A Heterogeneous Graph to Abstract Syntax Tree Framework for Text-to-SQL

Ruisheng Cao, Lu Chen, Jieyu Li, Hanchong Zhang, Hongshen Xu, Wangyou Zhang, *Member, IEEE*, and Kai Yu, *Senior Member, IEEE*

***Abstract*—Text-to-SQL is the task of converting a natural language utterance plus the corresponding database schema into a SQL program. The inputs naturally form a heterogeneous graph while the output SQL can be transduced into an abstract syntax tree (AST). Traditional encoder-decoder models ignore higher-order semantics in heterogeneous graph encoding and introduce permutation biases during AST construction, thus incapable of exploiting the refined structure knowledge precisely. In this work, we propose a generic heterogeneous graph to abstract syntax tree (HG2AST) framework to integrate dedicated structure knowledge into statistics-based models. On the encoder side, we leverage a line graph enhanced encoder (LGESQL) to iteratively update both node and edge features through dual graph message passing and aggregation. On the decoder side, a grammar-based decoder first constructs the equivalent SQL AST and then transforms it into the desired SQL via post-processing. To avoid over-fitting permutation biases, we propose a golden tree-oriented learning (GTL) algorithm to adaptively control the expanding order of AST nodes. The graph encoder and tree decoder are combined into a unified framework through two auxiliary modules. Extensive experiments on various text-to-SQL datasets, including single/multi-table, single/cross-domain, and multilingual settings, demonstrate the superiority and broad applicability.**

***Index Terms*—Abstract syntax tree, grammar-based constrained decoding, heterogeneous graph neural network, knowledge-driven natural language processing, permutation invariant problem, text-to-SQL.**

I. Introduction

KNOWLEDGE, such as Wikipedia documents and knowledge graphs, is critical to equip statistics-based models with logical, structural, and other external information. Incorporating knowledge into data-driven deep learning models has been a rising trend in natural language processing (NLP). As a typical knowledge-augmented task, text-to-SQL [2], [3] aims to convert a user question and the corresponding database schema into an executable SQL program, illustrated in Fig. 1(a). Text-to-SQL distinguishes itself from other NLP tasks in that both the input and output require integrating knowledge of refined structures, namely the *graph structure* of input database schema and the *grammatical structure* of output SQL program.

On the input side, database schema is an additional knowledge source to facilitate the comprehension of a user question. It can be represented by a heterogeneous graph containing multiple table or column nodes and various relations[1] among them. For example, edges in the database schema graph of Fig. 1(a) denote foreign key constraints between two columns and table-column belonging relations. Previous work typically utilizes a heterogeneous graph neural network [4], such as relational graph attention network (RGAT) [5], to jointly encode the question words and database schema items and inject the structured knowledge. However, they merely utilize static matrices or vectors to parametrize edge features. This common practice fails to learn useful multi-hop relations and distinguish between 1-hop and multi-hop neighbors for each node. In Fig. 1(b), relation T-HAS-C and C-EXACTMATCH-Q can form a 2-hop relation indicating that, table "*arrangement*" has one column "*course*" which is exactly mentioned in the question. Even though there is no literal overlapping between table "*arrangement*" and the question, we can assert that this table is highly correlated to the question and likely to occur in the target SQL. Although RATSQL [6] introduces some useful multi-hop relations and constructs a dense complete graph as input, it incurs additional difficulty in extrapolating the target SQL sub-graph from the entire database schema graph, see Fig. 1(c). Therefore, 1-hop and multi-hop relations should be treated differently to alleviate the notorious over-smoothing problem [7].

On the output side, SQL programs exhibit strict syntactic and semantic constraints. If no grammatical knowledge of SQL structure is integrated, the decoder will waste enormous amounts of computation in producing ill-formed programs. To resolve this constrained decoding problem, one prominent approach [8] generates the equivalent abstract syntax tree (AST) of the SQL program instead (bottom part in Fig. 1(a)). Unfortunately, previous work introduces unnecessary permutation biases in grammar rules and generates the target tree in a pre-defined canonical order, usually depth-first-search (DFS) and left-to-right (L2R).

Manuscript received 21 July 2022; revised 22 May 2023; accepted 19 July 2023. Date of publication 26 July 2023; date of current version 3 October 2023. This work was supported in part by China NSFC under Grants 62106142 and 62120106006, in part by Shanghai Municipal Science and Technology Major Project under Grant 2021SHZDZX0102, and in part by the Startup Fund for Youngman Research at SJTU (SFYR at SJTU). Part of this work has been accepted as "LGESQL: Line Graph Enhanced Text-to-SQL Model with Mixed Local and Non-Local Relations" in the 59th Annual Meeting of the Association for Computational Linguistics [DOI: 10.18653/v1/2021.acl-long.198]. Recommended for acceptance by M. Moens. *(Corresponding authors: Kai Yu; Lu Chen.)*

The authors are with the X-LANCE Lab, Department of Computer Science and Engineering, MoE Key Lab of Artificial Intelligence, AI Institute, Shanghai Jiao Tong University, Shanghai 200240, China (e-mail: 211314@sjtu.edu.cn; chenlusz@sjtu.edu.cn; oracion@sjtu.edu.cn; zhanghanchong@sjtu.edu.cn; xuhongshen@sjtu.edu.cn; wyz-97@sjtu.edu.cn; kai.yu@sjtu.edu.cn).

Digital Object Identifier 10.1109/TPAMI.2023.3298895

[1]The terms *relation* and *edge* are used interchangeably in this work.

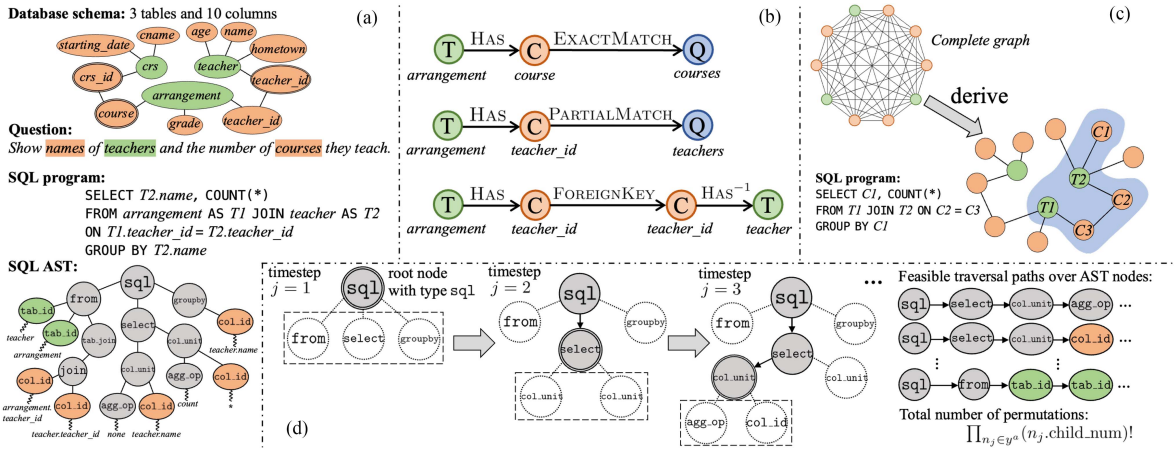

Fig. 1. (a) A text-to-SQL example. (b) Empirically effective multi-hop relations (Q/T/C denotes question/table/column node respectively). (c) 1-hop and multi-hop relations should be distinguished to extract the schema sub-graph. (d) Treat AST generation as a structured Seq2Set problem.

Essentially, the AST generation can be formulated as a structured sequence-to-set (Seq2Set) process, see Fig. 1(d). The decoder selects and expands one node (called *frontier node*) from a set of unfinished nodes (called *frontier node set*, dashed rectangles in Fig. 1(d)) at each timestep. But there is no consensus on which typed child node to choose next. And this decision needs to be made recursively during DFS traversal. Imposing biased order potentially introduces unreasonable priors that prevent the model from learning a more interpretable inference path. For example, Lin et al. [9] has proven that prioritizing FROM clause according to the execution order of SQL is superior to the written order where SELECT clause comes first. More generally, predicting some units with precedence will provide more hints which can benefit the inference of others. However, it would be unrealistic to discover the optimal sequence via enumeration and ranking, because the total number of traversal paths increases exponentially with the number of nodes.

Confronted with the challenges mentioned above, this work makes the following contributions:

1) Considering higher-order semantics embedded in the structure of input graph, we propose a **L**ine **G**raph **E**nhanced encoder (LGESQL). It utilizes line graph [10] to update features of 1-hop relations layerwise. The node-centric (original graph) and edge-centric (line graph) graphs capture the topological structure and contextual semantics of nodes and edges respectively. Specifically, each "node" in either graph gathers information from its neighborhood and fuse "edge" features from the dual graph to update its representation. For the node-centric graph, we further combine both 1-hop and multi-hop relations in different manners. This distinction encourages the graph encoder to pay more attention to adjacent neighbors while maintaining information from remote nodes.
2) To eliminate permutation biases injected in the grammatical knowledge of SQL structure, we devise a **G**olden **T**ree-oriented **L**earning (GTL) algorithm. During training, the target SQL AST is constructed step-by-step guided by the annotated tree. It breaks away from the convention that the AST is tiresomely learned in a canonical order. Concretely, the GTL algorithm inspires the decoder to inject some randomness when selecting which child node to expand next and greedily determine the corresponding golden output action based on the history and current output distributions. This training scheme is also compatible with the beam search strategy during evaluation. To our knowledge, this is the first work to consider the traversing order of AST nodes in text-to-SQL.
3) On the whole, we propose a unified **H**eterogeneous **G**raph to **A**bstract **S**yntax **T**ree (HG2AST) framework for text-to-SQL from a fully structured perspective. It contains an LGESQL encoder (Section IV), an AST decoder with GTL algorithm (Section V-B), and two auxiliary modules (Section V-A) to connect them together. Experiments under different configurations, including single/multi-table, single/cross-domain, and multilingual, verify the effectiveness and universality.

II. Background and Related Work

A. Knowledge-Driven NLP and Text-to-SQL

Knowledge refers to information absent from the input text yet beneficial to attain the output [11], such as internal linguistic features and external structured knowledge graphs. Knowledge-driven NLP [12] has been a resurgent area since the advent of pre-trained language models (PLMs) [13], [14]. Various methods have been proposed catering to the prevailing encoder-decoder [15], [16] architecture: 1) Enrich input or intermediate language representation with specific information. For example, a topic-centric [17] or persona-aware [18] attention vector can be fused into computation to generate coherent or personalized dialogue responses. 2) Encode knowledge into the objective function to acquire desired behaviors. By devising self-supervised tasks such as concept order recovering [19] and entity category prediction [20], knowledge of concepts and entities can be implicitly injected into model parameters for language understanding. 3) Enlarge or constrain the search space during inference. To reduce hallucination in grounded language

generation [21], both copying [22] and retrieving [23] mechanisms can be adapted to choose chunks in the external memory and insert them at proper places in the output sequence [24].

Unlike other NLP tasks, text-to-SQL [2], [3] is characterized by its dependence on knowledge of well-defined structures, namely the *graph structure* of input database schema and the *grammatical structure* of output SQL program. Unlike unstructured documents where knowledge is serialized in the free-text format and knowledge graphs where triplets are stored in a schema-free manner [25], a relational database organizes data points following its rigidly-structured and domain-specific schema. In contrast to natural language generation [26], where minor noise is tolerable, the output side of text-to-SQL must strictly adhere to the syntactic constraints of SQL grammar, which can be described as a tree structure. In this work, 1) we systematically combine the graph encoder and the tree decoder and propose a unified framework that seamlessly integrates dedicated structure knowledge into deep learning models. 2) We further analyze the limitations on capturing higher-order semantics in the input heterogeneous graph and eliminating permutation biases in the output linearized tree.

B. Heterogeneous Graph Neural Network

A heterogeneous graph contains multiple types of nodes and edges. To address the heterogeneity of node attributes, Zhang et al. [4] designs a type-based content encoder, and Fu et al. [27] utilizes a type-specific linear transformation. As for different types of edges, both relational graph convolution network (RGCN) [28] and relational graph attention network (RGAT) [5] have been proposed to parametrize different relations. These two models only use static edge features without contextual or layerwise updates. On top of all that, Chen et al. [29], Zhu et al. [30] and Zhao et al. [31] construct the line graph of the original graph and explicitly model the computation over edge features.

Considering the distinct structure of input graph, state-of-the-art text-to-SQL models take advantage of heterogeneous graph neural networks [32]. To obtain more informative node features, SQLNet [33] proposes the "column attention" strategy to gather information from columns to each question word. TypeSQL [34] incorporates prior knowledge of column types as additional features. EditSQL [35] considers cross-attention between question words and database schema nodes similar to the common practice in text matching [36]. X-SQL [37], F-SQL [38], and BRIDGE [9] further leverage the database content to enrich the column representation. Regarding edge features, GNNSQL [39] and RATSQL [6] adapt the RGCN and RGAT models, respectively, to handle various pre-defined relations. From the perspective of graph structure, ShadowGNN [40] splits the original schema graph into two complementary parts: abstract and semantic schema graphs. SADGA [41] and S ^{2} SQL [42] further consider the syntactic structure in the linear input question. However, previous literature neglects the graph structure among edges and merely utilizes static edge features without higher-order semantics. In this work, we update node and edge features via dual iteration over the original and line graphs.

C. Constrained Decoding and Sequence-to-Set

Existing methods to tackle the constrained decoding problem for text-to-SQL can be classified into three categories: *token-based*, *module-based* and *grammar-based*. 1) A token-based decoder [9], [43] directly generates each token (including keywords like SELECT and JOIN) in the SQL autoregressively. This method severely relies on bulky generative pre-trained language models (PLMs) such as BART [44] and T5 [14]. Nonetheless, it still inevitably produces grammatically incorrect programs and requires syntax- or execution-guided decoding [45] to periodically eliminate invalid partial programs from the beam during inference. 2) A module- or sketch-based decoder [33], [34] devises tailored modules such as classifiers and pointer networks for different clauses. Connections among different modules need careful consideration. Consequently, this methodology merely applies to the single-table setting and fixed SQL sketches. Performances easily degenerate in the multi-table setting, where the JOIN operation of multiple tables, several interdependent clauses, and even nested sub-SQLs may appear. 3) As a compromise, a grammar-based decoder [8], [46], [47] predicts a sequence of actions (or grammar rules) to construct the equivalent SQL AST instead. Type constraints [46] can be easily incorporated into grammar rules. In this branch, both top-down (IRNet [8]) and bottom-up (SMBOP [48]) grammars have been applied to instruct the decoding.

In structured SQL prediction, countable equivalent sequences all lead to the same acceptable outcome. To address this *sequence-to-set* (Seq2Set) [49] or *permutation invariant* [50] problem, the three genres of decoders respond in different ways. 1) Token-based sequence-to-sequence models can be fine-tuned with reinforcement learning [51]. However, this scheme endures high variance and attains limited improvements. Executing SQL queries in the database for a single reward (Seq2SQL [2]) incurs additional I/O latency. 2) Module-based decoders can circumvent this problem by predicting each fine-grained goal in parallel [33], [52], [53]. For example, SQLNet [33] individually determines whether each column will appear as one condition in the WHERE clause. However, dependencies and interactions among those primitives are ignored. And they can hardly generalize to complicated SQL sketches. 3) The structured Seq2Set problem is less investigated in grammar-based decoders. Inspired by solutions in other fields, we can choose one specific ordering with the maximum score for each sample based on model predictions. Specifically, the likelihoods of all permutations are calculated if affordable [54], [55], [56]. Otherwise, a bipartite graph matching algorithm such as Hungarian algorithm [57] is applied to resolve the assignment [58]. However, neither scenario applies to the AST decoder due to the enormous traversal paths and rigid output structure. We focus on this branch and adapt it to the structured prediction setting in this work.

D. Basic Network Modules

This section introduces four basic modules. The first two are used in the encoder, while the latter two for decoder.

TABLE I
PART OF 1-HOP AND MULTI-HOP RELATIONS, SEE RATSQL [6] FOR THE COMPLETE CHECKLIST

Type of source node x_i	Type of target node x_j	Relation name	Description
Different Types of 1-hop relations Z_1			
QUESTION	QUESTION	DISTANCE+1	x_j is the word to the right of x_i.
COLUMN	COLUMN	FOREIGNKEY	x_j is the foreign key of x_i.
TABLE	COLUMN	HAS	The column x_j belongs to the table x_i.
QUESTION	TABLE	EXACTMATCH	Word x_i is part of table x_j, and x_j is a span of the entire question.
COLUMN	QUESTION	PARTIALMATCH	Word x_j is part of column x_i, but the entire question does not contain x_i.
Different Types of Multi-hop relations $Z_{>1}$ (usually derived with a rule of thumb)			
QUESTION	QUESTION	DISTANCE+2	x_j is the second word to the right of x_i.
COLUMN	COLUMN	SAMETABLE	Column x_i and x_j belong to the same table.
TABLE	TABLE	TABLEFOREIGNKEY	Table x_i has one column which is the foreign key of table x_j.
TABLE	COLUMN	GENERIC	Column x_j does not belong to table x_i.

Attentive Pooling: This module [59] compresses a matrix $\mathbf{X} = [\mathbf{x}_1; \cdots; \mathbf{x}_{|\mathrm{X}|}] \in \mathbb{R}^{|\mathrm{X}| \times d_x}$ with $|\mathrm{X}|$ rows[2] and d_x dimensions into one output vector $\tilde{\mathbf{x}} \in \mathbb{R}^{1 \times d_x}$ via attention-based pooling function:

$$\tilde{\mathbf{x}} = \mathrm{AttentivePooling}(\mathbf{X}). \tag{1}$$

Relational Graph Attention Network (RGAT): The Transformer [60] architecture can be regarded as a specific implementation of graph attention network (GAT) [61]. To incorporate the edge feature $\mathbf{z}_{ji}$ from node j to i, we revise the computation of attention weight α_{ji} and context vector $\tilde{\mathbf{x}}_i$ like Shaw et al. [62]:

$$\alpha_{ji} = \underset{j \in \mathcal{N}(i)}{\mathrm{softmax}} \left\{ (\mathbf{x}_i \mathbf{W}_Q)(\mathbf{x}_j \mathbf{W}_K + \mathbf{z}_{ji} \mathbf{W}_Z)^{\mathrm{T}} / \sqrt{d_x} \right\},$$

$$\tilde{\mathbf{x}}_i = \sum_{j \in \mathcal{N}(i)} \alpha_{ji} (\mathbf{x}_j \mathbf{W}_V + \mathbf{z}_{ji} \mathbf{W}_Z), \tag{2}$$

where $\mathcal{N}(i)$ is the neighborhood function for node i. Indices of head are omitted here. Let $\mathbf{X} \in \mathbb{R}^{|\mathrm{X}| \times d_x}$ and $\mathbf{Z} \in \mathbb{R}^{|\mathrm{Z}| \times d_z}$ denote matrices of stacked node and edge features respectively and G as the graph structure, one RGAT layer is formulated as:

$$\mathbf{X}' = \mathrm{RGAT}(\mathbf{X}, \mathbf{Z}, G).$$

Multi-Head Cross-Attention (MHCA): Given a query $\mathbf{x} \in \mathbb{R}^{1 \times d_x}$ and a memory matrix $\mathbf{M} = [\mathbf{m}_1; \cdots; \mathbf{m}_{|\mathrm{M}|}] \in \mathbb{R}^{|\mathrm{M}| \times d_m}$ with $|\mathrm{M}|$ rows, the attention vector $\tilde{\mathbf{m}} \in \mathbb{R}^{1 \times d_m}$ is computed by

$$\tilde{\mathbf{m}} = \mathrm{MultiHeadCrossAttention}(\mathbf{x}, \mathbf{M}).$$

Pointer Network: It is widely used in the decoder to copy raw tokens from the input memory [22]. Given a query vector $\mathbf{x} \in \mathbb{R}^{1 \times d_x}$, we re-use MHCA and take average of attention weights from different heads as the probability $p_{\mathrm{ptr}}(i)$ of choosing the ith entry in memory $\mathbf{M} \in \mathbb{R}^{|\mathrm{M}| \times d_m}$. Formally,

$$p_{\mathrm{ptr}}(i) = \mathrm{PointerNetwork}(\mathbf{x}, \mathbf{M})[i].$$

III. OVERVIEW

Given a user question $Q = (q_1, q_2, \ldots, q_{|Q|})$, the corresponding database schema $T \bigcup C = \{t_i\}_{i=1}^{|T|} \bigcup \{c_i\}_{i=1}^{|C|}$ (including multiple tables and columns), and various relations Z among these inputs, the goal of a text-to-SQL parser is to generate the SQL query y.

[2]Vectors throughout this paper are all row vectors.

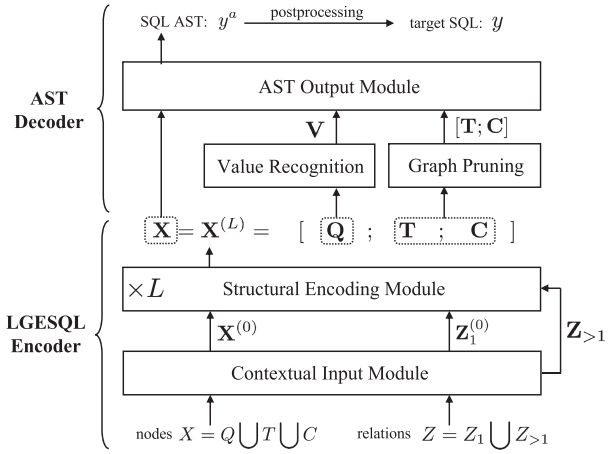

Fig. 2. An overview of the HG2AST framework.

It can be formalized as a graph-to-tree problem. Each question word q_i, table t_i and column c_i represents one node $x_i \in X = Q \bigcup T \bigcup C$ in the input graph. A truncated list of relation types between these input nodes is provided in Table I. The complete set of $Z = \{z_{ji}\}_{1 \le i,j \le |X|}$ can be split into two parts: 1-hop relations Z_1 (e.g., TABLE-HAS-COLUMN) and multi-hop relations $Z_{>1}$ (with hop number > 1, e.g., COLUMN-SAMETABLE-COLUMN). Nodes of question words q_i are integrated into the database schema graph through string-based matching schemes, e.g., QUESTION-EXACTMATCH-TABLE relation in Table I. As for the output, instead of producing the SQL query y token-by-token, we first constructs the equivalent tree form y^a through continuous actions $(a_1, \ldots, a_{|y^a|})$, and converts it into y via post-processing.

Our HG2AST framework is split into four parts (see Fig. 2): 1) a *contextual input* module, 2) a *structural encoding* module, 3) two auxiliary modules, and 4) an *AST output* module. Concretely,

a) First, the *contextual input* module initializes features for all nodes $X = Q \bigcup T \bigcup C$ and edges $Z = Z_1 \bigcup Z_{>1}$ (Section IV-A).
b) Next, the *structural encoding* module considers the structures of both the node-centric and edge-centric graphs, and updates features of nodes $\mathbf{X}^{(l)}$ and 1-hop edges $\mathbf{Z}_1^{(l)}$ layerwise (Section IV-B).

TABLE II
NOTATIONS USED IN THIS PAPER, PARTITIONED BY THE ENCODER AND DECODER

Notation	Definition
q_i, Q	a question word, the entire question
$\mathbf{q}_i, \mathbf{Q}$	vector of q_i, encoded question memory of Q
t_i, T	one table, the set of all tables
$\mathbf{t}_i, \mathbf{T}$	vector of t_i, encoded table memory of T
c_i, C	one column, the set of all columns
$\mathbf{c}_i, \mathbf{C}$	vector of c_i, encoded column memory of C
G^n, G^e	original node-centric graph, edge-centric line graph
x_i, X	one node in G^n (can be q_i, t_i or c_i), the set of all nodes
$\mathbf{x}_i, \mathbf{X}$	node embedding of x_i, node embedding matrix of X
$z_{ji}, \mathbf{z}_{ji}$	edge/relation and its feature from node x_j to node x_i
$Z_1, Z_{>1}$	the set of 1-hop and multi-hop relations respectively
$\mathbf{Z}_1, \mathbf{Z}_{>1}$	edge embedding matrices of Z_1 and $Z_{>1}$ respectively
v_i, V	a SQL value candidate, the set of all value candidates
$\mathbf{v}_i, \mathbf{V}$	vector of v_i, encoded value memory of V
$n_j, \mathbf{n}_j$	input AST node chosen at timestep j and its feature
$n_j^\tau, \psi(n_j^\tau)$	type of node n_j and type embedding function
$a_j, \mathbf{a}_j$	output action at timestep j and its action embedding
$r_j, \phi(r_j)$	grammar rule at timestep j and rule embedding function
y, y^a	the target SQL program, the equivalent SQL AST of y
y_j^a	the incomplete SQL AST obtained at timestep j

c) After getting encoded node features $\mathbf{X}$, we append two auxiliary modules, namely *value recognition* and *graph pruning*, to construct the value/table/column memories $\mathbf{V}/\mathbf{T}/\mathbf{C}$ for the AST decoder to retrieve or copy from (Section V-A).

d) Finally, the *AST output* module creates the SQL AST y^a, based on the encodings $\mathbf{X}$ and memory banks $\mathbf{V}/\mathbf{T}/\mathbf{C}$ (Section V-B).

In Table II, we summarize frequently used notations in this work for quick reference.

IV. HETEROGENEOUS GRAPH ENCODER

This section introduces the contextual input module and the structural encoding module of our **L**ine **G**raph **E**nhanced text-to-**SQL** (LGESQL) encoder.

A. Contextual Input Module

It aims to provide initial embeddings for nodes X and edges Z.

For nodes, features can be initialized from either static word vectors (SWV) such as GLOVE [63] and Tencent embeddings [64], or a pre-trained language model (PLM) such as BERT [13].

SWV: The question Q and each schema item $t_i \in T$, $c_j \in C$ can be initialized by looking up the embedding dictionary. To attain contextual information, these vectors are passed into three type-aware bidirectional LSTMs (BiLSTM [65]). Note that each table t_i or column c_j is described by its name and composed of several words. The forward and backward hiddens after BiLSTM are concatenated to obtain $\mathbf{x}_i^{(0)} \in \mathbb{R}^{1\times d_x}$. All node representations are stacked together and constitute the matrix $\mathbf{X}^{(0)} = [\mathbf{Q}^{(0)}; \mathbf{T}^{(0)}; \mathbf{C}^{(0)}]$.

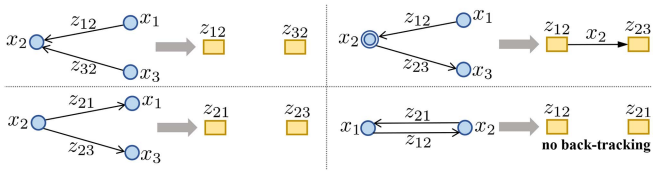

Fig. 3. Four primitives to construct G^e from G^n.

PLM: First, we flatten all question words and schema items into a sequence, where columns belonging to the same table are clustered together after the occurrence of their common table[3]: `[CLS]` $q_1 \cdots q_{|Q|}$ `[SEP]` t_1 $c_1 c_2 \cdots c_i$ t_2 $c_{i+1} c_{i+2} \cdots$ `[SEP]`.

Since each word may be tokenized into multiple subwords, we attach a subword attentive pooling layer after PLM, see (1), to obtain word-level representations. After that, we also feed them into three type-aware BiLSTMs and get the initial matrix $\mathbf{X}^{(0)}$.

For edges, both 1-hop relations $\mathbf{Z}_1^{(0)} \in \mathbb{R}^{|Z_1|\times d_z}$ and multi-hop relations $\mathbf{Z}_{>1} \in \mathbb{R}^{(|Z|-|Z_1|)\times d_z}$ are initialized from a trainable parameter matrix. Matrix $\mathbf{Z}_{>1}$ is static without layerwise update.

B. Structural Encoding Module

Before integrating structural knowledge into node features, we introduce how to construct the original node-centric graph $G^n = (\mathcal{V}^n, \mathcal{E}^n)$ and edge-centric line graph [10] $G^e = (\mathcal{V}^e, \mathcal{E}^e)$.

1) Construction of the Dual Graphs: Original Graph G^n: The node-centric graph $G^n = (\mathcal{V}^n, \mathcal{E}^n)$ is a complete graph containing all three types of nodes, that is $\mathcal{V}^n = X = Q \bigcup T \bigcup C$ and $\mathcal{E}^n \in \mathcal{V}^n \times \mathcal{V}^n$ is exactly the relation set $Z = Z_1 \bigcup Z_{>1}$ mentioned before.

Line Graph G^e: Each node in G^e can be uniquely mapped to a directed edge in G^n. Symmetrically, edges in G^e are uniquely identified by nodes in G^n. Fig. 3 illustrates how to construct the line graph through four primitives. For example, in the top right part, there is an edge $z_{12} \xrightarrow{x_2} z_{23}$ in G^e, if and only if in the node-centric graph G^n, the target node of edge z_{12} equals the source node of z_{23}. The edge between nodes z_{12} and z_{23} in G^e can be represented by the middle node x_2 in G^n. We also prevent back-tracking cases where two reverse edges will not be connected in G^e, as illustrated in the bottom right of Fig. 3.

Back to this work, we construct $G^e = (\mathcal{V}^e, \mathcal{E}^e)$ from G^n based on 4 primitives above. Note that, we only use 1-hop relations Z_1 as the node set $\mathcal{V}^e$ to avoid creating too many nodes in G^e.

The core component of LGESQL encoder is a stack of L dual RGAT layers. In each layer l, two RGATs capture the structure of the original graph G^n and line graph G^e, respectively. Node embeddings in one graph serve as edge features in another graph. For example, in Fig. 4, edge features with index 4 in G^e is provided by embeddings of node 4 in G^n.

[3]Following Suhr et al. [66], we randomly shuffle the order of tables and columns in different mini-batches to discourage over-fitting.

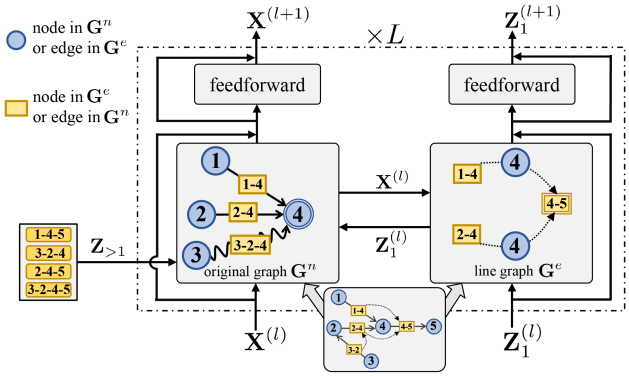

Fig. 4. Illustration of the structural encoding module in LGESQL encoder. We take node 4 and edge 4-5 as our main focuses.

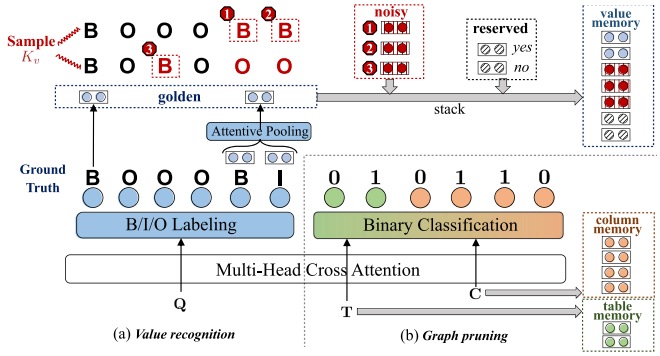

Fig. 5. Two auxiliary modules to construct memory banks $\mathbf{V}/\mathbf{T}/\mathbf{C}$.

Formally, the iterative computation is defined as follows:

$$\mathbf{X}^{(l+1)} = \text{RGAT}^n(\mathbf{X}^{(l)}, [\mathbf{Z}_1^{(l)}; \mathbf{Z}_{>1}], G^n),$$
$$\mathbf{Z}_1^{(l+1)} = \text{RGAT}^e(\mathbf{Z}_1^{(l)}, \mathbf{X}^{(l)}, G^e), \quad l = 0, 1, \ldots, L-1.$$

2) RGAT for the Original Graph: The node-centric graph G^n is a complete graph where any two nodes are connected. Since there are two genres of relations (1-hop and multi-hop), we design two schemes to integrate them.

Mixed Static and Dynamic Embeddings (MSDE) If the edge feature $\mathbf{z}_{ji}$ in (2) belongs to 1-hop relations Z_1, we use the dynamically updated node embedding $\mathbf{Z}_1^{(l)}$ provided by the line graph; otherwise, we always retrieve the static embedding from the parameter matrix $\mathbf{Z}_{>1}$.

Multi-head Multi-view Concatenation (MMC) An alternative is to split the muli-head attention module into two parts. In half of the heads, each node only has access to its neighbors within 1-hop, and the edge feature $\mathbf{z}_{ji}$ is provided by $\mathbf{Z}_1^{(l)}$ layerwise. In the other heads, the neighborhood function $\mathcal{N}(\cdot)$ in (2) for any node includes all nodes in $\mathcal{V}^n$. In this global view, we use static entries in matrix $[\mathbf{Z}_1^{(0)}; \mathbf{Z}_{>1}]$.

In either scheme, the RGAT^n module relatively manipulates 1-hop edge features $\mathbf{Z}_1^{(l)}$ with more caution.

3) RGAT for the Line Graph: For the counterpart RGAT^e, we make one modification: the edge-related vector $\mathbf{z}_{ji}\mathbf{W}_Z$ in (2) is added on the "query" side instead of the "key" side when computing attention weights α_{ji}. Because this feature is merely tied to the query node in the line graph and irrelevant to incoming edges. For example, in Fig. 4, the edges of two node pairs $(1\text{-}4, 4\text{-}5)$ and $(2\text{-}4, 4\text{-}5)$ in the line graph are the same one with index 4. Thus, the feature of edge 4 will make no contribution if it is added on the "key" side.

The output matrix for nodes after layer L is the desired output of the LGESQL encoder. We remove the superscript and further split it based on node types for later usage:

$$\mathbf{X} = \mathbf{X}^{(L)} = [\mathbf{Q}; \mathbf{T}; \mathbf{C}].$$

V. Abstract Syntax Tree Decoder

A. Auxiliary Modules

Before constructing the SQL AST y^a, we utilize two auxiliary modules, namely *value recognition* and *graph pruning*, to prepare the value, table, and column memories (see Fig. 5) which will be accessed by the AST output module (Section V-B). These two modules are trained with the main text-to-SQL task in a multi-tasking way.

1) Value Recognition: This module aims to construct the value memory $\mathbf{V} \in \mathbb{R}^{|V| \times d_x}$ for the AST decoder. Each row vector $\mathbf{v}_i \in \mathbf{V}$ represents a SQL value candidate, which may come from three sources:

1) Possible SQL value spans in the question are extracted via the traditional B/I/O sequence labeling scheme [67]. Given the ith encoded vector $\mathbf{q}_i \in \mathbf{Q}$, a 3-class classification is performed (labels are B, I, and O). Features from schema items are incorporated to enhance the alignments. Formally,

$$\tilde{\mathbf{s}}_i = \text{MultiHeadCrossAttention}_{\text{aux}}(\mathbf{q}_i, [\mathbf{T}; \mathbf{C}]),$$
$$P_{\text{vr}}(y_i^q | \mathbf{q}_i, \mathbf{T}, \mathbf{C}) = \text{softmax}(\mathbf{W}_{vr}[\mathbf{q}_i \circ \tilde{\mathbf{s}}_i]).$$

where $\circ$ is vector concatenation. Assume the ground truth label for question word q_i is g_i^q, the training objective can be formulated as

$$\mathcal{L}_{\text{vr}} = -\sum_{i=1}^{|Q|} \log P_{\text{vr}}(y_i^q = g_i^q | \mathbf{q}_i, \mathbf{T}, \mathbf{C}).$$

After sequence labeling, hidden states $[\mathbf{q}_k, \ldots, \mathbf{q}_{k+l-1}]$ of the mth value span starting from position k with length l are aggregated into one vector $\mathbf{v}_m \in \mathbb{R}^{1 \times d_x}$ in $\mathbf{V}$ via attentive pooling, (1):

$$\mathbf{v}_m = \text{AttentivePooling}_{\text{vr}}([\mathbf{q}_k; \cdots; \mathbf{q}_{k+l-1}]).$$

2) During training, ground truth labels g_i^q can be automatically detected from annotated (question, SQL) pairs by fuzzy string matching and heuristic rules such as word-number conversion. For inference, greedy decoding comes in handy. To bridge the training-inference gap, we sample K_v sequences during training based on model predictions. Noisy value candidates are extracted from sampled B-I-O sequences as interference terms $\mathbf{v}_m \in \mathbf{V}$ (the dashed square with title "noisy" in Fig. 5).

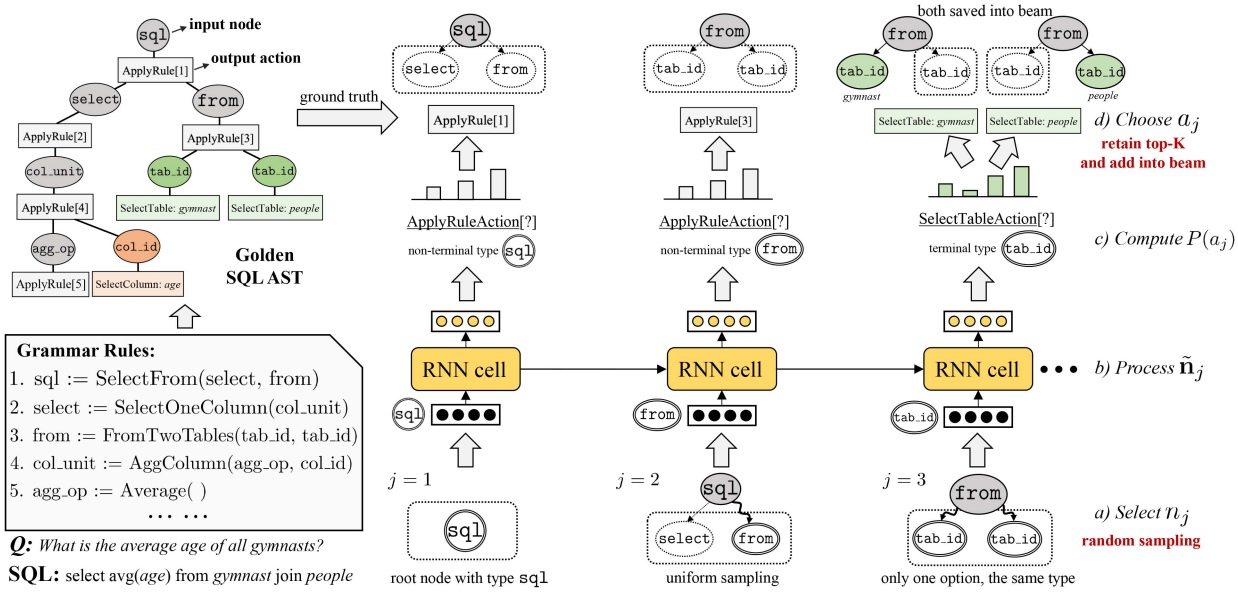

Fig. 6. Illustration of the AST output module and GTL algorithm (during training). In steps a) and d), a dashed square represents a frontier node set.

3) Separate parametric entries $\mathbf{v}_m \in \mathbb{R}^{1\times d_x}$ are reserved for common values such as "yes", which do not appear in the question.

SQL value candidates from the three sources above together constitute the complete value memory $\mathbf{V} \in \mathbb{R}^{|V|\times d_x}$ during training. After SQL generation, value nodes in the AST are further post-processed to deal with time normalization and morphological and numeric transformations based on column types.

2) Graph Pruning: We hypothesize that the model should be discerning to distinguish the question-oriented schema subgraph from the entire schema graph. To introduce this inductive bias, we propose graph pruning to perform a binary classification for each schema node.

Take column c_i as an example, we first compute the context vector $\tilde{\mathbf{q}}_i$ from question memory $\mathbf{Q}$ via MHCA. Next, a binary classifier with sigmoid unit $\sigma(\cdot)$ determines whether the compressed $\tilde{\mathbf{q}}_i$ and the encoded column representation $\mathbf{c}_i$ are correlated. Formally,

$$\tilde{\mathbf{q}}_i = \text{MultiHeadCrossAttention}_{\text{aux}}(\mathbf{c}_i, \mathbf{Q}),$$
$$P_{\text{gp}}(y_i^c|\mathbf{c}_i, \mathbf{Q}) = \sigma(\mathbf{W}_{gp}[\mathbf{c}_i \circ \tilde{\mathbf{q}}_i]).$$

The ground truth label g_i^c is 1 iff column c_i appears in the target SQL program y. And the training object for column nodes is

$$\mathcal{L}_{\text{gp}}^c = -\sum_{i=1}^{|C|}[g_i^c \log P_{\text{gp}}(y_i^c|\mathbf{c}_i, \mathbf{Q}) + (1-g_i^c)\log(1-P_{\text{gp}}(y_i^c|\mathbf{c}_i, \mathbf{Q}))].$$

The computation of $P_{\text{gp}}(y_i^t|\mathbf{t}_i, \mathbf{Q})$ and $\mathcal{L}_{\text{gp}}^t$ for tables can be easily inferred. We still use the complete table/column memories $\mathbf{T}/\mathbf{C}$ in AST generation rather than pruned ones to avoid error propagation.

B. AST Output Module

This section introduces the AST output module, which constructs the SQL AST y^a in a transition-based top-down manner given encodings $\mathbf{X} = [\mathbf{Q}; \mathbf{T}; \mathbf{C}]$ and value memory $\mathbf{V}$. The input of the decoder at each timestep is a typed node to expand, while the output is the corresponding action indicating how to extend it.

1) Basic Concepts for AST: To recap briefly, an abstract syntax tree (AST) is a tree representation of the source code.

Node Type: Each node n_j in the AST is assigned a *type* attribute n_j^τ representing its syntactic role. For example, in Fig. 6, one node with type `sql` indicates the root of a complete SQL program. Nodes can be classified into *non-terminal* and *terminal* nodes based on types. In this task, terminal types include `tab_id`, `col_id`, and `val_id`, which denote the index of table, column, and SQL value in their memories respectively. Embeddings of each node type n_j^τ is provided by function $\psi(n_j^\tau) \in \mathbb{R}^{1\times d_n}$.

Grammar Rule: Each grammar rule r_j takes the form of

$$\texttt{p\_type} := \texttt{RuleName}(\texttt{c\_type1}, \texttt{c\_type2}, \cdots),$$

where `p _ type` is the type of the parent non-terminal node to expand, while (`c _ type1`, $\cdots$) denote types of children nodes to attach (can be duplicated or empty). Only non-terminal types can appear on the left-hand side. Excerpted grammar rules are provided at the bottom left of Fig. 6. We directly retrieve the feature of one grammar rule r_j from an embedding function $\phi(r_j) \in \mathbb{R}^{1\times d_r}$.

2) The General Procedure of AST Generation: At each timestep j, the AST output module sequentially performs tasks below to construct the incomplete AST y_j^a (see Fig. 6):

a). *Select* one node n_j (called *frontier node*) to expand in the incomplete AST y_{j-1}^a from a frontier node set (Section V-C2).

b). *Process* the feature of frontier node n_j in the *neural space* and get the attention vector $\tilde{\mathbf{n}}_j$ (Section V-B3).
c). *Compute* the distribution of output action $P(a_j)$ based on the type n_j^τ of frontier node (Section V-B4).
d). *Choose* a valid action a_j based on $P(a_j)$ (and golden AST y^a if available) and update y_{j-1}^a in the *symbolic space* (Section V-C3).

These steps are performed recurrently until no frontier node can be found in y_j^a. In other words, all nodes have been expanded.

3) Input of the Decoder: Given the chosen frontier node n_j, the input of the decoder includes three parts: 1) previous action embedding $\mathbf{a}_{j-1} \in \mathbb{R}^{1\times d_r}$, defined later in Section V-B4, to inform the decoder of how to update AST y_{j-1}^a in the neural space, 2) previous hidden states $\mathbf{s}_{j-1} \in \mathbb{R}^{1\times d_s}$ as usual in the recurrent decoder, and 3) structural position embedding (SPE) $\mathbf{n}_j$ which depicts the position of n_j in the partial AST.

Structural Position Embedding (SPE): Typically, $\mathbf{n}_j$ is a concatenation of four vectors: 1) the type embedding $\psi(n_j^\tau) \in \mathbb{R}^{1\times d_n}$ of n_j, 2) the type embedding of its parent node $\psi(n_{p_j}^\tau) \in \mathbb{R}^{1\times d_n}$, where p_j denotes the timestep when the parent of n_j is expanded, 3) the action embedding $\mathbf{a}_{p_j} \in \mathbb{R}^{1\times d_r}$ used to expand its parent node n_{p_j}, and 4) the decoder hidden states $\mathbf{s}_{p_j} \in \mathbb{R}^{1\times d_s}$ when its parent serves as the frontier node. Mathematically,

$$\mathbf{n}_j = \psi(n_j^\tau) \circ \psi(n_{p_j}^\tau) \circ \mathbf{a}_{p_j} \circ \mathbf{s}_{p_j}. \tag{3}$$

Next, the decoder hidden states $\mathbf{s}_j \in \mathbb{R}^{1\times d_s}$ and the attention vector $\tilde{\mathbf{n}}_j \in \mathbb{R}^{1\times d_s}$ are computed via

$$\begin{aligned}
\mathbf{s}_j &= f_{\text{dec}}(\mathbf{a}_{j-1} \circ \mathbf{n}_j, \mathbf{s}_{j-1}),\\
\tilde{\mathbf{x}}_j &= \text{MultiHeadCrossAttention}_{\text{dec}}(\mathbf{s}_j, \mathbf{X}),\\
\tilde{\mathbf{n}}_j &= \text{FFN}(\mathbf{s}_j \circ \tilde{\mathbf{x}}_j),
\end{aligned}$$

where $f_{\text{dec}}(\cdot,\cdot)$ represents a single-layer unidirectional RNN-series network such as ONLSTM [68]. It is initialized by

$$\begin{aligned}
\mathbf{s}_0 &= \text{AttentivePooling}_{\text{dec}}(\mathbf{X}),\\
\mathbf{a}_0 &= \mathbf{0}, \quad \mathbf{n}_1 = \psi(n_1^\tau) \circ \mathbf{0} \circ \mathbf{a}_0 \circ \mathbf{s}_0,
\end{aligned}$$

where n_1 is always the root node with type `sql`.

4) Output of the Decoder: After obtaining $\tilde{\mathbf{n}}_j$, the distribution $P(a_j)$ of output actions is computed based on the type n_j^τ of frontier node. There are two categories of actions, namely APPLYRULE for non-terminals and SELECTITEM for terminals (ITEM$\in$ {TABLE, COLUMN, VALUE}).

APPLYRULE action:This action chooses one rule r_i from a constrained set of grammar rules determined by the non-terminal type n_j^τ. Specifically, the left-hand side of rule r_i must equal n_j^τ (node type constraint [46]). When applying this action to AST y_{j-1}^a in the symbolic space, we attach all children types on the right-hand side of r_i to the parent node n_j and mark these children nodes as unexpanded. Given $\tilde{\mathbf{n}}_j$, the distribution of APPLYRULE action is calculated by

$$\begin{aligned}
P(a_j &= \text{APPLYRULE}[r_i] | a_{<j}, \mathbf{X}, \mathbf{V})\\
&= \text{softmax}_i(\tilde{\mathbf{n}}_j \mathbf{W}_{\text{ar}} \phi(r_i)^{\text{T}}).
\end{aligned}$$

SELECTITEM action:Taking SELECTTABLE action as an example, the other two (SELECTCOLUMN and SELECTVALUE) can be easily inferred. To expand frontier node with terminal type `tab _ id`, we directly choose one table entry and attach it to the input terminal node. The probability of selecting the ith entry from the table memory $\mathbf{T}$ is computed by

$$\begin{aligned}
P(a_j &= \text{SELECTTABLE}[t_i] | a_{<j}, \mathbf{X}, \mathbf{V})\\
&= \text{PointerNetwork}_{\text{st}}(\tilde{\mathbf{n}}_j, \mathbf{T})[i].
\end{aligned}$$

Until now, the action embedding $\mathbf{a}_j \in \mathbb{R}^{1\times d_r}$ (as decoder input for the next and future timesteps) can be defined,

$$\mathbf{a}_j = \begin{cases} \text{Linear}_{\text{st}}(\mathbf{t}_j) & \text{if } a_j \in \text{SELECTTABLE} \\ \phi(r_j) & \text{if } a_j \in \text{APPLYRULE} \end{cases},$$

where $\mathbf{t}_j$ is the selected entry in table memory $\mathbf{T}$, and r_j is the grammar rule chosen at the current timestep j.

The prediction of the target AST y^a can be decoupled into a sequence of actions $(a_1, \ldots, a_{|y^a|})$. The training objectives for the text-to-SQL task and the entire model are

$$\begin{aligned}
\mathcal{L}_{\text{ast}} &= -\sum_{j=1}^{|a|} \log P(a_j | a_{<j}, \mathbf{X}, \mathbf{V}),\\
\mathcal{L}_{\text{total}} &= \mathcal{L}_{\text{vr}} + \mathcal{L}_{\text{gp}}^t + \mathcal{L}_{\text{gp}}^c + \mathcal{L}_{\text{ast}}.
\end{aligned}$$

C. Selection of Input Node and Output Action

What remains unsolved is how to select the frontier node and output action given multiple choices. The proposed **G**olden **T**ree-oriented **L**earning (GTL) algorithm is summarized in Algorithm.1.

1) Find the Frontier Node Set Via DFS: To restrict the frontier node set available at each timestep j, we follow the depth-first-search (DFS) order similar to previous work [69] because it conforms to the top-down characteristics of grammar rules. Concretely, at timestep $j = 1$, the frontier node set only contains one node with root type `sql`. Each time after applying action a_j to expand the frontier node n_j, we directly switch to the collection of its unexpanded children nodes. These children nodes constitute the frontier node set for timestep $j + 1$. If n_j is a node of terminal type or all children of n_j have been expanded, we backtrack to its parent node. And this backtracking keeps going until we find one parent node which contains at least one unexpanded children node. Naturally, these unexpanded children nodes form the new frontier node set for the next timestep.

2) Selection of Frontier Node: Previous work [47] stipulates the access order of children nodes for each grammar rule in advance. For example, the decoder always expands the node of type `select` first, followed by type `from`, when it encounters the rule `sql` := SelectFrom(`select`, `from`). We attempt to break this convention. During training, to select the frontier node n_j as input, we uniformly sample one from the frontier node set for each AST hypothesis (line 7 in Algorithm 1). For instance, in Fig. 6, we sample the node of type `from` at timestep $j = 2$. While at timestep $j = 3$, we only have one option since all nodes in the set are of the same type `tab _ id`. There is no difference

Algorithm 1: GTL Training Algorithm.

Input: Golden AST y^a; Encoder outputs $\mathbf{X}$, $\mathbf{V}$; Beam size K_g;
Output: The top-ranked AST in the final beam with its score;

1: initialize AST y_0^a with a single node of root type `sql`;
2: initialize beam $\mathcal{B}_0 = \{y_0^a\}$ with one hypothesis;
3: **for** $j = 1$ to $|y^a|$ **do**
4: $\mathcal{F}$, $\mathcal{A}$, $\mathcal{B}_j = \emptyset, \emptyset, \emptyset$;
5: **for** y_{j-1}^a in $\mathcal{B}_{j-1}$ **do** ▷ in the *symbolic space*
6: ▷ For inference, add all typed nodes n_j into $\mathcal{F}$;
7: uniformly sample node n_j in frontier node set of y_{j-1}^a;
8: $\mathcal{F} = \mathcal{F} \cup \{(y_{j-1}^a, n_j)\}$;
9: **end for**
10: **for** (y_{j-1}^a, n_j) in $\mathcal{F}$ **do** ▷ in the *neural space*
11: $\tilde{\mathbf{n}}_j \leftarrow$ ProcessInput$(y_{j-1}^a, n_j, \mathbf{X})$ (see Section V-B3);
12: compute $P(a_j|\mathbf{X}, \mathbf{V})$ based on type n_j^τ (see Section V-B4);
13: ▷ For inference, since no y^a, add all valid actions a_j into $\mathcal{A}$;
14: compare y_{j-1}^a with y^a and get all golden actions a_j;
15: add all feasible (y_{j-1}^a, n_j, a_j) triples into $\mathcal{A}$;
16: **end for**
17: ▷ score of y_j^a is sum of logprobs $P(a_{\le j})$
18: sort and retain top-K_g triples (y_{j-1}^a, n_j, a_j) in $\mathcal{A}$;
19: **for** (y_{j-1}^a, n_j, a_j) in $\mathcal{A}$ **do** ▷ in the *symbolic space*
20: $y_j^a \leftarrow$ UpdateAst(y_{j-1}^a, n_j, a_j);
21: update frontier node set for y_j^a via DFS (see Section V-C1);
22: $\mathcal{B}_j = \mathcal{B}_j \cup \{y_j^a\}$
23: **end for**
24: **end for**
25: **return**The top-ranked AST in $\mathcal{B}_{|y^a|}$ with its score

in the input SPE $\mathbf{n}_j$, see (3). As for inference, to enlarge the search space and avoid stochasticity caused by sampling, we enumerate all possible options of frontier node types into the beam (line 6 in Algorithm 1), which dive into different branches.

3) Choice of Output Action: During training, the golden AST y^a is available. By comparing the partial AST y_{j-1}^a with y^a at each timestep j, we can retrieve eligible actions based on the frontier node n_j (line 14 in Algorithm 1). For example, at timestep $j = 2$ in Fig. 6, the golden actions for nodes with different types (`select` or `from`) can be differentiated without ambiguity. While at timestep $j = 3$, two acceptable golden actions for terminal type `tab _ id` can appear in any order. Since it is infeasible to enumerate and rank all paths, we construct a beam of K_g hypotheses in the history and only preserve the top-K_g permutations according to model predictions $P(a_{\le j})$ (line 18 in Algorithm 1). Once finished (hypotheses should terminate simultaneously), the model is only optimized upon the action sequence with the highest score. For inference, we adopt beam search but add action a_j (grammar rules or memory entries) into beam (line 13 in Algorithm 1) iff it satisfies the node type constraint.

In general, the GTL algorithm randomly selects one node type from the frontier node set at each timestep to avoid overfitting the permutation bias, and greedily chooses a golden action sequence with relatively higher score in the beam during training. Other combinations of selecting nodes and actions are analyzed in Section VI-E.

VI. Experiment

In this section, we evaluate our HG2AST framework in different experimental configurations. Codes will be publicly available[4].

A. Evaluation Metrics

Exact Set Match without Values (EM): This metric measures the equivalence of two SQL queries by comparing each component. The prediction is correct only if each fine-grained clause or unit is correct. Order issues and SQL values will be ignored.

Exact Set Match with Values (EMV): Based on EM, this metric further inspects the correctness of SQL values without execution. It is also known as *logical form accuracy*. It may be too strict in that semantically equivalent SQLs may have different sketches.

Execution Accuracy (EX): It measures the accuracy by comparing the execution results instead. However, this criterion suffers from *spurious programs*, where erroneous SQLs happen to obtain the same execution results as the golden one on a particular database.

Execution Accuracy with Testsuite (EXT): To alleviate the problem of spurious programs, Zhong et al. [70] proposed test suite accuracy. The predicted SQL is correct iff the execution results are equivalent to the ground truth on multiple databases.

B. Datasets

We choose eight typical text-to-SQL benchmarks, including 3 cross-domain multi-table datasets (Spider [3] [5], DuSQL [71] [6], and CSpider [72][7]), 2 cross-domain single-table datasets (WikiSQL [2] and NL2SQL [73]), and 3 single-domain multi-table datasets (ATIS [74], GEO [75] and Scholar [76]). In cross-domain settings, the databases of training, validation, and test sets do not overlap, while the same database is used across all data splits under single-domain settings. Specifically, CSpider is a dataset with multilingual inputs, where the question is Chinese, and the database schema is English. Notice that the test datasets for Spider and CSpider are not publicly available, and we submit models to the challenge organizers for evaluation (public availability of both testset input and output are ✗). Besides, participants only have access to the input of test samples for DuSQL and NL2SQL and need to submit the predicted SQLs to the website for results (public availability of testset output is ✗). For a fair comparison, we do not utilize any reranking, data augmentation, and ensemble methods throughout all experiments. The official

[4]Code link: https://github.com/rhythmcao/text2sql-graph2tree.
[5]Leaderboard for Spider: https://yale-lily.github.io//spider.
[6]Leaderboard for DuSQL and NL2SQL (a.k.a., TableQA): https://aistudio.baidu.com/aistudio/competition/detail/47/0/leaderboard.
[7]Leaderboard for CSpider: https://taolusi.github.io/CSpider-explorer/.

TABLE III
STATISTICS OF ALL BENCHMARKS

Dataset	Cross-domain Multi-table			Cross-domain Single-table		Single-domain Multi-table		
	Spider	DuSQL	CSpider	WikiSQL	NL2SQL	ATIS	GEO	Scholar
Language	EN	ZH	ZH+EN	EN	ZH	EN	EN	EN
# samples	10181	28762	9691	80654	54059	5280	877	817
# DBs	200	200	166	24241	5291	1	1	1
AVG # question nodes per sample	13.4	18.5	11.7	13.4	14.5	10.7	7.5	7.5
AVG # table nodes per sample	6.2	4.3	6.2	1	1	25	7	12
AVG # column nodes per sample	29.6	25.3	29.6	6.4	8.9	132	30	29
Visibility of testset input	✗	✓	✗	✓	✓	✓	✓	✓
AVG # actions/AST nodes per sample	21.6	23.5	17.3	9.4	12.5	66.5	21.5	33.7
Public availability of testset output	✗	✗	✗	✓	✗	✓	✓	✓
Official evaluation metrics	EM, EXT	EM, EMV	EM	EMV, EX	EMV	EMV	EMV	EMV

EN: English; ZH: Chinese; DB: Database

evaluation metrics for each dataset are reported. Statistics are detailed in Table III.

C. Experiment Setup

We tokenize questions, table names, and column names with toolkit Stanza [77] and LAC [78] for English and Chinese, respectively. Our model is implemented with Pytorch [79], and the original and line graphs are constructed with DGL [80] library. For static word vectors (SWV), features of the most frequent 100 words in the training set will be fine-tuned while the remaining are fixed. PLM and its initial checkpoint are downloaded from the transformers [81] library. The set of relations is borrowed from RATSQL [6]. For training, the sampling size K_v for noisy value candidates is 5, and beam size K_g for the GTL algorithm is 4. During inference, we adopt beam search decoding with size 5. In the encoder, the GNN hidden size for nodes (d_x) and edges (d_z) are set to $256/32$ for SWV and $512/64$ for PLM respectively. The number of GNN layers L is 8. In the decoder, the dimension of hidden state (d_s), action/rule embedding (d_r) and node type embedding (d_n) are set to 512, 128 and 128 respectively. The recurrent dropout rate [82] is 0.2 for decoder LSTM. The number of heads in multi-head attention is 8, and the dropout rate of features is 0.2 in both the encoder and decoder. Throughout the experiments, we use AdamW [83] optimizer with a linear warmup scheduler. The warmup ratio of total training steps is 0.1. For SWV, the learning rate is $5e$-4 and the weight decay coefficient is $1e$-4; For PLM, we use smaller leaning rate $4e$-4 (`small`), $2e$-4 (`base`) or $1e$-4 (`large`), and larger weight decay rate 0.1. The optimization for PLM encoders is carried out more carefully with learning rate layerwise decay (coefficient 0.8). Batch size is 20, and the maximum gradient norm is 5. The number of training epochs is 100 for SWV and 200 for PLM.

There are some essential text-to-SQL baselines classified by the category of decoders (summarized in Fig. 7):

Token-based: This category directly generates each token in the SQL query sequentially. 1) BRIDGE [9]: it further incorporates database content to enhance the representation of schema items. 2) SeaD [84]: it trains a sequence-to-sequence (Seq2Seq) model with schema-aware denoising objectives. 3) PICARD [43]: it utilizes a syntax-guided decoding strategy to eliminate invalid programs from the beam during inference. 4) UNIFIEDSKG [85] and UNISAR [86]: they formulate text-to-SQL as a traditional Seq2Seq problem and design a particular input prompt.

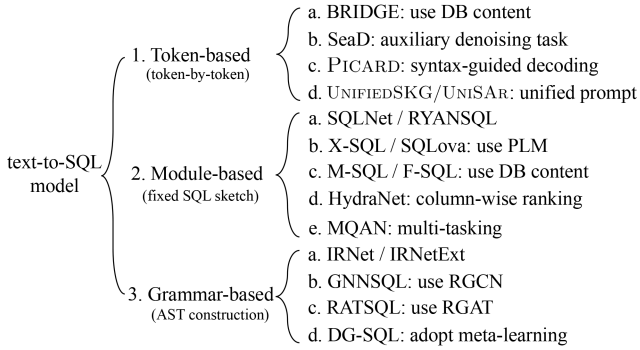

Fig. 7. Different baselines categorized by decoder types.

Module-based: Researchers design tailored modules for each clause in the fixed SQL sketch in this branch. 1) SQLNet [33] and RYANSQL [87]: they are classic module-based approaches applicable in single- and multi-table cases, respectively. 2) X-SQL [37] and SQLova [88]: they improve the contextual embeddings with a PLM. 3) M-SQL [89] and F-SQL [38]: based on X-SQL, they leverage database content to enhance the schema representation. 4) HydraNet [90]: it uses column-wise ranking and decoding strategies. 5) MQAN [91]: it further experiments in multi-tasking settings.

Grammar-based: Methods in this genre construct an AST of the target SQL instead. 1) IRNet [8]: it adopts a grammar-based decoder while neglecting relations in the input graph. 2) IRNetExt [71]: based on IRNet, it generates SQL values to obtain executable SQLs. 3) GNNSQL [39]: it employs an RGCN to tackle relations among database schema. 4) RATSQL [6]: it is the most prevailing method that adopts a fully connected RGAT and achieves competitive performances in many benchmarks. 5) DG-SQL [92]: it applies meta-learning to improve cross-domain generalization. Our HG2AST framework also belongs to this category.

D. Main Results

1) Main Results for Cross-Domain Multi-Table Datasets: Performances for cross-domain multi-table benchmarks are

TABLE IV
MAIN RESULTS FOR ENGLISH SPIDER

Category	Model	PLM	EM		EXT	
			Dev	Test	Dev	Test
With Static Word Vectors						
module-based	SQLNet [33]	w/o PLM	10.9	12.4	-	-
token-based	Seq2Seq [16]	w/o PLM	1.8	4.8	-	-
	EditSQL [35]		36.4	32.9	-	-
grammar-based	GNNSQL [39]	w/o PLM	52.7	47.4	-	-
	IRNet [8]		53.2	46.7	-	-
	RATSQL [6]		62.7	57.2	-	-
grammar-based (ours)	LGESQL†	w/o PLM	67.6	62.8	-	-
	LGESQL		67.1	-	65.3	-
	LGESQL+GTL		**70.4**	**65.3**	**69.4**	**64.2**
With Pre-trained Language Model: BERT-Large						
module-based	RYANSQL [87]	BERT	66.6	58.2	-	-
token-based	EditSQL	BERT	57.6	53.4	-	-
	BRIDGE [9]		70.0	65.0	68.0	64.3
grammar-based	IRNet	BERT	61.9	54.7	-	-
	RATSQL		69.7	65.6	-	-
grammar-based (ours)	LGESQL†	BERT	74.1	68.3	-	-
	LGESQL		71.9	-	72.2	-
	LGESQL+GTL		**74.3**	**69.4**	**74.0**	**66.2**
With Task Adaptive Pre-trained Language Models						
token-based	UNIFIEDSKG [85]	T5-Large	66.6	-	-	-
	PICARD [43]	T5-Large	69.1	-	72.9	-
	PICARD	T5-3B [14]	75.5	71.9	**79.3**	**75.1**
grammar-based	RATSQL	GAP [93]	71.8	69.7	-	-
	NatSQL [94]	GAP	73.7	68.7	75.0	73.3
	RATSQL	GRAPPA [95]	73.4	69.6	-	-
	SMBOP [48]	GRAPPA	74.7	69.5	75.0	71.1
	S^2SQL [42]	ELECTRA [96]	76.4	72.1	-	-
grammar-based (ours)	LGESQL†	ELECTRA	75.1	72.0	-	-
	LGESQL		74.6	-	75.1	-
	LGESQL+GTL		**76.8**	**72.3**	78.1	71.4

† Refers to the published version of LGESQL which does not generate SQL values.

TABLE V
MAIN RESULTS FOR CHINESE DUSQL

Category	Model	PLM	EM		EMV	
			Dev	Test	Dev	Test
With Static Word Vectors						
token-based	Seq2Seq [16]	w/o PLM	6.6	3.9	2.6	1.9
grammar-based	SyntaxSQLNet [97]	w/o PLM	14.6	8.6	7.1	5.2
	IRNet [8]		38.4	34.2	18.4	15.4
	IRNetExt [71]		59.8	54.3	56.2	50.1
grammar-based (ours)	RGATSQL	w/o PLM	78.2	66.7	77.3	65.4
	RGATSQL+GTL		79.9	70.0	78.8	68.3
	LGESQL		80.6	69.2	79.6	67.7
	LGESQL+GTL		**80.8**	**70.3**	**79.9**	**69.0**
With Pre-trained Language Models						
token-based	UNISAR [86]	BART [44]	84.3	-	-	-
grammar-based	IRNetExt	BERT	-	-	-	53.7
	IRNetExt+DataAug [98]		-	-	-	60.5
grammar-based (ours)	RGATSQL	BERT	82.5	72.8	81.4	71.0
	RGATSQL+GTL		83.5	74.0	**83.0**	72.6
	LGESQL		83.8	73.7	**83.0**	71.7
	LGESQL+GTL		**83.9**	**74.9**	82.8	**73.0**
	LGESQL	MacBERT [99]	87.3	77.1	86.7	75.5
	LGESQL+GTL		**88.8**	**78.9**	**88.1**	**77.4**

RGATSQL is our baseline system where the line graph is not utilized. It is a variant of RATSQL [6].

TABLE VI
MAIN RESULTS FOR MULTILINGUAL CSPIDER, WHERE THE QUESTION IS CHINESE AND THE SCHEMA IS ENGLISH

Category	Model	PLM	Dev EM	Test EM
With Multilingual Inputs				
module-based	RYANSQL [87]	mBERT	41.3	34.7
grammar-based	RATSQL [6]	mBERT	41.4	37.3
	DG-SQL [92]		50.4	46.9
	XL-SQL‡		54.9	47.8
grammar-based (ours)	RGATSQL	mBERT	56.9	-
	RGATSQL+GTL		58.2	-
	LGESQL		57.8	-
	LGESQL+GTL		**58.6**	**52.7**
	LGESQL+GTL	XLM-ROBERTA [100]	60.3	-
		INFOXLM [101]	**61.0**	**57.0**
With Question Translation				
grammar-based	RATSQL+Adv‡	GRAPPA [95]	59.7	56.2
	LGESQL	ELECTRA [96]	**64.5**	58.1
grammar-based (ours)	LGESQL+GTL	ELECTRA	64.0	**60.3**

‡ Means unpublished work on the leaderboard.

given in Tables IV–VI. The complete HG2AST framework combines the LGESQL encoder and the AST decoder with GTL algorithm (LGESQL+GTL). For non-English benchmarks without the grammar-based baseline RATSQL [6], we implement the variant RGATSQL with two auxiliary modules and self-curated grammar rules. From the results, some conclusions can be summarized:

1) Our HG2AST framework ranks in the top tier on all three benchmarks, in metrics both w/ and w/o SQL values. On English Spider, the performance with PLM ELECTRA has been at the top level on the authoritative leaderboard for more than one year in metric EM. On Chinese DuSQL, the pure LGESQL+GTL model outperforms the previous best-published work IRNetExt+data augmentation [98] by 12.5 points using the same PLM BERT. As for multilingual CSpider, we achieve state-of-the-art results on the leaderboard in different partitions (e.g., with mBERT 52.7).
2) Among models on the same scale, grammar-based decoders always perform the best in multi-table settings, while the module-based approach presents the worst performance. And the smaller the model size is, the superior the grammar-based method is. For example, in Table IV, our grammar-based decoder LGESQL+GTL with GLOVE defeats the token-based method EditSQL [35] by a large margin. Equipped with larger PLMs, this advantage is consistent compared to BRIDGE [9] with BERT and PICARD [43] with T5-Large [14]. This phenomenon can be explained by the complicated SQL sketch and strict output structure.
3) Among models in the same grammar-based category, our HG2AST framework also achieves the best performance across different datasets. In Table IV, HG2AST+GLOVE is competitive with previous work RATSQL+BERT (65.3 compared to 65.6 in test EM). Notably, the model size of HG2AST with static word vectors is less than 4% of BERT. Moreover, results are at the same level (roughly 69.5) when comparing HG2AST+BERT with RATSQL+task-adaptive PLMs on Spider. Besides, each component (LGESQL encoder and GTL algorithm) contributes to the eventual performance gains. When considering SQL values on Spider and DuSQL, which incurs extra difficulties, the metric EXT or EMV declines by 0-2 points. There is still a long way from the top-ranked method PICARD+T5-3B regarding metric EXT on the test set of Spider (71.4 v.s. 75.1). This divergence may be caused by our skewed post-processing tricks (further discussed in Section VII) and the considerable discrepancy in model size.
4) Equipped with more powerful PLMs, performances of HG2AST can be further promoted stably. In other words, the benefits of the structural bias are compatible with those introduced by more powerful contextual input modules. For instance, the EMV accuracy on the test set of DuSQL (Table V) increases from 73.0 to 77.4 when replacing BERT with MacBERT [99]. Furthermore, we speculate those task-adaptive PLMs which focus on

TABLE VII
MAIN RESULTS FOR ENGLISH WIKISQL (SINGLE-TABLE) WITHOUT EXECUTION-GUIDED DECODING

Category	Model	PLM	EMV		EX	
			Dev	Test	Dev	Test
With Static Word Vectors						
module-based	SQLNet [33]	w/o PLM	63.2	61.3	69.8	68.0
	SQLova [88]		65.8	64.6	72.9	71.7
	MQAN [91]		76.1	75.4	82.0	81.4
token-based	Ann. Seq2Seq [102]	w/o PLM	72.1	72.1	82.1	82.2
grammar-based	RATSQL [6]	w/o PLM	73.6	73.3	79.5	78.8
grammar-based (ours)	LGESQL	w/o PLM	77.1	76.7	82.9	82.6
	LGESQL+GTL		77.7	77.2	83.6	83.2
	LGESQL+GTL$^\diamond$		**82.7**	**82.8**	**89.4**	**89.2**
With Pre-trained Language Models						
module-based	SQLova	BERT	81.6	80.7	87.2	86.2
	HydraNet [90]		83.5	83.4	88.9	88.6
	F-SQL$^\diamond$ [38]		-	85.6	-	91.4
	X-SQL [37]	MTDNN [103]	83.8	83.3	89.5	88.7
	HydraNet	RoBERTa [104]	83.6	83.8	89.1	89.2
token-based	BRIDGE$^\diamond$ [9]	BERT	86.2	85.7	91.7	91.1
	UNISAR$^\diamond$ [86]	BART [44]	**86.7**	85.8	91.7	91.4
	SeaD [84]		84.9	84.7	90.2	90.1
grammar-based (ours)	LGESQL	BERT	83.6	83.3	89.6	89.1
	LGESQL+GTL		83.8	83.7	89.7	89.3
	LGESQL+GTL$^\diamond$		85.8	**86.3**	**92.0**	**92.1**

$^\diamond$Means database content is used.

TABLE VIII
MAIN RESULTS FOR CHINESE NL2SQL (SINGLE-TABLE)

Category	Model	PLM	Dev EMV	Test EMV
With Static Word Vectors				
module-based	SQLNet [33]	w/o PLM	61.3	61.4
	MQAN [91]		75.7	74.8
grammar-based (ours)	RGATSQL	w/o PLM	83.9	84.6
	RGATSQL+GTL		84.4	85.2
	LGESQL		84.5	85.1
	LGESQL+GTL		**85.2**	**85.4**
With Pre-trained Language Models				
module-based	SQLova [88]	BERT	81.4	81.7
	X-SQL [37]		82.9	83.3
	M-SQL [89]		89.1	89.3
	F-SQL [38]		-	**90.4**
token-based	Seq2Seq [16]	BART [44]	88.7	90.7
	UNISAR [86]		89.9	**91.8**
grammar-based (ours)	RGATSQL	BERT	89.3	89.7
	RGATSQL+GTL		89.4	90.0
	LGESQL		89.4	89.7
	LGESQL+GTL		90.1	**90.4**
	LGESQL	MacBERT [99]	90.4	90.6
	LGESQL+GTL		**91.0**	**91.8**

RGATSQL is a variant of RATSQL [6]. All models below leverage database content by default.

TABLE IX
MAIN RESULTS FOR SINGLE-DOMAIN MULTI-TABLE DATASETS ON THE TESTSET

Method and Dataset		ATIS	GEO	Scholar
module-based	Template [74]	46	57	52
token-based	Seq2Seq [16]	8	27	19
	+Attention [105]	46	63	33
	+Copy [22]	51	71	59
	+DataAug [76]	45	66	44
grammar-based	SEQ2TREE [52]	46	62	44
grammar-based (ours)	RGATSQL	54.8	69.5	67.4
	RGATSQL+GTL	56.0	71.3	71.6
	LGESQL	55.3	71.0	69.8
	LGESQL+GTL	**56.4**	**72.0**	**72.6**

For fair comparison, we use static word vectors glove.

enhancing the discriminative capability and improving cross-segment alignments are more suitable for the text-to-SQL task. The evidence is that PLMs such as GRAPPA [95], ELECTRA [96], MacBERT, and INFOXLM [101] all design pre-training tasks like syntactic role classification, word replacement discrimination, and contrastive learning between cross-lingual texts.

5) Turning to multilingual dataset CSpider (Table VI), we try two approaches depending on whether translating the question into English with an off-the-shelf translator such as EasyNMT[8]. With multilingual inputs and XLM-series encoders, the most advanced models still struggle with cross-lingual semantic alignments. The best performance (57.0%) is still lower than that with monolingual inputs by 3.3 points. This gap poses challenges for future work.

2) Main Results for Cross-Domain Single-Table Datasets: Next, we switch to single-table cases where each database only contains one table (Table VII–VIII). The output SQL sketch is much simpler. It is worth mentioning that different methods vary in post-processing tricks which remarkably influence the performance. Thus, we only compare results without execution-guided decoding [45]. From the results, we can summarize that:

1) Performances of the other two categories start to catch up with that of the grammar-based method. The token-based method benefits from large quantities of labeled data and simplified output structures in the single-table setting. Coupled with a powerful PLM, the module-based approach can circumvent the design of intertwined connections between different modules and predict each slot in the sketch in isolation.

2) Although HG2AST may be over-qualified in the single-table setting, it still achieves competitive performances in both datasets. On benchmark NL2SQL in the PLM partition (see Table VIII), the complete LGESQL+GTL ties for the first place. However, the LGESQL encoder demonstrates little advantage over the RGATSQL encoder which does not utilize the line graph. It can be attributed to the fact that only one table exists and the database schema is more like a column set than a graph. Nevertheless, the GTL training algorithm still improves upon strong baselines (from 90.6 to 91.8 with MacBERT). The reason is that even in single-table cases, multiple columns and conditions may be required in the SELECT and WHERE clauses, respectively. These selected items and filtering conditions can each constitute a set.

3) Previous work on WikiSQL seldom mentions whether the database content is used. In Table VII, we notice that concatenating candidate cells after each column (BRIDGE [9]) will dramatically boost the performance to another level (6 points increase with GLOVE and 3 points with BERT). We conjecture that in the single-table setting, deep semantic understanding is more significant than extrapolation of the SQL structure.

3) Main Results for Single-Domain Multi-Table Datasets: In this part, we evalute HG2AST on three more realistic single-domain multi-table datasets[9]. These datasets are collected from real scenarios and contain relatively more schema items and larger output ASTs on average (see statistics in Table III). According to results in Table IX, we can safely summarize that:

1) Even though the same database schema is repeatedly used across all training samples in single-domain settings, the

[8]Tool link: https://github.com/UKPLab/EasyNMT.git.

[9]Datasets repository: https://github.com/jkkummerfeld/text2sql-data.

TABLE X
ABLATION OF THE ENCODER

Encoder	Method	Spider EM	Spider EXT	DuSQL EM	DuSQL EMV
RGATSQL	w/o $Z_{>1}$	65.9	65.3	82.8	81.7
	w/ SE	67.6	68.0	84.8	83.6
	w/ MMC	67.8	67.2	85.6	84.4
LGESQL	w/o $Z_{>1}$	68.1	67.2	84.1	82.9
	w/ MSDE	69.1	**68.8**	85.1	84.0
	w/ MMC	**69.2**	68.6	**85.8**	**84.7**

SE: Static embeddings; MSDE:Mixed static and dynamic embeddings; MMC: Multi-head multi-view concatenation.

TABLE XI
ABLATION OF ORDER METHODS

row	Order Method	Spider EM	Spider EXT	DuSQL EM	DuSQL EMV
1	FN-C+GA-C	67.1	65.3	83.6	82.4
2	FN-C+GA-R	66.5	66.3	84.6	83.7
3	FN-C+GA-E	67.7	67.7	84.4	83.6
4	FN-R+GA-C	68.3	67.6	83.0	81.9
5	FN-R+GA-R	68.2	66.8	84.9	84.0
6	**FN-R+GA-E**	**69.2**	**68.6**	**85.8**	**84.7**
7	FN-E+GA-C	45.9	45.8	44.7	43.5
8	FN-E+GA-R	50.0	49.4	47.0	44.8
9	FN-E+GA-E	47.4	47.9	52.4	50.8

FV: Frontier node; GA: Golden action; C: Canonical order; R: Random sampling; E: Enumeration into beam. For methods FN-C, during inference, we follow the pre-defined priority of children types for each rule instead of enumeration (line 6 in algorithm 1).

TABLE XII
ABLATION OF AUXILIARY MODULES. NOTE THAT "W/O VALUE RECOGNITION" DENOTES USING A POINTER-GENERATOR MODULE FOR SQL VALUE GENERATION

Auxiliary Modules	Spider EM	Spider EXT	DuSQL EM	DuSQL EMV
complete model	**69.2**	**68.6**	**85.8**	**84.7**
w/o graph pruning	66.8	66.4	84.9	83.8
w/ pruned memories $\mathbf{T}/\mathbf{C}$	68.1	66.5	84.0	83.2
w/o value recognition	67.9	67.4	84.5	83.5
w/o noisy value sampling	69.0	67.3	85.1	83.4

Note that "w/o value recognition" denotes using a pointer-generator module for SQLvalue generation.

more advanced LGESQL encoder still facilitates the eventual performances compared to the classic graph encoder. Besides, it is worth noting that the explicit encoding of schema items strikingly defeats baselines such as SEQ2TREE [52] which does not consider the structures and relations of database schema.

2) The proposed order-insensitive GTL algorithm steadily promotes the testset accuracy on all three benchmarks. Owing to more complicated output structures (e.g., some SQL ASTs in ATIS refer to more than 20 tables in the FROM clause), the decoding order will potentially affect the model performance. While it is infeasible to discover the optimal permutation before training, GTL algorithm can instruct the model to explore it via bootstrapping.

E. Ablation Studies

In this section, we analyze the effect of each module in HG2AST. For the sake of time and memory space, the following experiments are conducted with GLOVE on Spider and small-series PLM ELECTRA on DuSQL. We report the average accuracy on the validation dataset with 3 random seeds.

From Table X, we can discover that: 1) if multi-hop relations are removed (w/o $Z_{>1}$), the performance will decrease roughly by one point in LGESQL, while two or more points drop for RGATSQL. However, our LGESQL with merely 1-hop relations is still competitive, even slightly better than RGATSQL with multi-hop relations on Spider (68.1 compared with 67.8 in metric EM). It consolidates our motivation that by exploiting the structure among edges, the line graph can capture long-range relations to some extent. 2) Two strategies of combining 1-hop and multi-hop relations proposed in Section IV-B2 (w/ MSDE or MMC) are both beneficial to the eventual performances of LGESQL (roughly 1% and 1.5% gains in metrics EM and EXT/EMV, respectively). It corroborates our assumption that 1-hop and multi-hop relations should be treated with distinction.

Next, we focus on the decoding order. Recap that different from previous grammar-based methods, which defines a canonical (-C) traversing order over the AST before training, our GTL algorithm randomly (-R) samples one input node to expand from a typed set at each timestep and enumerates (-E) all feasible golden actions into the beam given multiple choices (the current frontier node set contains several nodes of the same type). Other combinations of order methods are listed in Table XI. Row 1 is the traditional practice, and row 5 is equivalent to randomly sampling a traversing order over the AST via DFS per training sample in each iteration. From the overall results, we can see that: 1) By comparing rows 4-6 with 1-3 one-by-one, injecting some randomness into the choice of frontier nodes will improve the final performance (FN-R > FN-C). Intuitively, this practice prevents the model from over-fitting the canonical order of how to expand children nodes. 2) Among each bin (e.g., rows 4-6), the method -E for golden action (GA-E) usually ranks at the top. This observation complies with the discovery by Vinyals et al. [49] that there exists a particular order which optimally interprets the data when learning the underlying model. 3) However, the expected performance severely deteriorates if we enumerate all possible frontier nodes into the beam during training (rows 7-9). Through the lenses of changes in the top-scored traversing path, we find that with FN-E methods, each training sample sticks to one customized order of children types for each rule. For instance, one training sample prefers to first expand node `from` for the rule `sql_unit` := SelectFrom(`select`, `from`), while another one may prioritize type `select`. However, they seldom achieve a consensus and become stubborn to their insistence, refusing to attempt another traversing order. This interplay leads to contradictions, and the model can hardly converge. While for the FN-R methods, after being exposed to multiple permutations through mandatory random sampling, each sample is sensible and insightful to pick their preference.

We also disentangle auxiliary modules from the entire framework and report the resultant changes in Table XII. To generate SQL values w/o value recognition, we incorporate a traditional pointer-generator module [22] for terminal type `val_id` into the decoder network instead. The pointer-generator needs to emit a sentinel token "<eos>" each time when it finishes the generation of a specific SQL value. We can easily observe that: 1) both components are conducive to the eventual performances. Graph pruning encourages the encoder to focus on 1-hop relations in order to extract the question-related schema sub-graph. And the value recognition module constructs the value memory in advance, thus eliminating the trouble of determining when to terminate for each multi-token entity. These two modules together extract and prepare salient evidences. And the structured

TABLE XIII
CASE STUDY ON THE VALIDATION SET OF SPIDER

Q: *Find the number of dog pets that are raised by female students (with sex F).*
Gold SQL: `SELECT count(*) FROM Student AS T1 JOIN Has_Pet AS T2 ON T1.StuID = T2.StuID JOIN Pets AS T3 ON T2.PetID = T3.PetID WHERE T1.Sex = "F" AND T3.PetType = "dog"`
IRNet: `SELECT count(*) FROM Pets JOIN Student ON Pets.PetID = Student.StuID WHERE Student.Sex = "dog" AND Pets.PetType = "F"` ✗
RATSQL: `SELECT count(*) FROM Pets JOIN Student JOIN Has_Pet ON Pets.PetID = Has_Pet.PetID AND Student.StuID = Has_Pet.StuID WHERE Student.Sex = "F" AND Pets.PetType = "dog"` ✓
LGESQL(ours): `SELECT count(*) FROM Pets JOIN Student JOIN Has_Pet ON Has_Pet.PetID = Pets.PetID AND Student.StuID = Has_Pet.StuID WHERE Pets.PetType = "dog" AND Student.Sex = "F"` ✓
(a) Database: pets_1

Q: *Find the name of the makers that produced some cars in the year of 1970?*
Gold SQL: `SELECT T1.Maker FROM car_makers AS T1 JOIN model_list AS T2 ON T1.Id = T2.Maker JOIN car_names AS T3 ON T2.Model = T3.Model JOIN cars_data AS T4 ON T3.MakeId = T4.Id WHERE T4.Year = "1970"`
IRNet: `SELECT car_makers.Maker FROM car_makers WHERE cars_data.Year = "1970"` ✗
RATSQL: `SELECT car_makers.Maker FROM car_makers JOIN cars_data JOIN model_list ON car_makers.Id = model_list.Maker AND cars_data.Year = car_makers.Country WHERE cars_data.Year = "1970"` ✗
LGESQL(ours): `SELECT car_makers.Maker FROM car_makers JOIN cars_data JOIN model_list JOIN car_names ON model_list.Maker = car_makers.Id AND model_list.Model = car_names.Model AND car_names.MakeId = cars_data.Id WHERE cars_data.Year = "1970"` ✓
(b) Database: car_1

Q: *What is Kyle's id?*
Gold SQL: `SELECT ID FROM Highschooler WHERE name = "Kyle"`
IRNet: `SELECT ID FROM Highschooler WHERE name = "Kyle"` ✓
RATSQL: `SELECT ID FROM Highschooler WHERE name = "Kyle"` ✓
LGESQL(ours): `SELECT ID FROM Likes JOIN Highschooler ON Highschooler.ID = Likes.student_id WHERE Highschooler.name = "Kyle"` ✗
(c) Database: network_1

We reproduce and extend the IRNet and RATSQL systems to support SQL value prediction.

decoder effectively reasons upon these clues and organizes them into the surface form. 2) Leveraging the complete table/column memories $\mathbf{T}/\mathbf{C}$ instead of the pruned ones can effectively amend the cascade error. On the other side, sampling erroneous question spans as interference candidates can improve model robustness, especially in metrics containing SQL values (EXT or EMV).

VII. DISCUSSION

In this section, we investigate the performance of our proposed HG2AST framework through case studies.

First, in Table XIII, we compare the outputs of our LGESQL encoder with other baselines, namely IRNet [8] (which ignores relation features) and RATSQL [6] (which utilizes static embeddings to parametrize relations). We can discover that: 1) In case (a), both RATSQL and LGESQL can identify the intermediate table `Has_Pet`, while IRNet often fails in SQLs that involve the `JOIN` of multiple tables. 2) LGESQL outperforms both baseline systems, especially on complicated SQL sketches that require more induction and inference. For instance, in case (b), where the connections of four tables are included, even RATSQL cannot identify the existence of the fourth table `car_names`. But LGESQL still successfully constructs a connected component by linking table "`car_names`" to other tables. It demonstrates the strengths of context-aware edge features with iterative updates over static ones. 3) However, in the last case, LGESQL is stupid to overthink and introduce an unnecessary table "`Likes`".

TABLE XIV
ERRORS IN SQL VALUES ON THE VALIDATION SET OF SPIDER

Error Type	Ratio	Cases
Morphological Changes	84%	**Q:** ··· for all *French* singers? **Gold:** ··· `WHERE` country = "*France*" **Pred:** ··· `WHERE` singer.Country = "*French*"
		Q: What is the average edispl for all *volvos*? **Gold:** ··· `FROM` car_names `AS` T1 ··· `WHERE` T1.Model = "*volvo*" **Pred:** ··· `WHERE` car_names.Model = "*volvos*"
		Q: What country is *Jetblue Airways* affiliated with? **Gold:** ··· `WHERE` Airline = "*JetBlue Airways*" **Pred:** ··· `WHERE` airlines.Airline = "*Jetblue Airways*"
		Q: How many degrees does the engineering department offer? **Gold:** ··· `WHERE` Departments.department_name = "*engineer*" **Pred:** ··· `WHERE` Departments.department_name = "*engineering*"
Synonyms or Abbreviations	8%	**Q:** ··· full names of all *left* handed players ··· **Gold:** ··· `WHERE` hand = "*L*" ··· **Pred:** ··· `WHERE` players.hand = "*left*" ···
Prediction Errors	8%	**Q:** ··· the TV Channel that supports high definition TV? **Gold:** ··· `WHERE` high_definition_TV = "*yes*" **Pred:** ··· `WHERE` TV_Channel.High_definition_TV = "*high*"

We attribute it to the side effects of over-emphasizing 1-hop relations.

To figure out the potential regularity in generation order, we mark AST nodes with decoding timestamps during inference in Fig. 8. Accordingly, we can draw some heuristic conclusions: 1) Different samples probably expand children nodes in the same frontier node set with different orders. For example, the order of expanding children nodes `from` and `orderby` in rule `sql_unit` := FromSelectOrderBy(`from`, `select`, `orderby`) are divergent in parts (a) and (d) of Fig. 8. Another heuristic phenomenon is that in case (c), where two tables need to be connected, the decoder first extends the child type `from` to construct a virtual table view. But this decision is deferred until the completion of type `select` in the other three cases where only one table is required. 2) Arguably, there may still exist a dominant order in some grammar rules. We observe that for the parent type `condition`, see Fig. 8(b), the majority of samples expand the child type `col_unit` first and then switch to the operator and value node. Sadly, it is unrealistic to determine the optimal order of expanding children types in advance. 3) The expanding order of nodes with the same type not necessarily conforms to a fixed pattern. In cases (b) and (c), the prediction of two requested columns exactly follows the order of occurrence in the question, while in cases (a) and (d), columns are produced in the reverse order. In a nutshell, the resulting order of the action sequence is affected by the interplay of multiple factors, including the appearing order in the question, the graph structure of database schema, syntactic constraints of SQL structure, and the intrinsic stochasticity triggered by the training process.

In view of the poor results concerning execution accuracy on the test set in Table IV, we further analyze erroneous samples with correct SQL sketches (EM is correct, while EXT is wrong). According to Table XIV, the causes are three-fold: 1) Not surprisingly, errors caused by morphological changes occupy the majority. For instance, the adjective "*French*" needs to be mapped into the noun "*France*". Other examples in Table XIV include singular/plural and upper/lower case transformations. Post-processing skills such as text fuzzying matching with DB cells can be applied to alleviate this problem. And we achieve 1.7% improvements with PLM ELECTRA instantly after application. 2) The diversity in natural language, such as synonyms and abbreviations, poses great difficulty on value normalization.

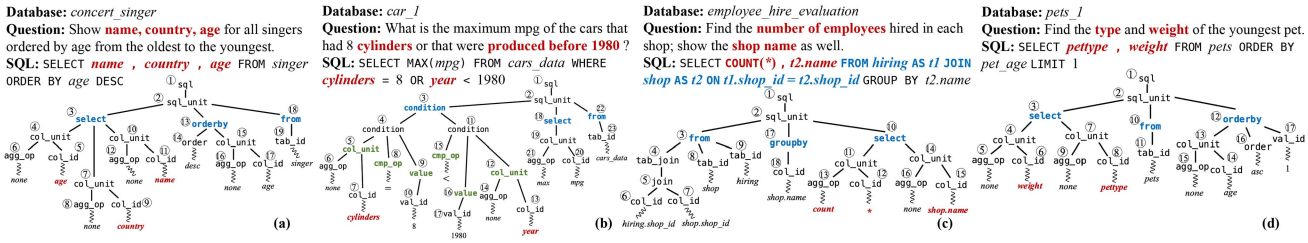

Fig. 8. AST examples marked by decoding timesteps on Spider. We omit output actions (endpoints of wavy lines) of non-leaf nodes for clarity.

In this situation, simple string matching does not work. And the value memory $\mathbf{V}$ needs to be augmented with DB cells in order to retrieve the corresponding cell. 3) The model itself inevitably makes mistakes. In the last case, the boolean value "yes" needs to be generated from the reserved vocabulary instead of copied from the input. How to combine the complementary benefits of large generative PLMs such as T5 [14] and structured AST architecture is a prospective future work.

VIII. Conclusion

This work proposes a generic heterogeneous graph to abstract syntax tree framework (HG2AST) for text-to-SQL. The LGESQL encoder leverages a line graph to capture the topological connections and contextual semantics among edges. And the distinction in processing 1-hop and multi-hop relations proves to be effective in extrapolating multi-table SQL sketches. The AST decoder with the GTL algorithm adaptively controls the expanding order of AST nodes. This relaxation can eliminate unreasonable permutation biases in the grammatical knowledge of SQL program. Besides, two auxiliary modules are both conducive to promoting the eventual performance. The HG2AST framework achieves top-ranked performances on the leaderboards of all eight benchmarks and shows promising perspectives on future improvements:

- Introducing higher-order structure knowledge about edges into the heterogeneous inputs can boost the performance of sizeable pre-trained language models. Other knowledge about graph structures, such as the latent semantic dependency among the linear input question [41], [42], can be further exploited to enhance the heterogeneous graph encoding.
- The adaptive maneuver in generation order grants the model self-motivation in exploring an interpretable inference path. Interesting future work includes adopting more predisposed strategies rather than random sampling in frontier node selection (Section V-C2).
- By explicitly injecting structure knowledge of SQL programs, grammar-based decoders exhibit significant superiority over other categories on the same model scale. Lightweight models with static word vectors in this genre provide a practical solution to resource-constrained scenarios.
- Predicting executable SQLs that contain values remains a challenging problem. Refined techniques such as entity linking and value normalization are demanded to tackle morphological changes, synonyms, abbreviations, and implicit mentions.

Acknowledgment

The authors would like to thank Tao Yu, Yusen Zhang, Hongjin Su, Qingkai Min, Yuefeng Shi, and the team of Baidu Qianyan Project for their careful assistance with the evaluation on the test datasets. The authors also want to thank the anonymous reviewers for their thoughtful comments.

References

[1] R. Cao, L. Chen, Z. Chen, Y. Zhao, S. Zhu, and K. Yu, "LGESQL: Line graph enhanced text-to-SQL model with mixed local and non-local relations," in *Proc. 49th Annu. Meeting Assoc. Comput. Linguistics*, 2021.
[2] V. Zhong, C. Xiong, and R. Socher, "Seq2SQL: Generating structured queries from natural language using reinforcement learning," 2017, *arXiv:1709.00103*.
[3] T. Yu et al., "Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, 2018, pp. 3911–3921.
[4] C. Zhang, D. Song, C. Huang, A. Swami, and N. V. Chawla, "Heterogeneous graph neural network," in *Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2019, pp. 793–803.
[5] K. Wang, W. Shen, Y. Yang, X. Quan, and R. Wang, "Relational graph attention network for aspect-based sentiment analysis," in *Proc. Annu. Meeting Assoc. Comput. Linguistics*, 2020, pp. 3229–3238.
[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 *Proc. 49th Annu. Meeting Assoc. Comput. Linguistics*, 2020.
[7] D. Chen, Y. Lin, W. Li, P. Li, J. Zhou, and X. Sun, "Measuring and relieving the over-smoothing problem for graph neural networks from the topological view," in *Proc. Conf. Assoc. Adv. Artif. Intell.*, 2020, pp. 3438–3445.
[8] J. Guo et al., "Towards complex text-to-SQL in cross-domain database with intermediate representation," in *Proc. 49th Annu. Meeting Assoc. Comput. Linguistics*, 2019, pp. 4524–4535.
[9] X. V. Lin, R. Socher, and C. Xiong, "Bridging textual and tabular data for cross-domain text-to-SQL semantic parsing," in *Proc. Conf. Empirical Methods Natural Lang. Process. Findings*, 2020.
[10] J. L. Gross and J. Yellen, *Graph Theory and its Applications*. Boca Raton, FL, USA: CRC Press, 2005.
[11] W. Yu et al., "A survey of knowledge-enhanced text generation," *ACM Comput. Surv.*, 2022, pp. 1–38.
[12] C. Zhu, Y. Xu, X. Ren, B. Lin, M. Jiang, and W. Yu, "Knowledge-augmented methods for natural language processing," in *Proc. 49th Annu. Meeting Assoc. Comput. Linguistics*, 2022.
[13] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics Hum. Lang. Technol.*, 2019, Art. no. 2.
[14] C. Raffel et al., "Exploring the limits of transfer learning with a unified text-to-text transformer," *J. Mach. Learn. Res.*, vol. 21, pp. 5485–5551, 2020.
[15] K. Cho et al., "Learning phrase representations using RNN encoder–decoder for statistical machine translation," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, 2014.

[16] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2014.

[17] Y. Liu, Z. Lin, F. Liu, Q. Dai, and W. Wang, "Generating paraphrase with topic as prior knowledge," in *Proc. Conf. Inf. Knowl. Manage.*, 2019, pp. 2381–2384.

[18] S. Zhang, E. Dinan, J. Urbanek, A. Szlam, D. Kiela, and J. Weston, "Personalizing dialogue agents: I have a dog, do you have pets too?," in *Proc. 49th Annu. Meeting Assoc. Comput. Linguistics*, 2018.

[19] W. Zhou, D.-H. Lee, R. K. Selvam, S. Lee, and X. Ren, "Pre-training text-to-text transformers for concept-centric common sense," *Int. Conf. Learn. Representations*, 2021.

[20] D. Yu, C. Zhu, Y. Yang, and M. Zeng, "Jaket: Joint pre-training of knowledge graph and language understanding," in *Proc. Conf. Assoc. Adv. Artif. Intell.*, 2022, pp. 11630–11638.

[21] A. Balakrishnan, J. Rao, K. Upasani, M. White, and R. Subba, "Constrained decoding for neural NLG from compositional representations in task-oriented dialogue," in *Proc. 49th Annu. Meeting Assoc. Comput. Linguistics*, 2019.

[22] A. See, P. J. Liu, and C. D. Manning, "Get to the point: Summarization with pointer-generator networks," in *Proc. 49th Annu. Meeting Assoc. Comput. Linguistics*, 2017, pp. 1073–1083.

[23] H. Li, Y. Su, D. Cai, Y. Wang, and L. Liu, "A survey on retrieval-augmented text generation," 2022, *arXiv:2202.01110*.

[24] H. Zhang, Z. Liu, C. Xiong, and Z. Liu, "Grounded conversation generation as guided traverses in commonsense knowledge graphs," in *Proc. 49th Annu. Meeting Assoc. Comput. Linguistics*, 2020, pp. 2031–2043.

[25] S. Ji, S. Pan, E. Cambria, P. Marttinen, and S. Y. Philip, "A survey on knowledge graphs: Representation, acquisition, and applications," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 33, no. 2, pp. 494–514, Feb. 2022.

[26] A. Gatt and E. Krahmer, "Survey of the state of the art in natural language generation: Core tasks, applications and evaluation," *J. Artif. Intell. Res.*, vol. 61, pp. 65–170, 2018.

[27] X. Fu, J. Zhang, Z. Meng, and I. King, "MAGNN: Metapath aggregated graph neural network for heterogeneous graph embedding," in *Proc. 24th Int. Conf. World Wide Web*, 2020, pp. 2331–2341.

[28] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. Van Den Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," in *Proc. Eur. Semantic Web Conf.*, 2018, pp. 593–607.

[29] Z. Chen, L. Li, and J. Bruna, "Supervised community detection with line graph neural networks," in *Proc. Int. Conf. Learn. Representations*, 2019.

[30] S. Zhu, C. Zhou, S. Pan, X. Zhu, and B. Wang, "Relation structure-aware heterogeneous graph neural network," in *Proc. Int. Conf. Data Mining*, 2019, pp. 1534–1539.

[31] Y. Zhao, L. Chen, Z. Chen, R. Cao, S. Zhu, and K. Yu, "Line graph enhanced AMR-to-text generation with mix-order graph attention networks," in *Proc. 49th Annu. Meeting Assoc. Comput. Linguistics*, 2020, pp. 732–741.

[32] C. Shi, Y. Li, J. Zhang, Y. Sun, and S. Y. Philip, "A survey of heterogeneous information network analysis," *IEEE Trans. Knowl. Data Eng.*, vol. 29, no. 1, pp. 17–37, Jan. 2017.

[33] X. Xu, C. Liu, and D. Song, "SQLNet: Generating structured queries from natural language without reinforcement learning," 2017, *arXiv:1711.04436*.

[34] T. Yu, Z. Li, Z. Zhang, R. Zhang, and D. R. Radev, "TypeSQL: Knowledge-based type-aware neural Text-to-SQL generation," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics Hum. Lang. Technol.*, 2018, pp. 588–594.

[35] R. Zhang et al., "Editing-based SQL query generation for cross-domain context-dependent questions," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, 2019, pp. 5338–5349.

[36] Q. Chen, X. Zhu, Z.-H. Ling, S. Wei, H. Jiang, and D. Inkpen, "Enhanced LSTM for natural language inference," in *Proc. 49th Annu. Meeting Assoc. Comput. Linguistics*, 2017.

[37] P. He, Y. Mao, K. Chakrabarti, and W. Chen, "X-SQL: Reinforce schema representation with context," 2019, *arXiv:1908.08113*.

[38] X. Zhang, F. Yin, G. Ma, B. Ge, and W. Xiao, "F-SQL: Fuse table schema and table content for single-table Text2SQL generation," *IEEE Access*, vol. 8, pp. 136409–136420, 2020.

[39] B. Bogin, J. Berant, and M. Gardner, "Representing schema structure with graph neural networks for text-to-SQL parsing," in *Proc. 49th Annu. Meeting Assoc. Comput. Linguistics*, 2019, pp. 4560–4565.

[40] Z. Chen et al., "ShadowGNN: Graph projection neural network for text-to-SQL parser," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics Hum. Lang. Technol.*, 2021.

[41] R. Cai, J. Yuan, B. Xu, and Z. Hao, "SADGA: Structure-aware dual graph aggregation network for Text-to-SQL," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2021, pp. 7664–7676.

[42] B. Hui et al., "S^2 SQL: Injecting syntax to question-schema interaction graph encoder for Text-to-SQL parsers," 2022, *arXiv:2203.06958*.

[43] T. Scholak, N. Schucher, and D. Bahdanau, "PICARD: Parsing incrementally for constrained auto-regressive decoding from language models," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, 2021, pp. 9895–9901.

[44] M. Lewis et al., "BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," in *Proc. 49th Annu. Meeting Assoc. Comput. Linguistics*, 2020, pp. 7871–7880.

[45] C. Wang, P. Huang, A. Polozov, M. Brockschmidt, and R. Singh, "Execution-guided neural program decoding," 2018, *arXiv:1807.03100*.

[46] J. Krishnamurthy, P. Dasigi, and M. Gardner, "Neural semantic parsing with type constraints for semi-structured tables," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, 2017, pp. 1516–1526.

[47] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," in *Proc. 49th Annu. Meeting Assoc. Comput. Linguistics*, 2017, pp. 440–450.

[48] O. Rubin and J. Berant, "SmBoP: Semi-autoregressive bottom-up semantic parsing," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics: Hum. Lang. Technol.*, 2021, pp. 311–324.

[49] O. Vinyals, S. Bengio, and M. Kudlur, "Order matters: Sequence to sequence for sets," in *Proc. Int. Conf. Learn. Representations*, 2016.

[50] D. Yu, M. Kolbæk, Z.-H. Tan, and J. Jensen, "Permutation invariant training of deep models for speaker-independent multi-talker speech separation," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, 2017, pp. 241–245.

[51] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: MIT Press, 2018.

[52] L. Dong and M. Lapata, "Language to logical form with neural attention," in *Proc. 49th Annu. Meeting Assoc. Comput. Linguistics*, 2016.

[53] Y. Guo, Z. Lin, J.-G. Lou, and D. Zhang, "Hierarchical poset decoding for compositional generalization in language," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2020, pp. 6913–6924.

[54] M. Kolbæk, D. Yu, Z.-H. Tan, and J. Jensen, "Multitalker speech separation with utterance-level permutation invariant training of deep recurrent neural networks," *IEEE/ACM Trans. Audio, Speech, Lang. Process.*, vol. 25, no. 10, pp. 1901–1913, Oct. 2017.

[55] Y. Qian, X. Chang, and D. Yu, "Single-channel multi-talker speech recognition with permutation invariant training," *Speech Commun.*, vol. 104, pp. 1–11, 2018.

[56] Y. Guo, T. Ge, and F. Wei, "Fact-aware sentence split and rephrase with permutation invariant training," in *Proc. Conf. Assoc. Adv. Artif. Intell.*, 2020, pp. 7855–7862.

[57] H. W. Kuhn, "The Hungarian method for the assignment problem," *Nav. Res. Logistics Quart.*, vol. 2, pp. 83–97, 1955.

[58] J. Ye, T. Gui, Y. Luo, Y. Xu, and Q. Zhang, "One2Set: Generating diverse keyphrases as a set," in *Proc. 49th Annu. Meeting Assoc. Comput. Linguistics*, 2021, pp. 4598–4608.

[59] Z. Lin et al., "A structured self-attentive sentence embedding," in *Proc. Int. Conf. Learn. Representations*, 2017.

[60] A. Vaswani et al., "Attention is all you need," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2017.

[61] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," in *Proc. Int. Conf. Learn. Representations*, 2018.

[62] P. Shaw, J. Uszkoreit, and A. Vaswani, "Self-attention with relative position representations," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics Hum. Lang. Technol.*, 2018, pp. 464–468.

[63] J. Pennington, R. Socher, and C. Manning, "GloVe: Global vectors for word representation," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, 2014, pp. 1532–1543.

[64] Y. Song, S. Shi, J. Li, and H. Zhang, "Directional skip-gram: Explicitly distinguishing left and right context for word embeddings," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics Hum. Lang. Technol.*, 2018, pp. 175–180.

[65] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997.

[66] A. Suhr, M.-W. Chang, P. Shaw, and K. Lee, "Exploring unexplored generalization challenges for cross-database semantic parsing," in *Proc. 49th Annu. Meeting Assoc. Comput. Linguistics*, 2020, pp. 8372–8388.

[67] Z. Huang, W. Xu, and K. Yu, "Bidirectional LSTM-CRF models for sequence tagging," 2015, *arXiv:1508.01991*.

[68] Y. Shen, S. Tan, A. Sordoni, and A. Courville, "Ordered neurons: Integrating tree structures into recurrent neural networks," in *Proc. Int. Conf. Learn. Representations*, 2018.
[69] P. Yin and G. Neubig, "TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, 2018.
[70] R. Zhong, T. Yu, and D. Klein, "Semantic evaluation for text-to-SQL with distilled test suites," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, 2020.
[71] L. Wang et al., "DuSQL: A large-scale and pragmatic Chinese text-to-SQL dataset," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, 2020.
[72] Q. Min, Y. Shi, and Y. Zhang, "A pilot study for chinese SQL semantic parsing," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, 2019.
[73] N. Sun, X. Yang, and Y. Liu, "TableQA: A large-scale chinese Text-to-SQL dataset for table-aware SQL generation," 2020, *arXiv:2006.06434*.
[74] C. Finegan-Dollak et al., "Improving text-to-SQL evaluation methodology," in *Proc. 49th Annu. Meeting Assoc. Comput. Linguistics*, 2018, pp. 351–360.
[75] A.-M. Popescu, O. Etzioni, and H. Kautz, "Towards a theory of natural language interfaces to databases," in *Proc. 8th Int. Conf. Intell. User Interfaces*, 2003, pp. 149–157.
[76] S. Iyer, I. Konstas, A. Cheung, J. Krishnamurthy, and L. Zettlemoyer, "Learning a neural semantic parser from user feedback," in *Proc. 49th Annu. Meeting Assoc. Comput. Linguistics*, 2017, pp. 963–973.
[77] P. Qi, Y. Zhang, Y. Zhang, J. Bolton, and C. D. Manning, "Stanza: A python natural language processing toolkit for many human languages," in *Proc. 49th Annu. Meeting Assoc. Comput. Linguistics: Syst. Demonstrations*, 2020, pp. 101–108.
[78] Z. Jiao, S. Sun, and K. Sun, "Chinese lexical analysis with deep Bi-GRU-CRF network," 2018, *arXiv:1807.01882*.
[79] A. Paszke et al., "PyTorch: An imperative style, high-performance deep learning library," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2019, pp. 8026–8037.
[80] M. Wang et al., "Deep graph library: A graph-centric, highly-performant package for graph neural networks," 2019, *arXiv:1909.01315*.
[81] T. Wolf et al., "Transformers: State-of-the-art natural language processing," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, 2020.
[82] Y. Gal and Z. Ghahramani, "A theoretically grounded application of dropout in recurrent neural networks," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2016, pp. 1027–1035.
[83] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," in *Proc. Int. Conf. Learn. Representations*, 2019.
[84] K. Xuan, Y. Wang, Y. Wang, Z. Wen, and Y. Dong, "Sead: End-to-end Text-to-SQL generation with schema-aware denoising," 2021, *arXiv:2105.07911*.
[85] T. Xie et al., "UnifiedSKG: Unifying and multi-tasking structured knowledge grounding with text-to-text language models," 2022, *arXiv:2201.05966*.
[86] L. Dou et al., "UniSAr: A unified structure-aware autoregressive language model for Text-to-SQL," 2022, *arXiv:2203.07781*.
[87] D. Choi, M. C. Shin, E. Kim, and D. R. Shin, "RyanSQL: Recursively applying sketch-based slot fillings for complex Text-to-SQL in cross-domain databases," *Comput. Linguistics*, vol. 47, pp. 1–24, 2021.
[88] W. Hwang, J. Yim, S. Park, and M. Seo, "A comprehensive exploration on wikiSQL with table-aware word contextualization," 2019, *arXiv:1902.01069*.
[89] X. Zhang, F. Yin, G. Ma, B. Ge, and W. Xiao, "M-SQL: Multi-task representation learning for single-table text2SQL generation," *IEEE Access*, vol. 8, pp. 43156–43167, 2020.
[90] Q. Lyu, K. Chakrabarti, S. Hathi, S. Kundu, J. Zhang, and Z. Chen, "Hybrid ranking network for Text-to-SQL," 2020, *arXiv:2008.04759*.
[91] B. McCann, N. S. Keskar, C. Xiong, and R. Socher, "The natural language Decathlon: Multitask learning as question answering," 2018, *arXiv:1806.08730*.
[92] B. Wang, M. Lapata, and I. Titov, "Meta-learning for domain generalization in semantic parsing," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics Hum. Lang. Technol.*, 2021, pp. 366–379.
[93] P. Shi et al., "Learning contextual representations for semantic parsing with generation-augmented pre-training," in *Proc. Conf. Assoc. Adv. Artif. Intell.*, 2021, pp. 13806–13814.
[94] Y. Gan et al., "Natural SQL: Making SQL easier to infer from natural language specifications," in *Proc. Conf. Empirical Methods Natural Lang. Process. Findings*, 2021, pp. 2030–2042.
[95] T. Yu et al., "GRAPPA: Grammar-augmented pre-training for table semantic parsing," in *Proc. Int. Conf. Learn. Representations*, 2020.
[96] K. Clark, M. Luong, Q. V. Le, and C. D. Manning, "ELECTRA: Pre-training text encoders as discriminators rather than generators," in *Proc. Int. Conf. Learn. Representations*, 2020.
[97] T. Yu et al., "SyntaxSQLnet: Syntax tree networks for complex and cross-domain Text-to-SQL task," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, 2018, pp. 1653–1663.
[98] K. Wu et al., "Data augmentation with hierarchical SQL-to-question generation for cross-domain Text-to-SQL parsing," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, 2021.
[99] Y. Cui, W. Che, T. Liu, B. Qin, S. Wang, and G. Hu, "Revisiting pre-trained models for Chinese natural language processing," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, 2020.
[100] A. Conneau et al., "Unsupervised cross-lingual representation learning at scale," in *Proc. 49th Annu. Meeting Assoc. Comput. Linguistics*, 2020.
[101] Z. Chi et al., "InfoXLM: An information-theoretic framework for cross-lingual language model pre-training," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics Hum. Lang. Technol.*, 2021.
[102] W. Wang, Y. Tian, H. Xiong, H. Wang, and W.-S. Ku, "A transfer-learnable natural language interface for databases," 2018, *arXiv:1809.02649*.
[103] X. Liu, P. He, W. Chen, and J. Gao, "Multi-task deep neural networks for natural language understanding," in *Proc. 49th Annu. Meeting Assoc. Comput. Linguistics*, 2019, pp. 4487–4496.
[104] Y. Liu et al., "RoBERTa: A robustly optimized bert pretraining approach," 2019, *arXiv:1907.11692*.
[105] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *Proc. Int. Conf. Learn. Representations*, Y. Bengio and Y. LeCun, Eds., 2015.

Ruisheng Cao received the BEng degree and MEng degree in computer science from Shanghai Jiao Tong University, Shanghai, China, in 2018 and 2021 respectively. He is currently working toward the PhD degree with the X-LANCE Lab, Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China. His research interests include semantic parsing, code generation, spoken language understanding, and dialogue systems.

Lu Chen is an assistant research professor with the department of computer science and engineering, Shanghai Jiao Tong University (SJTU). His research interests include dialogue systems, question answering, and natural language processing. He has published more than 30 journal articles (e.g., IEEE/ACM Transactions) and peer-reviewed conference papers (e.g., ACL, EMNLP, NAACL, AAAI), one of them was selected as COLING2018 area chair Favorites.

Jieyu Li received the BEng degree in computer science from Shanghai Jiao Tong University, Shanghai, China, in 2020. He is currently working toward the MEng degree with the X-LANCE Lab, Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China. His research interests include semantic parsing, spoken language understanding, and dialogue management.

Hanchong Zhang received the BEng degree in software engineering from Shanghai Jiao Tong University, Shanghai, China, in 2022. He is currently working toward the MEng degree in computer science with the X-LANCE Lab, Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China. His research interests include semantic parsing and language modeling.

Hongshen Xu received the BEng degree from the Department of Electronic and Information Engineering, Huazhong University of Science and Technology, Wuhan, China, in 2019. He is currently working toward the PhD degree in Shanghai Jiao Tong University, Shanghai, China. His current research interests include machine reading comprehension and multimodal information extraction.

Wangyou Zhang (Member, IEEE) received the BEng degree from the Department of Electronic and Information Engineering, Huazhong University of Science and Technology, Wuhan, China, in 2018. He is currently working toward the PhD degree with the X-LANCE Lab, Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China. His current research interests include robust speech recognition, speech signal processing and deep learning.

Kai Yu (Senior Member, IEEE) received the BEng and MSc degrees from Tsinghua University, Beijing, China, in 1999 and 2002, respectively, and the PhD degree from the Machine Intelligence Lab, the Engineering Department, Cambridge University, Cambridge, U.K., in 2006. He is currently a professor with omputer Science and Engineering Department, Shanghai Jiao Tong University, Shanghai, China. His main research interests include speech-based human machine interaction including speech recognition, synthesis, language understanding, and dialogue management. He was a Member of the IEEE Speech and Language Processing Technical Committee during 2017–2019.