL. arXiv preprint arXiv:2306.00739 (2023)
43. Simitsis, A., Koutrika, G.: Comprehensible answers to précis qeries. In: International Conference on Advanced Information System Engineering, pp. 142–156 (2006)
44. Simitsis, A., Koutrika, G., Alexandrakis, Y., Ioannidis, Y.E.: Synthesizing structured text from logical database subsets. In: International Conference Extended Database Technology, pp. 428–439 (2008)
45. Zhuang, H., et al.: RankT5: fine-tuning T5 for text ranking with ranking losses. arXiv preprint arXiv:2210.10634 (2022)
46. Yates, A., Nogueira, R.F., Lin, J.: Pretrained transformers for text ranking: BERT and beyond. In: Proceedings of the ACM SIGIR International Conference on Research and Development in Information Retrieval, pp. 2666–2668 (2021)
47. Lin, J., Nogueira, R.F., Yates, A.: Pretrained transformers for text ranking, BERT and beyond. arXiv preprint arXiv:2010.06467 (2020)
48. Nogueira, R.F., Jiang, Z., Pradeep, R., Lin, J.: Document ranking with a pretrained sequence-to-sequence model. In: Proceedings of the Conference on Empirical Methods Natural Language Process, pp. 708–718 (2020)
49. Ju, J., Yang, J., Wang, C.: Text-to-text multi-view learning for passage re-ranking. In: Proceedings of the ACM SIGIR International Conference on Research and Development in Information Retrieval, pp. 1803–1807 (2021)
50. Li, H., Zhang, J., Li, C., Chen, H.: Decoupling schema linking and skeleton parsing for text-to-SQL. In: Proceedings of the AAAI Conference on Artificial Intelligence, pp. 13067–13075 (2023)
51. Brown, T.B., Mann, B., Ryder, N., et al.: Language models are few-shot learners. In: Proceedings of the Advances Neural Informational Processing System (2020)
52. Rajbhandari, S., Rasley, J., Ruwase, O., He, Y.: ZeRO: memory optimizations toward training trillion parameter models. In: International Conference on High Perfomence Computer Network Storage Analysis, vol. 20 (2020)
53. Fan, Y., et al.: MetaSQL: a generate-then-rank framework for natural language to SQL translation. arXiv preprint arXiv:2402.17144 (2024)
54. Ren, T., et al.: PURPLE: making a large language model a better SQL writer. arXiv preprint arXiv:2403.20014 (2024)
55. Pourreza, M., Rafiei, D.: DIN-SQL: decomposed in-context learning of text-to-SQL with self-correction. In: Proceedings of the Advances Neural Information Processing System (2023)
56. Wang, B., et al.: MAC-SQL: a multi-agent collaborative framework for text-to-SQL. arXiv preprint arXiv:2312.11242 (2023)
57. Li, H., et al.: CodeS: towards building open-source language models for text-to-SQL. arXiv preprint arXiv:2402.16347 (2024)