KQA Pro: A Large-Scale Dataset with Interpretable Programs and Accurate SPARQLs for Complex Question Answering over Knowledge Base

Jiaxin Shi
Tsinghua University
Beijing, China
shijx12@gmail.com

Shulin Cao
Tsinghua University
Beijing, China
caosl19@mails.tsinghua.edu.cn

Liangming Pan
National University of Singapore
Singapore, Singapore
e0272310@u.nus.edu

Yutong Xiang
Tsinghua University
Beijing, China
xiangleona@gmail.com

Lei Hou
Tsinghua University
Beijing, China
houlei@tsinghua.edu.cn

Juanzi Li
Tsinghua University
Beijing, China
lijuanzi@tsinghua.edu.cn

Hanwang Zhang
Nanyang Technological University
Singapore, Singapore
hanwangzhang@ntu.edu.sg

Bin He
Huawei Noah's Ark Lab
Beijing, China
hebin.nlp@huawei.com

ABSTRACT

Complex question answering over knowledge base (Complex KBQA) is challenging because it requires various compositional reasoning capabilities, such as multi-hop inference, attribute comparison, set operation, and etc. Existing benchmarks have some shortcomings that limit the development of Complex KBQA: 1) they only provide QA pairs without explicit reasoning processes; 2) questions are either generated by templates, leading to poor diversity, or on a small scale. To this end, we introduce KQA Pro, a large-scale dataset for Complex KBQA. We define a compositional and highly-interpretable formal format, named Program, to represent the reasoning process of complex questions. We propose compositional strategies to generate questions, corresponding SPARQLs, and Programs with a small number of templates, and then paraphrase the generated questions to natural language questions (NLQ) by crowdsourcing, giving rise to around 120K diverse instances. SPARQL and Program depict two complementary solutions to answer complex questions, which can benefit a large spectrum of QA methods. Besides the QA task, KQA Pro can also serves for the semantic parsing task. As far as we know, it is currently the largest corpus of NLQ-to-SPARQL and NLQ-to-Program. We conduct extensive experiments to evaluate whether machines can learn to answer our complex questions in different cases, that is, with only QA supervision or with intermediate SPARQL/Program supervision. We find that state-of-the-art KBQA methods learnt from only QA pairs perform very poor on our dataset, implying our questions are more challenging than previous datasets. However, pretrained models learnt from our NLQ-to-SPARQL and NLQ-to-Program annotations surprisingly achieve about 90% answering accuracy, which is even close to the human expert performance. This inspires us that combining pretrained models and large-scale annotations like KQA Pro is an effective way to bridge the gap between human and machine language.

TheWebConf '21, in submission, Ljubljana, Slovenia

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

CCS CONCEPTS

• **Computing methodologies** → **Natural language processing**; **Reasoning about belief and knowledge**.

KEYWORDS

datasets, KBQA, machine reasoning, interpretability

ACM Reference Format:
Jiaxin Shi, Shulin Cao, Liangming Pan, Yutong Xiang, Lei Hou, Juanzi Li, Hanwang Zhang, and Bin He. 2020. KQA Pro: A Large-Scale Dataset with Interpretable Programs and Accurate SPARQLs for Complex Question Answering over Knowledge Base. In *The Web Conference 2021, April 19–21, 2021, Ljubljana, Slovenia.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Thanks to the recent advances in deep models, especially large-scale unsupervised representation learning [11], question answering of simple questions over knowledge base (Simple KBQA), i.e., single-relation factoid questions [8], begins to saturate [20, 31, 45]. However, tackling complex questions (Complex KBQA) is still an ongoing challenge, due to the unsatisfied capability of compositional reasoning. To promote the community development, several benchmarks are proposed for Complex KBQA, including LC-QuAD2.0 [15], ComplexWebQuestions [40], MetaQA [49], CSQA [34], and so on. However, they suffer from the following problems:

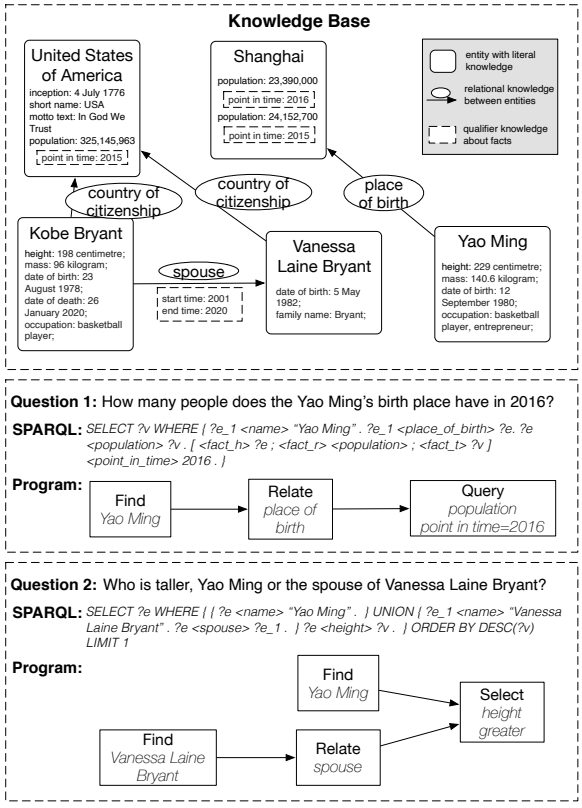


Figure 1: Example of our knowledge base and questions. Our knowledge base is a dense subset of Wikidata [43], including multiple types of knowledge. Our questions are paired with executable SPARQL queries and functional Programs.

1) Most of them only provide QA pairs without explicit reasoning processes, making it challenging for models to learn the compositional reasoning. Some researchers try to learn the reasoning processes with reinforcement learning [2, 26, 33] and searching [17]. However, the prohibitively huge search space hinders both the performance and speed, especially when the question complexity increases. For example, Saha et al. [33] achieved a 96.52% F1 score on simple questions in CSQA, whereas only 0.33% on complex questions that require comparative count. We believe that an intermediate supervision is needed for learning the compositional reasoning, mimicking the learning process of human beings [19].

2) Questions are weak in diversity and scale. For example, MetaQA [49] contains about 400K instances, but the questions are generated using just 36 templates, and they only consider relations between entities, ignoring literal attributes; LC-QuAD2.0 [15] and ComplexWebQuestions [40] have human-written questions that are fluent and diverse, but the scale is less than 40K, which is far from satisfactory.

To address the above problems, we create **KQA Pro**, a large-scale benchmark of Complex KBQA. In KQA Pro, we define **Program** to describe the reasoning process of complex questions. Programs are composed of reusable **functions**, which define the basic operations on the knowledge base. For example, Question 1 in Fig. 1, *How many people does the Yao Ming's birth place have in 2016?*, can be solved by three functions: first, `Find` the entity with name "*Yao Ming*"; second, `Relate` to the entity that is the "*place of birth*" of *Yao Ming*, i.e., *Shanghai*; finally, `Query` the "*population*" of *Shanghai* in "*2016*". Each function receives a fixed number of parameters. We analyze the basic reasoning operations over KB and define 27 functions accordingly. By composing these limited functions into Programs, we can model the compositional reasoning processes of unlimited complex questions. Besides Program, following previous work [38, 48], we also provide the corresponding SPARQL for each question, which solve a complex question by parsing it into a query graph. We believe the compositionality of Program and the graph structure of SPARQL are two complementary directions for Complex KBQA.

To get a large number of questions, we first generate textual questions, corresponding SPARQLs, and Programs via templates, and then rewrite generated questions into natural language questions (NLQ) by crowdsourcing. The paraphrased NLQ must have the totally same meaning as the generated one. To get diverse questions in a scalable way, we propose to split the question generation into two stages: locating and asking. The locating stage describes an entity or entity set in various ways (e.g., we can describe *Kobe Bryant* with *the spouse of Vanessa Laine Bryant* or *the American basketball player born on 23 August 1978*). The asking stage queries various information about an entity or entity set (e.g., query the attribute value or compare multiple entities). We generate questions by composing these two stages. Besides, we propose a recursive strategy to construct multi-hop questions, in which we can increase the hop of a question by recursively unfolding its entity (e.g., replacing "*Kobe Bryant*" with "*the spouse of Vanessa Laine Bryant*" to get Question 2 in Fig. 1).

KQA Pro consists of 117,970 QA pairs. Besides as a QA dataset, it can also serve as a semantic parsing dataset due to the SPARQLs and Programs which are parallel to questions. As far as we know, it is currently the largest corpus of NLQ-to-SPARQL and NLQ-to-Program. KQA Pro questions involve varied reasoning skills, including multi-hop reasoning, comparisons between quantity values and between time values, and set operations. We explore multiple baselines and state-of-the-art KBQA models that learn to answer questions with only QA supervision, and find that their performance drops a lot on KQA Pro than on other existing datasets. For example, EmbedKGQA [35] achieves a 66.6% accuracy on WebQuestionSP [48], while just 28.36% on KQA Pro. It demonstrates the challenge of our questions and validates our idea that QA pairs are not enough to teach machines how to solve complex questions.

However, when incorporating the intermediate supervision of logical reasoning forms, we find that the pretrained model performs surprisingly well. Specifically, we consider the semantic parsing from textual questions to SPARQLs and Programs, regard them as sequence-to-sequence tasks and use pretrained BART [25], a sequence-to-sequence variant of BERT [11], as our parser. After finetuning BART model with our large-scale parallel corpus, we achieve up to 89.68% answering accuracy, which is much better than using only QA pairs and is close to the human expert performance, 97.5%. This inspires us that with a powerful pretrained model and a large-scale parallel corpus such as KQA Pro, the translation from complex natural questions to formal languages like SPARQL and

Table 1: Comparison with existing datasets of Complex KBQA. The column *multiple kinds of knowledge* means whether the dataset considers multiple types of knowledge or just relational knowledge (introduced in Sec.3). The column *natural language* means whether the questions are in natural language or written by templates.

Dataset	multiple kinds of knowledge	number of questions	natural language	SPARQLs	reasoning processes
WebQuestions [6]	✓	5,810	✓	×	×
WebQuestionSP [48]	✓	4,737	✓	✓	×
GraphQuestions [38]	✓	5,166	✓	✓	×
LC-QuAD2.0 [15]	✓	30,000	✓	✓	×
ComplexWebQuestions [40]	✓	34,689	✓	✓	×
MetaQA [49]	×	400,000	×	×	×
CSQA [34]	×	1.6M	×	×	×
KQA Pro (ours)	✓	117,970	✓	✓	✓

Program can achieve near-human performance. Our proposed approach to construct KQA Pro in this paper is able to generate such expected parallel corpora in a low cost, which can be applied in practical business to bridge the gap between human and machine language.

To sum up, our contributions are as follows.

- We introduce KQA Pro, a large-scale KBQA dataset with challenging complex questions, accompanied with corresponding SPARQLs and Programs as additional supervision.
- We propose a question generation approach to build large-scale and diverse parallel corpora of NLQ-to-SPARQL and NLQ-to-Program. By finetuning a pretrained sequence-to-sequence model, machines can learn very well to translate NLQ to formal form.
- We conduct extensive evaluations on KQA Pro. We find when using only QA pairs, state-of-the-art models perform very poor, demonstrating that our questions are more challenging than previous datasets. When using semantic supervision to finetune a pretrained language model, machines perform very well and learn strong capability to translate a natural question to a correct executable formal form, showing great potential in practical applications.

2 RELATED WORK

2.1 Simple KBQA

The simple KBQA task is also named as factoid question answering. A simple question over knowledge base usually consists of a subject entity and a single relation, which can be mapped into the knowledge base and then retrieve the answer entity or value directly. Widely-used datasets of simple KBQA include SimpleQuestions [8], WikiMovies [28], SimpleDBpediaQA [4], FreebaseQA [22], and etc. [31] analyzed that due to the ambiguity in the data, the upper bound of the SimpleQuestions dataset was 83.4%, and their simple baseline could reach up to 78.1%. [50] achieved 85.4% accuracy on SimpleQuestions, implying that simple questions about KB have been nearly solved.

2.2 Complex KBQA

Compared with simple questions, complex questions usually require the reasoning capability, including multi-hop inference, quantitative comparison, set operation, and etc. Widely-used datasets of Complex KBQA include WebQuestions [6], WebQuestionSP [48], LC-QuAD2.0 [15], ComplexWebQuestions [40], MetaQA [49], and etc. Table 1 lists these datasets and their features. SPARQLs are provided in some datasets (i.e., WebQuestionSP, GraphQuestions, LC-QuAD2.0, and ComplexWebQuestions) to help solve complex questions. However, as a sort of query graph, SPARQL is weak in describing the intermediate procedure of the solution. We argue that explicit reasoning processes are necessary to help machines learn the capability of compositional reasoning. Besides, the scale of these existing question-to-SPARQL datasets is small, which is far from satisfactory for the training of modern deep models. MetaQA and CSQA have a large number of questions, but they ignore literal knowledge, lack SPARQLs, and their questions are written by templates rather than natural language expressions. Compared with these existing datasets, our KQA Pro has annotations of reasoning process, has annotations of SPARQL, has large-scale and natural questions, and considers multiple knowledge types. Therefore, KQA Pro serves as a more well-rounded benchmark.

2.3 Complex QA in other domains

Complex QA that requires the reasoning capability has been a hot research topic recently. In the visual domain, based on the large-scale datasets CLEVR [23] and GQA [21], which both provide the functional programs as additional supervision, many models that push the boundaries of vision and language reasoning are proposed [1, 37]. In the textual domain, the widely-used DROP [14] and HOTPOTQA [46] provide multi-hop QA pairs but lack annotations of the reasoning process. Existing models learn implicit reasoning based on graph neural networks [12, 16] or heuristics [18]. In this paper, we focus on the domain of structural knowledge base.

2.4 Semantic Parsing

A popular approach for solving complex questions is semantic parsing, that is, parsing questions from the textual form to the logical form. Multiple kinds of logical forms have been proposed and researched, including λ-calculus [3], λ-DCS [27, 29, 30, 44], SQL [51],

AMR [5], SPARQL [39], and etc. Many datasets with annotated logical forms are built to solve complex questions via semantic parsing, but most of them have a small scale (e.g., WebQuestionSP [48] consists of only 4,737 questions), and thus are not satisfactory for current large neural models. KQA Pro is proposed mainly for the task of Complex KBQA, meanwhile it can also serves as a large-scale dataset for the task of semantic parsing. It includes about 120k questions paired with two different logical forms, i.e., SPARQL and Program, and it is the largest NLQ-to-SPARQL dataset as far as we know.

3 KNOWLEDGE BASE DEFINITION

Our knowledge base is constructed from one of the largest wiki knowledge base—Wikidata [43]. Its definition is universal for other knowledge bases, which consists of

Entity, the most basic item in KB.

Predicate, the link between two entities.

Attribute, the literal information of an entity. An attribute has a key and a value. The value is one of four types[1]: string, number, date, and year. The number value has an extra unit, e.g., 229 centimetres.

Concept, a set of entities. Entities are linked to concepts via the Wikidata predicate *instance of*. Concepts are organized into a tree structure via the Wikidata predicate *subclass of*.

Relational knowledge, the triple whose form is (entity, predicate, entity), e.g., *(Yao Ming, place of birth, Shanghai)*.

Literal knowledge, the triple whose form is (entity, attribute key, attribute value), e.g., *(Yao Ming, height, 229 centimetres)*.

Qualifier knowledge, the triple whose head is a relational or literal triple, e.g., *((Kobe Bryant, spouse, Vanessa Laine Bryant), start time, 2001)*. A qualifier has a key and a value, similar to attribute.

Popular knowledge bases usually consist of millions of entities and triples, such as well-known Freebase [7] and Wikidata [43]. If we directly take them as our knowledge base, most of entities may be never used in questions and the huge amount of knowledge may cause both time and space issues. So we extracted a subset of Wikidata as our knowledge base.

Specifically, we took the entities of FB15k-237 [41], a popular subset of Freebase, as seeds, and then aligned them with Wikidata via Freebase IDs[2], so that we could extract their rich literal and qualifier knowledge from Wikidata. Besides, we added 3,000 other entities with the same name as one of FB15k-237 entities, to further increase the difficulty of disambiguation. For the relational knowledge, we manually merged the predicates of FB15k-237 (e.g., */people/person/spouse_s./people/marriage/spouse*) and Wikidata (e.g., *spouse*), obtaining 363 predicates totally. Finally, we manually filtered out useless attributes (e.g., about images and Wikidata pages) and entities (i.e., never used in triples). The statistics of our final knowledge base is listed in Table 2.

Table 2: Statistics of our knowledge base. Top section are the numbers of concepts, entities, unique entity names, predicates, and attributes. Bottom section are the numbers of different types of knowledge.

# Con.	# Ent.	# Name	# Pred.	# Attr.
794	16,960	14,471	363	846

# Relational	# Literal	# Qualifier
415,334	174,539	309,407

4 KQA PRO CONSTRUCTION

To build KQA Pro dataset, we first generate textual questions, corresponding SPARQLs, and corresponding Programs with novel compositional strategies, and then rewrite textual questions into natural language via crowdsourcing. In Section 4.1 we introduce the generation strategies taking the textual questions as example, and then we describe how to paraphrase generated questions in Section 4.2. In Section 4.3 we introduce how to obtain the SPARQL and Program for each textual question.

4.1 Question Generation Strategies

To generate diverse complex questions in a scalable manner, we propose to divide the generation into two stages: locating and asking. In **locating stage** we describe a single entity or an entity set with various restrictions, while in **asking stage** we query specific information about the target entity or entity set. We define several strategies and templates for each stage. By sampling from them and composing the two stages, we can generate large-scale yet diverse questions with a small number of templates. Along with the generation and combination of textual questions, corresponding SPARQLs and Programs are produced at the same time, of which the details will be introduced in Section 4.3. Fig. 2 gives an example of our generation process.

Our generated instance consists of five elements: question, SPARQL, Program, 10 answer choices, and the golden answer. The choices are selected by executing an abridged SPARQL, which randomly drops one or more clauses from the complete SPARQL. With these choices that partially satisfy conditions of the question, our KQA Pro can support the **multiple-choice setting**, which is easier than **open-ended setting**, i.e., predicting answer from scratch, but is still challenging.

We randomly generate lots of questions, and only preserve those with unique answer. For example, since *Shanghai* has different populations in different years, we will drop questions like *What is the population of Shanghai*, unless the time constraint (e.g., *in 2016*) is specified. For an entity set including multiple entities, we do not ask its member names. Instead, we ask its size or select one target from it. As a result, our dataset can easily take accuracy as the metric, which we believe will benefit the development and comparison of models.

4.1.1 Locating Stage. Locating stage is to find the target entity or entity set for the question from KB. In terms of the KB structure, we propose 7 strategies for locating stage. The top section of Table 3 shows these strategies, their templates, and examples. We can fill the placeholders of templates by sampling from the KB to randomly locate an entity set. Besides, we can find a condition that uniquely locates a single entity by going through its relevant knowledge.

[1]Wikidata also has other types like geographical and time. We omit them for simplicity and leave them for future work.

[2]Wikidata provides the Freebase ID for most of its entities, but the predicates are not aligned.

Table 3: Templates and examples of our locating stage and asking stage. Placeholders in template have specific implication: *<E>*-description of an entity or entity set; *<C>*-concept; *<K>*-attribute key; *<op>*-operator, selected from {=, !=, <, >}; *<V>*-attribute value; *<QK>*-qualifier key; *<QV>*-qualifier value; *<P>*-predicate description, e.g., *was born in.*

Strategy	Template	Example
Locating Stage		
Entity Name	-	*Yao Ming*
Concept Name	*<C>*	*basketball players*
Concept + Literal	*the <C> whose <K> is <op> <V> (<QK> is <QV>)*	*the city whose population is greater than 23,000,000 (point in time is 2016)*
Concept + Relational	*the <C> that <P> <E> (<QK> is <QV>)*	*the basketball player that was born in Shanghai*
Recursive Multi-Hop	unfold *<E>* in a *Concept + Relational* description	*the basketball player that was born in the city whose population is greater than 23,000,000 (point in time is 2016)*
Intersection	Condition 1 *and* Condition 2	*the humans whose height is greater than 190 centimetres and less than 220 centimetres*
Union	Condition 1 *or* Condition 2	*the humans that were born in Shanghai or New York*
Asking Stage		
QueryName	*What/Who is <E>*	*Who is the basketball player whose height is equal to 229 centimetres?*
Count	*How many <E>*	*How many basketball players that were born in Shanghai?*
QueryAttribute	*For <E>, what is his/her/its <K> (<QK> is <QV>)*	*For Shanghai, what is its population (point in time is 2016)?*
Relation	*What is the relation from <E1> to <E2>*	*What is the relation from Kobe Bryant to United States of America?*
SelectAmong	*Among <E>, which one has the largest/smallest <K>*	*Among basketball players, which one has the largest mass?*
SelectBetween	*Which one has the larger/smaller <K>, <E1> or <E2>*	*Which one has the larger mass, Kobe Bryant or Yao Ming?*
Verify	*For <E>, is his/her/its <K> <op> <V> (<QK> is <QV>)*	*For the human that is the spouse of Vanessa Laine Bryant, is his/her height greater than 180 centimetres?*
QualifierLiteral	*For <E>, his/her/its <K> is <V>, what is the <QK>*	*For Shanghai, its population is 24,152,700, what is the point in time?*
QualifierRelational	*<E1> <P> <E2>, what is the <QK>*	*Kobe Bryant is the spouse of Vanessa Laine Bryant, what is the start time?*

An entity belongs to all its ancestor concepts, so a high-level concept like *athlete* will locate more entities than a low-level one like *basketball player*.

In *Concept + Literal*, we support quantitative comparisons of 4 operations: equal, not equal, less than, and greater than, indicated by "*<op>*" of the template. For convenience, we only allow comparisons between values with the same unit. For temporal comparison, we also support 4 operations: in (e.g., 01/01/1990 in 1990), not in, before, and after. In *Concept + Literal* and *Concept + Relational*, there are optional qualifier restrictions, indicated by "*(<QK> is <QV>)*" of the templates, which can narrow the located entity set.

In *Recursive Multi-Hop*, we replace the entity of a relational condition with a more detailed description, which should uniquely locate the certain entity. For example, suppose *Shanghai* can be uniquely located by *the city whose population is greater than 23,000,000 (point in time is 2016)*, then we can unfold it in *the basketball player that was born in Shanghai*, leading to a 2-hop condition shown in Table 3. If we replace *Shanghai* with another relational condition, such as *the city that is the financial center of China*, then we can unfold *China* recursively and get description with more hops.

4.1.2 Asking Stage. Asking stage is to determine what we want to ask about the target entity or entity set. To support the diversity of the questions, we propose 9 strategies. The bottom section of Table 3 shows their templates and examples.

QueryName: Asking the entity name. It only receives a unique entity. An entity set with multiple members is not allowed.
Count: Asking the size of an entity set.
QueryAttribute: Asking the attribute value. Qualifier restrictions are optionally appended to disambiguate the answer.
Relation: Asking the relation between two entities.
SelectAmong: Selecting the entity with the maximum or minimum attribute value from an entity set. For simplicity, the selected attribute is not restricted by qualifier knowledge in this question type. It is similar to *argmax* and *argmin* operations in λ-DCS.
SelectBetween: Selecting the entity with the larger or smaller attribute value from two entities. Similar to SelectAmong, there is no qualifier restrictions.
Verify: Asking whether the attribute value satisfies the condition. Qualifier restrictions are optionally appended to disambiguate the answer.
QualifierLiteral: Asking qualifier knowledge about a literal triple.
QualifierRelational: Asking qualifier knowledge about a relational triple.

4.2 Question Paraphrasing and Evaluation

After large-scale generation, we release the generated questions on Amazon Mechanical Turk (AMT) and ask the workers to paraphrase them without changing the original meaning. For the convenience of paraphrasing, we visualize the Program flowcharts like Fig. 1 to help workers understand complex questions. Besides, we manually annotate textual templates for each attribute and each predicate to make sure the generated description is fluent. For example, for the predicate *member of*, we create two templates, *S is the member of O* and *O has a member S*, to describe the triple *(S, member of, O)* forward and backward. We allow workers to mark a question as confusing if they cannot understand it or find some logical errors in it. These instances will be removed from our dataset.

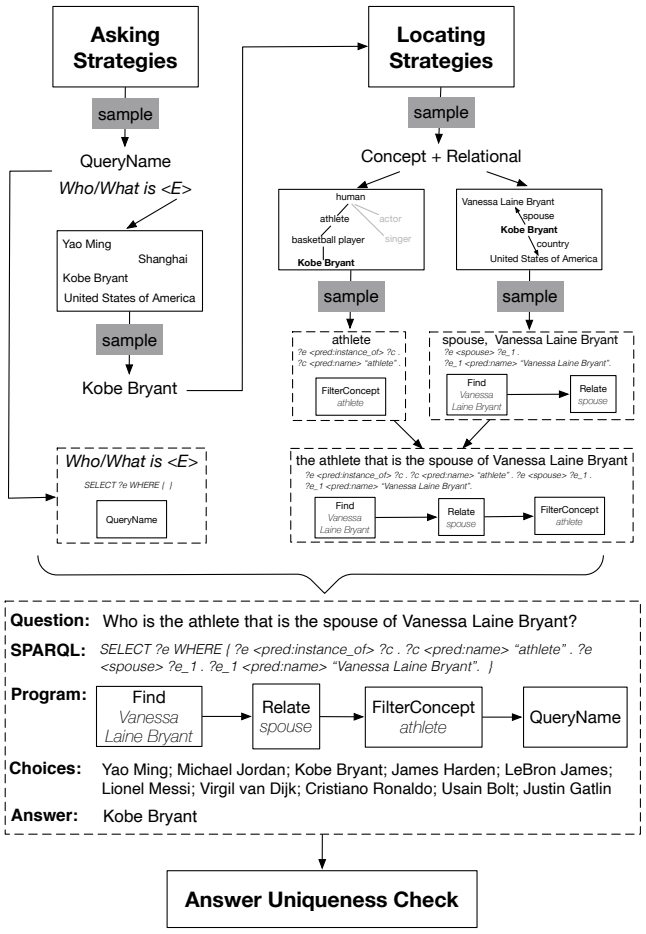


Figure 2: Process of our question generation. First, we sample a question type from asking strategies and sample a target entity from KB. Next, we sample a locating strategy and sample detailed conditions to describe the target entity. Finally, we combine intermediate snippets into the complete question and check whether the answer is unique. Note that the snippets of textual question, SPARQL, and Program are operated simultaneously.

After paraphrasing, we evaluate the quality by 5 other workers. They are asked to check whether the paraphrase keeps the original meaning and give an additional fluency rating from 1 to 5. We reject those paraphrases which fall into one of following cases: (1) marked as different from the original template-formed question by more than 2 workers; (2) whose average fluency rating is lower than 3; (3) having a very small edit distance with the template-formed question.

4.3 Generation of SPARQL and Program

Along with the generation and combination of textual question, the corresponding SPARQL and Program are produced meanwhile. That is, in each strategy of Table 3, when generating textual snippet from template, we also generate the snippet of SPARQL and Program using predefined rules. When combining textual snippets into a complete question, we also combine the snippets of SPARQL and Program.

Fig. 2 gives an example. The asking stage samples the strategy *QueryName* and samples *Kobe Bryant* as the target entity, whose corresponding textual description, SPARQL, and Program is "*Who is <E>*", "*SELECT ?e WHERE { }*", "*QueryName*", respectively. Then the locating stage samples a strategy, *Concept + Relational*, to locate *Kobe Bryant*. For the concept part, *athlete* is sampled, and the corresponding textual description, SPARQL, and Program is "*athlete*", "*?e <pred:instance_of> ?c . ?c <pred:name> "athlete" .*.", "*FilterConcept(athlete)*", respectively. For the relation part, *(spouse, Vanessa Laine Bryant)* is sampled, and the corresponding textual description, SPARQL, and Program is "*the spouse of Vanessa Laine Bryant*", "*?e <spouse> ?e_1 . ?e_1 <pred:name> "Vanessa Laine Bryant" .*.", "*Find(Vanessa Laine Bryant)* $\rightarrow$ *Relate(spouse)*", respectively. The locating stage combines the concept and the relation, obtaining the entity description "*the athlete that is the spouse of Vanessa Laine Bryant*" and the corresponding SPARQL and Program. Finally, we combine the results of the two stages and output the complete question.

Following we introduce some implementation details about our SPARQLs and Programs.

4.3.1 SPARQLs. We build a SPARQL engine with Virtuoso [3] to execute generated SPARQLs. To denote qualifiers, we create a virtual node for those literal and relational triples. For example, to denote the start time of *(Kobe Bryant, spouse, Vanessa Laine Bryant)*, we create a node *_BN* which connects to the subject, the predicate, and the object with three special edges, and then add *(_BN, start time, 2001)* into the graph. Similarly, we use virtual node to represent the attribute value of number type, which has an extra unit. For example, to represent the height of *Yao Ming*, we need *(Yao Ming, height, _BN), (_BN, value, 229), (_BN, unit, centimetre).*

4.3.2 Functions and Programs. Functions are basic operations on the knowledge base and each function receives a fixed number of parameters. In terms of basic reasoning operations over KB, we design 27 functions and describe the reasoning process of our questions by composing them into Programs. Note that a question may have multiple solutions, and our Program just depicts one feasible process. Compared with SPARQL, Program divides the question into multiple steps, which is easier to be interpreted by human. Besides better interpretability, Program allows human to control the model behaviour better. When the computer gives a wrong answer, we can easily locate the error by checking the outputs of intermediate functions.

Each function has two kinds of input. The textual inputs come from the question, and the functional inputs come from the output of previous functions. Here we introduce two functions, leaving the complete instructions in Appendix:

Relate. Two textual inputs: predicate and direction (i.e., *forward* or *backward*, meaning the output is object or subject). One functional input: a unique entity. Output: a set of entities that hold the specific relation with the input entity. For example, in Question 1 of Fig. 1,

[3] https://github.com/openlink/virtuoso-opensource

the function *Relate(place of birth, forward)* locates *Shanghai* from *Yao Ming* (the direction is omitted for simplicity in the figure).
SelectBetween. Two textual inputs: attribute key and operation. Two functional inputs: two unique entities. Output: the entity with the greater or smaller attribute value. For example, in Question 2 of Fig. 1, the function *Select(height, greater)* selects the taller one from *Yao Ming* and *Kobe Bryant* (the function name is simplified in the figure).

5 DATASET ANALYSIS

Our KQA Pro dataset consists of 117,970 instances. Fig. 3 shows some examples. We split it to training/validation/test set by 8/1/1, resulting in three sets including 94,376/11,797/11,797 instances. There are 24,724 unique answers in KQA Pro. We show the top 20 most frequent answers and their fractions in Fig. 4(a). "yes" and "no" are the most frequent answers, because they cover all questions of type *Verify*. "0", "1", "2", "3", and other quantity answers are for questions of type *Count*, which accounts for 11.5% according to Fig. 4(b). About 30% answers of the test set are not visible in the training set, implying that the model must have the ability of zero-shot learning to tackle our dataset.

Fig. 4(b) shows the question type distribution of KQA Pro. Within the 9 types shown in the bottom section of Table 3, *SelectAmong* accounts for the least fraction (4.6%), while others account for more or less than 10%. Taking the questions that locate the target entity by the link from another entity or from some attribute values as multi-hop questions, they cover 73.7% of KQA Pro, much more than single-hop questions that locate the target directly by the entity name. Fig. 4(d) shows the Program length distribution. Most of our problems (28.42%) can be solved by 4 functional steps. Some extreme complicated ones (1.24%) need more than 10 steps. We compare the question length distribution of different Complex KBQA datasets in Fig. 4(e). We observe that on average, our KQA Pro has longer questions than others, implying that our dataset is more challenging. In KQA Pro, the average length of questions/Programs/SPARQLs is 14.95/4.79/35.52 respectively.

6 EXPERIMENTS

Existing models of Complex KBQA fall into two categories: information retrieval based and semantic parsing based models.

The models of information retrieval focus on finding the answer from KB by integrating the information of KB and questions. **KVMemNet** [28] is a well-known model of this type. It reorganizes the knowledge into a memory of key-value pairs, and iteratively reads memory to update its query vector. Some other works aim to tackle multi-hop questions by path search [32, 49, 52]. They usually start from a topic entity and predict a sequential relation path to find the target entity. These models are inherently limited to relational knowledge and unable to handle quantitative questions. **SRN** [32] is the state-of-the-art among this line of work. **EmbedKGQA** [35] is another model aimed at multi-hop questions. It incorporates knowledge embeddings to improve the reasoning performance and achieves state-of-the-art on MetaQA. Besides, due to the natural graph structure of KB, it is intuitive to apply graph neural networks to tackle the Complex KBQA task. We explore **RGCN** [36], a variant of graph convolutional networks which considers edge labels, on our KQA Pro.

The models of semantic parsing focus on translating the question into corresponding logical form and then executing the rule engine to get the answer. There are two commonly-used logical forms: query graph (e.g., SPARQL) and Program. Many previous works focus on the weakly-supervised setting, which learns the query graph parser [6, 47] or the Program parser [2, 17, 26, 33] in terms of question-answer pairs. Since KQA Pro provides the annotations of SPARQL and Program, we directly learn our parsers using supervised learning. We regard the semantic parsing as a sequence-to-sequence task, since our Program and SPARQL can be easily converted to the sequential format.[4] We explore the widely-used sequence-to-sequence model—**RNN** with attention mechanism [13], and the recently-proposed pretrained model—**BART** [25]—a variant of BERT [11] specially for sequence-to-sequence tasks, as our SPARQL and Program parsers.

To compare machine with **Human**, we sample 200 instances from the test set, and ask human experts to answer them by searching our knowledge base.

6.1 Implementation Details

We used the optimizer Adam [24] for all models. The learning rate was initialized as 0.001 and decreased to 0.00001 during the learning process. We trained models for 100 epochs and selected the checkpoint which performed best on the validation set to report its performance on the test set.
KVMemNet. For literal and relational knowledge, we concatenated the subject and the attribute/predicate as the memory key, e.g., "Kobe Bryant spouse", leaving the object as the memory value. For high-level knowledge, we concatenated the fact and the qualifier key as the memory key, e.g., "Kobe Bryant spouse Vanessa Laine Bryant start time". For each question, we pre-selected a small subset of the KB as its relavant memory. Following [28], we retrieved 1,000 key-value pairs where the key shares at least one word with the question with frequency < 1000 (to ignore stop words). KVMemNet iteratively updates a query vector by reading the memory attentively. In our experiment we set the update steps to 3.
SRN. SRN can only handle relational knowledge. It must start from a topic entity and terminate with a predicted entity. So we filtered out questions that contain literal knowledge or qualifier knowledge, retaining 5,004 and 649 questions as its training set and test set. Specifically, we retained the questions with *Find* as the first function and *QueryName* as the last function. The textual input of the first *Find* was regarded as the topic entity and was fed into the model during both training and testing phase.
EmbedKGQA. EmbedKGQA utilizes knowledge graph embedding to improve multi-hop reasoning. To adapt to existing knowledge embedding techniques, we added virtual nodes to represent the qualifier knowledge of KQA Pro (see Section 4.3.1). Different from SRN, we applied EmbedKGQA on the entire KQA Pro dataset, because its classification layer is more flexible than SRN and can predict answers outside the entity set. The topic entity of each

[4] The structure of our Programs is binary tree. We convert it to sequence by post-order traversal.

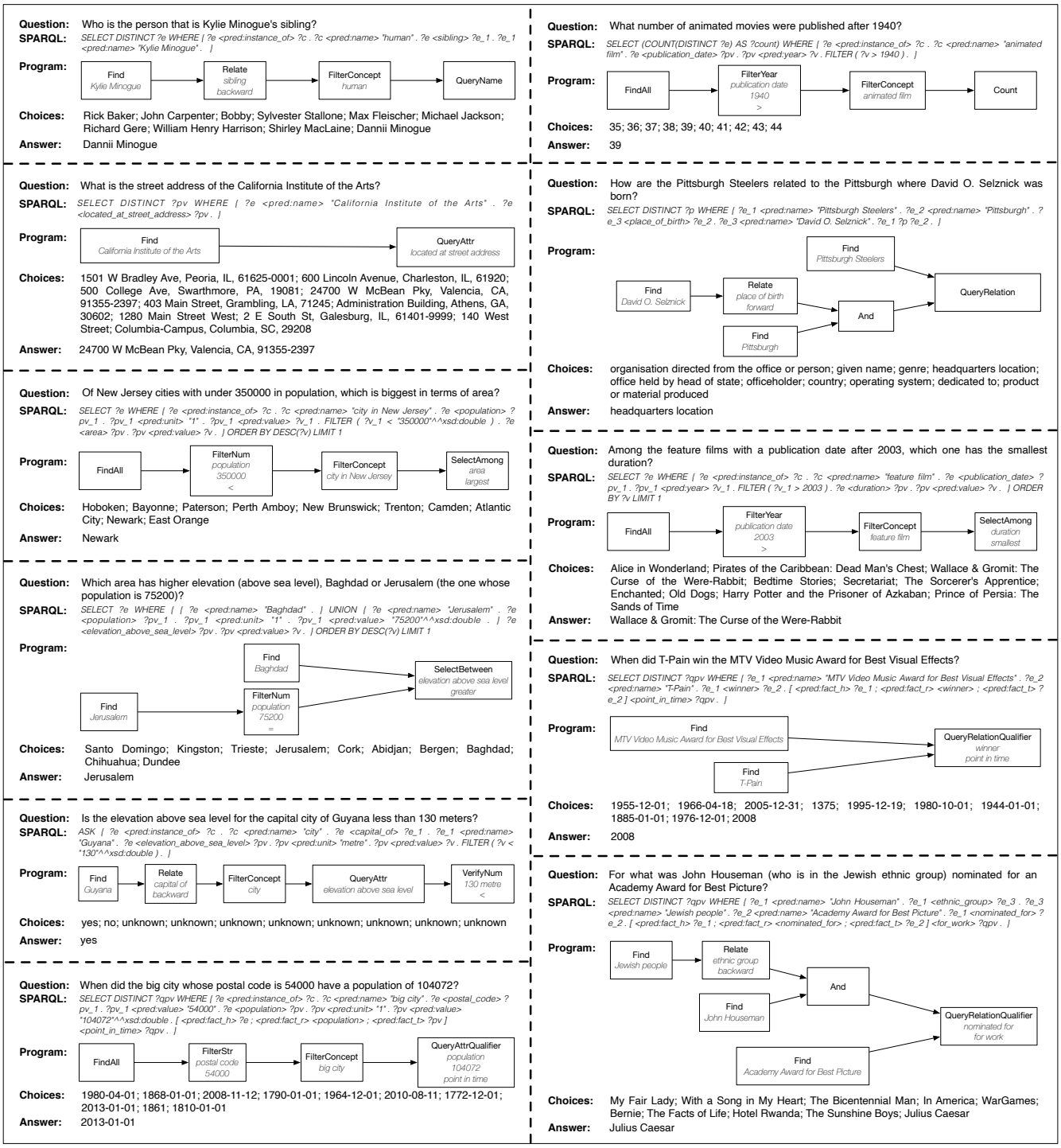
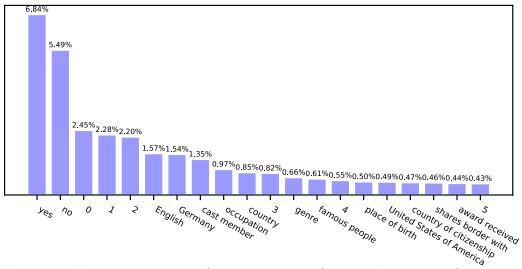
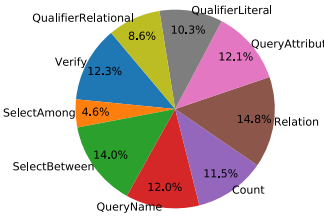


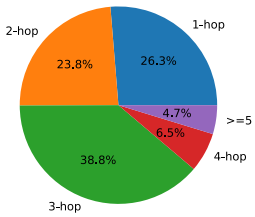
Figure 3: Examples of KQA Pro. In KQA Pro, each instance consists of 5 components: the textual question, the corresponding SPARQL, the corresponding Program, 10 candidate choices, and the golden answer. We provide at least one example for each type of our 9 asking strategies. Choices are separated by semicolons in this figure. For questions of *Verify* type, the choices are composed of "yes", "no", and 8 special token "unknown" for padding.



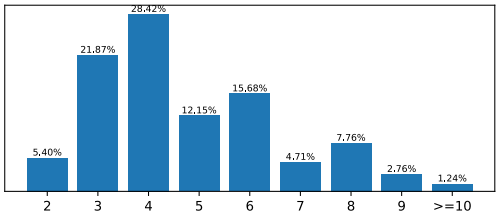
(a) Top 20 most occurring answers in KQA Pro. The most frequent one is "yes", which is the answer of about half of type Verify.



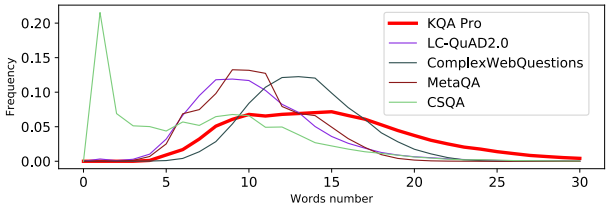
(b) Distribution of 9 different types of questions.



(c) Distribution of question hops. 73.7% of our questions require multiple-hops.



(d) Distribution of program lengths.



(e) Question length distribution of Complex KBQA datasets. We can see that KQA Pro questions have a wide range of lengths and are longer on average than all others.

Figure 4: Question, answer, and Program statistics of KQA Pro.

Table 4: Accuracy of different models on KQA Pro test set. We categorize the test questions to measure fine-grained ability of models. Specifically, *Multi-hop* means multi-hop questions, *Qualifier* means questions containing qualifier knowledge, *Comparison* means quantitative or temporal comparison between two or more entities, *Logical* means logical union or intersection, *Count* means questions that ask the number of target entities, *Verify* means questions that take "yes" or "no" as answer, *Zero-shot* means questions whose answer is not seen in the training set. For those methods that output a probability distribution over the answer vocabulary, we take the 10 candidate choices as a mask and get results of the multiple-choice setting.

Model	**Overall**	Multi-hop	Qualifier	Comparison	Logical	Count	Verify	Zero-shot
Open-Ended Setting								
KVMemNet	16.61	16.50	18.47	1.17	14.99	27.31	54.70	0.06
SRN	-	12.33	-	-	-	-	-	-
EmbedKGQA	28.36	26.41	25.20	11.93	23.95	32.88	61.05	0.06
RGCN	35.07	34.00	27.61	30.03	35.85	41.91	65.88	0.00
RNN SPARQL	41.98	36.01	19.04	66.98	37.74	50.26	58.84	26.08
RNN Program	43.85	37.71	22.19	65.90	47.45	50.04	42.13	34.96
BART SPARQL	89.68	88.49	83.09	96.12	88.67	85.78	92.33	87.88
BART Program	87.17	85.57	80.57	93.83	86.33	84.20	84.94	86.29
Multiple-Choice Setting								
KVMemNet	39.15	36.78	50.09	18.71	38.61	34.69	56.08	6.73
EmbedKGQA	45.19	42.39	54.66	23.01	42.36	36.34	61.95	8.88
RGCN	53.75	51.70	58.24	48.64	55.88	47.93	66.57	1.46
Human Performance								
Human	97.50	97.24	95.65	100.00	98.18	83.33	95.24	100.00

question was extracted from the golden Program and then fed into the model during both training and testing.

RGCN. To build the graph, we took entities as nodes, connections between them as edges, and predicates as edge labels. We concatenated the literal attributes of an entity into a sequence as the node description. For simplicity, we ignored the qualifier knowledge.

Given a question, we first initialized node vectors by fusing the information of node descriptions and the question, then conducted RGCN to update the node features, and finally aggregated features of nodes and the question to predict the answer via a classification layer. Our RGCN implementation is based on DGL,[5] a high performance Python package for deep learning on graphs. Due to the memory limit, we set the graph layer to 1 and set the hidden dimension of nodes and edges to 32.

RNN-based Program and SPARQL Parsers. For Program prediction, we first parsed the question to the sequence of functions, and then predicted textual inputs for each function. We used Gated Recurrent Unit (GRU) [9, 10], a well-known variant of RNNs, as our encoder of questions and decoder of functions. Attention mechanism [13] was applied by focusing on the most relavant question words when predicting each function and each textual input. The SPARQL parser used the same encoder-decoder structure to produce SPARQL token sequences. We tokenized the SPARQL query by delimiting spaces and some special punctuation symbols.

BART-based Program and SPARQL Parsers. BART is a pretrained sequence-to-sequence model based on Transformer [42], achieving state-of-the-art performance on a range of language comprehension and generation tasks. We used the widely-used online implementation[6] and finetuned the pretrained model on our NLQ-to-SPARQL and NLQ-to-Program corpus. For Program learning, we concatenated textual inputs with functions to obtain a totally sequential format of Program, e.g., Question 2 of Fig. 1 is converted to the sequence "*Find(Yao Ming), Find(Vanessa Laine Bryant), Relate(spouse, backward), SelectBetween(height, greater)*".

Table 5: State-of-the-art models of Complex KBQA and their performance on different datasets. WebQSP is short for WebQuestionSP. SRN's result on KQA Pro, 12.33%, is obtained on questions about only relational knowledge.

Model	MetaQA			WebQSP	KQA Pro
	1-hop	2-hop	3-hop		
KVMemNet	96.2	82.7	48.9	46.7	16.61
SRN	97.0	95.1	75.2	-	(12.33)
EmbedKGQA	97.5	98.8	94.8	66.6	28.36

6.2 Experimental Results

As each question of KQA Pro has a unique answer, which may be an entity name, an attribute value, a predicate name, a number, "yes", or "no", we directly use accuracy as the evaluation metric.

Table 4 shows overall and detailed accuracy of aforementioned models. When not using SPARQL and Program annotations, state-of-the-art models of Complex KBQA perform very poor in our dataset. KVMemNet achieves an overall 16.61%, EmbedKGQA achieves 28.36%, and RGCN's 35.07% is the best among them. Especially for zero-shot questions, these models all have a close to zero performance. These results demonstrate the weakness of information retrieval based models, i.e., it is difficult for them to handle complicated questions and unseen answers. Table 5 shows their performance on MetaQA and WebQuestionSP. We can see that their performance drops significantly on KQA Pro. EmbedKGQA achieves 94.8% on MetaQA 3-hop questions, while only 28.36% on KQA Pro test set, demonstrating that KQA Pro is much more challenging. Besides the difficulty of our questions, another reason of such a low accuracy is that these models mostly focus on relational knowledge and are weak in other reasoning skills.

Table 6: Test accuracy using partial training set. The performance drops along with the reducing of training data.

Model	$\frac{1}{10}$	$\frac{1}{5}$	$\frac{1}{3}$	$\frac{1}{2}$	1
BART SPARQL	79.22	84.50	86.83	88.62	89.68
BART Program	75.53	81.25	83.83	85.58	87.17

When using SPARQL and Program annotations, RNN-based semantic parsing models obtain better results than information retrieval models. Specifically, RNN-based SPARQL parser achieves an overall 41.98% accuracy, and RNN-based Program parser achieves 43.85%. When injecting our large-scale SPARQL and Program annotations into pretrained models, we get surprising results. BART-based SPARQL and Program parsers achieve up to 89.68% and 87.17% accuracy, which is close to the human expert performance, 97.50%. Despite of the complexity of these two logical forms and the difficulty of our questions, BART can still have a so excellent performance, implying its powerful learning ability. To explore the influence of the scale of training data, we reduce the training set from the original scale to its $\frac{1}{10}$, remaining 9,437 instances. Results are shown in Table 6. As the decrease of training data, the performance drops gradually, implying that the parsers can perform even better if more annotations are fed in.

To further understand the quality of logical forms predicted by the BART parser, we show two cases in Fig. 5, for which the SPARQL and Program parsers both give wrong predictions. For the first question, the natural language description omits the *relative* relation between the two mentioned people. As a result, the SPARQL parser cannot recognize it as a qualifier question and it directly queries their relation. The Program parser performs correctly in function prediction, but gives a wrong function input, i.e., the first input of *QueryRelationQualifier*. The second question is paraphrased very well, fluent and keeping the original meaning. The SPARQL parser fails to understand *prior to David Lloyd George* and gives a totally wrong prediction for this part. The Program parser gives a function prediction which is semantically correct but very different from our generated golden one. It is a suprising result, revealing that the Program parser can understand the semantics and learn multiple solutions for each question, similar to the learning process of humans. We manually correct the errors of predicted SPARQLs and Programs and mark them in red. Compared to SPARQLs, Programs are easier to be understood and more friendly to be modified.

7 CONCLUSIONS

We introduce KQA Pro, a new benchmark of Complex KBQA with following features: 1) with additional supervision for each question,

[5] https://github.com/dmlc/dgl
[6] https://github.com/huggingface/transformers

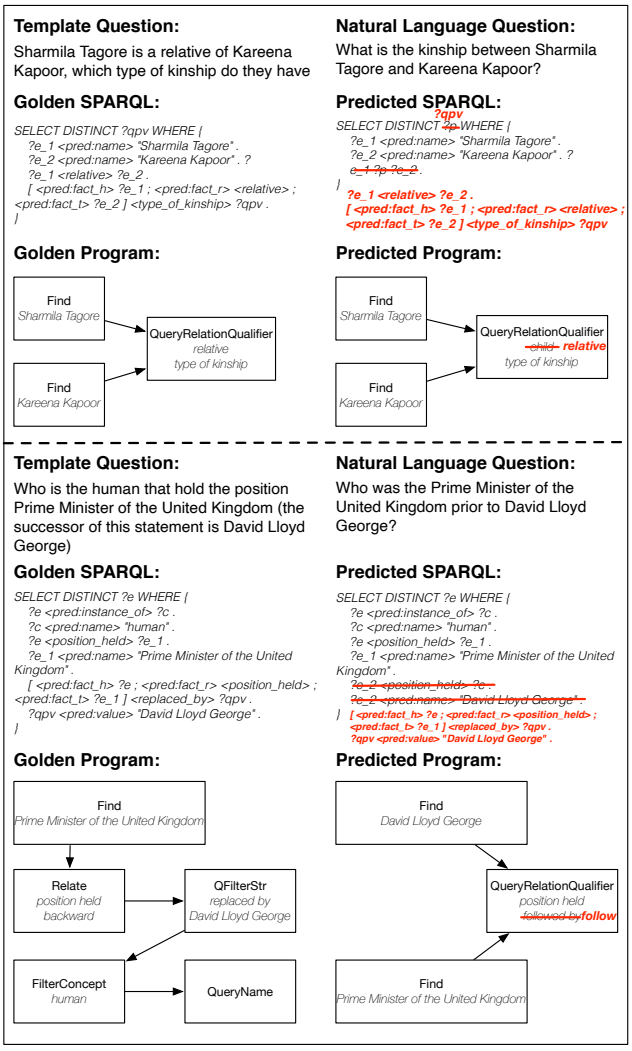


Figure 5: Predicted SPARQLs and Programs by BART. Besides the natural language questions, we also show the corresponding template questions before human rewriting. We mark the error corrections of the wrong predictions in red.

i.e., SPARQLs and Programs; 2) large-scale and diverse, with about 120K natural questions. We conduct extensive experiments on KQA Pro and obtain some very useful insights. When we train state-of-the-art KBQA models using only QA pairs, machines perform very poor, demonstrating the challenge of our questions. However, when we incorporate the intermediate supervision and feed parallel corpora into pretrained sequence-to-sequence models, machines give an excellent performance, specifically, achieving near 90% answering accuracy. These results validate our idea that intermediate supervision is necessary for machines to learn complicated reasoning capability. Besides, they also demonstrate the great power of current pretrained language models. Machines can translate natural questions to logical forms very well as long as we finetune pretrained models on a large parallel corpus like KQA Pro. We hope our findings can inspire both academic and industrial world so that we can finally close the gap between human language and machine language in the future.

REFERENCES

[1] Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. 2016. Neural module networks. In *CVPR*.

[2] Ghulam Ahmed Ansari, Amrita Saha, Vishwajeet Kumar, Mohan Bhambhani, Karthik Sankaranarayanan, and Soumen Chakrabarti. 2019. Neural program induction for KBQA without gold programs or query annotations. In *IJCAI*.

[3] Yoav Artzi, Nicholas FitzGerald, and Luke S Zettlemoyer. 2013. Semantic Parsing with Combinatory Categorial Grammars. *ACL (Tutorial Abstracts)* 3 (2013).

[4] Michael Azmy, Peng Shi, Jimmy Lin, and Ihab Ilyas. 2018. Farewell freebase: Migrating the simple questions dataset to dbpedia. In *COLING*.

[5] Laura Banarescu, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer, and Nathan Schneider. 2013. Abstract meaning representation for sembanking. In *Proceedings of the 7th linguistic annotation workshop and interoperability with discourse*. 178–186.

[6] Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. 2013. Semantic parsing on freebase from question-answer pairs. In *EMNLP*.

[7] Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. 2008. Freebase: a collaboratively created graph database for structuring human knowledge. In *SIGMOD*.

[8] Antoine Bordes, Nicolas Usunier, Sumit Chopra, and Jason Weston. 2015. Large-scale simple question answering with memory networks. *arXiv preprint arXiv:1506.02075* (2015).

[9] Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. 2014. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259* (2014).

[10] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555* (2014).

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL-HLT*.

[12] Ming Ding, Chang Zhou, Qibin Chen, Hongxia Yang, and Jie Tang. 2019. Cognitive Graph for Multi-Hop Reading Comprehension at Scale. In *ACL*.

[13] Li Dong and Mirella Lapata. 2016. Language to Logical Form with Neural Attention. In *ACL*.

[14] Dheeru Dua, Yizhong Wang, Pradeep Dasigi, Gabriel Stanovsky, Sameer Singh, and Matt Gardner. 2019. DROP: A Reading Comprehension Benchmark Requiring Discrete Reasoning Over Paragraphs. In *NAACL-HLT*.

[15] Mohnish Dubey, Debayan Banerjee, Abdelrahman Abdelkawi, and Jens Lehmann. 2019. Lc-quad 2.0: A large dataset for complex question answering over wikidata and dbpedia. In *International Semantic Web Conference*. Springer, 69–78.

[16] Yuwei Fang, Siqi Sun, Zhe Gan, Rohit Pillai, Shuohang Wang, and Jingjing Liu. 2019. Hierarchical Graph Network for Multi-hop Question Answering. *arXiv preprint arXiv:1911.03631* (2019).

[17] Daya Guo, Duyu Tang, Nan Duan, Ming Zhou, and Jian Yin. 2018. Dialog-to-action: Conversational question answering over a large-scale knowledge base. In *Advances in Neural Information Processing Systems*.

[18] Nitish Gupta, Kevin Lin, Dan Roth, Sameer Singh, and Matt Gardner. 2019. Neural Module Networks for Reasoning over Text. *arXiv preprint arXiv:1912.04971* (2019).

[19] John Holt. 2017. *How children learn*. Hachette UK.

[20] Xiao Huang, Jingyuan Zhang, Dingcheng Li, and Ping Li. 2019. Knowledge graph embedding based question answering. In *WSDM*.

[21] Drew A Hudson and Christopher D Manning. 2019. GQA: A New Dataset for Real-World Visual Reasoning and Compositional Question Answering. *arXiv preprint arXiv:1902.09506* (2019).

[22] Kelvin Jiang, Dekun Wu, and Hui Jiang. 2019. FreebaseQA: A New Factoid QA Data Set Matching Trivia-Style Question-Answer Pairs with Freebase. In *NAACL-HLT*.

[23] Justin Johnson, Bharath Hariharan, Laurens van der Maaten, Li Fei-Fei, C Lawrence Zitnick, and Ross Girshick. 2017. Clevr: A diagnostic dataset for compositional language and elementary visual reasoning. In *CVPR*.

[24] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[25] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. 2019. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461* (2019).

[26] Chen Liang, Jonathan Berant, Quoc Le, Kenneth Forbus, and Ni Lao. 2017. Neural Symbolic Machines: Learning Semantic Parsers on Freebase with Weak Supervision. In *ACL*.

[27] Percy Liang. 2013. Lambda dependency-based compositional semantics. *arXiv preprint arXiv:1309.4408* (2013).

Table 7: Details of our 27 functions. Each function has 2 kinds of inputs: the functional inputs come from the output of previous functions, while the textual inputs come from the question.

Function	**Functional Inputs × Textual Inputs → Outputs**	**Description**	**Example** (only show textual inputs)
FindAll	() × () → (*Entities*)	Return all entities in KB	-
Find	() × (*Name*) → (*Entities*)	Return all entities with the given name	*Find(Kobe Bryant)*
FilterConcept	(*Entities*) × (*Name*) → (*Entities*)	Find those belonging to the given concept	*FilterConcept(athlete)*
FilterStr	(*Entities*) × (*Key, Value*) → (*Entities, Facts*)	Filter entities with an attribute condition of string type, return entities and corresponding facts	*FilterStr(gender, male)*
FilterNum	(*Entities*) × (*Key, Value, Op*) → (*Entities, Facts*)	Similar to *FilterStr*, except that the attribute type is number	*FilterNum(height, 200 centimetres, >)*
FilterYear	(*Entities*) × (*Key, Value, Op*) → (*Entities, Facts*)	Similar to *FilterStr*, except that the attribute type is year	*FilterYear(birthday, 1980, =)*
FilterDate	(*Entities*) × (*Key, Value, Op*) → (*Entities, Facts*)	Similar to *FilterStr*, except that the attribute type is date	*FilterDate(birthday, 1980-06-01, <)*
QFilterStr	(*Entities, Facts*) × (*QKey, QValue*) → (*Entities, Facts*)	Filter entities and corresponding facts with a qualifier condition of string type	*QFilterStr(language, English)*
QFilterNum	(*Entities, Facts*) × (*QKey, QValue, Op*) → (*Entities, Facts*)	Similar to *QFilterStr*, except that the qualifier type is number	*QFilterNum(bonus, 20000 dollars, >)*
QFilterYear	(*Entities, Facts*) × (*QKey, QValue, Op*) → (*Entities, Facts*)	Similar to *QFilterStr*, except that the qualifier type is year	*QFilterYear(start time, 1980, =)*
QFilterDate	(*Entities, Facts*) × (*QKey, QValue, Op*) → (*Entities, Facts*)	Similar to *QFilterStr*, except that the qualifier type is date	*QFilterDate(start time, 1980-06-01, <)*
Relate	(*Entity*) × (*Pred, Dir*) → (*Entities, Facts*)	Find entities that have a specific relation with the given entity	*Relate(capital, forward)*
And	(*Entities, Entities*) × () → (*Entities*)	Return the intersection of two entity sets	-
Or	(*Entities, Entities*) × () → (*Entities*)	Return the union of two entity sets	-
QueryName	(*Entity*) × () → (string)	Return the entity name	-
Count	(*Entities*) × () → (number)	Return the number of entities	-
QueryAttr	(*Entity*) × (*Key*) → (*Value*)	Return the attribute value of the entity	*QueryAttr(height)*
QueryAttrUnderCondition	(*Entity*) × (*Key, QKey, QValue*) → (*Value*)	Return the attribute value, whose corresponding fact should satisfy the qualifier condition	*QueryAttrUnderCondition(population, point in time, 2016)*
QueryRelation	(*Entity, Entity*) × () → (*Pred*)	Return the predicate between two entities	*QueryRelation(Kobe Bryant, America)*
SelectBetween	(*Entity, Entity*) × (*Key, Op*) → (string)	From the two entities, find the one whose attribute value is greater or less and return its name	*SelectBetween(height, greater)*
SelectAmong	(*Entities*) × (*Key, Op*) → (string)	From the entity set, find the one whose attribute value is the largest or smallest	*SelectAmong(height, largest)*
VerifyStr	(*Value*) × (*Value*) → (boolean)	Return whether the output of *QueryAttr* or *QueryAttrUnderCondition* and the given value are equal as string	*VerifyStr(male)*
VerifyNum	(*Value*) × (*Value, Op*) → (boolean)	Return whether the two numbers satisfy the condition	*VerifyNum(20000 dollars, >)*
VerifyYear	(*Value*) × (*Value, Op*) → (boolean)	Return whether the two years satisfy the condition	*VerifyYear(1980, >)*
VerifyDate	(*Value*) × (*Value, Op*) → (boolean)	Return whether the two dates satisfy the condition	*VerifyDate(1980-06-01, >)*
QueryAttrQualifier	(*Entity*) × (*Key, Value, QKey*) → (*QValue*)	Return the qualifier value of the fact (*Entity, Key, Value*)	*QueryAttrQualifier(population, 23,390,000, point in time)*
QueryRelationQualifier	(*Entity, Entity*) × (*Pred, QKey*) → (*QValue*)	Return the qualifier value of the fact (*Entity, Pred, Entity*)	*QueryRelationQualifier(spouse, start time)*

[28] Alexander Miller, Adam Fisch, Jesse Dodge, Amir-Hossein Karimi, Antoine Bordes, and Jason Weston. 2016. Key-Value Memory Networks for Directly Reading Documents. In *EMNLP*.

[29] Panupong Pasupat and Percy Liang. 2015. Compositional semantic parsing on semi-structured tables. *arXiv preprint arXiv:1508.00305* (2015).

[30] Panupong Pasupat and Percy Liang. 2016. Inferring logical forms from denotations. *arXiv preprint arXiv:1606.06900* (2016).

[31] Michael Petrochuk and Luke Zettlemoyer. 2018. SimpleQuestions Nearly Solved: A New Upperbound and Baseline Approach. In *EMNLP*.

[32] Yunqi Qiu, Yuanzhuo Wang, Xiaolong Jin, and Kun Zhang. 2020. Stepwise Reasoning for Multi-Relation Question Answering over Knowledge Graph with Weak Supervision. In *WSDM*.

[33] Amrita Saha, Ghulam Ahmed Ansari, Abhishek Laddha, Karthik Sankaranarayanan, and Soumen Chakrabarti. 2019. Complex Program Induction for Querying Knowledge Bases in the Absence of Gold Programs. *Transactions of the Association for Computational Linguistics* (2019).

[34] Amrita Saha, Vardaan Pahuja, Mitesh M Khapra, Karthik Sankaranarayanan, and Sarath Chandar. 2018. Complex sequential question answering: Towards learning to converse over linked question answer pairs with a knowledge graph. In *AAAI*.

[35] Apoorv Saxena, Aditay Tripathi, and Partha Talukdar. 2020. Improving Multi-hop Question Answering over Knowledge Graphs using Knowledge Base Embeddings. In *ACL*.

[36] Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. 2018. Modeling relational data with graph convolutional networks. In *ESWC*.

[37] Jiaxin Shi, Hanwang Zhang, and Juanzi Li. 2019. Explainable and explicit visual reasoning over scene graphs. In *CVPR*.

[38] 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 *EMNLP*.

[39] Yawei Sun, Lingling Zhang, Gong Cheng, and Yuzhong Qu. 2020. SPARQA: Skeleton-Based Semantic Parsing for Complex Questions over Knowledge Bases.. In *AAAI*. 8952–8959.

[40] Alon Talmor and Jonathan Berant. 2018. The Web as a Knowledge-Base for Answering Complex Questions. In *NAACL-HLT*.

[41] Kristina Toutanova, Danqi Chen, Patrick Pantel, Hoifung Poon, Pallavi Choudhury, and Michael Gamon. 2015. Representing text for joint embedding of text and knowledge bases. In *EMNLP*.

[42] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[43] Denny Vrandečić and Markus Krötzsch. 2014. Wikidata: a free collaborative knowledge base. (2014).

[44] Yushi Wang, Jonathan Berant, and Percy Liang. 2015. Building a semantic parser overnight. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. 1332–1342.

[45] Peng Wu, Shujian Huang, Rongxiang Weng, Zaixiang Zheng, Jianbing Zhang, Xiaohui Yan, and Jiajun Chen. 2019. Learning Representation Mapping for Relation Detection in Knowledge Base Question Answering. In *ACL*.

[46] Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W Cohen, Ruslan Salakhutdinov, and Christopher D Manning. 2018. Hotpotqa: A dataset for diverse, explainable multi-hop question answering. *arXiv preprint arXiv:1809.09600* (2018).

[47] 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 *ACL-IJCNLP*.

[48] Wen-tau Yih, Matthew Richardson, Christopher Meek, Ming-Wei Chang, and Jina Suh. 2016. The value of semantic parse labeling for knowledge base question answering. In *ACL*.

[49] Yuyu Zhang, Hanjun Dai, Zornitsa Kozareva, Alexander J Smola, and Le Song. 2018. Variational reasoning for question answering with knowledge graph. In

AAAI.
[50] Wenbo Zhao, Tagyoung Chung, Anuj Goyal, and Angeliki Metallinou. 2019. Simple Question Answering with Subgraph Ranking and Joint-Scoring. In *NAACL-HLT*.
[51] Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103* (2017).
[52] Mantong Zhou, Minlie Huang, and Xiaoyan Zhu. 2018. An Interpretable Reasoning Network for Multi-Relation Question Answering. In *COLING*.

A FUNCTION DETAILS

Table 7 shows our 27 functions and their explanations. Note that we define specific functions for different attribute types (i.e., string, number, date, and year), because the comparison of these types are quite different. Following we explain some necessary items in our functions.

Entities/Entity: *Entities* denotes an entity set, which can be the output or functional input of a function. When the set has a unique element, we get an *Entity*.
Name: A string that denotes the name of an entity or a concept.
Key/Value: The key and value of an attribute.
Op: The comparative operation. It is one of {=, ≠, <, >} when comparing two values, one of {*greater*, *less*} in *SelectBetween*, and one of {*largest*, *smallest*} in *SelectAmong*.
Pred/Dir: The predicate and direction of a relation.
Fact: A literal fact, e.g., (*Yao Ming*, *height*, *229 centimetres*), or a relational fact, e.g., (*Kobe Bryant*, *spouse*, *Vanessa Laine Bryant*).
QKey/QValue: The key and value of a qualifier.