Uni-Parser: Unified Semantic Parser for Question Answering on Knowledge Base and Database

Ye Liu¹, Semih Yavuz¹, Rui Meng¹, Dragomir Radev², Caiming Xiong¹, Yingbo Zhou¹
¹ Salesforce Research, ² Yale University
{yeliu, syavuz, ruimeng, yingbo.zhou, cxiong}@salesforce.com, dragomir.radev@yale.edu

Abstract

Parsing natural language questions into executable logical forms is a useful and interpretable way to perform question answering on structured data such as knowledge bases (KB) or databases (DB). However, existing approaches on semantic parsing cannot adapt to both modalities, as they suffer from the exponential growth of the logical form candidates and can hardly generalize to unseen data. In this work, we propose Uni-Parser, a unified semantic parser for question answering (QA) on both KB and DB. We introduce the primitive (relation and entity in KB, and table name, column name and cell value in DB) as an essential element in our framework. The number of primitives grows linearly with the number of retrieved relations in KB and DB, preventing us from dealing with exponential logic form candidates. We leverage the generator to predict final logical forms by altering and composing top-ranked primitives with different operations (e.g., select, where, count). With sufficiently pruned search space by a contrastive primitive ranker, the generator is empowered to capture the composition of primitives enhancing its generalization ability. We achieve competitive results on multiple KB and DB QA benchmarks more efficiently, especially in the compositional and zero-shot settings.

1 Introduction

With the recent advances in deep neural networks, question answering (QA) systems enable users to interact with massive data using queries in natural language. However, it remains challenging to assess structured data, such as knowledge bases and databases. Semantic parsing is a core step of question answering for structured data. The goal is to convert a natural language question to an executable logical form (Berant et al., 2013; Yih et al., 2015), e.g., SQL for databases and S-expression for knowledge bases.

To improve the accuracy and faithfulness in execution of semantic parsing, recent KBQA studies propose to generate logical form candidates by enumerating and selecting the best logical form by ranking (Berant and Liang, 2014; Yih et al., 2015; Sun et al., 2020; Ye et al., 2021). However, the number of logical form candidates may grow exponentially with the increase of reasoning depth for complex questions. Thus this approach can suffer from poor runtime performance due to the time-consuming logical form enumeration (Gu et al., 2021) and inefficient candidate ranking (Ye et al., 2021). For example, given an entity in a KB, we collect its logical form candidates by enumerating paths up to two hops. If the first hop with respect to the given entity contains N relations and the second hop contains M relations, the enumeration would result in $N \times M$ logical forms. As KBs typically contain massive structured knowledge, an entity can have hundreds of linked relations (Bollacker et al., 2008). Moreover, for complex questions, which require combining logic or aggregation operations, such as COUNT, ARGMIN, or ARGMAX, into the logical form, the situation could be even worse. Therefore, traditional enumeration methods may fail in cases requiring complicated reasoning (e.g., involving large amounts of entities and long reasoning chains). The situation becomes more severe and prohibitive if one wants to apply the enumeration method to other structured data with dense connections between entities such as in databases. Designing a unified semantic parsing method for various modalities of structured data have significant theoretical and practical value, yet it is still an understudied topic.

To avoid the problem of exponential growth in the logical form enumeration, we consider logical forms composed of two types of elements – primitives and operations. Primitives are defined by the schema of the structured data source and operations are a set of grammars associated with primitives.
For example, in the context of knowledge bases, primitives are defined as relations and entities in the knowledge graph. Whereas in databases, primitives are presented as tables, columns, and cells. Through this formulation, the number of candidates can be greatly reduced, going down from $N \times M$ to $N + M$.

In this study, we present Uni-Parser, a unified semantic parser for question answering on both knowledge bases (KBs) and databases (DBs). Our model follows the framework of Enumeration-Ranker-Generator proposed in RnG-KBQA (Ye et al., 2021). We first enumerate possible question-relevant primitives of a given KB or DB. Then a cross-encoder ranker is utilized to select the best candidates with contrastive learning (Chang et al., 2020), and it is further enhanced by a special hard negative sampling strategy. After getting the top-k ranked primitives for each hop, we filter out the high-order primitives that cannot be reached from the KB or do not exist in the DB through selected low-order primitives. Next, we introduce a generator that consumes both the question and filtered top-k primitives with predicted operations to compose the final logical form. Starting from primitives rather than logical forms, our generator needs to understand the semantic meaning of each primitive to compose them into the logical form.

Our contributions can be summarized as follows:
- We propose a unified semantic parser working for both KB and DB question answering.
- We enumerate primitives rather than logical forms, which greatly reduces the search space and makes candidate generation and ranking more efficient and scalable.
- The composition of logical forms from primitives and operations is postponed to the generation phase. Thus, the generator is required to learn the compositional relations among primitives. This leads to a more generalized model that can work on complex logical forms and generalize to questions involving unseen schema.
- Extensive empirical results on four KB and DB QA datasets demonstrate the effectiveness, flexibility and scalability of our unified framework.

### 2 Problem Formulation

Given a structured data source $D$ and a question $X$ in natural language, a semantic parser model is tasked to generate the corresponding logical form $Y$. Specifically, we illustrate the details depending on the type of $D$ as follows: (1) **Knowledge Base:** Data is stored in the form of subject-relation-object $(s, r, o)$, where $s$ is an entity, $r$ is a relation and $o$ is an entity or a literal (e.g., integer values, data, etc.). We use S-expressions (Gu et al., 2021) to represent logical forms for KB. S-expression is used to query a KB with the entity type, and the operations on the KB are treated as operations on a set of entities. This formulation greatly reduces the number of operations in traditional lambda DCS (Liang, 2013). (2) **Database:** A DB QA dataset typically consists of multiple tables $T = \{t_1, \cdots, t_N\}$. Each table $T_i$ contains $M$ columns: $C = \{c_1, \cdots, c_M\}$. Each column includes multiple cell values $V = \{v_1, \cdots, v_L\}$, where $L$ is the number of rows. For tabular data, SQL is used to represent logical forms (Yu et al., 2018).

Logical forms can be decomposed into the primitives and operations, both of which are defined by the schema of the structured data. We define **primitives** as the atomic elements that are entities themselves or can be used to navigate to entities. **Operations** are a set of grammars to associate primitives. Thus, a logical form can be decomposed to primitives and operations. We list specific primitives and operations considered in this study in Table 1.

<table>
<thead>
<tr>
<th>KB/DB</th>
<th>KB/DB</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>S-expression</strong></td>
<td><strong>SQL</strong></td>
</tr>
<tr>
<td>Relation, Entity</td>
<td>Table Name, Column Name, Cell Value</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>KB/DB</th>
<th>KB/DB</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Logical Operation</strong></td>
<td><strong>Logical Operation</strong></td>
</tr>
<tr>
<td>AND, JOIN, R</td>
<td>SELECT, WHERE, ORDER/GROUP BY</td>
</tr>
<tr>
<td><strong>Aggregation Operation</strong></td>
<td><strong>Conditional Operation</strong></td>
</tr>
<tr>
<td>COUNT, ARGMAX, ARGMIN</td>
<td>AVG, COUNT, MAX, MIN, SUM</td>
</tr>
<tr>
<td>&lt;, &lt;=, &gt;, &gt;=, between, !=, in, like, is, exists, not in, not like, not between, is not</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Primitives and Operations in KB and DB logical form.
We define $<|firsthop|>$ category primitives as the entities with first-hop relation and likewise for $<|secondhop|>$ category primitives (examples are shown at the top of Figure 1).

As for databases, we consider two formats of primitives. The first category is $<|tb_cl|>$, denoting the format table_name.column_name. The second one is $<|tb_cl_vl|>$, representing the format table_name.column_name $<op>$ cell_value. The $<op>$ represents a conditional operation as shown in Table 1. To enumerate the first category of primitives, we can simply use all table names together with their column names. However, for the second category, including cell values in enumeration will lead to a vast amount of candidates. For instance, on the Spider dataset (Yu et al., 2018), if we treat every cell value as a candidate, we can get up to 263K candidates for one question. To address this issue, following (Lin et al., 2020), we perform a fuzzy string match between question $X$ and the cell value $V$ under each column name $C$, and pair the matched value with its corresponding column name. One shortcoming of using string match is, that we can only obtain coverage of 15% for the Spider dataset. This is because many cell values are of numeric type, where string match fails to detect. For example, for the question "How many heads of departments are older than 56?", there is no cell value that matches 56. Therefore, if a question contains numbers, we pair them with all column names in the table.

### 3.1 Primitive Enumeration

Rather than enumerating all possible logical forms as in RnG-KBQA (Ye et al., 2021), here we only enumerate primitives that are relevant to the question.

In cases of knowledge bases, we start by detecting the entity mentioned in the question with the help of an out-of-the-box NER system and then run fuzzy matching (Lin et al., 2020) with the entity names in the knowledge base to identify relevant entities. This technique is also used in (Gu et al., 2021; Chen et al., 2021). To alleviate the issue of entity disambiguation, we follow (Ye et al., 2021) to use a ranker model to select entity candidates based on the similarity between the question and the one-hop in/out relations of the entity. Nevertheless, since most questions contain two-hop reasoning in KBs, we also extract two-hop paths associated with question entities as a related sub-graph.
special category token as a prompt in the input to differentiate primitives in the input. Note that all primitives from different categories share the same ranker. Specifically, given a question X and a primitive \( p \) with a category token \( p_c \), we use a BERT-based encoder that takes as input the concatenation of their vector representations, and outputs a logit representing the similarity between the primitive and the question:

\[
s(X, p, p_c) = \text{FFN}(\psi_\theta(p_c \oplus X \oplus p))
\]

where \( \oplus \) denotes a concatenation operation. \( \psi_\theta \) denotes the [CLS] representation of the concatenated input after BERT embedding; FNN is a projection layer reducing the representation to a scalar similarity score. \( p_c \) is the special token to distinguish the category of the primitive (in KB, \( p_c \in \{\langle \text{firsthop} \rangle, \langle \text{secondhop} \rangle \} \) and in DB, \( p_c \in \{\langle \text{tb_cl} \rangle, \langle \text{tb_cl_v1} \rangle \} \).

The ranker is optimized to minimize the contrastive loss:

\[
\mathcal{L}_\theta (X, p^+, \mathcal{P}^-, p_c) = - \frac{e^{s(X, p^+, p_c)}}{\sum_{p \in (p^+) \cup \mathcal{P}^-} e^{s(X, p, p_c)}
\]

where \( p^+ \) is the positive primitive extracted from the gold logical form and \( \mathcal{P}^- \) is the set of negative primitives from the same category \( p_c \).

3.2.1 Negative Sampling

Since a large number of negative primitive candidates can be paired with a positive example, it is necessary to apply negative sampling. A straightforward way for this is random sampling, however it may suffer from the domination of uninformative negatives (Xiong et al., 2020). To this end, we design a strategy to sample hard negative candidates for training the ranker (Liu et al., 2021b). In cases of KBs, the number of second hop relations can grow exponentially compared to the first hop. Thus the hard negative candidates of the second hop can only be sampled from the primitives connected to the ground truth first hop. In cases of DBs, for \( \langle \text{tb_cl} \rangle \) category primitives, we treat those having the same table name with ground truth but different column names as the hard negatives. And for the \( \langle \text{tb_cl_v1} \rangle \) category, we treat candidates with the same table and column name with ground truth, but having a different cell value as the hard negatives. Moreover, the bootstrap negative sampling strategy is leveraged; that is, the model is trained recursively using the false positive candidates generated from the last training epoch.

3.2.2 Primitive Candidates Filtering

In KBs, the top ranked first hop primitives and second hop primitives can be formed into two-hop paths by combining one first hop primitive with each of the second hop. However, the resulting paths may not exist in the KB. To provide valid primitive candidates to the generator, we filter out the second hop primitives that cannot be reached from any of the first hop primitives.

3.3 Logical Form Generation by Composing Primitives

In the final stage, we need to employ a generator model to predict the target logical form by composing the top primitive candidates provided by the ranker. We use a T5 model (Raffel et al., 2020) as the basis of our logical form generator, as it demonstrates strong performance on various text generation tasks. We construct the inputs by concatenating the question and the top-k primitive candidates. As shown in Figure 3, the input for KBs is formatted as: [X; \( \langle \text{secondhop} \rangle \) primitives; \( \langle \text{firsthop} \rangle \) primitives]. As for DBs, its input is formatted as: [X; |table_name| column_name1, column_name2 <op> value ... |table_name2| ...]. We train the model by teacher forcing – the target logical form is generated token by token and the model is optimized with the loss of cross-entropy. At inference time, we use beam-search to decode top-k target logical forms in an autoregressive manner.

During training, it is often the case that the ranker performs well and predicts the gold primitive at the top place, but this positional information can be misused by the generator. If the generator is biased by the ranking of primitives, it may gener-
alize poorly to unseen data. In order to encourage the generator to focus on the semantic meaning of each primitive rather than their positions, we shuffle the order of primitives in the input during training. According to our experiments, we find that the generator can benefit from this shuffle augmentation and perform robustly against the noise in the ranked primitives.

Our generator learns to generate logical forms by understanding the meaning of its elements – primitives and operations – and composing them. Compared to RnG-KBQA (Ye et al., 2021), whose generator predicts logical forms based on a list of logical form candidates, the compositionality on the basis of primitives and operations can make our model likely to generalize better on unseen structured data.

4 Experiments

4.1 Dataset and Evaluation

**KBQA** (1) GRAILQA (Gu et al., 2021) contains 64,331 questions and carefully splits the data to evaluate three levels of generalization in the task of KBQA, including i.i.d. setting, compositional generalization to unseen composition of KB schema, and zero-shot generalization to unseen KB schema. The fraction of each setting in the test set is 25%, 25%, and 50%, respectively. (2) WebQSP (Yih et al., 2016) is a dataset that evaluates KBQA approaches in i.i.d. setting. It contains 4,937 questions and requires reasoning chains with up to 2 hops. Similar to Ye et al. (2021), we randomly sample 200 examples from the training set for validation.

**DBQA** (1) Spider (Yu et al., 2018) is a multi-table text-to-SQL dataset, which contains 10,181 questions and 5,693 complex SQL queries on 200 databases. There is no overlap between train/dev/test databases. (2) WikiSQL (Zhong et al., 2017) is a single-table text-to-SQL dataset, which contains 80,654 questions and SQL queries distributed across 24,241 tables from Wikipedia. 49.6% of its dev tables and 45.1% of its test tables are not in the training set. Therefore, both datasets require models to generalize to the unseen schema composition (compositional generalization) and unseen schema (zero-shot generalization).

**Evaluation Metrics.** We use their official evaluation script for each dataset with two metrics to measure logical form of program exact match accuracy (EM) and answer accuracy (F1).

4.2 Results on KBQA

We first test our approach on KBQA with the GRAILQA and WebQSP datasets.

4.2.1 Implementation Details

For GRAILQA and WebQSP, we use the entity linking results provided by (Ye et al., 2021). After identifying a set of entities, we extract the primitives within 2 hops from the question entities. We initialize the primitive ranker using BERT-base-uncased. For each primitive category, 96 negative candidates are sampled. We trained the ranker for 3 epochs using a learning rate of 1e-5 and a batch size of 8. Bootstrap sampling is applied after every epoch. It is also noteworthy that we perform teacher-forcing while training the ranker, i.e., we use ground truth entity linking for enumerating training candidates. We base our generation model on T5-base (Raffel et al., 2020). We use top-10 primitives from each category returned by the ranker and fine-tune the T5 generation model for 10 epochs using a learning rate of 3e-5 and a batch size of 8. A vanilla T5 generation model is used without syntactic constraints, which does not guarantee the syntactic correctness nor executability of the produced logical forms. Therefore, we use an execution-augmented inference procedure, which is commonly used in previous semantic parsing related work (Devlin et al., 2017; Ye et al., 2020). We first decode top-k logical forms using beam search and then execute each logical form until finding one that yields a valid (non-empty) answer. In case none of the top-k logical forms is valid, the top-ranked primitives obtained using the ranker is returned. The rule-based method is used to formulate the final logical form, which is guaranteed to be executable. This inference schema can ensure finding one valid logical form for each problem.

4.2.2 Overall Evaluation

Table 2 and 3 summarize the results on GRAILQA and WebQSP, respectively. Our approach achieves the highest overall performance among all approaches. Compared with the methods enumerating logical forms like Bert Ranking and RnG-KBQA, our approach achieves better performance in compositional and zero-shot settings. Especially, we get 3.1% improvement over baselines on F1 on dev set and 1.3% improvement over baselines on EM on test set in the compositional setting. This matches our expectation that our generator learns the composition of the primitives. RnG-KBQA enumerates
Table 2: Exact match (EM) and F1 scores on test/dev split of the GRAILQA. The numbers of the baselines are taken from leaderboard and their research works. The reported models are based on BERT-base model for ranker and T5-base for generator. Best results among dev are bolded and the results of test better than dev are underlined.

<table>
<thead>
<tr>
<th>Model</th>
<th>EM</th>
<th>F1</th>
<th>EM</th>
<th>F1</th>
<th>EM</th>
<th>F1</th>
<th>EM</th>
<th>F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bert Ranking (Test) (Gu et al., 2021)</td>
<td>50.6</td>
<td>58.0</td>
<td>59.9</td>
<td>67.0</td>
<td>45.5</td>
<td>53.9</td>
<td>48.6</td>
<td>55.7</td>
</tr>
<tr>
<td>ReTrack (Test) (Chen et al., 2021)</td>
<td>58.1</td>
<td>65.3</td>
<td>84.4</td>
<td>87.5</td>
<td>61.5</td>
<td>70.9</td>
<td>44.6</td>
<td>52.5</td>
</tr>
<tr>
<td>UnifiedSKG (Test) (Xie et al., 2022)</td>
<td>62.4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>RNG-KBQA (Test) (Ye et al., 2021)</td>
<td>68.8</td>
<td>74.4</td>
<td>86.2</td>
<td>89.0</td>
<td>63.8</td>
<td>71.2</td>
<td>63.0</td>
<td>69.2</td>
</tr>
<tr>
<td>UnifiedSKG (Dev) (Xie et al., 2022)</td>
<td>60.0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>RNG-KBQA (Dev) (Ye et al., 2021)</td>
<td>70.7</td>
<td>75.7</td>
<td>86.4</td>
<td>88.6</td>
<td>61.7</td>
<td>68.3</td>
<td>67.4</td>
<td>73.1</td>
</tr>
<tr>
<td>Uni-Parser (Dev)</td>
<td>70.8</td>
<td>76.5</td>
<td>85.7</td>
<td>88.3</td>
<td>62.8</td>
<td>71.4</td>
<td>67.7</td>
<td>73.4</td>
</tr>
<tr>
<td>Uni-Parser (Test)</td>
<td>69.5</td>
<td>74.6</td>
<td>85.5</td>
<td>88.5</td>
<td>65.1</td>
<td>71.1</td>
<td>64.0</td>
<td>69.8</td>
</tr>
</tbody>
</table>

Table 3: Exact match (EM) and F1 scores on the test split of WebQSP. The reported models are based on BERT-base model for ranker and T5-base for generator.

<table>
<thead>
<tr>
<th>Model</th>
<th>EM</th>
<th>F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Topic Units (Lan et al., 2019)</td>
<td>-</td>
<td>67.9</td>
</tr>
<tr>
<td>STAGG (Yih et al., 2015)</td>
<td>63.9</td>
<td>71.7</td>
</tr>
<tr>
<td>QGG (Lan and Jiang, 2020)</td>
<td>-</td>
<td>74.0</td>
</tr>
<tr>
<td>CBR (Das et al., 2021)</td>
<td>70.0</td>
<td>72.8</td>
</tr>
<tr>
<td>ReTrack (Chen et al., 2021)</td>
<td>-</td>
<td>71.0</td>
</tr>
<tr>
<td>RNG-KBQA (Ye et al., 2021)</td>
<td>71.1</td>
<td>75.6</td>
</tr>
<tr>
<td>AncaneQA (Gu and Su, 2022)</td>
<td>-</td>
<td>75.3</td>
</tr>
<tr>
<td>Uni-Parser</td>
<td>71.4</td>
<td>75.8</td>
</tr>
</tbody>
</table>

Table 4: Exact match (EM) and F1 scores on the dev split of Spider. The upper block shows the comparison of the small pre-trained models, and the lower block shows the comparison of the large pre-trained models. Our model trains T5-3B model 100 epochs while T5-3B * trains 3072 epochs.

<table>
<thead>
<tr>
<th>Model</th>
<th>EM</th>
<th>F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global-GNN (Bogin et al., 2019)</td>
<td>52.7</td>
<td>-</td>
</tr>
<tr>
<td>EditSQL (Zhang et al., 2019b)</td>
<td>57.6</td>
<td>-</td>
</tr>
<tr>
<td>T5-Base (Scholak et al., 2021)</td>
<td>57.2</td>
<td>57.9</td>
</tr>
<tr>
<td>UnifiedSKG(T5-Base) (Xie et al., 2022)</td>
<td>58.1</td>
<td>-</td>
</tr>
<tr>
<td>Uni-Parser(T5-Base)</td>
<td>61.2</td>
<td>62.8</td>
</tr>
<tr>
<td>RAT-SQL (Wang et al., 2019)</td>
<td>69.7</td>
<td>-</td>
</tr>
<tr>
<td>BRIDGE(Large) (Lin et al., 2020)</td>
<td>70.0</td>
<td>68.0</td>
</tr>
<tr>
<td>T5-3B* (Scholak et al., 2021)</td>
<td>71.5</td>
<td>74.4</td>
</tr>
<tr>
<td>UnifiedSKG(T5-3B) (Xie et al., 2022)</td>
<td>71.7</td>
<td>-</td>
</tr>
<tr>
<td>Uni-Parser(T5-3B)</td>
<td>71.2</td>
<td>71.8</td>
</tr>
</tbody>
</table>

4.2.3 Efficiency Analysis

We compare the running time of the Uni-Parser and ranking-based model. To make a fair comparison, we time the process from the enumeration step to the logical form generation, using 1,000 randomly sampled questions on GrailQA datasets. We also report the average running time per question on an A100 GPU. Our model uses 19.4s, which is considerably faster than BERT+Ranking (76.3s) and RnG-KBQA (53.5s). Unlike logical form-based models, our model doesn’t need to enumerate a large amount of logical forms. Instead, only a small number of relevant primitives are considered, which leads to faster tokenization and efficient ranking.

4.3 Results on DBQA

We also evaluate our approach to the DBQA task with Spider and WikiSQL datasets.

4.3.1 Implementation Details

To construct the <|tb_cl_vl|> category primitives mentioned in Section 3.1, we find the relevant cell
values related to the question. Given a question and DB, we compute the string matching between the arbitrary length of phrase in question and the list of cell values under each column of all tables. We followed (Lin et al., 2020) to use a fuzzy matching algorithm to match a question to a possible cell value mentioned in the DB. We also detect the number value in the question and form all column names with the value as the primitives. We find that the column name in the WikiSQL dataset is vague, like “No.”, “Pick #”, and “Rank”, so we use the cell value to supplement the meaning of the column name. We use the matching cell value to locate the row and match the column name with the cell value in the same row.

We initiate the primitive ranker using BERT-base-uncased. We sample 48 negative candidates for each primitive category. We trained the ranker for 10 epochs using a learning rate of 1e-5 and a batch size of 8. Bootstrap hard negative sampling is conducted after every two epochs. We also use ground truth entity linking for enumerating training candidates. For the generator, we trained it using T5-base and 3B on Spider datasets. We use top-15 \(<\text{tb}_\text{cl}\rangle\) category primitives and top-5 \(<\text{tb}_\text{cl}_\text{vl}\rangle\) category primitives returned by the ranker and fine-tune the T5-base model for 200 epochs using a learning rate of 5e-5 and a batch size of 64. For the T5-3B model, we run it on 16 A100 GPUs with 100 epochs using a batch size of 1024. And on the WikiSQL dataset, we use T5-base and T5-large, and use top-5 \(<\text{tb}_\text{cl}\rangle\) category primitives and top-3 \(<\text{tb}_\text{cl}_\text{vl}\rangle\) category primitives as the input of the generator. We finetune the T5-base/large model for 20 epochs using a learning rate of 3e-5 and a batch size of 16.

### Table 5: Exact match (EM) and F1 scores on the test split of WikiSQL.

<table>
<thead>
<tr>
<th>Method</th>
<th>EM</th>
<th>F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLova (Hwang et al., 2019)</td>
<td>80.7</td>
<td>86.2</td>
</tr>
<tr>
<td>X-SQL (He et al., 2019)</td>
<td>83.3</td>
<td>88.7</td>
</tr>
<tr>
<td>IE-SQL (Ma et al., 2020)</td>
<td>84.6</td>
<td>88.8</td>
</tr>
<tr>
<td>NL2SQL (Guo and Gao, 2019)</td>
<td>83.7</td>
<td>89.2</td>
</tr>
<tr>
<td>HyrdaNet (Lyu et al., 2020)</td>
<td>83.8</td>
<td>89.2</td>
</tr>
<tr>
<td>BRIDGE(Large) (Lin et al., 2020)</td>
<td>85.7</td>
<td>91.1</td>
</tr>
<tr>
<td>TAPEX (Liu et al., 2021a)</td>
<td>-</td>
<td>89.5</td>
</tr>
<tr>
<td>Uni-Parser(T5-Base)</td>
<td>85.8</td>
<td>91.3</td>
</tr>
<tr>
<td>Uni-Parser(T5-Large)</td>
<td><strong>86.9</strong></td>
<td><strong>92.1</strong></td>
</tr>
</tbody>
</table>

Table 6: The Recall of the top-K ranked primitive on KB and Table. CG means category, HN means Hard Negative.

### 4.3.2 Overall Evaluation

Table 4 and 5 summarize the results on Spider and WikiSQL respectively. On the challenging Spider dataset, our model achieves competitive performance among all baseline models. Compared with generation models that use whole DB table schema as input like BRIDGE and UnifiedSKG on T5-base models, our model achieves 3% improvement, suggesting the advantage of our method. During primitive enumeration and ranking, we filter out irrelevant candidates based on the DB table schema. Compared with the other T5-3B models, our model achieves comparable performance with fewer training epochs where T5-3B* trained 3K epochs, while we train only 100 epochs. For WikiSQL, we compare the Text2SQL methods and answer generation method (TAPEX) in Table 5, and Uni-Parser outperforms all the baselines.

### 4.4 Analysis

#### Ablation Study

We perform an ablation study on the effects of hard negative strategies in ranking on WebQSP and WikiSQL, and the result is shown in Table 6. No CG means that we do not differentiate primitives by their categories. The intention of this design is to help the ranker to distinguish whether a question has two or one hop reasoning. No CG shows lower performance than the setting using categories in the input. By comparing the settings of with and without hard negative (rightmost two columns), we can see that the proposed hard negative sampling can help the ranker to better determine the positive primitive from the negative ones. Moreover, the accuracy of entity linking is WebQSP is 72.5, which means the upper bound of the ranking stage. Therefore, our model achieves 69.0 among Top-10 primitive is close to the oracle performance.

<table>
<thead>
<tr>
<th>Method</th>
<th>w/o CG</th>
<th>w/o HN</th>
<th>w/ HN</th>
</tr>
</thead>
<tbody>
<tr>
<td>KBQA: WebQSP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Top-1</td>
<td>54.5</td>
<td>56.2</td>
<td>56.2</td>
</tr>
<tr>
<td>Top-10</td>
<td>67.8</td>
<td>68.8</td>
<td>69.0</td>
</tr>
<tr>
<td>Top-20</td>
<td>70.4</td>
<td>71.4</td>
<td>71.8</td>
</tr>
<tr>
<td>TableQA: WikiSQL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Top-1</td>
<td>80.5</td>
<td>81.9</td>
<td>84.0</td>
</tr>
<tr>
<td>Top-3</td>
<td>86.5</td>
<td>87.9</td>
<td>91.6</td>
</tr>
<tr>
<td>Top-5</td>
<td>88.4</td>
<td>90.0</td>
<td>94.0</td>
</tr>
</tbody>
</table>

8864
Q: How many different dog breeds from czecho-slovakia have the same temperament as the cairn terriers?

Gold S-expression:

```
(COUNT (AND (JOIN breed_temperament.breeds (JOIN animal_breed.place_of_origin m.01mk6)) (JOIN (R animal_breed.temperament) m.02z57c)))
```

Entity Label Mapping:

- "czecho-slovakia": "m.01mk6": "Czechoslovakia"
- "Cairn Terrier": "m.02z57c": "Cairn Terrier"

**RnG-KBQA:**

```
(R (R animal_breed.temperament) m.02z57c)
```

**Final Output:**

```
Top 5 Ranked Logical Forms:
1. (COUNT (AND (JOIN breed_temperament.breeds (JOIN animal_breed.place_of_origin m.01mk6)) (JOIN (R animal_breed.temperament) m.02z57c)))
2. (COUNT (AND (JOIN breed_temperament.breeds (JOIN animal_breed.place_of_origin m.01mk6)) (JOIN (R animal_breed.temperament) m.02z57c)))
3. (COUNT (AND (JOIN breed_temperament.breeds (JOIN animal_breed.place_of_origin m.01mk6)) (JOIN (R animal_breed.temperament) m.02z57c)))
4. (COUNT (AND (JOIN breed_temperament.breeds (JOIN animal_breed.place_of_origin m.01mk6)) (JOIN (R animal_breed.temperament) m.02z57c)))
5. (COUNT (AND (JOIN breed_temperament.breeds (JOIN animal_breed.place_of_origin m.01mk6)) (JOIN (R animal_breed.temperament) m.02z57c)))
```

Case study For a more intuitive understanding of our model, we show a concrete example to illustrate the results of our model and the logical form enumeration based model RnG-KBQA (Ye et al., 2021) in Figure 4. The top-5 ranked logical forms in RnG-KBQA contain much redundant information, and none of them equals the gold S-expression. In contrast, our output from the ranker is simple and includes the ingredient of the gold S-expression. The output of the ranker is used as the input to the generator. The generator output of the RnG-KBQA is as same as the top-1 logical form from the ranker. This indicates that their generator is more like a correctness rewriter that performs minor edits on the input candidates. In comparison, our model can find the correct first and second hop primitives and generate the correct logical form. It’s worth noticing that even though proper primitive is not ranked as the top-1, our generator has the capability to find the correct.

### Related Work

We focus on semantic parsing rather than directly getting the answer, as semantic parsing is more explainable (Zhang and Buloğ, 2020; Lin et al., 2020). Many papers have applied seq2seq models to solve semantic parsing in either KB (Gu and Su, 2022; Ye et al., 2021) or Table scenarios (Dong...
and Lapata, 2016; Lin et al., 2018), treating it as a translation problem that takes a natural question as input and outputs a logical form.

**Semantic Parsing on KBQA** Past works have attempted to generate logical forms using a grammar-based bottom-up parser (Berant et al., 2013; Pasupat and Liang, 2015) or a seq2seq network (Hao et al., 2017; Zhang et al., 2019a). An alternative approach is to produce a list of logical form candidates and then use a ranker to find the ones that best match the intent of the question (Lan and Jiang, 2020; Sun et al., 2020; Luo et al., 2018). Ye et al. (2021) further employs a generation stage beyond the rank to remedy or supplement existing logical form candidates. Rather than enumerating the complete logical form, our Uni-Parser only gets the relevant primitives, which greatly improves the parser’s efficiency and compositional generalization ability. Recently, ArcaneQA proposed a generation-based model with dynamic programming induction in the KB search space to improve the faithfulness of the generated programs (Gu and Su, 2022) but it is still not as accurate as the rank-based model. DECAF (Yu et al., 2022) jointly generates both logical forms and direct answers, which help them leverage both KB and text to get better final answers.

**Semantic Parsing on DBQA (Text2SQL)** Text2SQL models take both the natural language question and the database table schema as input (Dou et al., 2022; Zhang et al., 2020b). To get the sequential version of the table schema, prior work commonly linearizes the input as a table name followed by all the column names. Lin et al. (2020); Zhang et al. (2020a) further show that using table content as supplemental information in Seq2Seq model can provide a better understanding of the table schema. Moreover, it supports the prediction of the conditional part in the logical form as mentioned in (Yavuz et al., 2018). Shaw et al. (2020) first shows that the pre-trained Seq2Seq model (Raffel et al., 2020) with 3 Billion parameters achieves competitive performance on the Spider dataset. (Scholak et al., 2021) proposes a constrained decoding method that can be compatible with various large pre-trained language models and achieves promising performance on Spider.

**Unified Question Answering** Many unified QA models convert the structured (KB) or semi-structured data (DB) to unstructured texts, which provides additional information for missing knowledge in the open-domain textual QA by directly linearizing the structured schema into text Oguz et al. (2020); Xie et al. (2022); Tay et al. (2022). Ma et al. (2021) further uses the data-to-text generator to revise the linearized schema into natural language. Li et al. (2021) proposes a hybrid QA model that either answers questions using text or generates the SQL queries from table schema on the textual and Tabular QA datasets. Our Uni-Parser works in a different direction that efficiently parses the questions into the executable logical forms on both KB and DB in a unified framework.

6 Conclusion

For unified semantic parsing on both KB and DB structured data, we propose Uni-Parser, which has three modules: primitive enumeration, ranker, and compositional generator. Our enumeration at the primitive level rather than the logical-form level produces a smaller number of potential candidates, leading to high efficiency in the enumeration and ranker steps. Moreover, training a generator to produce the logical form from the primitives leads to a more generalized and robust compositional generator. Experimental results on both KB and DB QA demonstrate the advantages of Uni-Parser, especially in the compositional and zero-shot settings.

7 Acknowledge

The authors would like to thank the members of Salesforce AI Research team for fruitful discussions, as well as the anonymous reviewers for their helpful feedback.

Limitations

The current Uni-Parser model needs to independently train on each of the datasets. In this work, we test it on four datasets. But if having more datasets, this process will be very time costing. A more unified way is having one model trained on all the datasets, either from KB or DB, once and producing a good performance on each dataset.

The other limitation is that the current model needs to indicate whether the question is from KB or DB. This makes the model hard to be applied to reality where which source can answer the question is unknown. Those limitations are challenging and we leave them for further explorations.
References


Yu Gu and Yu Su. 2022. Arcaneqa: Dynamic program induction and contextualized encoding for knowledge base question answering. COLING.


Yunshi Lan, Shuohang Wang, and Jing Jiang. 2019. Knowledge base question answering with topic units.


Ye Liu, Kazuma Hashimoto, Yingbo Zhou, Semih Yavuz, Caiming Xiong, and Philip S Yu. 2021b. Dense hierarchical retrieval for open-domain question answering. EMNLP.


Scott Wen-tau Yih, Ming-Wei Chang, Xiaodong He, and Jianfeng Gao. 2015. Semantic parsing via staged query graph generation: Question answering with knowledge base. In Proceedings of the Joint Conference of the 53rd Annual Meeting of the ACL and the 7th International Joint Conference on Natural Language Processing of the AFNLP.


Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 2018. Spider: A


