QueryAgent: A Reliable and Efficient Reasoning Framework with Environmental Feedback-based Self-Correction

Anonymous ACL submission

Abstract

Employing Large Language Models (LLMs) for semantic parsing has achieved remarkable success. However, we find existing methods fall short in terms of reliability and efficiency when hallucinations are encountered. In this paper, we address these challenges with a framework called QueryAgent, which solves a question step-by-step and performs step-wise self-correction. We introduce an environmental feedback-based self-correction method called ERASER. Unlike traditional approaches, ERASER leverages rich environmental feedback in the intermediate steps to perform selective and differentiated self-correction only when necessary. Experimental results demonstrate that QueryAgent notably outperforms all previous few-shot methods using only one example on GrailQA and GraphQ by 7.0 and 15.0 F1. Moreover, our approach exhibits superiority in terms of efficiency, including runtime, query overhead, and API invocation costs. By leveraging ERASER, we further improve another baseline (*i.e.,* AgentBench) by approximately 10 points, revealing the strong transferability of our approach.

1 Introduction

Recent advances in employing Large language models (LLMs) on various tasks have exhibited impressive performance (Brown et al., 2020; OpenAI, 2023). Among these tasks, Knowledge Base Question Answering (KBQA), which aims to answer questions over knowledge base (KB), has emerged as a critical and complex challenge, serving as an ideal testbed for assessing the reasoning capabilities of LLMs over structured data (Gu et al., 2023).

However, despite their remarkable achievements, we find that existing LLM-backend KBQA methods fall short in both reliability (the credibility of results) and efficiency (*i.e.,* running time, query times, and API invocation cost). Following the popular In-Context Learning (ICL) paradigm,
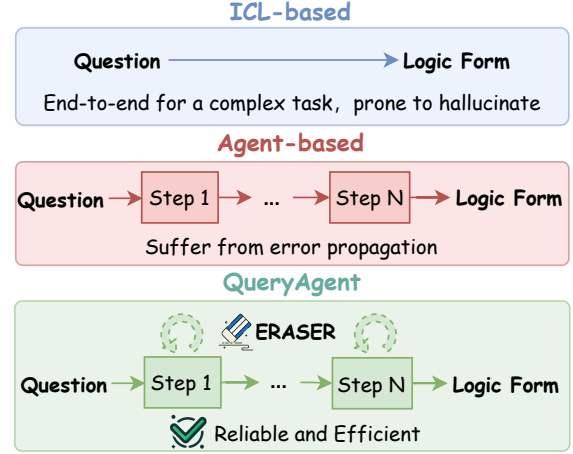

Figure 1: QueryAgent compared with two mainstream KBQA paradigms employing LLMs.

Li et al. (2023) and Nie et al. (2023) generate the target query with few-shot demonstrations. They consider LLMs as a black box and complete a complex task in one go. As a result, it lacks interpretability and is prone to hallucination (Yao et al., 2023), leading to lower accuracy of the top-1 candidate. To alleviate these issues, they employ beam search and self-consistency (Wang et al., 2023). However, these also result in numerous unreliable candidates, thus increasing the running time and query times. Typically, it requires querying thousands of SPARQL queries and several minutes to obtain the final answer.

For a complex task, solving it step-by-step has emerged as a promising solution (Wei et al., 2022; Zhou et al., 2023). AgentBench (Liu et al., 2024) implements an Agent-based (Yao et al., 2023) KBQA system by progressively invoking tools to build the target query. However, its iterative nature dictates that each step strictly relies on the previous steps. When hallucination occurs, subsequent reasoning processes would be built upon erroneous foundations, resulting in unreliable candidates and meaningless resource wastage. Additionally, the necessity to invoke an LLM at each step renders

beam search unaffordable, placing a high demand on the accuracy of the top-1 results. In our preliminary experiments, we observed that 35% of the questions in AgentBench suffer from various hallucinations. As a result, AgentBench achieves unsatisfactory performance, only 57% F1 of the state-of-the-art ICL-based methods on GrailQA.

In view of these challenges, we introduce a framework called **QueryAgent** to explore more reliable and efficient reasoning in complex environments. Specifically, QueryAgent models KBQA as a multi-turn generation task to step-by-step construct the target query with tools and perform stepwise self-correction. To mitigate the error accumulation issue of multi-step reasoning, we propose an environmental feedback-based self-correction method called **ERASER** (**E**nvi**R**onmental feedb**A**ck-based **SE**lf-co**R**rection). For each LLM generated text, ERASER detect whether it is erroneous and analyzes the possible causes based on the feedback from environments (*e.g.,* KB execution results, Python interpreter execution status, previous reasoning memory) in the intermediate steps. Upon analyzing this feedback, ERASER provides potential causes of errors and general guidelines for correction. Based on the guidelines, LLM can reconsider and correct the erroneous result.

Unlike previous self-correction methods (Pourreza and Rafiei, 2023; Chen et al., 2023) which purposelessly correct every generated result with the same few-shot demonstrations, the idea of ERASER is to actively identify and differentiate various errors based on the rich environmental feedback in the intermediate reasoning steps and then provide tailored guidelines for the distinct error type. With the help of various environmental feedback, ERASER has a more solid basis for precise detection, analysis, and correction, rather than relying solely on the final answer. Moreover, ERASER distinguishes between different types of errors, allowing it to provide guidelines specifically tailored for each error type. This targeted approach makes ERASER more purposeful and scalable. In situations where there are numerous potential error scenarios, the guidelines for different errors can be independently developed without the need to encode all possible error cases to a single prompt.

We conduct extensive experiments to evaluate the effectiveness of QueryAgent and ERASER. With only 1 example, QueryAgent notablely surpasses all few-shot methods, which require up to 100 shots, on GrailQA (+7.0), GraphQ (+15.0), WebQSP (+3.4), and MetaQA (+2.0). Moreover, our approach exhibits significant efficiency improvements. Compared with ICL-based methods, QueryAgent reduces runtime and query overhead to several orders. Compared with Agent-based methods, QueryAgent allows for approximately a 50% reduction in API invocation costs and runtime. These results highlight the reliability and efficiency of our methods. We also evaluate QueryAgent on a Text2SQL dataset (WikiSQL), and adapt ERASER to another system (Agent-Bench), to demonstrate their versatility. Results reveal that QueryAgent outperforms the previous 32-shot method by 6.9 points. Besides, ERASER relatively yields an additional improvement for Agent-Bench by 26% and 42% in F1 on the GrailQA and GraphQ, respectively [1].

2 Related Work

2.1 Few-shot KBQA

Recent advances in adopting LLMs for few-shot KBQA can be broadly categorized into 3 groups:

1) **ICL-based** KB-BINDER (Li et al., 2023) and KB-Coder (Nie et al., 2023) implement an ICL-based system by taking dozens of annotated examples into the prompt. Since they model this complex task as a simple end-to-end generation process, LLMs are directly confronted with a large search space and thus more likely to generate unreliable results. Although they incorporate beam search and self-consistency to increase the likelihood of encompassing the correct logic form, these also introduce the need to process a large number of candidates. On average, to solve a question, it takes executing thousands of candidate queries and several minutes to obtain the final answer.

2) **IR-based** Starting from an entity, StructGPT (Jiang et al., 2023), and ToG (Sun et al., 2024) iteratively walk on the graph, selecting the next neighboring entity to jump to, until finding the answer. Compared with the methods that generate an executable query, these methods can only solve questions whose reasoning process can be modeled as a single, non-branching chain. They cannot model questions with multi-constraints whose reasoning process is a tree or graph. As they traverse in the KG to obtain the answer, they have limitations on questions whose answer is not an entity in the KG (*e.g.,* aggregation or boolean question).

---

[1] Our code will be released after acceptance

3) **Agent-based** AgentBench (Liu et al., 2024) utilizes some pre-defined SPARQL templates to solve the question step-by-step, including acquiring the one-hop relation, merging two reasoning paths, adding aggregation, and so on. For a complex task, solving it step by step aligns with human intuition and helps reduce the potential search space. However, at each step, AgentBench heavily relies on the previous results, hence demanding high precision. We observe that AgentBench encounters various unexpected outputs during reasoning, leading to serious error accumulation. When hallucinations arise in the preceding steps, the subsequent become meaningless or unreliable. These factors contribute to inferior performance, which is only half as effective as the ICL-based methods.

In this work, based on the agent paradigm, we propose a reliable and efficient framework called QueryAgent, and alleviate LLM's hallucination by introducing a self-correction method.

2.2 Self-Correction

As the concern persists in the accuracy and appropriateness of LLM's generated content, self-correction has been proposed as a remedy to these issues (Pan et al., 2023). DIN-SQL (Pourreza and Rafiei, 2023) utilizes a zero-shot prompt to rectify errors in the generated SQL queries. The prompt asks LLMs to examine the generated SQL queries for potential errors and correct them while skipping those that are deemed error-free. Such intrinsic self-correction, which is solely based on LLMs' inherent capabilities without the crutch of external feedback, fails to achieve significant improvement and is unreliable (Huang et al., 2024). An intuitive improvement would be to incorporate few-shot demonstrations in the prompt (Chen et al., 2023). However, this would result in longer prompts, and can only cover a limited number of scenarios. Since they indiscriminately apply the same prompt to all cases, LLMs may be confused about which example fits the current situation. Some works like SALAM (Wang and Li, 2023) train a model to retrieve the most similar error case. Even so, it still can not ensure precise error discrimination and is heavyweight. Besides, the above methods overlook the rich feedback that the environment (*e.g.,* KB, DB) can provide for error correction. These approaches rely solely on the final output as the basis for error correction, presenting substantial challenges for LLMs to make accurate judgments. To address the above issues, we propose

ERASER, an environmental feedback-based self-correction method. Based on the feedback from the environment in the intermediate steps, ERASER proactively identifies when errors arise and provides tailored guidelines.

3 Method

3.1 Overview

In this work, we model KBQA as a semantic parsing task. We propose an LLM-backed symbolic agent called QueryAgent which step-by-step constructs formal queries with tools and performs stepwise self-correction. The process of QueryAgent can be divided into two parts: Query construction and Self-Correction (ERASER). At each step, QueryAgent first generates the action to be taken, then leverages environmental feedback to identify and distinguish potential errors. If an error is detected, the system provides tailored guidelines to help LLMs perform error correction.

The reliability and efficiency of QueryAgent are reflected in the following aspects. For reliability: 1) It solves questions progressively rather than directly outputting the complete answer. 2) We introduce a correction process, namely ERASER, during reasoning. 3) ERASER is purposeful and more precise than traditional self-correction method. For efficiency: 1) The high accuracy of our top-1 candidate eliminates the need for beam search and self-consistency. 2) Self-correction reduces meaningless reasoning along erroneous paths. 3) We perform self-correction only when necessary and only incorporate related guideline to the prompt.

3.2 Query Construction

To interact with KB and step-by-step build a target query, we leverage PyQL (Huang et al., 2023) to systematically construct the workflow. PyQL is a management toolset designed for query building over knowledge bases, including various tools to incorporate clauses to the final executable query (*i.e.,* SPARQL) , such as adding triple patterns, filters, aggregations, etc. As the final query can be transformed from a sequence of PyQL functions, our objective is to generate these functions incrementally during the multi-turn interaction.

As shown in Figure 2, at each step, the LLM provides its *thoughts* over the current step and suggests the next *action* to be taken. The *action* is a PyQL function, we execute it to obtain the results as the *observation* from the environment. For the exam-
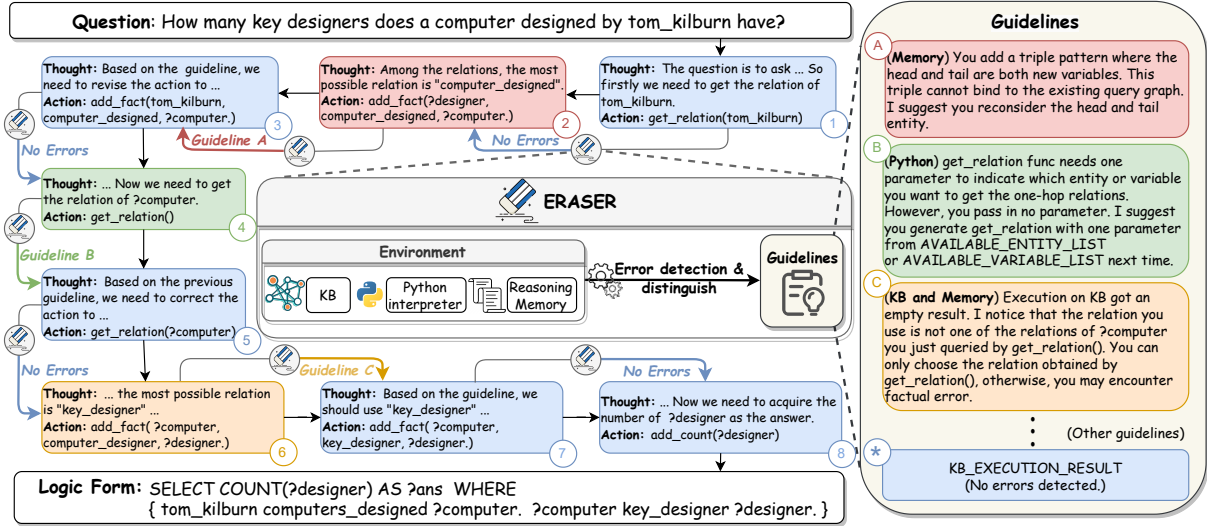
3

Figure 2: An example of QueryAgent and ERASER. At each step, the LLM generates *thought* and *action* based on the previous steps. Based on the *action*'s execution status (KB and Python) and reasoning memory, ERASER detects whether an error exists. If no error is detected, the *observation* of this step is the execution result on KB(*i.e.,* guideline *), and LLM is conducting normal reasoning. Otherwise, the *observation* is the corresponding self-correction guideline(*e.g.,* guideline A/B/C), and LLM is conducting self-correction.

ple in Figure 2, the LLM suggests firstly to obtain the one-hop relations of "tom kilburn" (*thought*) and the function `get_relation(tom_kilburn)` should be invoked at this step (*action*). By executing this function, we obtain relations around "tom kilburn" for the next step (*observation*). This process is iteratively repeated. When the reasoning process concludes we execute all generated PyQL functions to obtain the answer. Given that each step corresponds to an executable query, we can easily observe the result of the current reasoning process, similar to how humans progressively write, execute, and validate a query.

The prompt consists of four parts: the task description, the document of available functions, a running example, and the new question. We first provide an overview of the task and the rules that must be followed. Then we provide a brief document of all available functions. Following that, we present a detailed step-by-step reasoning process of an example question. Finally, we concatenate the new question that needs to be solved at the end.

3.3 ERASER

In this section, we propose an **E**nvi**R**onmental feedb**A**ck-based **SE**lf-co**R**rection method called ERASER. The key ideas underlying ERASER are to let the environment "speak out" and distinguish different types of errors. We require the system to provide feedback on its current status and any encountered errors. Based on this feedback, we attempt to identify what types of errors arise and then provide targeted and valuable guidance.

The feedback mainly originates from three environments: Knowledge Base, Python Interpreter, and Reasoning Memory. For example, KB can provide feedback such as: whether the executed result is empty, whether the reasoning process ends with a blank node (CVT) or multiple variables, error messages from the query engine, and so on. The Python interpreter can provide error messages of various invalid function calls (*e.g.,* not enough values to unpack). For reasoning memory, we can access information including but not limited to: what steps have been taken, what variables have been created, and the executed results of the previous steps.

By analyzing the above feedback, we can detect some errors and determine the cause of them. As illustrated in Figure 2, an error is raised by the Python interpreter at the fourth step due to insufficient parameters in the generated action. We choose the corresponding guideline (guideline B) as the *observation* of this step. By injecting the guideline into the reasoning process, normal reasoning and self-correction are under the same framework, without the need to design another model or agent for self-correction. In the sixth step, the query engine yields an empty result after a triple pattern constraint is added. According to the reasoning memory, we have acquired the relations of "*?computer*", but the chosen relation is not any of them. It is likely an incorrect relation

was chosen in the previous steps. This example also showcases the importance of leveraging various feedback from different environments for error distinction. For instance, whether or not the system has obtained the relations of the head/tail entity can be indicative of two distinct causes of error, but they both manifest as empty results in the execution. Compared with the previous methods which only focus on the final answer, this rich environmental feedback in the intermediate steps can serve as crucial observational points for detecting and distinguishing various errors.

The guidelines in ERASER are the description of what abnormal conditions occur and some possible solutions. They are manually written and built in within the code. Examples are shown in the right part of Figure 2. Compared with some code generation work which simply returns the original system error message (Chen et al., 2023), the guideline provided in the prompt can be seen as an intermediate language. It shields the LLM from directly considering the original error, instead focusing on easier-to-comprehend guidance, which ultimately contributes to a successful correction. Besides, by injecting the guidelines into the reasoning process, ERASER has no need for designing another specific module or agent to perform self-correction.

In this manner, we only need to establish checks for potential errors based on various environmental feedback and provide correction suggestions for each type of error. During reasoning, the system will choose the corresponding guideline based on the type of error being triggered to perform self-correction. To summarize, ERASER has the following advantages: 1) **Purposeful and Precise**: ERASER has the ability to detect errors. For each error, it provides tailored guidelines that relate to the current situation. 2) **Independent and Scalable**: The trigger for each type of error is independent. It provides convenience for incremental development without affecting the results of other questions. 3) **Lightweight and Economical**: Invocation of the LLM occurs exclusively when an error is detected. The correction prompt is a general guideline rather than lengthy few-shot examples.

4 Experiment

4.1 Datasets

We experiment QueryAgent on four KBQA datasets. The statistics can be found in Table 1. We report the performance on the dev set of GrailQA

Dataset	Training	Dev	Test
GRAILQA	44,337	6,763	13,231
GRAPHQ	2,381	-	2,395
WEBQSP	3,098	-	1,639
METAQA-3HOP	114,196	14,274	14,274
WIKISQL	56,355	8,421	15,878

Table 1: Statistics of experiment datasets.

and the performance on test set of other datasets.
GRAILQA (Gu et al., 2021) is a large-scale complex dataset that evaluates three levels of generalization (*i.e., i.i.d., compositional*, and *zero-shot*)
GRAPHQ (Su et al., 2016) is a particularly challenging dataset given that it exclusively focuses on non-i.i.d. generalization. In this paper, we use the processed version by Gu and Su (2022).
WEBQSP (Yih et al., 2016) is a simple KBQA dataset with questions from Google query logs. It mainly tests i.i.d. generalization.
METAQA (Zhang et al., 2018) consists of 1-3 hops question based on Wiki-Movies KG. We experiment on the most difficult 3-hop subset (denoted as MetaQA-3Hop).

4.2 Experimental Setup

We use `gpt-3.5-turbo` (OpenAI, 2022) for our experiments by default. All datasets use F1 as the evaluation metric. For baselines with the same setting, we report the performance from their original paper. KB-BINDER uses `Codex` which has been deprecated. For a fair comparison, we report the performance reproduced by KB-Coder with `gpt-3.5-turbo`. For KB-BINDER and KB-Coder, we compare the setting without similarity retrieval since it is not a strict few-shot setting that requires the whole annotated training set can be accessed. AgentBench reports performance on a mixed subset and uses golden linking results. We reproduce AgentBench with the same entity linking result as ours. We also implement the one-shot setting of KB-BINDER based on their public code.

4.3 Main Result

As shown in Table 2, with only one example, our method outperforms all few-shot methods that require up to 100 annotations on all four datasets. For GrailQA and GraphQ, our method notably surpasses the best few-shot methods by 7.0 and 15.0 points. On WebQSP, QueryAgent slightly surpasses 100-shot methods by 3.4 points. It is expected considering the inherent characteristics

Methods	GrailQA	GraphQ	WebQSP	MetaQA-3Hop
fine-tuning				
ArcaneQA (Gu and Su, 2022)	73.7	31.8	75.6	-
TIARA (Shu et al., 2022)	78.5	-	76.7	-
DecAF (Yu et al., 2023)	81.4	-	78.8	-
Pangu(T5-3B) (Gu et al., 2023)	83.4	57.7	79.6	-
few-shot				
Pangu (Gu et al., 2023)	53.5	35.4	48.6	-
KB-BINDER (Li et al., 2023)	50.8	34.5	56.6	96.5
KB-Coder (Nie et al., 2023)	51.7	35.8	60.5	-
McL-KBQA (Tan et al., 2023)	54.8	-	59.8	-
one-shot				
KB-BINDER (Li et al., 2023)	16.8	4.8	9.0	65.3
AgentBench (Liu et al., 2024)	30.5	25.1	26.4	-
Ours	**60.5**	**50.8**	**63.9**	**98.5**
w/ GPT4	66.8	63.0	69.0	99.9

Table 2: Overall results on GrailQA, GraphQ, WebQSP, and MetaQA-3Hop. All datasets are evaluated by F1. For the few-shot setting, Pangu uses 100-shot for all datasets. KB-BINDER and KB-Coder use 40-shot for GrailQA and 100-shot for GraphQ and WebQSP. KB-BINDER uses 5-shot for MetaQA-3Hop. McL-KBQA uses 221-shot and 144-shot for GrailQA and WebQSP, respectively.

of the datasets. Since all WebQSP questions are under I.I.D. setting and this dataset is relatively small, few-shot methods have more opportunities to encounter similar questions within the prompts. In contrast, most of the questions of GrailQA are compositional and zero-shot questions, and 100% of GraphQ are compositional questions. Few-shot methods lose this advantage on such question types, which can reasonably explain why our approach exhibits a more pronounced advantage on GrailQA and GraphQ. Additionally, all few-shot methods incorporate beam search or self-consistency to further boost the performance. It also implies that there is still space for improvement in our method if we also choose a more costly setting.

Compared with the one-shot methods, the performance of QueryAgent approximately doubles that of Agentbench, elevating the Agent-based method and one-shot KBQA to a new level. We also reproduce the one-shot result of KB-BINDER. The dramatic decline in performance exposes some limitations of the ICL-based method in terms of example quantity.

5 Detailed Analysis

To gain more insights into QueryAgent's strong performance, we conduct some in-depth analysis.

Method	GrailQA	GraphQ
Ours	**60.5**	**50.8**
w/o ERASER	43.7	35.3
w/ zero-shot SC	38.5	30.2
w/ few-shot SC	48.0	40.1

Table 3: Ablation study of ERASER and a comparison with other methods. The w/o ERASER setting replaces the guideline with the original error message. Zero-shot SC indicates the "generic" self-correction prompt of DIN-SQL (Pourreza and Rafiei, 2023). Few-shot SC indicates the "explanation feedback prompt" of Self-Debug (Chen et al., 2023). We follow and implement their ideas in our tasks.

5.1 Ablation Study

In this section, we analyze how ERASER contributes to reliable reasoning and compare it with other self-correction methods. The result is shown in Table 3. ERASER improves for 16.8 and 15.5 points for GrailQA and GraphQ, demonstrating the effectiveness of our method. For the baseline method, zero-shot SC failed to boost the performance further and even exhibited negative gains. The few-shot method has made some improvements but not that significant and its prompt is considerably longer than ERASER. It is expected since few-shot SC can only cover limited scenarios and LLM needs to figure out which part in the

Methods	GrailQA			GraphQ			WebQSP		
	TPQ	QPQ	CPQ	TPQ	QPQ	CPQ	TPQ	QPQ	CPQ
KB-BINDER	51.2 s	3297.7	**$ 0.010**	84.0 s	2113.8	$ 0.024	138.6 s	8145.1	$ 0.017
AgentBench	40.0 s	7.4	$ 0.034	65.1 s	7.2	$ 0.035	70.4 s	7.2	$ 0.038
Ours	**16.6 s**	**5.2**	$0.019	**15.3 s**	**6.2**	**$ 0.021**	**12.6 s**	**4.7**	**$ 0.014**

Table 4: Efficiency comparison with KB-BINDER and AgentBench. The TPQ, QPQ, and CPQ respectively represent the time cost, SPARQL query times, and `gpt-3.5-turbo` invocation cost per question.

prompt is related to the current situation. We also manually analyzed 200 questions of GrailQA to investigate how ERASER influences the reasoning process. We find that 43% of questions utilized ERASER in their reasoning processes. Among them, 30% questions were completely corrected. Given that our error detection strategy is conservative, each steps that triggered the ERASER were indeed found to contain errors during reasoning.

5.2 Efficiency Analysis

In this section, we evaluate the running efficiency. We conduct both horizontal and vertical comparisons by comparing KB-BINDER, which utilizes a different paradigm, and AgentBench, which is similar to ours. We analyzed the time cost per question (TPQ), query times per question (QPQ), and LLM calling cost per question (CPQ). All tests were conducted in the same network environment, with each experiment running independently.

As shown in Table 4, compared with KB-BINDER, our method exhibits overwhelming advantages in terms of TPQ and QPQ, while CPQ is a little higher on GrailQA. This outcome aligns with our expectations. KB-BINDER needs to conduct a beam search step by step to collect a large pool of candidates and then execute them one by one to find the first executable query, which requires querying numerous SPARQLs. Additionally, KB-BINDER uses self-consistency by repeating this paradigm for K times to boost the performance, leading to $(K-1)\times$ extra cost. To some extent, these also lead to a longer running time. Another thing worth noting is that more attempts also imply a lower accuracy of the top-1 candidate and a higher proportion of low-quality candidates. In contrast, our method only selects the top-1 candidate at a time, which means it requires the method to possess a high level of precision at each step. However, even under such extreme constraints, our approach still outperforms other methods.

As for the CPQ, our method incurs slightly higher costs in terms of LLM invocation compared to KB-BINDER. Our method is a step-by-step reasoning process, and while it has many advantages, we acknowledge that it also has an inevitable issue of requiring multiple requests to the LLM. However, on the flip side, KB-BINDER needs to concatenate many examples, which also faces the challenge of having a long prompt. In fact, on the 100-shot setting, the CPQ of using KB-BINDER has already exceeded that of our method.

On the other hand, compared with AgentBench, our method also surpasses it on all three criteria. It is noteworthy that our method is not only faster and cost-effective but also achieves approximately double the QA performance compared to AgentBench. At first glance, the incorporation of ERASER is a negative factor for efficiency evaluation since the prompt becomes longer than a regular reasoning process. Nonetheless, from a different perspective, timely and accurate error correction prevents the system from deviating further in the wrong direction and reduces the overhead caused by meaningless reasoning processes. Consequently, to some extent, a reliable reasoning process ultimately contributes to achieving efficient reasoning. Besides, by only performing corrections when necessary and distinguishing different types, we have managed to minimize the costs of ERASER.

5.3 Generalization Ability

In this section, we analyze the generalization ability of our method and ICL-based method from qualitative analysis and experimental comparisons.

Methodologically speaking, our method tackles the question step-by-step with atomic symbolic tools. By decomposing the problem into multiple reasoning steps, we bridge the semantic gap between different questions and datasets, as all questions can be represented using these limited tools. However, the combination of these steps can be numerous, posing challenges for compositional generalization. ICL-based methods learn and gen-

Methods	WikiSQL
few-shot(32 shot)	
Davinci-003	49.1
ChatGPT	51.6
StructGPT(Davinci-003)	64.6
StructGPT(ChatGPT)	65.6
one-shot	
AgentBench	57.6
Ours	**72.5**
w/o ERASER	67.0

Table 5: The results of QueryAgent on WikiSQL. We evaluate denotation accuracy.

Methods	GrailQA	GraphQ	WebQSP
AgentBench	30.5	25.1	26.4
w ERASER	**38.5**	**35.6**	**32.0**

Table 6: Performance of AgentBench with ERASER.

erate the complete query at once, directly facing and bearing the significantly larger search space.

From the perspective of the experiment, KB-BINDER is sensitive to whether similar examples appear in the prompt. If the most similar questions are retrieved as examples in the prompt, KB-BINDER can achieve up to 20 point improvement on WebQSP (100% i.i.d.) but a negative boost on GraphQ (100% non-i.i.d.). In contrast, our method uses the same example for all questions. Another observation is that, the higher the proportion of non-iid questions in the dataset, the greater the degree to which our approach exceeds the ICL-based approach. Compared to GrailQA (75% non-i.i.d.), QueryAgent demonstrates greater improvement on GraphQ (100% non-i.i.d.). This can also serve as evidence that QueryAgent has better generalization on unrelated examples.

5.4 Transfer Experiment

In the previous sections, we choose KBQA as a representative testbed to instantiate QueryAgent and ERASER. To illustrate the versatility of our reasoning framework and ERASER, in this section, we conduct another two experiments: 1) we implement QueryAgent framework on another semantic parsing task, namely Text2SQL. 2) we adapt ERASER to AgentBench.

We choose the test set of WikiSQL (Zhong et al., 2017) as the experiment dataset. To acquire the execution feedback from the database environment, we implement a SQL-version PyQL to help LLM access the database and provide tools to construct the SQL query. We compare our method with Struct-GPT (Jiang et al., 2023). The baseline results of Dacinci-003 and ChatGPT also come from Struct-GPT. Our method outperforms the few-shot method

with 32 examples. Besides, ERASER contributes to 7.6% of performance, indicating the generalization ability of our self-correction method.

Another experiment (*i.e.,* AgentBench + ERASER) is to further verify that ERASER can enhance the existing Agent-based KBQA system. Table 6 shows that ERASER further improves the performance of AgentBench by 8.0 and 10.5 points on GrailQA and GraphQ. By integrating ERASER, we have elevated the performance of another method to a new level, highlighting the versatility and plug-and-play nature of ERASER.

6 Conclusion

In this paper, we present a reliable and efficient framework called QueryAgent, which constructs the target query step-by-step with tools and performs stepwise self-correction. We also introduce a novel self-correction method called ERASER. It leverages rich environmental feedback to enable selective and differentiated self-correction, departing from the conventional approach which only uses the final result to conduct correction on every output with the same prompt. Experimental results demonstrate that QueryAgent notably outperforms all existing few-shot methods on four KBQA datasets with only a single example, especially on GrailQA (+7.0) and GraphQ (+15.0). Moreover, QueryAgent also exhibits superiority in efficiency with faster solving speed and lower resource utilization. Compared to ICL-based methods, our approach reduces runtime and query costs by a factor of tens, while compared to Agent-based methods, it reduces time and API invocation costs by more than half. We also show the versatility of QueryAgent and ERASER by evaluating them on a Text2SQL dataset and applying ERASER on another system (AgentBench). QueryAgent outperforms previous few-shot method and ERASER further boosts the performance of AgentBench.

Limitations

Here we would like to discuss several limitations of our method. Firstly, the various feedback is the basis to detect and distinguish different errors. If

8

the feedback is unavailable or too simplistic, such as only providing the final answer, there is insufficient information to confidently conduct error detection and differentiate between various error types. Therefore, ERASER may have limited benefits in end-to-end approaches or when applied to a too simple environment. Another limitation is that, while step-by-step solving is widely recognized as a promising way of addressing complex tasks, it inevitably leads to the issue of lengthy prompts. The cost can be further minimized by optimizing historical encoding and prompt engineering. However, these engineering techniques are not the primary focus of this study.

References

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug.

Yu Gu, Xiang Deng, and Yu Su. 2023. Don't generate, discriminate: A proposal for grounding language models to real-world environments. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4928–4949, Toronto, Canada. Association for Computational Linguistics.

Yu Gu, Sue Kase, Michelle Vanni, Brian Sadler, Percy Liang, Xifeng Yan, and Yu Su. 2021. Beyond I.I.D.: Three levels of generalization for question answering on knowledge bases. *The Web Conference 2021 - Proceedings of the World Wide Web Conference, WWW 2021*, 2021:3477–3488.

Yu Gu and Yu Su. 2022. ArcaneQA: Dynamic program induction and contextualized encoding for knowledge base question answering. In *Proceedings of the 29th International Conference on Computational Linguistics*, pages 1718–1731, Gyeongju, Republic of Korea. International Committee on Computational Linguistics.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. 2024. Large language models cannot self-correct reasoning yet. In *The Twelfth International Conference on Learning Representations*.

Xiang Huang, Sitao Cheng, Yuheng Bao, Shanshan Huang, and Yuzhong Qu. 2023. MarkQA: A large scale KBQA dataset with numerical reasoning. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 10241–10259, Singapore. Association for Computational Linguistics.

Jinhao Jiang, Kun Zhou, Zican Dong, Keming Ye, Xin Zhao, and Ji-Rong Wen. 2023. StructGPT: A general framework for large language model to reason over structured data. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 9237–9251, Singapore. Association for Computational Linguistics.

Tianle Li, Xueguang Ma, Alex Zhuang, Yu Gu, Yu Su, and Wenhu Chen. 2023. Few-shot in-context learning on knowledge base question answering. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6966–6980, Toronto, Canada. Association for Computational Linguistics.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. 2024. Agentbench: Evaluating LLMs as agents. In *The Twelfth International Conference on Learning Representations*.

Zhijie Nie, Richong Zhang, Zhongyuan Wang, and Xudong Liu. 2023. Code-style in-context learning for knowledge-based question answering.

OpenAI. 2022. Introducing chatgpt.

OpenAI. 2023. GPT-4 Technical Report.

Liangming Pan, Michael Saxon, Wenda Xu, Deepak Nathani, Xinyi Wang, and William Yang Wang. 2023. Automatically correcting large language models: Surveying the landscape of diverse self-correction strategies.

Mohammadreza Pourreza and Davood Rafiei. 2023. DIN-SQL: Decomposed in-context learning of text-to-SQL with self-correction. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Yiheng Shu, Zhiwei Yu, Yuhan Li, Börje Karlsson, Tingting Ma, Yuzhong Qu, and Chin-Yew Lin. 2022. TIARA: Multi-grained retrieval for robust question answering over large knowledge base. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 8108–8121, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Yu Su, Huan Sun, Brian Sadler, Mudhakar Srivatsa, Izzeddin Gür, Zenghui Yan, and Xifeng Yan. 2016. On generating characteristic-rich question sets for QA evaluation. In *Empirical Methods in Natural Language Processing (EMNLP)*, Austin, Texas, USA. Association for Computational Linguistics.

Jiashuo Sun, Chengjin Xu, Lumingyuan Tang, Saizhuo Wang, Chen Lin, Yeyun Gong, Lionel Ni, Heung-Yeung Shum, and Jian Guo. 2024. Think-on-graph: Deep and responsible reasoning of large language model on knowledge graph. In *The Twelfth International Conference on Learning Representations*.

Chuanyuan Tan, Yuehe Chen, Wenbiao Shao, and Wenliang Chen. 2023. Make a choice! knowledge base question answering with in-context learning.

Danqing Wang and Lei Li. 2023. Learning from mistakes via cooperative study assistant for large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 10667–10685.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*.

Wen-tau Yih, Matthew Richardson, Chris Meek, Ming-Wei Chang, and Jina Suh. 2016. The value of semantic parse labeling for knowledge base question answering. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 201–206, Berlin, Germany. Association for Computational Linguistics.

Donghan Yu, Sheng Zhang, Patrick Ng, Henghui Zhu, Alexander Hanbo Li, Jun Wang, Yiqun Hu, William Yang Wang, Zhiguo Wang, and Bing Xiang. 2023. DecAF: Joint decoding of answers and logical forms for question answering over knowledge bases. In *The Eleventh International Conference on Learning Representations*.

Yuyu Zhang, Hanjun Dai, Zornitsa Kozareva, Alexander Smola, and Le Song. 2018. Variational reasoning for question answering with knowledge graph. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *CoRR*, abs/1709.00103.

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc V Le, and Ed H. Chi. 2023. Least-to-most prompting enables complex reasoning in large language models. In *The Eleventh International Conference on Learning Representations*.

10

798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847

A Preliminary

Knowledge Base (KB) A knowledge base is a collection of subject-relation-object triples. Formally, a KB can be denoted as $\mathcal{G} = \{\langle s, r, o \rangle \mid s, r \in \mathcal{E}, r \in \mathcal{R}\}$ where $\mathcal{E}$ and $\mathcal{R}$ denote the entity set and relation set, respectively.

Knowledge Based Question Answering (KBQA) Given a KB $\mathcal{G}$ and a nature language question q, KBQA aims to find the answer a of q based on $\mathcal{G}$. Typically, KBQA is modeled as semantic parsing, where the q is mapped to an executable logic form l (*e.g.,* SPARQL, S-expression, PyQL) whose denotation is the answer. In this work, we chose PyQL as the target format of l which can be converted into equivalent SPARQL queries.

S-expression S-expression is a commonly used logical form in KBQA. It was initially introduced in the Lisp programming language and first introduced to KBQA by Gu et al. (2021). S-expression can represent some simple SPARQL grammar and can be transformed into a standard SPARQL query. Due to its simplicity compared to SPARQL, it is often used as the target format in generative KBQA approaches.

PyQL PyQL (Huang et al., 2023) stands for **Py**thonic **Q**uery **L**anguage for SPAR**QL**. It is a logic form written in Python that can be converted to SPARQL losslessly. A PyQL for a question is a sequence of PyQL functions. In this manner, PyQL exhibits how a question is solved step-by-step and can be regarded as a structural CoT (Chain-of-Thought).

B Implementation Details of QueryAgent

B.1 Detailed Elaboration of QueryAgent

Here we provide a detailed elaboration of QueryAgent along with a formal algorithmic representation (Algorithm 1).

At each step, LLM first generates the *thought* and *action* for this step based on *prompt*. The *prompt* at the first step is the one-shot reasoning case concatenated with the new question to solve and its entity linking result. For the following steps, the prompt fed to the LLM is the prompt of the last step concatenated with the (*thought*, *action*, and *observation*) of the last step. We then execute the *action* against the KB to get the feedback from KB (*kb_fb*) and the Python interpreter (*py_fb*). If the *action* is successfully executed, the *kb_fb* is the execution result of the current unfinished SPARQL query and the *py_fb* is an empty string. Otherwise,

the KB or Python interpreter will return some error logs as feedback. We also accumulate all available structural information (*current_info*) in every step as the Reasoning Memory (*RM*) to provide comprehensive information for error detection and distinguishing. The information in *RM* not only includes the directly generated result (*e.g.,* previous action and query result) but also includes the information required for the second parsing of the generated result of LLM or KB (*e.g.,* the parameters of *action*). Based on all this available info from the environment (*env_fb*), we detect if there exists an error and distinguish the error type (*error*). If an error exists, we get its corresponding guideline based on the recognized error type, and this step's *observation* is set to this *guideline*. If no error is detected, the *kb_fb* is a valid execution result against KB, and this step's *observation* is set to this *execution_result*. This process is iterated until exceeding the maximum iteration time or the LLM generates the terminate action (*i.e., execute()*). For more details, the example can be found in Appendix G.

Algorithm 1: QueryAgent

1: **Input:** question Q, entity linking result E, instruction I, maximum iteration times T;
2: $t \leftarrow 1$;
3: $prompt \leftarrow I + Q + E$;
4: $RM \leftarrow \emptyset$; // Reasoning Memory
5: **while** $t \leq T$ **do**
6:     $thought, action \leftarrow \textbf{LLM}(prompt)$;
7:     $KB\_fb, Python\_fb \leftarrow \textbf{Execute}(action)$;
8:     $current\_info \leftarrow \textbf{Get\_Info\_in\_This\_Step}()$;
9:     $RM \leftarrow RM \cup current\_info$;
10:     $env\_fb \leftarrow \{kb\_fb, py\_fb, RM\}$;
11:     $error \leftarrow \textbf{Detect\_and\_Distinguish\_Err}(env\_fb)$;
12:     **if** $error$ **then**
13:         $guideline \leftarrow \textbf{Get\_Guideline}(error)$;
14:         $obs \leftarrow guideline$;
15:         $RM \leftarrow RM - \{current\_info\}$;
16:     **else**
17:         $execution\_result \leftarrow kb\_fb$
18:         $obs \leftarrow execution\_result$;
19:     **end if**
20:     $prompt \leftarrow prompt + thought + action + obsservation$;
21:     **if** $action = execute()$ **then**
22:         **break**;
23:     **end if**
24:     $t \leftarrow t + 1$;
25: **end while**
26: **return** $execution\_result$;

B.2 Detail Elaboration of Guidelines

In QueryAgent, the guidelines are served as the *observation* in our ReAct (Yao et al., 2023) style Agent. The content of the guideline depends on the

848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874

11

situation.

- When an error arises, the guideline is the description of what abnormal conditions occur and some possible solutions. For example, the guideline A, B, C,... in Figure 2. In this case, the LLM is conducting self-correction.

- When no error arises, there is no need for self-correction. Therefore, the guideline is the execution result on KB. For example, the guideline * (When no error is detected, the guideline is KB_EXECUTION_RESULT) in Figure 2. In this case, the LLM is conducting normal reasoning.

Therefore, writing a guideline for self-correction does not require much experience. As long as the current abnormal conditions and the possible solution are given, the LLM would benefit from it. It is worth noting that the ICL-based self-correction method also needs manually written examples, thus this part of the effort is hard to avoid. Set this aside, ERASER has the advantage in coverage, scalability, precision, and length of the prompt.

B.3 Relation Ranking

In the body of this paper, we primarily focus on query construction and error correction. Here, we supplement a technical detail in query construction, namely the coarse ranking of candidate relations. Given that the one-hop relation of an entity can be numerous, considering all candidate relations will increase the prompt length, potentially exceeding the maximum context length permitted by the LLM. Thus, upon retrieving the one-hop relations for an entity or variable, we perform a coarse ranking of candidates if the number of candidates exceeds 40. We first encode the question and each relation candidate using OpenAI embeddings (ada v2)[2]. Then we calculate the cosine similarity between the encoding of each candidate and the question. Based on their similarity scores, only the top 40 relation candidates are retained as the result for this coarse ranking. The selected relations are then shuffled to mitigate any potential bias in the model's selection process due to the order of scores. The cost of invoking OpenAI embeddings on the entire GrailQA (dev), Graph (test), and WebQSP (test) datasets are $0.012, $0.002, and $0.001, respectively, almost negligible. The MetaQA dataset fea-

---

[2]https://platform.openai.com/docs/guides/embeddings

tures a smaller number of relations, thus negating the need for the coarse ranking module.

C Other Experiment Details and Analysis

C.1 Baseline Methods

We compare QueryAgent with fine-tuning and few-shot methods.

C.1.1 Fine-tuning Method

ArcaneQA (Gu and Su, 2022) is a generation-based method that incrementally synthesizes a program by dynamically predicting a sequence of sub-programs. It leverages constrained decoding to prune the search space.

TIARA (Shu et al., 2022) proposes a multi-grained retrieval method to retrieve relevant KB elements. It also applies constrained decoding to reduce grammar errors.

DecAF (Yu et al., 2023) jointly generates both logical forms and the direct answer, then combines the merits of them to get the final answer. It adopts BM25 for retrieval to obtain relevant KB subgraphs to eliminate the need for entity linking.

Pangu (Gu et al., 2023) consists of a symbolic agent to collect valid candidate plans, and an LM to select the best one. It capitalizes on the discriminative ability of LM rather than the generative ability.

C.1.2 Few-shot Method

KB-BINDER (Li et al., 2023) is an ICL-based KBQA method utilizing dozens of (Question, S-expression) pairs as examples.

KB-Coder (Nie et al., 2023) converts the s-expression to a sequence of function calls thus reducing the format error rate.

Pangu (Gu et al., 2023) is a general framework with experiments on both fine-tuning and few-shot settings. For the few-shot setting, Pangu also adopts the ICL paradigm.

McL-KBQA (Tan et al., 2023) use a rank-based method to enumerate and score logic form candidates then use LLM to choose the final logic form via ICL.

AgentBench (Liu et al., 2024) proposes an Agent-based baseline by modeling KBQA as a multi-turn open-ended generation task.

StructGPT (Jiang et al., 2023) is a few-shot based method for complex reasoning on structured data (including Table, DB, and KG). It predicts a function invocation sequence in a pre-defined order.

Methods	GrailQA	GraphQ	WebQSP
QueryAgent			
w/o RR	45.6	34.5	33.5
w HS	52.1	47.0	50.2
w ST	**59.1**	53.8	60.8
w OE (default)	56.3	**55.0**	**61.2**

Table 7: Ablation study of QueryAgent with different relation ranking (RR) strategies. HS and ST indicates Hybrid Search and Sentence Transformer, respectively. We experiment on 500 random selected questions.

C.2 API invocation Cost

The computational costs of API calls in this paper are calculated following the official methodology provided by OpenAI[3]. For gpt-3.5-turbo, that is: $cost = input\_token\_num/1000 * \$0.0015 + output\_token\_num/1000 * \$0.0020$. The count of tokens is implemented by tiktoken[4].

C.3 Impact of Different Relation Ranking Strategy

In this section, we analyze the influence of the relation ranking (RR) module and the effect of different RR strategies. The default strategy in this paper is OE which is based on the cosine similarity of the OpenAI embedding for question and candidate relation. We compare the OE setting with another two baseline strategies: HS and ST. HS indicates Hybrid Search which is implemented by BM25 and Faiss search. ST indicates Sentence Transformer. It is similar to the OE setting but replaces the OpenAI embedding with Sentence Transformer embedding. The result in Table 7 shows that without the RR module, the performance drops significantly. It is expected since we simply retained the top 40 candidates in their default order without any re-ranking process. Among all ranking strategies, the OE setting achieves the best performance on GraphQ while the ST setting achieves the best on GrailQA and WebQSP. This implies that with a proper ranking strategy, QueryAgent still has room for improvement.

C.4 Result with Golden Entity

Here we provide detailed experiment results of QueryAgent with gold entity linking. The result is

[3]https://openai.com/pricing
[4]https://github.com/openai/tiktoken/tree/main

shown in Table 8. With the gold entity linking, the performance on GrailQA, GraphQ, and WebQSP further is improved by 9.1, 10.5, and 4.9, respectively.

Methods	GrailQA	GraphQ	WebQSP
QueryAgent	60.5	50.8	63.9
w/ gold entity	**71.6**	**61.3**	**68.8**

Table 8: The results of QueryAgent with golden entity linking result.

C.5 More Analysis of ERASER

C.5.1 The Contribution of Different Environments

Take GrailQA as an example, we have designed 33 guidelines for various situations. We manually classify these guidelines according to the environment they belong to.

Type	Number	Trigger times
KB	4	204
Python	15	862
RM	9	4729
KB+RM	3	1923
Python+RM	2	341

Table 9: The distribution of different environmental feedback, which necessitates feedback from the KB, Python interpreter(Python), or Reasoning Memory(RM).

As shown in Table 9, the environment feedback that is most beneficial for ERASER is the Reasoning Memory. This also reflects the advantage of step-by-step solving, as it allows us to utilize the memory from previous steps at any step, providing more evidence for error correction.

The top 10 most frequently triggered guidelines account for 86% of the total number of triggers. The distribution is shown in Table 10.

C.5.2 The Error that can not be Detected

Note that ERASER is not omnipotent, it cannot detect all errors. Some errors may be undetectable because they do not exhibit any anomalies in any given environment.

Undetectable errors are typically higher-level semantic errors. For example, errors in the planning level may result in a query that can be executed but differs in semantic meaning from the target question. The planning level error refers to selecting

13

Trigger	Trigger times	Type
get_relation is applied to entities or variables that have not appeared before.	2867	RM
add_fact yields an empty result and get_relation has been invoked.	821	KB+RM
add_fact yields an empty result and get_relation has not been invoked yet.	740	KB+RM
add_fact introduces two new variables and get_relation has been invoked.	635	RM
add_fact introduces two new variables and get_relation has not been invoked.	464	RM
get_relation yields an empty result.	362	KB+RM
get_relation has not been invoked before add_filter.	335	RM
The output action is not one of the available valid functions.	314	Python
The operator of add_filter is used incorrectly.	242	Python+RM
The first parameter of add_filter is invalid.	171	Python+RM

Table 10: The top 10 most frequently triggered guidelines.

the wrong function to invoke or selecting too many or too few functions. For example, if the question includes an aggregation operation, but the LLM stops after adding all the triples without adding the aggregation operation, the final query is still error-free, but it would convey a different meaning than intended by the question (*e.g.,* Gold function list: add_fact, add_fact, COUNT; Predict function list: add_fact, add_fact).

Another scenario happens when LLM needs to choose a relation from candidates. If an incorrect but valid one-hop relation is chosen, the query can still be executed correctly, but the semantics are incorrect. This issue is particularly evident in the WebQSP dataset. This indicates that the relation linking is still a main obstacle for both fine-tuning-based and prompt-based KBQA. In QueryAgent we have a relation ranking model to mitigate this issue, which can be found in Section B.3.

D   Prompt used in QueryAgent

You need to understand the question and plan for the process of constructing a query to solve this question.

Remember the following RULES:
1. Never try to get a relation from a relation, for example, get_relation(measurement_unit.inverse_temperature_unit.measurement_system).

You can only choose actions from these eight actions:
1. get_relation(Freebase_mid_or_variable)
Get the one-hop relation list of a freebase mid or a variable.
2. add_fact(head,relation,tail)
Add a triple pattern: (head, relation, tail) to the query.
3. add_max(max_var)
Constrain the query by only returning the result when max_var is the biggest.
4. add_min(min_var)
Constrain the query by only returning the result when min_var is the smallest.
5. add_count(count_var,new_var)
Add a query step to count the number of elements in the variable count_var and store the result in the variable new_var.
6. add_filter(ob1,op,ob2)
Add a filter constraint, the result needs to satisfy "ob1 op ob2". ob1 and ob2 can be variables or digits, op can be >,<,>=,<=. For example, ?x,>,?y or ?x <= 0.3 is valid.
For example, if you need ?length is greater than 10, you can use add_filter(?length,>,10)
7. set_answer(answer_var)
For a SQL-like query, you need to determine which variable to return. Function set_answer is to determine this, it will constrain that the query only returns answer_var.
8. execute()
This is used when you think the query has been completely constructed. Calling this action will execute the query and get the final answer.

Figure 3: Prompt of GrailQA (Task description and tools document).

Let's start with an example

Question : which was the most recently formed cyclone that is in the same category as tropical storm rachel?

Entity : {Tropical Storm Rachel: m.0crbvqz}

Thought 1 : This question aims to find a cyclone(?cyclone), this cyclone(?cyclone) has the latest formed time(?time). Besides, the category(?category) of this cyclone(?cyclone), is also the category(?category) of storm rachel(m.0crbvqz). To construct the query, we need to first constrain ?category is the category of storm rachel(m.0crbvqz), then constrain that ?category is the category of ?cyclone, then constrain ?time as the formed time of ?cyclone, and finally, filter the max ?time and only return ?cyclone.

Action 1 : `get_relation(m.0crbvqz)`

Observation 1 : {'forward': ['kg.object_profile.prominent_type', 'meteorology.tropical_cyclone.category', 'meteorology.tropical_cyclone.tropical_cyclone_season', ...], 'backward': ['meteorology.tropical_cyclone_category.tropical_cyclones', 'meteorology.tropical_cyclone_season.tropical_cyclones', ...]}

Thought 2 : Based on the relations, the meteorology.tropical_cyclone.category is the most possible relation for storm rachel(m.0crbvqz). This is a forward relation. So we constrain (m.0crbvqz, meteorology.tropical_cyclone.category, ?category) to get the ?category of storm rachel(m.0crbvqz).

Action 2 : `add_fact(m.0crbvqz, meteorology.tropical_cyclone.category, ?category)`

Observation 2 : The value of the variable category is ['Tropical storm (SSHS)']

Thought 3 : By executing the current query, we get some instances of ?categoty, they seem some cyclone category. This result meets our expectations. Now we need to constrain the category of ?cyclone is ?category. So let us first query for the relations of ?category.

Action 3 : `get_relation(?category)`

Observation 3 : {'forward': ['kg.object_profile.prominent_type', 'meteorology.tropical_cyclone_category.max_storm_surge' 'meteorology.tropical_cyclone_category.min_wind_speed', 'meteorology.tropical_cyclone_category.tropical_cyclones', ...], 'backward': ['meteorology.tropical_cyclone.category', ...]}

Thought 4 : Base on the relations, the meteorology.tropical_cyclone.category is the most possible relation for ?category. This is a backward relation. So we constrain (?cyclone, meteorology.tropical_cyclone.category, ?category) to make sure the category of ?cyclone is ?category.

Action 4 : `add_fact(?cyclone, meteorology.tropical_cyclone.category, ?category)`

Observation 4 : The value of the variable category is ['Tropical storm (SSHS)']. The value of variable cyclone is ['Hurricane Katrina', 'Tropical Storm Rachel', 'Tormenta tropical Arlene', 'Tempesta tropicale Lee', 'Hurricane Iselle']

Thought 5 : By executing the current query, we get some instances of ?cyclone, they seem some cyclone. This result meets our expectations.
Now we need to query for the forming time of ?cyclone. So let us first query for the relations of ?cyclone.

Action 5 : `get_relation(?cyclone)`

Observation 5 :
{'forward': ['kg.object_profile.prominent_type', 'meteorology.tropical_cyclone.affected_areas', 'meteorology.tropical_cyclone.category', 'meteorology.tropical_cyclone.formed', 'meteorology.tropical_cyclone.highest_winds', 'meteorology.tropical_cyclone.indirect_fatalities', ...], 'backward': ['dataworld.gardening_hint.last_referenced_by', 'meteorology.cyclone_affected_area.cyclones', ...]}

Thought 6 : Based on the relations, the meteorology.tropical_cyclone.formed is the most possible relation for ?cyclone. This is a forward relation. So we constrain (?cyclone, meteorology.tropical_cyclone.formed, ?time) to get the ?time when ?cyclone formed.

Action 6 : `add_fact(?cyclone, meteorology.tropical_cyclone.formed, ?time)`

Observation 6 : The value of variable ?category is ['Tropical storm (SSHS)']. The value of variable ?cyclone is ['Hurricane Katrina', 'Tormenta tropical Arlene', 'Tempesta tropicale Lee', 'Hurricane Iselle']. The value of variable ?time is ['2014-07-31-08:00', '2005-06-08-08:00', '2011-09-01-08:00', '1981-11-08:00']

Thought 7 : By executing a current query, we get some instances of ?time, they seem some times. This result meets our expectations. Now we need to add an argmax constraint. The query only returns the result with max ?time.

Action 7 : `add_max(?time)`

Observation 7 : The value of a variable ?category is ['Tropical storm (SSHS)']. The value of a variable ?cyclone is ['Hurricane Iselle']. The value of a variable ?time is ['2014-07-31-08:00']

Thought 8 : By executing a current query, we get only one result, and the ?time in this result is the biggest among all ?time shown in the last step. This result meets our expectations. Now we need to determine which variable is the answer to this question. Based on the question, we know ?cyclone represents the answer, so we only return this variable.

Action 8 : `set_answer(?cyclone)`

Observation 8 : The value of variable ?cyclone is ['Hurricane Iselle']

Thought 9 : This SPARQL seems to represent the intent of the question, execute it to get the answer.

Action 9 : `execute()`

Question : NEW QUESTION
Entity list : ENTITY LINKING RESULT

Figure 4: Prompt of GrailQA (1-shot example and new question).

E Tools Sets

PyQL function	Brief description
SPARQL version (for KB)	
Tools for interacting with KB	
get_relation(entity_or_variable)	Get the one-hop relations of an entity or a variable. The parameter should be an entity (m.02xlbx) in KB or a variable (?computer).
execute()	Execute the current SPARQL query, and return the execution result on KB.
Tools for constructing query	
add_fact(head, relation, tail)	Add a triple pattern of (head, relation, tail).
add_max(max_var)	Calculate the maximum value of a given variable (max_var).
add_min(min_var)	Calculate the minimum value of a given variable (min_var).
add_count(count_var)	Count the occurrences of a given variable (count_var).
add_filter(ob1,op,ob2)	Add a comparative constraint. "ob1" and "ob2" are two objects for comparison. The "op" can be one of >, <, =, >=, <=.
set_answer(ans_var)	Set the ans_var as the variable that this query returns.
SQL version (for DB)	
Tools for interacting with DB	
get_column(column)	Get the value stored in the given column.
execute()	Execute the current SQL query, and return the execution result on DB.
Tools for constructing query	
add_condition(column, op, value)	Add a constrain which requires the SQL meets "column op value". The "column" should be the column_name of all columns. The op can be one of: =, >, <. The "value" is a specific value (*e.g.,*string, digit). For example, add_condition(Lyrics theme/style, =, Romance) is a valid case. It requires the SQL query should only return the rows that the value stored in "Lyrics theme/style" column is "Romance".
set_answer(column, aggregation_type)	Specify the answer to this SQL. What column do you want to return (column) and what aggregation function (aggregation_type) do you need to perform on it. The aggregation_type can only be: None, MAX, MIN, COUNT, SUM, AVG. If the question just needs to return the content of the column corresponding to column, and does not need to do anything else with it, just set the aggregation_type to None. If the question needs to return the number of all items in the column corresponding to column_name, set the aggregation_type to COUNT. If the question needs to return the maximum or minimum item among all items in the column corresponding to column, set the aggregation_type to MAX or MIN. If the question needs to return the summation or average of all items in column_name, set the aggregation_type to SUM or AVG.

Table 11: The toolset we used in this work. We design two versions of PyQL functions: SPARQL version (for KB) and SQL version (for DB). The tools for interacting with KB/DB need to execute against KB/DB to obtain the execution result and will not add clauses to the target query. The tools for constructing KB/DB are used to add clauses to the target query.

17

F Example Guidelines of ERASER

Trigger: When the result of add_filter is empty. All parameters are valid.
Guidelines: You choose add_filter as the action in this step. However, we get an empty result. I strongly suggest you carefully check if a comparison step is needed. If not needed and the result already meets our expectations, you can use set_answer() to determine which variable to return. If there is a need for a filter constraint, please carefully check the two comparison objects and the operator. Please re-generate only Thought STEPS + 1 and Action STEPS + 1.

Trigger: When setting CVT node as the answer.
Example: `set_answer(?conflict)`
Guidelines: You should not set ?conflict as the answer, because its value is "UnName_Entity". Please check again and re-generate only Thought STEPS + 1 and Action STEPS + 1.

Trigger: Got empty result after adding an add_fact(h,r,t) and "r" is a valid relation from the result of get_relation().
Example: Added two triple patterns to the same entity with different relations.
Guidelines: Got empty result after adding this triple pattern. You should carefully check whether this triple is needed. You likely add a triple pattern that can not match any graph on KB.

Table 12: Example guidelines which leverage KB feedback.

Trigger: When generated action is not in action_list.
Example: None
Guidelines: Invalid action, next time you must choose an action from get_relation(), add_fact(), add_max(), add_min(), add_count(), add_filter(), set_answer(), execute(). Please re-generate only Thought STEPS + 1 and Action STEPS + 1.

Trigger: When the parameter list of the generated action does not match the requirements specified in the API documentation.
Example: `add_fact(?x, ?y)`
Guidelines: add_fact(head,relation,tail) should have 3 parameters. You have 2 parameters. Please check again. (specific guidelines vary depending on the actual situation)

Trigger: When the operator is invalid in add_filter.
Example: `add_filter(?engine, aviation.aircraft_model.part_of_line, m.031vqw)`
Guidelines: You used aviation.aircraft_model.part_of_line as operator in add_filter, which is invalid. I strongly suggest you carefully check whether a comparison step and add_filter() is needed. If not needed and the result already meets our expectation, use set_answer() to determine which variable to return as the answer. If a comparison step is indeed needed, make sure the second argument is one of [>, <, >=, <=, =, !=]. Please re-generate only Thought STEPS + 1 and Action STEPS + 1.

Table 13: Example guidelines which leverage Python feedback.

Trigger: When add_fact is used but no relations have been queried.

Example: `add_fact(m.0bj4p9h, ?relation, ?newspaper)`

Guidelines: You should use get_relation() first. Suggestion: choose valid options from get_relation(m.0bj4p9h). Please check again and re-generate only Thought STEPS + 1 and Action STEPS + 1.

Trigger: When the parameter of get_relation is not a linked entity or created variable.

Example: `get_relation(g.1233lk8r)`

Guidelines: The parameter of "g.1233lk8r" is not a linked entity or created variable. Suggestion: choose from: get_relation(?instrument), get_relation(m.01663r) for next Action. Please check again and re-generate.

Trigger: When parameter of set_answer is not a existing variable.

Example: `set_answer(?boat)`

Guidelines: ?boat is not a created variable. You must set an existing variable as the answer. Existing variables include: ?car, ?speed. Please choose the proper variable and set it again.

Trigger: When two new variables are introduced in add_fact.

Example: `add_fact(?exhibition, exhibition.exhibition_subject.subject, ?subject)`

Guidelines: You introduced unexisting variable in add_fact(), which is invalid. You should find another approach to solve the question. Please re-generate only Thought STEPS + 1 and Action STEPS + 1.

Table 14: Example guidelines which leverage Reasoning Memory feedback.

G   Case Study

GrailQA
Question: is angular frequency be defined by 1.0 radians per second?
Entity: {'1.0': '1.0^^http://www.w3.org/2001/XMLSchema#float'}

Step 1:
Thought 1: This question aims to determine if the angular frequency is defined as 1.0 radians per second. To construct the query, we need to first constrain the angular frequency to be 1.0 radians per second and then check if there is any result.
Action 1: add_filter(?angular_frequency, =, 1.0)
Observation 1: ERROR_INFO: You should use get_relation() first. Choose a parameter from ['1.0']. Please re-generate only Thought 2 and Action 2.

Step 2:
Thought 2: Based on the question, we need to find the relation for the entity '1.0'. Let's use the get_relation() action to get the relations for this entity.
Action 2: get_relation(1.0)
Observation 2: {'forward': [], 'backward': ['astronomy.celestial_object.magnitude', 'base.years.years.leap_year', 'baseball.batting_statistics.rbis', 'measurement_unit.angular_velocity_unit.frequency_in_radians_per_second', ...]}

Step 3:
Thought 3: Based on the relations obtained for the entity '1.0', there is a relation called 'measurement_unit.angular_velocity_unit.frequency_in_radians_per_second' which seems to be related to angular frequency. We can use this relation to constrain the angular frequency to be 1.0 radians per second.
Action 3: add_filter(?angular_frequency, measurement_unit.angular_velocity_unit.frequency_in_radians_per_second, 1.0)
Observation 3: ERROR_INFO: You used measurement_unit.angular_velocity_unit.frequency_in_radians_per_second as operator in add_filter, which is invalid. I strongly suggest you carefully check whether a comparsion step and add_filter() is needed. If not needed and the result already meets our expectation, use set_answer() to determine which variable to return as answer. If comparision step is indeed needed, make sure the second argument is one of [>, <, >=, <=, =, !=]. Please re-generate only Thought 4 and Action 4.

Step 4:
Thought 4: Based on the error message, it seems that the relation 'measurement_unit.angular_velocity_unit.frequency_in_radians_per_second' cannot be used as an operator in the add_filter() action. We need to find another way to construct the query.
Thought 4: Since we cannot directly use the relation 'measurement_unit.angular_velocity_unit.frequency_in_radians_per_second' as an operator, we can try to find the relations for the entity '1.0' and see if there is any relation that can help us determine if the angular frequency is defined as 1.0 radians per second.
Action 4: get_relation(1.0)
Observation 4: {'forward': [], 'backward': ['astronomy.celestial_object.magnitude', 'baseball.lifetime_batting_statistics.ibbs', 'measurement_unit.angular_velocity_unit.frequency_in_radians_per_second', ...]}

Step 5:
Thought 5: Based on the relations obtained for the entity '1.0', there is a relation called 'measurement_unit.angular_velocity_unit.frequency_in_radians_per_second' which seems to be related to angular frequency. We can use this relation to constrain the angular frequency to be 1.0 radians per second.
Action 5: add_fact(?angular_frequency, measurement_unit.angular_velocity_unit.frequency_in_radians_per_second, 1.0)
Observation 5: The value of variable ?angular_frequency is ['Radian per second']

Step 6:
Thought 6: By executing the current query, we get the value of variable ?angular_frequency as ['Radian per second']. This means that the angular frequency is defined as 1.0 radians per second. Therefore, the answer to the question is "yes".
Action 6: set_answer("yes")
Observation 6: ERROR_INFO: yes is invalid. You must set a existing variable as answer. Existing variables includes: ['?angular_frequency']. Please choose proper variable and set again.

Step 7:
Thought 7: Based on the error message, it seems that I cannot set the answer as "yes". I need to set the answer as the variable that represents the angular frequency, which is "?angular_frequency".
Action 7: set_answer(?angular_frequency)
Observation 7: The value of variable ?angular_frequency is ['Radian per second']

Step 8:
Thought 8: This SPARQL query seems to represent the intent of the question. We can now execute it to get the final answer.
Action 8: execute()
Observation 8: The value of variable ?angular_frequency is ['Radian per second']

Answer: ['Radian per second']
F1: 1.0

Figure 5: A reasoning and self-correction example of GrailQA.

GraphQ

Question: find all the streams available on live365.

Entity: {'live365': 'm.04rtjj'}

Step 1:

Thought 1: This question aims to find all the streams available on live365. To construct the query, we need to first constrain that the streams are available on live365. We can use the relation 'broadcast.broadcast_on' to represent this constraint. Finally, we need to set the answer variable to be the streams (?stream) and execute the query.

Action 1: get_relation(m.04rtjj)

Observation 1: forward: ['broadcast.content.artist', 'broadcast.distributor.distributes', 'kg.object_profile.prominent_type', ...]

backward: ['broadcast.artist.content', 'broadcast.broadcast.distributor', 'type.type.instance', ...]

Step 2:

Thought 2: Based on the relations, the 'broadcast.broadcast_on' relation is the most possible relation. This is a forward relation. So we constrain (m.04rtjj, broadcast.broadcast_on, ?stream).

Action 2: add_fact(m.04rtjj, broadcast.broadcast_on, ?stream)

Observation 2: ERROR_INFO: The relation broadcast.broadcast_on that you choose leads to an empty result. The relations of m.04rtjj includes: {'forward': ['broadcast.content.artist', 'broadcast.distributor.distributes', 'kg.object_profile.prominent_type', ...], 'backward': ['broadcast.artist.content', 'broadcast.broadcast.distributor', 'type.type.instance', ...]}. Try to change a relation that matches the question best. If it still doesn't work, you may use other approach to solve the question. Please check again and re-generate only Thought 3 and Action 3.

Step 3:

Thought 3: Based on the relations provided, none of them seem to directly represent the concept of "streams available on live365". We may need to use a different approach to solve this question.

Action 3: Since we couldn't find a suitable relation to directly represent the concept of "streams available on live365", we need to use a different approach to solve this question.

Observation 3: ERROR_INFO: Invalid action, next time you must choose a action from: get_relation(), add_fact(), add_max(), add_min(), add_count(), add_filter(), set_answer(), execute(). Please re-generate only Thought 4 and Action 4.

Step 4:

Thought 4: Since we couldn't find a suitable relation to directly represent the concept of "streams available on live365", we may need to use a different approach to solve this question. One possible approach is to find a relation that represents the concept of "stream" and then filter the results to only include streams that are available on live365. Let's try to find a relation that represents the concept of "stream".

Action 4: get_relation(?stream)

Observation 4: ERROR_INFO: You cannot query for the relations of ?stream because it hasn't been binded to anything. You need to first query for get_relation(m.04rtjj). Let's modify our approach and re-generate only Thought 5 and Action 5.

Step 5:

Thought 5: Since we couldn't find a suitable relation to directly represent the concept of "streams available on live365", we may need to use a different approach to solve this question. One possible approach is to find a relation that represents the concept of "stream" and then filter the results to only include streams that are available on live365. Let's try to find a relation that represents the concept of "stream".

Action 5: get_relation(m.04rtjj)

Observation 5: forward: ['broadcast.content.artist', 'broadcast.distributor.distributes', 'type.object.key', ...]

backward: ['broadcast.artist.content', 'broadcast.broadcast.distributor', 'type.type.instance', ...]

Step 6:

Thought 6: Based on the relations, the 'broadcast.distributor.distributes' relation is the most possible relation. This is a backward relation. So we constrain (?stream, broadcast.distributor.distributes, m.04rtjj).

Action 6: add_fact(?stream, broadcast.distributor.distributes, m.04rtjj)

Observation 6: The value of variable ?stream is ['ASCAP Nashville - 128kbps Stream', 'Capital Radio - 16kbps Stream', 'ASCAP Concert - 128kbps Stream', "POP GOES THE 70's! - 16kbps Stream", "Wild 4 The 80's - 16kbps Stream", ...]

Step 7:

Thought 7: By executing the current query, we get some instances of ?stream, and these instances seem to represent different streams. This result meets our expectation. Now we need to set the answer variable to be ?stream and execute the query to get the final answer.

Action 7: set_answer(?stream)

Observation 7: The value of variable ?stream is ['ASCAP Nashville - 128kbps Stream', 'Capital Radio - 16kbps Stream', 'ASCAP Concert - 128kbps Stream', "POP GOES THE 70's! - 16kbps Stream", "Wild 4 The 80's - 16kbps Stream", ...]

Step 8:

Thought 8: This SPARQL query seems to represent the intent of the question. We can now execute it to get the final answer.

Action 8: execute()

Observation 8: The value of variable ?stream is ['ASCAP Nashville - 128kbps Stream', 'Capital Radio - 16kbps Stream', 'ASCAP Concert - 128kbps Stream', "POP GOES THE 70's! - 16kbps Stream", "Wild 4 The 80's - 16kbps Stream", ...]

Answer: ['ASCAP Nashville - 128kbps Stream', 'Capital Radio - 16kbps Stream', 'ASCAP Concert - 128kbps Stream', ...]

F1: 1.0

Figure 6: A reasoning and self-correction example of GraphQ.

WebQSP

Question: where do florida panthers play

Entity: {'florida panthers': 'm.0j8js'}

Step 1:

Thought 1: This question aims to find the place where the Florida Panthers (m.0j8js) play. To construct the query, we need to first find the relation between the Florida Panthers and the place they play, and then constrain the query to only return the corresponding place as the answer.

Action 1: get_relation(m.0j8js)

Observation 1: ['base.ontologies.ontology_instance.equivalent_instances', 'broadcast.genre.content', 'ice_hockey.hockey_team.captain', 'ice_hockey.hockey_team.coach', 'ice_hockey.hockey_team.conference', 'ice_hockey.hockey_team.division', 'kg.object_profile.prominent_type', 'sports.professional_sports_team.draft_picks', 'sports.professional_sports_team.owner_s', 'sports.sports_team.arena_stadium', 'sports.sports_team.coaches', 'sports.sports_team.colors', 'sports.sports_team.league', 'sports.sports_team.location', 'sports.sports_team.roster', 'sports.sports_team.sport', 'sports.sports_team.team_mascot', 'sports.sports_team.venue', 'type.object.key', 'user.avh.default_domain.ellerdale_topic.ellerdale_id', 'user.robert.default_domain.daylife.hero_image_id']

Step 2:

Thought 2: Based on the above relations, the most possible relation to find the place where the Florida Panthers play is 'sports.sports_team.venue'. So we constrain (m.0j8js, sports.sports_team.venue, ?place).

Action 2: add_fact(m.0j8js, sports.sports_team.venue, ?place)

Observation 2: The value of variable ?place is ['UnName_Entity', 'UnName_Entity']

Step 3:

Thought 3: By executing the current query, we get two instances of ?place, which are 'UnName_Entity'. This result does not meet our expectation. We need to further constrain the query to get the actual place where the Florida Panthers play. Let's query for the relations of 'UnName_Entity'.

Action 3: get_relation('UnName_Entity')

Observation 3: ERROR_INFO: UnName_Entity is invalid in get_relation. It must be a existing variable, such as ['?place'] or a mid from entity list, such as ['m.0j8js']. Please check again.

Step 4:

Thought 4: There seems to be an error in the query. We cannot directly query for the relations of 'UnName_Entity'. Instead, we need to use the get_relation() action with a valid variable or Freebase mid. Let's modify the query to use a valid variable.

Action 4: get_relation(?place)

Observation 4: ['sports.team_venue_relationship.from', 'sports.team_venue_relationship.team', 'sports.team_venue_relationship.to', 'sports.team_venue_relationship.venue']

Step 5:

Thought 5: Based on the above relations, the most possible relation to find the place where the Florida Panthers play is 'sports.team_venue_relationship.venue'. So we constrain (?place, sports.team_venue_relationship.venue, m.0j8js).

Action 5: add_fact(?place, sports.team_venue_relationship.venue, m.0j8js)

Observation 5: ERROR_INFO: The relation sports.team_venue_relationship.venue that you choose leads to an empty result. Re-think the question again and choose another relation from ['sports.team_venue_relationship.from', 'sports.team_venue_relationship.team', 'sports.team_venue_relationship.to', 'sports.team_venue_relationship.venue']. Please check again and re-generate only Thought 6 and Action 6.

Step 6:

Thought 6: There seems to be an error in the query. The relation 'sports.team_venue_relationship.venue' does not provide any results. We need to re-think the question and choose another relation to find the place where the Florida Panthers play.

Action 6: get_relation(m.0j8js)

Observation 6: ['base.ontologies.ontology_instance.equivalent_instances', 'broadcast.genre.content', 'ice_hockey.hockey_team.captain', 'ice_hockey.hockey_team.coach', 'ice_hockey.hockey_team.conference', 'ice_hockey.hockey_team.division', 'kg.object_profile.prominent_type', 'sports.professional_sports_team.draft_picks', 'sports.professional_sports_team.owner_s', 'sports.sports_team.arena_stadium', 'sports.sports_team.coaches', 'sports.sports_team.colors', 'sports.sports_team.league', 'sports.sports_team.location', 'sports.sports_team.roster', 'sports.sports_team.sport', 'sports.sports_team.team_mascot', 'sports.sports_team.venue', 'type.object.key', 'user.avh.default_domain.ellerdale_topic.ellerdale_id', 'user.robert.default_domain.daylife.hero_image_id']

Step 7:

Thought 7: Based on the above relations, the most possible relation to find the place where the Florida Panthers play is 'sports.sports_team.arena_stadium'. So we constrain (m.0j8js, sports.sports_team.arena_stadium, ?place).

Action 7: add_fact(m.0j8js, sports.sports_team.arena_stadium, ?place)

Observation 7: The value of variable ?place is ['BB&T Center', 'Miami Arena']

Step 8:

Thought 8: This SPARQL query seems to represent the intent of the question. We can now execute it to get the final answer.

Action 8: execute()

Observation 8: The value of variable ?place is ['BB&T Center', 'Miami Arena']

Answer: ['BB&T Center', 'Miami Arena']

F1: 1.0

Figure 7: A reasoning and self-correction example of WebQSP.