☕ Teaching Broad Reasoning Skills via Decomposition-Guided Contexts

Harsh Trivedi Niranjan Balasubramanian Tushar Khot Ashish Sabharwal

Stony Brook University
Stony Brook, U.S.A.
{hjtrivedi,niranjan}@cs.stonybrook.edu

Allen Institute for AI
Seattle, U.S.A.
{tushark,ashishs}@allenai.org

Abstract

Question-answering datasets require a broad set of reasoning skills. We show how to use question decompositions to teach language models these broad reasoning skills in a robust fashion. Specifically, we use widely available QDMR representations to programmatically create synthetic contexts for real questions in six multihop reasoning datasets. These contexts are carefully designed to avoid common reasoning shortcuts prevalent in real contexts that prevent models from learning the right skills. This results in a pretraining dataset, named TeaBReaC, containing 525K multihop questions (with associated formal programs) covering about 900 reasoning patterns. We show that pretraining standard language models (LMs) on TeaBReaC before fine-tuning them on target datasets improves their performance by up to 13 EM points across 3 multihop QA datasets, with a 30 point gain on more complex questions. The resulting models also demonstrate higher robustness, with a 6-11 point improvement on two contrast sets. Furthermore, TeaBReaC pretraining substantially improves model performance and robustness even when starting with numeracy-aware LMs pretrained using recent methods (e.g., PReasM). Our work thus shows how one can effectively use decomposition-guided contexts to robustly teach multihop reasoning. [1]

1 Introduction

Multihop Question Answering (QA) is a complex problem that requires a wide variety of reasoning skills. In addition to basic reading comprehension (RC), models must connect multiple pieces of information, sometimes employ numerical and other forms of discrete reasoning, and compose these skills as required by the question. However, even though questions in multihop datasets often cover a broad range of interesting reasoning patterns, the datasets are dominated by only a few patterns, which is what trained models naturally focus on. Moreover, the contexts occurring in existing RC datasets often contain artifacts and reasoning shortcuts (Min et al., 2019a; Chen and Durrett, 2019; Trivedi et al., 2020). Such contexts allow models to find the answer while bypassing some reasoning steps, in turn preventing models from learning the intended reasoning skills.

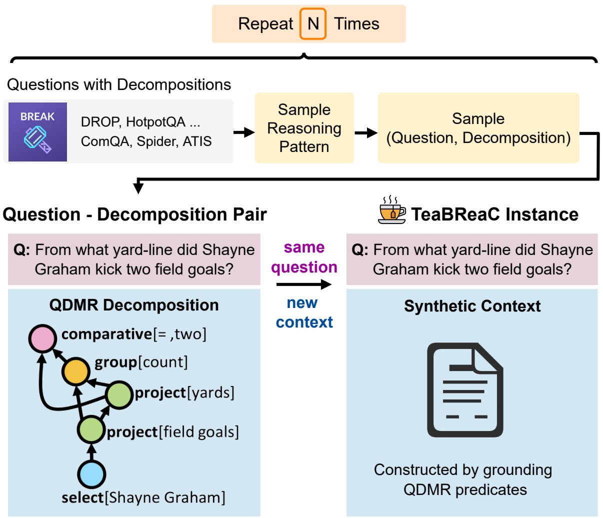

Repeat N Times
Questions with Decompositions
BREAK
DROP, HotpotQA ...
ComQA, Spider, ATIS
Sample Reasoning Pattern
Sample (Question, Decomposition)
Question - Decomposition Pair
TeaBReaC Instance
Q: From what yard-line did Shayne Graham kick two field goals?
same question
Q: From what yard-line did Shayne Graham kick two field goals?
QDMR Decomposition
comparative[= ,two]
group[count]
project[yards]
project[field goals]
select[Shayne Graham]
new context
Synthetic Context
Constructed by grounding QDMR predicates

Figure 1: TEABREAC ☕ Dataset Construction: We leverage widely available question decomposition annotations (QDMRs) for real questions from a broad range of datasets to carefully construct synthetic contexts such that answering the resulting ☕ question requires proper multihop reasoning. These questions are further rebalanced to help teach a broad set of reasoning skills.

How, then, can we teach models broad multihop reasoning skills? One way is to have greater control over the types of input contexts models see during training—contexts that cover a wide variety of reasoning patterns while not allowing models to easily succeed via shortcuts. We observe that questions in existing datasets (henceforth referred to as "real questions") already cover a wide variety of reasoning patterns. The challenge, then, is to teach these reasoning patterns *robustly*, even when

[1] Code and data available at https://github.com/stonybrooknlp/teabreac.

they are relatively rare in multihop datasets (e.g., 4-6 hops reasoning). As a means to this end, we turn to *synthetic context generation for real questions*. Specifically, we propose to construct contexts for real questions synthetically from scratch (instead of perturbing existing contexts), resulting in much greater control over reasoning shortcuts. Further, context generation also enables us to balance out the distribution of reasoning patterns in our dataset, e.g., by synthesizing additional contexts, and thereby examples, for questions from the long-tail of underrepresented reasoning patterns.

Our use of synthetic contexts to reliably teach broad skills is inspired by three strands of recent RC QA research. One strand has shown that skills learnt over synthetic data can indeed transfer to real datasets (Geva et al., 2020; Yang et al., 2021; Yoran et al., 2022; Pi et al., 2022). A second strand has shown that perturbing the existing (natural) contexts of RC instances in a targeted fashion can reduce artifact-based reasoning (Jia and Liang, 2017; Trivedi et al., 2020). A third strand has shown that carefully constructing contexts (for synthetic questions) to have sufficient distractors can reduce artifacts exploitable by current models (Trivedi et al., 2022; Khot et al., 2022).

Building upon these three strands, we introduce TEABREAC,[2] a teaching dataset that includes carefully constructed synthetic contexts for a broad set of real multihop questions sourced from six existing datasets. TEABREAC was designed with the goals of strong control over cheatability and balanced coverage of reasoning patterns. To identify the intended reasoning, we leverage question decomposition annotations, specifically Question Decomposition Meaning Representation or QDMR annotations which are widely available for a broad set of datasets (Wolfson et al., 2020).

Figure 1 shows the overview of our construction process for TEABREAC. We turn the basic structured representation of the reasoning steps in QDMRs into a precise *program* that can be executed against a synthetic context to arrive at an answer. We then construct a synthetic context by asserting a set facts that relate to the parts of the multihop question. We do this by grounding the predicates of QDMR (e.g., *field goals of Shayne Graham* in Fig. 1) with randomly generated entities. We also add distractor statements to the context to ensure that bypassing reasoning steps results in an incorrect answer. This forces models to learn the intended reasoning. We then add an outer loop around this process that ensures that the reasoning patterns—as measured by the program signatures of the questions—remain balanced in the final dataset. This forces models to learn a broad range of reasoning patterns instead of focusing on the few dominant ones. Finally, similar to prior work (Geva et al., 2020), we also add simpler single-step questions to teach individual primitive skills underlying our formal programs.

Our experiments demonstrate that pretraining large language models (LMs) on TEABREAC before fine-tuning on target multihop QA datasets results in significant improvements on multiple in-distribution evaluation sets (DROP (Dua et al., 2019), TAT-QA (Zhu et al., 2021), IIRC (Ferguson et al., 2020)) by up to 13 points (exact match), as well as on two contrastive evaluation sets of DROP by upto 11 points. Furthermore, even if we start with numeracy-aware LMs already pretrained on similar past work (Geva et al., 2020; Yang et al., 2021; Yoran et al., 2022), TEABREAC provides further improvement by upto 14 EM points. Interestingly, TEABREAC is substantially more beneficial for more complex questions (those with more reasoning steps), improving the T5 model by about 30 EM points on questions with 5 or more steps.

In summary, we make three contributions:

(1) A novel methodology to create a teaching dataset (a) with broad reasoning skills covering a wide range of multihop reasoning patterns and (b) leveraging existing QDMR annotations to carefully construct contexts that require true multi-hop reasoning. (2) The TEABREAC teaching dataset with over 525K questions covering about 900 reasoning patterns or program signatures. (3) An empirical demonstration that pretraining on TEABREAC before fine-tuning makes both regular and numeracy-aware LMs is much more effective and robust at multihop reasoning, especially for more complex questions.

2 Teaching Broad-Coverage Reasoning Skills in a Robust Fashion

Multihop questions come in a wide variety. Some involve numeric operations (Dua et al., 2019), some involve assessing whether complete information is present or not (Ferguson et al., 2020), some involve tables and text (Zhu et al., 2021), and so on. One

[2] TEABREAC stands for "**T**eaching **B**road **Rea**soning skills via decomposition-guided **C**ontexts", and is pronounced as "Tea Break".

way to surface the reasoning needed for answering these questions is to look at their *decomposition* into smaller reasoning steps that can be composed together in order to arrive at the correct answer. For example, consider the question in Fig. 1, *From what yard-line did Shayne kick two field goals?*. This can be decomposed as follows: list the field goals by Shayne Graham, identify the yard-lines corresponding to each of them, map each yard-line with the field goal and count them, and select the yard-line with two field goals.

While questions in multihop datasets are authored with the intent that such multi-step reasoning will be used to answer them, the context associated with the questions often allows models to cheat by taking shortcuts (Min et al., 2019a; Trivedi et al., 2020). E.g., if the context mentions field goals only by Shayne Graham and no one else, models can ignore the player name and still succeed.

Our key observation is that the decomposition of a question can be leveraged to carefully design a synthetic context for this question that avoids cheating, thereby allowing us to teach models a broad range of reasoning skills in a robust fashion. To achieve this, we procedurally create a large pretraining RC QA dataset, TeaBReaC, by using real multihop questions (from existing datasets) and their decomposition annotations (already available in the form of QDMRs), and carefully constructing synthetic contexts.

QDMR or Question Decomposition Meaning Representation (Wolfson et al., 2020) is a common way to represent the reasoning in many types of multihop questions as a structured decomposition graph. QDMR has standardized operators (represented as nodes) such as `select`, `project`, `group`, `comparative`, etc., that transform their input. These are connected together to a final node which produces the answer. Figure 1 shows the above example question paired with its QDMR graph. Importantly, QDMRs are already available for several multihop datasets. This allows us to take on our challenge of building contexts without shortcuts. It also provides evidence that QDMR is a general representation covering a broad spectrum of multihop reasoning phenomena.

Briefly, our method involves the following main steps; these are described in more detail in §3.

Making QDMRs more precise. To create QA instances that teach the precise reasoning captured in QDMR, we need a precise and formal representation of reasoning captured in QDMRs. QDMRs, although structured, don't quite do so, as they are written in natural language and don't specify the datatypes of their input and output. Since this will be crucial for our approach, we will convert QDMRs into formal programs with over 44 executable primitive operations along with their input/output types (§ 3.1).

Teaching robust compositional skills. Past work has shown that compositional questions don't necessitate multihop reasoning as datasets often have reasoning shortcuts (Min et al., 2019a; Chen and Durrett, 2019; Trivedi et al., 2020). To teach the reasoning reflected our formal programs robustly, our QA instances must be such that models cannot bypass the reasoning steps and still arrive at the correct answer. To achieve this goal, we create a synthetic QA instance from a question-program pair, where the question is the same as the original question, but the context is procedurally constructed by grounding the predicates in QDMR in a careful way such that models can't cheat their way to the correct answer.

Teaching a broad range of reasoning patterns Although QDMRs cover a broad range of reasoning patterns, we find that the natural distribution of reasoning patterns in QDMRs is extremely skewed towards popular reasoning patterns (§ 3.2). Training on QA instances generated from such a distribution leads models to overfit to only a few most representative reasoning patterns, and not learn broad-range reasoning skills. To ensure this doesn't happen, we make sure our synthetic dataset is more balanced in terms of reasoning patterns (§ 3.2).

Teaching a broad range of reasoning primitives. In addition to our process of constructing a pretraining dataset to teach compositional skills described thus far, we observe that it also helps if we teach models the constituent primitive reasoning skills. To achieve this, similar to prior work (Geva et al., 2020), we procedurally generate QA instances based on fixed templates for each of the 44 primitives present in our formal programs (§ 3.3).

3 TeaBReaC Dataset Construction

The overview of TeaBReaC construction pipeline is shown in Fig. 2. We discuss the QA instance generator in § 3.1, and discuss the dataset generator used to create QA dataset in § 3.2.

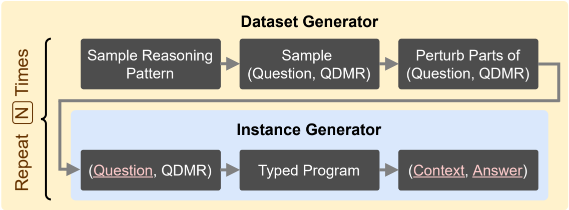


Figure 2: High-level schematic of TEABREAC construction process.

3.1 Instance Generator

The Instance Generator takes a question Q and its QDMR decomposition D as input, and generates a synthetic context C and the corresponding answer A as its output. The tuple (Q, C, A) is the generated reading comprehension QA instance. This conversion happens in two steps: (i) QDMR to Typed Program, (ii) Typed Program to Context and Answer.

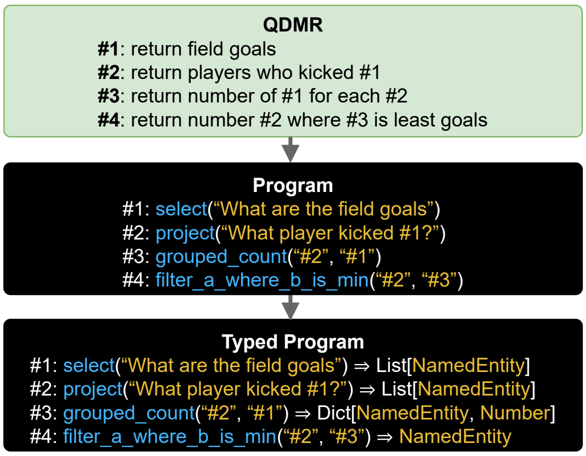
QDMR to Typed Program:

Our goal is to generate a synthetic context C that can be used to answer the question Q (based on the QDMR D), and to also provide the answer A. To be able to generate C and A, we must be able to create facts corresponding to each step in the QDMR reasoning graph (i.e., ground the QDMR predicates[3]) and compute the final answer by stepping through the QDMR program.

To achieve this, we need a formal representation (a **Program**) that captures the precise reasoning implied by the QDMR decomposition D, and that can be executed step-by-step (e.g., in a programming language such as Python). This isn't possible directly via QDMRs as (i) although structured, they are written in natural language and have variation inherent in natural language; (ii) they don't have input and output type information, e.g., it is unclear whether the `project` operator should generate a dictionary, a list, or a scalar, making it difficult to have the full program be executable.

To convert a QDMR D into a Program P, we define a set of functions (in Python) such as `select`, `filter`, `grouped_count`, etc, and then parse QDMRs into these functions using rules and heuristics. An example conversion is shown in Fig. 3.

In our representation, we have 44 Python functions (primitives) operating over various types (number, date, named entity) and structures (scalar, list, dictionary) of inputs and outputs. The full list is given in Appendix 6.



QDMR
#1: return field goals
#2: return players who kicked #1
#3: return number of #1 for each #2
#4: return number #2 where #3 is least goals

Program

```
#1: select("What are the field goals")
#2: project("What player kicked #1?")
#3: grouped_count("#2", "#1")
#4: filter_a_where_b_is_min("#2", "#3")
```

Typed Program

```
#1: select("What are the field goals") ⇒ List[NamedEntity]
#2: project("What player kicked #1?") ⇒ List[NamedEntity]
#3: grouped_count("#2", "#1") ⇒ Dict[NamedEntity, Number]
#4: filter_a_where_b_is_min("#2", "#3") ⇒ NamedEntity
```

Figure 3: Example conversion of a QDMR decomposition (top) to a Typed Program (bottom).

Note that these primitives don't always have a clearly defined output type. While in most cases the output type is obvious (e.g., `arithmetic_sum` returns a number), for some of these primitives (`select`, `project`, `filter`), it's under-defined. E.g., `select("number of soldiers in USA")` should output a single number, `select("when did India get independence")` should output a single date, and `select("countries surrounding India")` should output a list of named entities. For such operations, we again use heuristic rules and type propagation on the global structure of P to infer expected types and structures of output. We call the program with type information inferred for each step a **Typed Program** $\tilde{P}$, an example of which is shown in Fig. 3.

Synthetic Context + Answer:

Next, we generate C and A from the typed program $\tilde{P}$. We first describe the construction at a high level, then discuss an example generation in the context of the desirable properties we want our QA instances to have, and finally describe the general construction algorithm.

We generate C by grounding the **predicates** derived from the QDMR D with random **entities**. Fig. 4 shows an example of C for a simple program with three steps. **Predicates**: The predicates that need to be grounded can belong to 4 of the 44 operators (Table 6), which we refer to as grounding operators: `select`, `project`, `filter`, `boolean`. E.g., Fig. 4 uses `select` and `filter`. Examples

[3] E.g., the step "`return players who kicked #1`" has the predicate "`return players who kicked __`".

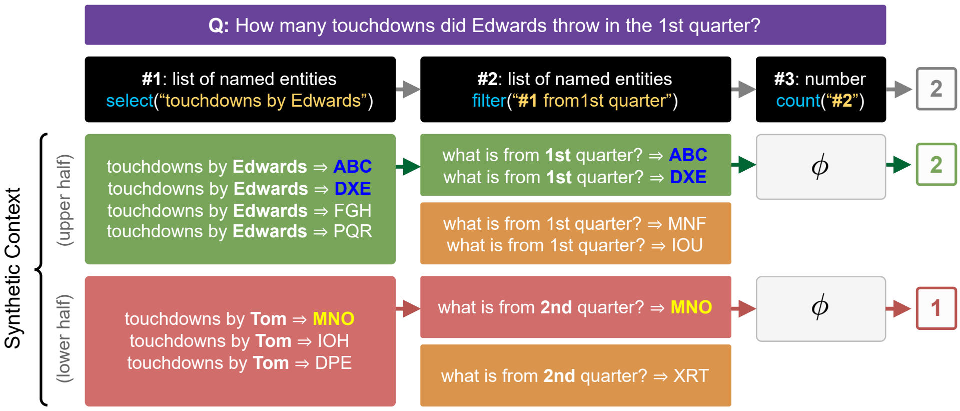

Q: How many touchdowns did Edwards throw in the 1st quarter?
#1: list of named entities
select("touchdowns by Edwards")
#2: list of named entities
filter("#1 from1st quarter")
#3: number
count("#2")
2
Synthetic Context
(upper half)
touchdowns by Edwards ⇒ ABC
touchdowns by Edwards ⇒ DXE
touchdowns by Edwards ⇒ FGH
touchdowns by Edwards ⇒ PQR
what is from 1st quarter? ⇒ ABC
what is from 1st quarter? ⇒ DXE
what is from 1st quarter? ⇒ MNF
what is from 1st quarter? ⇒ IOU
ϕ
2
(lower half)
touchdowns by Tom ⇒ MNO
touchdowns by Tom ⇒ IOH
touchdowns by Tom ⇒ DPE
what is from 2nd quarter? ⇒ MNO
what is from 2nd quarter? ⇒ XRT
ϕ
1

Figure 4: A simplified example of a QA instance in TeaBReaC, with a (simplified) real question from the DROP dataset and the synthetic context we construct for it using the question's 3-step decomposition. The instance satisfies desirable properties **P1**, **P2**, and **P3**, and thus helps robustly teach multihop reasoning skills.

involving `project` and `boolean` are shown in Appendix 6. **Entities:** The grounded entities can be of 3 types: number, date, or named entity. A random number can be anywhere from 0 to 1 million, a random date can be anywhere from year 1100 to 2022, and a random named entity can be any sequence of 3 letters (e.g., `ADC`). Since our programs are typed, we know which predicate should be grounded with which entity type. E.g., `select("number of soldiers in USA")` should be grounded using a number, whereas `select("countries surrounding India")` should be grounded with a list of named entities.

Minimizing reasoning shortcuts. Naively creating C using QDMR D can introduce simple shortcuts that models can exploit, thus circumventing the necessary reasoning. Note that QDMR is a sequence of steps where each step s_i can use answers from zero or more previous steps; e.g., "`return number #2 where #3 is least goals`" in Fig 3 (top) uses the answer from step #2 and #3. However, if there is only one player who scored field goals, all reasoning steps can be ignored. To ensure models do learn the intended reasoning, our goal is to create C such that one cannot bypass the intended reasoning (or program) steps and still arrive at the correct answer A. To this end, we ground the predicates with entities such that the following three properties hold:

P1: Answers to dependent steps can't be ignored. If step s_j is dependent on step s_i, then the answer to s_j can't be identified without knowing the answer to s_i. E.g., in Fig 4, step #2 is asking about *"1st quarter"*, in particular it's asking *"which of the touchdowns by Edward are from the first quarter"*. Since there are many touchdowns *"from the 1st quarter"*, and only some of them are *"touchdowns by Edward"* (indicated in blue), one cannot narrow down the answer to step #2 without knowing step #1's answer. We ensure this property holds for different operators in different ways. E.g., for `filter` operator, we ensure the answer to the step is always a *proper* subset of all the entities grounded with that predicate ({ABC, DXE} ⊂ {ABC, DXE, MNF, IOU} in Fig. 4).

P2: Steps can't be no-op. The input and output of any step cannot be the same, as otherwise the reasoning in that step can be bypassed. E.g., in Fig 4, step #2 is asking about *"1st quarter"*, in particular it's asking *"which of the touchdowns by Edwards are from the 1st quarter"*. Again, there are many *"touchdowns by Edwards"*, and only some of them are *"from the 1st quarter"* (indicated in blue). Therefore, ignoring step #2 (i.e., treating it as a no-op) would result in an incorrect answer being used for subsequent steps. Similar to the first property, we again ensure this property holds for different operators in different ways. E.g., for `filter` operator, we ensure the answer to the step is always a *proper* subset of the answer to the dependent step ({ABC, DXE} ⊂ {ABC, DXE, FGH, PQR} in Fig. 4).

Properties **P1** and **P2** describe the upper half of the synthetic context in Fig. 4, that is, facts pertaining to the gold reasoning chain. Although this ensures step-by-step execution will result in the gold answer, there is only one possible complete execution that leads to an answer. As a result, the question can be completely ignored. To fix this, we introduce a third property:

P3: Context also supports a different answer to a contrastive question. Just the way we generate facts corresponding to the gold chain of reasoning (upper half in Fig. 4), we also generate facts corresponding to a distractor chain (lower half of Fig. 4), potentially using perturbed predicates (e.g., Edwards ⇒ Tom, 1st ⇒ 2nd). This ensures there is always one minimally different (contrastive (Gardner et al., 2020)) question that results in a different answer in the same context. E.g., *"How many touchdowns did Tom throw in the 2nd quarter"* results in the answer 1, which is different from the gold answer 2 in Fig. 4). To generate predicate perturbations, we swap numbers, dates, and named entities (PERSON, ORG, etc) with a similar entity of the same type. The cases where predicate doesn't have any entity of these types, we use a similar but different and type-consistent predicate from a different question as a perturbed predicate. E.g., `yards of rushing touchdowns` could be perturbed to `yards of passing touchdowns`. To do this, we retrieve the top 30 type-consistent predicates with the highest word-overlap not exceeding 75%, and sample one at random.

General Algorithm. Algorithm 1 shows the pseudo-code for generating QA instances satisfying the properties mentioned above.

The `GenQAInstance` function takes question Q, QDMR D and expected answer cardinality `N` of the answer, and attempts to generate a QA instance with desirable properties for 200 maximum tries. For a given question, QDMR pair, we vary `N` $\in \{1, 2, 3, 4\}$.

The `facts` represent list of grounded predicates that form the context, `state.ans` represents step-wise answers for gold reasoning chain (e.g., green boxes in Fig. 4), and `state.dis` represents step-wise answers for distractor reasoning chain (e.g., red boxes in Fig. 4). These are initialized to ∅(`L3`) and updated during the instance generation.

To construct a QA instance, we iterate through the program (or QDMR) steps. For each step, we create facts for the gold reasoning chain by grounding the predicate in the QDMR and update the facts and answer state accordingly using the `execute` function. e.g., In step #2 in Fig. 4, the facts in the top-half are added and {ABC, DXE} is marked as the current answer state. The `execute` function will generate these facts and answers such that P1 and P2 are satisfied or return False if it can't. We similarly generate facts and update the state for the distractor reasoning chain (`L7`) by using a perturbed (`L6`) QDMR predicate (e.g., Edward ⇒ Tom, 1st ⇒ 2nd in Fig. 4). This generates the facts and reasoning chain shown in the lower half of Fig. 4 ensuring P3 is satisfied.

Algorithm 1 Pseudo-code for generating QA instances from question Q, QDMR D, and answer cardinality N

```
 1: function GenQAInstance( Q, D, N)
 2:   for $1 \leq i \leq 200$ do                              ▷ Max retries
 3:     state.ans, state.dis, facts ← ∅
 4:     for step ∈ qdmr.steps do
 5:       ans_succ ← execute(step, state.ans, facts)    ▷ Update for gold reasoning chain
 6:       maybe_perturb(step)            ▷ Perturb predicate for distractor chain
 7:       dis_succ ← execute(step, state.dis, facts)    ▷ Update for distractor reasoning chain
 8:       if not ans_succ or not dis_succ then
 9:         failed ← True
10:         break
11:       end if
12:     end for
13:     if (not failed and
14:        accept(state, facts, ans_num)) then
15:       return QA(Q,                                  ▷ question
16:                 facts,                              ▷ context
17:                 state.ans[-1])                      ▷ gold answer
18:     end if
19:   end for
20: end function
```

The implementation of `execute` function is dependent on the program primitives (Table 6) and will be provided in the released code `https://github.com/stonybrooknlp/teabreac`. But broadly speaking there are two classes of primitives: (1) primitives like `select` and `filter` that need to first add facts by grounding the predicate, and then update the answer state for that step (e.g., step #1 and #2 in Fig. 4) (2) primitives like count with no additional grounding of facts and only need to update the state based on the underlying computation (e.g., step #3 in Fig. 4).

If all the steps finish with success, we check if the generation is `acceptable` (`L14`) before creating a QA instance. For it to be acceptable, the generated answer cardinality must match the expected value, the number of facts must be within 25, and the final answer for gold and distractor reasoning chains must be different. We create a reading comprehension QA instance with the input question Q as question, facts as the context (concatenated after shuffling), and the answer at the final step as the gold answer.

3.2 Dataset Generator

Now that we have a way to generate QA instance from a (question, QDMR) pair, we can generate a dataset by just using questions from datasets with annotated QDMRs. However, we find that the natural distribution of the *reasoning patterns* in these datasets is extremely long-tailed. We define **reasoning pattern** as a unique sequence of primitives present in the program. e.g., program in Fig. 4 has 3 steps having `select`, `filter` and `count` primitives, in that order, so the reasoning pattern is "`select filter count`"

If we invoke instance generator uniformly over the available QDMRs, we get a QA dataset that is extremely skewed towards popular patterns. Pretraining models on such a dataset overfits it only on few reasoning patterns and prevents it from learning a broad range of reasoning skills. To fix this, we employ the following strategy in the dataset generator: (i) sample a reasoning pattern (ii) sample a question-QDMR pair from that reasoning pattern (iii) possibly perturb the entities (named entities, dates, numbers, ordinals) in the question (and accordingly in the QDMR) with a closely similar entity of the same type[4] (iv) invoke the instance generator for $N \in \{1, 2, 3, 4\}$. The resulting training dataset has about 900 reasoning patterns with the top 10 common patterns having only 4% of examples (compared to 50% in the source typed programs).

3.3 Additional QA Instances for Primitives

Lastly, in addition to these synthetic multi-hop instances, we also have instances to teach 44 individual primitives, similar to Geva et al. (2020). These instances are created based on simple templates. E.g., for primitive `filter_a_where_b_is_compared_to`, a question could be "Entities that have value larger than 948768.92?" and context could be "Entity AFE has value 871781. Entity RQX has value 989,517.24." resulting in answer ['RQX']. Example QA instances for various other primitives are given in the appendix (Table 7). In all, we've 30K training and 1K development instances for each primitive.

3.4 Final Dataset

Final TeaBReaC dataset has 525K and 15K train and dev multihop QA instances respectively, and has about 900 reasoning patterns.

To create it we used QDMRs from QA and semantic parsing datasets, DROP (Dua et al., 2019), ComplexWebQuestions (Talmor and Berant, 2018), HotpotQA (Yang et al., 2018), SPIDER (Yu et al., 2018), ComQA (Abujabal et al., 2019), ATIS (Price, 1990).[5] We use both `low` and `high` level decompositions from QDMR limited to two to six reasoning steps.

4 Experiments

4.1 Experimental Setup

To test the effectiveness of TeaBReaC pretraining, we compare models directly fine-tuned on target datasets with models first pretrained on TeaBReaC and then fine-tuned on target datasets. We report the exact match metric (EM) for all evaluations.

4.1.1 Datasets

We evaluate **in-domain performance** using DROP, TAT-QA and IIRC. The reported numbers are on their dev sets.[6] For IIRC, we consider two settings: gold-setting (IIRC-G) which uses only gold supporting sentences as reading comprehension context, and retrieved-setting (IIRC-R) which retrieves paragraphs using a retrieval marginalization method (Ni et al., 2021). We evaluate **robustness** using DROP contrast set (Gardner et al., 2020) and DROP BPB contrast set (Geva et al., 2022)[7]. For robustness evaluation, we only fine-tune on DROP dataset and evaluate on the contrast sets directly.

4.1.2 Models

We evaluate TeaBReaC pretraining on two kinds of (language) models. **Plain language models**: T5-Large (Raffel et al., 2020) and Bart-Large (Lewis et al., 2020). **Numeracy-aware language models**: These are language models pretrained

[4] e.g., Edwards ⇒ Tom to create a new question: "How many touchdowns did Tom throw in the 1st quarter?". Since this perturbation is similar to the one used to create distractor chains, it makes distinguishing these distractor chains in the unperturbed questions from the gold chains in the perturbed questions much harder and better enforces property P3 in §3.1.

[5] We did not use visual QA datasets CLEVR-humans (Johnson et al., 2017) and NLVR2 (Suhr et al., 2019) as the questions in it are vastly different from ones we see in reading comprehension-based datasets.

[6] We selected training hyper-parameter (learning rate) for each baseline model and dataset, based on the dev set performance. Our experiments with TeaBReaC use this identical learning rate. Thus, the reported results slightly favor the baseline models that *don't* use TeaBReaC. A proper evaluation on unseen data will be included in the next revision of this paper.

[7] We use the human validated set.

on synthetic dataset from two past works. NT5 (Yang et al., 2021), which is a T5-small[8] model pretrained on a dataset released by Geva et al. (2020), and PReasM-Large, which is a T5-Large model pretrained on dataset released by Yoran et al. (2022). Our models are implemented using PyTorch (Paszke et al., 2019), Huggingface Transformers (Wolf et al., 2019) and AllenNLP (Gardner et al., 2017). The implementation details and training hyperparameters are given in the appendix, § A.

Tokenization. We find that character tokenization for numbers (a trick adopted from NT5 (Yang et al., 2021)) significantly improves model performance. For instance, as shown in Table 1, PReasM-large fine-tuned on DROP without any such tokenization gets 69.4 (as reported in (Yoran et al., 2022)), but with such tokenization (our implementation) gets 76.6. We see similar trends on T5 and Bart as well. So we use this tokenization as a default for all models across all our experiments.

Method	Dig. Tok.	DROP
T5 (reported in (Yoran et al., 2022))	✖	61.8
T5 (our implementation)	✔	**70.3**
PReasM (reported in (Yoran et al., 2022))	✖	69.4
PReasM (our implementation)	✔	**76.6**

Table 1: Digit tokenization (character tokenization for numbers) siginifantly helps downstream perforamnce, so we use it everywhere.

4.2 Experimental Results

Learnability of TeaBReaC

Since our goal is to teach models the reasoning skills in TeaBReaC, we first assess how well models do on the TeaBReaC dataset. As shown in Table 2, models are able to learn both primitive and multihop QA skills required in TeaBReaC. On primitives instances models get 92-99 accuracy, and on multihop instances, models get 82-86 accuracy. We'll later show that these scores are good enough to make progress on real datasets. At the same time, these aren't perfect scores, demonstrating limitations of vanilla LM-based neural models. Thus, TeaBReaC can also serve as a benchmark to help design better multihop models, especially for questions requiring 4 or more steps of reasoning.

Model	☕ Primitive	☕ Multihop
T5 ☕	94.6	83.7
Bart ☕	92.1	83.0
PReasM ☕	99.2	81.6
NT5 ☕	95.0	86.2

Table 2: Models learn the skills required in TeaBReaC during pretraining well, but achieving perfect score is challenging for vanilla LM-based neural models.

	Model	DROP	TAT-QA	IIRC-G	IIRC-R
Plain LMs	T5	70.3	38.6	64.2	42.3
	T5 ☕	**78.3**	**50.9**	**69.2**	**43.1**
	Bart	66.5	35.5	62.1	42.0
	Bart ☕	**78.0**	**44.4**	**72.0**	**45.4**
Num. LMs	NT5	69.2	44.3	**66.6**	**41.8**
	NT5 ☕	**71.6**	**47.0**	66.0	40.7
	PReasM	76.9	40.0	70.0	42.0
	PReasM ☕	**80.1**	**54.2**	**73.3**	**47.2**
Others	GenBERT	68.8	–	–	–
	POET$_{\text{BART}}$	77.7	41.5	–	–

Table 3: Model Performance: EM scores with and without ☕ TeaBReaC pretraining (on dev sets). Pretraining language models (LMs) on TeaBReaC improves their performance across multiple QA datasets, for both plain and numeracy-aware (Num.) LMs. NT5 is the only small-sized LM considered and exhibits a somewhat different trend.

TeaBReaC improves model performance

Table 3 compares performance on DROP, TAT-QA, IIRC-G and IIRC-R. For both plain language models, T5 and Bart, TeaBReaC pretraining results in substantial improvements across all datasets — 8-11 points gain on DROP, 9-13 points on TAT-QA, 6-10 points on IIRC-G and 1-3 points on IIRC-R. For numeracy-aware language models, NT5 and PReasM, TeaBReaC pretraining results in 2-3 points of improvement on DROP and 3-14 points of improvement on TAT-QA. TeaBReaC pretraining doesn't improve NT5 performance on IIRC-G and IIRC-R, but it improves PReasM performance on both datasets by 3-5 points.

We also compare with GenBERT (Geva et al., 2020) and POET$_{\text{BART}}$ (Pi et al., 2022)[9] numbers as reported in the respective papers. GenBERT is a

[8] NT5 has only been released in small size.

[9] Training data and model checkpoints of POET aren't publicly available at the time of the submission.

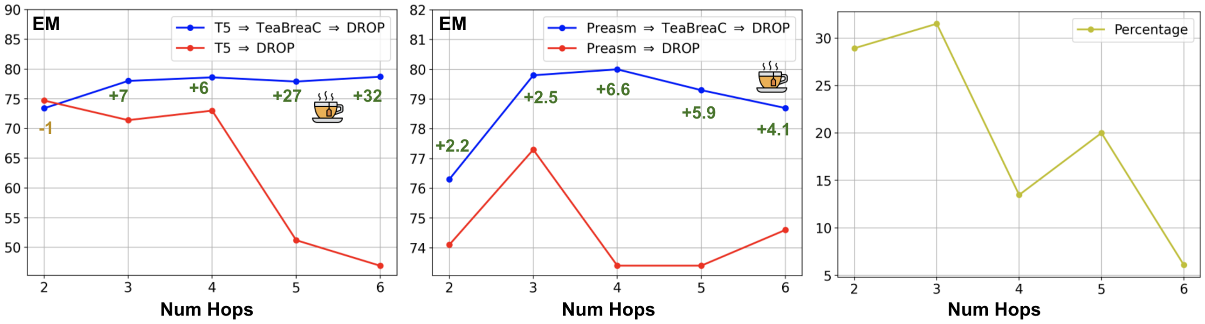


Figure 5: (Left) EM scores with and without 🍵 TEABREAC pretraining on DROP across varying numbers of hops, as determined by QDMR decompositions. (Middle) Same as left, but with PReasM model as the starting point. (Right) Number of questions for each number of hops in DROP dev set. TEABREAC pretraining helps more on more complex questions (Left, Middle). The average performance metric doesn't show such large improvements because more complex questions are less frequent (Right).

BERT-large model specialized for DROP dataset trained with synthetic data proposed by Geva et al. (2020). Our models of comparable size (T5 and Bart) when pretrained with TEABREAC perform substantial better than GenBERT (9-10 points). POET is a pretraining method on arithmetic, logic-based, and SQL-based synthetic data. Our Bart-large model of comparable size, performs 0.3 and 3 points better than POET$_{\text{BART}}$.

TEABREAC improves model robustness

Table 4 compares robustness via performance on the DROP contrast set and the DROP BPB set.

	Model	DROP-CS	DROP-BPB
Plain LMs	T5	44.6	51.1
	T5 🍵	**52.6**	**57.0**
	Bart	43.2	45.3
	Bart 🍵	**53.6**	**56.3**
Num. LMs	NT5	38.6	46.2
	NT5 🍵	**43.4**	**47.6**
	PReasM	49.8	50.3
	PReasM 🍵	**53.5**	**58.8**

Table 4: Robustness Evaluation: EM scores with and without 🍵 TEABREAC pretraining on two DROP contrast sets. Pretraining LMs on TEABREAC improves models robustness, for both plain and numeracy aware (num.) LMs. NT5 is the only small-sized LM considered.

For both plain language models, T5 and Bart, TEABREAC pretraining shows substantial improvements in robustness — 8-10 points improvements on DROP contrast set, and 6-11 points on DROP BPB Set. For numeracy-aware language models, NT5 and PReasM, TEABREAC pretraining results in 4-5 points of improvement on DROP contrast set and 1-8 points of improvement on DROP BPB set. The fact that TEABREAC improves models pretrained on previous synthetic datasets demonstrates its complementary nature.

🍵 improves more on more complex questions

We further investigate how the improvements provided by TEABREAC vary based on the complexity or number of reasoning steps of the question. We can obtain the number of reasoning steps from QDMRs, but the QDMRs are not available for all questions in DROP dev set. Therefore, we train a T5-Large based QDMR parser, run it on the entire DROP dev set, and use the number of reasoning steps of the *predicted* QDMR as a proxy.

Figure 5 compares the performance of TEABREAC pretraining on questions with increasing (estimated) hop lengths. For plain language model, T5, the baseline model significantly drops from 75 to 45 as the number of hops or complexity of the question increases. In contrast, model with TEABREAC pretraining stays the same or improves with increasing number of hops. In other words, there is a significantly larger improvement for more complex questions, where the original T5 model struggles (e.g., 30 points gain on 4+ hops vs 8 points gain on average). Similarly, for numeracy-aware language model, PReasM-Large, we see more improvement on more complex questions (e.g., 4.1-6.5 points on 3+ hops, 3.5 points on average).

We also observe that more complex questions are significantly less frequent in the DROP dev set as shown in the rightmost plot of Fig. 5. This makes our large gains on more complex questions (achieved in part via the balancing of reasoning patterns in TEABREAC) not quite visible in the aggregate dataset metric reported earlier in Table 3.

5 Related Work

Many strands of research have pursued the goals of building robust models with broad reasoning skills. These include teaching these skills to models using data augmentation (natural or synthetic), ensuring robust reasoning using datasets with minimal reasoning shortcuts and adversarial perturbation, or directly building multi-step reasoning models using question decompositions. Our work builds on these ideas by creating a synthetic dataset with minimal reasoning shortcuts to teach robust reasoning skills to existing models. Additionally, we develop a novel method to leverage existing decompositions to capture broad reasoning skills in our dataset.

Question Decomposition. This aims to represent complex questions in terms of their component reasoning steps. These decompositions can be useful for building more interpretable and modular systems or just provide a shared question meaning representation across multiple datasets.

To enable development of better systems, several recent multihop QA datasets come with question decompostion annotations (Khot et al., 2020; Talmor and Berant, 2018; Geva et al., 2021; Trivedi et al., 2022; Khot et al., 2022). These works have enabled the development of *explicit* multistep reasoning systems that first decomposes a multihop question into sub-questions, and answers the subquestions step-by-step to arrive at the answer (Min et al., 2019b; Khot et al., 2021; Trivedi et al., 2022). Our goal in this work is to use decompositions to instead teach black-box language to perform multistep reasoning implicitly (within the model).

Since each dataset uses its own decomposition format, they have also led to narrow dataset-specific solutions. The BREAK dataset (Wolfson et al., 2020) on the other hand, defined a standardized meaning representation format (inspired by semantic parsing) for several QA datasets. This shared representation has allowed the development of contrastive datasets (Geva et al., 2022). In this work, we leverage these annotations to build a dataset to teach broad reasoning skills to models.

Robust Multihop Reasoning. Past work has shown how to perturb existing multihop QA instances to prevent shortcuts and incentivize robust reasoning. Jiang and Bansal (2019); Ding et al. (2021) created adversarial multihop question by perturbing the reasoning chains in HotpotQA (Yang et al., 2018). Other datasets (Trivedi et al., 2020, 2022; Lee et al., 2021) ensure robust reasoning via minimally perturbed unanswerable questions. Our approach targets a broader set of questions and eliminates multiple reasoning shortcuts.

The closest work in this line is the Break-Perturb-Build dataset (Geva et al., 2022). BPB dataset also used QDMR but with the goal of creating contrastive questions via minor question perturbation (Kaushik et al., 2019; Gardner et al., 2020). Importantly, they use the existing context with reasoning shortcuts that can be hard to eliminate with only question perturbation (e.g., no distractors). Additionally this dataset is mainly used as an evaluation set (as we also do) and has not been shown to result in better models by training on it.

Data Augmentation for QA. Several past works have used data augmentation via synthetic datasets to improve QA performance. Following recent works are most relevant to our approach. Geva et al. (2020) created a synthetic dataset using a few hand-crafted templates for injecting numerical reasoning skills (along with a specialized architecture). This dataset was also later used to build a numeracy-aware T5 (Raffel et al., 2020) model: NT5 Yang et al. (2021). Yoran et al. (2022) created a synthetic dataset using 13 handcrafted multihop QA reasoning patterns applied on wikipedia tables. Lastly, Pi et al. (2022) showed that pretraining language models on synthetic dataset derived from input and output of program executors (arithmetic, logic-based and SQL-based) can also improve downstream QA performance. In contrast to these works, we use actual questions from a wide range of real datasets to teach a broad range of multi-hop reasoning skills.

Past work has also explored synthetic QA dataset generation leveraging generation capabilities of large language models(LM). In addition to many works on generating single-hop QA datasets (Bartolo et al., 2021; Alberti et al., 2019; Puri et al., 2020), Pan et al. (2021) have used LMs to create a multihop QA dataset. However, they focus on generating questions for only two types of reasoning (composition and comparison) given real paragraphs. We leave the prospect of combining our synthetic context generation method with the generation capabilities of LMs as a future work.

Multi-task learning over several QA datasets (Khashabi et al., 2020; Talmor and Berant, 2019) can also help models learn a broad range of QA skills. However, these works have only targeted simple QA datasets. We view such multi-task learning as orthogonal to our method; it can be used to train models on TEABREAC along with other QA datasets.

6 Conclusions

Large language models demonstrate impressive reading comprehension abilities and a wide variety of reasoning skills. Despite these abilities and the availability of large scale multihop QA datasets, large LM-based QA models do not reliably learn to use such reasoning skills for answering complex questions. In this work, we show that the greater control that synthetic contexts offer can be leveraged to create a teaching dataset where models can learn a broad range of reasoning skills in a reliable manner, especially for more complex questions. Our transfer results on actual QA datasets also add to the line of work that shows synthetic datasets can be used to inject useful skills that transfer over to real natural language tasks. Given the artifact issues in real datasets (specifically, in their contexts) and the difficulty in controlling for them via perturbations, leveraging existing multihop questions for their broad reasoning patterns but using synthetic contexts appears to be a viable alternative for carefully constructing teaching datasets, where models can learn the *right way* to reason.

References

Abdalghani Abujabal, Rishiraj Saha Roy, Mohamed Yahya, and Gerhard Weikum. 2019. ComQA: A community-sourced dataset for complex factoid question answering with paraphrase clusters. In *NAACL*, pages 307–317.

Chris Alberti, Daniel Andor, Emily Pitler, Jacob Devlin, and Michael Collins. 2019. Synthetic QA corpora generation with roundtrip consistency. In *ACL*, pages 6168–6173.

Max Bartolo, Tristan Thrush, Robin Jia, Sebastian Riedel, Pontus Stenetorp, and Douwe Kiela. 2021. Improving question answering model robustness with synthetic adversarial data generation. In *EMNLP*, pages 8830–8848.

Jifan Chen and Greg Durrett. 2019. Understanding dataset design choices for multi-hop reasoning. In *NAACL-HLT*.

Jiayu Ding, Siyuan Wang, Qin Chen, and Zhongyu Wei. 2021. Reasoning chain based adversarial attack for multi-hop question answering. *arXiv preprint arXiv:2112.09658*.

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*.

James Ferguson, Matt Gardner, Hannaneh Hajishirzi, Tushar Khot, and Pradeep Dasigi. 2020. IIRC: A dataset of incomplete information reading comprehension questions. In *EMNLP*.

Matt Gardner, Yoav Artzi, Victoria Basmova, Jonathan Berant, Ben Bogin, Sihao Chen, Pradeep Dasigi, Dheeru Dua, Yanai Elazar, Ananth Gottumukkala, et al. 2020. Evaluating models' local decision boundaries via contrast sets. In *Findings of EMNLP*.

Matt Gardner, Joel Grus, Mark Neumann, Oyvind Tafjord, Pradeep Dasigi, Nelson F. Liu, Matthew Peters, Michael Schmitz, and Luke S. Zettlemoyer. 2017. AllenNLP: A deep semantic natural language processing platform. *arXiv preprint arXiv:1803.07640*.

Mor Geva, Ankit Gupta, and Jonathan Berant. 2020. Injecting numerical reasoning skills into language models. In *ACL*, pages 946–958.

Mor Geva, Daniel Khashabi, Elad Segal, Tushar Khot, Dan Roth, and Jonathan Berant. 2021. Did aristotle use a laptop? a question answering benchmark with implicit reasoning strategies. *TACL*, 9:346–361.

Mor Geva, Tomer Wolfson, and Jonathan Berant. 2022. Break, perturb, build: Automatic perturbation of reasoning paths through question decomposition. *TACL*, 10:111–126.

Robin Jia and Percy Liang. 2017. Adversarial examples for evaluating reading comprehension systems. In *EMNLP*.

Yichen Jiang and Mohit Bansal. 2019. Avoiding reasoning shortcuts: Adversarial evaluation, training, and model development for multi-hop QA. In *ACL*.

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*, pages 2901–2910.

Divyansh Kaushik, Eduard Hovy, and Zachary Lipton. 2019. Learning the difference that makes a difference with counterfactually-augmented data. In *ICLR*.

Daniel Khashabi, Sewon Min, Tushar Khot, Ashish Sabhwaral, Oyvind Tafjord, Peter Clark, and Hannaneh Hajishirzi. 2020. UnifiedQA: Crossing format boundaries with a single qa system. *Findings of EMNLP*.

Tushar Khot, Peter Clark, Michal Guerquin, Paul Edward Jansen, and Ashish Sabharwal. 2020. QASC: A dataset for question answering via sentence composition. In *AAAI*.

Tushar Khot, Daniel Khashabi, Kyle Richardson, Peter Clark, and Ashish Sabharwal. 2021. Text modular networks: Learning to decompose tasks in the language of existing models. In *NAACL*.

Tushar Khot, Kyle Richardson, Daniel Khashabi, and Ashish Sabharwal. 2022. Hey AI, can you solve complex tasks by talking to agents? In *Findings of ACL*, pages 1808–1823.

Kyungjae Lee, Seung-won Hwang, Sang-eun Han, and Dohyeon Lee. 2021. Robustifying multi-hop QA through pseudo-evidentiality training. In *ACL-IJCNLP*, pages 6110–6119.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *ACL*.

Sewon Min, Eric Wallace, Sameer Singh, Matt Gardner, Hannaneh Hajishirzi, and Luke Zettlemoyer. 2019a. Compositional questions do not necessitate multi-hop reasoning. In *ACL*.

Sewon Min, Victor Zhong, Luke Zettlemoyer, and Hannaneh Hajishirzi. 2019b. Multi-hop reading comprehension through question decomposition and rescoring. In *ACL*.

Ansong Ni, Matt Gardner, and Pradeep Dasigi. 2021. Mitigating false-negative contexts in multi-document question answering with retrieval marginalization. In *EMNLP*, pages 6149–6161.

Liangming Pan, Wenhu Chen, Wenhan Xiong, Min-Yen Kan, and William Yang Wang. 2021. Unsupervised multi-hop question answering by question generation. In *NAACL*.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An imperative style, high-performance deep learning library. In *NeurIPS*, pages 8024–8035.

Xinyu Pi, Qian Liu, Bei Chen, Morteza Ziyadi, Zeqi Lin, Yan Gao, Qiang Fu, Jian-Guang Lou, and Weizhu Chen. 2022. Reasoning like program executors. *arXiv preprint arXiv:2201.11473*.

Patti Price. 1990. Evaluation of spoken language systems: The atis domain. In *Speech and Natural Language: Proceedings of a Workshop Held at Hidden Valley, Pennsylvania, June 24-27, 1990*.

Raul Puri, Ryan Spring, Mohammad Shoeybi, Mostofa Patwary, and Bryan Catanzaro. 2020. Training question answering models from synthetic data. In *EMNLP*, pages 5811–5826.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *JMLR*.

Noam Shazeer and Mitchell Stern. 2018. Adafactor: Adaptive learning rates with sublinear memory cost. In *International Conference on Machine Learning*, pages 4596–4604. PMLR.

Alane Suhr, Stephanie Zhou, Ally Zhang, Iris Zhang, Huajun Bai, and Yoav Artzi. 2019. A corpus for reasoning about natural language grounded in photographs. In *ACL*.

Alon Talmor and Jonathan Berant. 2018. The web as a knowledge-base for answering complex questions. In *NAACL-HLT*, pages 641–651, New Orleans, Louisiana. Association for Computational Linguistics.

Alon Talmor and Jonathan Berant. 2019. MultiQA: An empirical investigation of generalization and transfer in reading comprehension. In *ACL*, pages 4911–4921.

Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. 2020. Is multihop QA in DiRe condition? Measuring and reducing disconnected reasoning. In *EMNLP*.

Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. 2022. MuSiQue: Multihop questions via single-hop question composition. *TACL*, 10:539–554.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, R'emi Louf, Morgan Funtowicz, and Jamie Brew. 2019. Huggingface's transformers: State-of-the-art natural language processing. *ArXiv*, abs/1910.03771.

Tomer Wolfson, Mor Geva, Ankit Gupta, Matt Gardner, Yoav Goldberg, Daniel Deutch, and Jonathan Berant. 2020. Break it down: A question understanding benchmark. *TACL*.

Peng-Jian Yang, Ying Ting Chen, Yuechan Chen, and Daniel Cer. 2021. Nt5?! training t5 to perform numerical reasoning. *arXiv preprint arXiv:2104.07307*.

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. In *EMNLP*.

Ori Yoran, Alon Talmor, and Jonathan Berant. 2022. Turning tables: Generating examples from semi-structured tables for endowing language models with reasoning skills. In *ACL*.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *EMNLP*, pages 3911–3921.

Fengbin Zhu, Wenqiang Lei, Youcheng Huang, Chao Wang, Shuo Zhang, Jiancheng Lv, Fuli Feng, and Tat-Seng Chua. 2021. TAT-QA: A question answering benchmark on a hybrid of tabular and textual content in finance. In *ACL-IJCNLP*, pages 3277–3287.

A Implementation Details

We train all models on a RTX8000 (48GB) GPU. The hyperparameters for pretraining and fine-tuning are given in Table 5. The only hyperparameter we sweeped over is learning rate (1e-5, 5e-5, 1e-4, 5e-4, 1e-3). The number of epochs were set to a large number with early stopping based on validation score. We've used Adafactor optimizer for all our experiments (Shazeer and Stern, 2018).

Model	Dataset	LR	Epochs	BS
T5	TEABREAC	10^{-4}	20	32
Bart	TEABREAC	10^{-5}	20	32
NT5	TEABREAC	10^{-3}	20	32
Preasm	TEABREAC	5×10^{-5}	20	32
T5	DROP	10^{-4}	20	32
Bart	DROP	10^{-5}	20	32
NT5	DROP	10^{-3}	40	32
Preasm	DROP	5×10^{-5}	20	32
T5	TAT-QA	10^{-4}	20	32
Bart	TAT-QA	10^{-5}	20	32
NT5	TAT-QA	10^{-3}	40	32
Preasm	TAT-QA	5×10^{-5}	20	32
T5	IIRC	10^{-4}	20	32
Bart	IIRC	10^{-5}	20	32
NT5	IIRC	10^{-3}	40	32
Preasm	IIRC	5×10^{-5}	20	32

Table 5: (Top) Hyperparameters for pretraining language models on TEABREAC. For large sized models (T5, Bart, Preasm), each epoch constitutes 100000/32=3125 steps. For small sized model (NT5), each epoch constitutes 1000000/32=31250 steps. For each step, we uniformly randomly sample a batch of TEABREAC compositional (multihop) instances or primitive instance. (Bottom) Hyperparameters for training language models on target datasets from scratch or fine-tuning language models pretrained on TEABREAC on target datasets. The hyperparameters for IIRC-gold and IIRC-retrieved experiments are the same. NT5 is a small-sized model, all others are large-sized. LR refers to learning rate and BS refers to batch size.

B Examples of Multihop QA Instances

Example multihop QA instances involving `project` and `boolean` primitives are given in Fig. 6.

C List of Primitives (Python Functions)

List of primitives (python functions) and a corresponding example is given in Table 6.

D Examples of Instances for Individual Primitives

Examples of template based QA instances for teaching individual primitives are given in Table 7.

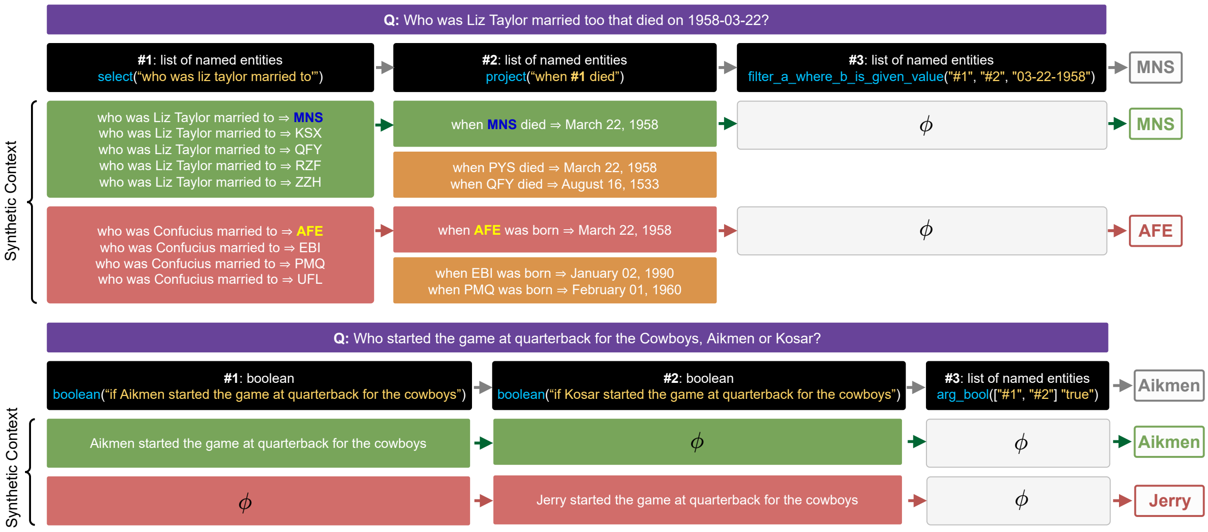

Q: Who was Liz Taylor married too that died on 1958-03-22?
#1: list of named entities
select("who was liz taylor married to"")
#2: list of named entities
project("when #1 died")
#3: list of named entities
filter_a_where_b_is_given_value("#1", "#2", "03-22-1958")
MNS
Synthetic Context
who was Liz Taylor married to ⇒ MNS
who was Liz Taylor married to ⇒ KSX
who was Liz Taylor married to ⇒ QFY
who was Liz Taylor married to ⇒ RZF
who was Liz Taylor married to ⇒ ZZH
when MNS died ⇒ March 22, 1958
when PYS died ⇒ March 22, 1958
when QFY died ⇒ August 16, 1533
ϕ
MNS
who was Confucius married to ⇒ AFE
who was Confucius married to ⇒ EBI
who was Confucius married to ⇒ PMQ
who was Confucius married to ⇒ UFL
when AFE was born ⇒ March 22, 1958
when EBI was born ⇒ January 02, 1990
when PMQ was born ⇒ February 01, 1960
ϕ
AFE
Q: Who started the game at quarterback for the Cowboys, Aikmen or Kosar?
#1: boolean
boolean("if Aikmen started the game at quarterback for the cowboys")
#2: boolean
boolean("if Kosar started the game at quarterback for the cowboys")
#3: list of named entities
arg_bool(["#1", "#2"] "true")
Aikmen
Synthetic Context
Aikmen started the game at quarterback for the cowboys
ϕ
ϕ
Aikmen
ϕ
Jerry started the game at quarterback for the cowboys
ϕ
Jerry

Figure 6: Synthetic reading comprehension QA instances involving `project` (top) and `boolean` (bottom) primitives.

Table 6: List of primitives (python functions) and a corresponding example.

Primitive	Example
compare_numbers	compare_numbers(#1, #2, “>") ⇒ False **State**: #1: 25 #2: 28
compare_dates	compare_dates(#1, #2, “>") ⇒ False **State**: #1: 25 Jan 2012 #2: 28 Jan 2012
maximum_date	maximum_date([#1, #2]) ⇒ 28 Jan 2012 **State**: #1: 25 Jan 2012 #2: 28 Jan 2012
minimum_date	minimum_date([#1, #2]) ⇒ 25 Jan 2012 **State**: #1: 25 Jan 2012 #2: 28 Jan 2012
date_subtraction	date_subtraction(#1, #2, “days") ⇒ 3 **State**: #1: 25 Jan 2012 #2: 28 Jan 2012
arg_maximum_date	arg_maximum_date([#1, #2]) ⇒ #2 **State**: #1: 25 Jan 2012 #2: 28 Jan 2012
arg_minimum_date	arg_minimum_date([#1, #2]) ⇒ #1 **State**: #1: 25 Jan 2012 #2: 28 Jan 2012
arg_bool	arg_bool([#1, #2], “true") ⇒ #1 **State**: #1: True #2: False
count	count(#1) ⇒ 3 **State**: #1: [ABC, XZE, PQR]
addition	addition(#1) ⇒ 2657.3 **State**: #1: [3, 2564.2, 90.1]
subtraction	subtraction(100, #1): 75 **State**: #1: 25
multiplication	multiplication(#1, 5): 125 **State**: #1: 25
division	division(#1, 100): 254.2 **State**: #1: 25420

Continued on next page

Table 6 – *Continued from previous page*

Primitive	**Example**
mean	mean(#1) ⇒ 885.8 **State**: #1: [3, 2564.2, 90.1]
maximum_number	maximum_number(#1) ⇒ 2564.2 **State**: #1: [3, 2564.2, 90.1]
minimum_number	minimum_number(#1) ⇒ 3 **State**: #1: [3, 2564.2, 90.1]
arg_maximum_number	arg_maximum_number([#1, #2, #3]) ⇒ #2 **State**: #1: 3 #2: 2564.2 #3: 90.1
arg_minimum_number	arg_minimum_number([#1, #2, #3]) ⇒ #1 **State**: #1: 3 #2: 2564.2 #3: 90.1
kth_highest	kth_highest(#1, 2) ⇒ 90.1 **State**: #1: [3, 2564.2, 90.1]
kth_lowest	kth_lowest(#1, 2) ⇒ 90.1 **State**: #1: [3, 2564.2, 90.1]
are_items_same	are_items_same(#1, #2) ⇒ False **State**: #1: ABC #2: EDX
are_items_different	are_items_different(#1, #2) ⇒ True **State**: #1: ABC #2: EDX
filter_a_where_b_is_max_num	filter_a_where_b_is_max_num(#1, #2) ⇒ PQR **State**: #1: [ABC, PQR, MNZ] #2: [3, 2564.2, 90.1]
filter_a_where_b_is_min_num	filter_a_where_b_is_min_num(#1, #2) ⇒ ABC **State**: #1: [ABC, PQR, MNZ] #2: [3, 2564.2, 90.1]
filter_a_where_b_is_given_value	filter_a_where_b_is_given_value(#1, #2, MNO) ⇒ ABC **State**: #1: [ABC, PQR, MNZ] #2: [MNO, XER, OIY]

Continued on next page

Table 6 – *Continued from previous page*

Primitive	**Example**
filter_a_where_b_is_compared_to	filter_a_where_b_is_compared_to(#1, #2, 80, >) ⇒ [PQR, MNZ] **State**: #1: [ABC, PQR, MNZ] #2: [3, 2564.2, 90.1]
filter_a_where_b_is_in_range	filter_a_where_b_is_in_range_num(#1, #2, 80, 100) ⇒ [MNZ] **State**: #1: [ABC, PQR, MNZ] #2: [3, 2564.2, 90.1]
filter_a_where_b_is_compared_to_date	filter_a_where_b_is_compared_to_date(#1, #2, 25 Feb 2012, >) ⇒ [PQR, MNZ] **State**: #1: [ABC, PQR, MNZ] #2: [25 Jan 2012, 18 March 2012, 13 Oct 2019]
filter_a_where_b_is_in_range_date	filter_a_where_b_is_in_range_date(#1, #2, 25 Feb 2012, 1 Nov 2021, 100) ⇒ [PQR, MNZ] **State**: #1: [ABC, PQR, MNZ] #2: [25 Jan 2012, 18 March 2012, 13 Oct 2019]
filter_a_where_b_is_max_date	filter_a_where_b_is_max_date(#1, #2) ⇒ MNZ **State**: #1: [ABC, PQR, MNZ] #2: [25 Jan 2012, 18 March 2012, 13 Oct 2019]
filter_a_where_b_is_min_date	filter_a_where_b_is_min_date(#1, #2) ⇒ ABC **State**: #1: [ABC, PQR, MNZ] #2: [25 Jan 2012, 18 March 2012, 13 Oct 2019]
grouped_count	grouped_count(#1, #2) ⇒ ABC: 2, XYI: 2, PQR: 1 **State**: #1: [ABC, XYI, ABC, PQR, XYI] #2: [UIQ, QWA, OUE, UHI, RVC]
grouped_sum	grouped_sum(#1, #2) ⇒ ABC: 4, XYI: 7, PQR: 4 **State**: #1: [ABC, XYI, ABC, PQR, XYI] #2: [1, 2, 3, 4, 5]
grouped_mean	grouped_mean(#1, #2) ⇒ ABC: 2, XYI: 3.5, PQR: 4 **State**: #1: [ABC, XYI, ABC, PQR, XYI] #2: [1, 2, 3, 4, 5]
union	union(#1, #2, #3) ⇒ [ABC, PQR, MNO, JHI, KMR] **State**: #1: [ABC, PQR] #2: [MNO] #3: [JHI, KMR]

Continued on next page

Table 6 – *Continued from previous page*

Primitive	**Example**
intersection	intersection(#1, #2) ⇒ [PQR] **State**: #1: [ABC, PQR, MNO] #2: [PQR]
arg_intersection	arg_intersection(#1, #2, #3) ⇒ [WEC] **State**: #1: [XYI, ORE, WEC] #2: [ABC, PQR, MNO] #3: [null, null, MNO]
list_subtraction	list_subtraction(#1, #2) ⇒ [XYI, WEC] **State**: #1: [XYI, ORE, WEC] ; #2: [ORE]
logical_and	logical_and(#1, #2) ⇒ False **State**: #1: False ; #2: True
logical_or	logical_or(#1, #2) ⇒ True **State**: #1: False ; #2: True
select	select("touchdowns by Edwards") ⇒ [ABC, DXE, FGH] **Facts in context**: touchdowns by Edwards ⇒ ABC touchdowns by Edwards ⇒ DXE touchdowns by Edwards ⇒ FGH
filter	filter("#1 from 1st quarter") ⇒ [ABC, DXE] **State**: #1: [ABC, DXE] **Facts in context**: what is from 1st quarter? ⇒ ABC what is from 1st quarter? ⇒ DXE what is from 1st quarter? ⇒ MNF what is from 1st quarter? ⇒ IOU
project	project("when #1 died") ⇒ [March 22, 1958] **State**: #1: [MNS] **Facts in context**: when PYS died ⇒ March 22, 1958 when MNS died ⇒ March 22, 1958 when QFY died ⇒ August 16, 1533
boolean	boolean("if Aikmen started the game at quarterback for the cowboys") ⇒ True **Facts in context**: Aikmen started the game at quarterback for the cowboys

Table 7: Examples QA instances for individual primitives (python functions)

Primitive	Example
compare_numbers	**Quesion:** Is 984,486.24 greater than 594147.75? **Context:** **Answer :** ['yes']
compare_dates	**Quesion:** Is 1934-9-4 greater than 27 May 1899? **Context:** **Answer :** ['yes']
maximum_date	**Quesion:** Which of the following dates come later? **Context:** 11/30/1690 , 1690-05-17 **Answer :** ['November 30, 1690']
minimum_date	**Quesion:** Which of the following dates come before the other? **Context:** 1925-4-12 , 18 Apr 1696 **Answer :** ['April 18, 1696']
date_subtraction	**Quesion:** How many days passed between 1567-6-29 and May 28, 1567? **Context:** **Answer :** ['32']
arg_maximum_date	**Quesion:** Which event has highest date: OUM or NKE? **Context:** Event OUM has date 1977-3-13. Event NKE has date November, 5 2011. **Answer :** ['NKE']
arg_minimum_date	**Quesion:** Which event happened earliest: KSX or KBO or JJT? **Context:** Event KSX has date 11/9/1705. Event KBO has date 04 Jul, 1786. Event JJT has date 04/11/1729. **Answer :** ['KSX']
count	**Quesion:** How many total entities the following list has? **Context:** DMX NQX LFD RJN AMG **Answer :** ['5']
addition	**Quesion:** Given the list of numbers, give their total sum. **Context:** 977.98 ; 710 ; seven ; 4.72 **Answer :** ['1699.7']
subtraction	**Quesion:** What is 721,251 - 32561? **Context:** **Answer :** ['688690']
multiplication	**Quesion:** If you multiply forty-eight with 41, what do you get? **Context:** **Answer :** ['1968']
division	**Quesion:** What is 47 divided by 6 in nearest integer? **Context:** **Answer :** ['7']
mean	**Quesion:** What is the average of the following numbers in nearest integer? **Context:** 172 ; 691 **Answer :** ['431']
maximum_number	**Quesion:** Given the following list, what is the largest number? **Context:** 6603 ; 3.76 ; 636,337.65 ; 91.72 **Answer :** ['636337.65']
minimum_number	**Quesion:** What is the smallest of the following numbers? **Context:** 60,810.74 ; 2.24 ; 48.8 **Answer :** ['2.24']

Continued on next page

Table 7 – *Continued from previous page*

Primitive	**Example**
arg_maximum_number	**Quesion:** Which entity has biggest value: ROJ or ZZH or KFI? **Context:** Entity ROJ has value 91,889. Entity ZZH has value 0.93. Entity KFI has value 9,223.7. **Answer :** ['ROJ']
arg_minimum_number	**Quesion:** Which entity has lowest value: TXM or KPG or JLD? **Context:** Entity TXM has value 195.35. Entity KPG has value 861878. Entity JLD has value 41. **Answer :** ['JLD']
kth_highest	**Quesion:** Give the 2nd maximum value of #17? **Context:** #17 has values 20787.56, 8265.18. #9 has values January 25, 1787, January 27, 1787, January 08, 1787, January 18, 1787. #3 has values February 14, 1994. #18 has values 3.47, 4692.13, 735.31. **Answer :** ['8265.18']
kth_lowest	**Quesion:** Which is the 3rd lowest value of #1? **Context:** #7 has values July 24, 1506, July 04, 1506, July 02, 1506, July 15, 1506. #1 has values 2, 9, 23866. #11 has values KFI, DXK, TFM. **Answer :** ['23866']
are_items_same	**Quesion:** Are the following entities the same? **Context:** Jan 07, 1696 and 01-7-1696. **Answer :** ['yes']
are_items_different	**Quesion:** Are the following entities different? **Context:** HUU and 09-29-1771. **Answer :** ['yes']
filter_a_where_b_is_max_num	**Quesion:** What entity has biggest value? **Context:** Entity OGQ has value 59. Entity HDU has value 94. Entity KLM has value 28,742. Entity LGV has value 713. Entity KGH has value 701. Entity DXK has value 373. **Answer :** ['KLM']
filter_a_where_b_is_min_num	**Quesion:** Which entity has the minimum value? **Context:** Entity FYO has value 266. Entity XHY has value 199052. Entity EQO has value 534. **Answer :** ['FYO']
filter_a_where_b_is_given_value	**Quesion:** Which entities with value equal to 6.45? **Context:** Entity KSX has value 6.45. Entity NLV has value 887.41. Entity OJP has value 603145.31. **Answer :** ['KSX']
filter_a_where_b_is_compared_to	**Quesion:** Entities that have value larger than 948768.92? **Context:** Entity AFE has value 871781. Entity RQX has value 989,517.24. **Answer :** ['RQX']
filter_a_where_b_is_compared_to_date	**Quesion:** List the entities with date below Jul 20 1646? **Context:** Entity ZBK has date 9-12-1560. Entity AGU has date July 17 1953. **Answer :** ['ZBK']
filter_a_where_b_is_max_date	**Quesion:** Which entity has latest date? **Context:** Entity SML has value 11-28-1882. Entity PYS has value Nov 19 1882. **Answer :** ['SML']
filter_a_where_b_is_min_date	**Quesion:** What entity has least recent date? **Context:** Entity SDA has value 5 March, 1523. Entity HXJ has value 14 March 1523. Entity RZO has value 1-26-1523. Entity ZMH has value 23 Jul, 1523. **Answer :** ['RZO']

Continued on next page

Table 7 – *Continued from previous page*

Primitive	**Example**
grouped_count	**Quesion:** How many times do each of EBC, HNQ occur in #14? **Context:** #14 has HNQ, EBC, HNQ. #3 has OZB, LNW, LYP, AGU, HVP, SDA. #17 has ULN, ZZH, RZO **Answer :** ['1', '2']
grouped_sum	**Quesion:** What are the addition of values for each of QWU, JLD? **Context:** QWU has value 179541.17. JLD has value 6,641.78. JLD has value 3.15. QWU has value 6,053.93. QWU has value 44,251.33. JLD has value 411.83. **Answer :** ['229846.43', '7056.76']
grouped_mean	**Quesion:** For each of TKR, NLV, what are the mean of values in integers? **Context:** TKR has value 929. TKR has value 737. TKR has value ninety-five. NLV has value 928. **Answer :** ['587', '928']
union	**Quesion:** Give answer union of #20, #12, #13? **Context:** #20 has answer 29.77. #12 has answer KBE. #11 has answer June 10, 1701. #13 has answer January 23, 1503. **Answer :** ['29.77', 'KBE', 'January 23, 1503']
intersection	**Quesion:** List the entities that occur in both #10 and #7? **Context:** #1 has entities ICU, WAT. #10 has entities WAT, ICU. #7 has entities WAT, ICU. **Answer :** ['ICU', 'WAT']
arg_intersection	**Quesion:** List the entities contain values common in both #9 and #20? **Context:** Entity KBE has value UJI for #20. Entity KLM has value ARU for #20. Entity KBE has no value for #9. Entity KLM has value ARU for #9. **Answer :** ['KLM']
logical_and	**Quesion:** What is logical AND of the given booleans? **Context:** True False **Answer :** ['no']
logical_or	**Quesion:** What is logical OR of the given booleans? **Context:** False False **Answer :** ['no']